CSSE 304 Assignment 3 Updated for Fall, 2019-20

Objectives You should learn

- to write procedures that meet certain specifications.
- to write more complex recursive procedures in a functional style.
- to test your code thoroughly.

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 to the PLC server for assignment 3. You should test your procedures offline, using the test code file or other means, **before submitting to the server**.

Assume that arguments 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 test for arguments with the wrong types.

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, 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)

Problems to turn in:

The first two problems refer to the definition of *sets in Scheme* from Assignment 2, which are repeated here.

> We represent a set by a list of objects. We say that such a list is a set if and only if it contains no duplicates. By "no duplicates", I mean that no two items in the list are `equal?` .

#1 (10 points) Write the procedure `(intersection s1 s2)` The intersection of two sets is the set containing all items that occur in both sets (order does not matter).

intersection: *Set* × *Set* → *Set*

Examples:

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

You may assume that each arguments is a sets; no need to test for that. Again, use `equal?` as your test for duplicate items.

#2 (10 points) A set X is a *subset* of the set Y if every member of X is also a member of Y. The procedure `(subset? s1 s2)` takes two sets as arguments and tests whether `s1` is a subset of `s2`. You may want to write a helper procedure. You may assume that both arguments are sets.

subset?: *Set* × *Set* → *Boolean*

Examples

```
(subset? '(c b) '(a c d b e)) ➔ #t
(subset? '(c b) '(a d b e))   ➔ #f
(subset? '() '(a d b e))      ➔ #t
(subset? '(a d b e) '())      ➔ #f
```

#3 (15 points) A *relation* is defined in mathematics to be a set of ordered pairs. The set of all items that appear as the first member of one of the ordered pairs is called the *domain* of the relation. The set of all items that appear as the second member of one of the ordered pairs is called the *range* of the relation. In Scheme, we can represent a relation as a list of 2-lists (a 2-list is a list of length 2). For example `((2 3) (3 4) (-1 3))` represents a relation with domain `(2 3 –1)` and range `(3 4)`. Write the procedure `(relation? obj)` that takes any Scheme object as an argument and determines whether or not it represents a relation. You will probably want to use `set?` from a previous assignment in your definition of `relation?`. [Note that because you were just getting

started on Scheme, the previous test cases for `set?` did not include any values that were not lists. Now you may want to go back and "beef up" your `set?` procedure so it returns `#f` if its argument is not a list. Note that you may use `list?` in your `set?` code if you wish.]

relation?: *Scheme-object* → *Boolean*

Examples

```
(relation? 5)                          ➔ #f
(relation? '())                        ➔ #t
(relation? '((a b) (b c)))             ➔ #t
(relation? '((a b) (b a) (b b) (a a))) ➔ #t
(relation? '((a b) (b c d)))           ➔ #f
(relation? '((a b) (c d) (a b)))       ➔ #f
(relation? '((a b) (c d) "5"))         ➔ #f
(relation? '((a b) . (b c)))           ➔ #f
```

#4 (10 points) Write a procedure `(domain r)` that returns the set that is the domain of the given relation. Recall that the *domain* of a relation is the set of all elements that occur as the first element of an ordered pair in the relation.

domain: *Relation* → *Set*

Examples

```
(domain '((1 2) (3 4) (1 3) (1 6))) ➔ (1 3)   ; or some permutation of it
(domain '())                        ➔ ()
(domain '((a b) (b d) (a e) (c e))) ➔ (a b c) ; or some permutation of it
```

#5 (15 points) A relation is *reflexive* if every element of the domain and range is related to itself. I.e., if `(a b)` is in the relation, so are `(a a)` and `(b b)`. The procedure `(reflexive? r)` returns `#t` if relation r is reflexive and `#f` otherwise. You may assume that `r` is a relation.

Note that this problem is considerably more challenging than most of the other problems in assignments 1-3.

reflexive?: *Relation* → *Boolean*

Examples:

```
(reflexive? '((a b) (b a) (b b) (a a))) ➔ #t
(reflexive? '((a b) (b c) (a c)))       ➔ #f
```

#6 (10 points) Consider hailstone sequences, related to the Collatz conjecture. If the positive integer *n* is a number in such a sequence, the next number in the sequence is (image below is from the linked Wikipedia page).

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2. \end{cases}$$

The first case is for *n* even, the second is for *n* odd. The conjecture is that all such sequences eventually reach the number 1. Define the procedure `(hailstone-step-count n)` to be the number of applications of the above function *f* required to reach 1 if we start with *n*. If the conjecture happens to be false, then there exists some n>0 such that hailstone-step-count(n) is infinite. You will not encounter any such numbers in the test cases for this problem, so your code does not need to attempt to check for this!

.

Examples:

```
(hailstone-step-count 1)   ➔ 0     ; already 1, so no applications of f are needed
(hailstone-step-count 2)   ➔ 1     ; 2 > 1
(hailstone-step-count 3)   ➔ 7     ; 3 > 10 > 5 > 16 > 8 > 4 > 2 > 1
(hailstone-step-count 4)   ➔ 2     ; 4 > 2  > 1
(hailstone-step-count 7)   ➔ 16    ; 7 > 22 > 11 > 34 > 17 > 52 > 26 > 13 > 40 > 20 > 10 > 5 > 16 > 8 > 4 > 2 > 1
(hailstone-step-count 871) ➔ 178
```

Background for problems 7-8:

> A *set* is a list of items that has no duplicates. In a *multi-set*, duplicates are allowed, and we keep track of how many of each element are present in the multi-set. For problems in this course, we will assume that each element of a multi-set is a symbol. We represent a multi-set as a list of 2-lists. Each 2-list contains a symbol as its first element and a positive integer as its second element. So the multi-set that contains one **a**, three **b**s and two **c**s might be represented by `((b 3) (a 1) (c 2))` or by `((a 1) (c 2) (b 3))` .

#7 (10 points) Write a Scheme procedure `(multi-set? obj)` that returns `#t` if `obj` is a representation of a multi-set, and `#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.

multi-set? : *scheme-object* → *Boolean*

Examples:
```
(multi-set? '())                  ➔ #t
(multi-set? '(a b))               ➔ #f
(multi-set? '((a 2)))             ➔ #t
(multi-set? '((a 0)))             ➔ #f
(multi-set? '(a b))               ➔ #f
(multi-set? '((a 2) (b 3)))       ➔ #t
(multi-set? '((a 2) (a 3)))       ➔ #f
(multi-set? '((a 3) b))           ➔ #f
(multi-set? 5)                    ➔ #f
(multi-set? (list (cons 'a 2)))   ➔ #f
(multi-set? '((a e) (b 3) (a 1))) ➔ #f
```

#8 (6 points) Write a Scheme procedure `(ms-size ms)` that returns the total number of elements in the multi-set `ms`.
(suggested, but not required) Can you do this with very short code that uses `map` and `apply`?

ms-size : *multi-set* → *integer*

Examples:
```
(ms-size '())           ➔ 0
(ms-size '((a 2)))      ➔ 2
(ms-size '((a 2)(b 3))) ➔ 5
```

#9 (3 points) Write a recursive Scheme procedure `(last ls)` which takes a list of elements and returns the last element of that list. This procedure is in some sense the opposite of `car`. You may assume that your procedure will always be applied to a non-empty proper list. You are not allowed to reverse the list or to use `list-tail`. [**Something to think about** (not directly related to doing this problem): Note that `car` is a constant-time operation. What about `last`?]

last: *Listof*(*SchemeObject*) → *SchemeObject*

Examples:
```
(last '(1 5 2 4))  ➔ 4
(last '(c))        ➔ c
```

#10 (5 points) Write a recursive Scheme procedure `(all-but-last lst)` which returns a list containing all of `lst`'s elements but the last one, in their original order. In a sense, this procedure is the opposite of `cdr`. You may assume that the procedure is always applied to a valid argument. You may not reverse the list. You may assume `lst` is a nonempty proper list. [**Something to think about** (not directly related to doing this problem): `cdr` is a constant-time operation. What about `all-but-last`?]

all-but-last: *Listof*(*SchemeObject*) → *Listof*(*SchemeObject*)

Examples:
```
(all-but-last '(1 5 2 4))  ➔ (1 5 2)
(all-but-last '(c))        ➔ ()
```

Piazza questions and answers from previous terms

Problem 3: relation?

are Strings allowed in sets?. a better questions would be: What data types are allowed in sets? and for this problem are we supposed to only be looking for lists (that are sets) that contain lists that only contain two pairs in them?

so:
is this '((a b) (c d) "5")) a set? but it shouldn't be allowed into a relation
is this '((a b) . (b c))) a set? and what does the dot in the middle do to the list?

thanks!

the students' answer,
where students collectively construct a single answer

In respect to a part of your second question, the dot denotes an improper list (that is, the cdr of a proper list is the empty list, '(), while the cdr of an improper list is the element itself).

E.g. let list1 be '(1 2): (cdr list1) -> (2), and (cdr (cdr list1)) -> '()
let list2 be '(1 . 2): (cdr list1) -> 2, and (cdr (cdr list2)) -> error! (try it yourself to see why).

the instructor's answer

According to the definition of **set** that I gave in the assignments, any Scheme object is allowed in a set.

> '((a b) . (b c))
((a b) b c)
> '((a b) (c d) "5"))
((a b) (c d) "5")

Each of these is a set of three items. The . is basically an abbreviation for **cons**. When we cons (a b) onto the front of (b c) we get a list of three items: (a b) , b, and c.

A3 problem 3

Could someone elaborate on what a relation is? for instance why is this example (relation? '((a b) (c d) (a b))) not a relation

the students' answer,
where students collectively construct a single answer

A relation by definition is a set. The fact that (a b) is in the list twice makes it not a set and thus not a relation

Assignment 3: Multi-set

For the multi-set problems in Assignment 4, does the "symbol" mentioned have to be a letter or can you have a multi-set that looks like: ((3 3) (q 4)), which would interpret as three 3's and four q's?

Edit: I actually just went ahead and let scheme decide by using (symbol?).

the students' answer,

where students collectively construct a single answer

From TSPL: "Scheme supports many types of data values ... including characters, strings, symbols, lists or vectors of objects, and a full set of numeric data types."
To me, this sounds like numbers and symbols are different data types. There are also no tests with numbers (like your ((3 3) (q 4)) example) in the code in the assignment or the test code.
That said, I don't think it should change the implementation of the problem much, if at all.

EDIT: Looks like I was beaten to the answer

~ An instructor (Claude Anderson) endorsed this answer ~

the instructors' answer,
where instructors collectively construct a single answer

3 is a number, not a symbol.
'a is a symbol.
'abc123 is a symbol.
'+ is a symbol
(car '(a b c)) is a symbol
'1 is not a symbol

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 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
```

What does reflexive mean?

A student wrote:

I'm having trouble understanding what I actually need to calculate. The description of what makes a relation reflexive is confusing. Is there any references that I could look at to understand what makes a relation reflexive?

Think of "(a, b) ε R" as another way of saying “In relation R, a is related to b”. So a relation is reflexive iff for every element **a** in the domain or range of R, it is true that (**a**, **a**) ε R.

Other links that may help:
https://en.wikipedia.org/wiki/Reflexive_relation
https://www.csee.umbc.edu/~stephens/203/PDF/10-2.pdf

Example (from http://www.math-only-math.com/reflexive-relation-on-set.html):
Consider, for example, a set A = {p, q, r, s}.
The relation R11 = {(p, p), (p, r), (q, q), (r, r), (r, s), (s, s)} in A is reflexive, since every element in A is R11-related to itself.
But the relation R22 = {(p, p), (p, r), (q, r), (q, s), (r, s)} is not reflexive in A since q, r, s ∈ A but (q, q) ∉ R22, (r, r) ∉ R22 and (s, s) ∉ R

Use let to create a local variable for `nearest-point?` (instructor note)

`let` is used to create local variables. For instance, if we wanted to have a local variable to hold the value of n-1 in the factorial function, we might write

```
(define fact-with-let
  (lambda (n)
    (if (= n 0)
        1
        (let ([fact-less (fact-with-let (- 1 n))])
          (* n fact-less)))))
```

This is just to show the syntax for creating a local variable; it doesn't really make the `fact` code better. But in nearest-point, after you apply `nearest-point` to p and the cdr of the list, what you do with the result is more complex, and using `let` to save the value of the recursive call in a local variable can allow you to avoid re-applying the function to the cdr.

```
(define nearest-point
  <em>; some stuff that comes before the let ...</em>
  (let ([nearest-from-cdr (nearest-point p (cdr ls))])
    <em>; compare distance between p and nearest-from-cdr to something else.</em>
```

Don't forget that you can trace your code!

If you are having trouble understanding what your code is doing, `trace` can help you see what your code is doing prior to the wrong answer or the error. You can trace procedures that you write as well as built-in procedures.

An extreme example:

```
> (trace - *)
Warning in trace: redefining -; existing references will not be traced
Warning in trace: redefining *; existing references will not be traced
(- *)
> (define fact
  (lambda (n)
    (if (= n 0)
       1
       (* n (fact (- n 1))))))
> (trace fact)
(fact)
> (fact 3)
|(fact 3)
| (- 3 1)
| 2
| (fact 2)
| |(- 2 1)
| |1
| |(fact 1)
| | (- 1 1)
| | 0
```

```
| | (fact 0)
| | 1
| |(* 1 1)
| |1
| (* 2 1)
| 2
|(* 3 2)
|6
6
> (untrace)
(- * fact)
>
```

relation on hw3

Question:
Below are two test cases.
I can't tell why the bottom one is false. It seems like the top one but with just one more pair. I don't think it is illegal to 3 but not 4.
I'm confused what condition it fails on.

(relation? '((a b) (b a) (b b) (a a))) è #t

(relation? '((a b) (c d) (a b))) è #f

Answer:

A relation is a *set* of ordered pairs. This one is not a set because (a b) occurs twice.