6.001 Jeopardy

Scheme Expressions: 100
These are the two methods which may be called on any object in the object-oriented programming system from Project 4.

* What are IS-A and TYPE?

Scheme Expressions: 200
This is printed in response to the second expression:

```
(define f
  (lambda (/)
    (lambda (a b)
      (b / a))))
((f 6) 2 -)
```

* What is 4?
The usual name for the built-in Scheme function computed by this procedure:

```
(define (what? p x)
  (fold-right
    (lambda (a b)
      (cons (p a) b))
    nil x))
```

* What is map?

If `double` is a procedure that takes a procedure of one argument and returns a procedure that applies the original procedure twice, this is the value returned by:

```
(((double (double double)) inc) 5)
```

* What is 21?

This function of one argument, an infinite stream, produces as output an infinite stream whose values are the pair-wise averages of the input stream. e.g.

```
(smooth <stream 1 3 6 2 ... >)  
  -> <stream 2 4.5 4 ... >
```

* What is
```
(define (smooth s)
  (cons-stream
    (/ (+ (stream-car s)
         (stream-car (stream-cdr s))))
    2)
    (smooth (stream-cdr s)))
  ?
```
The number of cons cells in the following data structure:

(list (cons (list 1 2) (list)) 3)

* What is 5?

A mathematical description for the stream:

(define foo
 (cons-stream 1
 (add-streams
 foo foo)))

* What are the powers of two?

It is the printed value of the last expression:

(define x '(a b x))
(define y (list x x (list 'x x)))
(set-cdr! (cdr y) (list (quote x)))
y

* What is ((a b x) (a b x) x)?
**Daily Double!**

Data: 400

This scheme code (which doesn't use quotation) would print out as:

```
((1 . 2) 3 . 4)
```

* What is

```
(cons (cons 1 2) (cons 3 4))
```

---

Data: 400

The scheme expression(s) needed to create this data structure:

```
(define x (list 1 2 3))
(set-car! x (cddr x))
(set-car! (cdr x) x)
(set-car! (cddr x) (cdr x))
(set-car! (cddr x) x)
```

---

Data: 500

The value of the following expression:

```
(let ((a 3))
  (let ((a 4)
        (b a))
    (list a b)))
```

---

Evaluation: 100

The value of the following expression:

```
(let ((a 3))
  (let ((a 4)
        (b a))
    (list a b)))
```
Evaluation: 100

The value of the following expression:

(let ((a 3))
  (let ((a 4)
        (b a))
      (list a b)))

* What is (4 3) ?

Evaluation: 200

The number of times m-eval is invoked when the following expression is entered into the evaluator:

((lambda (x) (* x 2)) 3)

* What is 7: combination, lambda, 3, (* x 2), *, x, 2 ?

Evaluation: 200

The number of times m-eval is invoked when the following expression is entered into the evaluator:

((lambda (x) (* x 2)) 3)

* What is 7: combination, lambda, 3, (* x 2), *, x, 2 ?

Evaluation: 300

Using this type of evaluation some constructs (such as if, and, & or) would not need to be special forms.

* What is normal order/lazy application?

Evaluation: 300

Using this type of evaluation some constructs (such as if, and, & or) would not need to be special forms.

* What is normal order/lazy application?

Evaluation: 400

The result of evaluating this expression:

(letrec
  ((fact (lambda (n)
           (* n (fact (decr n))))))
   (decr (lambda (x) (- x 1))))
 (fact 4))
**Evaluation: 400**

The result of evaluating this expression:

(letrec
  ((fact (lambda (n)
          (* n (fact (decr n))))))
  (decr (lambda (x) (- x 1))))
(fact 4))

* What is an infinite loop?

**Evaluation: 500**

The correct matching of the following three expressions:

A: In applicative order...
B: In normal order without memoization...
C: In normal order with memoization...

...the arguments passed in to a combination...

1: ... are evaluated at most once.
2: ... are evaluated exactly once.
3: ... may be evaluated many times or not at all.

**Evaluation: 500**

The correct matching of the following three expressions:

* What is A-2, B-3, C-1?

**Environment Model: 100**

If you program without these, the order of evaluation is not important and the substitution model is sufficient. Repeated evaluation of sub-expressions may affect performance, but not the resulting value.

**Environment Model: 100**

If you program without these, the order of evaluation is not important and the substitution model is sufficient. Repeated evaluation of sub-expressions may affect performance, but not the resulting value.

* What is a side effect?

**Environment Model: 200**

The opposite of syntax, changing this may affect how the environment model is drawn.
Environment Model: 200

The opposite of syntax, changing this may affect how the environment model is drawn.

* What are the semantics of a language?

Environment Model: 300

The environment diagram resulting from the evaluation of this expression:

```
(define f ((lambda ()
  (define x 10)
  (lambda (y)
    (+ x y))))
```

Environment Model: 300

What is:

f: p: b: (+ x y)

x: 10

Environment Model: 400

Under dynamic scoping, the value of the last expression below:

```
(define op square)
(define (foo op) (op a))
(define a 4)
(let ((a 9)
  (op (lambda (x) x)))
  (foo sqrt))
```

Environment Model: 400

Under dynamic scoping, the value of the last expression below:

```
(define op square)
(define (foo op) (op a))
(define a 4)
(let ((a 9)
  (op (lambda (x) x)))
  (foo sqrt))
```

* What is 3?

Environment Model: 500

This scheme expression results in the following environment diagram:

```
(define op square)
(define (foo op) (op a))
(define a 4)
(let ((a 9)
  (op (lambda (x) x)))
  (foo sqrt))
```

* What is 3?
Environment Model: 500

* What is
(define foo
  (let ((bar (let ((x 10))
              (lambda () x)))
       (lambda () (bar)))))?
Computing Theory: 300

The order of growth in space and the order of growth in time of this function:

```scheme
(define (sort lst)
  (define (insert elt lst)
    (if (or (null? lst) (< elt (car lst)))
      (cons elt lst)
      (cons (car lst) (insert elt (cdr lst)))))
  (define (sort-iter answer rest)
    (if (null? rest)
      answer
      (sort-iter (insert (car rest) answer)
        (cdr rest)))))
  (sort-iter '() lst))
```

* What is $O(n)$ space and $O(n^2)$ time?

---

Computing Theory: 400

The type of this Scheme expression:

```scheme
(define (swap-args f)
  (lambda (x y) (f y x)))
```

* What is $\text{swap-args}: (a,b\rightarrow c) \rightarrow (b,a\rightarrow c)$?

---

Computing Theory: 500

The order of growth in time and the order of growth in space of this function:

```scheme
(define (h n)
  (if (= n 0)
    1
    (+ (h (quotient n 2))
      (h (quotient n 2))))))
```

* What is $O(n)$ time and $O(\log n)$ space?
Terminology: 100

Any procedure that takes a procedure as an argument or returns a procedure as a value.

* What is a higher-order procedure?

Terminology: 200

This type of recursion does not require use of the stack.

* What is tail recursion?

Terminology: 300

Shorthand for “the contents of the address portion of the register”.

* What is car?
This object-oriented programming technique is often the most concise way to extend the interfaces of several types, although it can be challenging to correctly specify the behavior when names overlap.

* What is multiple inheritance?

The problem with the following fragment of code:

```
(define make-vector cons)
(define vector-x car)
(define vector-y cdr)
(define v1 (make-vector 2 3))
(define (magnitude v)
  (let ((cars (* (car vec) (car vec)))
        (cdrs (* (cdr vec) (cdr vec)))
        (sqrt (+ cars cdrs))))
```

* What is an abstraction violation?

The inner door combo to get into the 6.001 lab.

* What is 21634?
The hero of project 4 and his institution.

* Who is Hairy Cdr from the Wizard's Institute of Technocracy?

These guys make origami and download music from Napster and claim it's research.

* Who are Professors Erik Demaine and Frans Kaashoek?

The architect for our crazy new computer science building.

* Who is Frank O. Gehry?
* Who is Professor Eric Grimson, the 6.001 online lecturer?

---

Final Jeopardy

**Category:**

Capturing local state

---

This function takes one argument and returns `#t` if the argument has the same value as on the previous call to the function and `#f` otherwise. The first call to the function returns `#f`.

---

* What is

```
(define previous
    (let ((last #f)
            (initialized #f))
        (lambda (x)
            (if (and initialized (equal? x last))
                #t
                (begin (set! last x)
                        (set! initialized #t)
                        #f)))))
```

---

Not on the 6.001 Final: 500