```
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([eval_with_add(b) for b in t.branches])

    elif t.entry == '*':

        total = 1


        for b in t.branches:


            term, total = total, 0


            for _ in range(eval_with_add(b)):


                total = total + term

        return total

    else:

        return t.entry
```

```
(define (directions n sym)

    (define (search s exp)

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

        (cond ((number? s) (if (= s n) exp nil))

              ((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 (search (car s) (list 'car exp)))

              (rest  (search (cdr s) (list 'cdr exp))))

             (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)))
; 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 2))
(define (f s t)
    (cond ((pair? s) (+
            t
            (f (car s) 2)
            (f (cdr s) 0)))
          ((null? s) t)
          (else 0)))
```

```
(quote (1 . ((2 . ()) . (3 . ()))))
```