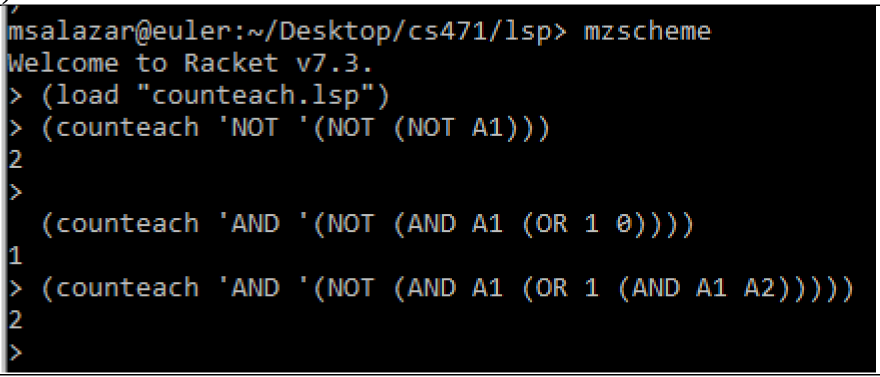
Marco Salazar Lisp Programming 11/18/2020

In this assignment I was supposed to experiment with Lisp programming. In it I was to make a function which counts the number of times a logical operator (OR, AND, NOT) is used in a valid circuit description. Next, I needed to make a program which uniquely lists all of the input variables. Finally, I needed to implement simple Boolean tautologies such that when given a circuit description, I could simplify it by breaking down it down with logical rules. The following is the code and a couple examples for each problem.

Problem 1

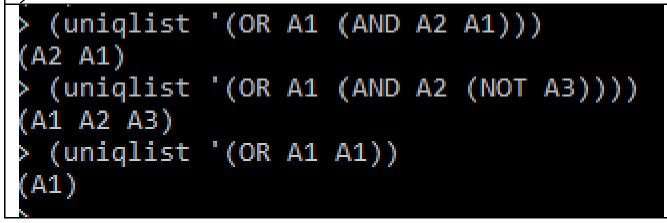
```
; Name: Marco Salazar
; Date: 11/18/2020
; Assignment: Practice with Lisp Programming
; Problem: In this code I count each time a logical operator is used in a Circuit Design.
; eg.  (counteach 'OR '(OR 1 (AND 1 A1))) = 1

;Precondition: L is a valid circuit design, and x is an atom
; Postcondition: the count of how many times the atom appeared in the circuit design
(define (counteach x L)
        (cond   ((null? L ) 0)
                ((not (list? L))
                (if (eq? x L) 1 0)) ;; if equal return 1 otherwise return 0
                (else (+ (counteach x (car L))
                         (counteach x (cdr L))
                      ) ;; finally sum the count of the first element and the rest.
                )
        )
)
```



```
msalazar@euler:~/Desktop/cs471/lsp> mzscheme
Welcome to Racket v7.3.
> (load "counteach.lsp")
> (counteach 'NOT '(NOT (NOT A1)))
2
>
  (counteach 'AND '(NOT (AND A1 (OR 1 0))))
1
> (counteach 'AND '(NOT (AND A1 (OR 1 (AND A1 A2)))))
2
>
```

Problem 2

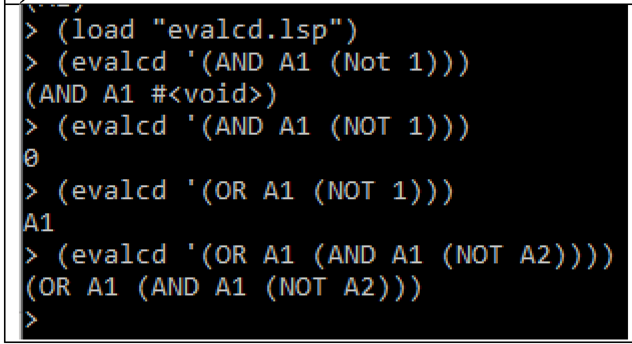
```
; Name: Marco Salazar
; Date: 11/18/2020
; Assignment: Practice with Lisp Programming
; Problem: In this code I Uniqueley list all of the input variables A1-1000
; eg.  (uniq '(OR A1 (AND A2 A1)))  == '(A1 A2)

;Precondition: L is a valid circuit design
; Postcondition: list all of the input variables A1-1000
(define (uniqlist L)
        (findinputvars (uniq (flatten L)))
)



; Pre: given a flattened List
; post: make a list of all the unique elements
(define (uniq L)
        (cond   ((null? L) '())
                ((not (list? L)) '())
                ((member (car L) (cdr L)) (uniq (cdr L))) ;;if the current element exists in the rest of
the list, skip it
                (else (cons (car L) (uniq (cdr L))))
        )
)



; Pre: given a list with no duplicates
; Post: make a new list with all the variables A1-1000
(define (findinputvars L)
        (cond   ((null? L) '())
                ((or    (eq? (car L) 1)
                        (eq? (car L) 0)
                        (eq? (car L) 'AND)
                        (eq? (car L) 'OR)
                        (eq? (car L) 'NOT))
                        (findinputvars (cdr L));; if it is not a variable, keep searching
                 )
                (else (cons (car L) (findinputvars (cdr L))))
        )
)
```



```
> (uniqlist '(OR A1 (AND A2 A1)))
(A2 A1)
> (uniqlist '(OR A1 (AND A2 (NOT A3))))
(A1 A2 A3)
> (uniqlist '(OR A1 A1))
(A1)
```

Problem 3

```
; Name: Marco Salazar
; Date: 11/18/2020
; Assignment: Practice with Lisp Programming
; Problem: In this code it reduces a Circuit design with simple boolean tautologies to its
simplest form
; eg.  (evalcd '(AND 1 S)) == S

;Precondition: L is a valid circuit design
; Postcondition: the simplified version of L using boolean tautologies
(define (evalcd CD)
        (cond   ((null? CD) '())
                ((not (list? CD)) CD)  ;; if it is Not, and, OR deal with it appropriately.
                ((eq? (car CD) 'NOT) (evalcd_not CD))
                ((eq? (car CD) 'AND) (evalcd_and CD))
                ((eq? (car CD) 'OR)  (evalcd_or  CD))
        )
)

; pre: given CD as a valid circuit design
; post: the reduced version of this not expression
(define (evalcd_not CD)
        (cond   ((eq? (evalcd (cadr CD)) 0) 1) ;; preform negation operations unless it is a
statement or variable
                ((eq? (evalcd (cadr CD)) 1) 0)
                (else (cons 'NOT (list (evalcd (cadr CD)))))
        )
)

; pre: given CD as a valid circuit design
; post: the reduced version of this AND expression
(define (evalcd_and CD)
        ;; AND tautology rules
        (cond   ((eq? (evalcd (cadr CD)) 0) 0)
                ((eq? (evalcd (caddr CD)) 0) 0)
                ((eq? (evalcd (cadr CD)) 1) (evalcd (caddr CD)))
                ((eq? (evalcd (caddr CD)) 1) (evalcd (cadr CD)))
                (else (cons 'AND
                            (list (evalcd (cadr CD))
                                  (evalcd (caddr CD)))
                      )
                )
        )
)

; pre: given CD as a valid circuit design
; post: the reduced version of this OR expression
(define (evalcd_or CD)
        ;; OR tautology rules
        (cond   ((eq? (evalcd (cadr CD)) 1) 1)
                ((eq? (evalcd (caddr CD)) 1) 1)
                ((eq? (evalcd (cadr CD)) 0) (evalcd (caddr CD)))
                ((eq? (evalcd (caddr CD)) 0) (evalcd (cadr CD)))
                (else (cons 'OR
                             (list (evalcd (cadr CD))
                                   (evalcd (caddr CD)))
                      )
                )
        )
)
```



```
> (load "evalcd.lsp")
> (evalcd '(AND A1 (Not 1)))
(AND A1 #<void>)
> (evalcd '(AND A1 (NOT 1)))
0
> (evalcd '(OR A1 (NOT 1)))
A1
> (evalcd '(OR A1 (AND A1 (NOT A2))))
(OR A1 (AND A1 (NOT A2)))
>
```