

CS1 Final Exam Review

December 6, 2007

Final Exam Topics

- *Make sure* you understand the topics from the second half of the course:
 - Lists and list processing
 - Message passing and generic types
 - Mutation, list mutation
 - Environment diagrams

Non-topics

- Topics from the first part of term will not be explicitly covered on the exam
(but if you do not understand this material, you will still have trouble with the exam)
 - Substitution model
 - Standard vs. Special Forms
 - Higher order procedures
 - Asymptotic Complexity

Exam Structure

- Three sections
- One section on each major concept
- Your score will be the total of your score on all of the sections
- Each section worth 6.0 points; total = 18.0 points
- Env. diagrams handed in on paper

Evaluating expressions

- Basic rule:
 - evaluate operands
 - evaluate operator
 - apply operator to operands

Environment model

- Still true with environment model, but important change from substitution model:
- Don't **substitute** the value of parameters into expression when we apply operator to operands
- Instead, when they are needed, we **look up** the value of parameters (variables) in the environment

Environment model

- Important: ALL expressions are evaluated in the context of an environment!
 - without an environment, expressions have no meaning
- Looking up a variable's value:
 - search in bottom frame of current env
 - if not found, search parent frame
 - then its parent frame, etc. etc.

Environment Model Rules

- **Rule 1: define**
 - creates a *new* binding in the current environment
- **Rule 2: set!**
 - does *not* create a new binding
 - it searches the environment for an existing binding
 - if it finds one, it changes it to the new value
 - if not, it looks in parent environment etc.
 - if it never finds one, error!

Environment Model Rules

- **Rule 3: lambda**
 - lambda expression creates a pair of
 - code of lambda expression
 - parameters
 - body
 - pointer to environment in which lambda expression is evaluated

Environment Model Rules

- **Rule 4:** applying **lambda** expression to its arguments
 - creates a new frame
 - parent environment of new frame is environment that **lambda** pair points to
 - formal parameters of **lambda** expression are bound to their arguments in frame
 - evaluate body of **lambda** in context of new frame

Environment Model Rules

- **Rule 5:** evaluating **let** expression
 - can evaluate correctly by desugaring to **lambda**, but there's a simpler way:
 - create a new frame
 - parent of new frame is environment in which **let** expression is evaluated in
 - make **let** bindings in new frame
 - evaluate body of let in context of new frame

Environment Model Rules

- **Rule 6:** evaluating **let** expression, part 2
 - when bindings require an expression be evaluated, evaluate that in the context of the *old* environment, not the newly created frame

- **Example:**

```
(define x 5)
(let ((x 10)
      (y (* x x)))
  (- y x))
→ 15
```

Environment Model Rules

- **Rule 6:** example

```
(let ((x 10))  
  (define (fact n)  
    (if (= n 0) 1  
        (* n (fact (- n 1)))))  
  (fact x))  
→ 3628800
```

Environment Model Rules

- **Rule 6:** example

```
(let ((x 10)
```

```
    (fact (lambda (n)
```

```
        (if (= n 0) 1
```

```
            (* n (fact (- n 1))))))
```

```
(fact x))
```

→ *reference to undefined identifier: fact*

- ???

Environment Model Rules

- Problem:

```
(let ((x 10)
      (fact (lambda (n)
              (if (= n 0) 1
                  (* n (fact (- n 1)))))))
    (fact x))
```

- `lambda` expression is bound to `fact` in new frame
- `lambda` expression is evaluated in *old* environment!
- There is no `fact` in old (global) environment!

Environment Model Rules

- Problem:

```
(let ((x 10)
      (fact (lambda (n)
              (if (= n 0) 1
                  (* n (fact (- n 1)))))))
    (fact x))
```

- Makes sense if desugared to `lambda`

```
((lambda (x fact) (fact x))
 10 (lambda (n) (if (= n 0) 1 (* n (fact (- n 1)))))
```

- Easier to just memorize rule

begin and friends

- Remember also that expressions in a **begin** statement **must** be evaluated in left to right order
- Some other expressions have this property too ("implicit **begin**")
 - body of **lambda** expressions
 - body of **let** expressions
 - consequent portion of **cond** statements

Procedures with local state

```
(define (make-accum initial)
  (let ((value initial))
    (lambda (change)
      (set! value (+ value change))
      value))))
```

Procedures with local state

- Compare to:

```
(define (make-accum initial)
  (lambda (change)
    (let ((value initial))
      (set! value (+ value change))
      value))))
```

- Completely different!
- Probably not what you want

Another Env. Diag. example

- Returning two functions

```
(define (make-two-functions)
  (let ((a 0)
        (b 0))
    (let ((f1 (lambda () (begin (set! a (+ a 1))
                                a)))
          (f2 (lambda (x) (begin (set! b (+ b x))
                                b))))
      (cons f1 f2))))
(define two-fns (make-two-functions))
((cdr two-fns) 3)
```

List processing

- In class we talked about algorithms for
 - Adding items to a list
 - Removing items from a list
 - Searching for items in a list
 - Doing the above with and without mutation

Example: DNA

- Making DNA sequences

- DNA seq: list of 'A 'G 'C 'T

```
(define (make-dna-sequence dna-sequence)
  (if (valid-dna-sequence? dna-sequence)
      (cons 'dna-sequence dna-sequence)
      (error "Invalid DNA sequence: "
             dna-sequence)))
```

Example: DNA

```
(define (valid-base? base)
  (if (symbol? base)
      (or (eq? base 'A)
          (eq? base 'T)
          (eq? base 'G)
          (eq? base 'C))
      #f))
```

Example: DNA

```
(define (valid-dna-sequence? seq)
  (cond ((null? seq) #t)
        ((valid-base? (car seq))
         (valid-dna-sequence? (cdr seq)))
        (else #f)))
```

Example: DNA

- Another attempt...

```
(define (valid-dna-sequence? seq)  
  (apply and (map valid-base? seq)))
```

- Why won't this work?
 - Hint: what is `and` ?

Example: DNA

- Mutate nth base pair to random value

```
(define (random-base)
```

```
  (list-ref '(A T C G) (random 4)))
```

```
(define (mutate-nth! n dna-seq)
```

```
  (if (= n 0)
```

```
      (set-car! dna-seq (random-base))
```

```
      (mutate-nth! (- n 1) (cdr dna-seq))))
```

Example: DNA

- Splice two DNA sequences (copying)

```
(define (splice-dna-seq dna-seq-obj-1 dna-seq-obj-2)
  (let ((dna-seq-1 (cdr dna-seq-obj-1))
        (dna-seq-2 (cdr dna-seq-obj-2)))
    (let ((splice-1 (random (length dna-seq-1)))
          (splice-2 (random (length dna-seq-2))))
      (make-dna-sequence
        (append (first-n splice-1 dna-seq-1)
                (last-n splice-2 dna-seq-2))))))
```

Example: DNA

- Copy the first/last n elements of a list

```
(define (first-n n seq) ; easy
```

```
  (if (= n 0)
```

```
      '())
```

```
      (cons (car seq) (first-n (- n 1) (cdr seq))))))
```

```
(define (last-n n seq) ; harder
```

```
  (if (= n (length seq)) ; inefficient! can do better
```

```
      seq
```

```
      (last-n n (cdr seq))))))
```

Example: DNA

- Splice two DNA sequences (mutating)

```
(define (splice-dna-seq! dna-seq-obj-1 dna-seq-obj-2)
  (let ((dna-seq-1 (cdr dna-seq-obj-1))
        (dna-seq-2 (cdr dna-seq-obj-2)))
    (let ((splice-1 (random (length dna-seq-1)))
          (splice-2 (random (length dna-seq-2))))
      (make-dna-sequence
       (set-cdr! (nth-pair splice-1 dna-seq-1)
                 (nth-pair splice-2 dna-seq-2))
       dna-seq-1))))
```

Example: DNA

- Return nth **cons** pair in a list

```
(define (nth-pair n lst)
```

```
  (cond ((= n 0) lst)
```

```
        ((null? lst) (error "bad!"))
```

```
        (else (nth-pair (- n 1) (cdr lst))))))
```

`eq?` and `equal?`: When is the whole not the sum of its parts?

- `eq?` tests whether the two operands are the exact same object
- `equal?` tests whether the two operands look exactly the same
- Two objects that can be `equal` but not `eq` would be:

```
(define a '(1 2 3))
```

```
(define b '(1 2 3))
```

List mutation

- Can change contents of data structures
- We did this with `set-car!` and `set-cdr!`
- `set-car!` points the first part of a `cons` pair to the same thing pointed to by its second argument
- `set-cdr!` does the same to the `cdr` part of the `cons` pair

Note!

- In a box-and-pointer diagram...
- `car` and `cdr` part of `cons` pairs point to Scheme objects
- Can point to a `cons` pair as a *whole*
- *Cannot* point to `car` or `cdr` part of `cons` pair!

Example

```
(define lst '((1 2) (3 4)))  
(set-cdr! (car lst) (cdr lst))
```

Another list mutation example

```
(define (list-swallow! lst)
  (if (or (null? lst) (null? (cdr lst)))
      lst
      (begin (set-cdr! lst (cddr lst))
              (list-swallow! (cdr lst)))))
```

Any idea what this does?

List mutation example

```
(define lst (list 1 2 3))
```

```
(list-swallow! lst)
```

```
lst
```

```
(define lst (list 1 2 3 4))
```

```
(list-swallow! lst)
```

```
lst
```

Message passing

- Create new "types" of objects that respond to messages
- Local state of objects stored in trapped environment frames
 - parameter list of constructor function, or
 - **let** bindings

Message passing

- Constructor function returns a **lambda** expression
 - whose environment is the trapped frame(s) containing local state
- Arguments to **lambda** expression include
 - message being invoked (symbol)
 - (sometimes) arguments of message

Message passing, form 1

```
(define (make-some-object)
  (let ((val1 100) ; local state stored here
        (val2 42))
    (lambda (op . args)
      (cond ((eq? op 'reset-val1!) (set! val1 0))
            ((eq? op 'set-val2!)
             (set! val2 (car args)))
            (else (error "not understood: " op))))))
```

Message passing, form 2

```
(define (make-some-object)
  (let ((val1 100) ; local state stored here
        (val2 42))
    (define (dispatch op . args)
      (cond ((eq? op 'reset-val1!) (set! val1 0))
            ((eq? op 'set-val2!)
             (set! val2 (car args)))
            (else (error "not understood: " op))))
    dispatch))
```

Message passing, form 2

- Advantage of form 2?
 - objects can recursively call dispatch function inside message-handling code
 - objects can send messages to themselves
 - more general than form 1

Message passing

- Design advice for MP objects:
- Define helper functions for all non-trivial message-handling code
- Call helper functions in body of `lambda`
- Keeps code clean, easier to debug and understand

Message passing example

- Interactive tic-tac-toe game

(define game (make-tic-tac-toe-game))

(game 'display)

| |

| |

| |

Message passing example

- Interactive tic-tac-toe game

(game 'move 'x 0 0)

(game 'display)

```
x |   |  
-----  
  |   |  
-----  
  |   |
```

Message passing example

- Interactive tic-tac-toe game

(game 'move 'o 0 1)

(game 'display)

```
x | o |  
-----  
  |   |  
-----  
  |   |
```

Message passing example

- Interactive tic-tac-toe game

(game 'over?') ;; → #f

(game 'won? 'x) ;; → #f

Skeleton

```
(define (make-tic-tac-toe-game)  
  ...)
```

Filling in...

```
(define (make-tic-tac-toe-game)
  (lambda (op . args)
    (cond ((eq? op 'display) ...)
          ((eq? op 'move) ...)
          ((eq? op 'won?) ...)
          ((eq? op 'over?) ...)
          (else (error "not understood: " op))))))
```

State variables

```
(define (make-tic-tac-toe-game)
  (let ((board ???)           ;; contents of board
        (to-move 'x)        ;; x or o
        (status 'ongoing))  ;; or (won x), (won o), draw
    (lambda (op . args)
      ...)))
```

Helper procedures

```
(define (make-tic-tac-toe-game)
  (define (new-board)
    '(e e e e e e e e e)) ;; 9 empty squares
  (let ((board (new-board))
        (to-move 'x)
        (status 'ongoing))
    (lambda (op . args)
      ...)))
```

More filling in...

```
(define (make-tic-tac-toe-game)
  ;; new-board
  (let (...)
    (lambda (op . args)
      (cond ((eq? op 'display) (display-board board))
            ((eq? op 'move)
             (make-move! board
                          (car args) (cadr args) (caddr args)))
            ((eq? op 'won?) (has-won? (car args)))
            ((eq? op 'over?) (over?))
            (else (error "not understood: " op))))))
```

Helper procedures

```
(define (make-tic-tac-toe-game)
  (define (new-board) '(e e e e e e e e e))
  (let (...)
    (define (over?) (or (eq? status 'draw)
                        (equal? status '(won x))
                        (equal? status '(won o))))
    (define (has-won? side) (equal? status (list 'won side)))
    (lambda (op . args) ...)))
```

Board accessors

```
(define (make-tic-tac-toe-game)
  (define (new-board) '(e e e e e e e e e))
  (define (get-square board row col)
    (list-ref board (+ (* 3 row) col)))
  (define (set-square! board row col value)
    ;; exercise... could use (list-tail lst n)
  )
  (let (...
    (define (over?) ...)
    (define (has-won? side) ...)
    (lambda (op . args) ...)))
```

More helper procedures

```
(define (make-tic-tac-toe-game)
  ;; new-board, get-square, set-square!
  (define (display-board board)
    ;; use board accessors to print board
  )
  (let (...
    ;; over?, has-won?
    (lambda (op . args) ...)))
```

More helper procedures

```
(define (make-tic-tac-toe-game)
  ;; new-board, get-square, set-square!, display-board
  (let (...)
    (define (make-move! board row col value)
      ;; -- check that value is valid (x or o)
      ;; -- check that square is empty
      ;; -- make the move (using set-square!)
      ;; -- update status and player to move
    )
    ;; over?, has-won?
    (lambda (op . args) ...)))
```

More helper procedures

```
(define (make-tic-tac-toe-game)
  ;; new-board, get-square, set-square!, display-board
  (let (...)
    (define (update-status!)
      ;; -- see if last move made 3 in a row
      ;; -- if so, set status to e.g. (won x)
      ;; -- else check for draws
      ;; -- else leave status as 'ongoing
    )
    ;; over?, has-won?, make-move!
    (lambda (op . args) ...)))
```

Bottom line

- We want to see good design
- Even if all details are not exactly right, get lots of points for logical design
- Given good design, rest is just writing standard list-processing procedures
- To summarize:
 - Decompose the problem into simple parts
 - Use wishful thinking
 - Create abstractions, and *use them!*

Bottom line

- Good luck on the exam!
- Hope to see you again...
 - in *another* course 😊