CSE 341 | Section 8

Language Implementation and Macros (Solutions)

1) Before we get too deep into evaluation, let's practice in LBI

- Define the negation of 2020

(negate (const 2020))

- Define the addition between 340 and 1

(add (const 340) (const 1))

- Define an if statement that compares whether the constants 10 and 15 are equal and returns true if true and false if false

(if-then-else (eq-num (const 10) (const 15)) (bool #t) (bool #f))

Implementing a Language for Arithmetic Expressions

```
(define (eval-exp e)
   (cond [(const? e) e]
         [(negate? e)
            (let ([v (eval-exp (negate-e1 e))])
               (if (const? v)
                (const (- (const-int v)))
                (error "negate applied to non-number")))]
          [(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")))]
          [(bool? e) e]
          [(eq-num? e)
           (let ([v1 (eval-exp (eq-num-e1 e))]
                 [v2 (eval-exp (eq-num-e2 e))])
                (if (and (const? v1) (const? v2))
                   ; creates (bool #t) or (bool #f)
```

```
                (bool (= (const-int v1) (const-int v2)))
                (error "eq-num applied to non-number")))]
      [(if-then-else? e)
         (let ([v-test (eval-exp (if-then-else-e1 e))])
           (if (bool? v-test)
             (if (bool-b v-test)
                (eval-exp (if-then-else-e2 e))
                (eval-exp (if-then-else-e3 e)))
             (error "if-then-else applied to non-boolean")))]
; 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 (orelse e1 e2)
     (if-then-else e1 (bool #t) e2))

2) (define (abs-eq e1 e2)
     (orelse (eq-num e1 e2) (eq-num (negate e1) e2)))

3) (define (negative-square e)
     (negate (multiply e e)))
```