1 Week of 01/30: set operations

In this recitation, we discussed how sets could be represented in Scheme as lists with no repeated elements, and implemented a few functions for performing basic manipulations on sets. If we encode mathematical sets as lists with no repeated elements, there may be many different ways of encoding the same set, for instance $\{1, 2, 3\}$ could be represented as `'(1 2 3)` or `'(2 3 1)` or `(3 1 2)` and so on. Since the order of the elements of a *list* matters in Scheme, two different representations of the same set may be reported as unequal by the `equal?` function. We'll remedy this later by writing our own function called `set-equal?` which checks whether or not two lists are equal *as sets*.

We'll start by writing a function called `set-insert` which takes two arguments - an element `x` and a list `ls` (assumed to be a set, i.e. a list with no repeated elements) - and returns a copy of the set with that element added to it. Since sets shouldn't contain duplicate elements, if the element `x` is already contained in the list `ls`, then our function should return `ls` unchanged. On the other hand, if it's not already in the list, it should return a copy of the list with that element added (it doesn't matter where we add to the list in this case). Here's a possible implementation of this function:

```
(define set-insert
  (lambda (x ls)
    (cond
      ((null? ls) (cons x '()))
      ((equal? x (car ls)) ls)
      (else (cons (car ls) (set-insert x (cdr ls))))
    )
  )
)
```

How does this function work? Given the arguments `x` and `ls`, it splits into three cases:

- If `ls` is empty, i.e. `(null? ls)`, then we should return the singleton list `(cons x '())` with `x` as its only element.
- Otherwise, if `ls` is not empty, we should compare `x` with the first element of `ls`. If `(equal? x (car ls))`, then clearly `x` would be a duplicated element, so we should just return the list `ls` unchanged.
- Else, if `ls` is nonempty and `x` is different from its first element, then we should `set-insert` the element `x` into the tail of the list `(cdr ls)`. This is where the recursive call happens.

We can test out our function in the REPL:

```
> (set-insert 4 '(1 2 3))
'(1 2 3 4)
> (set-insert 4 '(1 2 4))
'(1 2 4)
```

Notice that our function inserts new elements at the *end* of lists, but there's no reason why it need to work this way - `(set-insert 4 '(1 2 3))` could have also returned the correct answer `'(4 1 2 3)`, for instance. In fact, we can write a different implementation that behaves differently while still being correct. Here's an implementation of `set-insert` that uses the helper function `member?`:

```
(define member?
  (lambda (x ls)
    (if
     (null? ls)
     #f
     (or (equal? x (car ls)) (member? x (cdr ls))))
  )
)

(define set-insert2
  (lambda (x ls)
    (if (member? x ls) ls (cons x ls))
  )
)
```

This helper function `member?` decides whether or not the argument `x` is a member of the input list `ls`. Given this helper function, we can write a *non-recursive* version of our set insertion function which either returns `ls` or `(cons x ls)` depending on whether or not `x` is already in `ls`, as determined by the `member?` function. Notice that in some cases, this second function produces different (yet still correct) output from the original:

```
> (set-insert2 4 '(1 2 3))
'(4 1 2 3)
> (set-insert 4 '(1 2 4))
'(1 2 4)
```

Now let's write a function called `union-two` that computes the union of two input sets `ls1` and `ls2`. To accomplish this, we can use our previously defined `set-insert` function. Here's a possible implementation:

```
(define union-two
  (lambda (set1 set2)
    (if
     (null? set1)
     set2
     (union-two (cdr set1) (set-insert (car set1) set2)))
  )
)
```

Essentially, this function leverages the fact that unioning the empty set {} with any other set leaves that set unchanged, and unioning a nonempty set `set1` with another set `set2` can be accomplished by extracting its elements and inserting them into the second set one at a time. We can think of this recursive definition as "popping" the elements out of `set1` one at a time, as if it were a stack, and inserting each of them into `set2`. Keep in mind, however, that no *mutation* is happening (even though the way we talk about these sets might suggest that they're being "changed"). Here's a test case of the `union-two` function

```
> (union-two '(1 2 3 4 5) '(2 4 6 8))
'(2 4 6 8 1 3 5)
```

Of all the possible ways the elements of this set could be ordered, can you see (from the way we defined `union`) why they come out in this specific order?

Next, let's write a function `contains?` that decides whether or not its first argument `set1` is a subset of its second argument `set2`, i.e. whether or not `set1` ⊂ `set2`, or whether all of the elements of `set1` are also elements of `set2`. Here's one possible implementation:

```
(define contains?
  (lambda (set1 set2)
    (if
      (null? set1)
      #t
      (and
        (element-of? (car set1) set2)
        (contains? (cdr set1) set2)))
  )
)
```

This definition is taking advantage of the fact that the empty set {}, which is represented by the empty list `'()`, is contained in *every other set*, meaning that we should always return `#t` in the case of `set1` being `null?`, whereas if `set1` is nonempty, then it's contained in `set2` if and only if its first element belongs to `set2` and all the rest of its elements belong to `set2`. This second case is where the recursive call comes in.

There's actually a clever way of implementing this function with much less code (and no recursive call!) using our `union-two` function:

```
(define contains2?
  (lambda (set1 set2)
    (equal? set2 (union-two set1 set2))
  )
)
```

Recall that our `union-two` function pops the elements out of `set1` one at a time and inserts them into `set2`, if they aren't already there. But if `set1` ⊂ `set2`, then each of the elements that are inserted this way already exist in `set2`, meaning that the result of `union-two set1 set2` will be identical to `set2` and our function will give `#t`. On the other hand, if `set1` is not a subset of `set2`, then it will contain some novel element that isn't in `set2`, meaning that `(union-two set1 set2)` will be different from `set2` and our function will give `#f` as expected. Theoretically, this function is taking advantage of the fact that $A \subset B$ if and only if $B = A \cup B$ for sets A and B. Be careful, though: a subtle change to this function will cause it to malfunction. Consider the following:

```
(define bad-contains?
  (lambda (set1 set2)
    (equal? set2 (union-two set2 set1))
  )
)
```

Try using this function to test whether the set represented by `(2 3 4)` is contained in the set represented by `(1 2 3 4 5)`. Why does it give a wrong result?

Now we'll just write one more function - the one called `set-equal?` that we promised at the beginning, which checks whether two lists are equal *as sets*. That is, given two lists with no duplicated elements, this function will check whether the elements of the two lists are rearrangements of each other. With a little bit of set theory, this can be accomplished without a single recursive call: a commonly used fact is that two sets A and B are equal if and only if they contain each other, i.e. if $A \subset B$ and $B \subset A$. Hence, we can implement our function as follows:

```
(define set-equal?
  (lambda (set1 set2)
    (and (contains? set1 set2) (contains? set2 set1))
  )
)
```

If you're looking for a little bit of extra practice, try your hand at writing a couple of these set-related functions:

- `intersection` which calculates the intersection $A \cap B$ of two sets A and B
- `union-all` which, given a list of sets, calculates the union of all of them
- `intersection-all` which, given a list of sets, calculates the intersection of all of them
- `powerset` which, given a set A, calculates a set consisting of all of its subsets - for instance, `(powerset '(1 2 3))` might give `'(() (1) (2) (3) (1 2) (1 3) (2 3) (1 2 3))`

- `cartesian-product` which, given two sets A and B, calculates their cartesian product $A \times B$ (the elements of this new set can be represented as pairs in Scheme)
- `product?` which, given a set of pairs, determines whether that set is equal to the cartesian product of some pair of sets (for instance `'((1 . 2) (3 . 4))` is not the product of two sets, but `'((1 . 2) (1 . 4) (3 . 2) (3 . 4))` is the product of the sets `'(1 3)` and `'(2 4)`)