University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Summer 2002 Instructor: Kurt Meinz 2002-08-16

CS 61A FINAL

Personal Information

*First and Last **Name***	
*Your **Login***	cs61a-__ __ *(legibly!)*
*The **First Letter** of Your Login (please circle)*	a b c d e f g h i j k l m n o p q r s t u v w x y z
*The **Second Letter** of your login (please circle)*	a b c d e f g h i j k l m n o p q r s t u v w x y z
*Lab Section Time, TA, & Location **You Attend***	
*Discussion Section Time, TA, & Location **You Attend***	
"All the work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61a who may not have taken it yet."	***(please sign)***

Instructions

- Partial credit may be given for incomplete / wrong answers, so please write down as much of the solution as you can.
- Feel free to use any Scheme function that was described in lecture or sections of the textbook we have read without defining it yourself. Unless specifically prohibited, you are allowed to use helper functions on any problem.
- Please use "true" instead of #t , and "false" instead of #f. We have found that handwritten #t and #f look too much alike.
- Please write legibly! If we can't read it, we won't grade it!
- **Please comment your code extensively!**

Grading Results

Question	*Max. Points*	*Points Earned*
1	**10**	
2	**12**	
3	**12**	
4	**12**	
5	**12**	
6	**12**	
Total	**70**	

Name: _____ Login: _____

Question 1: Quickies

Part A:

a. What would be the result (return value) of typing '(define (foo a b c))' into the vanilla MCE? If it would be an error, please describe it.

_____

b. How about if you typed the same into the analyzing evaluator?

_____

Part B:

Imagine that you type the following into the logo evaluator:

```
make "x 1
make "y 2

to square :x
output :x * :x
end

to erwinize :m :n
output square :x + square :y + :m + :n
end

to janeinate :x :y
output erwinize :x :y
end
```

What would be the result of evaluating 'erwinize 2 3'?

If the evaluator uses dynamic scoping?	If the evaluator uses lexical scoping?

What about evaluating 'janeinate 2 3'?

If the evaluator uses dynamic scoping?	If the evaluator uses lexical scoping?

Part C: Are Kurt's shirts ugly or pretty? Circle one only.

Name: _____ **Login:**_____

Question 2: MCE

Although we have only covered one version of 'let' in this class, standard scheme provides several others: 'letrec', 'let*', and 'named let'. In this question, we would like you to modify the regular MCE to include support for named lets.

Everyone knows that the regular, vanilla let is just shorthand for a procedure definition and immediate invocation. Normally, this temporary procedure does not have a name associated with it. However, in named lets, a binding of the name of the let to the temporary procedure is created in the same frame as the other (usual) let bindings. This name-procedure binding can be used to implement recursion as follows:

```
;;; M-Eval input:
(named-let fib ((a 4)) ;; 'fib' is the name to bind the proc to.
  (if (< a 2)          ;; '4' is the initial value for the argument.
      1
      (+ (fib (- a 1)) (fib (- a 2)))))

;;; M-Eval output: ;; fib of 4 calls fib with a=3 and a=2, etc.
5
```

The general form of a named let is
'(named-let <name> <bindings> <body>)'

Part A: There are several different ways of constructing the environments for named lets – we don't care which one you choose as long as it behaves correctly. To show us which way you intend to do it, please use the space below to construct an environment diagram to simulate the above named-let expression. Note: you don't have to go through the recursion – just show us the environment as it exists right before going into the body of the let.

Name: _____ **Login:**_____

Question 2: MCE Continued

Part B: Now please define the selectors that you will use to extract the pieces of named-let expressions:

(define (nl-name expr) _____)

(define (nl-bindings expr) _____)

(define (nl-body expr) _____)

Part C: Lastly, please provide a procedure named 'eval-named-let' that takes a named-let expression and an environment and evaluates that expression in that environment. You may assume that we will modify 'mc-eval' itself to test for named-let expressions and pass them off to your procedure.

Your procedure should do all of the necessary work of popping out new frames, adding bindings, etc. In particular, **do not** implement named lets as a syntactic translation to regular lets and definitions!

Please comment your code (so we can give you partial credit)**!**

(define (eval-named-let expr env)

Name: _____ **Login:**_____

Question 3: Lazy Streams

Someday, somewhere, you will be overtaken with the desire to transform an infinite stream of infinite streams into a single stream. Well, maybe not, but is it best to be prepared.

An infinite stream of streams might be something like this:

(cons-stream ones (cons-stream twos (con-stream threes …

In general, this is not the easiest problem in the world. However, it is made much easier under the assumption that you are operating with a normal-order evaluator.

Part A: Assuming that you are using a **normal order evaluator**, please fill in the blanks to finish a procedure named 'flatten' that takes an infinite stream of infinite streams and returns a single infinite stream of all of the elements of each substream. **The order of elements doesn't really matter as long as each element will eventually appear in the stream.** In particular, your flatten procedure does not have to be 'fair' about how it orders the elements from streams.

```
(define (flatten streamostreams)

     (interleave _____________________________________________

                 _____________________________________________))
```

Part B: In 15 words or fewer, please state how the assumption of normal order evaluation made this procedure simpler. In other words, what could you not have done if you were using applicative order?

Name: _____ **Login:**_____

Question 4: Amb From Above

We claim that 'sqrt' can be written using only the non-deterministic language constructs (amb and require) and some of the following arithmetic primitives: +, -, *, =, <, >, abs.

Please define a procedure 'amb-sqrt' that takes in a number (to take the square root of) and a tolerance. The idea is that your sqrt procedure should return a number that is within the real square root plus/minus the tolerance.

For example:
(amb-sqrt 9 0.1) should return a number between 2.9 and 3.0.

The only restrictions are that you must use amb and you are not allowed to use any mathematical functions other than those listed above. You may use helpers if necessary. You should aim to make your algorithm as simple and elegant as possible – **we are not interested in efficiency.** This can be done in 3 lines.

You may make reference to this procedure:

```
(define (a-number-from x step)
   (amb x (a-number-from (+ x step) step)))
```

Please comment your code.

```
(define (amb-sqrt val tol)
```

Name: _____ **Login:**_____

Question 5: AMB From Below

Kurt loves adding special forms to evaluators – even when they are unnecessary or redundant. Please indulge him by adding 'require' as a special form to one of the non-deterministic evaluators.

You may assume that we have modified the evaluator to check for the require special form, and, if found, to call a procedure named 'ambeval-require'. The return value of a successful require should be the literal 'ok'.

Part A: Which evaluator will you use? The one in the book, or the one used in lecture? (We prefer that you use the one from lecture, but feel free to use whichever one you are most comfortable with.) Please circle one:

The Book's The one in Lecture

Part B: Indicate all of the changes necessary to implement require as a regular, non-derived special form. [Don't forget that the test clause may contain an amb itself!]

Please comment your code!

```
(define (ambeval-require exp env succeed fail)
```

Name: _____ **Login:**_____

Question 6: Logic Programming

We're going to build on the representation of number in the logic language by adding a few new 'procedures'. In summary, numbers can be represented as unary lists as follows:
0 = () 1 = (1) 2 = (1 1) 3 = (1 1 1) 4 = (1 1 1 1) etc.

Part A: Write rules for the relation named 'count' that, in essence, can deduce the length of a given object. The first coordinate to count can be any list (including the empty list) and the second coordinate should be the numeral (as defined above) that corresponds to the length of the first object.

For example,
(count (jeff erwin greg jane) ?x) should find ?x = (1 1 1 1)

Part B: Now, write rules for counting the number of times an element exists in a given list.

For example,
(count-n jane (jane jane jane) ?x) will find ?x = (1 1 1)
(count-n ilya (ilya greg ilya) ?x) will find ?x = (1 1)

You may find the following rule helpful:

```
(assert! (rule (equal ?x ?x)))
```