CS61A Spring 2001
MT2
Professor Brian Harvey and Professor Dan Garcia

Problem #1

Consider the following definitions:

```
> (define c (cons '() '(lets (go) bears)))
> (define a (append '() '(lets (go) bears)))
> (define 1 (list '() '(lets (go) bears)))
```

(a) 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. If the value of an expression is a procedure, just write "procedure"; you don't have to show the form in which Scheme prints procedures. Also, draw a box and pointer diagram for the value produced by each expression.

```
> c
```

```
> a
```

```
> 1
```

(b) What will Scheme print in response to the following expressions? Do not draw the box and pointer diagrams for them.

```
> (cdr c)
```

```
> (cdr a)
```

```
> (cdr 1)
```

Problem #2

As you already know, lists are made out of pairs. Suppose we want to use an OOP version of pairs to make lists in proper object-oriented style. Louis Reasoner gives the following implementation of OOP pairs:

```
(define-class (pair the-car the-cdr)
  (method (length)
    (if (null? the-cdr)
      1
      (+ 1 (ask the-cdr 'length)))))
```

(a) Happy with his implementation, Louis interacts the following way with his pairs (fill in the blanks):

```
> (define (new-cons a b) (instantiate pair a b))
> (define (new-cdr p) (ask p 'the-cdr))
> (define my-list (new-cons 3 '()))
> (ask my-list 'length)
```

____

```
> (define other-list (new-cdr my-list))
> (ask other-list 'length)
```

____

(b) Reimplement OOP-style lists so that the examples in part (a) return 1 and 0 as they should. You may create or modify classes, methods, and/or procedures as needed. You must use OOP style.

Problem #3

(a) Complete the definition below for the lock class. When a lock is instantiated, it must be given a PIN. You can close a lock at any time, but you may only open it if you have the secret PIN (matching the one given when it was created). You cannot open or close a lock twice in a row. (An attempt to open an already-open lock with the wrong PIN should complain about the PIN rather than about the openness.) Each lock is initially closed.

```
> (define my-lock (instantiate lock '1234))
> (ask my-lock 'open 1234)
(lock opens)
> (ask my-lock 'open 1234)
(lock is open already!)
> (ask my-lock 'open 3424)
(sorry wrong pin)
> (ask my-lock 'close)
```

```
(lock closes)
> (ask my-lock 'close)
(lock is closed already!)

(define-class (                          )

  (method (open pin)

    (cond (

        '(sorry wrong pin))

      (

        '(lock is open already!))

      (else

        '(lock opens))))

  (method (close)

    (cond (

        '(lock is closed already!))

      (else

        '(lock closes)))))
```

(b) Define a class smart-lock which has the same features as lock with one small exception: If you try to open a smart-lock three times with a wrong PIN (the three trials need not be consecutive), it will disallow any future open attempts, even if you eventually pass in the correct PIN:

Use the proper object-oriented programming style.

```
> (define my-smart-lock (instantiate smart-lock '1234))
okay
> (ask my-smart-lock 'open 3423) ; first wrong attempt
(sorry wrong pin)
> (ask my-smart-lock 'open 4532) ; second wrong attempt
(sorry wrong pin)
> (ask my-smart-lock 'open 1234) ; (opening and closing
(lock opens)
> (ask my-smart-lock 'close)          ; between wrong attempts)
(lock closes)
```

```
> (ask my-smart-lock 'open 2923) ; third wrong attempt
(sorry wrong pin)
> (ask my-smart-lock 'open 8623) ; now lock is shut down
(lock shut down)
> (ask my-smart-lock 'open 1234) ; correct PIN but too late
(lock shut down)
> (ask my-smart-lock 'close)          ; closing unaffected
(lock is closed already!)
```

Problem #4

This question is about binary trees, as defined in SICP:

Constructor: (make-tree entry left right) Selectors: (entry tree) (left-branch tree) (right-branch tree)

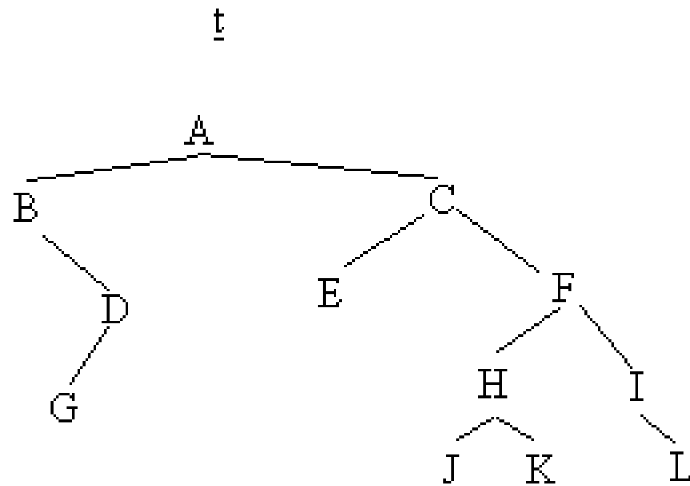
We'll call a binary tree full if all leaves are at the same depth, and there are no missing nodes. Your goal is to take a tree that is not full and fill it.

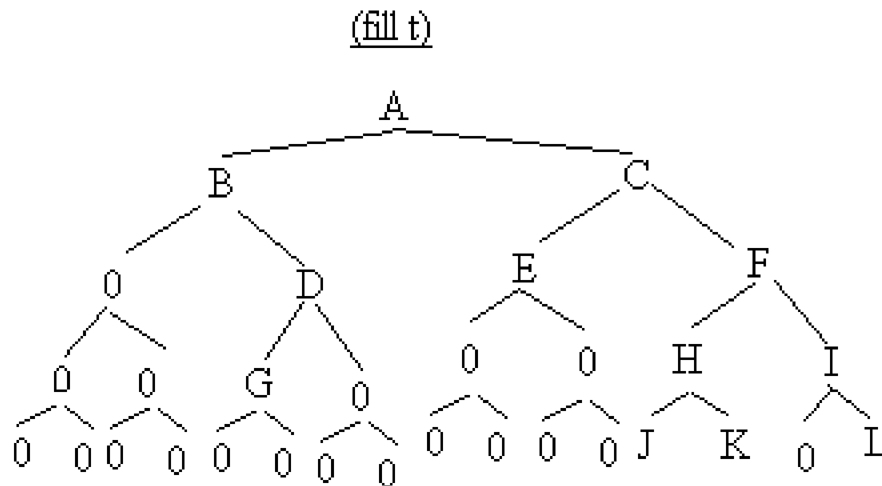
(a) Write make-filler, which takes a nonnegative integer as its argument. If the argument is zero, it should return the empty list. Otherwise, it should return a full binary tree, with zeros in every entry, and with the number of levels specified by the argument. Some examples will help make this clear (showing pictures of trees rather than the representation as printed by Scheme):

```
> (make-filler 1)

0

> (make-filler 2)

  0
 / \
0   0

> (make-filler 3)

   0
  / \
 0   0
/\   /\
0 0 0 0
```

(b) Now we can write fill, which takes in a tree and returns a filled version of that tree.

For example:





Someone has written depth for us; it returns the maximum depth of the tree. (The depth of a tree with only a root node is zero.) We've given you the procedure below as a starter. You write fill-help.

```
(define (fill tree)
  (fill-help tree (depth tree)))
```

Problem #5

We are going to design a data-directed Adventure game program. (Later, in project 3, you'll work on a different object-oriented design for an Adventure game.) Note -- this explanation is a lot longer than the program you have to write! Don't get scared.

In an Adventure game, you move around through a sort of maze with rooms connected by passageways. Here is a typical interaction (the user types the **bold** text):

> **(adventure)**
You are at the entrance to a cave. There is a tunnel heading east.
Your move? **east**
You are in a large cavern. The walls are glowing faintly. There are tunnels to the east and south. There is a wand lying on the ground.
Your move? **wave wand**
There is a puff of smoke. You are now in a small room with golden walls. A flask containing a green potion is on the floor. There are doors to the north and south.

... and so on. The game is data-directed because the program will not have specific information about the rooms; instead, we'll use put to set up the rooms, like this:

```
(put 'entrance 'description "You are at the entrance ... heading east.")
(put 'entrance 'east (lambda () (visit 'cavern)))
(put 'cavern 'description "You are in a large cavern. ... on the ground.")
(put 'cavern 'east (lambda () (visit 'torture-chamber)))
(put 'cavern 'south (lambda () (visit 'lecture-hall)))
(put 'wand 'wave (lambda () (visit 'gold-room)))
```

You are given the following procedures:

```
(define (adventure)
  (visit 'entrance))

(define (visit room)
  (newline)
  (display (get room 'description))
  (newline)
  (display "Your move? ")
  (act room (read-line)))
```

Assume that read-line is a primitive that returns what the user types as a list, e.g., (east) or (wave wand).

Your job is to write the procedure **act** that takes a room and a user request as arguments. If the user's request contains just one word, then it should be directed to the current room. If the request contains two words, then the request should be directed to the thing named by the second word (e.g., wand).

You may assume that the user's request is legitimate. That is, the request will always be either one or

two words long, the user won't ask to move in an impossible direction, and the user won't try to manipulate an object that isn't in the room. Don't put in error checks for these situations.

Posted by HKN (Electrical Engineering and Computer Science Honor Society)
University of California at Berkeley
If you have any questions about these online exams
please contact mailto:examfile@hkn.eecs.berkeley.edu