General

You may submit solutions for three of the problems described in this document instead of doing a project. All three are due when the final project is due. You must also have one of them complete and debugged to show a TA in your lab section for the first checkoff and another to demonstrate for the second checkoff.

Grading Guidelines

The 40 points you can earn for these problems constitute 10% of your course grade, as would the project. (Note that the checkoff points for your solutions count toward your project grade, not your lab grade.) **The only thing you lose by not doing the Shall We Play a Game™ project is that your course grade will not be higher than B+.**

The problems are intended to prepare you better for the final exam. Thus time spent working on the problems that you don't turn in solutions for will not be wasted.

There are three groups of problems. You must solve at least one problem from each of the first two groups.

You may use any construct we've covered in *Simply Scheme*; in particular, you may use either higher-order functions or recursion for any of the problems. Efficiency will not be a criterion for grading. Appropriate use of Scheme will be, however; for instance, you should generally use a list with `assoc` rather than a `cond` with a large number of conditions, and provide auxiliary functions rather than coding a single complicated function. You should provide comments for each of your functions that describe the arguments and the result returned, and you should use good names for functions and their placeholders.

Test cases on which you are to test your functions are listed with each problem. These test cases are not guaranteed to reveal all your bugs, and you are encouraged to test your code more comprehensively. You will lose fewer points for a bug you point out to the reader than for a bug that the reader must detect on his or her own.

Group A (do at least one problem from this group)

Problem A1

Write a function called `days-in-debt` to determine how many days a bank account contains a negative amount of money. `Days-in-debt` takes two arguments, an initial balance (a nonnegative number) and a transaction list, and returns the number of days in a 31-day month for which the account balance is less than 0 at the end of the day.

Each element of the transaction list—a *transaction*—is a two-element list whose first element is a day in a month (1, 2, …, 31) and whose second element is an amount (negative for withdrawal, positive for deposit). A transaction list may be empty, and may contain any number of transactions for a given day. Transactions within a transaction list are arranged in order of day.

Here are some examples.

initial balance	*transaction list*	*result to return*
anything greater than or equal to 40	`((1 -10) (1 -10) (15 -10) (30 -10))`	`0`
anything at least 30 and less than 40	`((1 -10) (1 -10) (15 -10) (30 -10))`	`2`
anything at least 20 and less than 30	`((1 -10) (1 -10) (15 -10) (30 -10))`	`17` (the 15th through the 31st)
anything at least 10 and less than 20	`((1 -10) (2 -20) (2 15) (2 5) (13 -10) (15 10) (20 5) (31 5))`	`2` (13th and 14th; note that the 2nd is not counted)
anything at least 5 and less than 10	`((1 -10) (2 -20) (2 15) (2 5) (13 -10) (15 10) (20 5) (31 5))`	`19` (1st through 19th)
anything less than 5	`((1 -10) (2 -20) (2 15) (2 5) (13 -10) (15 10) (20 5) (31 5))`	`30` (1st through 30th)
anything at least 5 and less than 10	`((1 -10) (2 -20) (2 15) (2 5) (13 -10) (15 10) (20 5) (31 -2) (31 -3))`	`20` (1st through 19th, plus the 31st)

You may assume that the arguments will be a legal balance and transaction list as just described. Test `days-in-debt` at least on the examples given above.

Problem A2

Background

Data stored on computers can often be *compressed* to take less space. The compression process detects patterns in the data, and codes those patterns more compactly. (Pack rats like me find this very handy.)

Data representing pictures is especially appropriate for compression. Conceptually, a black-and-white picture is just a grid of black and white "dots." It can be represented by a sequence of 0's and 1's, with a 0 representing each white dot and a 1 representing each black dot. Pictures often contain large regions of a single color, however, and such a region can also be represented just a few values: a number representing the color, and one or two numbers saying how much of that color there is in the region.

This technique can also be applied to lists of 0's and 1's. A sequence of consecutive identical elements in the list can be replaced by two values: the number of elements in the sequence, and a single copy of the repeated element. **Non-repeated elements are untouched**. Here's an example (the fifth element in the uncompressed list was untouched):

uncompressed list	*compressed list*
`(0 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1 1 1)`	`(4 0 1 4 0 9 1)`

Problem

Write a function called `compress` that implements the process just described. The list returned by `compress` will have several properties. Its elements are all nonnegative integers.

- Every occurrence of an integer greater than 1 is immediately followed in the list by a 0 or a 1. For example, `compress` should never return the list `(2 1 2)` or `(3 2 1)`.
- Consecutive 0's or 1's will not occur. A sequence of, say, five consecutive 0's would have been replaced by the values 5 and 0, and a sequence of two consecutive 1's would have been replaced by the values 2 and 1.
- Consecutive groups of 0's or 1's will not occur. For example, the sequence `(4 1 5 1)` represents a sequence of nine 1's that would have been compressed to `(9 1)`.

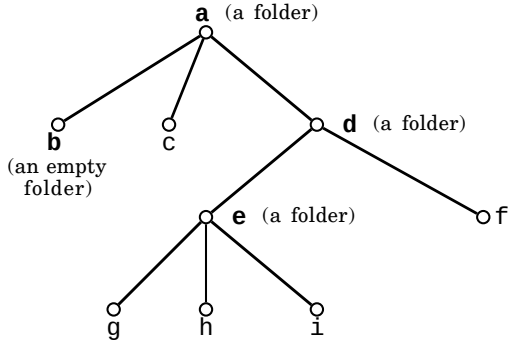
You may assume that the argument to `compress` is a list that contains only 0's and 1's. Test `compress` on at least one list in each of the following categories:

a list starting with a group of 1's;	a list ending with a group of 1's;
a list starting with a group of 0's;	a list ending with a group of 0's;
a list starting with 1 0;	a list ending with 1 0;
a list starting with 0 1;	a list ending with 0 1.

Group B (do at least one problem from this group)

Problem B1

Folders on the Macintosh may contain other folders, which may contain other folders, and so on. The files in the Macintosh file system may thus be represented as a nested list, as pictured in the diagram below. A file that's not a folder is represented as an atom; a folder is represented as a list whose first element is the atom folder name, and whose remaining elements are the files and folders contained in the folder.



Scheme representation:

```
(a
   (b)
   c
   (d (e g h i) f) )
```

One often wishes to locate a file in the Macintosh file system, that is, figure out how to get to it through the chain of folders that contain it. You are to write a function called `path-to`, which, when given arguments representing a file system and a file name, returns a list whose last element is the file name, whose first element is the name of the "root" file or folder, and whose other elements are the names of folders that must be opened to find the file. If no file with the given name is anywhere in the file system, your function should return `#f`. If a file with the given name appears twice or more in the file system, you may return the path to any of the files with that name.

Test your function with the above directory, searching for the following files:

name of file to search for	*list to return*
`x`	`#f`
`b`	`(a b)`
`a`	`(a)`
`g`	`(a d e g)`
`i`	`(a d e i)`
`e`	`(a d e)`

Problem B2

Write a function `arith-eval` that works similar to Gambit's evaluation function when applied to expressions containing only arithmetic operations. `Arith-eval` will take two arguments: one is an expression, written in prefix notation as in Scheme, in which only the operators +, –, and * are used; the other is a table of variable-value associations. Evaluation of a variable in the expression—something that's not a number—should return the value associated with the variable. Note that arithmetic operators may take more than two arguments in Scheme.

Some examples:

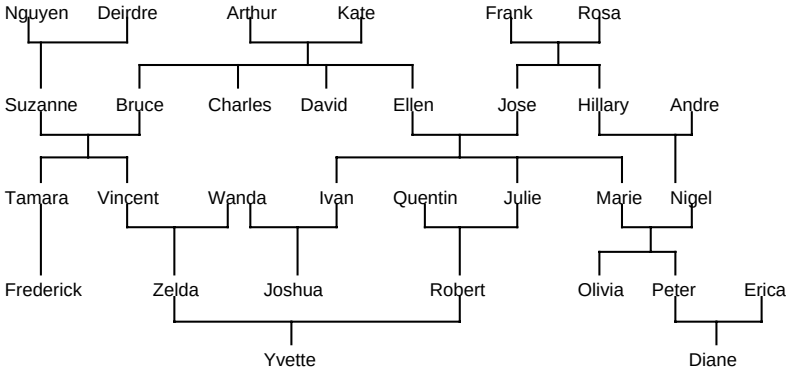
expression	*returned value*
`(arith-eval '(+ x) '((x 2) (y 1) (z 0)))`	`2`
`(arith-eval '(+ x y z) '((x 2) (y 1) (z 0)))`	`3`
`(arith-eval '(+ (* x z) (* 2 y) 13 (- 4 1)) '((x 2) (y 14) (z 3)))`	`50`

You may assume that the expression argument is a legal Scheme expression in which the only function calls are to +, –, and *, and in which all variables used also appear in the table argument. Test your function on the above examples.

Problem B3

Background

Consider a genealogical data base represented as a collection of *nuclear families*. Each nuclear family is a list whose first element is the name of the father (possibly the atom `unknown`, if the father is not known), whose second element is the name of the mother (also possibly the atom `unknown`), and whose remaining elements are the names of their children. Given below is a diagram for a family and the corresponding genealogical data base.




```
((nguyen deirdre suzanne)
 (arthur kate bruce charles david ellen)
 (frank rosa jose hillary)
 (bruce suzanne tamara vincent)
 (jose ellen ivan julie marie)
 (andre hillary nigel)
 (unknown tamara frederick)
 (vincent wanda zelda)
 (ivan wanda joshua)
 (quentin julie robert)
 (nigel marie olivia peter)
 (robert zelda yvette)
 (peter erica diane) )
```

Some interesting features of this family: Marie has married her first cousin Nigel. Wanda has had one child with Vincent and another with Ivan. Zelda and Robert, the parents of Yvette, have two great grandparents in common. And only Tamara knows who Frederick's father is, and she's not telling.

Problem

Write a function `ancestors` that, given a data base and a person in the data base as arguments, returns a list of ancestors of that person. (A person's ancestors are his or her parents plus the ancestors of his or her parents.) You may assume that the data base is a legal one, that the given person is somewhere in the data base, and that no two people in the data base have the same name. However, as in Yvette's case, a person may be an ancestor of another in more than one way; the list returned by `ancestors` should not return duplicate names.

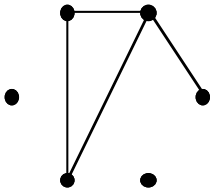
Test your function at least on a person with no ancestors, on Yvette, and on Frederick using the given data base (online in the file family db.scm).

Group C

Problem C1

Background

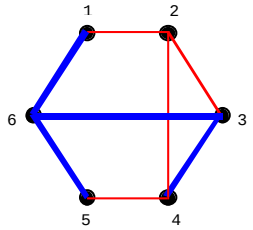
First, some vocabulary: This project involves a *network*, a diagram with *points*, some of which may be connected by *lines*. A sample network with six points, four of which have lines connecting them, appears below.



Consider the following game played on a network of six points. Two players play, one assigned the color red, the other the color blue. Each player takes turns connecting two unconnected points with a line of the player's own color. If a player is forced to color a line that makes a triangle of his or her own color—a set of three points, all connected by lines of that color—that player loses. (There can be no ties in this game, since one may show that if all the lines are colored, there must be a triangle of one of the colors.)

Here's the start of an example game. Assume that blue moves are the bold lines.

red move: (1 2)
blue move: (3 4)
red move: (2 4)
blue move: (5 6)
red move: (2 3); note that (1 4) is a losing move
blue move: (3 6)
red move: (4 5)
blue move: (1 6)



Problem

Write a function `next-blue-move` that chooses a move for the blue player. `Next-blue-move` will take a single argument that represents the current board situation. This argument will be a two-element list, whose first element is a list of two-element lists representing the red moves already made and whose second element is a list of two-element lists representing the blue moves already made. The list

```
(((1 2) (2 4) (2 3) (4 5)) ((3 4) (5 6) (3 6) (1 6)))
```

thus represents the board above.

`Next-blue-move`'s highest priority should be to avoid losing. Next most desirable is to make a winning move, one that forces the red player to form a red triangle. If there is neither a threatening loss nor a winning move, `next-blue-move` may return any as-yet-uncolored line.

Assume that the argument to `next-blue-move` will be a legal board representation as just described. Note, however, that the red moves may appear in any sequence in the first board element; similarly, the blue moves may appear in any sequence in the second board element. Also, a particular move may be specified in either order; for example, the move (3 1) is the same as the move (1 3).

Problem C2

A *matrix* is a nonempty list of lists, all the same length. Some examples:

```
((a b c)
 (d e f)
 (g h i))

((1 2 3 4)
 (5 6 7 8))

((a b)
 (c d)
 (e f)
 (c d))
```

The *diagonal* of a matrix is a list of the elements down the diagonal, namely the first element of the first list of the matrix, followed by the second element in the second list of the matrix, and so on. For a matrix with more columns than rows, the missing rows should be assumed to be filled with ( ); the same applies for missing columns in a matrix with more rows than columns. Thus the diagonals of the matrices above are the lists `(a e i)`, `(1 6 ( ) ( ))`, and `(a d ( ) ( ))`.

Write a function `diagonal` that takes a matrix as argument and returns its diagonal. Assume that the argument to diagonal is a legal matrix as just described. Test your function on the matrices above.

Problem C3

Write a function to execute commands to move blocks on a table.
A table configuration is a list of lists of names of blocks; each element of the configuration represents a stack of blocks. A block may be moved atop another if both blocks are at the top of their respective stacks.
Your function should be called `move`, and take three arguments. The first two arguments are names of blocks; the third argument is a table configuration. If the named blocks do not exist in the configuration, or if they are not both at the top of their respective stacks, or if they name the same block, `move` should print `CANT MOVE` and return the given table configuration. If the move is legal, `move` should return the configuration that results from moving the first block atop the second.

Test your function using the following additional code:

```
(define (execute-commands table)
   (display table)
   (newline)
   (newline)
```

```
(let ((cmd (read)))
   (if (equal? (first cmd) 'move)
      (execute-commands
         (move (second cmd) (third cmd) table) ) ) ) )
```

The `execute-commands` function will successively execute commands until a command that isn't a move is typed. (A command that isn't a move should terminate the program.) Here is a sample interaction, with user input in boldface:

```
: (execute-commands '((a) (b) (c) (d)))
((A) (B) (C) (D))
(move a e)
(CANT MOVE)
((A) (B) (C) (D))
(move d e)
(CANT MOVE)
((A) (B) (C) (D))
(move e a)
(CANT MOVE)
((A) (B) (C) (D))
(move e d)
(CANT MOVE)
((A) (B) (C) (D))
(move a a)
(CANT MOVE)
((A) (B) (C) (D))
(move a b)
((A B) (C) (D))
(move b d)
(CANT MOVE)
((A B) (C) (D))
(move b a)
(CANT MOVE)
((A B) (C) (D))
(move d b)
(CANT MOVE)
((A B) (C) (D))
(move a d)
((B) (C) (A D))
(move a c)
((B) (A C) (D))
(move b d)
((A C) (B D))
(move c d)
(CANT MOVE)
((A C) (B D))
(move c b)
(CANT MOVE)
((A C) (B D))
(move a d)
(CANT MOVE)
((A C) (B D))
(move a b)
((C) (A B D))
(move c a)
((C A B D))
(quit)
```

Test your function using the commands given above.

Problem C4

Background

To determine whether or not a merger of two corporations would violate antitrust regulations, the Justice Department uses the *Herfindahl index*, a mathematical formula developed by a Washington economist. The index is calculated by adding together the sum of the squares of every company's market share in a particular industry or business.

For example, if five companies each have 20% of a market, the Herfindahl index for that market would be 2000, which is 5 times the square of 20. A merger of two of those companies would result in a market distribution of 20% for three of the companies and 40% for the new company, and would push the Herfindahl index to 2800 (3 times the square of 20, plus the square of 40), an 800-point increase. By contrast, the index for a market in which twenty companies each have 5% of the business would total 500. A merger between two of them would increase the index by only 50.

In general, the Justice Department considers an increase of over 500 in the Herfindahl index to be excessive, and thus grounds for not allowing two companies to merge. [Source: *TIME Magazine*, June 28, 1982, p. 49.]

Problem

Write and test a function called `ok-to-merge?`. This function will take three arguments:

- an "industry list" of two-element lists, where each list contains a company name and that company's market share;
- the names (two atoms) of two companies intending to merge.

`Ok-to-merge?` should return true if the two given companies could merge with a Herfindahl index increase of 500 points or less; it should return false if the merger would increase the Herfindahl index by more than 500 points.

Example

Suppose we have the following list of companies:

company	*market share*
IBM	45
Honeywell	20
DEC	15
UNISYS	5
Apple	15

The Herfindahl index for this group of companies is 2900. An industry list representing these companies would then be

```
((IBM 45) (Honeywell 20) (DEC 15) (UNISYS 5) (Apple 15))
```

Given below is a table of all possible mergers of two of these companies, the change in Herfindahl index that results, and the desired result of `ok-to-merge?`.

merger	*Herfindahl index change*	*ok-to-merge? result*
IBM + Honeywell	1800	`#f`
IBM + DEC	1350	`#f`
IBM + UNISYS	450	`true`
IBM + Apple	1350	`#f`
Honeywell + DEC	600	`#f`
Honeywell + UNISYS	200	`true`
Honeywell + Apple	600	`#f`
DEC + UNISYS	150	`true`
DEC + Apple	450	`true`

UNISYS + Apple 150 `true`

You may assume that the industry list argument is as described above, and that the two name arguments correspond to companies in the list. Test your function at least on arguments that produce a Herfindahl index change of less than 500, arguments that produce a change of more than 500, and arguments that produce a change of exactly 500.