CSSE 304 Assignment 2

Objectives You should learn

- to write procedures that precisely meet given specifications.
- to gain experience with picking out parts of lists.
- to write procedures that make simple decisions.
- to practice using previously-written procedures as helpers for new procedures
- to test your code thoroughly.

Administrative preliminaries **(most of these apply to later assignments also)**. If you did not read the administrative preliminaries for Assignment 1 in detail, you should do so. They all apply here.

Abbreviations for the textbooks:

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

Reading Assignment: see the schedule page

Some of the EOPL-1 reading covers topics that are similar to the reading in TSPL, but we believe it is good for you to get more than one perspective on this (in particular, a perspective that is similar to that of EoPL).

This is an individual assignment. You can talk to anyone 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. You should never give or receive code for the individual assignments.

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

Submitting this assignment. Write all of the required procedures in one file, **2.ss**, and upload it for assignment A2 to the PLC grading server. As with A1, do testing on your own computer first, so the server does not get bogged down.

Restriction on Mutation continues. One of the main goals of the first few assignments is to introduce you to the functional style of programming, in which the values of variables are never modified. Until further notice, you may not use `set!` or any other built-in procedure whose name ends in an exclamation point. It will be best to not use any exclamation points at all in your code. **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.**

Assume valid inputs. As in assignment 1, you do not have to check for illegal arguments to your procedures. Note that in the `set?` problem, any Scheme list is a valid input.

Problems to turn in:

#1 (5 points)

(a) (0) Write the procedure `(fact n)` which takes a non-negative integer *n* and returns *n* factorial. You can just copy this procedure from Assignment 0, and call it from your `choose` procedure from part (b).

fact: *NonNegativeInteger* → *Integer*

Examples:

```
(fact 0) ➔  1
(fact 1) ➔  1
(fact 5) ➔ 120
```

(b) (5) Write the procedure `(choose n k)` which returns the number of different subsets of *k* items that can be chosen from a set of *n* distinct items. This is also known as the binomial coefficient and is sometimes written as $\binom{n}{k}$ or ${}_nC_k$. If you've forgotten the formula for this, a Google search for "Binomial Coefficient" should be helpful.

choose: *NonNegativeInteger* × *NonNegativeInteger* → *NonNegativeInteger* **(examples on next page)**

#1 examples:

```
(choose 0 0)  ➔  1
(choose 5 1)  ➔  5
(choose 10 5) ➔ 252
```

#2 (5 points) Write a procedure `(sum-of-squares lon)` that takes a (single-level) list of numbers, `lon`, and returns the sum of the squares of the numbers in `lon`.

sum-of-squares: *Listof(Number)* → *Number*

Examples:

```
(sum-of-squares '(1 3 5 7)) ➔ 84
(sum-of-squares '())        ➔ 0
```

#3 (8 points) Write the procedure `(range m n)` that returns the ordered list of integers starting at the integer *m* and increasing by one until just before the integer *n* is reached (do not include n in the resulting list). This is similar to Python's *range* function. If *n* is less than or equal to *m*, `range` returns the empty list.

range: *Integer* × *Integer* → *Listof*(*Integer)*

Examples:

```
(range 5 10) ➔ (5 6 7 8 9)
(range 5 6)  ➔ (5)
(range 5 5)  ➔ ( )
(range 5 3)  ➔ ( )
```

#4 (10 points) In mathematics, we informally define a *set* to be a collection of items with no duplicates. In Scheme, we could represent a set by a (single-level) list. We say that a list is a set if and only if it contains no duplicates. We say that two objects `o1` and `o2` are duplicates if `(equal? o1 o2)`. Write the predicate `(set? list)`, that takes any list as an argument and determines whether it is a set.

set? : *list* → *Boolean*

Examples:

```
(set? '())                          ➔ #t   ; empty set
(set? '(1 (2 3) (3 2) 5))           ➔ #t   ; (2 3) and (3 2) are not equal?
(set? '(r o s e - h u l m a n))     ➔ #t
(set? '(c o m p u t e r s c i e n c e)) ➔ #f   ; duplicates
```

#5 (5 points) The `union` of two sets is the set of all items that occur in either or both sets (the order of the items in the list does not matter).

union: *Set* × *Set* → *Set*

Examples:

```
(union '(a f e h t b) '(g c e a b)) ➔ (a f e h t b g c)  ; (or some permutation of it)
(union '(2 3 4) '(1 a b 2))        ➔ (2 3 4 1 a b)      ; (or some permutation of it)
```

The remaining problems in A2 continue the ideas of the points, lines and vectors problems from A1. Here is a repeat of some of the instructions from A1. You may want to copy some of your A1 procedures into your A2 solution code file, so you can use them as helpers for your A2 procedures.

We will represent a point or a vector by a list of 3 numbers. For example, the list `(5 6 -7)` can represent either the vector 5**i** + 6**j** - 7**k** or the point (5, 6, -7). In the procedure type specifications below, I'll use *Point* and *Vector* as the names of the types, even though both will be implemented by the same underlying Scheme type.

Note that Scheme has a built-in `vector` type and associated procedures to manipulate vectors. Scheme's `vector` type is similar to the `Object[]` array type in Java. In order to avoid having your code conflict with this built-in type, you should use `vec` instead of `vector` in the names of your functions and their arguments. We could use Scheme's `vector` type to represent the vector in this problem, but I choose not to do so, so that you will get additional practice with picking out parts of lists.

#6 (5 points) Write the procedure `(cross-product v1 v2)` that returns the cross-product (vector product) of the two vectors `v1` and `v2`. Recall that cross-product is only defined for three-dimensional vectors.

cross-product: *Vector* × *Vector* → *Vector*

Examples:
```
(cross-product '(1 3 4) '(3 6 2))  ➔ (-18 10 -3)
(cross-product '(1 3 4) '(3 9 12)) ➔ (0 0 0)
```

#7 (5 points) Write the procedure `(parallel? v1 v2)` that returns `#t` if `v1` and `v2` are parallel vectors, `#f` otherwise. Note that the zero vector is parallel to everything. You only have to guarantee that your procedure will work if the coefficients of both vectors contain only integers or rational numbers. Otherwise round-off error may make two parallel vectors appear to be non-parallel or vice-versa.

parallel?: *Vector* × *Vector* → *Boolean*

Examples:
```
(parallel? '(1 3 4) '(3 6 2))     ➔ #f.
(parallel? '(1 3 4) '(-3 -9 -12)) ➔ #t.
```

#8 (3 points) Write the procedure `(collinear? p1 p2 p3)` that returns `#t` if the points p1, p2, and `p3` are all on the same straight line, `#f` otherwise. Same disclaimer about round-off error as in the previous problems.

collinear?: *Point* × *Point* × *Point* → *Boolean*

Examples:
```
    (collinear? '(1 3 4) '(3 6 2) '(7 12 -2)) ➔ #t
    (collinear? '(1 3 4) '(3 6 2) '(7 12 1))  ➔ #f.
```

Piazza questions and answers from previous terms

list? vs. pair?

Recall that a pair is simply a container for two values; the simplest way to make one is to apply cons.
A list is a linked list of pairs. Each pair except the last one is a reference to the next pair in the list; the `cdr` of the last pair must be null, otherwise the list is improper.

`pair?` is a constant-time procedure that simply asks, "is this value a reference to a pair?"
`list?` is a linear-time operation that asks, "is this value either null or a reference to the first pair of a proper list?

So efficiency is one basis to choose between the two tests.

I hope that the following transcript will help you better understand these procedures.

```
> (list? '())
#t
> (pair? '())
#f
> (list? '(a b c))
#t
> (pair? '(a b c))
#t
> (list? '(a b . c))
#f
> (pair? '(a b .c))
#t
> (pair? 'a)
#f
> (list? 'a)
#f
```

duplication in lists

Can you ever do better than O(n^2) for determining the duplication of numbers in a list?

A novel approach might be for each index compare this value to all other index values.

Another way is to pre-process with a sorting algorithm with Big O lower than O(n^2). Then compare each value next to each other. This is O(n) + complexity of sort.
However sorting isn't an easy option in this case since we can't mutate anything yet.

So are we left to O(n^2) efficiency for now until we learn more about manipulation in scheme?

the instructors' answer,

where instructors collectively construct a single answer

Actions

If you don't use any auxiliary data structure, N^2 is the best you can do. If you copy the numbers into a binary heap or balanced tree you can do O(N log N) worst case. With a hash table you can have O(N) expected value, but O(N^2) worst case.

Finally, I don't think Scheme's `sort` procedure mutates anything.

Operations with lists

I had no problem using - and + between a number and a list, but I do if I use it with two lists, even if there is only one element on them. Are there special operators for adding and subtracting the elements in a list? Or is it not possible to operate with them being in a list?

Thank you

hw1

This private post is only visible to Instructors and 2 others

edit · good question 0

Updated 2 months ago by

Ana Huerta

the instructors' answer,

where instructors collectively construct a single answer

Depending on what you are trying to do, **`apply`** may be what you need. Or (for things like A2-A5) you can use a loop or (for A1 where all lists have fixed length) you can just write two or three - or + expressions.

Comparing numeric quantities

If x and y are known to be numbers, use `(= x y)` instead of `(equal? x y)` to compare them.