Yotam Dikstein 201212198, Amir Arbel 302726310

1. Expression types:
   a. (- 3 7) - **Number**
   b. (lambda (x y) (< (+ x 1) y)) - **Closure [Number*Number -> Boolean]**
   c. (lambda (x) (display x) (* 2 x)) - **Closure [Number-> Number]**
   d. (procedure? 5) - **Boolean**
   e. ((lambda (x) (if (< x 0) 0 x)) 2) - **Closure [Number->Number] returns 2 (the first expression is evaluated but no returned)**

2. Evaluation results:

> (- (/ 0 1) (- 1 0))
-1

> (* 2 3 4)
24

> (+ z 1)
z: undefined;
 cannot reference an identifier before its definition
The error is given since z isn't defined in the global table.

> (define z 1)
(void)

> (lambda () (4 5))
#<procedure>
(closure is created [Empty->Error]; note if you'd try to run this procedure you will receive an error).

> ((lambda (x) ((display x) (* 2 x))) 5)
application: not a procedure;
 expected a procedure that can be applied to arguments
  given: #<void>
Error; (display x) returns a void, not a legal procedure, thus it can't be used on the number evaluated from (* 2 x).

> (+ z 1)
2
(z was previously defined).

> (< 8.1 2.3)
#f

3. a. (/ 1 0) - is correct but the procedure will give an error since zero is an illegal value to divide by.
b. (1 0) - is correct syntactically, but has no semantic meaning since 1 is not a procedure.

4.a.
> (cond ((< 3 3) #f)
((< 5 4) (and (< 5 1) (/ 1 0) #f))
(else (or #t (< (/ 1 0) 2))))

1. Read cond and evaluate it as special form (each condition is evaluated until one returns #t).
2. evaluate (< 3 3) - false.
3. evaluate (< 5 4) - false.
4. evaluate else - true (since all other conditions were false).
5. in the or part, since it is special form, each part is evaluated before the next, so #t is evaluated, and returned.

> (and #t (or (> 1 1) (lambda (x) (2 < 3))) (+ 3 9))

1. Read and and determine the procedure is special form, meaning conditions are evaluated one after the other until getting false (or finishing all conditions).
2. evaluate #t.
3. Read the or in it's section, and as above, go through all conditions until one that isn't false:
    a. evaluate (> 1 1) - false.
    b. evaluate the lambda (without actually running it), and since a closure isn't considered false, return it. the and then considers it also as not false.
4. evaluate (+ 3 9). Since 12 isn't false and this is the last predicate, return 12.

b. If (or #t (< (/ 1 0) 2)) was not in special form, then an unnecessary extra evaluation of (< (/ 1 0) 2) would occur and then since we cannot divide by zero, an error was generated.