Programming Abstractions

Week 12-1: Implementation details and macros

Stephen Checkoway

Dynamic binding

Lexical vs. dynamic binding

```
(let ([y 3])
  (let ([f (lambda (x) (+ x y))]
        [y 18])
    (f 2)))
```

Lexical binding: 5

Dynamic binding: 20

Dynamic binding in MiniScheme

We need only make minimal changes to interp.rkt

We don't need to store the current environment when we construct a closure

When we apply a procedure to a list of arguments, we need to extend the current environment

Changes to `apply-proc`

```
-(define (apply-proc proc args)
+(define (apply-proc proc args e)
   (cond [(prim-proc? proc)
          (apply-primitive-op (prim-proc-op proc) args)]
         [(closure? proc)
          (let ([params (closure-params proc)]
-               [body (closure-body proc)]
-               [c-env (closure-env proc)])
+               [body (closure-body proc)])
            (if (= (length params) (length args))
-               (eval-exp body (env params (map box args) c-env))
+               (eval-exp body (env params (map box args) e))
                (error 'apply-proc "incorrect number of parameters")))]
         [else (error 'apply-proc "bad procedure: ~s" proc)]))
```

Changes to `eval-exp`

```diff
           [(app-exp? exp-tree)
            (apply-proc
             (eval-exp (app-exp-proc exp-tree) e)
-            (map (λ (exp) (eval-exp exp e)) (app-exp-args exp-tree)))]
+            (map (λ (exp) (eval-exp exp e)) (app-exp-args exp-tree))
+            e)]
```

That's it!

```
MS> (let ([y 3])
      (let ([f (lambda (x) (+ x y))]
            [y 18])
        (f 2)))
20
```

Pass by reference

Pass by value vs. pass by reference

```
(let ([x 0]
      [f (lambda (y) (set! y 34))])
  (begin
   (f x)
   x))
```

Pass by value: 0

Pass by reference: 34

Pass by reference in MiniScheme

Need to make just a few changes

First, the env constructor must not box the values

- If you followed Bob's advice and made env box its values
  `(list 'env syms (map box vals) previous-env)`
  you need to change that to
  `(list 'env syms vals previous-env)`
- fix up *all* of your calls to `env` to box the values
  this includes all of the calls in the tests as I discovered

Pass by reference in MiniScheme

Second, when evaluating arguments for a `app-exp`,

- if the argument is a `var-exp`, it should be looked up, but not evaluated
- if the argument isn't a var-exp, it should be evaluated and boxed

```
          [(app-exp? exp-tree)
-          (apply-proc
-           (eval-exp (app-exp-proc exp-tree) e)
-           (map (λ (exp) (eval-exp exp e)) (app-exp-args exp-tree)))]
+          (let ([args (map (λ (exp)
+                             (if (var-exp? exp)
+                                 (env-lookup e (var-exp-sym exp))
+                                 (box (eval-exp exp e))))
+                           (app-exp-args exp-tree))])
+            (apply-proc
+             (eval-exp (app-exp-proc exp-tree) e)
+             args))]
```

Pass by reference in MiniScheme

Third, all of the arguments passed to apply-proc are boxes, not values

- For primitive procedures, we need to unbox them
- For closures, we need to bind parameters to the existing boxes

```
 (define (apply-proc proc args)
   (cond [(prim-proc? proc)
-         (apply-primitive-op (prim-proc-op proc) args)]
+         (apply-primitive-op (prim-proc-op proc) (map unbox args))]
         [(closure? proc)
          (let ([params (closure-params proc)]
                [body (closure-body proc)]
                [c-env (closure-env proc)])
            (if (= (length params) (length args))
-               (eval-exp body (env params (map box args) c-env))
+               (eval-exp body (env params args c-env))
                (error 'apply-proc "incorrect number of parameters")))]
         [else (error 'apply-proc "bad procedure: ~s" proc)]))
```

That's it!

```
MS> (let ([x 0]
          [f (lambda (y) (set! y 34))])
      (begin
       (f x)
       x))
34
MS> (let ([x 0]
          [f (lambda (y) (set! y 34))])
      (begin
       (f (+ x 0))
       x))
0
```

Pass by name

Pass by value vs name

Pass by name

```
(let* ([v 0]
       [f (λ (x) ; Don't need begin in λ body
            (set! v (+ v 1))
            x)])
  (f (+ v 5)))
```

Pass by name

- The text of `f`'s body becomes the two expressions (by replacing `x` with the text of the argument)

  `(set! v (+ v 1))`

  `(+ v 5)`

- `v` is set to 1 and then 6 is returned

Pass by name in MiniScheme

This is more difficult

First, change calls to apply-proc

- Do not evaluate arguments
- Pass the argument expressions and the environment to `apply-proc`

```
          [(app-exp? exp-tree)
           (apply-proc
            (eval-exp (app-exp-proc exp-tree) e)
-           (map (λ (exp) (eval-exp exp e)) (app-exp-args exp-tree)))]
+           (app-exp-args exp-tree)
+           e)]
```

Pass by name in MiniScheme

Second, change `apply-proc`

- Take the current environment as a parameter, only needed for `prim-procs`
- Evaluate arguments for a `prim-proc`
- Reconstruct a closure's body by substituting argument expressions for parameters

Pass by name in MiniScheme

```
-(define (apply-proc proc args)
+(define (apply-proc proc args e)
   (cond [(prim-proc? proc)
-          (apply-primitive-op (prim-proc-op proc) args)]
+          (apply-primitive-op (prim-proc-op proc)
+                              (map (λ (exp) (eval-exp exp e)) args))]
         [(closure? proc)
          (let ([params (closure-params proc)]
                [body (closure-body proc)]
                [c-env (closure-env proc)])
            (if (= (length params) (length args))
-               (eval-exp body (env params (map box args) c-env))
+               (eval-exp (substitute (map list params args) body)
+                         (env params (map box args) c-env))
                (error 'apply-proc "incorrect number of parameters")))]
         [else (error 'apply-proc "bad procedure: ~s" proc)]))
```

Create an association list

Substitution is tricky

Given

```
(let ([v 0])
  (let ([f (lambda (x)
             (begin
               (set! v (+ v 1))
               x))])
    (f (+ v 5))))
```

the body of `f` needs to be reconstructed as

```
(begin
  (set! v (+ v 1))
  (+ v 5))
```

substitute

```
(define (substitute args exp)
  (cond [(lit-exp? exp) exp] ; lit-exp doesn't change
        [(var-exp? exp) ...]
        [(app-exp? exp) ...]
        [(ite-exp? exp) ...]
        [(let-exp? exp) ...]
        [(lam-exp? exp) ...]
        [(set-exp? exp) ...]
        [(seq-exp? exp) ...]
        [else (error ...)]))
```

Variable expressions

For a variable expression, look up the variable in the list of (param arg-exp) and replace it with the corresponding arg-exp, if the variable is in the list

```
[(var-exp? exp)
 (let ([arg (assoc (var-exp-sym exp) args eq?)])
   (if arg
       (second arg)
       exp))]
```

Application, if-then-else, sequence expressions

For an application, if-then-else, and sequence (begin) expressions, recursively substitute in each of the sub-expressions

```
[(app-exp? exp)
 (app-exp (substitute args (app-exp-proc exp))
          (map (λ (arg-exp) (substitute args arg-exp))
               (app-exp-args exp)))]
[(ite-exp? exp)
 (ite-exp (substitute args (ite-exp-cond exp))
          (substitute args (ite-exp-then exp))
          (substitute args (ite-exp-else exp)))]
[(seq-exp? exp)
 (seq-exp (map (λ (exp) (substitute args exp))
               (seq-exp-exps exp)))]
```

Lambda and let expressions

Tricky! Recursively substitute in let bindings

Recursively substitute in body, except for arguments that are shadowed by the let-binding or lambda parameters

```
[(let-exp? exp)
 (let* ([syms (let-exp-syms exp)]
        [vals (map (λ (v) (substitute args v)) (let-exp-exps exp))]
        [args (filter-not (λ (sym) (assoc sym args eq?)) syms)]
        [body (substitute args (let-exp-body exp))])
   (let-exp syms vals body))]
[(lam-exp? exp)
 (let* ([params (lam-exp-params exp)]
        [args (filter-not (λ (sym) (assoc sym args eq?)) params)]
        [body (substitute args (lam-exp-body exp))])
   (lam-exp params body))]
```

Set expression

If `x` in `(set! x exp)` is a parameter to be replaced with an argument expression then

- if the argument is a variable, replace x with the symbol for the variable
- if the argument is not a variable, it's an error

Recursively substitute in the expression

```
[(set-exp? exp)
 (let* ([old-sym (set-exp-sym exp)]
        [new-sym (assoc old-sym args)]
        [new-exp (substitute args (set-exp-exp exp))])
   (cond [(not new-sym) (set-exp old-sym new-exp)]
         [(var-exp? (second new-sym))
          (set-exp (var-exp-sym (second new-sym)) new-exp)]
         [else (error …)]))]
```

Painful, but that's it

```
MS> (let ([v 0])
      (let ([f (lambda (x)
                 (begin
                   (set! v (+ v 1))
                   x))])
        (f (+ v 5))))
6
```

define-syntax: hygienic macros

Macros in C: text replacement

```
#include <stdio.h>
#define multiply(x, y) x * y
int main() {
  int z = multiply(2, 3);
  printf("%d\n", z);
  return 0;
}
```

Preprocessor performs a textual replacement

```
  int z = 2 * 3;
```

Prints out 6

`multiply(1+2, 3)` will expand to `1+2 * 3` which is 7 rather than 9!

Similar, but better, rewriting in Scheme

```
(define-syntax keyword
  (syntax-rules ()
    [pattern1 transformation1]
    [pattern2 transformation2]
    ...
    [patternn transformationn]))
```

Patterns can specify variables that can be used in the corresponding transformation

What does this code print out?

```
(define (zero! var)
  (set! var 0))

(let ([x 10])
  (displayln x) ; prints out the value of x
  (zero! x)
  (displayln x)) ; prints out the value of x
```

A. 0
   0

B. 10
   0

C. 10
   10

D. This is an error

Our zero! didn't work correctly

What we'd like to do is transform `(zero! var)` *into* `(set! var 0)`

```
(define-syntax zero!
  (syntax-rules ()
    [(_ var) (set! var 0)]))
```

The pattern `(_ var)` means that this rule will match things like `(zero! x)`

- The leading _ means it matches the keyword

The matching transformation `(set! var 0)` means that `(zero! y)` will be replaced with `(set! y 0)`

- Variables in the pattern match parts of the input which can be used in the transformed output

```
(define-syntax zero!
  (syntax-rules ()
    [(_ var) (set! var 0)]))

(let ([x 10])
  (displayln x) ; Prints out 10
  (zero! x)
  (displayln x)) ; Prints out 0
```

Let's extend zero! to zero out multiple vars

We can use ... in a pattern to mean "match zero or more of the previous thing" and we can pair that with ... in the transformation to mean repeat the previous thing once per input item

```
(define-syntax zero!
  (syntax-rules ()
    [(_ var ...)
     (begin
       (set! var 0) ...)]))
```

Now `(zero! foo bar baz)` expands to

```
(begin
  (set! foo 0)
  (set! bar 0)
  (set! baz 0))
```

What does this code print out?

```
(define-syntax zero!
  (syntax-rules ()
    [(_ var ...)
     (begin
       (set! var 0) ...)]))
(let ([x 10]
      [y 20])
  (displayln (format "x=~s y=~s" x y))
  (zero! x y)
  (displayln (format "x=~s y=~s" x y)))
```

A.
```
x=0 y=0
x=0 y=0
```

B.
```
x=10 y=20
x=0 y=20
```

C.
```
x=10 y=20
x=10 y=0
```

D.
```
x=10 y=20
x=0 y=0
```