Project Partners: Jonathan Luetze, Ashley Tran and Tiana Terrell
Date: February 7th, 2018

<u>Source Code:</u>

```
;Written by Jonathan Luetze, Ashley Tran and Tiana Terrell

;Defining what a constant is to differentiate the rules
(define (constant? x) (number? x))

;To check for list and the operator of the list
(define (is-sum x)
  (and (pair? x) (equal? (car x) '+)))

(define (is-product x)
  (and (pair? x) (equal? (car x) '*)))

(define (is-difference x)
  (and (pair? x) (equal? (car x) '-)))

;To check for a legal operation
(define (is-operation x)
  (or (is-sum x) (is-product x) (is-difference x)))

;To check for the correct operator
(define (is-plus x)  (equal? x '+))
(define (is-minus x) (equal? x '-))
(define (is-mul x)   (equal? x '*))

;To retrieve the right and left side of a list within a list
(define (rightSide x) (caddr x))
(define (leftSide x)  (cadr x))

;To see if both args are constants
(define (2const? op arg1 arg2)
      (and (constant? arg1) (constant? arg2)))

;Function to apply all rules
(define (apply-rules op arg1 arg2)
  (cond
  ;Rules 1,3,5
   ((2const? op arg1 arg2)
       (cond   ((is-plus op)  (+ arg1 arg2))
               ((is-minus op) (- arg1 arg2))
               ((is-mul op)   (* arg1 arg2))
       )
   )
  ;Rules 2,4,6
   ((constant? arg2)
       (cond   ((is-plus op)  (simplify (list '+ arg2 arg1)))
               ((is-mul op)   (simplify (list '* arg2 arg1)))
               ((is-minus op) (list '+ (list '- arg2) arg1))
        )
   )
  ;Rules 7,9
   ((and (is-plus op) (is-sum arg2))
       (simplify(list '+ (list '+ arg1 (leftSide arg2)) (rightSide arg2)))
   )
  ;Rules 8,10
   ((and (is-mul op) (is-product arg2))
       (simplify(list '* (list '* arg1 (leftSide arg2)) (rightSide arg2)))
   )
   ;Rules 11-14, 18
   ((and (is-mul op) (is-sum arg2))
       (simplify(list '+ (list '* arg1 (leftSide arg2)) (list '* arg1 (rightSide
arg2))))
   )
   ;Rules 15,16
   ((and (is-mul op) (is-difference arg2)(constant? arg1))
       (simplify(list '- (list '* arg1 (leftSide arg2)) (list '* arg1 (rightSide
arg2))))
   )
   ;Rule 17
   ((and (is-mul op) (is-sum arg1))
           (simplify (list '+ (list '* (leftSide arg1) arg2) (list '* (rightSide arg1)
arg2)))
   )
   ;Rule 19
   ((and (is-mul op) (is-difference arg1))
           (simplify (list '- (list '* (leftSide arg1) arg2) (list '* (rightSide arg1)
arg2)))
   )
   ;Rule 20
   ((and (is-mul op) (is-difference arg2))
           (simplify (list '- (list '* arg1 (leftSide arg2)) (list '* arg1 (rightSide
arg2))))
   )
   ;Else return list
   (else (list op arg1 arg2))
  )
)
;Main simplify function that recursively simplifies the equations
(define (simplify exp)
  (cond ((is-operation exp)
           (apply-rules (car exp)
                        (simplify (cadr exp))
                        (simplify (caddr exp))
           )
        )
        (else exp)
  )
)
```

Design Description:
We decided as a group to approach this assignment with a conditional statement in mind and to check the input for each rule given in the assignment file. Before entering the conditional statement however, there are functions that would take any inputted equation and break it into its most simplest parts with the use of car, cadr, and caddr - vital syntax covered in class. For example, there is a function to define what a constant is, define and differentiate between the operators, and so on. Most importantly is the function that will set the 'leftSide' and 'rightSide' of a list within a list. All of these mechanisms combined allowed us to determine patterns in the rules and write matching conditions for each one with the use of our defined functions. After a rule matched the condition, it created the in the assignment specified list and sent it back through the function recursively until it was completely resolved. The structure of our code was based off of what was recommended to us in the lab and implemented by Jonathan, who took the reigns in this project with the coding, design and implementation while Tiana and Ashley contributed in making rules, adding recursion, and debugging.

Difficulties:
The most difficult part of this project was pairing and keeping track of parenthesis as well as learning how the recursion works. Scheme is very sensitive and does not seem to have a good error tracking system like other languages to correctly show where the missing parenthesis (or brackets) are. Despite there being a slight underline under the parenthesis as you checked to see where it went, it was still a very inadequate way of making sure every statement was closed properly. Often times a function would not work due to one misplaced parenthesis or would not throw an error when both parenthesis were left out.

What did we dislike about Scheme:
The lack of uses of other functional characters (for example the curly brace) besides the parenthesis makes Scheme less clear to read. This causes an abundance of parenthesis in the easiest of tasks and made the entire project more difficult than necessary.

What did we like about Scheme:
While the tediousness of the language made us not enjoy it as much as other languages, scheme's focus on recursion and efficiency is something that made it easier to finish the project once we figured out how it worked.