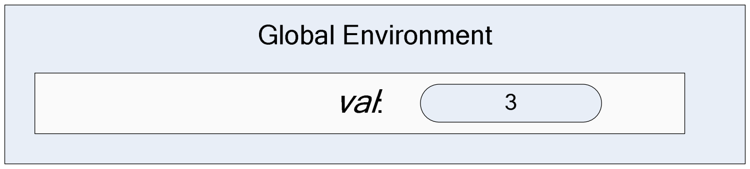
cs1120: Exam 2 Comments

State and Mutation

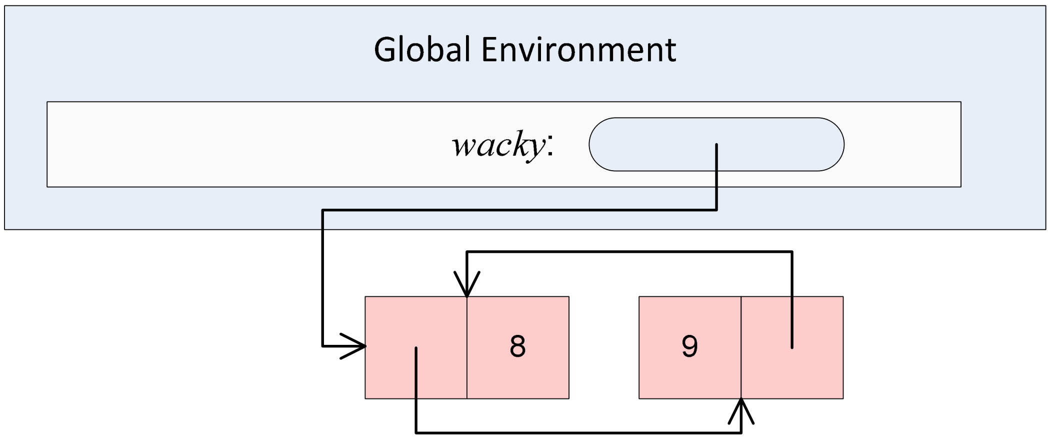
For questions 1-3, each picture shows an environment resulting from evaluating one or more Scheme definitions and expressions. For each part, answer by providing Scheme code that produces the environment shown, starting from the initial global environment. There are many possible correct answers, but you should aim to make your code as simple as possible.

1. (average score: 9.7/10)



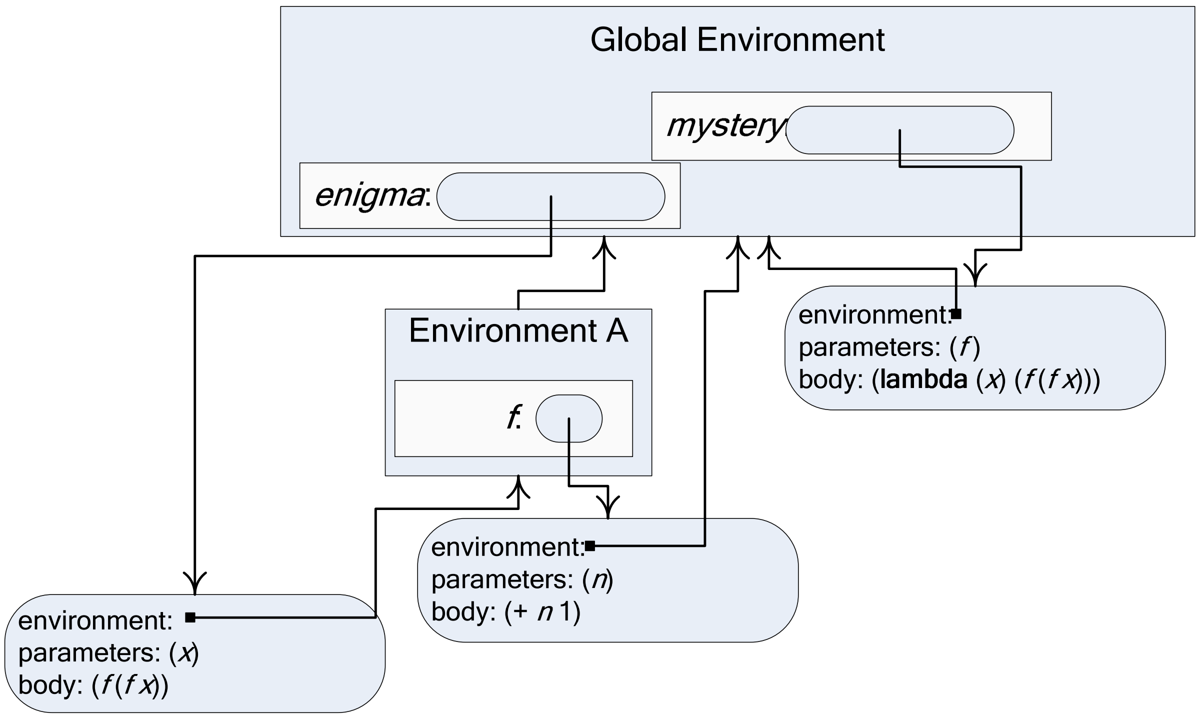
```
(define val 3)
```

2. (5.5)



```
(define wacky (mcons X (mcons 9 Z))
(set-mcdr! (mcdr wacky) wacky)
(set-mcar! wacky (mcdr wacky))
(set-mcdr! wacky 8)
```

3. (7.1)



```
(define (mystery f)
   (lambda (x) (f (f x))))
(define enigma (mystery (lambda (n) (+ n 1))))
```

Space and Time Analysis

Consider the list-merge and mlist-merge! Scheme procedures below:

```
(define (list-merge p1 p2)
    (if (null? p1) null
        (cons (car p1)
               (cons (car p2)
                      (list-merge (cdr p1) (cdr p2))))))

(define (mlist-merge! m1 m2)
  (if (null? m1) (void)
     (let ((m1next (mcdr m1))
           (m2next (mcdr m2)))
       (set-mcdr! m2 (mcdr m1))
       (set-mcdr! m1 m2)
       (mlist-merge! m1next m2next))))
```

For list-merge, assume both inputs are immutable lists of length *N*.
For mlist-merge!, assume both inputs are mutable lists of length *N*.

4a. (9.4/10) What is the asymptotic *running time* of **list-merge**?

The asymptotic running time is in Θ(*N*). There are N recursive calls, and each call involves applications of only constant-time procedures.

4b. What is the asymptotic *running time* of **mlist-merge!**?

The asymptotic running time is in Θ(*N*). There are N recursive calls, and each call involves applications of only constant-time procedures.

5a. (8.8/10) What is the asymptotic *memory use* of **list-merge**?

The asymptotic memory use is in Θ(*N*). For each element in the original input list, two new cons cells are created. So, the memory use is linear in the number of elements in the input list.

5b. What is the asymptotic *memory use* of **mlist-merge!**?

Constant. There are no new cons cells created. (The only memory use is for the stack, but since it is tail recursive this is constant space.)

6. (8.0) Consider this Python procedure (from Class 27):

```
def histogram(text):
   d = {}
   words = text.split()
   for w in words:
      if w in d:
         d[w] = d[w] + 1
      else:
         d[w] = 1
   return d
```

What is the asymptotic running time for histogram? State carefully all your assumptions and define all the variables you use.

Use *N* to represent the length on the input text.
The asymptotic running time is in Θ(*N*). The text.split() procedures splits the text into words separated by spaces. At worst, the number of words is N/2, so the size of words is in Θ(*N*). The loop iterates through the words. For each iteration, we do up to 3 dictionary lookups, but these are constant time, so the total work for each iteration is constant time. Hence, the running time is in Θ(*N*). This assumes the running time for dictionary operations is constant, and text.split() has running time in O(*N*).

Objects and Inheritance

Cordelia Lear does not believe in inheritance. She has defined the following three Python classes:

```
class Spider:
   def number_of_legs(self):  return 8
   def weave(self):  print "Weaving a web."

class SalticidaeSpider:
   def number_of_legs(self):  return 8
   def weave(self):  print "Weaving a web."
   def jump(self): print "Jumping!"

class TarantulaSpider:
   def number_of_legs(self): return 8
   def weave(self):  print "Weaving a web."
   def bite(self): print "Biting!"
```

7. (9.6) Convince Cordelia that inheritance can be useful by defining the Spider, SalticidaeSpider, and TarantulaSpider classes with the exact same behavior as her classes, but with as few lines of code as possible.

```
class Spider:
   def number_of_legs(self):  return 8
   def weave(self):  print "Weaving a web."

class SalticidaeSpider(Spider):
   def jump(self): print "Jumping!"

class TarantulaSpider(Spider):
   def bite(self): print "Biting!"
```

Interpreters

The Charme interpreter from PS7/Chapter 11 does not define the **set!** special form, indeed it provides no mutation operators. Suppose we want extend Charme to include the set! special form, as defined by the Scheme evaluation rule for assignment:

> **Evaluation Rule 7: Assignment.** To evaluate an assignment,
>
> **(set!** *Name Expression***)**
>
> evaluate the Expression, and replace the value associated with the name with the value of the expression. An assignment has no value.

You may assume this **updateVariable(self, name, value)** method has been added to the **Environment** class:

```
def updateVariable(self, name, value):
  if self._frame.has_key(name):
    self._frame[name] = value
  elif (self._parent):
    self._parent.updateVariable(name, value)
  else:
    evalError('Undefined name: %s' % (name))
```

8. (7.5) Define an **evalAssignment(expr, env)** procedure. It should take as inputs a parsed Charme expression, which you may assume is an assignment expression, and the evaluation environment.

> Note: you may use Python and the ps7 code to test your procedure. But, it is not necessary or encouraged to do so. You will not lose points on this question for unimportant syntactic mistakes (e.g., missing closing parentheses).

```
def evalAssignment(expr, env):
  env.updateVariable(expr[1], meval(expr[2], env))
```

(Many people also added error checking code to this, which is definitely a good thing to do in practice, but wasn't necessary for full credit.)

9. Suppose the assignment expression is also added to MemoCharme (the language you have at the end of Problem Set 7 where the results of procedure applications are memoized). Illustrate the problems this might cause by showing a sequence of Charme expressions that will evaluate to different values in the versions of Charme with and without memoization.

The key is that now we have assignment the value of a previously memoized application can change. For example,

```
MemoCharme> (define x 3)
MemoCharme> (define get-x (lambda () x))
MemoCharme> (get-x)
3
MemoCharme> (set! x 4)
MemoCharme> (get-x)
3                      should be 4, but wrong result memoized
```

Sally Snake has grown fond of the iteration constructs provided by Python, and wants you to extend Charme to provide a while expression similar to the while statement in Python. Its evaluation rule is:

Evaluation Rule: While. To evaluate a while expression,

(while *Expression*$_{\text{pred}}$ *Expression*$_{\text{body}}$ *Expression*$_{\text{final}}$**)**

evaluate the predicate expression (*Expression*$_{\text{pred}}$). If it evaluates to a false value, the value of the while expression is the value of the final expression (*Expression*$_{\text{final}}$). Otherwise, evaluate the body expression (*Expression*$_{\text{body}}$).

For example, here is a Charme program using while (this also assumes the assignment expression from question 7):

```
Charme> (define loopy (lambda (n) (while (< n 10) (set! n (* n 2)) n)))
Charme> (loopy 3)
12
```

10. (7.2) Define an evalWhile(expr, env) procedure that implements the evaluation rule for a while expression. It should take as inputs a parsed Charme expression, which you may assume is an assignment expression, and the evaluation environment.

```
def evalWhile(expr, env):
   while meval(expr[1], env) != False:
      meval(expr[2], env)
   return meval(expr[3], env)
```

Another approach is very similar to evalIf:

```
def evalWhile(expr, env):
   if meval(expr[1], env) != False:
      meval(expr[2], env)
      return meval(expr, env) # note: the whole while, not just expr[2]
   else
      return meval(expr[3], env)
```