CS61A Fall 1999

Midterm #2

Problem #1 (3 Points)

What will Scheme print in response to the following expressions? If an expression produces an error message, you may just say "error"; you don't have to provide the exact text of the message. If the value of an expression is a procedure, just say "procedure"; you don't have to show the form in which Scheme prints procedures. **Also, draw a box and pointer diagram, of the value produced by each expression**.

- (cons (cons 3 4) (list 3 4))
- (append '((3 4)) '(5 6))
- (list 3 (list 4 5))

Problem #2 (4 Points)

Consider this object class:

```
(define-class (echoer)

          (instance-vars (last 'first-time))

          (method (echo it)

                    (let ((result last))

                              (set! last it)

                              result)))
```

(a) Demonstrate what this class does by filling in the blanks in the following interaction:

> (define e (instantiate echoer))

> (ask e 'echo 'foo)

____

> (ask e 'echo 'baz)

____

(b) Create a new class called **rewarder** with an instantiation variable **secret-word**, with **echoer** as its parent class. A rewarder should handle **echo** messages just like an echoer, except that if the argument to the message is the secret word, it should instead return the sentence **(You win $50)**. Your solution must not copy the body of the echo method from the echoer class, but must instead rely on inheritance from its parent class.

Problem #3 (4 points)

In this question we want a simulation of automobiles (class **auto**) and junkyards (class **junkyard**).

(a) An auto is either wrecked or not wrecked. When instantiated, the auto is not wrecked. Write a **wreck** method that tells the car to become wrecked, and a **wrecked?** method that returns **#T** if the car is wrecked, or **#F** if not. (Once a car is wrecked, it stays wrecked forever.) An auto also should have an instantiation variable called **description**; when an auto is instantiated, it will be given a description such as **(1990 Toyota)**.

(b) Here is a partial definition of the junkyard class:

```
(define-class (junkyard)

(instance-vars (wrecked '()) (to-be-wrecked '()))

(method (add-auto auto)

        (if (ask auto 'wrecked?)

                (set! wrecked (cons auto wrecked))

                (set! to-be-wrecked (cons auto to-be-wrecked))))

...)
```

Old autos, wrecked or otherwise, may be sent to a junkyard. Autos that are already wrecked are in one pile; autos that aren't wrecked go in anther pile, from which they will be sent through a wrecking machine so they take up less space. We have provided the **add-auto** method to put an auto in the right pile.

Add a **wreck** method to the junkyard class. When invoked, this method should wreck all the autos in the to-be-wrecked pile and return a list of their descriptions.

Example:

```
> (define c1 (instantiate auto '(99 Pinto)))

> (define c2 (instantiate auto '(56 Chevy)))

> (define jy (instantiate junkyard))

> (ask jy 'add-car c1)

> (ask c1 'wrecked?)

#f

> (ask jy 'add-car c2)

> (ask jy 'wreck)

((56 Chevy) (99 Pinto))

> (ask c1 'wrecked)

#t

> (ask jy 'wrecked)

()

> (define c3 (instantiate auto '(99 Mustang)))

> (ask jy 'add-car c3)

> (ask jy 'wreck)

((99 Mustang))
```

Problem #4 (4 Points)

We're going to use data-direct programming to handle English word forms such as plurals of nouns. Most nouns form their plural by adding the letter s, but there are many exceptions. In the first week of class we wrote this procedure to handle some of the cases:

```
(define (plural wd)

          (if (and (equal? (last wd) 'y) (not (vowel? (last (bl wd)))))

                    (word (bl wd) 'ies)

                    (word wd 's)))
```

The form of the plural depends on one or two letters at the end of the word, except for really exceptional cases based on the entire word, such as man and men. We encode all these cases in the operation table:

```
(put 'plural "" 's)

(put 'plural 'y 'ies)

(put 'plural 'ay 'ays)

(put 'plural 'ey 'eys)

(put 'plural 'iy 'iys)

(put 'plural 'oy 'oys)

(put 'plural 'uy 'uys)

(put 'plural 'man 'men)

(put 'plural 's 'ses)

(put 'plural 'alumnus 'alumni)
```

The table entry for each set of ending letters specifies a new set of ending letters to replace them in order to form the plural. The largest suffix found in the table determines the result. For examples, to find the plural of sky, first we look for sky in the table and don't find it. Then we look for ky and don't find that either. Then we look for y and find the word ies. So we combine the unused letters sk with the table value ies to get the plural skies. To find the plural of boy, first we look for boy in the table and don't find it. Then we look for oy in the table and do find it, with the value oys. So we combine the unused letter b with the table value oys to find the plural boys.

The first put expression above established what to do for ordinary words, such as dog. We look for dog, og, and g without finding any of them, and then finally we look for the empty word and find the value s. So we combine the unused letters dog with the table value s to make the plural dogs. There will always be an entry for the empty word, so there will always be some entry matching any possible word.

The same strategy can be used for other word forms, such as the past tense of verbs:

```
(put 'past "" 'ed)

(put 'past 'e 'd)

(put 'past 'have 'had)

...
```

so we want a general function that can handle any word operation. (Yes, we know this isn't really general enough; it'll give wrong answer for other verbs that happen to end in have, such shaved. Ignore that problem.)

We define the function plural this way:

```
(define (plural wd)

          (word-operate 'plural wd))
```

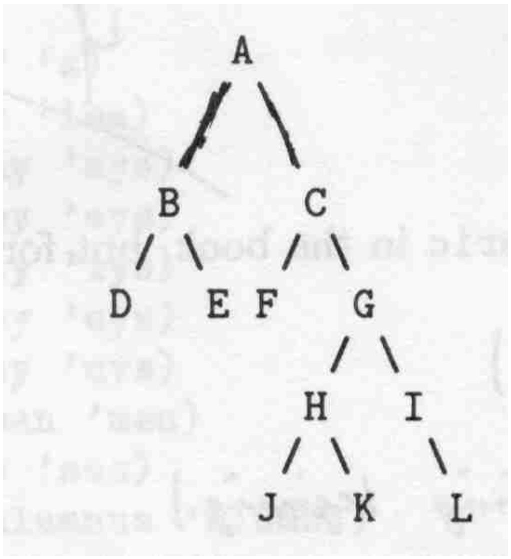
Write word-operate, analogous to apply-generic in the book, but for functions of just one argument, to implement this technique.

Problem #5 (4 points)

This question concerns binary trees, as implemented in SICP pages 156-157

```
(define (entry tree) (car tree))

(define (left-branch tree) (cadr tree))

(define (right-branch tree) (caddr tree))

(define (make-tree entry left right)

        (list entry left right))
```

Write a procedure tree-ref analogous to the list-ref procedure for sequences. Tree-ref takes two arguments a binary tree and a list whose elements are all either L or R. It returns sub tree of the argument tree, chosen by following the left branch for each L or the right branch for each R. You may assume that these branches exist. For example, if t is the tree



then (tree-ref t '(R R L)) should return the subtree whose root node has the entry H; starting from the root node of t it follows the right branch, then the right branch, then the left branch.

Respect the data abstraction.

please contact mailto:examfile@hkn.eecs.berkeley.edu