Mutation and
The Environment Model

Edited by 6.001-zombies@mit.edu
Original material by Eric Grimson
The \textit{let} Special Form

\begin{verbatim}
(let ( (<name1> <expr1>)
      (<name2> <expr2>) ...)
   <body>)
\end{verbatim}

\begin{verbatim}
(let ( (z (/ (- x2 x1) num-steps)))
   (square z))
\end{verbatim}
Previously, on 6.037....

- Basics of Scheme
- Substitution Model
- Recursion, plus iterative and recursive processes
- Procedural abstraction
- Abstract data types (cons cells and lists)
- Higher-order procedures
- Symbols and quotation
- Tagged Data
Data Mutation

• Syntax
  • \texttt{set!} for names
  • \texttt{set-car!}, \texttt{set-cdr!} for pairs

• Semantics
  • Simple case: one global environment
  • Complex case: many environments: environment model
**Primitive Data**

```scheme
(define x 10)  creates a new binding for name; special form
```

```
x
returns value bound to name
```

- **To Mutate:**
  ```scheme
  (set! x "foo")  changes the binding for name; special form (value is undefined)
  ```
Assignment -- set!

• Substitution model -- *functional programming*:
  
  (define x 10)
  (+ x 5) ==> 15
  
  ...  
  (+ x 5) ==> 15

  - expression has same value each time it's evaluated (in same scope as binding)

• With mutation:
  
  (define x 10)
  (+ x 5) ==> 15
  
  ...  
  (set! x 94)
  
  ...  
  (+ x 5) ==> 99

  - expression "value" depends on *when* it is evaluated
Syntax: Expression Sequences

• With side-effects, sometimes you want to do some things and then return a value. Use the `begin` special form.

```scheme
(begin
  (set! x 2)
  (set! y 3)
4) ; return value
```

• `lambda`, `let`, and `cond` accept sequences

```scheme
(define frob
  (lambda ()
    (display "frob called") ; do this
    (set! x (+ x 1)) ; then this
    x))
```
Mutating Compound Data

• constructor:
  \((\text{cons } x \ y)\) creates a new pair \(p\)

• selectors:
  \((\text{car } p)\) returns car part of pair \(p\)
  \((\text{cdr } p)\) returns cdr part of pair \(p\)

• mutators:
  \((\text{set-car! } p \ \text{new-x})\) changes car part of pair \(p\)
  \((\text{set-cdr! } p \ \text{new-y})\) changes cdr part of pair \(p\)

; Pair,anytype -> undef ---- side-effect only!
Example 1: Pair/List Mutation

(define a (list 1 2))
(define b a)

a \rightarrow (1 2)
b \rightarrow (1 2)

(set-car! a 10)
b \rightarrow (10 2)

Compare with:

(define a (list 1 2))
(define b (list 1 2))
(set-car! a 10)
b \rightarrow (1 2)
Example 2: Pair/List Mutation

(define x (list 'a 'b))

1. Evaluate (cdr x) to get a pair object
2. Change car part of that pair object

(set-car! (cdr x) (list 1 2))
Sharing, Equivalence, and Identity

- How can we tell if two things are equivalent?
  - Well, what do you mean by "equivalent"?
- The **same object**: test with `eq?`
  \[(eq? \ a \ b) \implies \ #t\]

- Objects that "look the same": test with `equal?`
  \[(equal?\ (list\ 1\ 2)\ (list\ 1\ 2)) \implies \ #t\]
  \[(eq?\ (list\ 1\ 2)\ (list\ 1\ 2)) \implies \ #f\]
Sharing, Equivalence, and Identity

- How can we tell if two things are equivalent?
  Well, what do you mean by "equivalent"?
  - The *same object*: test with `eq?`
    \[(eq? a b) ==> #t\]
  - Objects that *"look" the same*: test with `equal?`
    \[(equal? (list 1 2) (list 1 2)) ==> #t\]
    \[(eq? (list 1 2) (list 1 2)) ==> #f\]

- If we change an object, is it the same object?
  -- Yes, if we retain the same pointer to the object

- How do we tell if part of an object is *shared* with another?
  -- If we mutate one, see if the other also changes

- Notice: No way to tell the difference without mutation!
One last example...

\[
\begin{align*}
x & \mapsto (3 \ 4) \\
y & \mapsto (1 \ 2)
\end{align*}
\]

\(\text{(set-car! } x \ y)\)

\[
\begin{align*}
x & \mapsto ((1 \ 2) \ 4)
\end{align*}
\]

followed by

\(\text{(set-cdr! } y \ (\text{cdr } x))\)

\[
\begin{align*}
x & \mapsto ((1 \ 4) \ 4)
\end{align*}
\]

\(\text{(set-car! } (\text{cdr } x) \ 5)\)

\[
\begin{align*}
x & \mapsto ((1 \ 5) \ 5)
\end{align*}
\]
Functional vs Imperative Programming

- Functional programming
  - No assignments
  - As computing mathematical functions
  - No side effects
  - Easy to understand: use the substitution model!
- Imperative programming
  - A style that relies heavily on assignment
  - Introduces new classes of bugs
- This doesn't mean that assignment is evil
  - It sure does complicate things, but:
  - Being able to modify local state is powerful as we will see
Queue Data Abstraction (Non-Mutating)

• **constructor:**
  
  `(make-queue)`  
  returns an empty queue

• **accessors:**
  
  `(front-queue q)`  
  returns the object at the front of the queue. If queue is empty signals error

• **operations:**
  
  `(insert-queue q elt)`  
  returns a new queue with elt at the rear of the queue

  `(delete-queue q)`  
  returns a new queue with the item at the front of the queue removed

  `(empty-queue? q)`  
  tests if the queue is empty
Queue Contract

Given q is a queue, created by \texttt{(make-queue)} and subsequent queue procedures, where \( i \) is the number of \texttt{inserts}, and \( j \) is the number of \texttt{deletes}

- If \( j > i \) then it is an error
- If \( j = i \) then \texttt{(empty-queue? q)} is true, and \texttt{(front-queue q)} is an error
- If \( j < i \) then \texttt{(empty-queue? q)} is false, and \texttt{(front-queue q)} is the \((j+1)\)th element inserted into the queue
Simple Queue Implementation – pg. 1

• Let the queue simply be a list of queue elements:

\[ \rightarrow \quad \rightarrow \quad \rightarrow \]
\[ b \quad c \quad d \]

• The front of the queue is the first element in the list

• To insert an element at the tail of the queue, we need to “copy” the existing queue onto the front of the new element:

\[ \rightarrow \quad \rightarrow \quad \rightarrow \rightarrow \]
\[ b \quad c \quad d \quad \text{new} \]
Simple Queue Implementation – pg. 2

(define (make-queue) '())

(define (empty-queue? q) (null? q)); Queue<A> -> boolean

(define (front-queue q) ; Queue<A> -> A
 (if (not (empty-queue? q))
  (car q)
  (error "front of empty queue:" q)))

(define (delete-queue q) ; Queue<A> -> Queue<A>
 (if (not (empty-queue? q))
  (cdr q)
  (error "delete of empty queue:" q)))

(define (insert-queue q elt) ; Queue<A>, A -> Queue<A>
 (if (empty-queue? q)
  (cons elt '())
  (cons (car q) (insert-queue (cdr q) elt))))
Simple Queue - Efficiency

• How efficient is the simple queue implementation?
  • For a queue of length $n$
    – Time required – number of iterations?
    – Space required – number of pending operations?

• front-queue, delete-queue:
  • Time: Constant
  • Space: Constant

• insert-queue:
  • Time: Linear
  • Space: Linear
Limitations in our Queue

- Queue does not have identity

```
(define q (make-queue))
q ==> ()

(insert-queue q 'a) ==> (a)
q ==> ()

(set! q (insert-queue q 'b))
q ==> (b)
```
Queue Data Abstraction (Mutating)

- **constructor:**
  
  (make-queue) returns an empty queue

- **accessors:**

  (front-queue q) returns the object at the front of the queue. If queue is empty signals error

- **mutators:**

  (insert-queue! q elt) inserts the elt at the rear of the queue and returns the modified queue

  (delete-queue! q) removes the elt at the front of the queue and returns the modified queue

- **operations:**

  (queue? q) tests if the object is a queue

  (empty-queue? q) tests if the queue is empty
Better Queue Implementation – pg. 1

- We’ll attach a type tag as a defensive measure
- Maintain queue **identity**
- Build a structure to hold:
  - a list of items in the queue
  - a pointer to the front of the queue
  - a pointer to the rear of the queue
Queue Helper Procedures

• Hidden inside the abstraction

(define (front-ptr q) (cadr q))
(define (rear-ptr q) (cddr q))

(define (set-front-ptr! q item)
  (set-car! (cdr q) item))

(define (set-rear-ptr! q item)
  (set-cdr! (cdr q) item))
Better Queue Implementation – pg. 2

(define (make-queue)
  (cons 'queue (cons '() '())))

(define (queue? q) ; anytype -> boolean
  (and (pair? q) (eq? 'queue (car q))))

(define (empty-queue? q) ; Queue<A> -> boolean
  (if (queue? q)
    (null? (front-ptr q))
    (error "object not a queue:" q)))

(define (front-queue q) ; Queue<A> -> A
  (if (not (empty-queue? q))
    (car (front-ptr q))
    (error "front of empty queue:" q)))
(define (insert-queue! q elt); Queue<A>, A -> Queue<A>
  (let ((new-pair (cons elt '())))
    (cond ((empty-queue? q) (set-front-ptr! q new-pair)
      (set-rear-ptr! q new-pair))
      (else (set-cdr! (rear-ptr q) new-pair)
        (set-rear-ptr! q new-pair)))
    q)))

- front-ptr
- rear-ptr

queue: a → b → c → d → e
(define (delete-queue! q) ; Queue<A> -> Queue<A>
  (if (not (empty-queue? q))
    (set-front-ptr! q (cdr (front-ptr q)))
    (error "delete of empty queue:" q))
  q)
Mutating Queue - Efficiency

• How efficient is the mutating queue implementation?
  • For a queue of length $n$
    – Time required -- number of iterations?
    – Space required -- number of pending operations?

• front-queue, delete-queue!:
  • Time: Constant
  • Space: Constant

• insert-queue!:
  • Time: $T(n) = \text{Constant}$
  • Space: $S(n) = \text{Constant}$
Summary - Catch your breath

• Built-in mutators which operate by **side-effect**
  • `set!` (special form)
  • `set-car!` ; Pair, anytype -> undefined
  • `set-cdr!` ; Pair, anytype -> undefined

• Extend our notion of data abstraction to include **mutators**

• Mutation is a powerful idea
  • enables new and efficient data structures
  • can have surprising side effects
  • breaks our model of "functional" programming (substitution model)
Can you figure out why this code works?

```
(define make-counter
  (lambda (n)
    (lambda () (set! n (+ n 1))
      n)))

(define ca (make-counter 0))
(ca) ==> 1
(ca) ==> 2 ; not functional programming!

(define cb (make-counter 0))
(cb) ==> 1
(ca) ==> 3 ; ca and cb are independent
```

Need a new model of mutation for closures.
What the Environment Model is:

• A precise, completely mechanical description:
  • name-rule looking up the value of a variable
  • define-rule creating a new definition of a var
  • set!-rule changing the value of a variable
  • lambda-rule creating a procedure
  • application applying a procedure

• Enables analyzing more complex scheme code:
  • Example: make-counter

• Basis for implementing a scheme interpreter
  • for now: draw EM state with boxes and pointers
  • later on: implement with code
A shift in viewpoint

• As we introduce the environment model, we are going to shift our viewpoint on computation

• Variable:
  • OLD – name for value
  • NEW – place into which one can store things

• Procedure:
  • OLD – functional description
  • NEW – object with inherited context

• Expressions
  • Now only have meaning with respect to an environment
Frame: a table of bindings

- Binding: a pairing of a name and a value

Example: \( x \) is bound to 15 in frame A
\( y \) is bound to \( (1 \ 2) \) in frame A
the value of the variable \( x \) in frame A is 15
Environment: a sequence of frames

- Environment E1 consists of frames A and B
- Environment E2 consists of frame B only
  - A frame may be shared by multiple environments

![Diagram of environments and frames]

- This arrow is called the enclosing environment pointer.
Evaluation in the environment model

• All evaluation occurs \textit{with respect to an environment}
  • The \textit{current environment} changes when the interpreter applies a procedure

• The top environment is called the \textit{global environment (GE)}
  • Only the GE has no enclosing environment

• To \textit{evaluate} a combination
  • Evaluate the subexpressions \textit{in the current environment}
  • Apply the value of the first to the values of the rest
Name-rule

• A name X evaluated in environment E gives the value of X in the first frame of E where X is bound.

\[
\begin{align*}
\text{z} & \mid_{\text{GE}} \Rightarrow z \mid_{\text{E1}} \Rightarrow x \mid_{\text{E1}} \Rightarrow \\
\text{x} & \mid_{\text{GE}} \Rightarrow 3
\end{align*}
\]

• In E1, the binding of x in frame A shadows the binding of x in B.

\[
\begin{array}{c}
\text{z: 10} \\
\text{x: 3}
\end{array}
\]

\[
\begin{array}{c}
\text{x: 15} \\
\text{y:}
\end{array}
\]
Define-rule

- A define special form evaluated in environment $E$ creates or replaces a binding in the first frame of $E$.

$$(\text{define } z \ 20) \ |_{GE} \quad (\text{define } z \ 25) \ |_{E1}$$
Set!-rule

- A set! of variable X evaluated in environment E changes the binding of X in the first frame of E where X is bound.

\[(\text{set! } z \ 20) \mid_{GE} \quad (\text{set! } z \ 25) \mid_{E1}\]
Your turn: evaluate the following in order

\[
(+ z 1) \mid_{E_1} \quad \Rightarrow \quad 11
\]

\[
(set! \ z \ (+ \ z \ 1)) \mid_{E_1} \quad \text{(modify EM)}
\]

\[
(define \ z \ (+ \ z \ 1)) \mid_{E_1} \quad \text{(modify EM)}
\]

\[
(set! \ y \ (+ \ z \ 1)) \mid_{GE} \quad \text{(modify EM)}
\]
Your turn: evaluate the following in order

\[(+ \ z \ 1) \mid_{E_1} \implies 11\]
\[(\text{set!} \ z \ (+ \ z \ 1)) \mid_{E_1} \quad \text{(modify EM)}\]
\[(\text{define} \ z \ (+ \ z \ 1)) \mid_{E_1} \quad \text{(modify EM)}\]
\[(\text{set!} \ y \ (+ \ z \ 1)) \mid_{GE} \quad \text{(modify EM)}\]

Error: unbound variable: y
Double bubble: how to draw a procedure

\((\text{lambda} \ (x) \ (* \ x \ x))\)

A compound proc that squares its argument

Environment pointer

Code pointer

parameters: x
body: \((* \ x \ x)\)
Lambda-rule

• A lambda special form evaluated in environment E creates a procedure whose environment pointer is E

\[
\text{Evaluating a lambda returns a pointer to the procedure object}
\]

\[
\begin{align*}
\text{E1} & \quad \text{environment pointer points to frame A because the lambda was evaluated in E1 and E1} \rightarrow A \\
A & \quad \text{parameters: } x \\
B & \quad \text{body: } (* x x) \\
\end{align*}
\]

\[
\text{(define square (lambda (x) (* x x))) } \mid E_1
\]
To apply a compound procedure $P$ to arguments:

1. Create a new frame $A$
2. Make $A$ into an environment $E$:
   - $A$'s enclosing environment pointer goes to the same frame as the environment pointer of $P$
3. In $A$, bind the parameters of $P$ to the argument values
4. Evaluate the body of $P$ with $E$ as the current environment
(square 4) | \text{GE}

\text{square:}
\begin{align*}
\text{parameters: } & x \\
\text{body: } & (* x x)
\end{align*}

\text{square} \mid \text{GE} \implies \#\langle \text{proc} \rangle

(* x x) \mid \text{E1} \implies 16

* \mid \text{E1} \implies \#[\text{prim}]

x \mid \text{E1} \implies 4
Lessons from this example

• EM doesn't show the complete state of the interpreter
  • missing the stack of pending operations

• The GE contains all standard bindings (*, cons, etc)
  • omitted from EM drawings

• Useful to link environment pointer of each frame to the procedure that created it
Let special form

- A let expression evaluated in environment E evaluates the values for the new variables, and then drops a new frame whose parent frame is E, binding them to the given names.

\[
\text{let } ((x 15) \\
(z (+ x 5)) \\
(* z 2)) \mid_{E1}
\]

- The binding values are evaluated before the new frame is created.
- The body is evaluated in the new environment.
- Sounds familiar....
Let special form

• A let expression evaluated in environment E evaluates the values for the new variables, and then drops a new frame whose parent frame is E, binding them to the given names

\[
\text{(let ((x 15) (z (+ x 5)) (* z 2)) |}_{E1}
\]

• Hidden lambda!

\[
\text{( (lambda (x z) (* z 2)) 15 (+ x 5) )}
\]
Example: make-counter

- Counter: something which counts up from a number

```
(define make-counter
  (lambda (n)
    (lambda () (set! n (+ n 1))
      n
    )))

(define ca (make-counter 0))
(ca) ==> 1
(ca) ==> 2 ; not functional programming
(define cb (make-counter 0))
(cb) ==> 1
(ca) ==> 3
(cb) ==> 2 ; ca and cb are independent
```
\[(\text{define ca (make-counter 0))} \mid_{GE}\] 

\begin{align*}
\text{make-counter:} \\
\text{ca:} \\
p: \ n \\
b: (\lambda () \ (\text{set! } n \ (+ n \ 1)) \ n)
\end{align*}

- \text{environment pointer points to E1 because the lambda was evaluated in E1}
\((ca) \mid_{GE} \Rightarrow 1\)
\[(ca) \mid_{GE} \Rightarrow 2\]

\[
\text{make-counter:}
\]

\[
p: n
\]

\[
b: \text{(lambda ()
(set! n (+ n 1))
n)}
\]

\[
(p: n) \mid_{E1} (n: 0 1 2) \mid_{E3} \Rightarrow \text{empty}
\]

\[
(set! n (+ n 1)) \mid_{E3} n \mid_{E3} \Rightarrow 2
\]
(define cb (make-counter 0))  \[\text{GE}\]

\begin{align*}
\text{make-counter:} & \\
\text{p: n} & \\
\text{b: (lambda () (set! n (+ n 1)) n)} & \\
\text{cb:} & \\
\text{n: 0} & \\
\end{align*}

\begin{align*}
\text{E1} & \\
\text{n: 2} & \\
\text{E3} & \\
\text{E4} & \\
\end{align*}
(cb) \mid_{GE} \Rightarrow 1
Capturing state in local frames & procedures

```
(p: n
 b: ((lambda ()
        (set! n (+ n 1))
        n))

(make-counter:
    ca:
    cb:

E1:
 n: 2

E2:

E3:
 n: 1

E4:
```

```
```
Lessons from the `make-counter` example

- Environment diagrams get complicated very quickly
  - Rules are meant for the computer to follow, not to help humans
- A lambda inside a procedure body captures the frame that was active when the lambda was evaluated
  - this effect can be used to store local state

![Diagram of `make-counter` example]
Recitation Time!