Wu Xiaoyun NUS AY20/21 Sem 1

Check for Prime no.

```
function is_prime(n) {
  function g(d){
    return d === 1
    ? true
    : ( n % d !== 0) && g(d - 1);
  }
  return g(n - 1);}
```

HOF

```
function sum(term, a, next, b) {
  return a > b
      ? 0
      : term(a) + sum(term, next(a), next, b);
}
e.g. function sum_odd(n) {
   return sum(x => x, 1, x => x + 2, 2 * n - 1); }
```

List Processing

Length

```
function length(xs) {
          return is_null(xs) ? 0 : 1 +
length(tail(xs));}
```

Reverse

```
function reverse(xs) {
    function rev(original, reversed) {
      return is_null(original)
          ? reversed
          : rev(tail(original),
pair(head(original), reversed));
    }
    return rev(xs, null); }
```

Map

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

Map with accum

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

Multi-map

```
  function multi_map(f, xss) {
    if (is_null(head(xss))) {
      return null;
    } else {
      return pair(f(map(head, xss)),
            multi_map(f, map(tail, xss)));}}
```

Accumulate

```
function accumulate(f, initial, xs) {
    return is_null(xs)
          ? initial
          : f(head(xs),
          accumulate(f, initial, tail(xs))); }
  accumulate((current, rest) => current + rest,
0, list(1,2,3));
  // f(1, f(2, f(3, 0))); returns 6
```

Filter

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

Filter with accum

```
function filter(pred, xs) {
    return accumulate( (x, acc) => pred(x) ?
pair(x, acc) : acc,
              null,
              xs);}
```

Build list

```
function build_list(n, fun) {
    function build(i, fun, already_built) {
     return i < 0 ? already_built : build(i - 1,
fun, pair(fun(i), already_built));
    }
    return build(n - 1, fun, null);}
build_list(3, x => 2 * x); //returns list(0,2,4);
// function takes in a list from 0 to (n - 1)
```

Checking if all elements are diff

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

Tree Processing

Length

```
function count_data_items(tree) {
  return is_null(tree)
          ? 0
          : (is_list(head(tree))
          ? count_data_items(head(tree))
          : 1) + count_data_items(tail(tree));}
```

Reverse

```
function tree_reverse(tree) {
  function op(origin, reversed) {
    if (is_null(origin)) {
      return reversed;
    } else if (is_list(head(origin))) {
      return op(tail(origin),
pair(op(head(origin), null), reversed));
    } else {
      return op(tail(origin), pair(head(origin),
reversed));
    }
  }
   return op(tree, null);}
```

Map

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

Accumulate

```
function accumulate_tree(f1, f2, initial, tree) {
  if (is_null(tree)) {
    return initial;
  } else {
    const x = is_list(head(tree))
? accumulate_tree(f1, f2, initial, head(tree)) :
f1(head(tree));
    const y = accumulate_tree(f1, f2, initial,
tail(tree));
  return f2(x, y);  }}
```

Accumulate BST

```
function accumulate_bst(op, initial, bst) {
    if (is_empty_tree(bst)) {
      return initial;
    } else {
      const s = accumulate_bst(op, initial,
right_branch(bst));
      const t = op(entry(bst), s);
      return accumulate_bst(op, t,
left_branch(bst));}}
```

Smallest in Tree

```
function BST_min(bst) {
  return is_null(bst)
    ? Infinity
    : is_null(head(tail(bst)))
      ? head(bst)
      : BST_min(head(tail(bst)));}
```

Tree to list

```
function enumerate_tree(tree) {
  return is_null(tree)
      ? null
      : ! is_pair(tree)
       ? list(tree)
       : append(enumerate_tree(head(tree)),
enumerate_tree(tail(tree)));}
```

test tree of numbers

```
function is_tree_of_numbers(x) {
  return is_list(x) &&
      accumulate( (a,b) => (is_number(a) ||
is_tree_of_numbers(a)) && b,
              true,
              x);}
```

check if xs1 is a permutation of xs2

```
function are_permutations(xs1, xs2) {
   return is_null(xs1) && is_null(xs2)
      ? true
      : !is_null(xs1) && !is_null(xs2)
         ? !is_null(member(head(xs1), xs2)) &&
are_permutations(tail(xs1), remove(head(xs1),
xs2))
          : false;}
// order of growth = (length of xs1 * length of
xs2)
```

Permutations

```
// return the permutations of all elements in
the list xs
function permutations(xs) {
   if (is_null(xs)) {
      return list(null);
   } else {
      return accumulate(append, null,
                  map(x => map(p => pair(x,p),
permutations(remove(x, xs))), xs));}}
// return the permutation of r elements in the
list xs
function permutations_r(xs, r) {
   if (r === 0) {
// There is 1 permutation of length 0
      return list(null);
   } else if (is_null(xs)) {
// There is no permutation if xs is empty but r
is non-zero
      return null;
   } else {
      return accumulate(append, null,
                  map(x => map( p => pair(x, p),
permutations_r(remove(x, xs), r - 1)), xs));}}
```

Combinations

```
// returns the combinations of k elements in
the given list xs
function combinations(xs, k) {
   if (k === 0) {
      return list(null);
   } else if (is_null(xs)) {
      return null;
   } else {
      const s1 = combinations(tail(xs), k - 1);
      const s2 = combinations(tail(xs), k);
      const x = head(xs);
      const has_x = map(s => pair(x, s), s1);
      return append(has_x, s2);}}
```

Find Ranks

```
function find_ranks(lst) {
   return map(y => length(filter(x => x <= y,
lst)), lst);
} //order of growth = big-theta n^2
// list(9, 8, 5, 6) → list(4,3,1,2)
```

Coin Change

No. of permutations

```
function count_change(amount) {
   return cc(amount, 6);}
function cc(amount, kinds_of_coins) {
   return amount === 0
        ? 1
        : amount < 0 ||
          kinds_of_coins === 0
          ? 0
          : cc(amount, kinds_of_coins - 1)
            +
            cc(amount - first_denomination(
                        kinds_of_coins),
               kinds_of_coins);}
function first_denomination(kinds_of_coins) {
   return kinds_of_coins === 1 ? 1 :
        kinds_of_coins === 2 ? 5 :
        kinds_of_coins === 3 ? 10 :
        kinds_of_coins === 4 ? 25 :
        kinds_of_coins === 5 ? 50 :
        kinds_of_coins === 6 ? 100 : 0; }
```

list of permutations of coins that makes up x cents

```
function makeup_amount(x, coins) {
   if (x === 0) {
      return list(null);
   } else if (x < 0 || is_null(coins))  {
      return list(); // list() = 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); }}
```

Sorts

Selection sort

```
function smallest(xs) {
    return accumulate(math_min, Infinity, xs);}
function largest(xs) {
    return accumulate(math_max, null, xs);}
function selection_sort(xs) {
   if (is_null(xs)) {
      return xs;
   } else {
   const x = smallest(xs);
      return pair(x,
            selection_sort(remove(x, xs)));}}
```

Insertion Sort

```
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)));}
```

Merge Sort

```
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 = math_floor(length(xs) / 2);
      return merge(merge_sort(take(xs, mid)),
            merge_sort(drop(xs, mid)));}}
```

Quick Sort

```
function partition(xs, p) {
   function list_lte(xs, p) {
      return filter((x => x <= p), xs);}
function list_gt(xs, p) {
      return filter((x => x > p), xs);}
   return pair(list_lte(xs,p), list_gt(xs, p));}
function quicksort(xs) {
   if (length(xs) <= 1) {
      return xs;
   } else {
      return
append(quicksort(head(partition(tail(xs),
head(xs)))),
            pair(head(xs),
               quicksort(tail(partition(tail(xs),
head(xs))))));}}
```