Lambda Calculus - 100

What the following code returns

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

Lambda Calculus - 200

The value returned by this expression:

```
((lambda (x f)
   (f (f x)))
 3
 (lambda (y)
   (+ y y)))
```

Lambda Calculus - 300

Given the following definition of f:

```
(define f
  (lambda (x)
    (x (lambda (y) (* y 2)))))
```

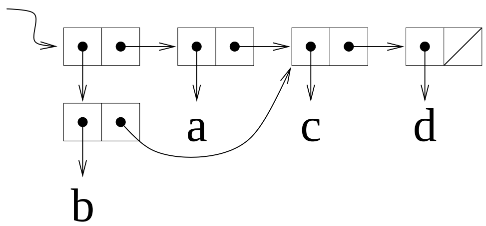
It's the expression which, when f is applied to it, returns 6.

Lambda Calculus - 400

A function, that when applied to itself, returns a function, that when applied to 17 returns 17.

Lists - 100

The printed representation in Scheme of the following box and pointer diagram:



Lists - 200

The expression returned by the following code:

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

Lists - DAILY DOUBLE

If we were to implement cons, car, and cdr as procedures, by writing cons as a procedure of its two arguments, like so:

```
(define (cons x y)
  (lambda (m) (m x y)))
```

then this is how “cdr” would be defined.

Lists - 400

The missing expressions in this following definition

```
(define (accumulate f init lst)
  (if (null? lst)
      init
      (___________ ___________
         (accumulate f init
           (cdr lst)))))
```

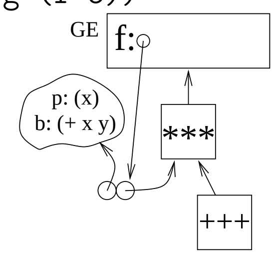
Environment Model - 100

The reason that the environmental model is useful:
(a) procedures may contain free variables
(b) environments use frames
(c) the substitution model is inadequate to deal with procedural side effects
(d) your TA likes to see you extremely confused
(e) garbage collection takes a shorter amount of time for environmental models

Environment Model - 200

The expressions that should appear in place of the asteriks and the pluses in the environment diagram below, corresponding to the following code:

```
(define (f x)
  (let ((y 10))
    (lambda (x) (+ x y))))
(define g (f 5))
```



Environment Model - 300

In a lexically scoped language like scheme, this is, by definition, where free variables in procedures passed as arguments are looked up:
(a) in the environment where the procedure is called
(b) in the environment where the lambda expression was evaluated
(c) in the global environment
(d) in the primitive list of the global environment
(e) in Billings, Montana

Environment Model - 400

These are the steps that result from applying a procedure in the environment model.

Register Machines - 100

It is the error in this statement:

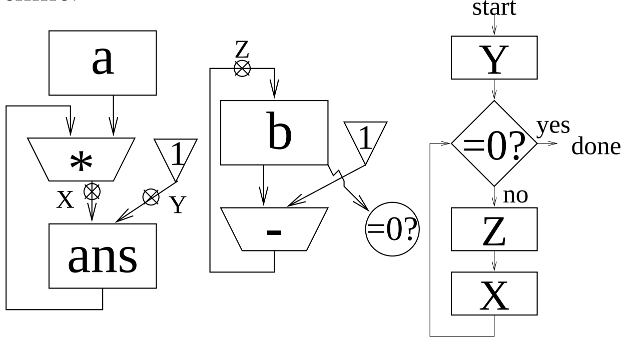
```
(assign lst (car (cdr (reg lst))))
```

Register Machines - 200

The definition of stack discipline.

Register Machines - 300

The function computed by the following machine:



Register Machines - 400

The function performed on registers x and y by the following register machine.

```
(define-machine mystery
 (register x y aux val continue)
 (controller
   (assign continue (label mystery-done))
 mystery-loop
   (test (op null?) (reg x))
   (branch (label base-case))
   (assign aux (op car) (reg x))
   (save continue)
   (save aux)
   (assign x (op cdr) (reg x))
   (assign continue (label after-loop))
   (goto (label mystery-loop))
 after-loop
   (restore x)
   (restore continue)
   (assign val (op cons) (reg x) (reg val))
   (goto (reg continue))
 base-case
   (assign val (reg y))
   (goto (reg continue))
 mystery-done))
```

Compilers - 100

Either of the two biggest advantages of a compiler over an interpreter.

Compilers - 200

The Scheme fragment that created the following code:

```
  (assign proc (op lookup-variable-value) (const lst) (reg env))
  (assign val (op lookup-variable-value)  (const null?) (reg env))
  (assign argl (op list) (reg val))
  (test (op primitive-procedure?) (reg proc))
  (branch (label prim-branch11))
compound-branch12
  (assign continue (label after-call71))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
prim-branch11
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
after-call71
```

Compilers - 300

When interpreted code and compiled code are compared, these are the instructions eliminated most often.

Compilers - 400

The missing line in the code, which is the result of compiling `(f (+ 1 x) y)`:

```
  (assign proc (op lookup-variable-value) (const f) (reg env))
  (save proc)
  (save env)
  (assign proc (op lookup-variable-value) (const +) (reg env))
  (assign val (op lookup-variable-value) (const x) (reg env))
  (assign argl (op list) (reg val))
  (assign val (const 1))
  (assign argl (op cons) (reg val) (reg argl))
  <apply-dispatch>
after-call21
  ________________________________
  (restore env)
  (assign val (op lookup-variable-value) (const y) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (restore proc)
  <apply-dispatch>
```

Miscellaneous - 100

Carver Mead is now working on these; Alan Turing was working on the same when he died.

Miscellaneous - 200

Your recitation instructor's email address (spelled correctly)

Miscellaneous - 300

This is commonly used to protect a disclosed invention from being used by others.

(a) Copyright
(b) Patent
(c) Court Order
(d) Jesse "The Body" Ventura
(e) Trade Secret

Miscellaneous - 400

He developed LISP.

Orders of Growth - 100

The simplest way the following expression can be written in big theta notation:

$$n \log(n^2) + (\log(n))^2$$

Orders of Growth - 200

The orders of growth in time and space of:

```
(define (f n)
  (if (= n 0)
      1
      (f (- n 1))))
```

Orders of Growth - 300

The orders of growth in time and space of:

```
(define (g n)
  (if (= n 0)
      1
      (+ (g (- n 1))
         (g (- n 1)))))
```

Orders of Growth - 400

The orders of growth in time and space of:

```
(define (h n)
  (if (= n 0)
      1
      (+ (h (quotient n 3))
         (h (quotient n 3)))))
```

Streams - 100

It's the method streams use that prevents the need for repetitive calculations.

Streams - 200

The missing expressions in the definition below, which produces the following stream:
`(2,1,4,3,6,5,8,7,10,...)`

```
(define s
  (cons-stream 2
    (cons-stream 1
      (stream-map + __________ __________ ))))
```

Streams - 300

Lists are to streams as __________ order is to __________ order.

Streams - 400

What the following mystery stream calulates:

```
(define foo
  (cons-stream 1
    (cons-stream 2
      (stream-map *
        (stream-cdr
          (stream-cdr integers))
        (stream-cdr foo)))))
```

Object Oriented Programming - 100

In the following example, this class inherits from this (other) class:

```
(define (make-dairy-product name temp)
  (let ((container 'none)
        (bad false)
        (scent 'lemon)
        (food-obj (make-food name temp)))
   (lambda (message)
     (cond ((eq? message 'name) (lambda (self) name))
           ((eq? message 'scent) (lambda (self) scent))
           ((eq? message 'spoiled?)
             (lambda (self) (set! scent 'vile) true))
                 (else (get-method food-obj message))))))
```

Object Oriented Programming - 200

The value of inheritance in object oriented languages is that it makes it convenient to define new kinds of objects:

(a) recursively
(b) that send messages to other objects
(c) that enable a student to pass 6.001
(d) as variants of previously defined objects
(e) without using applicative order

Object Oriented Programming - 300

By convention, we implement all methods in object- oriented code to accept an argument named "self" for this reason.

Object Oriented Programming - 400

In an effort to better integrate the worlds of biology and computer science, Ben Bitdiddle sets out to write a Scheme program which could be used to determine the gender of a woman's imminent child, as an alternative to the more invasive clinical procedures:

```
(define (make-kid)
  (lambda (self msg)
    (cond ((eq? msg 'male?) (not (ask self 'female?)))
  ((eq? msg 'female?) (not (ask self 'male?))))))

(define (ask kid msg) (kid kid msg))

(define patients-kid (make-kid))

(ask patients-kid 'female?)
```

This would be the response:
(a) true
(b) false
(c) no response (runs forever)
(d) error response
(e) none of the above

Meta Circular Evaluator - 100

This is how environments are represented in our meta-circular evaluator.

Meta Circular Evaluator - 200

The value of the following expression in a *dynamic-binding* Scheme:

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

Meta Circular Evaluator - 300

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

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

Meta Circular Evaluator - Daily Double

The one and only line needed to modify the evaluator to handle define statements of the form:

```
(<variable> := <binding>)
```

Potpourri - 100

What LISP stands for

Potpourri - 200

Any one of Professor Grimson's bad jokes from lecture

Potpourri - 300

The inventors of Scheme.

Potpourri - 400

The person(s) to whom there is a seat dedicated in the 10-250 lecture hall.

(a) Hal Abelson
(b) Eric Grimson
(c) Gerry Sussman
(d) Ben and Alyssa P. (Hacker) Bitdiddle
(e) Louis Reasoner

Final Jeopardy - Lists

The value of the following expression:

```
(apply map
   (cons list
     (quote
       ((good thanks have) (luck for a)
        (on a fun) (the great summer)
        (final semester break)))))
```