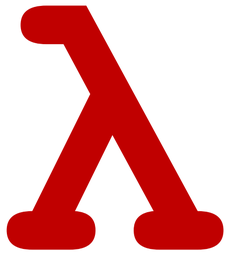
CSCI 275: Programming Abstractions

Lecture 06: Environments & Evaluation

Spring 2025

Stephen Checkoway, **Oberlin College**
Slides gratefully borrowed from Molly Q Feldman



Goals for Today's Class

More helpful language constructs

Gain a more nuanced sense of how we evaluate terms and store information in Racket

Why? This helps your mental model of execution! You can better learn how to solve errors you encounter

Useful Racket

Core Functional Procedure: `filter`

`(filter pred lst)`

`filter` takes a predicate and a list and returns a list as follows:

- For each element `x` in `lst`, run `(pred x)`
- If `(pred x)` returns anything other than `#f`, add `x` to the list to return

Examples

```
(filter positive? '(2 -3 4 5 -1 0)) => '(2 4 5)

(filter (lambda (s) (string-prefix? s "A"))
        '("Ari" "Jane" "Ali")) => '("Ari" "Ali")
```

Let’s write a filter function!

But first, some useful syntactic sugar that will save you some typing

```
(define my-filter
  (lambda (pred lst)
    (cond [… …]
          …
          [else …])))

(define (my-filter pred lst)
  (cond [… …]
        …
        [else …]))
```

Passing a *closure* to filter

```
(define (filter pred lst)
  (cond [(empty? lst) empty]
        [(pred (first lst)) (cons (first lst)
                                  (filter pred (rest lst)))]
        [else (filter pred (rest lst))]))

(define (foo prefix lst)
  (filter (lambda (s) (string-prefix? s prefix)) lst))
```

An implementation of filter, follows the "list recursion" pattern

It's a value, we can pass it around!

How can we use `filter` to return the elements of lst that are either negative numbers or even numbers?

A.
```
(filter (or negative? even?) lst)
```

B.
```
(filter (λ (x) (or (negative? x) (even? x))) lst)
```

C.
```
(filter (or (λ (x) (negative? x))
            (λ (x) (even? x)))
        lst)
```

D. More than one of the above (which?)

Some (hint: useful) Racket built-ins

`member` determines whether an element is in a list or not; returns `#f` if not, the list starting with the element if so

```
> (member '(2 3) '(1 2 3 4))
#f
> (member '(2 3) '(1 (2 3) 4))
'((2 3) 4)
```

Recall that Racket treats anything other than #f as true so
(if (member x lst) then-branch else-branch)
will do what you'd expect

Some (hint: useful) Racket built-ins

`remove` takes an element `e` and removes the first instance of `e` in the provided list; returns the resulting list

```
> (remove 'x '(a b c x z))
'(a b c z)
> (remove 'x '(x a x z))
'(a x z)
> (remove 'x '(1 2 3))
'(1 2 3)
```

Some (hint: useful) Racket built-ins

`max` takes any number of numeric arguments and returns the largest

```
> (max 4 5)
5
> (max -1 0 -3)
0
```

Extending Procedures

Multiple closures

The result of `(lambda (x y z) …)` is a **closure** and closures are values

Hence `(define fun (lambda (x y z) …))` defines `fun` to be the closure and we can call `(fun 1 2 3)`

But we can also return closures from procedures

```
(define f
  (lambda(x)
    (lambda (y)
      (+ x y))))

(define (f x)
  (lambda (y)
    (+ x y)))
```

```
(define g
  (lambda (x)
    (lambda (y)
      (- x y))))
```

What is `(g 3 4)`?

A. 3
B. 4
C. -1
D. 1
E. An error

Evaluating Racket Terms

Expression evaluation

Scheme evaluates s-expressions to produce values

The value of `'()` is `'()`

The value of a variable is the value bound to it
e.g., the variable `null` is bound to `'()`

The value of an atom is the atom itself

The value of a non-null list depends on the head of the list.
Special form? Special evaluation.
Something else? Procedure application.

We've seen this already with `define` (special form) and `list` (built-in procedure)

Procedure evaluation

`(foo 1 2 #t)` applies the procedure bound to the variable `foo` to the arguments `1`, `2`, and `#t`

`(+ 1 2 3)` applies `+` to `1`, `2`, and `3`, performing addition

`(* 5 (- x y) (/ z 8))` computes 5(*x* - *y*)(*z* / 8)

`(list 32 5 8)` creates the list `'(32 5 8)`

`(list-ref (list 32 5 8) 2)` returns the element of `'(32 5 8)` at index `2` namely `8`

Note that `(1 2 3)` is invalid because `1` isn't a special form nor is it a procedure

Procedure evaluation order

`(s-exp0 s-exp2 ... s-expn)`

Racket evaluates each of the s-expressions **in turn**

- `s-exp0` must evaluate to a procedure value
- `s-exp1` through `s-expn` are evaluated to produce values
- Only then, the procedure is applied to the *n* arguments

`(+ (* 2 3) 8)`

- `+` evaluates to the addition procedure
- `(* 2 3)` is evaluated
  - `*` evaluates to the multiplication procedure
  - `2` and `3` evaluate to themselves
  - multiplication procedure is applied to `2` and `3`, producing `6`
- `8` evaluates to itself
- addition procedure is applied to 6 and 8, producing 14

Next Up

HW0 is due **TODAY** at 11:59pm – make sure to check your *Github account online* to make sure all the code pushed

HW1 is live – first commit Monday