CSSE 304 Assignment 4

Objectives: You should learn

- to write more complex recursive procedures in a functional style.
- To learn to check Scheme data for conformance to a particular form

.

This is an individual assignment. You can talk to anyone you want and get as much help as you need, but you should type in the code and do the debugging process, as well as the submission process.

At the beginning of your file, there should be a comment that includes your name and the assignment number. Before the code for each problem, place a comment that includes the problem number. Place the code for the problems in order by problem number.

Turning in this assignment. Write all of the required procedures in one file, and upload it for assignment 4. You should test your procedures offline, using the test code file or other means, **before submitting to the server**.

Unless it is specified otherwise for a particular problem, assume that arguments passed to your procedures will have the correct format. If a problem description says that an argument will have a certain type, you may assume that this is true; your code does not have to check for it.

Restriction on Mutation continues. As in the previous assignments, you will receive zero credit for a problem if any procedure that you write for that problem uses mutation or calls a procedure that mutates something.

Abbreviations for the textbooks:

EoPL	- Essentials of Programming Languages, 3rd Edition
TSPL	- The Scheme Programming Language, 4th Edition (available free scheme.com)
EoPL-1	- Essentials of Programming Languages, 1st Edition (small 4-up excerpt handed out in class, also on Moodle)

#1 (3 points) A *matrix* is a rectangular grid of numbers. We can represent a matrix in Scheme by a list of lists (the inner lists must all have the same length). For example, we represent the matrix

$$\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ 4 & 3 & 2 & 1 & 5 \\ 5 & 4 & 3 & 2 & 1 \end{array}$$

by the list of lists `((1 2 3 4 5) (4 3 2 1 5) (5 4 3 2 1))`. We say that this matrix has 3 rows and 5 columns or (more concisely) that it is a 3×5 matrix. **A matrix must have at least one row and one column.**

Write a Scheme procedure `(matrix-ref m row col)`, where `m` is a matrix, and `row` and `col` are integers. Like every similar structure in modern programming languages, the index numbers begin with 0 for the first row or column. This procedure returns the value that is in row `row` and column `col` of the matrix `m`. Your code does not have to check for illegal inputs or out-of-bounds issues. You can use `list-ref` in your implementation.

matrix-ref : *Listof*(*Listof*(*Integer*)) × *Integer* × *Integer* → *Integer*   (assume that the first argument actually is a matrix)

Examples:
If *m* is the above matrix,

```
       (matrix-ref m 0 0) → 1
       (matrix-ref m 1 2) → 2
```

#2 (15 points) The predicate `(matrix? obj)` should return `#t` if the Scheme object `obj` is a matrix (a nonempty list of nonempty lists of numbers, with all sublists having the same length), and return `#f` otherwise. In this problem, you may *not* assume that the argument to the procedure has the correct type. The point of this problem is to test to see whether the argument has the correct type.

matrix?: *SchemeObject* → Boolean   ; (examples are on the next page)

Examples:
```
      (matrix? 5)                               ➔ #f
      (matrix? "matrix")                        ➔ #f
      (matrix? '(1 2 3))                        ➔ #f
      (matrix? '((1 2 3)(4 5 6)))               ➔ #t
      (matrix? '#((1 2 3)(4 5 6)))              ➔ #f
      (matrix? '((1 2 3)(4 5 6)(7 8)))          ➔ #f
      (matrix? '((1)))                          ➔ #t
      (matrix? '(()()()))                       ➔ #f
```

#3 (10 points) Each row of `(matrix-transpose m)` is a column of `m` and vice-versa.

matrix-transpose: *Listof*(*Listof*(*Integer*)) → *Listof*(*Listof*(*Integer*))   (assume that the argument actually is a matrix)

Examples:
```
      (matrix-transpose '((1 2 3) (4 5 6))) ➔ ((1 4) (2 5) (3 6))
      (matrix-transpose '((1 2 3)))         ➔ ((1) (2) (3))
      (matrix-transpose '((1) (2) (3)))     ➔ ((1 2 3))
```

#4 (10 points) Write `(filter-in pred? lst)` where the type of each element of the list `lst` is appropriate for an application of the predicate `pred?`. It returns a list (in their original order) of all elements of `lst` for which `pred?` returns a true value. **Do not use a built-in procedure that does the whole thing.**

filter-in: *Procedure* × *List* → *List*

Examples:
```
(filter-in positive? '(-1 2 0 3 -6 5))         ➔ (2 3 5)
(filter-in null? '(() (1 2) (3 4) () ()))      ➔ (() () ())
(filter-in list? '(() (1 2) (3 . 4) #2(4 5))) ➔ (() (1 2))
(filter-in pair? '(() (1 2) (3 . 4) #2(4 5))) ➔ ((1 2) (3 . 4))
(filter-in null? '())                          ➔ ()
```

#5 (10 points) `invert` EoPL 1.16, page 26 [Note that this is EoPL, not EoPL-1]

#6 (20 points) `pascal-triangle`. If you are not familiar with Pascal's triangle, see this page: http://en.wikipedia.org/wiki/Pascal_triangle . The first recursive formula that appears on that page will be especially helpful for this problem.
Write a Scheme procedure `(pascal-triangle n)` that takes an integer `n`, and returns a "list of lists" representation of the first n+1 rows of Pascal's triangle. The required format should be apparent from the examples below (note that line-breaks are insignificant; it's just the way Scheme's pretty-printer displays the output in a narrow window) **Don't forget: no mutation!**

pascal-triangle: *Integer* → *Listof*(*Listof*(*Integer*))

```
> (pascal-triangle 4)
((1 4 6 4 1) (1 3 3 1) (1 2 1) (1 1) (1))
> (pascal-triangle 12)
((1 12 66 220 495 792 924 792 495 220 66 12 1)
 (1 11 55 165 330 462 462 330 165 55 11 1)
 (1 10 45 120 210 252 210 120 45 10 1)
 (1 9 36 84 126 126 84 36 9 1)
 (1 8 28 56 70 56 28 8 1)
 (1 7 21 35 35 21 7 1)
 (1 6 15 20 15 6 1)
 (1 5 10 10 5 1)
 (1 4 6 4 1)
 (1 3 3 1)
 (1 2 1)
 (1 1)
 (1))
> (pascal-triangle 0)
((1))
> (pascal-triangle -3) ; if argument is negative, return the empty list
()
```

You should seek to do this simply and efficiently. You may need more than one helper procedure. If your collection of procedures for this problem starts creeping over 25 lines of code, perhaps you are making it too complicated. There is a straightforward solution that is considerably shorter than that.

A challenge for the Pascal Triangle problem (efficiency)

My challenges will not affect your score for the problem.

A simple way to do this problem is to use call the `choose` procedure from a previous homework. But on the average `(choose n k)` has $\Theta(n)$ running time, so producing only the largest row of `(pascal-triangle n)` will be $\Theta(n^2)$. This means that the entire call to `(pascal-triangle n)` will be $\Theta(n^3)$.

Your first challenge is to reduce that to $\Theta(n^2)$. This will require that every number in the triangle be calculated in constant time and that list operations you do will also be constant time for each number in the triangle).

Second challenge: Multiplication is slower than addition. Can you meet the above constraints without doing any multiplications or divisions?

Piazza questions and answers from previous terms

cons two numbers together

I am working on invert on assignment 4, and have found out that whenever 2 numbers are cons'd together, the result is an improper list. e.g.,

> (cons '1 '2)
(1 . 2)

I was wondering why that is, and if there is a simple way to work around this or if I need to completely rethink the problem. Thanks!

hw4scheme_features

edit · good question1

Updated 2 months ago by

Sam

the students' answer,

where students collectively construct a single answer

To make it a proper list, you need to have the last element of the list a pair. That means the last element should be (2) instead of just a 2.

The improper list forms because the last element isn't (). In a list like (list 1 2 3) you actually have a () at the end of 3.

you could try applying a cons with a () at the end to make it a list:

like (cons 2 '())
then (cons 1 previous line)

so it looks like (cons 1 (cons 2 '()))

the instructors' answer,

where instructors collectively construct a single answer

One other comment on the original question:

There is never a need to quote a number, because the value of a number is itself.

```
> (= '3 3)
#t
```

Running pascal-triangle function times out and crashes

Running my pascal-triangle procedure times out after a couple second, then crashes.
Is my procedure too inefficient?

It is hard to see what is going on because SWL editor and repl both automatically exit. I only know the error is "timed out" because I ran it on the PLC Server.

hw4

edit · good question0

Updated 2 months ago by

Megan Phillips

the students' answer,

where students collectively construct a single answer

Did you possibly miss a base case? Because that could be causing infinite recursion.

the instructors' answer,

where instructors collectively construct a single answer

If you are not willing to use emacs or some other fancy editor, try running scheme in a command window, and paste in your scheme code. Maybe this will show you what is going on.

Another thing to try:

Run pascal-triangle for n=1, then 2, then 3, ... until you get the error. If any of the answers along the way are strange, try tracing those.

Chances are good that you have infinite recursion. Does every recursive call make progress toward a base case? If not, this could be the problem.

If these suggestions don't help, you may want to go see the TAs in F217 this afternoon or evening.