MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring 2006

Recitation 25 — 5/17/2006
Final Review

Random Streams

Assume that **ran** is a primitive Scheme procedure that generates random numbers in the range - to 1, e.g.

```
(ran)
0.486726

(ran)
0.929204

(ran)
0.08849

(ran)
0.283186
```

Assume that successive calls to RAN *never* produce the same number.

Louis Reasoner wants to define a stream whose elements consist of different random numbers, as in the sequence above. He attempts to define a stream of random numbers as follows:

```
(define random-stream
  (cons-stream (ran)
               random-stream))
```

Lem E. Tweakit isn't sure that Louis' definition will work, and he suggests the following:

```
(define (make-random-stream)
  (cons-stream (ran)
               (make-random-stream)))

(define random-stream (make-random-stream))
```

The two friends show their work to Alyssa P. Hacker who suggests that they use PRINT-STREAM to examine the first few elements of their streams. Furthermore she suggests that they run their code on two different Scheme interpreters, one that implements lazy pairs using memoization, and one that does not.

Lous and Lem take her advice, and just to be sure, they print out their streams twice. Shown below are pairs of printouts, of the sort that either Louis or Lem might have produced.

Possible Outcomes:

```
(print-stream random-stream)            ;;; OUTCOME A
0.486726 0.929204 0.008849 0.283186 ...

(print-stream random-stream)
0.486726 0.929204 0.008849 0.283186 ...


(print-stream random-stream)            ;;; OUTCOME B
0.486726 0.929204 0.008849 0.283186 ...

(print-stream random-stream)
0.486726 0.521080 0.297045 0.991644 ...


(print-stream random-stream)            ;;; OUTCOME C
0.486726 0.929204 0.008849 0.283186 ...

(print-stream random-stream)
0.365913 0.521080 0.297045 0.991644 ...


(print-stream random-stream)            ;;; OUTCOME D
0.486726 0.486726 0.486726 0.486726 ...

(print-stream random-stream)
0.486726 0.486726 0.486726 0.486726 ...


(print-stream random-stream)            ;;; OUTCOME E
0.486726 0.486726 0.486726 0.486726 ...

(print-stream random-stream)
0.591003 0.591003 0.591003 0.591003 ...
```

List all of the Possible outcomes (chosen from A,B,C,D,E) that could have been produced in each of the following cases, or indicate **none** if none of these outcomes is possible.

1. Louis' definition; no memoization
2. Louis' definition; with memoization
3. Lem's definition; no memoization
4. Lem's definition; with memoization

Evaluator

Suppose we want to add some simple type-checking to our language, that is, to specify conditions or constraints on the types of arguments that a procedure may take, with the idea that before we apply the procedure, we ensure that the supplied arguments meet those constraints. For example, we could extend our syntax to allow:

```
(define (foo (number? x ) (list? y) z)
   some-body)
```

The idea is that when we are about to appl `foo` to some arguments, we will ensure that the first argument satisfies `number?` and the second argument satisfies `list?` before proceeding. Since there are no constraints specified on the last agument, anything is acceptable.

We will do this by making two changes to our evaluator. First, when creating a `lambda`, we will get the actual procedure object associated with each type checking clause (e.g., the procedure object for `number?`). Second, when we get to `m-apply` we will actually use those procedure objects to ensure that the arguments meets the constraints. We change the dispatch clause in `m-eval` to:

```
((lambda? exp))
 (make-procedure (CONVERT (lambda-parameters exp) env)
                 (lambda-body exp)
                 env))
```

Here is the framework for `convert`, which given a list of parameters (each of which is either a name, or a list of a procedure name and a variable name), should return a list of the same form, in which each name is kept, but the procedure name is replaced by the actual procedure. Thus `((number? x) (list? y) z)` would be converted to `((<actual procedure bound to number?> x) (<actual procedure bound to list?> y) z)`:

```
(define (convert params env)
  (cond ((null? params)
         '())
        (symbol? (car params)
         (cons (car params) (convert (cdr params) env)))
        (else (cons QUESTION-1
                    (convert (cdr params) env)))))
```

1. What expression should be provided for QUESTION-1?

Next, we change the standard version of `m-apply` (which in a real final exam would be attacned at the end for reference) to handle the type checking as follows:

```
(define (m-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (let ((params (procedure-parameters procedure)))
           (if (type-ok? params arguments)
               (eval-sequence
                (procedure-body procedure)
                (extend-environment
                 (map (lambda (x) (if (symbol? x) x (cadr x)))
                      params)
                 arguments
                 (procedure-environment procedure)))
               (error "incorrect argument type" procedure))))
        (else (error "Unknown procedure type -- APPLY" procedure))))
```

To complete this change, we need to implement `type-ok?` which should check that each argument meets the specified type constraint:

```
(define (type-ok? params args)
  (cond ((null? params)
         QUESTION-2)
        ((null? args) #f)
        ((symbol? (car params))
         QUESTION-3)
        (QUESTION-4
         (type-ok? (cdr params) (cdr args)))
        (else #f)))
```

2. What expression should be used for QUESTION-2?

3. What expression should be used for QUESTION-3?

4. What expression should be used for QUESTION-4?

Lexical vs dynamic scope

```
(let ((x 20))
  (let ((f (lambda (y) (- y x))))
     (let ((x 10))
        (f 30))))
```

1. Value in a dynamically-scoped Scheme:

2. Value in a lexically-scoped Scheme:

Objects

This problem explores a small object-oriented world, consisting of Documents, Folders and Cabinets. The properties of the classes (defined by the code below) are as follows:

- a Document is an object with a name, and a number of sheets.
- a Folder is a collection of documents.
- a Cabinet is a structure that can hold Folders. It behaves as if it were a giant folder.

```
(define (create-document name sheets)
  (create-instance document name sheets))

(define (document self name sheets)
  (let ((root-part (root-object self)))
    (make-handler
     'document
     (make-methods
      'NAME    (lambda () name)
      'INSTALL (lambda () 'installed)
      'SHEETS (lambda () sheets))
     root-part)))

(define (folder self name)
  (let ((root-part (root-object self))
        (contents '()))
    (make-handler
      'folder
      (make-methods
        'NAME (lambda () name)
        'CONTENTS (lambda () contents)
        'ADD-THING (lambda (thing)
                     (set! contents (cons thing contents))
                     (map (lambda (thing) (ask thing 'NAME)) contents))
        'DOCUMENTS (lambda ()
                     (fold-right + 0
                                 (map (lambda (doc) 1) contents)))
        )
      root-part)))

(define (create-folder name)
  (create-instance folder name))

(define (cabinet self name)
  (let ((root-part (root-object self))
        (folder-part (folder self name)))
    (make-handler
      'cabinet
      (make-methods
        'NAME (lambda () name)
        'CONTENTS (lambda () (ask folder-part 'contents))
        'ADD-THING (lambda (thing)
```

```
                    (ask folder-part 'add-thing thing))
        'SHEETS (lambda () 0)
        )
      folder-part root-part)))

(define (create-cabinet name)
  (create-instance cabinet name))
```

Assume the following definitions have been evaluated:

```
(define doc1 (create-document 'doc1 10))

(define doc2 (create-document 'doc2 100))

(define folder1 (create-folder 'folder1))

(ask folder1 'add-thing doc1)
(ask folder1 'add-thing doc2)

(define cab (create-cabinet 'cab))

(ask cab 'add-thing folder1)
```

What is the value of each of the following expressions, assuming they are evaluated in the order shown? (Write *unspec* for unspecified, *no-method* for an error because there is no method, *error* for some other error, or *procedure* for a procedure value.)

1. (ask folder1 'DOCUMENTS)

2. (ask folder1 'SHEETS)

3. Add an explicit SHEETS method to Folders so that these objects will now return the total number of sheets in the documents it contains.

With that change, suppose we again evaluate:

```
(define doc1 (create-document 'doc1 10))

(define doc2 (create-document 'doc2 100))

(define folder1 (create-folder 'folder1))

(ask folder1 'add-thing doc1)
(ask folder1 'add-thing doc2)

(define cab (create-cabinet 'cab))

(ask cab 'add-thing folder1)
```

What is the value of each of the following expressions, assuming they are evaluated in the order shown? (Write *unspec* for unspecified, *error* for error, or *procedure* for a procedure value.)

4. (ask folder1 'SHEETS)

5. (ask cab 'SHEETS)

Suppose we remove the SHEETS method from the class definition for a cabinet.

With that change, suppose we again evaluate:

```
(define doc1 (create-document 'doc1 10))

(define doc2 (create-document 'doc2 100))

(define folder1 (create-folder 'folder1))

(ask folder1 'add-thing doc1)
(ask folder1 'add-thing doc2)

(define cab (create-cabinet 'cab))

(ask cab 'add-thing folder1)
```

What is the value of the following expression? (Write *unspec* for unspecified, *error* for error, or *procedure* for a procedure value.)

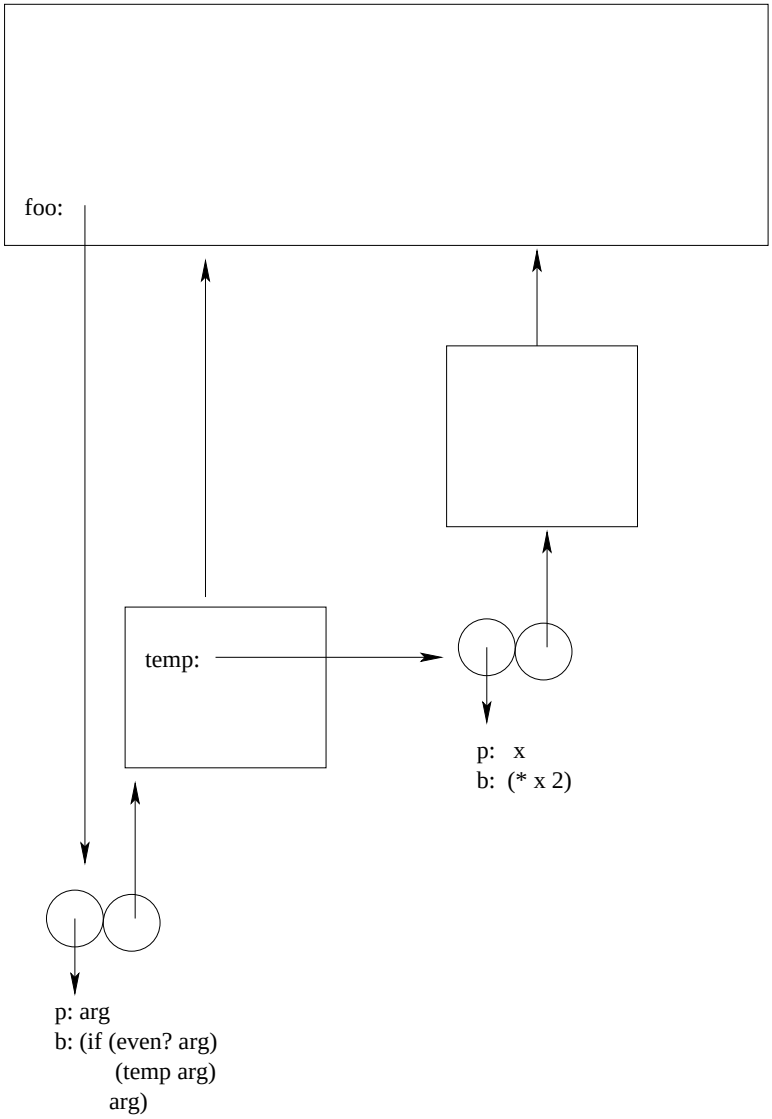
6. (ask cab 'SHEETS)

Environment Diagrams

Assume that the following definition has been evaluated:

```
(define (doubler) (lambda (x) (* x 2)))
```

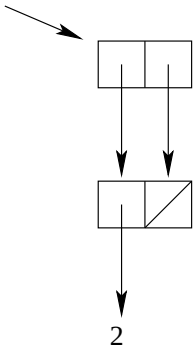
and consider the following environment diagram.



1. Write an expression (without mutation or internal defines) whose evaluation would result in this environment diagram.

Lists and pairs

Consider the following box and pointer diagram:



1. What does this print out as?

For each of the following expressions, indicate whether the expression gives rise to the box and pointer structure shown above.

2. `'((2) 2)`

3.
```
(list (list 2) 2)
```

4.
```
(let ((temp (list 2)))
   (cons temp temp))
```

5.
```
(let ((temp (list 2)))
   (let ((foo (cons (list 2) temp)))
      foo))
```

6.
```
(let ((temp (list 2)))
   (let ((foo (cons (list 2) temp)))
      (set-car! foo temp)
      foo))
```

7.
```
(let ((temp (list 2)))
   (let ((foo (cons (list 2) temp)))
      (set! (car foo) temp)
      foo))
```

Types

What is the type of the procedure `test`?

```
(define (test a b)
  (lambda (x) (a (b x))))
```

Lambda Obfuscations

What does the following evaluate to:

```
((lambda (+ - *) (* + -))
 (* 3 6)
 ((lambda (/ ^) (* ^ /))
  4 6)
 (lambda (- *) (+ - *)))
```