CS1101S Cheatsheet

for midterms AY23-24

m. **zaidan**

Recursion & Iteration

Recursion: Increasing deferred operations
Iteration: Constant deferred operations
(rule of thumb: if the final call is the application call, it is usually *iterative*)

Definitions of List & Tree

```
const pair = (x, y) => f => f(x, y);
const head = p => p((x, y) => x);
const tail = p => p((x, y) => y);
```

A list is either null or a pair whose tail is a list.
A tree is either null or head is an element, tail is a tree or head is an tree, tail is a tree

List Abstractions

```
function map(f, xs) {
    return is_null(xs)
    ? null
    : pair(f(head(xs)), map(f, tail(xs)));
}
```

Iterative process; **time: O(n), space: O(n)**, where n is the length of xs.

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

Iterative process; **time: O(n), space: O(n)**, where n is the length of xs.

```
function accumulate(op, initial, xs) {
    return is_null(xs)
    ? initial
    : op(head(xs), accumulate(op,
                              initial,
                              tail(xs)));
}

accumulate((curr, wish) => .., initial, xs);
```

Iterative process; **time: O(n), space: O(n)**, where n is the length of xs assuming f takes constant time.

Tree Abstractions

```
function accumulate_tree(f,op,initial,tree) {
    return accumulate((x, ys) => is_list(x)
        ? op(accumulate_tree(f, op, initial,
            x), ys)
        : op(f(x), ys), initial, tree);
}

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

function flatten_tree(xs) {
    function h(xs, prev) {
        return is_null(xs)
            ? prev
            : is_list(xs)
                ? append(flatten(xs), prev)
                : pair(xs, prev);
    }
    return accumulate(h, null, xs);
}

function insert(bst, item) {
    if (is_empty_tree(bst)) {
        return make_tree(item,
            make_empty_tree(),
                 make_empty_tree());
    }
    else {
        if (item < entry(bst)) {
            return make_tree(entry(bst),
              insert(left_branch(bst),item),
              right_branch(bst));
        }
        else if (item > entry(bst)) {
            return make_tree(entry(bst),
              left_branch(bst),
              insert(right_branch(bst), item)
                  );
        }
        else {
            return bst;
        }
    }
}

function find(bst, name) {
    return is_empty_tree(bst)
        ? false
        : name === entry(bst)
            ? true
            : name < entry(bst)
              ? find(left_branch(bst),name)
              : find(right_branch(bst),name);
}
```

Permutations & Combinations

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

function subsets(s) {
    return accumulate(
        (x, s1) => append(s1,
                   map(ss => pair(x, ss), s1)
                      ),
        list(null),
        s);
}

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;
    }
}

function combinations(xs, r) {
    if ( (r !== 0 && xs === null) || r < 0) {
        return null;
    } else if (r === 0) {
        return list(null);
    } else {
        const no = combinations(tail(xs), r);
        const yes = combinations(tail(xs), r
            - 1);
        const yes_item = map(x => pair(head(
            xs), x), yes);
        return append(no, yes_item);
    }
}

function makeup_amount(x, coins) {
    if (x === 0) {
        return list(null);
```

```
    } else if (x < 0 || is_null(coins)) {
        return null;
    } else {
        const combi_A = makeup_amount(x, tail
            (coins));
        const combi_B = makeup_amount(x -
            head(coins), tail(coins));
        const combi_C = map(x => pair(head(
            coins), x), combi_B);
        return append(combi_A, combi_C);
    }
}
```

Rule of Thumb for Abstractions

Is input a list?	if not, don't use
Is length(output) = length(input)?	use *map*
Is items in output = items in input?	use *filter*
Else	use *accumulate*

Useful Math Functions

```
math_pow(base, exponent);
math_round(x);
math_floor(x);
math_ceil(x);
math_sqrt(x);
```

Orders of Growth

for $r(n)$,

Big O $O(g(n))$**:** if there is a positive constant k such that $r(n) \leq k * g(n)$ for any sufficiently large value of n
Big Theta $\theta(g(n))$**:** if there is positive constants k_1 and k_2 and a number n_0 such that $k_1 * g(n) \leq r(n) \leq k_2 * g(n)$ for any n $> n_0$.
Big Omega $\Omega(g(n))$**:** if there is a positive constant k such that $k * g(n) \leq r(n)$ for any sufficiently large value of n
Order (ascending): 1, log n, n, n log n, n^2, n^3, 2^n, 3^n, n^n

Ignore constants, lower order terms. $O(2n) = O(3n) = O(n)$
For a sum, take the larger term. $O(n) + O(n^2) = O(n^2)$
For a product, multiply the two terms. $O(n) \times O(n) = O(n^2)$

Recurrence Relations

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

Sorting/Search Algorithms

	Time		Space
Binary Search	θ(log n)	O(log n)	O(1)
Selection Sort	$\theta(n^2)$	$O(n^2)$	O(n)
Insertion Sort	θ(n)	$O(n^2)$	O(n)
Merge Sort	θ(n log n)	O(n log n)	O(n log n)
Quick Sort	θ(n log n)	$O(n^2)$	$O(n^2)$

Selection Sort

```
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)));
    }
}
```

Merge Sort

```
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)));
    }
}

const middle = x => math_floor(x/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_sort(xs) {
    if (is_null(xs) || is_null(tail(xs))) {
       return xs;
    } else {
       const m = middle(length(xs));
       return merge(merge_sort(take(xs, m)),
                    merge_sort(drop(xs, m)));
    }
}
```

Quick Sort

```
function partition(xs, p) {
    return pair(filter(x => x <= p, xs),
        filter(x => x > p, xs));
}

function quicksort(xs) {
    if (is_null(xs)) {
        return null;
    }
    else if (is_null(tail(xs))) {
        return xs;
    }
    else {
        const pivot = head(xs);
        const ptn = partition(tail(xs), pivot
            );
        return accumulate(append, null, list(
            quicksort(head(ptn)), list(pivot),
             quicksort(tail(ptn))));
    }
}
```