CS 61A, Fall 1997
Midterm #2
Professor Harvey

Problem #1

This problem concerns the tree abstract data type defined by the constructor make-tree, invoked with (make-tree <datum> <children>), and the selectors datum and children.

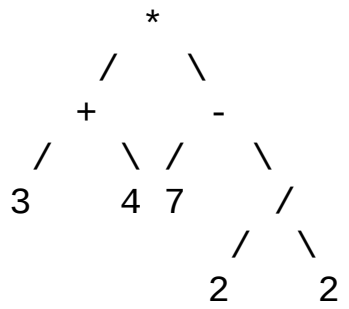
(a) Write the procedure treeify:

-The argument to treeify is a nonempty list representing arithmetic computation in infix notation, such as this example:

((3 + 4) * (7 - (2 / 2)))

More precisely, a computation is either a number or a list of three elements, in which the first and last are computations and the middle element is a symbol representing an operator (such as +).

-The value returned by treeify is a tree in which the branch nodes are operator symbols and the leaf nodes are numbers:



(b) Suppose that the time required for an addition or subtraction is one microsecond, and the time required for a multiplication or a division is five microseconds.

+ - 1
* / 5

Write a time procedure that takes as its argument a computation tree as produced by treeify, and returns the number of microseconds required to compute the desired value.

>(time (treeify '((3 + 4) * (7 - (2 / 2)))

Problem #2

The procedure addup, shown later, takes two arguments: a list of numbers, and a goal number. It returns a list of numbers, a subset of the original list, whose sum is the goal number, or #f if there is no such list. It returns the shortest possible list.

```
> (addup '(2 6 3 4 5) 10)
(4 6)
> (addup '(2 3 4 5) 10)
(5 3 2)
```

Note that (5 3 2) would be a solution in the first example, but addup chooses the two-number solution because it's shorter.

You do not have to write addup or figure out its algorithm. We have written it for you; your job is to modify the program to use data abstraction, as follows:

The program works by maintaining a queue of trials. Each trial is a list of three elements: a list of the numbers being tried, their sum, and a list of the reamining numbers from the argument list that aren't in this trial. For each trial in the queue, the program checks whether the sum is equal to the goal. If so, that's the solution. If not, the program appends to the queue all the trials that can be made from this trial by adding one more number.

Your job is to improve the readability of this program by designing an abstract data type for trials, writing the necessary constructor(s) and selector(s), and rewriting the procedures on the next page to use them correctly.

(a) Write the constructor(s) and selector(s) here:

Here is the code you are modifying:

```
(define (addup nums goal)
        (addup-helper (list (list '() 0 nums)) goal))

(define (addup-helper queue goal)
        (cond ((null? queue) #f)
              ((eq? (cadar queue) goal) (caar queue))
              (else (addup-helper
                     (append (cdr queue)
                             (map (lambda (now)
                                    (list (cons new (caar queue))
```

```
                                    (+ new (cadar queue))
                                    (remove new (caddar queue))))
                            (caddar queue)))
                goal))))
```

(b) Rewrite addup and addup-helper to use your abstract data type:

Problem #3

Computers don't directly understand languages loke Scheme. Instead each computer model has a machine language in which instructions are represented as sequences of numbers. We'll simulate a machine language instruction as as list of numbers, although that isn't the actual format.

Since sequences of numbers aren't very readable, we generally represent instructions in an assembly language that's slightly easier for a human to understand. In the simplified assembly language of this problem, the first number is replaced by an operation name, and the other numbers may be rearranged.

Here are a few typical instructions in both forms:

machine language --> assembly language

(2 8 9 10) ---> (sub 10 8 9)
(5 29 10 4) ---> (store 10 4 (29))
(6 10 8 400) ---> (beq 10 8 400)

The first number in each machine language instruction is the opcode, which represents the operation to be performed. The other numbers are operands.

We are going to write a disassembler - a translator that takes a machine language instruction as its argument and returns the corresponding assembly language instruction. We are going to use data-directed programming. You are given the following instruction in the following instruction formats:

(define register '(0 3 1 2))
(define memory '(0 2 3 (1)))
(define branch '(0 1 2 3))
(define immediate '(0 2 1 3))

In each of these format lists, the number 0 represents the operation name, the other numbers represent the positions of operands in the machine language instruction. For example, the machine instruction (2 8 9 10) is translated by finding out that the opcode 2 has the name sub and is of register type, so its operands should be presented in the order third, first, second.

The program must find out that the opcode 2 means sub and is of register type. You are given the

following table of instruction names and types:

```
(define ops (list (list 'add register)
                  (list 'addi immidiate)
                  (list 'sub register)
                  (list 'subi immidiate)
                  (list 'load memory)
                  (list 'store memory)
                  (list 'beq branch)
                  (list 'bne branch)))
```

In this list, the nth element (counting from zero) represents opcode n.

You are also given a procedure that takes a machine language instruction with the opcode replaced by the operation name as argument, and returns the assembly language form:

```
(define (printer instr format)
        (map (lambda (piece)
               (if (list? piece)
                       (printer instr piece)
                       (list-ref instr piece)))
             format))

> (printer '(store 29 10 4) '(0 2 3 (1)))
(store 10 4 (29))
```

Your job is to write the procedure assembly that takes one argument, a machine language instruction, and returns the assembly language equivalent.

Problem #4

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 actual 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.

(append '(a b) '())

(cons '(a b) '())

(list '(a b) '())

(caadr '((1 2) (3 4)))
