a && b → a ? b : false
a || b → a ? true : b

Code stops evaluating when false && (premise) or true || (premise)
→ CS1231S: F ∧ p ≡ F , T ∨ p ≡ T
→ Usually use it when base case is true/false.

```
function all_different (xs) {
  return is_null(xs)
    ? true
    : is_null(member(head(xs), tail(xs))) && all_different (tail(xs));
}
```

CS1101S Mid-Terms Cheat Sheet

Lists	Trees
- Handle null - Operate on head: data item - Wish for tail	- Handle null - Operate on head > Data item > Tree - Wish for tail

```
const R = list(list(1,2,3), list(4,5,6), list(7,8,9))
list_ref(R,1) = list(4,5,6)
list_ref(list_ref(R,1),1) = 5
```

reverse(R) = list(list(7,8,9), list(4,5,6), list(1,2,3))
map(reverse, R) = list(list(3,2,1), list(6,5,4), list(9,8,7))
reverse(map(reverse, R)) = list(list(9,8,7), list(6,5,4) list(3,2,1))

*<u>Applicative Order</u>: Evaluate arguments, then apply fn
*<u>Normal Order</u>: Evaluate fn, then apply fn onto arguments

Order of Growth

<u>Time</u>
Time taken for the program to run/ number of steps.

<u>Space</u>
Memory taken for the program:
- Max number of deferred operators used
- How many times the function calls itself to complete the program

- Big O (Upper Bound/ Worst Case)
- Big Θ (Tight Bound/ Average Case)
- Big Ω (Lower Bound/ Best Case)

- For $O(n^2)$, we are just saying that the worst case is n^2. But it could be the case where its actually O(n), O(1), O(nlogn) etc (less than $O(n^2)$)
- If we use $\Theta(n^2)$, it means that the order or growth is strictly n^2
- Then for $\Omega(n^2)$ it means that the best case is n^2, but there could be instances where the OOG is larger, e.g $\Omega(n^3)$, $\Omega(2^n)$

*Ignore constant terms & minor terms
- If f(n) has order of growth Θ(g(n)) then f(n) has order of growth Ω(g(n))
- If f(n) has order of growth Θ(g(n)) then f(n) has order of growth O(g(n))

Common Recurrence Relations:

<u>Generally:</u> $T(n) = T(n-1) + O(n^k)$
$T(n) = O(n^{k+1})$

O(n)	
$T(n) = T(n-1) + O(1)$ → $T(n) = O(n)$	$T(n) = T(n/2) + O(n)$ → $T(n) = O(n)$
$T(n) = 2T(n/2) + O(1)$ → $T(n) = O(n)$	
O(log n)	**$O(n \log^2 n)$**
$T(n) = T(n/2) + O(1)$ → $T(n) = O(\log n)$	$T(n) = 2T(n/2) + O(n \log n)$ → $T(n) = O(n \log^2 n)$
O(n log n)	
$T(n) = T(n-1) + O(\log n)$ → $T(n) = O(n \log n)$	$T(n) = 2T(n/2) + O(n)$ → $T(n) = O(n \log n)$
$O(n^2)$	
$T(n) = T(n-1) + O(n)$ → $T(n) = O(n^2)$	
$O(2^n)$	
$T(n) = 2T(n-1) + O(1)$ → $T(n) = O(2^n)$	$T(n) = T(n-1) + T(n-2) + O(1)$ → $T(n) = O(2^n)$

IMPT Functions

<u>Map:</u>
```
function map(f, xs) {
    return is_null(xs)
        ? null
        : pair(f(head(xs)), map(f, tail(xs)));
}
```

<u>Accumulate:</u>
```
function accumulate(f, initial, xs) {
    return is_null(xs)
        ? initial
        : f(head(xs), accumulate(f, initial,
               tail(xs)));
}
```
For bst, change
Is_null to
is_empty_binary_tree
head→ left
tail→ right

```
function append(xs, ys){
  if (is_emty_list(xs)) {return ys;}
  else{ return pair(head(xs), append(tail(xs), ys));
}
```

head → (a,b) => a
tail → (a,b) => b

Flatten_List
```
function flatten_list(lst){
    return accumulate(append, null, lst);
}
```

Filter
```
function filter(pred, xs) {
    return is_null(xs)
        ? null
        : pred(head(xs))
        ? pair(head(xs), filter(pred, tail(xs)))
        : filter(pred, tail(xs));
}
```

Tree_Sum
```
function tree_sum(tree) {
    return accumulate_tree(x => x,
                           (x, y) => x + y, 0 , tree);
}
```

Accumulate_Tree
```
function accumulate_tree(f, op, initial, tree) {
    function accum(x,y) {
        return is_list(x)
            ? accumulate_tree(f, op, y, x)
            : op(f(x),y);
    }
    return accumulate(accum, initial, tree);
}
```

Map_Tree
```
function map_tree(f, tree) {
    return map(sub_tree => !is_list(sub_tree)
        ? f(sub_tree)
        : map_tree(f, sub_tree) , tree);
}
```

Count_Data_Item(Tree)
```
function count_data_items(tree) {
    return accumulate_tree(x => 1,
                           (x, y) => x + y, 0 , tree);
}
```

Scale_Tree
```
function scale_tree(tree, k) {
    return map_tree(data_item =>
               data_item * k,
           tree);
}
```

Count Coins
```
function cc(amount , kinds_of_coins ) {
    return amount === 0
        ? 1
        : amount < 0 || kinds_of_coins === 0
        ? 0
        : cc( amount - first_denomination
              (kinds_of_coins), kinds_of_coins )
          + cc( amount , kinds_of_coins - 1);
}
```

<u>List:</u>
A list is either null or a pair whose tail is a list

A list of a certain data type is null or a pair whose head is of that data type and whose tail is a list of that data type

<u>Tree</u>
- A tree of a certain data type is a list whose elements are of that data type, or trees of that data type
- A tree is a list whose elements are data items, or trees
  Caveat: Cannot consider null & pair as "certain data type"
  So, we cannot have trees of nulls and trees of pairs

<u>Binary Tree (BT)</u>
A **BT** is either an **empty tree**, or it has
- an **entry** (which is the data item)
- a **left branch/subtree** (which is a BT)
- a **right branch/subtree** (which is a BT)

<u>Binary Search Tree(BST)</u>
A **BST** is a binary tree where
- all entries in the **left subtree** < entry, and
- all entries in the **right subtree** > entry
- A BST is an abstraction for binary search

<u>Find (BST)</u>
```
function find(bst, name) {
  if (is_empty_tree(bst)){
    return false;
  }
  else if (name > entry(bst)){
    return find(right_branch(bst), name);
  }
  else if (name < entry(bst)){
    return find(left_branch(bst), name);
  }
  else {
    return name === entry(bst) ? true : false;
  }
}
```
Right: head(tail(tail(bst)));
Left: head(tail(bst));

** entry = value_of(bst)

<u>Insert (BST)</u>
```
function insert(bst, item) {
  if (is_empty_tree(bst)){
    return make_tree(item, null, null);
  }
  else if (item < entry(bst)){
    return make_tree(entry(bst),
               insert(left_branch(bst),item), right_branch(bst));
  }
  else if (item === entry(bst)){
    return insert(entry(bst), item);
  }
  else if (item > entry(bst)){
    return make_tree(entry(bst), left_branch(bst),
                insert(right_branch(bst), item));
  }
  else {
    return null;
  }
}
```

accumulate_BST
```
function accumulate_bst(op, initial, bst){
   if (is_empty_binary_tree(bst)) { return initial;}
   else{ const s = accumulate_bst(op, initial, right_subtree_of(bst);
        const t = op(value_of(bst), s);
        return accumulate_bst(op, t, left_subtree_of(bst));
```

How to Draw Box-Pointer Diagrams?

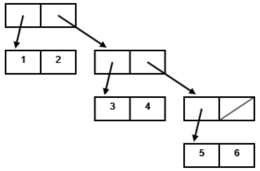
Tip:
- Count the number of elements
- Count the number of elements in the sub-list
- Draw the skeleton of the box & pointer diagram first, then input the details.

<u>Box Notation</u>
- Pair(x, y) is printed as [x, y]
- Empty lists are printed as nulls
- Pair(1, pair(2, pair(3, null))); → [1, [2, [3, null]]]

<u>List Notation</u>
- Same as box notation, but any sub-structure that is a list is nicely formatted and printed as list(...)
- E.g display_list(pair(pair(7, 8), pair(1, pair(2, null))),6)); → [list([7, 8], 1, 2), 6]

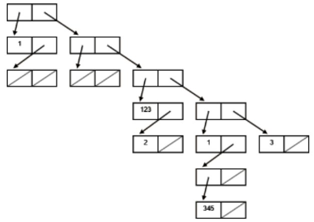
The "last-pair-box" will always be null

list(pair(1,2), pair(3,4), pair(5,6));



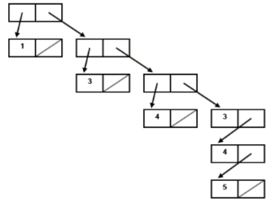
[[1, 2], [[3, 4], [[5, 6], null]]]

list(list(1,null), list(null), list(123,2),list(1, list(345)), 3);

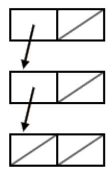
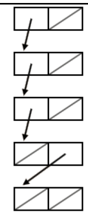


[[1, [null, null]],[[null, null], [[123, [2, null]], [[1, [[345, null], null]], [3, null]]]]]

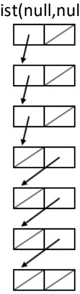
list(list(1), list(3), list(4),3,4,5);



[[1, null], [[3, null], [[4, null], [3, [4, [5, null]]]]]]

list(list(list(list())));	list(list(list(list(null,null))));
 [[[null, null], null], null]	 [[[[null, [null, null]], null], null], null]

list(list(list(list(null,null,null,null))));



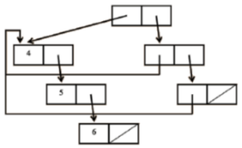
[[[[null, [null, [null, [null, null]]]], null], null], null]

The head of the list will all point to the head. Anything after the head, separted by "," will cause a new box to be drawn

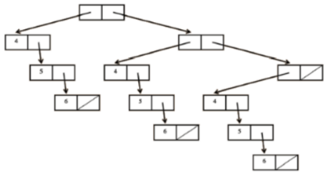
For qns that ask you to draw a box & pointer diagram with map/accumulate in the list, evaluate the code first, then draw the diagram ☹ they want to kill us. NEED TO KNOW HOW THE FUNCTION WORKS!

Tricky:

const ys = list(4, 5, 6);
const x = map(x => ys, ys);

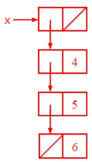


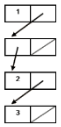
OR



[[4, [5, [6, null]]], [[4, [5, [6, null]]], [[4, [5, [6, null]]], null]]]

const x =
accumulate((x, ys) => map(y => pair(y, x), ys),
list(null), list(4, 5, 6));





list(1, list(2,3));
[1, [[2, [3, null]], null]]

[[[[null, 6], 5], 4], null]

RECURSIVE vs ITERATIVE

RECURSIVE:

- Repeat function call until basecase
- Any function that calls itself (directly or indirectly)
- Number of Deferred Operators Accumulates (Increases)
- **For recursion functions, if the execution of the recursion function call is not the only and last step, all of the other steps have to wait for it, then they will become deferred operations.
- Basecase, Scale, Sub-Problems

ITERATIVE:

- Loops until condition is met
- Constant Deferred Operators/No Accumulation of deferred operators.

E.g
```
function fib(n) {
  return fib_iter (1, 0, n);
}
function fib_iter (a, b, count ) {
  return count === 0
    ? b
    : fib_iter (a + b, a, count - 1);
}
```

```
function sum_of_list (xs) {
  function sum(to_process , total_sum) {
    return is_null(to_process)
      ? total_sum
      : sum(tail(to_process ),
          total_sum + head( to_process ));
  }
  return sum(xs , 0);
}
```

Sorting

Insert Sort (Time Best: Θ(n), Worst: Θ(n²), Space: Θ(1))

- Sort the tail of the given list using inductive hypothesis(wishful thinking)
- Insert the head in the right place
- Worst case: Ordered list (ascending order)

```
function insert(x, xs) {
return is_null(xs)
? list(x)
: x <= head(xs)
? pair(x, xs)
: pair(head(xs), insert(x, tail(xs)));
}
function insertion_sort(xs) {
return is_null(xs)
? xs
: insert(head(xs), insertion_sort(tail(xs)));
}
```

Selection Sort (Time Θ(n²), Space: Θ(1))

- Find the smallest element x & remove it from list
- Sort the remaining list, and put x infront

```
function smallest(xs) {
return accumulate((x, y) => x < y ? x : y,
                  head(xs), tail(xs));
} // To find the smallest element of non-empty list
function selection_sort(xs) {
if (is_null(xs)) {
return xs;} else {
const x = smallest(xs);
return pair(x, selection_sort(remove(x, xs)));
  } }
```

Merge Sort (Time Θ(n log n), Space: Θ(n))

- Split list in half, sort each half using wishful thinking
- Merge the sorted list together

```
function middle(n) {
    return math_floor(n / 2);   }
function take(xs, n) {
    return n === 0
      ? null
      : pair(head(xs), take(tail(xs), n - 1));
} //Put the first n elements of xs into a list
function drop(xs, n) {
      return n === 0 ? xs
                 : drop(tail(xs), n - 1);
} // Drop the first n elements from list & return rest
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)));
     }  } // Merge two sorted lists into one sorted list
function merge_sort(xs) {
     if (is_null(xs) || is_null(tail(xs))) {
          return xs;
     } else {
          const mid = middle(length(xs));
          return merge(merge_sort(take(xs, mid)),
                       merge_sort(drop(xs,mid)));
     }  }
```

```
return accumulate(merge, null, xs);
```
→ flatten + merge_sort LOL

Quick Sort (Time B: Θ(nlogn), W: Θ(n²), Space: Θ(n))

- Partition the list using pivots. Pivot → any element.
- The partition returns a pair of list. The head is a list of elements smaller than the pivot, while the tail is a list of elements larger than the pivot.
- Append the 2 lists w the head to return sorted list.
- For this example, Head is used as the pivot (worst)

```
function partition(xs, p){
  const small_equal = filter(y => y <= p, xs);
  const bigger = filter(y => y > p, xs);
  return pair(small_equal, bigger);
} // Filter out the elements vs pivot                    Θ(n)
function quicksort(xs) {                 Similar to BST_to_list
  if (is_null(xs) || is_null(tail(xs))) {
    return xs;
  } else {
  const sort_left = quicksort(head(partition(tail(xs),
head(xs))));
  const sort_right = quicksort(tail(partition(tail(xs),
head(xs))));
    return append(sort_left, pair(head(xs), sort_right));  }    }
```

ETC Helpful stuff

Permutations

```
function permutations(s) {
    return is_null(s)
       ? list(null)
       : accumulate(append, null,
            map( x => map( p=> pair(x,p),
                permutations(remove(x,s))),
            s));
```

Subset

```
function subsets(xs){
  if (is_null(xs)){
    return list(null);
  } else {
    const subset_rest = subsets(tail(xs));
    const x = head(xs);
    const has_x = map(s => pair(x,s), subset_rest);
    return append(subset_rest, has_x);
  }
}

function subsets_2(xs) {
  return accumulate(
      (x, ss) => append(ss, map(s => pair(x, s), ss)),
      list(null),
      xs);
}
```

Choose

```
function choose(n, r) {
   if (n < 0 || r < 0) {
     return 0;
   } else if (r === 0) {
     return 1;
   } else {
// Consider the 1st item, there are 2 choices:
// To use, or not to use
// Get remaining items with wishful thinking
       const to_use = choose(n - 1, r - 1);
       const not_to_use = choose(n - 1, r);
       return to_use + not_to_use;
   }
}
```

Map (using accumulate)

```
function my_map(f, xs) {
  return accumulate(
      (x, ys) => pair(f(x), ys), null, xs);
}
```

```
function my_filter(pred, xs) {
  return accumulate( (a, b) => pred(a) ? pair(a, b) : b, null, xs);
}
```

```
function find_ranks(lst) {
   return map(y => length(filter(x => x <= y, lst)), lst);
}
```

Combinations

```
function combinations(xs, r) {
    if ( (r !== 0 && xs === null) || r < 0) {
        return null;
    } else if (r === 0) {
        return list(null);
    } else {
        const no_choose = combinations(tail(xs), r);
        const yes_choose = combinations(tail(xs), r - 1);
        const yes_item = map(x => pair(head(xs), x),
                                          yes_choose);
        return append(no_choose, yes_item);
    }
}
```

Remove Duplicates

```
function remove_duplicates(lst) {
  return is_null(lst)
    ? null
    : pair(
      head(lst),
      remove_duplicates(
        filter(
          x => !equal(x, head(lst)),
          tail(lst))));
}

function remove_duplicates(lst) {
  return accumulate(
    (x, xs) => is_null(member(x, xs))
          ? pair(x, xs)
          : xs,
    null,
    lst);
}
```

Make_up amount

```
function makeup_amount(x, coins) {
   if (x === 0) {
      return list(null);
   } else if (x < 0 || is_null(coins)) {
      return null;
   } else {
// Combinations that do not use the head coin.
   const combi_A = makeup_amount(x, tail(coins));
// Combinations that do not use the head coin
// for the remaining amount.
   const combi_B = makeup_amount(x-
                       head(coins),tail(coins));
// Combinations that use the head coin.
   const combi_C = map(x => pair(head(coins), x),
                         combi_B);
return append(combi_A, combi_C);
}
```

Sum(odd rank, even rank)

```
function sums(xs) {
  if (is_null(xs)) {
    return list(0, 0);
  } else if (is_null(tail(xs))) {
    return list(head(xs), 0);
  } else {
    const wish = sums(tail(tail(xs)));
    return list(head(xs) + head(wish), head(tail(xs)) +
              head(tail(wish)));
  }
}
```

BST to list

```
  if (is_null(bst)) {
     return null;
       } else {
         const ltree = head(tail(bst));
         const num = head(bst);
         const rtree = head(tail(tail(bst)));
         return append(BST_to_list(ltree),
                   pair(num, BST_to_list(rtree)));        }
```