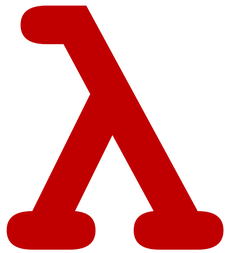
CSCI 275: Programming Abstractions

Lecture 25: MiniScheme G (`set!` & `begin`)

Spring 2025

Stephen Checkoway
Slides from Molly Q Feldman



Notes on MiniScheme

- There is no `'(1 2 3)` list in MiniScheme, only `(list 1 2 3)`

- If you're not using structs, please stop, reimplement, and continue.
  - Make sure they include `#:transparent` for debugging purposes!

`set!` and `begin` expressions

Are you annoyed you can’t change a variable’s value in Racket?

I’ve got good news: you totally can

I’ve also got bad news: in this class, we’re going to use this ability only for implementing one aspect of MiniScheme

Functions in Scheme/Racket that change values have a ! in their name (pronounced “bang”)

Mutation in Racket is done using `set!`

To mutate variables as we would in other languages,
we can use `(set! var value)`

```
(let ([v 10])
     (displayln v)
     (set! v 20)
     (displayln v))
```

produces

```
10
20
```

What is the value of

```
(let ([x 10])
   (+ x
      (let ([x 20])
        x)
      x))
```

This is the sum of 3 numbers

A. 30

B. 40

C. 50

D. 60

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

returns what in Racket?

A. 6

B. 5

C. 0

D. 1

E. Error

Evaluation of `set!`

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

`f` is called with value 5, so `x` is bound to 5

`v` is set to 1

`x` equal to `5` is returned

What is the result of calling `(is-empty '(1 2))`?

```
(define (is-empty lst)
    (if (empty? lst)
        0
        (displayln lst)
        (is-empty (rest lst))))
```

A. Returns: 0

B. Prints: (1 2)
(2)
Returns: 0

C. Prints: (1 2)
(2)

D. Error

E. Something else

`begin` in Racket

A special form to allow multiple things to evaluate, returning only the result of the last one

```
(begin
  (define y 23)
  y)
> 23
```

`lambdas`, `let` and `cond` have "implicit begin" behavior - most useful in combination with `set!` or printing

Side effects

Functions compute and return values `(lambda (x) (+ x 3))`

Everything else they might do is a **side effect**

Examples

- Modifying a global variable `(set! var value)`
- Performing I/O (e.g., `(read)` or `(displayln x)`
- Raising exceptions `(error 'foo "error message")`

Side Effects in functional languages

In functional languages, we tend to want our code to have as few *side effects* as possible

- We do not want to affect the scope *outside* of a function's body

This plays a role in why functional languages do not typically employ:

- Graphics
- Easy print debugging
- Web programming (but Elm!)

What does running the following code output in DrRacket?

```
(+ 1
     (begin
       (println “hello world”)
       2))
```

A. “hello world”
   3

B. “hello world”

C. Error

D. 3

   “hello world”

E. Something else

What is the value of

```
(let ([x 10])
   (+ x
      (begin
         (set! x 20)
         x)
      x))
```

This is the sum of 3 numbers

A. 30

B. 40

C. 50

D. 60

MiniScheme G

MiniScheme G: set! and begin

EXP → number	parse into `lit-exp`
| symbol	parse into `var-exp`
| (if *EXP EXP EXP*)	parse into `ite-exp`
| (let (*LET-BINDINGS*) *EXP*)	parse into `let-exp`
| (lambda (*PARAMS*) *EXP*)	parse into `lambda-exp`
| (set! symbol *EXP*)	parse into `set-exp`
| (begin *EXP**)	parse into `begin-exp`
| (*EXP EXP**)	parse into `app-exp`

LET-BINDINGS → *LET-BINDING**

LET-BINDING → [symbol *EXP*]*

PARAMS → symbol*

Assignments

Assignment expressions are different than the functional parts of MiniScheme

The `set!` expression introduces mutable state into our language

We're going to use a Racket `box` to model this state

Boxes in Scheme

`box` is a data type that holds a mutable value

Constructor: `(box val)`
Recognizer: `(box? obj)`
Getter: `(unbox b)`
Setter: `(set-box! b val)`

Example usage

We can create a `box` holding the value 275 with `(define b (box 275))`

We can get the value in the box with `(unbox b)`

We can change the value in the box with `(set-box! b 572)`

If we use `(unbox b)` afterward, it'll return 572

This models the way variables work in non-functional languages

set-box! vs. set!

Boxes add a layer of indirection

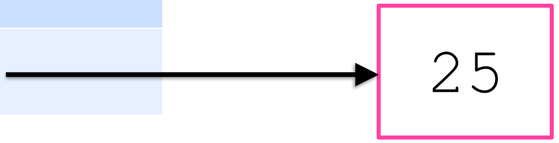
```
(let ([num 25]
      [boxed-num (box 25)])
  …)
```

Changing num via set! modifies the value in the environment

Changing boxed-num via set-box! modifies the value in the box

Environment

Variable	Value
num	25
boxed-num	



```
(define (increment x)
  (set! x (+ x 1)))

(define (increment-box b)
  (set-box! b (+ (unbox b) 1)))

(let ([num 25]
      [boxed-num (box 25)])
  (printf "before num=~v~n" num)
  (increment num)
  (printf "after num=~v~n" num)

  (printf "before boxed-num=~v~n" (unbox boxed-num))
  (increment-box boxed-num)
  (printf "after boxed-num=~v~n" (unbox boxed-num)))
```

Output:

```
before num=25
after num=25
before boxed-num=25
after boxed-num=26
```

Implementing `set!`

To implement set! in MiniScheme

- [Prep Work] Change the values in the environment so that *everything* in the environment is in a box

- [Prep Work] When we evaluate a `var-exp`, we'll lookup the variable in the environment, `unbox` the result, and return it

- [Main Implementation] When we evaluate a set expression such as `(set! x 23)`, we'll lookup `x` in the environment to get its box and then set the value using `set-box!`

We can do this in *four simple steps*

Step 1 for Implementing `set!`

Why? We want to support being able to run `set!` on any sym!

We need to box *every value* in the environment

Find every place you extend the environment and replace each call

```
(env syms vals old-env)
```

with

```
(env syms (map box vals) old-env)
```

Step 2 for Implementing `set!`

Do *not* change your `env-lookup` procedure

Do change the line in eval-exp that evaluates var-exp expressions to

```
[(var-exp? tree)
     (unbox (env-lookup e (var-exp-sym tree)))]
```

At this point, the interpreter should **work exactly as it did before you introduced boxes!**

Step 3 for Implementing `set!`

Set expressions have the form `(set! sym exp)`

You need a new data type for these, I used `set-exp`

When parsing, put the unparsed symbol
(i.e., `'x` rather than `(var-exp 'x)`)
into the `set-exp` and the parsed expression `exp`

Step 4 for Implementing `set!`

Inside eval-exp, you'll need some code

```
[(set-exp? tree)
 (set-box! (env-lookup …)
           (eval-exp …))]
```

What value should `(set! x 10)` *return* in MiniScheme?

A. The original value of x

B. The new value of x (10 in this case)

C. `False`

D. `null`

E. Nothing (which Racket calls `void`)

Running `set!` in Racket

```
Welcome to DrRacket, version 8.5 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (define x 21)
> (define res (set! x 30))
> res
> (void? res)
#t
```