Implementing a Language for Arithmetic Expressions

Here is a definition of a language for arithmetic expressions.

; a larger arithmetic language with two kinds of values, booleans and numbers
; an expression is any of these:

(struct const (int) #:transparent) ; int should hold a number

(struct negate (e1) #:transparent) ; e1 should hold an expression of type const

(struct add (e1 e2) #:transparent) ; e1, e2 should hold expressions of type const

(struct multiply (e1 e2) #:transparent) ; e1, e2 should hold expressions of type const

(struct bool (b) #:transparent) ; b should hold #t or #f

(struct eq-num (e1 e2) #:transparent) ; e1, e2 should hold expressions of type const

(struct if-then-else (e1 e2 e3) #:transparent) ; e1, e2, e3 should hold expressions, e1 must be type bool

1) Before we get too deep into evaluation, let's practice in LBI
   ● Define the negation of 2020
   
   ● Define the addition between 340 and 1
   
   ● Define an if statement that compares whether the constants 10 and 15 are equal and returns true if true and false if false

2) Below is a partial implementation of the interpreter. Fill in the implementation of the interpreter for the parts with TODO.
(define (eval-exp e)
  (cond [(const? e) e]
        [(bool? e) e]
        [(negate? e) ;; TODO: This branch
          ;; close negate?
          ] ;; close negate?
        [(add? e)
          (let ([v1 (eval-exp (add-e1 e))]
                [v2 (eval-exp (add-e2 e))])
            (if (and (const? v1) (const? v2))
                (const (+ (const-int v1) (const-int v2)))
                (error "add applied to non-number")))
        [(multiply? e)
          (let ([v1 (eval-exp (multiply-e1 e))]
                [v2 (eval-exp (multiply-e2 e))])
            (if (and (const? v1) (const? v2))
                (const (* (const-int v1) (const-int v2)))
                (error "multiply applied to non-number")))
        [(eq-num? e) ;; TODO: This branch
          ] ;; close eq-num? [(if-
          then-else? e) ;; TODO: This branch
          ] ;; close if-then-else?]

; not strictly necessary but helps debugging
[#t (error "eval-exp expected an exp")])))

Defining Macros in Racket for our Arithmetic Expression Language (AEL)

1) Define a Racket function orelse that takes two AEL expressions and returns an AEL expression that when run returns (bool #t) if at least one of given expressions are true, otherwise it returns (bool #f).
2) Define a Racket function `negative-square` which takes an AEL expression \( e \) and returns an AEL expression that when run evaluates to \(-e^2\).

3) Define a Racket function `abs-eq` that takes two AEL expressions and returns an AEL expression that when run returns \((\text{bool } \#t)\) if the two expressions have equal absolute values. (Hint: one of the previously defined macros might be useful for this problem)