CS1101S Finals Cheat Sheet

Notes

Code stops evaluating when false && (premise) or true || (premise)

- ➔ Recall CS1231S: F ∧ p ≡ F , T ∨ p ≡ T
- ➔ Usually use it when base case is Boolean.
- ➔ a && b → a ? b : false
- ➔ a || b → a ? true : b

*Applicative Order: Evaluate arguments, then apply function

*Normal Order: Evaluate function, then apply function onto argument

Order of Growth

Time	Space
Time taken for the program to run/ number of steps.	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 $Θ(n^2)$, it means that the order or growth is strictly n^2
>
> ➢ Then for $Ω(n^2)$ it means that the best case is n^2, but there could be instances where the OOG is larger, e.g $Ω(n^3)$, $Ω(2^n)$

*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:

Generally: $T(n) = T(n-1) + O(n^k)$

$T(n) = O(n^{k+1})$

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)	T(n) = O(n/2) + T(n/2) ➔ 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)
$O(n^2)$	$O(n^{k+1})$
T(n) = T(n-1) + O(n) ➔ $T(n) = O(n^2)$	$T(n) = T(n-1) +O(n^k)$ ➔ $T(n) = O(n^{k+1})$
$O(2^n)$	
T(n) = 2T(n-1) + O(1) ➔ $T(n) = O(2^n)$	T(n) = T(n-1) + T(n-2) +O(1) ➔ $T(n) = O(2^n)$

OOG Loops

- When the iterations of the inner loop depend on the outer loop, just calculate the sum over the total iterations of the inner loop to get OOG

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.

List

- 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

Tree

- 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

Binary Tree(BT)

A **BT** is either an **empty tree**, or it has

- an **entry** (which is the data item)
- **left branch/subtree** (which is a BT); **right branch/subtree** (which is a BT)

Binary Search Tree(BST)

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

Streams

- A ***stream*** is either the **empty list**, or
- A **pair** whose **head** is a **data item**, and whose **tail** is a **nullary function** that returns a stream
- Wrap tail in a function (i.e () => recursion, with a nullary function)
- Use stream_tail instead of tail (Recursive via stream_tail)
- const mt = pair(null, ()=> mt); → undefined stream
- An empty stream is undefined, as we don't know what data structure it is, e,g is it an array? List? Tree? Boolean etc.

Arrays

- An array is a data structure that stores a sequence of data elements
- Elements in [ ] still can be changed by array assignment/ the elements in [ ] are mutable
- Let: Mutation & Reassignment; Const: Mutation ONLY
- **Array access** — each data element can be accessed by using the array's name & a non-negative integer index (index → position of the element in the array)
- ***Array assignment*** — each data element can be assigned to with new value

Random Access:

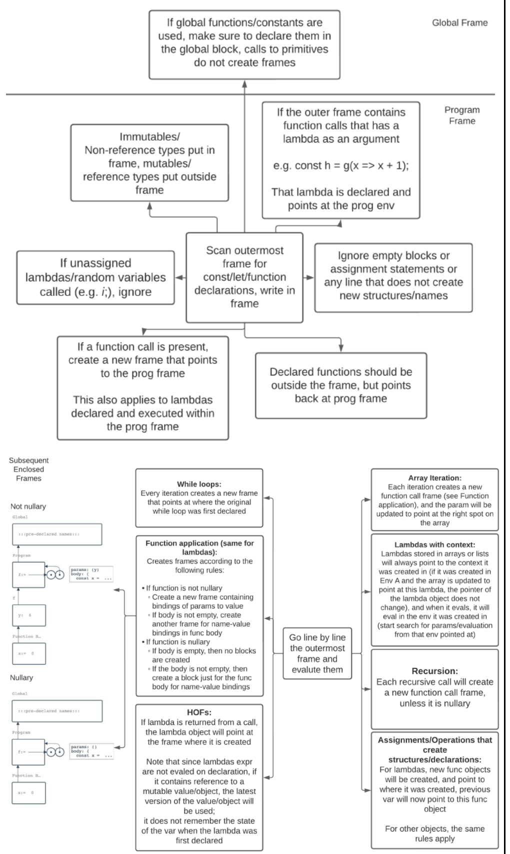
- Elements can be accessed (read) & assigned (written) in constant time

Exception: Assigning to an array element **A[i]**,

where index **i ≥ array_length(A), takes **Θ(i – array_length(A))** time

Break	terminates the current execution of the loop and also terminates the entire loop; if its an inner loop, proceed with outer loop evaluation
Continue	terminates the **current** execution of the loop and continues with the loop

Environment Model



If global functions/constants are used, make sure to declare them in the global block, calls to primitives do not create frames

Immutables/ Non-reference types put in frame, mutables/ reference types put outside frame

If the outer frame contains function calls that has a lambda as an argument

e.g. const h = g(x => x + 1);

That lambda is declared and points at the prog env

If unassigned lambdas/random variables called (e.g. *i*;), ignore

Scan outermost frame for const/let/function declarations, write in frame

Ignore empty blocks or assignment statements or any line that does not create new structures/names

If a function call is present, create a new frame that points to the prog frame

This also applies to lambdas declared and executed within the prog frame

Declared functions should be outside the frame, but points back at prog frame

While loops: Every iteration creates a new frame that points at where the original while loop was first declared

Function application (same for lambdas): Creates frames according to the following rules:

- If function is not nullary
  - Create a new frame containing bindings of params to value
  - If body is not empty, create another frame for name-value bindings in func body
- If function is nullary
  - If body is empty, then no blocks are created
  - If the body is not empty, then create a block just for the func body for name-value bindings

HOFs: If lambda is returned from a call, the lambda object will point at the frame where it is created

Note that since lambdas expr are not evaled on declaration, if it contains reference to a mutable value/object, the latest version of the value/object will be used; it does not remember the state of the var when the lambda was first declared

Go line by line the outermost frame and evalute them

Array Iteration: Each iteration creates a new function call frame (see Function application), and the param will be updated to point at the right spot on the array

Lambdas with context: Lambdas stored in arrays or lists will always point to the context it was created in (if it was created in Env A and the array is updated to point at this lambda, the pointer of the lambda object does not change), and when it evals, it will eval in the env it was created in (start search for params/evaluation from that env pointed at)

Recursion: Each recursive call will create a new function call frame, unless it is nullary

Assignments/Operations that create structures/declarations: For lambdas, new func objects will be created, and point to where it was created, previous var will now point to this func object

For other objects, the same rules apply

Memoization

- Memo function stores a "local table" of values previously computed
- Memoization avoids repeated calculation of the result of a function applied to the same arguments
- Why do we use "global memory" instead of "local memory"? → we want the scope to be accessible in all cases (More below)

Binary Search Time: Θ (log n), O(n logn)

Binary Search (Recursion)

```
function binary_search(A, v) {
function search(low, high) {
if (low > high) {
   return false;
} else {const mid =
         math_floor((low+high)/2);
  return (v === A[mid]) ||
         (v < A[mid]
            ? search(low, mid - 1)
            : search(mid + 1, high));
} }
return search(0, array_length(A)- 1);}
```

Binary Search (Loop)

```
function binary_search(A, v) {
let low = 0;
let high = array_length(A) - 1;
while (low <= high) {
const mid =
math_floor((low+high)/2 );
if (v === A[mid]) {
   break;
} else if (v < A[mid]) { high = mid -
1;
} else { low = mid + 1;
}}
return (low <= high);  }
```

Insertion Sort (Time Best: Θ(n), Worst: $Θ(n^2)$, 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)

List

```
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)));
}
```

Array

```
function insertion_sort(A) {
  const len = array_length(A);
  for (let i = 1; i < len; i = i + 1) {
    let j = i - 1;
    while (j >= 0 && A[j] > A[j + 1]) {
      swap(A, j, j + 1);
       j = j - 1;
    } } }

function insertion_sort2(A) {
  const len = array_length(A);
  for (let i = 1; i < len; i = i + 1) {
    const x = A[i];
    let j = i - 1;
    while (j >= 0 && A[j] > x) {
      A[j + 1] = A[j];
      j = j - 1;
    }
    A[j + 1] = x;  } }
```

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

List

```
function middle(n) {
     return math_floor(n / 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)));
     }
}

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)));
       }  }
```

Array

```
function merge_sort(A) {
  MSH(A, 0, array_length(A) - 1);
  return A; }

function MSH(A, low, high) {
  if (low < high) {
    const mid =
       math_floor((low + high) / 2);
       MSH(A, low, mid);
       MSH(A, mid + 1, high);
       merge(A, low, mid, high);
  }  }           merge_sort_helper*

function merge(A, low, mid, high) {
  const B = [];
  let left = low;
  let right = mid + 1;
  let Bidx = 0; //B-index
  while (left <= mid && right <=
high) {
     if (A[left] <= A[right]) {
       B[Bidx] = A[left];
       left = left + 1;
     } else {
       B[Bidx] = A[right];
       right = right + 1;
     }
       Bidx = Bidx + 1;
  }
  while (left <= mid) {
     B[Bidx] = A[left];
     Bidx = Bidx + 1;
     left = left + 1;
  }
  while (right <= high) {
     B[Bidx] = A[right];
     Bidx = Bidx + 1;
     right = right + 1;
  }
  for (let k = 0; k < high - low + 1; k
= k + 1) {
     A[low + k] = B[k];     } }
```

Selection Sort (Time $Θ(n^2)$, Space: Θ(1))

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

List

```
function smallest(xs) {
return accumulate(
      (x, y) => x < y ? x : y,
          head(xs), tail(xs));}

function selection_sort(xs) {
if (is_null(xs)) {
return xs;} else {
const x = smallest(xs);
return pair(x,
selection_sort(remove(x,
xs)));    }  }
```

Array

```
function find_min_pos(A, low, high) {
  let min_pos = low;
  for (let j = low + 1; j <= high; j = j + 1) {
    if (A[j] < A[min_pos]) {
      min_pos = j;
  }  }     return min_pos;}

function selection_sort(A) {
  const len = array_length(A);
  for (let i = 0; i < len - 1; i = i + 1) {
    let min_pos =
          find_min_pos(A, i, len - 1);
    swap(A, i, min_pos);
  }  }
```

Quick Sort (Time B: Θ(nlogn), W: $Θ(n^2)$, 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);
}

function quicksort(xs) {
  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));  }    }
```

Similar to BST_to_list

Bubble Sort: (Time: Avg $Θ(n^2)$, Best case O(n) – sorted; Space: O(1))

- Repeatedly swaps adjacent elements if they are in the wrong order

List

```
function bubblesort_list(L) {
  const len = length(L);
  for (let i = len - 1; i >= 1; i = i - 1) {
    let p = L;
    for (let j = 0; j < i; j = j + 1) {
      if (head(p) > head(tail(p))) {
        const temp = head(p);
        set_head(p, head(tail(p)));
        set_head(tail(p), temp);
      }
      p = tail(p);
  }   }    return L;  }
```

Array

```
function bubblesort_array(A) {
  let clone = A;
  const len = array_length(A);
  for (let i = len - 1; i >= 1; i = i - 1) {
    for (let j = 0; j < i; j = j + 1) {
      if (clone[j] > clone[j + 1]) {
        const temp = clone[j];
        clone[j] = clone[j + 1];
        clone[j + 1] = temp;
      }
    }
  }   return clone;   }
```

Linear Search Time: O(n)

```
function linear_search(A, v) {
  const len = array_length(A);
  let i = 0;
  while (i < len && A[i] !== v) {
    i = i + 1;
  }     return (i < len); }
```

Reuse your pairs (Destructive mergesort)

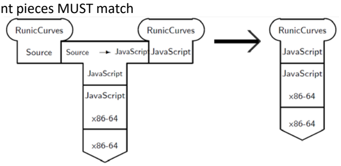
```
function d_split_list(xs) {
  if(is_null(xs)){
    return xs;
  }else{
  const len = length(xs);
  let right_list = list();
  let current = xs;
  const split_len = len % 2 === 0
        ? len / 2
        : math_ceil(len / 2);
  for(let i = 1; i < split_len; i = i + 1){
    current = tail(current);
  }
  right_list = tail(current);
  set_tail(current, null);
  return pair(xs, right_list); } }
```

```
function d_merge(xs, ys) {
  if(is_null(xs)){
    return ys;
  }
  else if(is_null(ys)){
    return xs;
  } else {
    if(head(xs) < head(ys)){
  set_tail(xs, d_merge(tail(xs), ys));
    return xs;
     }else{
  set_tail(ys, d_merge(xs, tail(ys)));
     return ys;
    }
  }
}
```

```
function d_merge_sort(xs) {
  if (is_null(xs) || is_null(tail(xs))) {
    return xs;
  } else {
    const split = d_split_list(xs);
    const a = head(split);
    const b = tail(split);
    return d_merge(d_merge_sort(a), (d_merge_sort(b)));     } }
```

Tombstone Compiler

- A compiler is a program that translates from one language (the from-language) to another language (the to-language).
- A compiler, which translates programs from one language to another, is just another program.
- Adjacent pieces MUST match



Compiling "RunicCurves" from Source to JS in browser, & then running the JS program in browser.

Memoized Function

```
function memoize(f) {
  const mem = [];
  function mf(x) {
    if (mem[x] !== undefined) {
      return mem[x];
    } else {
      const result = f(x);
      mem[x] = result;
      return result;
    } }
  return mf;
}
```

Memoization w Lazy Evaluation

```
function memo_fun(fun) {
  let already_run = false;
  let result = undefined;
  function mfun() {
    if (!already_run) {
      result = fun();
      already_run = true;
      return result;
    } else {
      return result;
    }   }
  return mfun;   }
```

- To Memoize, functions should be encased in the memorized function. I.e substeps should also be memorized as well, not just a specific computed value.
- Memoization is good for functions that requires repeated computation of the same things.

Read

```
function read(n, k) {
  return mem[n] === undefined
    ? undefined
    : mem[n][k];
}
```

Write

```
function write(n, k, value) {
  if (mem[n] === undefined) {
    mem[n] = [];
  }
  mem[n][k] = value;   }
```

Order of growth? Space and time: O(nk)

```
function mcc(n, k) {
  if (n >= 0 && k >=0 && read(n, k) !== undefined) {
    return read(n, k);
  } else {
    const result = n === 0 ? 1 : n < 0 || k === 0 ? 0
              : mcc(n, k - 1) + mcc(n - first_denomination(k), k);
    if (n >= 0 && k >= 0) {
      write(n, k, result);
    }
    return result;    } }
```

Subsets

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

```
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 {
   const to_use = choose(n - 1, r - 1);
   const not_to_use = choose(n - 1, r);
    return to_use + not_to_use;  }  }
```

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));
}
```

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); } }
```

BST to List

```
function BST_to_list(bst){
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)));
}
```

Makeup Amount

```
function mcc(x, c) {
  if (x === 0) {
    return list(null);
  } else if (x < 0 || is_null(c)) {
    return null;
  } else {
const A = mcc(x, tail(c));
const B = mcc(x - head(c),tail(c));
const C = map(x => pair(head(c), x), B);
return append(A, 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)));  }  }
```

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);
}
```

ARRAY Functions

Accumulate Array

```
function accum_array(op, init, A) {
 let x = init;
 for (let i = init; i < array_length(A);
i = i + 1) {
   x = op(x, A[i]);
 }
  return x;    }
```

Clone/Copy an Array

```
function copy_array(A) {
  const len = array_length(A);
  const B = [];
  for (let i = 0; i < len; i = i + 1) {
    B[i] = A[i];
  }
  return B;  }
```

Map Array

```
function map_array(f, arr) {
     for (let I = 0; I <
array_length(arr); I = I + 1) {
      arr[i] = f(arr[i]);
     }
     return arr;   }
```

Swap

```
function swap(A, x, y) {
  const temp = A[x];
  A[x] = A[y];
  A[y] = temp;
}
```

Add Element to back of Array

```
function add_back(element, arr){
  const len = array_length(arr);
  arr[len] = element;
  //return arr;   (return optional)
}
```

Build Array

```
function build_array(n, f){
    const A = [];
    for(let I = 0; I < n; I = I + 1){
     A[i] = f(i);
    }
    return A;    }
```

Convert Array to List

```
function array_to_list(A) {
 const len = array_length(A);
 let L = null;
 for (let I = len – 1; I >= 0; I = I – 1) {
     L = pair(A[i], L);
    }
    return L;  }
```

Convert List to Array

```
function list_to_array(L) {
 const A = [];
 let I = 0;
 for (let p = L; !is_null(p); p = tail(p)) {
   A[i] = head(p);
   I = I + 1;}
 return A; }
```

Flatten an Array

```
function flatten_array(arr){
 let index_count = 0;
 let result = [];
 function helper(a) {
  const len = array_length(a);
  for (let i = 0; i < len; i = i + 1) {
   const curr = a[i];
    if (is_array(curr)) {
       helper(curr);
    } else {
    result[index_count] = curr;
    index_count = index_count + 1;
    }  }   }
   helper(arr);
   return result;  }
```

Append Array

```
function append_array(arr1, arr2) {
 let final = [];
 for (let i = 0; i < array_length(arr1); i =
i + 1) {
   final[i] = arr1[i];
  }
 for (let j = 0; j < array_length(arr2); j =
j + 1) {
   final[j + array_length(arr1)] = arr2[j];
 }
  return final;
}
```

Array Reverse

```
function arr_reverse(arr) {
 const reversed_arr = [];
 const len =array_length(arr);
for (let i = 0; i < len ; i = i + 1){
  reversed_arr[len - i - 1] = arr[i];
 }
 return reversed_arr;  }
```

Destructive Array Reverse

```
function d_arr_reverse(arr) {
 const len = array_length(arr);
 for (let i = 0; i < len/2; i = i + 1) {
    const temp = arr[len - i - 1];
    arr[len - i - 1] = arr[i];
    arr[i] = temp;    }
  return arr;  }
```

Tree Functions

Accumulate Trees

```
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);    }
```

Destructive Filter

```
function d_filter(pred, xs){
 if(is_null(xs)){
   return xs;
 }else if(pred(head(xs))){
   set_tail(xs, d_filter(pred, tail(xs)));
}else{
   d_filter(pred, tail(xs));
   }
}
```

Map Trees

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

Scale Tree

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

Tree Sum

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

Count Data items in Tree

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

Tree to Array Tree

```
function tree_to_arraytree(xs){
 if(is_number(xs)){
  return xs;
  }else{
   let A = [];
   let counter = 0;
   let clone = xs;
   while(!is_null(clone)){
    A[counter] =
  tree_to_arraytree(head(clone));
    counter = counter + 1;
    clone = tail(clone);
    }
  return A;   } }
```

Array Tree to Tree

```
function arraytree_to_tree(a) {
 if (is_number(a)) {
   return a;
 } else {
   let xs = null;
   const len = array_length(a);
 for (let i = len - 1; i >= 0; i = i - 1) {
 xs = pair(arraytree_to_tree(a[i]), xs);
 }
  return xs;
  }
}
```

Streams Functions

Pre-declared (S4):

stream_append	stream_length	stream_member	stream_reverse
stream_filter	stream_map	stream_remove	stream_tail
stream_for_each	stream_ref	stream_remove_all	stream_to_list
list_to_stream	build_stream	enum_stream	eval_stream
Integers_from	is_stream		

Every other

```
function every_other(s) {
  return pair(head(s), () =>
every_other(stream_tail(
             stream_tail(s))));
}
```

Alternating Streams

```
function make_alternating_stream(s) {
   return pair(head(s),
    () => stream_map(x => -x,
make_alternating_stream(
                     stream_tail(s)))); }
```

List to infinite stream

```
function list_to_inf_stream(xs){
 function inner(ys){
  if(is_null(ys)){
    return inner(ys);
  }else{
    return
    pair(head(ys), ()=>
inner(tail(ys)));
  }
 }
 return is_null(xs) ? null : inner(xs);
}
```

```
function powerS2(n){
 function helper(count){
 return pair(count*math_pow(n,count),
            ()=> helper(count +1));
 }
 return helper(0);
}

function S2(n){
 let integers = integers_from(1);
 return stream_map(x=> x *
 math_pow(n, x-1), integers);
}
```

n of n stream

```
function n_of_n_stream() {
  function more2(a, b) {
    return (a > b)
      ? more2(1, 1 + b)
      : pair(b, () => more2(a+1, b));
  }
  return more2(1,1);
}
```

Lazy Prime Generation with Streams

```
function is_divisible(x, y) {
   return x % y === 0;
}
function sieve(s) {
 return pair(head(s),
  () => sieve(stream_filter(
    x => !is_divisible(x, head(s)),
                   stream_tail(s))));
}
```

Array to Stream

```
function array_to_stream(a) {
  function helper(count){
    if(is_undefined(a[count])){
      return null;
    }else{
      return pair(a[count],
        ()=> helper(count+1));
    }
  }
  return helper(0);
}
```

Loop Stream

```
function loop_stream(s) {
 function helper(p) {
  if (is_null(stream_tail(p))) {
   return pair(head(p), () => helper(s));
   } else {
   return pair(head(p), () =>
                helper(stream_tail(p)));
   }
 }
 return helper(s);
}
```

Extend

```
function extend(bno){
 return (x, y) =>
  pair(bno(head(x), head(y)),
     ()=>extend(bno)(stream_tail(x),
            stream_tail(y)));   }
```

Stream Combine

```
function stream_combine(f, s1, s2) {
  return pair(f(head(s1), head(s2)),
   ()=> stream_combine(f,
    stream_tail(s1), stream_tail(s2)));
}
```

Timelapse

```
function time_lapse(s, n) {
 function helper(count){
  if(s!==null){
   return pair(stream_ref(s, count),
     ()=> helper(count + n));
   }
  }
 return helper(0); }
```

Zip stream

```
function zip_streams(s1, s2) {
 if(is_null(s1)){
   return s2;
 } else if (is_null(s2)){
   return s1;
 } else{
   return pair(head(s1), () =>
     pair(head(s2), () =>
       zip_streams(stream_tail(s1),
         stream_tail(s2))));
} }
```

Partial Sums

```
function partial_sums(s) {
  let a = head(s);
  return pair(a, () => stream_map(
   x => x + a, partial_sums(
                stream_tail(s))));}
```

Fib Generator

```
function fibgen(a, b) {
  return pair(a, () => fibgen(b, a + b));
}
```

More and More

```
function more(a, b) {
return (a > b) ? more(1, 1 + b)
       : pair(a, () => more(a + 1, b));
}
const moremore = more(1, 1);
eval_stream(moremore, 15);
```

Shorten stream

```
function shorten_stream(s, k) {
  return k === 0
  ? list(): is_null(s)
  ? null : pair(head(s), () =>
     shorten_stream(stream_tail(s),
   k - 1));   }
```

Misc

Transpose

```
function transpose(M) {
  const Tmatrix = [];
  const row = array_length(M);
  const col = array_length(M[0]);
   for(let r = 0; r < col; r = r + 1){
    Tmatrix[r] = [];
    for(let c = 0; c < row; c = c + 1){
      Tmatrix[r][c] = M[c][r];
    }
  }
  return Tmatrix;   }
```

Rotate Matrix

```
function rotate_matrix(M) {
  const n = array_length(M);
  function swap(r1, c1, r2, c2) {
    const temp = M[r1][c1];
    M[r1][c1] = M[r2][c2];
    M[r2][c2] = temp;
  }
  for (let r = 0; r < n; r = r + 1) {
    for (let c = r + 1; c < n; c = c + 1) {
      swap(r, c, c, r);
   }  }
  const half_n = math_floor(n / 2);
  for (let r = 0; r < n; r = r + 1) {
    for (let c = 0; c < half_n; c = c + 1) {
      swap(r, c, r, n - c - 1);
    }   } }
```

Mutable Reverse(List)

```
function mutable_reverse(xs){
  if(is_null(xs)){
    return xs;
  }else if(is_null(tail(xs))){
    return xs;
  }else{
    let temp =
    mutable_reverse(tail(xs));
    set_tail(tail(xs), xs);
    set_tail(xs, null);
    return temp;
  } }
```

Make Circular(List)

```
function make_circular(xs){
 function inner(zs, ys){
  if(is_null(zs)){
    return ys;
  }else{
 return pair(head(zs), inner(tail(zs), ys));
  }
}
 if(is_null(xs)){
   return null;
  }else{
   let ys = pair(head(xs), null);
   set_tail(ys, inner(tail(xs), ys));
   return ys;
  }
}
```

Make Linear(List)

```
function make_linear(xs) {
  function inner(ys) {
    if (tail(ys) === xs) {
      set_tail(ys, null);
    } else {
      inner(tail(ys));
    }
  }
  if (!is_null(xs)) {
    inner(xs);
  } else {}
  return xs;
}
```

Equal Sets(Lists)

```
function are_equal_sets(set1, set2) {
 if (length(set1) !== length(set2)) {
   return false;
 } else {
   return accumulate(
     (x1, y1) => accumulate(
       (x2, y2) => x1 === x2 ||
           y2, false, set2) &&
             y1, true, set1);
}
```