Lab 5 – (Recursive (Practice (Practice (Practice))))

1. Write the function **`laugh`**, which takes a number as input and returns a list containing that many `ha` symbols. Your function should behave as shown in the examples below:

```
(laugh 4) → (ha ha ha ha)
(laugh 0) → ()
(laugh 1) → (ha)
```

2. Write the function **`(concat`** *`list1 list2`*`)`, which takes two input lists and returns the concatenation of the lists — that is, a new list containing all of *list1*'s elements followed by *list2*'s elements, without any inner parentheses.

```
(concat '(a b c) '(d e f g)) → (a b c d e f g)
(concat '() '(x y z)) → (x y z)
(concat '(1 2 3 4 5 6 7 8) '(nine ten eleven)) → (1 2 3 4 5 6 7 8 nine ten eleven)
```

3. Write the function **`(times-ten`** *`numbers`*`)`, which takes a list of numbers as input, and returns a new list containing each number from the original list multiplied by 10.

```
(times-ten '(1 2 3 4 5)) → (10 20 30 40 50)
(times-ten '()) → ()
(times-ten '(7)) → (70)
```

4. Write the function **`x-odds`**, which takes a list of numbers as input, and returns a new list with all of the odd numbers replaced by the literal symbol `x`.

```
(x-odds '(1 2 3 4 5 9)) → (x 2 x 4 x x)
(x-odds '(2 4 6)) → (2 4 6)
(x-odds '(1 3 5)) → (x x x)
```

5. Write the function **`classify-nums`**, which takes a list of numbers as input, and returns a new list with all odd numbers replaced by the symbol `odd` and all even numbers replaced by the symbol `even`.

```
(classify-nums '(1 2 3 4 5)) → (odd even odd even odd)
(classify-nums '(7 7 7 9)) → (odd odd odd odd)
(classify-nums '(8)) → (even)
```

6. In class on Tuesday, we wrote the functions `remove-one` and `subst-one`. Using these functions as a guide, write the function **`insertL-one`**, which takes a *new* symbol, an *old* symbol, and an input list, and inserts the new symbol *to the left* of the first occurrence of the old symbol in the input list.

```
(insertL-one 'frog 'x '(a b c x d x x e)) → (a b c frog x d x x e)
(insertL-one 'frog 'e '(a b c x d x x e)) → (a b c x d x x frog e)
(insertL-one 'frog 'x '()) → ()
(insertL-one 'frog 'x '(a b c d)) → (a b c d)
```

7. Write the function **`insertR-one`**, which takes a *new* symbol, an *old* symbol, and an input list, and inserts the new symbol *to the right* of the first occurrence of the old symbol in the input list.

```
(insertR-one 'frog 'x '(a b c x d x x e)) → (a b c x frog d x x e)
(insertR-one 'frog 'e '(a b c x d x x e)) → (a b c x d x x e frog)
(insertR-one 'frog 'x '()) → ()
(insertR-one 'frog 'x '(a b c d)) → (a b c d)
```

8. We also wrote the functions `remove-all` and `subst-all`. Using these functions as a guide, write the function **`insertL-all`**, which takes a *new* symbol, an *old* symbol, and an input list, and inserts the new symbol to the left of *every occurrence* of the old symbol in the input list.

```
(insertL-all 'frog 'x '(a b c x d x x e)) → (a b c frog x d frog x frog x e)
(insertL-all 'frog 'e '(a b c x d x x e)) → (a b c x d x x frog e)
(insertL-all 'frog 'x '()) → ()
(insertL-all 'frog 'x '(a b c d)) → (a b c d)
```

9. Write the function **`insertR-all`**, which takes a *new* symbol, an *old* symbol, and an input list, and inserts the new symbol to the right of *every occurrence* of the old symbol in the input list.

```
(insertR-all 'frog 'x '(a b c x d x x e)) → (a b c x frog d x frog x frog e)
(insertR-all 'frog 'e '(a b c x d x x e)) → (a b c x d x x e frog)
(insertR-all 'frog 'x '()) → ()
(insertR-all 'frog 'x '(a b c d)) → (a b c d)
```

10. Write the function **`every-other`**, which takes an input list and returns a new list containing every other element of the original list. Hint: you will need to check for two different base cases.

```
(every-other '(a b c d e f g)) → (a c e g)
(every-other '(a b c d e)) → (a c e)
(every-other '(a b c d)) → (a c)
(every-other '(a)) → (a)
```

11. Write the function (**`zip`** *`list1 list2`*), which takes two input lists of the same length and forms a new *list of lists* by combining the corresponding elements of *list1* and *list2* into "pairs", as shown below.

```
(zip '(1 2 3 4) '(a b c d)) → ((1 a) (2 b) (3 c) (4 d))
(zip '(a) '(b)) → ((a b))
(zip '() '()) → ()
```

12. Write the function (**`pair-up`** *`symbol input-list`*), which takes a symbol and a list as input and creates a new list consisting of the symbol paired up with each element of the input list, as shown below.

```
(pair-up 'x '(a b c d e)) → ((x a) (x b) (x c) (x d) (x e))
(pair-up 'nothing '()) → ()
(pair-up 'thing '(one two)) → ((thing one) (thing two))
(pair-up 'hee '(hee)) → ((hee hee))
```

13. Write the function (**`cross-product`** *`list1 list2`*), which takes two input lists and returns all elements of *list1* paired up with all elements of *list2* as shown in the examples below. Hint: use your `pair-up` and `concat` functions as helpers.

```
(cross-product '(a b c) '(1 2)) → ((a 1) (a 2) (b 1) (b 2) (c 1) (c 2))
(cross-product '(a b c) '()) → ()
(cross-product '() '(a b c)) → ()
```

14. Write the function **`increasing-order?`**, which takes a list of numbers and returns `#t` if all numbers in the list are in increasing order from left to right, or `#f` otherwise. Adjacent numbers that are equal should be considered to be in "increasing" order.

```
(increasing-order? '(1 2 3 4 7 9)) → #t
(increasing-order? '(1 1 1 5 5 8)) → #t
(increasing-order? '(7 9 8 10 12)) → #f
```