Programming Abstractions

Week 2: Environments and Closures

Stephen Checkoway

Using variables

Recall that when Racket evaluates a variable, the result is the value that the variable is bound to

- If we have `(define x 10)`, then evaluating `x` gives us the value `10`
- If we have `(define (foo x) (- x y))`, then evaluating `foo` gives us the procedure `(λ (x) (- x y))` along with a way to get the value of `y`

Racket needs a way to look up values that correspond to variables: an environment

Environments

Environments are mappings from identifiers to values

There's a top-level environment containing many default mappings

- list ↦ #<procedure:list>

  (↦ is read as "maps to", #<procedure:xxx> is how DrRacket displays procedures)
- \+ ↦ #<procedure:+>

Each file in Racket (technically, a module) has an environment that extends the top-level environment that contains all of the defines in the file

Basic operations on environments

Lookup an identifier in an environment

Bind an identifier to a value in an environment

Extend an environment

- This creates a new environment with mappings from identifiers to values as well as a reference to the environment being extended
- The extended and original environment may both contain mappings for the same identifier

Modify the binding of an identifier in an environment (we will avoid doing this in this course)

Looking up an identifier in an environment

If an identifier has been bound in the current environment, its value is returned

Otherwise, if the current environment extends another environment, the identifier is (recursively) looked up in the other environment.

Otherwise, there's no binding for the identifier and an error is reported

Adding a new mapping to an environment

`(define identifier s-exp)`

`define` will add `identifier` to the current environment and bind the value that results from evaluating `s-exp` to it

In any environment, an identifier may only be defined once

- except in the interpreter which lets you redefine identifiers

Adding a new mapping to an environment

`(define (identifier params) body)`

Recall that `(define (foo x y) body)` is the same as
`(define foo (λ (x y) body))`
in that it binds the value of the λ-expression, namely a closure, to `foo`

A closure keeps a reference to the current environment in which the λ-expression was evaluated

Extending an environment

Calling a closure

Calling a closure extends the environment of the closure with the values of the arguments bound to the procedure's parameters

```
(define (sum lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (sum (rest lst)))]))

(define (average lst)
  (/ (sum lst) (length lst)))
```

Calling `(average '(1 2 3))` extends the environment of average (namely the module's environment which contains mappings for `sum` and `average`) with the mapping `lst ↦ '(1 2 3)` and runs `average` with that environment

Example bindings

Shadowing a binding

```
(define (sum lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (sum (rest lst)))]))

(define (foo sum x y)
  (average (list sum x y)))

(define (average lst)
  (/ (sum lst) (length lst)))
```

Inside the body of `foo`, `sum` refers to the parameter
Inside the body of `average`, `sum` refers to the procedure

Example bindings

Shadowing a binding

```
(define (sum lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (sum (rest lst)))]))

(define (foo sum x y)
  (average (list sum x y)))

(define (average lst)
  (/ (sum lst) (length lst)))
```

Inside the body of `foo`, `sum` refers to the parameter
Inside the body of `average`, `sum` refers to the procedure

Example bindings

Shadowing a binding

```
(define (sum lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (sum (rest lst)))]))

(define (foo sum x y)
  (average (list sum x y)))

(define (average lst)
  (/ (sum lst) (length lst)))
```

Inside the body of `foo`, `sum` refers to the parameter
Inside the body of `average`, `sum` refers to the procedure

Example bindings

Shadowing a binding

```
(define (sum lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (sum (rest lst)))]))

(define (foo sum x y)
  (average (list sum x y)))

(define (average lst)
  (/ (sum lst) (length lst)))
```

Inside the body of `foo`, `sum` refers to the parameter
Inside the body of `average`, `sum` refers to the procedure

Extending an environment

`(let ([id1 s-exp1] [id2 s-exp2]…) body)`

let enables us to create some new bindings that are visible only inside `body`

```
(let ([x 37]
      [y (foo 42)] ; binds the result of (foo 42) to y
  (if (< x y)
      (bar x)
      (bar y)))
```

`x` and `y` are only bound inside the body of the let expression

That is, the *scope* of the identifiers bound by `let` is `body`

Modifying a binding

(set! identifier s-exp)

`set!` (read "set bang") can modify an existing binding in an environment

```
(define (bar)
  (define x 10) ; We can use define inside procedures
  (writeln x) ; Output the value of x
  (set! x 25)
  (writeln x))
```

This outputs `10` on one line and then `25` on another

This type of side-effect makes reasoning about code much harder

Except for one time later in the semester, we're not going to be using `set!`

- (We won't actually *need* `set!`, it just makes things easier)

Variations on let

A common problem

When writing programs, it's not uncommon to define some local variables in terms of other local variables

Example: Return the elements of a list of numbers that are at least as large as the first element (the head) of the list, in reverse order

```
(define (at-least-as-large lst)
  (cond [(empty? lst) empty]
        [else
          (let ([head (first lst)]
                [bigger (filter (λ (x) (>= x head)) lst)])
            (reverse bigger))]))
```

This doesn't work; we can't use `head` in the definition of `bigger`

The issue

The issue is the scope of the binding for `head`: just the body of the `let`

One (bad) work around would be to use multiple `lets`

```
(define (at-least-as-large lst)
  (cond [(empty? lst) empty]
        [else
          (let ([head (first lst)])
            (let ([bigger (filter (λ (x) (>= x head)) lst)])
              (reverse bigger)))]))
```

Sequential let

`(let* ([id1 s-exp1] [id2 s-exp2]…) body)`

Later s-exps can use earlier ids, e.g.,

```
(let* ([x 5]
       [y (foo x)]
       [z (+ x y)])
  (bar z y))
```

Another problem: recursion

Often, we're going to want to define a recursive procedure but we can't do that with let or let*

```
(let ([fact (λ (n)
              (if (<= n 1)
                  n
                  (* n (fact (- n 1)))))])
  (fact 5))
```

We can't use `fact` in the definition of `fact`

Recursive let

`(letrec ([id1 s-exp1] [id2 s-exp2]…) body)`

All of the s-exps can refer to all of the ids

- This is used to make recursive procedures

```
(letrec ([fact (λ (n)
                 (if (<= n 1)
                     n
                     (* n (fact (- n 1)))))])
  (fact 5))
```

Recursive let drawback

The values of the identifiers we're binding can't be used in the bindings

Invalid (the value of `x` is used to define `y`)

- ```
  (letrec ([x 1]
           [y (+ x 1)])
    y)
  ```

Valid (the value of `x` isn't used to *define* `y`, only when `y` is called)

- ```
  (letrec ([x 1]
           [y (λ () (+ x 1))])
    (y))
  ```

We can use define inside procedures

```
(define (sum-of-squares lst)
  (define (sq x) (* x x))
  (cond [(empty? lst) 0]
        [else (+ (sq (first lst))
                 (sum-of-squares (rest lst)))]))
```

Avoiding defining sq each time

See also: premature optimization

```
(define sum-of-squares2
  (let ([sq (λ (x) (* x x))])
    (λ (lst)
      (cond [(empty? lst) 0]
            [else (+ (sq (first lst))
                     (sum-of-squares2 (rest lst)))]))))
```

The environment of `sum-of-squares2` contains `sq` whereas the environment for `sum-of-squares` is the module-level environment and `sq` is defined each time

Is this worth doing? Probably not. It's much harder to read

Accumulator-passing style

Loops and efficiency

Compare a C (or Java) function to compute the factorial

```
int fact(int n) {
  int product = 1;
  while (n > 0) {
    product *= n;
    n -= 1;
  }
  return product;
}
```

to our recursive Racket implementation

```
(define (fact n)
  (if (<= n 1)
      1
      (* n
         (fact (- n 1)))))
```

How do these differ?

Loops and efficiency

To be efficient, Racket internally converts all **tail-recursions** into loops

A function is tail-recursive if the last thing to does is to recurse and return the result of that recursion

Example:

```
(define (foo x y)
  (if (some-condition? x y)
    some-value
    (foo (f x y) (g x y))))
```

When the condition is satisfied, some-value is returned, otherwise foo is called again with some different parameters and that value is returned

Our factorial is *not* tail recursive

```
(define (fact n)
  (if (<= n 1)
      1
      (* n
         (fact (- n 1)))))
```

The last thing fact does is perform a multiplication; the recursion happens before the multiplication

Our factorial is *not* tail recursive

Given (fact 4), we end up with

```
(fact 4) => (* 4 (fact 3))
         => (* 4 (* 3 (fact 2)))
         => (* 4 (* 3 (* 2 (fact 1))))
         => (* 4 (* 3 (* 2 1)))
         => (* 4 (* 3 2))
         => (* 4 6)
         => 24
```

We can see this in DrRacket

Solution: Use an accumulator

(Accumulator-passing style isn't the real name of this technique)

```
(define (fact2 n)
  (define (fact-a n acc)
    (if (<= n 1)
        acc ; return the accumulator
        (fact-a (sub1 n) (* n acc))))
  (fact-a n 1))
```

Three things to notice

- We defined a recursive helper function that takes an additional param
- We provide an appropriate initial value for the accumulator in `fact2`'s call to `fact-a`
- `fact-a` is tail-recursive

fact2 is tail-recursive

```
(fact2 4) => (fact-a 4 1)
          => (fact-a 3 2)
          => (fact-a 2 6)
          => (fact-a 1 12)
          => 12
```

So how does this become a loop?

Use variables for the parameters and update them each time through the loop

```
(define (fact-a n acc)
  (if (<= n 1)
      acc ; return the accumulator
      (fact-a (sub1 n) (* n acc))))
```

becomes (pseudocode)

```
def fact-a(n, acc):
  loop:
    if n <= 1:
      return acc
    n, acc = n - 1, n * acc
```

Two strategies for tail recursive procedures

Accumulator-passing style with one or more accumulator parameters

- Usually, the procedure we really want doesn't have these parameters
- Use helper functions

Continuation-passing style

- This uses something called *continuations* which we'll talk about later in the semester

Let's write some tail-recursion procedures

`(sum lst)` — Add all the numbers in the lst

`(maximum lst)` — Find the maximum value in a nonempty list

`(reverse lst)` — Reverses the list lst

`(remove* x lst)` — Remove all instances of x from lst

`(remove x lst)` — Remove the first instance of x from lst

Map and apply

Map: the simple case

`(map proc lst)`

`map` applies the procedure `proc` to every element in list `lst`

```
(map f '(1 2 3 4)) => (list (f 1) (f 2) (f 3) (f 4))

(map sub1 '(10 15 20)) => '(9 14 19)

(map (λ (x) (list x x)) '(a b c)) => '((a a) (b b) (c c))

(map first '((a 5) (b 6) (c 7))) => '(a b c)

(map rest '((a 5) (b 6) (c 7))) => '((5) (6) (7))

(map (λ (x) (cons (first x) x)) '((1 2) (3 4))) =>
  '((1 1 2) (3 3 4))
```

How would we implement map?

Non-tail-recursive

- Simple, clear

Tail-recursive

- Use an accumulator to hold the reversed results, then reverse

General map

`(map proc lst1 lst2 … lstn)`

If `proc` is a procedure of n arguments, then `map` will apply proc to corresponding elements n lists (which all have the same length)

```
(map f '(a b c) '(1 2 3)) => (list (f a 1) (f b 2) (f c 3))

(map cons '(a b c) '(x y z)) => '((a . x) (b . y) (c . z))

(map list '(a b) '(c d) '(e f)) => '((a c e) (b d f))

(map * '(0 1 2) '(3 4 5) '(6 7 8)) => '(0 28 80)
```

How would we implement the general map?

Two issues

- How do we write a procedure that takes a variable number of arguments?
- How do we apply a procedure to a variable number of arguments?

Variable argument procedure

`(define foo (λ params body))`

When params is a list of identifiers, the identifiers are bound to the values of the procedure's arguments

When params is an identifier (i.e., not a list), then the identifier is bound to a list of the procedure's arguments

```
(define count-args
  (λ params
    (length params)))
```

```
(define list
  (λ elements elements))
```

Required parameters + variable parameters

`(define foo (λ (x y z . params)) body)`

Separate the required parameters from the list of variable parameters with a period

```
(define drop-2
  (λ (x y . lst) lst))
```

(drop-2 1 2 3 4)

- x is bound to 1
- y is bound to 2
- lst is bound to '(3 4)

Aside: The period syntax make some sense

Recall that `'(x . y)` is a pair (i.e., `(cons 'x 'y)`)

A list is either empty or it's a pair (x . lst) where lst is a list

The list '(x y z) is the shorthand notation for '(x . (y z))

(y z) is shorthand for '(y . (z)) and (z) is shorthand for '(z . ())

Lots of equivalent ways to write '(x y z)

- '(x . (y z))
- '(x y . (z))
- '(x y z . ())
- '(x . (y . (z . ())))
- '(x y . (z . ()))

Variable argument procedure with define

`(define (foo . params) body)`

```
(define (count-args . args)
  (length args))
```

With some required parameters

```
(define (drop-2 x y . others)
  others)
```

Applying a procedure to a list of arguments

(apply proc lst)

Applies proc to the arguments in lst

```
(apply max '(1 3 4 2)) => (max 1 3 4 2) => 4

(define (sum lst)
  (apply + lst))
(sum '(1 2 3)) => (apply + '(1 2 3)) => (+ 1 2 3) => 6
```

Applying with some fixed arguments

`(apply proc v… lst)`

`apply` takes a variable number of arguments where the final one is a list and applies `proc` to all of those arguments

```
(apply proc 1 2 3 '(4 5 6)) => (proc 1 2 3 4 5 6)
```

Simplifying sum and maximum

Let's rewrite `sum` and `maximum` using `apply` with + and `max`

Distance of a 3-d point from the origin

Recall that a point (x, y) lies $\sqrt{x^2 + y^2}$ from the origin

Let's make a procedure to compute this

```
(define (distance-from-origin x y)
  (sqrt (+ (* x x) (* y y))))

(distance-from-origin 3 4) => 5
```

Distance of a 3-d point from the origin

```
(define (distance-from-origin x y)
  (sqrt (+ (* x x) (* y y))))
```

If we have a point

```
(define p '(5 -8))
```

how we can't use `(distance-from-origin p)`

We can use apply

```
(apply distance-from-origin p)
```

Using map and apply together

Let's sum up all numbers in a structured (i.e., non-flat) list

```
(define (sum-all lst)
  (cond [(number? lst) lst]
        [(list? lst) (apply + (map sum-all lst))]
        [else
         (error 'sum-all
                "~v isn't a number or list"
                lst)]))
```

```
(sum-all '(1 2 (3 4 (5) () 6) 8)) => 29
(sum-all '(1 2 (x))) => error
```

General map implementation

Give this a try on your own!

Hints

- Start with `(define (map proc . lsts) …)`
- Define a helper function (map1 proc lst) that applies a single-argument `proc` to the elements of `lst`
- Define a helper function `(define (map-a lsts acc) …)`
- Use `map1` to get the heads and tails of elements in `lsts`
- Use `apply` to apply `proc` to the heads and `cons` the result onto `acc`

```
(define (map proc . lsts)
  (define (map1 proc lst) …)
  (define (map-a lsts acc)
    … (apply proc heads) …)
  (map-a lsts empty))
```