**Sample midterm 2 #1**

**Problem 1 (What will Scheme print?)**

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

```
(map caddr '((2 3 5) (7 11 13) (17 19)))
```

```
(list (cons 2 (cons 3 5)))
```

```
(append (list '(2) '(3)) (cons '(4) '(5)))
```

```
(list (cons '(0) '(1)) (append '(2) '(3)))
```

**Problem 2 (Tree recursion)**

(a) Using the binary tree abstract data type as defined on page 115 of the text (with selectors `entry`, `left-branch`, and `right-branch` and constructor `make-tree`), write the predicate `all-smaller?` that takes two arguments, a binary tree of numbers and a single number, and returns `#t` if every number in the tree is smaller than the second argument. Examples:

```
> (define my-tree (make-tree 8 (make-tree 5 '() '())
                               (make-tree 12 '() '())))
> (all-smaller? my-tree 15)
#T
> (all-smaller? my-tree 10)
#F
```

(b) Using `all-smaller?` and, if you wish, a similar `all-larger?` (which you don't have to write), write a predicate `bst?` that takes a binary tree of numbers as its argument, returning `#t` if and only if the tree is a binary *search* tree. (That is, your procedure should return true only if, at every node, all of the numbers in that node's left branch are smaller than the entry at the node, and all of the numbers in the node's right branch are larger than the entry.)

**Problem 3 (Tree recursion)**

This question concerns the Trees with constructor `make-tree` and selectors `datum` and `children` as discussed in lecture.

Every node of a Tree has some number of children, possibly zero. We'll call that number the *fanout* of the node. (We are talking about the node's own children, not its grandchildren or more remote descendants.) For a given Tree, there is some node with a fanout larger or equal to the fanout of any other node. Write the procedure `max-fanout` that takes a Tree as its argument, and returns the largest fanout of any node in the Tree.

**Problem 4 (Data-directed programming)**

You are implementing a calculator for physicists, in which arithmetic is performed on numbers with units attached, e.g., 3 dynes times 4 centimeters equals 12 ergs. There are two kinds of operations relevant to the project. (You are only required to implement one of these for the exam!) Two numbers with units can be multiplied, as above, if there is an appropriate unit for the answer. Two numbers with units can be *added* if their units are identical or if one is a multiple of the other.

To make this work, you are using `attach-tag` to attach a unit to a number. You plan to use data-directed programming, with entries like

```
(put 'dyne 'cm 'erg)
```

to tell the program about the conversion mentioned above. You also have conversions for units of the same kind:

```
(put 'ft 'in 12)
```

This indicates that a foot equals 12 inches.

Write the procedure `(plus x y)` that adds two typed quantities. If the two arguments are of the same type, just add the contents and preserve the type. If the two arguments are of different types, look them up with `get`. (Don't forget that the two arguments may not be in the same order as the types in the table entry. That is,

```
(plus (attach-tag 'ft 2) (attach-tag 'in 6))
(plus (attach-tag 'in 6) (attach-tag 'ft 2))
```

should both work with the foot-to-inch table entry above.) If you find a number, do the appropriate conversion and give a result like

```
(attach-tag 'in 30)
```

for the problem above. If you find another unit, as in the erg example, or you find no entry at all, then give the error message "you can't add apples and oranges."

**Problem 5 (Object-Oriented Programming)**

```
(define-class (scoop flavor)
  ; maybe (parent (cone)) -- see part (A) below
  )

(define-class (vanilla)
  (parent (scoop 'vanilla)))
(define-class (chocolate)
  (parent (scoop 'chocolate))

(define-class (cone)
  ; maybe (parent (scoop)) -- see part (A) below
  (instance-vars (scoops '()))
  (method (add-scoop new)
    (set! scoops (cons new scoops)))
  (method (flavors)
    (map ____see (B) below____ scoops)))
```

(A) Which of the `parent` clauses shown above should be used?

_____The `scoop` class should have `(parent (cone))`.

_____The `cone` class should have `(parent (scoop))`.

_____Both.

_____Neither.

(B) What is the missing expression in the `flavors` method?

(C) Which of the following is the correct way to add a scoop of vanilla ice cream to a cone named `my-cone`?

_____`(ask my-cone 'add-scoop 'vanilla)`

_____`(ask my-cone 'add-scoop vanilla)`

_____`(ask my-cone 'add-scoop (instantiate 'vanilla))`

_____`(ask my-cone 'add-scoop (instantiate vanilla))`