CS61A – Lab Assignment 1.1	**Kurt Meinz**
University of California, Berkeley	Summer 2002

Try to get as much done as possible, but don't panic if you don't finish everything.

0. Login to your user account and change your password – instructions are provided on the account form. Be aware that it may take several minutes for your new password to be recognized by all the machines.

1. Start the Emacs editor, either by typing `emacs` in your main window or by selecting it from the alt-middle mouse menu. (Your TA will show you how to do this.) From the `Help` menu, select the Emacs tutorial. You need not complete the entire tutorial at the first session, but you should do so eventually.

2. Start Scheme, either by typing `scm` in your main window or by typing meta-S in your Emacs window. Type each of the following expressions into Scheme, ending the line with the Enter (carriage return) key. **Think about the results!** Try to understand how Scheme interprets what you type.

```
3
(+ 2 3)
(+ 5 6 7 8)
(+)
(sqrt 16)
(+ (* 3 4) 5)
+
'+
'hello
'(+ 2 3)
'(good morning)
(first 274)
(butfirst 274)
(first 'hello)
(first hello)
(first (bf 'hello))
(+ (first 23) (last 45))
(define pi 3.14159)
pi
'pi
(+ pi 7)
(* pi pi)
(define (square x) (* x x))
(square 5)
(square (+ 2 3))
```

3. Use Emacs to create a file called `pigl.scm` in your directory containing the Pig Latin program shown below:

```
(define (pigl wd)
  (if (pl-done? wd)
      (word wd 'ay)
      (pigl (word (bf wd) (first wd)))))

(define (pl-done? wd)
  (vowel? (first wd)))

(define (vowel? letter)
  (member? letter '(a e i o u)))
```

If you end each line with the linefeed key, instead of the return key, Emacs will automatically indent the lines of your program properly.

4. Now run Scheme. You are going to create a transcript of a session using the file you just created, like this:

```
(transcript-on "lab1")       ; This starts the transcript file.
(load "pigl.scm")            ; This reads in the file you created earlier.
(pigl 'scheme)               ; Try out your program.
                             ; Feel free to try more test cases here!
(trace pigl)                 ; This is a debugging aid. Watch what happens
(pigl 'scheme)               ; when you run a traced procedure.
(transcript-off)
(exit)
```

5. Use `lpr` to print your transcript file.

Continued on next page.

Lab Assignment 1.1 continued...

6. Predict what Scheme will print in response to each of these expressions. *Then* try it and make sure your answer was correct, or if not, that you understand why!

```
(define a 3)

(define b (+ a 1))

(+ a b (* a b))

(= a b)

(if (and (> b a) (< b (* a b)))
    b
    a)

(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))

(+ 2 (if (> b a) b a))

(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
   (+ a 1))

((if (< a b) + -) a b)
```

7. In the shell, type the command

```
cp ~cs61a/lib/plural.scm .
```

(Note the period at the end of the line!) This will copy a file from the class library to your own directory. Then, using emacs to edit the file, modify the procedure so that it correctly handles cases like `(plural 'boy)`.

8. Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

9. Write a procedure `dupls-removed` that, given a sentence as input, returns the result of removing duplicate words from the sentence. It should work this way:

```
> (dupls-removed '(a b c a e d e b))
(c a d e b)
> (dupls-removed '(a b c))
(a b c)
> (dupls-removed '(a a a a b a a))
(b a)
```

CS61A – Lab Assignment 1.2	**Kurt Meinz**
University of California, Berkeley	Summer 2002

1. For each of the following expressions, what must `f` be in order for the evaluation of the expression to succeed, without causing an error? For each expression, give a definition of `f` such that evaluating the expression will not cause an error, and say what the expression's value will be, given your definition.

```
f

(f)

(f 3)

((f))

(((f)) 3)
```

2. Find the values of the expressions

```
((t 1+) 0)

((t (t 1+)) 0)

(((t t) 1+) 0)
```

where `1+` is a primitive procedure that adds 1 to its argument, and `t` is defined as follows:

```
(define (t f)
  (lambda (x) (f (f (f x)))) )
```

Work this out yourself before you try it on the computer!

3. Find the values of the expressions

```
((t s) 0)

((t (t s)) 0)

(((t t) s) 0)
```

where `t` is defined as in question 2 above, and `s` is defined as follows:

```
(define (s x)
  (+ 1 x))
```

4. Consider a Scheme function `g` for which the expression

```
((g) 1)
```

returns the value 3 when evaluated. Determine how many arguments `g` has. In one word, also describe as best you can the *type* of value returned by `g`.

5. Write a procedure `substitute` that takes three arguments: a *new* word, an *old* word, and a sentence. It should return a copy of the sentence, but with every occurrence of the old word replaced by the new word. For example:

```
> (substitute 'maybe 'yeah '(she loves you yeah yeah yeah))
(she loves you maybe maybe maybe)
```

Continued on next page.

Lab Assignment 1.2 continued...

6. First, type the definitions

```
(define a 7)
(define b 6)
```

into Scheme. Then, fill in the blank in the code below with an expression whose value depends on both `a` and `b` to determine a return value of 24. Verify in Scheme that the desired value is obtained.

```
(let
  ((a 3) (b (+ a 2)))
  ______________ )
```

7. Write and test the `make-tester` procedure. Given a word `w` as argument, `make-tester` returns a procedure of one argument `x` that returns true if `x` is equal to `w` and false otherwise. Examples:

```
> ((make-tester 'hal) 'hal)
#t
> ((make-tester 'hal) 'cs61a)
#f
> (define sicp-author-and-astronomer? (make-tester 'gerry))
> (sicp-author-and-astronomer? 'hal)
#f
> (sicp-author-and-astronomer? 'gerry)
#t
```

CS61A – Lab Assignment 2.1	**Kurt Meinz**
University of California, Berkeley	Summer 2002

This lab exercise concerns the change counting program on pages 40–41 of Abelson and Sussman.

1. Identify two ways to change the program to *reverse* the order in which coins are tried, that is, to change the program so that pennies are tried first, then nickels, then dimes, and so on.

2. Abelson and Sussman claim that this change would not affect the *correctness* of the computation. However, it does affect the *efficiency* of the computation. Implement one of the ways you devised in exercise 1 for reversing the order in which coins are tried, and determine the extent to which the number of calls to cc is affected by the revision. Verify your answer on the computer, and provide an explanation. Hint: limit yourself to nickels and pennies, and compare the trees resulting from `(cc 5 2)` for each order.

3. Modify the `cc` procedure so that its `kinds-of-coins` parameter, instead of being an integer, is a *sentence* that contains the values of the coins to be used in making change. The coins should be tried in the sequence they appear in the sentence. For the `count-change` procedure to work the same in the revised program as in the original, it should call `cc` as follows:

```
(define (count-change amount)
  (cc amount '(50 25 10 5 1)) )
```

4. Many Scheme procedures require a certain type of argument. For example, the arithmetic procedures only work if given numeric arguments. If given a non-number, an error results.

Suppose we want to write *safe* versions of procedures, that can check if the argument is okay, and either call the underlying procedure or return `#f` for a bad argument instead of giving an error. (We'll restrict our attention to procedures that take a single argument.)

```
> (sqrt 'hello)
ERROR: magnitude: Wrong type in arg1 hello
> (type-check sqrt number? 'hello)
#f
> (type-check sqrt number? 4)
2
```

Write `type-check`. Its arguments are a function, a type-checking predicate that returns `#t` if and only if the datum is a legal argument to the function, and the datum.

Continued on next page.

Lab Assignment 2.1 continued...

5. We really don't want to have to use `type-check` explicitly every time. Instead, we'd like to be able to use a `safe-sqrt` procedure:

```
> (safe-sqrt 'hello)
#f
> (safe-sqrt 4)
2
```

Don't write `safe-sqrt`! Instead, write a procedure `make-safe` that you can use this way:

```
> (define safe-sqrt (make-safe sqrt number?))
```

It should take two arguments, a function and a type-checking predicate, and return a new function that returns `#f` if its argument doesn't satisfy the predicate.

CS61A – Lab Assignment 2.2 **Kurt Meinz**
University of California, Berkeley Summer 2002

1. Try these in Scheme:

```
(define x (cons 4 5))

(car x)

(cdr x)

(define y (cons 'hello 'goodbye))

(define z (cons x y))

(car (cdr z))

(cdr (cdr z))
```

2. Predict the result of each of these before you try it:

```
(cdr (car z))

(car (cons 8 3))

(car z)

(car 3)
```

3. Enter these definitions into Scheme:

```
(define (make-rational num den)
  (cons num den))

(define (numerator rat)
  (car rat))

(define (denominator rat)
  (cdr rat))

(define (*rat a b)
  (make-rational (* (numerator a) (numerator b))
                 (* (denominator a) (denominator b))))

(define (print-rat rat)
  (word (numerator rat) '/ (denominator rat)))
```

4. Try this:

```
(print-rat (make-rational 2 3))

(print-rat (*rat (make-rational 2 3) (make-rational 1 4)))
```

5. Define a procedure `+rat` to add two rational numbers, in the same style as `*rat` above.

6. **Now do exercises 2.2, 2.3, and 2.4 from *SICP*.**

7. *SICP* ex. 2.18; this should take some thought, and you should make sure you get it right, but don't get stuck on it for the whole hour. **Note:** Your solution should reverse *lists*, not sentences! That is, you should be using `cons`, `car`, and `cdr`, not `first`, `sentence`, etc.

CS61A – Lab Assignment 3.1 **Kurt Meinz**
University of California, Berkeley Summer 2002

1. *SICP* ex. 2.25 and 2.53; these should be quick and easy.

2. *SICP* ex. 2.55; **explain your answer to your TA.**

3. *SICP* ex. 2.27. This is the central exciting adventure of today's lab! Think hard about it.

4. Each person individually make up a procedure named `mystery` that, given two lists as arguments, returns the result of applying *exactly two* of `cons`, `append`, or `list` to `mystery`'s arguments, using no quoted values or other procedure calls. Here are some examples of what is and is not fair game:

okay	not okay
`(define (mystery L1 L2)` `  (cons L1 (append L2 L1)))`	`(define (mystery L1 L2)` `  (cons L1 (cons L2 (cons L1 L2))))`
`(define (mystery L1 L2)` `  (list L1 (list L1 L1)))`	`(define (mystery L1 L2)` `  (cons L1 L2))`
`(define (mystery L1 L2)` `  (append (cons L2 L2) L1))`	`(define (mystery L1 L2)` `  (append L1 (cons L1 '(A B C))))`

Type your `mystery` definition into a file, and have one of your partners load it into Scheme and try to guess what it is by trying it out with various arguments.

After everyone has tried someone else's procedure, decide with your partners which procedure was hardest to guess and why, and what test cases were most and least helpful in revealing the definitions.

CS61A – Lab Assignment 3.2	**Kurt Meinz**
University of California, Berkeley	Summer 2002

Start by reading *SICP* section 2.3.3 (pages 151–161).

1. *SICP* ex. 2.62.

2. The file `~cs61a/lib/bst.scm` contains the binary search tree procedures from pages 156–157 of *SICP*. Using `adjoin-set`, construct the trees shown on page 156.

3. *SICP* ex. 2.74.

CS61A – Lab Assignment 4.1	**Kurt Meinz**
University of California, Berkeley	Summer 2002

1. Modify the `person` class given in the lecture notes for week 3 (it's in the file `demo2.scm` in the `~cs61a/lectures/3.0` directory) to add a `repeat` method, which repeats the last thing said. Here's an example of responses to the `repeat` message.

```
> (define brian (instantiate person 'brian))
brian
> (ask brian 'repeat)
()
> (ask brian 'say '(hello))
(hello)
> (ask brian 'repeat)
(hello)
> (ask brian 'greet)
(hello my name is brian)
> (ask brian 'repeat)
(hello my name is brian)
> (ask brian 'ask '(close the door))
(would you please close the door)
> (ask brian 'repeat)
(would you please close the door)
```

2. This exercise introduces you to the `usual` procedure described on page 9 of "Object-oriented Programming – Above-the-line View". Read about `usual` there to prepare for lab. Suppose that we want to define a class called `double-talker` to represent people that always say things twice, for example as in the following dialog.

```
> (define mike (instantiate double-talker 'mike))
mike
> (ask mike 'say '(hello))
(hello hello)
> (ask mike 'say '(the sky is falling))
(the sky is falling the sky is falling)
```

Consider the following three definitions for the `double-talker` class. (They can be found online in the file `~cs61a/lib/double-talker.scm`.)

```
(define-class (double-talker name)
  (parent (person name))
  (method (say stuff) (se (usual 'say stuff) (ask self 'repeat))) )

(define-class (double-talker name)
  (parent (person name))
  (method (say stuff) (se stuff stuff)) )

(define-class (double-talker name)
  (parent (person name))
  (method (say stuff) (usual 'say (se stuff stuff))) )
```

Determine which of these definitions work as intended. Determine also for which messages the three versions would respond differently.

CS61A – Lab Assignment 4.2	**Kurt Meinz**
University of California, Berkeley	Summer 2002

1. Given below is a simplified version of the `make-account` procedure on page 223 of Abelson and Sussman.

```
(define (make-account balance)
  (define (withdraw amount)
    (set! balance (- balance amount)) balance)
  (define (deposit amount)
    (set! balance (+ balance amount)) balance)
  (define (dispatch msg)
    (cond
      ((eq? msg 'withdraw) withdraw)
      ((eq? msg 'deposit) deposit) ) )
  dispatch)
```

Fill in the blank in the following code so that the result works exactly the same as the `make-account` procedure above, that is, responds to the same messages and produces the same return values. The differences between the two procedures are that the inside of `make-account` above is enclosed in the `let` below, and the names of the parameter to `make-account` are different.

```
(define (make-account init-amount)
  (let ( underbar               )
    (define (withdraw amount)
      (set! balance (- balance amount)) balance)
    (define (deposit amount)
      (set! balance (+ balance amount)) balance)
    (define (dispatch msg)
      (cond
        ((eq? msg 'withdraw) withdraw)
        ((eq? msg 'deposit) deposit) ) )
    dispatch) )
```

2. Modify either version of `make-account` so that, given the message `balance`, it returns the current account balance, and given the message `init-balance`, it returns the amount with which the account was initially created. For example:

```
> (define acc (make-account 100)
acc
> (acc 'balance)
100
```

Continued on next page...

Lab Assignment 4.2 continued:

3. Modify `make-account` so that, given the message `transactions`, it returns a list of all transactions made since the account was opened. For example:

```
> (define acc (make-account 100))
acc
> ((acc 'withdraw) 50)
50
> ((acc 'deposit) 10)
60
> (acc 'transactions)
((withdraw 50) (deposit 10))
```

4. Given this definition:

```
(define (plus1 var)
  (set! var (+ var 1))
  var)
```

Show the result of computing

```
(plus1 5)
```

using the substitution model. That is, show the expression that results from substituting `5` for `var` in the body of `plus1`, and then compute the value of the resulting expression. What is the actual result from Scheme?

CS61A – Lab Assignment 5.1 — **Kurt Meinz**

University of California, Berkeley — Summer 2002

1. This lab activity consists of example programs for you to run in Scheme. Predict the result before you try each example. If you don't understand what Scheme actually does, ask for help! Don't waste your time by just typing this in without paying attention to the results.

```
(define (make-adder n)
  (lambda (x) (+ x n)))

(make-adder 3)

((make-adder 3) 5)

(define (f x) (make-adder 3))

(f 5)

(define g (make-adder 3))

(g 5)

(define (make-funny-adder n)
  (lambda (x)
    (if (equal? x 'new)
        (set! n (+ n 1))
        (+ x n))))

(define h (make-funny-adder 3))

(define j (make-funny-adder 7))

(h 5)

(h 5)

(h 'new)

(h 5)

(j 5)

(let ((a 3))
  (+ 5 a))

(let ((a 3))
  (lambda (x) (+ x a)))

((let ((a 3))
   (lambda (x) (+ x a)))
 5)

((lambda (x)
   (let ((a 3))
     (+ x a)))
 5)

(define k
  (let ((a 3))
    (lambda (x) (+ x a))))

(k 5)

(define m
  (lambda (x)
    (let ((a 3))
      (+ x a))))

(m 5)

(define p
  (let ((a 3))
    (lambda (x)
      (if (equal? x 'new)
          (set! a (+ a 1))
          (+ x a)))))

(p 5)

(p 5)

(p 'new)

(p 5)

(define r
  (lambda (x)
    (let ((a 3))
      (if (equal? x 'new)
          (set! a (+ a 1))
          (+ x a)))))

(r 5)

(r 5)

(r 'new)
```

Continued on next page...

Lab Assignment 5.1 continued:

```
(define s
  (let ((a 3))
    (lambda (msg)
      (cond ((equal? msg 'new)
             (lambda ()
               (set! a (+ a 1))))
            ((equal? msg 'add)
             (lambda (x) (+ x a)))
            (else (error "huh?"))))))

(s 'add)

(s 'add 5)

((s 'add) 5)

(s 'new)

((s 'add) 5)

((s 'new))

((s 'add) 5)
```

```
(r 5)

(define (ask obj msg . args)
  (apply (obj msg) args)))

(ask s 'add 5)

(ask s 'new)

(ask s 'add 5)

(define x 5)

(let ((x 10)
      (f (lambda (y) (+ x y))))
  (f 7))

(define x 5)
```

2. Exercise 3.12 of Abelson and Sussman.

3. Suppose that the following definitions have been provided.

`(define x (cons 1 3)) (define y 2)` A CS 61A student, intending to change the value of `x` to a pair with `car` equal to 1 and `cdr` equal to 2, types the expression `(set! (cdr x) y)` instead of `(set-cdr! x y)` and gets an error. Explain why.

4a. Provide the arguments for the two `set-cdr!` operations in the blanks below to produce the indicated effect on `list1` and `list2`. Do not create any new pairs; just rearrange the pointers to the existing ones.

```
> (define list1 (list (list 'a) 'b))
list1
> (define list2 (list (list 'x) 'y))
list2
> (set-cdr! _______________ _______________ )
okay
> (set-cdr! _______________ _______________ )
okay
> list1
((a x b) b)
> list2
((x b) y)
```

4b. After filling in the blanks in the code above and producing the specified effect on `list1` and `list2`, draw a box-and-pointer diagram that explains the effect of evaluating the expression `(set-car! (cdr list1) (cadr list2))` .

5. Exercises 3.13 and 3.14 in Abelson and Sussman.

CS61A – Lab Assignment 5.2	**Kurt Meinz**
University of California, Berkeley	Summer 2002

In this lab there is no actual Scheme programming; you are to devise an algorithm in English to solve

The Dining Philosophers Problem

N philosophers are sitting around a round table for dinner. There is one chopstick between each pair of philosophers, for a total of N chopsticks altogether. You need two chopsticks to eat. Philosophers spend most of their time thinking, but every so often one gets hungry and wants to eat. A hungry philosopher must obtain the two chopsticks on his or her left and right. If one or both of those chopsticks is already in use, the philosopher must wait.

How can you use synchronization to solve that problem? You must come up with a solution that's

1. correct (so you don't have two philosophers using the same chopstick at the same time),

2. efficient (so you don't restrict eating more than necessary—for example, you shouldn't have only one person allowed to eat at a time),

3. not deadlocked (don't end up with each philosopher holding one chopstick), and

4. preferably fair (so they all get an equal chance to eat).

Be sure your solution works for both even and odd values of N.

CS61A – Lab Assignment 6.1	**Kurt Meinz**
University of California, Berkeley	Summer 2002

1. What is the type of the value of `(delay (+ 1 27))`? What is the type of the value of `(force (delay (+ 1 27)))`?

2. Evaluation of the expression

```
(stream-cdr (stream-cdr (cons-stream 1 '(2 3))))
```

produces an error. Why?

3. Consider the following two procedures.

```
(define (enumerate-interval low high)
  (if (> low high)
    '()
    (cons low (enumerate-interval (+ low 1) high)) ) )

(define (stream-enumerate-interval low high)
  (if (> low high)
    the-empty-stream
    (cons-stream low (stream-enumerate-interval (+ low 1) high)) ) )
```

What's the difference between the following two expressions?

```
(delay (enumerate-interval 1 3))
(stream-enumerate-interval 1 3)
```

4. An unsolved problem in number theory concerns the following algorithm for creating a sequence of positive integers $s_1, s_2, ...$

Choose s_1 to be some positive integer.
For $n > 1$,
    if s_n is odd, then s_{n+1} is $3\ s_n + 1$;
    if s_n is even, then s_{n+1} is $s_n\ /\ 2$.

No matter what starting value is chosen, the sequence always seems to end with the values 1, 4, 2, 1, 4, 2, 1, ... However, it is not known if this is always the case.

4a. Write a procedure `num-seq` that, given a positive integer `n` as argument, returns the stream of values produced for `n` by the algorithm just given. For example, `(num-seq 7)` should return the stream representing the sequence 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ...

4b. Write a procedure `seq-length` that, given a stream produced by `num-seq`, returns the number of values that occur in the sequence up to and including the first 1. For example, `(seq-length (num-seq 7))` should return 17. You should assume that there is a 1 somewhere in the sequence.

CS61A – Lab Assignment 6.2	**Kurt Meinz**
University of California, Berkeley	Summer 2002

1. List all the procedures in the metacircular evaluator that call `eval`.
2. List all the procedures in the metacircular evaluator that call `apply`.
3. Explain why `make-procedure` does *not* call `eval`.
4. Abelson and Sussman, exercises 4.1, 4.2, 4.4, 4.5

CS61A – Lab Assignment 7.1 — **Kurt Meinz**
University of California, Berkeley — Summer 2002

Part A: Abelson and Sussman, exercises 4.27 and 4.29.

Part B: In this lab exercise you will become familiar with the Logo programming language, for which you'll be writing an interpreter in project 4.

To begin, type `logo` at the Unix shell prompt — **not** from Scheme! You should see something like this:

```
Welcome to Berkeley Logo version 3.4
?
```

The question mark is the Logo prompt, like the > in Scheme. (Later, in some of the examples below, you'll see a > prompt from Logo, while in the middle of defining a procedure)

1. Type each of the following instruction lines and note the results. (A few of them will give error messages.) If you can't make sense of a result, ask for help.

```
print 2 + 3

print 2+3

print sum 2 3

print (sum 2 3 4 5)

print sum 2 3 4 5

2+3

print "yesterday

print "julia"

print revolution

print [blue jay way]

show [eight days a week]

show first [golden slumbers]

print first bf [she loves you]

pr first first bf [yellow submarine]

to second :stuff
output first bf :stuff
end

second "something

print second "piggies

pr second [another girl]

pr first second [carry that weight]
pr second second [i dig a pony]

to pr2nd :thing
print first bf :thing
end

pr2nd [the 1 after 909]

print first pr2nd [hey jude]

repeat 5 [print [this boy]]

if 3 = 1+1 [print [the fool on the hill]]

print ifelse 2=1+1  ~
      [second [your mother should know]] ~
      [first "help]

print ifelse 3=1+2 ~
      [strawberry fields forever] ~
      [penny lane]

print ifelse 4=1+2 ~
      ["flying] ~
      [[all you need is love]]
```

Continued on next page...

Lab Assignment 7.1 continued...

```
to greet :person
say [how are you,]
end

to say :saying
print sentence :saying :person
end

greet "ringo

show map "first [paperback writer]

show map [word first ? last ?] ~
         [lucy in the sky with diamonds]

to who :sent
foreach [pete roger john keith] "describe
end

to describe :person
print se :person :sent
end

who [sells out]

print :bass

make "bass "paul

print :bass

print bass

to bass
output [johnny cymbal]
end

print bass

print :bass

print "bass
```

```
to countdown :num
if :num=0 [print "blastoff stop]
print :num
countdown :num-1
end

countdown 5

to downup :word
print :word
if emptyp bl :word [stop]
downup bl :word
print :word
end

downup "rain

;;;; The following stuff will work
;;;; only on an X workstation:

cs

repeat 4 [forward 100 rt 90]

cs

repeat 10 [repeat 5 [fd 150 rt 144] rt 36]

cs repeat 36 [repeat 4 [fd 100 rt 90]
              setpc remainder pencolor+1 8
              rt 10]

to tree :size
if :size < 3 [stop]
fd :size/2
lt 30 tree :size*3/4 rt 30
fd :size/3
rt 45 tree :size*2/3 lt 45
fd :size/6
bk :size
end

cs pu bk 100 pd ht tree 100
```

2. Devise an example that demonstrates that Logo uses dynamic scope rather than lexical scope. Your example should involve the use of a variable that would have a different value if Logo used lexical scope. Test your code with Berkeley Logo.

3. Explain the differences and similarities among the Logo operators `"` (double-quote), `[ ]` (square brackets), and `:` (colon).

CS61A – Lab Assignment 7.2	**Kurt Meinz**
University of California, Berkeley	Summer 2002

1. Abelson and Sussman, exercises 4.35 and 4.38.

2. In this exercise we learn what a *continuation* is. Suppose we have the following definition:

```
(define (square x cont)
  (cont (* x x))
```

Here `x` is the number we want to square, and `cont` is the procedure to which we want to pass the result. Now try these experiments:

```
> (square 5 (lambda (x) x))

> (square 5 (lambda (x) (+ x 2)))

> (square 5 (lambda (x) (square x (lambda (x) x))))

> (square 5 display)

> (define foo 3)
> (square 5 (lambda (x) (set! foo x)))
> foo
```

Don't just type them in – make sure you understand why they work! The nondeterministic evaluator works by evalutating every expression with *two* continuations, one used if the computation succeeds, and one used if it fails.

```
(define (reciprocal x yes no)
  (if (= x 0)
      (no x)
      (yes (/ 1 x))))

> (reciprocal 3 (lambda (x) x) (lambda (x) (se x '(cannot reciprocate))))

> (reciprocal 0 (lambda (x) x) (lambda (x) (se x '(cannot reciprocate))))
```

CS61A – Lab Assignment 8.1	**Kurt Meinz**
University of California, Berkeley	Summer 2002

Abelson and Sussman, exercises 4.55 and 4.62:

4.55: Give simple queries that retrieve the following information from the data base:

All people supervised by Ben Bitdiddle;

The names and jobs of all people in the accounting division;

The names and addresses of all people who live in Slumerville.

4.62: Define rules to implement the `last-pair` operation of exercise 2.17, which returns a list containing the last element of a nonempty list. Check your rules on queries such as

```
(last-pair (3) ?x)
(last-pair (1 2 3) ?x)
(last-pair (2 ?x) (3))
```

Do your rules work correctly on queries such as `(last-pair ?x (3))`?

For the lab exercises and the homework problems that involve writing queries or rules, test your solutions using the query system. To run the query system and load in the sample data:

```
scm
(load "~cs61a/lib/query.scm")
(initialize-data-base microshaft-data-base)
(query-driver-loop)
```

You're now in the query system's interpreter. To add an assertion:

```
(assert! (foo bar))
```

To add a rule:

```
(assert! (rule (foo) (bar)))
```

Anything else is a query.