3. (12 points) Expression Trees

Your partner has created an interpreter for a language that can add or multiply positive integers. Expressions are represented as instances of the `Tree` class and must have one of the following three forms:

- (**Primitive**) A positive integer `entry` and no branches, representing an integer
- (**Combination**) The `entry '+'`, representing the sum of the values of its branches
- (**Combination**) The `entry '*'`, representing the product of the values of its branches

The `Tree` class is on the Midterm 2 Study Guide. The sum of no values is 0. The product of no values is 1.

(a) (6 pt) Unfortunately, multiplication in Python is broken on your computer. Implement `eval_with_add`, which evaluates an expression without using multiplication. You may fill the blanks with names or call expressions, but the only way you are allowed to combine two numbers is using addition.

```
def eval_with_add(t):
    """Evaluate an expression tree of * and + using only addition.

    >>> plus = Tree('+', [Tree(2), Tree(3)])
    >>> eval_with_add(plus)
    5
    >>> times = Tree('*', [Tree(2), Tree(3)])
    >>> eval_with_add(times)
    6
    >>> deep = Tree('*', [Tree(2), plus, times])
    >>> eval_with_add(deep)
    60
    >>> eval_with_add(Tree('*'))
    1
    """
    if t.entry == '+':
        return sum(_____________________________________________________)
    elif t.entry == '*':

        total = _____________________________________________________

        for b in t.branches:

            total, term = 0, _____________________________________________

            for __________ in _____________________________________________:

                total = total + term

        return total
    else:
        return t.entry
```

(c) (4 pt) Implement the Scheme procedure `directions`, which takes a number `n` and a symbol `sym` that is bound to a nested list of numbers. It returns a Scheme expression that evaluates to `n` by repeatedly applying `car` and `cdr` to the nested list. Assume that `n` appears exactly once in the nested list bound to `sym`.

Hint: The implementation searches for the number `n` in the nested list `s` that is bound to `sym`. The returned expression is built during the search. See the tests at the bottom of the page for usage examples.

```
(define (directions n sym)

    (define (search s exp)

        ; Search an expression s for n and return an expression based on exp.

        (cond ((number? s) ____________________________________________________)

              ((null? s) nil)

              (else (search-list s exp))))

    (define (search-list s exp)

        ; Search a nested list s for n and return an expression based on exp.

        (let ((first ____________________________________________________________)

              (rest  ____________________________________________________________))

             (if (null? first) rest first)))

    (search (eval sym) sym))

(define a '(1 (2 3) ((4))))
(directions 1 'a)
; expect (car a)
(directions 2 'a)
; expect (car (car (cdr a)))
(define b '((3 4) 5))
(directions 4 'b)
; expect (car (cdr (car b)))
```

6. (10 points) Pair Emphasis

(a) (6 pt) Implement `parens` by crossing out whole lines below. It takes a Scheme value and returns the number of parentheses required to write the value in standard Scheme notation.

```
; Return the number of parentheses in s.
;
; (parens 3)            -> 0
; (parens (list 3 3))   -> 2
; (parens '(3 . 3))     -> 2
; (parens '(3 . (3)))   -> 2  because (3 . (3)) simplifies to (3 3)
; (parens '((3)))       -> 4
; (parens '(()))        -> 4
; (parens '((3)((3)))) -> 8
(define (parens s)
    (f s 0))
    (f s 2))
    (f s #f))
    (f s #t))
(define (f s t)
    (cond ((pair? s) (+
           1
           2
           t
           (if t 1 0)
           (f (car s) 0)
           (f (cdr s) 0)
           (f (car s) 2)
           (f (cdr s) 2)
           (f (car s) #t)
           (f (cdr s) #f)
           (f (car s) t)
           (f (cdr s) t)
          ))
          ((null? s)
           0
           1
           2
           t
           (if t 1 0)
          )
          (else 0)))
```

(b) (2 pt) Write a quote expression that evaluates to the Scheme list `(1 (2) 3)` that has as many parentheses as possible in the expression. For example, a well-formed (but incorrect) answer is `(quote (1 (2) 3))`.

```
(quote ____________________________________________________________________)
```