```
#lang racket

;;;  Author:          Kong Jimmy Vang
;;;  File:            simplify.rkt
;;;  Project:         hw2 (Scheme Coding and Grammars - Expression Simplification)
;;;  Class:           CS 421
;;;  Date Created:    2/19/2021
;;;  Date Modified    2/28/2021

(define (atom? E) (and (not (null? E)) (not (pair? E))))

(define (error? E) (equal? E 'error))

(define (variable? E) (symbol? E))

(define (variable<? E1 E2) (symbol<? E1 E2))

(define (sum? E) (if (not (atom? E)) (eq? (car E) '+) #f))

(define (diff? E) (if (not (atom? E)) (eq? (car E) '-) #f))

(define (product? E) (if (not (atom? E)) (eq? (car E) '*) #f))

(define (quotient? E) (if (not (atom? E)) (eq? (car E) '/) #f))

(define (pick? E) (if (not (atom? E)) (eq? (car E) 'pick) #f))

;;; Compares two expressions.
;;; 1. For any simplified scheme expression involving either multiplication or
addition
;;;     a. If the operands are both variables, they are ordered alphabetically.
;;;     b. If the operands include a variable and a value, the variable precedes the
value.
;;;     c. If one operand is a sub-expression and the other is either a variable or
value,
;;;         the variable or value precedes the sub-expression.
;;;     d. If the operands are both sub-expressions, the ordering of the
sub-expressions
;;;         follows the ordering of their operand as given in: *, +, -, /, pick.
;;; 2. A fully simplified expression will have no sub-expression that can be
simplified by
;;;     application of one or more of the above rules.
(define (expression<? E1 E2)
  (cond ((and (variable? E1) (variable? E2) (variable<? E1 E2)) #t)
        ((and (variable? E1) (number? E2)) #t)
        ((and (number? E1) (list? E2)) #t)
        ((and (variable? E1) (list? E2)) #t)
        ((and (list? E1) (list? E2)) (or (symbol<? (car E1) (car E2)) (equal? (car
E1) (car E2))))
        (else #f)))

;(define (expression<? . EXPRS)
;  (cond ((null? (cdr EXPRS)) #t)
;        (else (and (less-than-helper? (car EXPRS) (cadr EXPRS)) (apply less-than?
(cdr EXPRS))))))

;(define (make-list OP E1 E2)
;  (cond ((or (error? E1) (error? E2)) 'error)
;        ((or (quotient? OP)
;             (diff? OP)) (list OP E1 E2))
;        ((less-than? E2 E1) (list OP E2 E1))
;        (else (list OP E1 E2))))

```

```
;(define (make-pick-list OP E1 E2 E3)
;  (list OP E1 E2 E3))

;;; Make a list and order it using the rules documented in the function
"expression<?".
(define (make-list OP . EXPRS)
  (cond ((or (diff? (list OP))
             (quotient? (list OP))
             (pick? (list OP))) (append (list OP) EXPRS))     ;; Do not Sort
        (else (append (list OP) (sort EXPRS expression<?))))) ;; Sort

;;; Simplify the sum of E1 and E2.
(define (make-sum E1 E2)
  (cond ((or (error? E1) (error? E2)) 'error)                ;; 12. (o X Y) → ERROR
if either X or Y is an ERROR
        ((and (number? E1) (number? E2)) (+ E1 E2))          ;; 11. (o X Y) → Z if X
and Y are numbers and Z is the result
        ((number? E2) (if (= E2 0) E1 (make-list '+ E1 E2))) ;; 1.  (+ X 0) → X for
any X
        ((number? E1) (if (= E1 0) E2 (make-list '+ E1 E2))) ;; 2.  (+ 0 X) → X for
any X
        (else (make-list '+ E1 E2)))) ;; ELSE return unsimplified.

;;; Simplify the difference of E1 and E2.
(define (make-diff E1 E2)
  (cond ((or (error? E1) (error? E2)) 'error)                ;; 12. (o X Y) → ERROR
if either X or Y is an ERROR
        ((and (number? E1) (number? E2)) (- E1 E2))          ;; 11. (o X Y) → Z if X
and Y are numbers and Z is the result
        ((equal? E1 E2) 0)                                   ;; 13. (- X X) → 0 for
any X other than ERROR
        ((number? E2) (if (= E2 0) E1 (make-list '- E1 E2))) ;; 7.  (- X 0) → X for
any X
        (else (make-list '- E1 E2)))) ;; ELSE return unsimplified.

;;; Simplify the product of E1 and E2.
(define (make-product E1 E2)
  (cond ((or (error? E1) (error? E2)) 'error)       ;; 12. (o X Y) → ERROR if either
X or Y is an ERROR
        ((and (number? E1) (number? E2)) (* E1 E2)) ;; 11. (o X Y) → Z if X and Y are
numbers and Z is the result
        ((number? E2) (cond ((= E2 0) 0)            ;; 3.  (* X 0) → 0 for any X
                            ((= E2 1) E1)           ;; 5.  (* X 1) → X for any X
                            (else (make-list '* E1 E2))))
        ((number? E1) (cond ((= E1 0) 0)            ;; 4.  (* 0 X) → 0 for any X
                            ((= E1 1) E2)           ;; 6.  (* 1 X) → X for any X
                            (else (make-list '* E1 E2))))
        (else (make-list '* E1 E2)))) ;; ELSE return unsimplified.

;;; Simplify the quotient of E1 and E2.
(define (make-quotient E1 E2)
  (cond ((or (error? E1) (error? E2)) 'error) ;; 12. (o X Y) → ERROR if either X or Y
is an ERROR
        ((and (equal? E1 E2)
              (not (equal? E1 0))
              (not (equal? E1 1))) 1)         ;; 10. (/ X X) → 1 for any X other than
ERROR and 0 and 1
        ((and (number? E1)
              (number? E2)
              (not (= E2 0))) (/ E1 E2))      ;; 11. (o X Y) → Z if X and Y are
numbers and Z is the result
        ((number? E2)
         (cond ((= E2 1) E1)                  ;; 8.  (/ X 1) → X for any X
```

```
106                ((= E2 0) 'error)              ;; 9.  (/ X 0) → ERROR for any X
107                (else (make-list '/ E1 E2))))
108          ((number? E1)
109           (cond ((and (= E1 0)
110                       (not (equal? E2 0))
111                       (not (equal? E2 1))) 0)  ;; 14.  (/ 0 X) → 0 for any X other
111 than 0 and 1
112                (else (make-list '/ E1 E2))))
113          (else (make-list '/ E1 E2)))) ;; ELSE return unsimplified.
114
115 ;;; Simplify the 'pick expression for E1 E2 E3.
116 (define (make-pick E1 E2 E3)
117   (cond ((error? E1) 'error)                   ;; 16.  (pick ERROR E2 E3) → ERROR
118         ((number? E1) (cond ((= E1 0) E2)      ;; 15.  (pick 0 E2 E3) → E2
119                             (else E3)))        ;; 17.  (pick X E2 E3) → E3 for any
119 non-zero number X
120         (else (make-list 'pick E1 E2 E3))))    ;; ELSE return unsimplified.
121
122 ;;; Accepts a scheme expression E and returns a new simplified scheme expression.
123 (define (simplify E)
124   (cond ((null? E ) 'error)
125         ((number? E) E)
126         ((variable? E) E)
127         ((sum? E) (make-sum (simplify (cadr E)) (simplify (caddr E))))
128         ((diff? E) (make-diff (simplify (cadr E)) (simplify (caddr E))))
129         ((product? E) (make-product (simplify (cadr E)) (simplify (caddr E))))
130         ((quotient? E) (make-quotient (simplify (cadr E)) (simplify (caddr E))))
131         ((pick? E) (make-pick (simplify (cadr E)) (simplify (caddr E)) (simplify
131 (cadddr E))))
132         (else E)))
133
134 (provide simplify)
```