Section 7 - Implementing Interpreters, eval/quote/quasiquote

This handout was composed by Porter Jones. There are probably plenty of typos/incorrect solutions/etc for you to catch! Please email me with any issues, comments, or feedback at pbjones@cs.washington.edu. All thoughts are welcome :)

Implementing a Language for Arithmetic Expressions

At the end of the handout is a definition of a language for arithmetic expressions. 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?
        [(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?
        [#t (error "eval-exp expected an exp")] ; not strictly necessary but
helps debugging
        ))
```

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 `(bool #t)` if the two expressions have equal absolute values.

4) Explain the difference between a AEL language expression such as `multiply` and a Racket function that behaves as a AEL macro, such as `negative-square`. What is the AEL expression being evaluated in the function call `(eval (multiply (const -3) (const 4)))`? What is the AEL expression evaluated in the function call `(eval (negative-square (multiply (const -3) (const 4))))`?

eval, quote, and quasiquote

For the following Racket expressions, determine what happens if they were to be run in the REPL.

1) (car (list 1 2 (list 3 4) 5))

a) 1 b) 2 c) error

2) (car (quote (1 2 (3 4) 5))) ; remember ' is shorthand for quote

a) 1 b) 2 c) error

3) (quote (+ 1 (* 3 4)))

a) 13 b) '(+ 1 (* 3 4)) c)'(+ 1 12) d) error

4) (quasiquote (+ 1 (unquote (* 3 4))))

a) 13 b) '(+ 1 (* 3 4)) c)'(+ 1 12) d) error

5) (eval (quote (1 2 (+ 3 4) 5)))

a) '(1 2 (+ 3 4) 5) b) '(1 2 7 5) c) error

6) Given the following definition what is produced by `(powh 2)`? `(powh 3)`? `(powh n)`?

```
(define (powh y)
  (if (eq? y 1)
      (quote x)
      (quasiquote (* x (unquote (powh (- y 1)))))))
```

7) Given your answer from problem 5, describe what the following function returns:

```
(define (pow y)
  (let ([mul-exp (powh y)])
    (eval (quasiquote (lambda (x) (unquote mul-exp))))))
```

```
; For use in the first part of the handout
; 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
```

Section 7 - Solutions

This handout was composed by Porter Jones. There are probably plenty of typos/incorrect solutions/etc for you to catch! Please email me with any issues, comments, or feedback at pbjones@cs.washington.edu. All thoughts are welcome :)

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))
               (bool (= (const-int v1) (const-int v2))) ; creates (bool #t) or
(bool #f)
               (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")))]
        [#t (error "eval-exp expected an exp")] ; not strictly necessary but
helps debugging
        ))
```

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)))
```

4) AEL expressions are made from Racket structs, which means they have functions to construct them, fields, and ways to check their type. AEL macros made from Racket functions simply take an AEL expression and expand it into more AEL expressions, but they do not define a new expression in the AEL language. You could never call `(abs-eq? e)`, for example.

   `(multiply (const -3) (const 4))` is an AEL expression in itself, whereas `(negative-square (multiply (const -3) (const 4)))` is a function call in Racket that when evaluated returns/expands to the AEL expression

```
(negate (multiply (multiply (const -3) (const 4))
                  (multiply (const -3) (const 4))))
```

eval, quote, and quasiquote

1) a

2) a

3) b

4) c

5) error

6) Produces a nested multiplication of n x's. For example, `(powh 3)` produces `'(* x (* x x))`

7) Returns a function which when called with a value x produces the equivalent of x^y. Note that it does this without recursion thanks to the use of `powh` and `quasiquote/unquote`