display(v)
display(v,s) → 's'+' '+'v'
error(v) → `Line 1: Error: "${v}"`
error(v,s) → `Line 1: Error: ${s} ${v}`

is_stream
math_max
math_min

is_boolean — math_abs — math_round — math_sqrt
is_function — math_sin — math_PI
is_number — math_cos — math_pow(base, exponent)
is_string, is_pair — math_log — math_max
is_undefined — math_log2 — math_min — math_hypot (4 calc. hypotenuse)

parse_int(s, i) → interpret string s with radix i
prompt(s)
stringify(v) → to_string function
math_trunc → integral part

build_list(n, fun) { ~ build_stream — weak build defect. [Yes]
function build(i, fun, alr_built) {
return i<0 ? alr_built : build(i-1, fun, pair(fun(i), alr_built));
} build: i ⇒ i >= n ? null : pair(fun(i), () ⇒ build(i+1)) } stream
return build(n-1, fun, null); build(0)
}

enum_list(start, end) { ~ enum_stream [Yes]
return start > end ? null : pair(start, enum_list(start+1, end));
} build_list(b-a+1, x → x+a)

equal(x,y) {
return is_pair(x) && is_pair(y) ? equal(head(x), head(y)) && equal(tail(x), tail(y)) : x === y }

filter(pred, xs) { O(n), O(n).
return is_null(xs) ? xs : pred(head(xs)) ? pair(head(xs), filter(pred, tail(xs))) : filter(pred, tail(xs)); }

length(xs) { O(n), O(1) ~ stream_length [No]
return is_null(xs) ? 0 : 1 + length(tail(xs)); }

list_ref(xs, n) { O(n), O(1). ~ stream_ref [sort-of]
return n === 0 ? head(xs) : list_ref(tail(xs), n-1))

list_to_string(xs) → text-based box n pointer.
return is_null(xs) ? "null" : is_pair(xs) ? "[" + LTS(head(xs)) + "," + LTS(tail(xs)) + "]" : stringify(xs)

map(f, xs) { O(n), O(n).
return is_null(xs) ? null : pair(f(head(xs)), map(f, tail(xs)));
}

member(v, xs) { O(n), O(1) ~ stream_member [sort_of]
return is_null(xs) ? null : v === head(xs) ? xs : member(v, tail(xs));
}

thrice:
t(t(t(t(f)))) = 16
t(t(t(t(t(f))))) = 256
t(t(t(t(t))))(f) = 256
t(t(t(t)))(t(f)) = 256 = 4^4
t(t(t(t)(t)))(f) = 65536

t(t(t(f))) = 16
t(t(t(t)(f))) = 16
t(t)(t(t(f))) = 32
t(t)(t(t)(f)) = 42

TWICE
t(t(f)) = 4
t(t)(f) = 4
t(t)(t)(f) = 3^3
t(t(t(f))) = 2^3

○ DON'T BREAK THE ABSTRACTION.

○ If check for lists, don't forget 'pair' case.

○ x === y checks if object is the same (same reference)
equal(x,y) only check if the content of x & y are the same.

○ Box pointer beware!

for_each(fun, xs) { if (is_null(xs)) { return true; } else { fun(head(xs)); return for_each(fun, tail(xs)); }
O(n) O(1) iterative. ~ stream_for_each (NO)

function: ① params ② body ③ env } compound
① params ② body } defn.
apply ← argument, fn_body — local names
evaluate(body, env) ← values
conds: ① predicate ② cons ③ alt.
assignment / const/var ① [name, "the name"] ② value.
application ① operator ② operands
lookup assign set.
enclosing env, first frame (head), tail
each frame — head: names; tail: values ← each (value, tag).

stream_to_list: xs ⇒ is_null(xs) ? null : pair(head(xs), stream_to_list(stream_tail(xs))); [NO]
list_to_stream = xs ⇒ is_null(xs) ? null : pair(head(xs), () ⇒ list_to_stream(tail(xs))); [YES]
stream_filter = (p, s) → is_null(s) ? null : p(head(s)) ? pair(head(s), () ⇒ stream_filter(p, stream_tail(s))) : stream_filter(p, stream_tail(s)) [Yes]
stream_map = (f, s) ⇒ is_null(s) ? null : pair(f(head(s)), () ⇒ stream_map(f, stream_tail(s))); [Yes]
stream_append = (xs, ys) → { is_null(xs) ? ys : pair(head(xs), () ⇒ stream_append(stream_tail(xs), ys)); [Yes]
eval_stream: (s, n) ⇒ n === 0 ? null : pair(head(s), eval_stream(stream_tail(s), n-1)); [sort_of]

add_index = build_list(length(xs), x ⇒ pair(list_ref(xs, x), x));
integers_from: n ⇒ pair(n, () ⇒ integers_from(n+1)) [Yes]
→ return (tail(xs), head(xs), null)

find_min(xs) { helper(lst, smallest, built) {
return is_null(lst) ? pair(smallest, built) : head(lst) < smallest ? pair(tail(lst), head(lst), pair(smallest, built)) : pair(tail(lst), smallest, pair(head(lst), built))

traverse(xs) → disp every item in tree
return if xs === null { display(null) } (!is_list(head(xs)) ? display(head(xs)) : traverse(head(xs))) return traverse(tail(xs));

reverse(xs) { O(n), O(n) ~ stream_reverse [NO]
func rev(ori, reversed) { return is_null(original) ? reversed : rev(tail(ori), pair(head(ori), reversed)) } return rev(xs, null);
}

remove_all(v, xs) { O(n), O(n) ~ stream_remove_all Lazy
return is_null(xs) ? null : v === head(xs) ? remove_all(v, tail(xs)) : pair(head(xs), remove_all(v, tail(xs)));
}

remove(v, xs) { O(n), O(n) ~ stream_remove [Lazy]
return is_null(xs) ? null : v === head(xs) ? tail(xs) : pair(head(xs), remove(v, tail(xs)));
}

LIST METHODS

accum_right(f, init, xs) – like accumulate, but folds from the left instead of right. [list → object]
```
function accum_left(f, init, xs) {
    return is_null(xs)
        ? init
        : f(head(xs), accum_left(f, init, tail(xs)));
}
```

all_diff(xs) – returns true if all elements of a list are different, and false otherwise. [list → bool]
```
function all_diff(xs) {
    return is_null(xs)
        ? true
        : is_null(member(head(xs), tail(xs))) &&
all_diff(tail(xs));
}
```

all_same(xs) – returns true if all elements of a list are the same, and false otherwise. [list → bool]
```
function all_same(xs) {
    return length(xs) < 2
        ? true
        : head(xs) === head(tail(xs)) &&
all_same(tail(xs));
}
```

binary_max(pred, low, high) – given a *continuous* predicate and lower/upper bounds, finds the max n within the boundaries such that pred(n) is true, in O(NlogN). returns NaN if pred(n) is false for all n in the bounds. [function, num, num → num]
```
function binary_max(pred, low, high) {
    function midpoint(a, b) {
        return (a + b) / 2;
    }
    const tol = 0.0000001; // max error tolerance
    const mid = midpoint(low, high);
    if (!pred(low)) {
        return NaN;
    } else if (high - low < tol) {
        return low;
    } else if (pred(mid)) {
        return binary_max(pred, mid, high);
    } else {
        return binary_max(pred, low, mid);
    }
}
```
Note: to change to binary_min(), change !pred(low) to !pred(high) and swap the two continuity statements.

count(n, xs) – returns the number of times n appears in xs. [object, list → num]
```
function count(n, xs) {
    return length(filter(x => equal(x, n), xs));
}
```

index_in(n, xs) – gives the index of the first appearance of n in xs, or NaN if n is not in xs. [object, list → num]
```
function index_in(n, xs) {
    return is_null(member(n, xs))
        ? NaN
        : length(xs) - length(member(n, xs));
}
```

linear_max(pred, low, high) – given a predicate and lower/upper bounds, finds the max *integer* n within the boundaries such that pred(n) is true. returns NaN if pred(n) is false for all n in the bounds. [function, num, num → num]
```
function linear_max(pred, low, high) {
    if (pred(high)) {
        return high;
    } else if (low === high) {
        return NaN;
    } else {
        return linear_max(pred, low, high - 1);
    }
}
```
Note: to change to linear_min(), change pred(high) to pred(low) and change the continuity statement to return linear_max(pred, low + 1, high).
Note: 1 can be substituted for some other number for higher or lower precision.

max(xs) – returns the largest value in xs. [list → object]
```
function max(xs) {
    return accumulate((x, y) => x > y ? x : y,
-Infinity, xs);
}
```

min(xs) – returns the smallest value in xs. [list → object]
```
function min(xs) {
    return accumulate((x, y) => x > y ? y : x, Infinity,
xs);
}
```

perms(xs) – returns ordered permutations of xs. [list → list]
```
function perms(xs) {
    return is_null(xs)
        ? list(null)
        : accumulate(append, null, map(n => map(p =>
pair(n, p), perms(remove(n, xs))), xs));
}
```

rank_list(xs) – given list, returns list of each item's rank in a descending-order list. [list → list]
```
function rank_list(xs) {
    return map(x => length(filter(y => y < x, xs)) +
1, xs);
}
```

subsets(xs) – returns all subsets of xs. [list → list]
```
function subsets(xs) {
    return accumulate((x, ss) => append(ss, map(s =>
pair(x, s), ss)), list(null), xs);
}
```

zip(f, xs, ys) – for lists xs and ys of equal length, returns a list zs such that zs[n] = f(xs[n], ys[n]). [function, list, list → list]
```
function zip(f, xs, ys) {
    return is_null(xs)
        ? null
        : pair(f(head(xs), head(ys)),
               zip(f, tail(xs), tail(ys)));
}
```

SORTING ALGORITHMS

SELECTION SORT: at each step, remove the smallest number from of the list, sort the rest of the list, and replace the smallest number at the front.
Best case: O(N²); worst case: O(N²); average case: O(N²)
```
function selection_sort(xs) {
    function min(xs) {
        return accumulate((x, y) => x > y ? y : x,
                          Infinity, xs);
    }
    return is_null(xs)
        ? null
        : pair(min(xs),
          selection_sort(remove(min(xs), xs)));
}
```

MERGE SORT: at each step, split the list into two halves and sort each half of the list. Then merge the two lists into a sorted list.
Best case: O(NlogN); worst case: O(NlogN), average case: O(NlogN)
```
function merge_sort(xs) {
    function mid(a, b) {
        return math_floor((a + b) / 2);
    }
    function take(xs, n) {
        return n === 0
            ? null
            : pair(head(xs), take(tail(xs), n - 1));
    }
    function drop(xs, n) {
        return n === 0
            ? xs
            : drop(tail(xs), n - 1);
    }
    function merge(xs, ys) {
        if (is_null(xs)) {
            return ys;
        } else if (is_null(ys)) {
            return xs;
        } else {
            const x = head(xs); const y = head(ys);
            return x < y
                ? pair(x, merge(tail(xs), ys))
                : pair(y, merge(xs, tail(ys)));
        }
    }
    if (is_null(xs) || is_null(tail(xs))) {
        return xs;
    } else {
    const n = math_floor(length(xs) / 2);
    const listA = take(xs, n);
    const listB = drop(xs, n);
    return merge(merge_sort(listA),
                 merge_sort(listB));
    }
}
```

INSERTION SORT (SPECIALIZED VERSION): at each step, take the head of the list and put it into the correct position in a holder list. At the end, return a holder list.
The notable thing here is that the comparison cmp isn't immediately specified, so it can be edited according to how the list should be sorted. For example:
- Ascending order: (x, y) => x < y
- Descending order: (x, y) => x > y
- Reverse original order: (x, y) => false

Best case: O(N), worst case: O(N²); average case: O(N²)
```
function insert_cmp(x, xs, cmp) {
    return is_null(xs)
        ? list(x)
        : cmp(x, head(xs))
            ? pair(x, xs)
            : pair(head(xs),
                   insert_cmp(x, tail(xs), cmp));
}
function insertion_sort_cmp(xs, cmp) {
    return is_null(xs)
        ? xs
        : insert_cmp(head(xs),
            insertion_sort_cmp(tail(xs), cmp),
            cmp);
}
```

QUICKSORT: at each step, choose the head of the list as a "pivot". Partition the rest of the list into two sublists based on whether they are smaller/equal to (<=) or larger than (>) the pivot. Then sort each sublist and combine small + pivot + large.
Best case: O(NlogN); worst case: O(N²); average case: O(NlogN)
```
function partition(xs, p) {
    return is_null(xs)
        ? pair(null, null)
        : pair(filter(x => x <= p, xs),
               filter(x => x > p, xs));
}

function quicksort(xs) {
    if (is_null(xs) || is_null(tail(xs))) {
        return xs;
    } else {
        const parted_lists = partition(tail(xs),
head(xs));
        return append(quicksort(head(parted_lists)),
pair(head(xs), quicksort(tail(parted_lists))));
    }
}
```

TREE METHODS

flatten(tree) – given a tree, returns a list containing all items from the tree in order. [tree → list]
```
function flatten(tree) {
    if (is_null(tree)) {
        return null;
    } else if (!is_list(head(tree))) {
        return pair(head(tree),
                    flatten(tail(tree)));
    } else {
        return append(flatten(head(tree)),
                      flatten(tail(tree)));
    }
}
```

map_tree(tree) – like map but works on trees. [tree → tree]
```
function map_tree(f, tree) {
    return map(s => !is_list(s) ? f(s) :
map_tree(f, s), tree);
}
```

accum_tree(tree) – like accumulate but works on trees. [tree → object]
```
function accum_tree(f, init, tree) {
    return accumulate((x, y) => is_list(x) ?
accum_tree(f, y, x) : f(x, y), init, tree);
}
```