CSE341 Section 2:
April 5th, 2018

Warm-up:

Write a function my-xor which takes 2 arguments, here are some examples:

```
(my-xor #t #f) -> #t
(my-xor (= 1 2) (= 2 3)) -> #f
```

Starter code:

```
(define-syntax my-xor
 (syntax-rules ()
   ((my-xor                    )
        (

               ))))
```

Note for this one: xor should really be done using a function instead, since we need to evaluate all its values. This is just for practice.

Q3 (Bonus) — placed here so Q1 and Q2 can have full pages.

Try to implement a macro that represents let*-expressions (call it my-let*). Remember that let* expressions add each binding to the environment one at a time. This requires a concept we haven't discussed in class yet, but is still an interesting problem.

Q1:
The lecture notes for macros include a definition for my-or that works just like the built-in or in Racket.

```
(define-syntax my-or
 (syntax-rules ()
   ((my-or) #f)
   ((my-or e1 e2 ...)
      (let ([temp e1])
        (if temp
             temp
             (my-or e2 ...))))))
```

Given this definition, if we expand `(my-or (= x 2))`, we get

```
(let ([temp (= x 2)])
  (if temp temp (my-or)))
```

This would further expand to

```
(let ([temp (= x 2)])
  (if temp temp #f))
```

Modify the rule so it just expands `(my-or (= x 2)) to (= x 2)` instead.
It should still work correctly for `(my-or).`

Starter code:

```
(define-syntax modified-or
 (syntax-rules ()
```

Q2:
Let’s try to implement a macro that represents let-expressions (call it parallel-let):

(a): implement parallel-let that allows no variable binding and allows one or more expressions
For example:

```
(parallel-let () (printf “cse”)) -> “cse”
(parallel-let () (printf “341”) (- 2 4)) -> “341”-2
```

(b): implement parallel-let that allows one or more variable binding with one or more expressions
For example:

```
(parallel-let (x y z) (3 2 6) (+ x y z)) -> 11
```

Starter code:

```
(define-syntax parallel-let
  (syntax-rules ()
```