NATIONAL UNIVERSITY OF SINGAPORE

CS1101S — PROGRAMMING METHODOLOGY

(AY2018/2019 SEMESTER 1)

MIDTERM ASSESSMENT (ADAPTED TO AY2020/21 IN 9/2020)

Time Allowed: **1 Hour 45 Minutes**

INSTRUCTIONS

1. This assessment paper contains **EIGHT (8)** questions and comprises **EIGHTEEN (18)** printed pages, including this page.
2. The full score of this paper is **75 marks**.
3. This is a **CLOSED BOOK** assessment, but you are allowed to bring in one A4 sheet of notes (handwritten or printed on both sides).
4. Answer **ALL** questions **within the space provided** in this booklet.
5. Where programs are required, write them in the **Source §2** language.
6. Write legibly with a **pen or pencil**. **Untidiness will be penalized**.
7. Do not tear off any pages from this booklet.
8. Write your **Student Number** below **USING A PEN**. Do not write your name.
9. Also write down your **Studio Group Number** in the provided box, if you can remember it.

(write with a pen)

Student No.:

Studio Group No. (leave blank if cannot remember)**:**

This portion is for examiner's use only

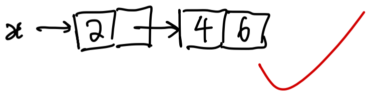
Q#	1	2	3	4	5	6	7	8	Σ
MAX	10	4	8	9	10	10	12	12	75
SC									

Question 1: Box-and-Pointer Diagrams [10 marks]

Draw the box-and-pointer diagram for the value of `x` after the evaluation of each of the following programs. Clearly show where `x` is pointing to.

1A. [2 marks]

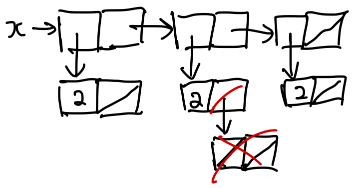
```
const x = pair(2, pair(4, 6));
```



1B. [2 marks]

null

```
const p = list(2);
const q = pair(2, list());
const x = list(p, q, pair(2, null));
```



1C. [3 marks]

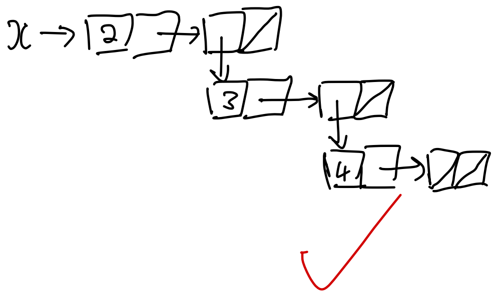
```
const x = list(list(list(null, null)));
```

2 elem.

list(2, list(3, list(4, null)))

1D. [3 marks]

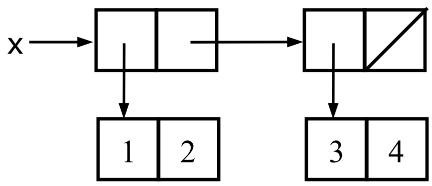
```
const x = accumulate(list, null, list(2, 3, 4));
```



Question 2: Making Pairs [4 marks]

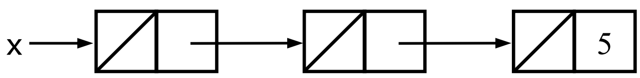
Write a Source §2 program that produces exactly the pairs shown in each of the following box-and-pointer diagrams. At the end of the execution of your program, the constant x must refer to the pair as shown in the diagram.

2A. [2 marks]



const x = list(pair(1,2) , pair(3,4));

2B. [2 marks]



const x = pair(null , pair(null , pair(null, 5)));

Question 3: Let's Be Logical [8 marks]

3A. [3 marks]

Consider the following function

```
function hoo(f, g, h, x) {
    if (f(x) || (g(x) && h(x))) {
        return 100;
    } else {
        return 50;
    }
}
```

Rewrite the function hoo without using the keyword **if** and without any logical operators (&&, ||, !). Your function must have the same order of evaluations and produce the same result as the original.

```
function hoo(f, g, h, x) {
        if (f(x)) {
            return 100;
        } else if ( g(x)) {
                if (h(x)) {
                    return 100;
                } else {
                    return 50;
                }
        } else {
                return 50;
        }.
}
```

3B. [5 marks]

Consider the following program

```
function gee(n) {
    return n <= 0 ? true : (false || gee(n - 1));
}
gee(4);
```

$T(n) = \Theta(1) + T(n-1)$

$\Theta(1) + \cdots + \Theta(1)$ n times.

$\Rightarrow \Theta(n)$.

(i) What kind of process does the program give rise to?
(ii) Justify your answer by showing the evaluation steps (hint: the substitution model).

(i) recursive. iterative

(ii) gee (4)

false || gee(3)

false || false || gee (2) evaluate to false

false || false || false || gee(1) no deferred ops.

false || false || false || false || gee(0)

false || false || false || false || true.

false || false || false || true

false || false || true.

false || true.

true.

Question 4: Recursive vs Iterative Processes [9 marks]

The `get_sublist` function takes as arguments two integer numbers, `start` and `end`, and a list `L`, and returns a list containing the element(s) of `L` from position `start` to position `end`, including both. The first element of `L` is at position 0. You can assume that 0 ≤ `start` ≤ `end` < `length(L)`.

Example :

```
const L = list(11, 12, 13, 14, 15, 16, 17, 18);
get_sublist(3, 5, L);  // returns list(14, 15, 16).
```

Complete the following two implementations of `get_sublist` that give rise to **(A)** a **recursive process** and **(B)** an **iterative process**. Both implementations' runtime should have an order of growth of O(n), where n is the length of the list `L`.

4A. [4 marks] Recursive version.

```
function get_sublist(start, end, L) {
    function helper(pos, ys) {
        if (pos < start) {
            return helper(pos + 1, tail(ys));
        } else if (pos <= end) {
            return pair(head(ys), helper(pos+1, tail(ys)));
        } else { return null; }
    }
    return helper(0, L);
}
```

4B. [5 marks] Iterative version.

```
function get_sublist(start, end, L) {
    function helper(pos, ys, result) {   acc
        if (pos < start) {
            return helper(pos+1, tail(ys), result);
        } else if (pos <= end) {
            return helper(pos+1, tail(ys), append(result, list(head(ys))));
        } else {
            return result;
        }.
    }
    return helper(0, L, null);
}
```

Question 5: The Benefits of Being Sorted [10 marks]

5A. [5 marks]

We represent a ***set*** of numbers using a list of **distinct** numbers **sorted** in **ascending order**. Complete the function, `is_subset(S, T)`, to determine whether set S is a subset of set T, which is true only if every element in S is also an element of T. We assume that $0 \le N_S$ and $0 \le N_T$, where N_S and N_T are the number of elements in S and T respectively. If $N_S = 0$, then S is always a subset of T. Your program should exploit the fact that both lists are sorted; its runtime should have an order of growth of $O(N_S + N_T)$.

S ε T

1 3 5 6

1 2 3 4 5 6 7 8

```
function is_subset(S, T) {

    if (is_null(S)) {
        return true;
    } else if (is_null(T)) {
        return false;
    } else if (head(S) < head(T)) {
        return is_subset(tail(S), T);
    } else if (head(S) === head(T)) {
        return is_subset(tail(S), tail(T));
    } else {   head(S) > head(T)
        return false;
    }
}
```

will never be a time 2<0 5 < yGT

false.

x is_subset(S, tail(T))

If it bigger

⇒ can be subset

5B. [5 marks]

Complete the following function, `super_merge`, that takes in a list `L` of **one or more** lists of numbers, where the numbers in each list are in **ascending order**, and returns a single list of all the numbers from `L`, sorted in **ascending order**. For example:

```
const L = list(list(1, 3, 4, 7, 7), list(2, 8), list(), list(3, 5, 6));
super_merge(L);  // returns list(1, 2, 3, 3, 4, 5, 6, 7, 7, 8).
```

Your function can make use of the `merge` function (for merge sort) presented in the lectures and given here for your reference:

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

To get full marks for this part, your function must use at least one of the functions `filter`, `map` and `accumulate` in a correct and meaningful way.

```
function super_merge(L) {

    const flatmap = accumulate( merge,
                                null,
                                L);

}
```

Question 6: Active Lists [10 marks]

An ***active list*** is a function that takes an integer number and returns an empty list or a list of length 1. It can be used as an alternative representation of a list, where it takes as argument an element's position in the list, and returns that element in a list of length 1. Note that the first element in a list is at position 0.

6A. [5 marks]

Define a function `make_active_list` that takes a list as its argument and returns an active list that represents the input list.

Example:

```
const act_list = make_active_list(list(8, 3, 5));
act_list(-1); // returns null
act_list(0);  // returns list(8)
act_list(1);  // returns list(3)
act_list(2);  // returns list(5)
act_list(3);  // returns null
```

Note that when the argument passed to `act_list` is negative, or is greater than or equal to the length of the input list to `make_active_list`, the function `act_list` should return an empty list.

```
function make_active_list(L) {
        return x => { if (x < 0 || x >= length(L)) {
                          return null;
                      } else {
                          return list(list_ref(L, x));
                      }};
}
```

6B. [5 marks]

Define a function `map_active_list` that takes as arguments a unary function `op` and an active list and returns an active list that represents the original list with all its elements transformed by `op`.

Example:

```
const act_list1 = make_active_list(list(8, 3, 5));
const act_list2 = map_active_list(x => x + 1, act_list1);
act_list2(-1); // returns null
act_list2(0);  // returns list(9)
act_list2(1);  // returns list(4)
act_list2(2);  // returns list(6)
act_list2(3);  // returns null
```

extract out the list

act_list(1)
(2)
(3)
⋮

act_list is not a list
=> it is a function

```
function map_active_list(op, act_list) {
    const list = map(x => act_list(x), enum_list(0, length(act_list - 1));
    const mapped_list = map(op, list);
    return make_active_list(mapped_list);

    function new_active_list(pos) {
        const x = act_list(pos);
        return is_null(x)
               ? x
               : list(op(head(x)));
    }
    return new_act_list;

    or

    return pos => map(op, act_list(pos));
}
```

0

Question 7: Binary Search Trees [12 marks]

We consider the binary search tree (BST) data structure presented in the lectures. For the subsequent parts of this question, you **must** make good use of the **binary tree abstraction**, consisting of the following functions:

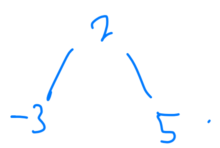
- `is_empty_binary_tree(tree)` — Tests whether the given binary tree `tree` is empty.
- `is_binary_tree(x)` — Returns `true` if `x` is a binary tree and `false` otherwise.
- `left_subtree_of(tree)` — Returns the left subtree of `tree` if `tree` is not empty.
- `value_of(tree)` — Returns the value of the root node of `tree` if `tree` is not empty.
- `right_subtree_of(tree)` — Returns the right subtree of `tree` if `tree` is not empty.
- `make_empty_binary_tree()` — Returns an empty binary tree.
- `make_binary_tree_node(left, value, right)` — Returns a binary tree with left subtree `left`, value `value`, and right subtree `right`.

Do not break this binary tree abstraction in your programs.

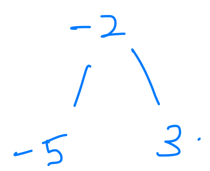
7A. [6 marks]

Complete the function `negate_bst` that takes in a BST of numbers and returns a new BST of numbers that has all the numbers from the input BST negated. The "shape" of the result BST must be a left-right reflection of that of the input BST. For example:

```
const B = make_binary_tree_node(
    make_binary_tree_node(
        make_empty_binary_tree(),
        -3,
        make_empty_binary_tree()),
    2,
    make_binary_tree_node(
        make_empty_binary_tree(),
        5,
        make_empty_binary_tree()));
```




```
negate_bst(B);
/* returns the same tree as:
make_binary_tree_node(
    make_binary_tree_node(
        make_empty_binary_tree(),
        -5,
        make_empty_binary_tree()),
    -2,
    make_binary_tree_node(
        make_empty_binary_tree(),
        3,
        make_empty_binary_tree()));
*/
```



```
function negate_bst(bst) {

    return is_empty_binary_tree(bst)
           ? make_empty_binary_tree(bst);

         : make_binary_tree_node(
                               negate_bst(right_subtree(bst))
                               -(value(bst))
                               negate_bst(left_subtree(bst)) )

}
```

7B. [6 marks]

Complete the function `accumulate_bst` that behaves like `accumulate` but can only work on BST. Note that the order of application of the input operation `op` must start from the largest value in the BST, in descending order, to the smallest value. For example, if the input BST `B` has the values 1, 2, 3, 4, 5, 6 and 7, then, regardless of the "shape" of the BST `B`, the call `accumulate_bst(pair, null, B)` should return `list(1,2,3,4,5,6,7)`, and the call `accumulate_bst((x, y) => x + y, 0, B)` should return 28.

```
function accumulate_bst(op, initial, bst) {

    function create_list( T ) {
        if( ! is_empty_binary_tree(T) ){
            const R = right_subtree_of(T);
            const L = left_subtree_of(T);

            if( value_of(R) <= value_of(T) &&
                value_of(T) <= value of(L)    ){
                    return list( create_list(R),
                                 value_of(bst),
                                 create_list(L));
            }else if( value_of(L) <= value_of(T) &&
                      value_of(T) <= value of(R)    ){
                    return list( create_list(L),
                                 value_of(bst),
                                 create_list(R));
            } else if ( value_of(T) <= value_of(R) &&
                        value_of(T) <= value of(L)    ){
                    return value_of(R) <= value_of(L)
                        ? list( create_list(T),
                                value_of(R),
                                create_list(L));
                        : list( create_list(T),
                                value_of(L),
                                create_list(R));
            } else if ( value_of(R) <= value_of(T) &&
                        value_of(L) <= value of(T)    ){
                    return value_of(R) <= value_of(L)
                        ? list( create_list(R),
                                value_of(L),
                                create_list(T));
                        : list( create_list(L),
                                value_of(R),
                                create_list(T));
            }.
        } else {}
    }

    return accumulate( op, initial, create_list(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));
    }.

}
```

iterating
fuck
recursion.

accumulate right branch.

add op to value & right branch

continue down left branch

Question 8: Permutations, Again! [12 marks]

8A. [6 marks]

Complete the function `insertions(x, ys)` that returns all possible ways to insert `x` into the list `ys`, without changing the relative order of the elements in `ys`. For example:

```
insertions(4, list(1, 2, 3));       inserted at diff pos in the list.
// returns list(list(4,1,2,3), list(1,4,2,3), list(1,2,4,3), list(1,2,3,4)).
```

Your function can make use of the `take` and `drop` functions (for merge sort) presented in the lectures/reflections. They are given here for your reference:

```
// put the first n elements of xs into a list
function take(xs, n) {
    return (n === 0) ? null : pair(head(xs), take(tail(xs), n - 1));
}

// drop the first n elements from the list and return the rest
function drop(xs, n) {
    return (n === 0) ? xs : drop(tail(xs), n - 1);
}
```

```
function insertions(x, ys) {   list(
        1st:  append(list(x), ys)                                    0
        2nd:  append(take(ys,1),
                  append(list(x), drop(ys,1)))                       1
        3rd:  append(take(ys,2),
                  append(list(x), drop(ys,2)).                       2
        4th ...                                                      3.
           )                                                         length(ys).

    function helper(count) {
        return count === length(ys) + 1
               ? null
               : pair(append(take(ys, count),
                             append(list(x), drop(ys, count))),
                      helper(count+1));
    }

    return helper(0);
}
```

ways of inserts x into all of ys.

8B. [6 marks]

Complete the function `permutations` that takes as argument a list of distinct numbers and returns a list of all permutations of the input numbers. Each permutation is a list of numbers. The permutations in the result list can be in any order. For example:

```
permutations(list(1, 2, 3));
// Example result: list(list(1,2,3), list(2,1,3), list(2,3,1),
//                      list(1,3,2), list(3,1,2), list(3,2,1)).
```

To get full marks for this part, your function **must make use of the `insertions` function** from Part A in a correct and meaningful way.

```
function permutations(xs) {

    append( insertion( list_ref(xs, 0), drop(xs, 1) ),
            insertion( list_ref(xs, 1), drop(xs, 2) ) ).        0
                                                                 1
                                                                 2/
    function helper(count) {                                     3
        return count === length(xs) - 2
            ? list()
            : append( insertion( list_ref(xs, count),
                                 drop(xs, count + 1) ),
                      helper(count + 1));
    }
    return helper(0);          ✗ gives only half the answer.

base case:
    if (is_null(xs)) {
        return list(null);
    } else {                   obtain all permutation of the back of the list
        const s = permutation(tail(xs));
        const t = map(ys => insertions(head(xs), ys), s);
        return accumulate(append, null, t);   insert current head into all of them
                              list.
    }
}
```

(1, 2, 3, 4)
2, 3, 4 ... 2, 4, 3 ...
for each elem inside, we insert function of each own.
(1, 2, 3, 4)
(2, 1, 3, 4)
... (2, 3, 4, 1))

END OF QUESTIONS

Appendix

List Support

The following list processing functions are supported:

- `pair(x, y)`: Makes a pair from `x` and `y`.
- `is_pair(x)`: Returns `true` if `x` is a pair and `false` otherwise.
- `head(x)`: Returns the head (first component) of the pair `x`.
- `tail(x)`: Returns the tail (second component) of the pair `x`.
- `is_null(xs)`: Returns `true` if `xs` is the empty list, and `false` otherwise.
- `is_list(x)`: Returns `true` if `x` is a list as defined in the lectures, and `false` otherwise. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of the chain of `tail` operations that can be applied to `x`.
- `list(x1, x2,..., xn)`: Returns a list with `n` elements. The first element is `x1`, the second `x2`, etc. Iterative process; time: $O(n)$, space: $O(n)$, since the constructed list data structure consists of n pairs, each of which takes up a constant amount of space.
- `length(xs)`: Returns the length of the list `xs`. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.
- `map(f, xs)`: Returns a list that results from list `xs` by element-wise application of `f`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `build_list(n, f)`: Makes a list with `n` elements by applying the unary function `f` to the numbers 0 to `n - 1`. Recursive process; time: $O(n)$, space: $O(n)$.
- `for_each(f, xs)`: Applies `f` to every element of the list `xs`, and then returns `true`. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.
- `list_to_string(xs)`: Returns a string that represents list `xs` using the text-based box-and-pointer notation `[...]`.
- `reverse(xs)`: Returns list `xs` in reverse order. Iterative process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`. The process is iterative, but consumes space $O(n)$ because of the result list.
- `append(xs, ys)`: Returns a list that results from appending the list `ys` to the list `xs`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `member(x, xs)`: Returns first postfix sublist whose head is identical to `x` (===); returns `null` if the element does not occur in the list. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.
- `remove(x, xs)`: Returns a list that results from `xs` by removing the first item from `xs` that is identical (===) to `x`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `remove_all(x, xs)`: Returns a list that results from `xs` by removing all items from `xs` that are identical (===) to `x`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `filter(pred, xs)`: Returns a list that contains only those elements for which the one argument function `pred` returns `true`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `enum_list(start, end)`: Returns a list that enumerates numbers starting from `start` using a step size of 1, until the number exceeds (>) `end`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`. For example, `enum_list(2, 5)` returns the list `list(2, 3, 4, 5)`.
- `list_ref(xs, n)`: Returns the element of list `xs` at position `n`, where the first element has index 0. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.

- `accumulate(op, initial, xs)`: Applies binary function `op` to the elements of `xs` from right-to-left order, first applying `op` to the last element and the value `initial`, resulting in r_1, then to the second-last element and r_1, resulting in r_2, etc, and finally to the first element and r_{n-1}, where n is the length of the list. Thus, `accumulate(op, zero, list(1,2,3))` results in `op(1, op(2, op(3, zero)))`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`, assuming `op` takes constant time.

(Scratch Paper. Do not tear off.)

(Scratch Paper. Do not tear off.)

NATIONAL UNIVERSITY OF SINGAPORE

CS1101S — PROGRAMMING METHODOLOGY

(AY2018/2019 SEMESTER 1)

MIDTERM ASSESSMENT (ADAPTED TO AY2020/21 IN 9/2020)

Time Allowed: **1 Hour 45 Minutes**

SOLUTIONS

INSTRUCTIONS

1. This assessment paper contains **EIGHT (8)** questions and comprises **EIGHTEEN (18)** printed pages, including this page.
2. The full score of this paper is **75 marks**.
3. This is a **CLOSED BOOK** assessment, but you are allowed to bring in one A4 sheet of notes (handwritten or printed on both sides).
4. Answer **ALL** questions **within the space provided** in this booklet.
5. Where programs are required, write them in the **Source §2** language.
6. Write legibly with a **pen or pencil**. **Untidiness will be penalized**.
7. Do not tear off any pages from this booklet.
8. Write your **Student Number** below **USING A PEN**. Do not write your name.
9. Also write down your **Studio Group Number** in the provided box, if you can remember it.

(write with a pen)

Student No.: | | | | | | | | | |

Studio Group No. (leave blank if cannot remember): | |

This portion is for examiner's use only

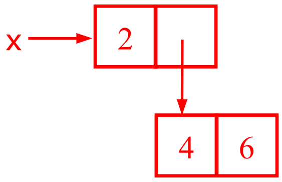
Q#	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Σ**
MAX	10	4	8	9	10	10	12	12	75
SC									

Question 1: Box-and-Pointer Diagrams [10 marks]

Draw the box-and-pointer diagram for the value of x after the evaluation of each of the following programs. Clearly show where x is pointing to.

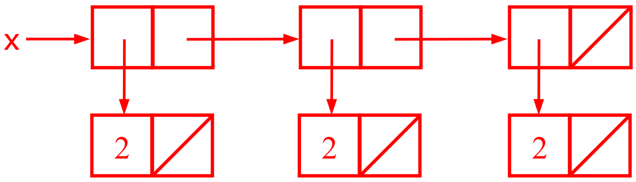
1A. [2 marks]

```
const x = pair(2, pair(4, 6));
```



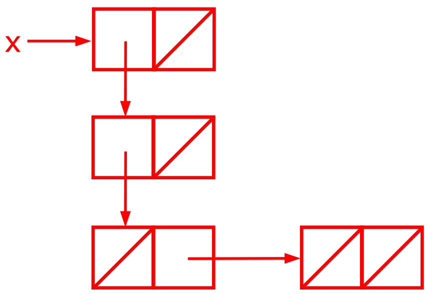
1B. [2 marks]

```
const p = list(2);
const q = pair(2, list());
const x = list(p, q, pair(2, null));
```



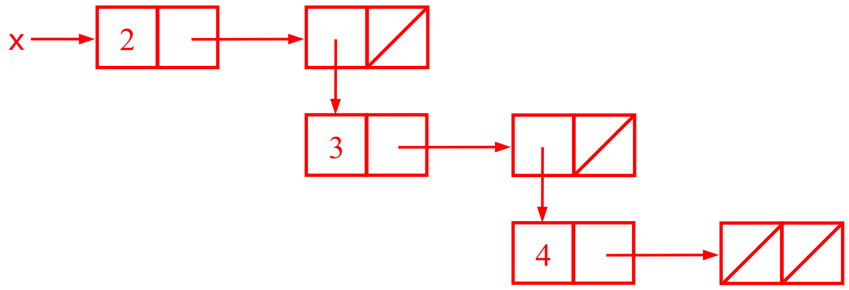
1C. [3 marks]

```
const x = list(list(list(null, null)));
```



1D. [3 marks]

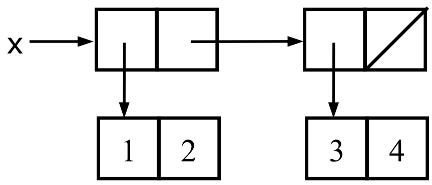
```
const x = accumulate(list, null, list(2, 3, 4));
```



Question 2: Making Pairs [4 marks]

Write a Source §2 program that produces exactly the pairs shown in each of the following box-and-pointer diagrams. At the end of the execution of your program, the constant x must refer to the pair as shown in the diagram.

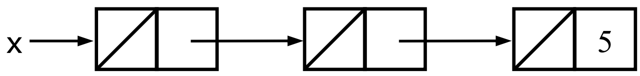
2A. [2 marks]




```
// SOLUTION 1:
const x = list(pair(1, 2), pair(3, 4));

// SOLUTION 2:
const x = pair(pair(1, 2), pair(pair(3, 4), null));
```

2B. [2 marks]




```
const x = pair(null, pair(null, pair(null, 5)));
```

Question 3: Let's Be Logical [8 marks]

3A. [3 marks]

Consider the following function

```
function hoo(f, g, h, x) {
    if (f(x) || (g(x) && h(x))) {
        return 100;
    } else {
        return 50;
    }
}
```

Rewrite the function hoo without using the keyword **if** and without any logical operators (&&, ||, !). Your function must have the same order of evaluations and produce the same result as the original.

```
function hoo(f, g, h, x) {

    return f(x) ? 100
                : g(x) ? (h(x) ? 100 : 50) : 50;





}
```

3B. [5 marks]

Consider the following program

```
function gee(n) {
    return n <= 0 ? true : (false || gee(n - 1));
}
gee(4);
```

(i) What kind of process does the program give rise to?
(ii) Justify your answer by showing the evaluation steps (hint: the substitution model).

(i) Iterative process.

(ii) There is no deferred operation in the evaluation, therefore it is an iterative process.

```
gee(4)
→ false || gee(3)
→ gee(3)
→ false || gee(2)
→ gee(2)
→ false || gee(1)
→ gee(1)
→ false || gee(0)
→ gee(0)
→ true
```

Question 4: Recursive vs Iterative Processes [9 marks]

The `get_sublist` function takes as arguments two integer numbers, `start` and `end`, and a list `L`, and returns a list containing the element(s) of `L` from position `start` to position `end`, including both. The first element of `L` is at position 0. You can assume that 0 ≤ `start` ≤ `end` < `length(L)`.

Example :

```
const L = list(11, 12, 13, 14, 15, 16, 17, 18);
get_sublist(3, 5, L);  // returns list(14, 15, 16).
```

Complete the following two implementations of `get_sublist` that give rise to **(A)** a **recursive process** and **(B)** an **iterative process**. Both implementations' runtime should have an order of growth of O(n), where n is the length of the list `L`.

4A. [4 marks] Recursive version.

```
function get_sublist(start, end, L) {

    function helper(pos, ys) {
        if (pos < start) {

            return helper(pos + 1, tail(ys));

        } else if (pos <= end) {

            return pair(head(ys), helper(pos + 1, tail(ys)));

        } else { return null; }
    }
    return helper(0, L);
}
```

4B. [5 marks] Iterative version.

```
function get_sublist(start, end, L) {

    function helper(pos, ys, result) {

        // SOLUTION 1:
        if (pos < start) {
            return helper(pos + 1, tail(ys), result);
        } else if (pos <= end) {
            return helper(pos + 1, tail(ys), pair(head(ys), result));
        } else {
            return reverse(result);
        }

        // SOLUTION 2:
        return pos < start
            ? helper(pos + 1, tail(ys), result)
            : pos <= end
                ? helper(pos + 1, tail(ys), pair(head(ys), result))
                : reverse(result);

    }
    return helper(0, L, null);
}
```

Question 5: The Benefits of Being Sorted [10 marks]

5A. [5 marks]

We represent a ***set*** of numbers using a list of **distinct** numbers **sorted** in **ascending order**. Complete the function, `is_subset(S, T)`, to determine whether set S is a subset of set T, which is true only if every element in S is also an element of T. We assume that $0 \leq N_S$ and $0 \leq N_T$, where N_S and N_T are the number of elements in S and T respectively. If $N_S = 0$, then S is always a subset of T. Your program should exploit the fact that both lists are sorted; its runtime should have an order of growth of $O(N_S + N_T)$.

```
function is_subset(S, T) {

    if (is_null(S)) {
        return true;
    } else if (is_null(T)) {
        return false;
    } else if (head(S) < head(T)) {
        return false;
    } else if (head(S) === head(T)) {
        return is_subset(tail(S), tail(T));
    } else {
        return is_subset(S, tail(T));
    }
}
```

5B. [5 marks]

Complete the following function, `super_merge`, that takes in a list `L` of **one or more** lists of numbers, where the numbers in each list are in **ascending order**, and returns a single list of all the numbers from `L`, sorted in **ascending order**. For example:

```
const L = list(list(1, 3, 4, 7, 7), list(2, 8), list(), list(3, 5, 6));
super_merge(L);  // returns list(1, 2, 3, 3, 4, 5, 6, 7, 7, 8).
```

Your function can make use of the `merge` function (for merge sort) presented in the lectures and given here for your reference:

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

To get full marks for this part, your function must use at least one of the functions `filter`, `map` and `accumulate` in a correct and meaningful way.

```
function super_merge(L) {

// SOLUTION 1:

    return accumulate(merge, null, L);


// SOLUTION 2:

    return accumulate((x, ys) => merge(x, ys), null, L);

}
```

Question 6: Active Lists [10 marks]

An ***active list*** is a function that takes an integer number and returns an empty list or a list of length 1. It can be used as an alternative representation of a list, where it takes as argument an element's position in the list, and returns that element in a list of length 1. Note that the first element in a list is at position 0.

6A. [5 marks]

Define a function `make_active_list` that takes a list as its argument and returns an active list that represents the input list.

Example:

```
const act_list = make_active_list(list(8, 3, 5));
act_list(-1); // returns null
act_list(0);  // returns list(8)
act_list(1);  // returns list(3)
act_list(2);  // returns list(5)
act_list(3);  // returns null
```

Note that when the argument passed to `act_list` is negative, or is greater than or equal to the length of the input list to `make_active_list`, the function `act_list` should return an empty list.

```
function make_active_list(L) {

    const len = length(L);

    return pos => (pos < 0 || pos >= len)
        ? null
        : list(list_ref(L, pos));

}
```

6B. [5 marks]

Define a function `map_active_list` that takes as arguments a unary function `op` and an active list and returns an active list that represents the original list with all its elements transformed by `op`.

Example:

```
const act_list1 = make_active_list(list(8, 3, 5));
const act_list2 = map_active_list(x => x + 1, act_list1);
act_list2(-1); // returns null
act_list2(0);  // returns list(9)
act_list2(1);  // returns list(4)
act_list2(2);  // returns list(6)
act_list2(3);  // returns null
```

```
function map_active_list(op, act_list) {

// SOLUTION 1:

    function new_act_list(pos) {
        const x = act_list(pos);
        return is_null(x) ? x : list(op(head(x)));
    }
    return new_act_list;



// SOLUTION 2:

    return pos => map(op, act_list(pos));

}
```

Question 7: Binary Search Trees [12 marks]

We consider the binary search tree (BST) data structure presented in the lectures. For the subsequent parts of this question, you **must** make good use of the **binary tree abstraction**, consisting of the following functions:

- `is_empty_binary_tree(tree)` — Tests whether the given binary tree `tree` is empty.
- `is_binary_tree(x)` — Returns `true` if `x` is a binary tree and `false` otherwise.
- `left_subtree_of(tree)` — Returns the left subtree of `tree` if `tree` is not empty.
- `value_of(tree)` — Returns the value of the root node of `tree` if `tree` is not empty.
- `right_subtree_of(tree)` — Returns the right subtree of `tree` if `tree` is not empty.
- `make_empty_binary_tree()` — Returns an empty binary tree.
- `make_binary_tree_node(left, value, right)` — Returns a binary tree with left subtree `left`, value `value`, and right subtree `right`.

Do not break this binary tree abstraction in your programs.

7A. [6 marks]

Complete the function `negate_bst` that takes in a BST of numbers and returns a new BST of numbers that has all the numbers from the input BST negated. The "shape" of the result BST must be a left-right reflection of that of the input BST. For example:

```
const B = make_binary_tree_node(
    make_binary_tree_node(
        make_empty_binary_tree(),
        -3,
        make_empty_binary_tree()),
    2,
    make_binary_tree_node(
        make_empty_binary_tree(),
        5,
        make_empty_binary_tree()));

negate_bst(B);
/* returns the same tree as:
make_binary_tree_node(
    make_binary_tree_node(
        make_empty_binary_tree(),
        -5,
        make_empty_binary_tree()),
    -2,
    make_binary_tree_node(
        make_empty_binary_tree(),
        3,
        make_empty_binary_tree()));
*/
```

```
function negate_bst(bst) {

// SOLUTION 1:

    if (is_empty_binary_tree(bst)) {
        return make_empty_binary_tree();
    } else {
        return make_binary_tree_node(
            negate_bst(right_subtree_of(bst)),
            -1 * value_of(bst),
            negate_bst(left_subtree_of(bst)));
    }


// SOLUTION 2:

    return is_empty_binary_tree(bst)
        ? make_empty_binary_tree()
        : make_binary_tree_node(
            negate_bst(right_subtree_of(bst)),
            -1 * value_of(bst),
            negate_bst(left_subtree_of(bst)));

}
```

7B. [6 marks]
Complete the function `accumulate_bst` that behaves like `accumulate` but can only work on BST. Note that the order of application of the input operation `op` must start from the largest value in the BST, in descending order, to the smallest value. For example, if the input BST `B` has the values 1, 2, 3, 4, 5, 6 and 7, then, regardless of the "shape" of the BST `B`, the call `accumulate_bst(pair, null, B)` should return `list(1,2,3,4,5,6,7)`, and the call `accumulate_bst((x, y) => x + y, 0, B)` should return 28.

```
function accumulate_bst(op, initial, bst) {


// SOLUTION 1:

    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));
    }


// SOLUTION 2:

    function listify_bst(b) {
        if (is_empty_binary_tree(b)) {
            return null;
        } else {
            const left_list = listify_bst(left_subtree_of(b));
            const value = value_of(b);
            const right_list = listify_bst(right_subtree_of(b));
            return append(left_list, pair(value, right_list));
        }
    }
    return accumulate(op, initial, listify_bst(bst));

}
```

Question 8: Permutations, Again! [12 marks]

8A. [6 marks]

Complete the function `insertions(x, ys)` that returns all possible ways to insert `x` into the list `ys`, without changing the relative order of the elements in `ys`. For example:

```
insertions(4, list(1, 2, 3));
// returns list(list(4,1,2,3), list(1,4,2,3), list(1,2,4,3), list(1,2,3,4)).
```

Your function can make use of the `take` and `drop` functions (for merge sort) presented in the lectures/reflections. They are given here for your reference:

```
// put the first n elements of xs into a list
function take(xs, n) {
    return (n === 0) ? null : pair(head(xs), take(tail(xs), n - 1));
}

// drop the first n elements from the list and return the rest
function drop(xs, n) {
    return (n === 0) ? xs : drop(tail(xs), n - 1);
}
```

```
function insertions(x, ys) {

// SOLUTION 1:
    return map(k => append(take(ys, k), pair(x, drop(ys, k))),
               enum_list(0, length(ys)));

// SOLUTION 2:
    function helper(k, result) {
        if (k < 0) {
            return result;
        } else {
            const u = append(take(ys, k), pair(x, drop(ys, k)));
            return helper(k - 1, pair(u, result));
        }
    }
    return helper(length(ys), null);

// SOLUTION 3:
    const len = length(ys);
    function helper(k) {
        return (k > len)
            ? null
            : pair(append(take(ys, k), pair(x, drop(ys, k))),
                   helper(k + 1));
    }
    return helper(0);

// SOLUTION 4:
    return is_null(ys)
        ? list(list(x))
        : pair(pair(x, ys),
               map(i => pair(head(ys), i), insertions(x, tail(ys))));
}
```

8B. [6 marks]

Complete the function `permutations` that takes as argument a list of distinct numbers and returns a list of all permutations of the input numbers. Each permutation is a list of numbers. The permutations in the result list can be in any order. For example:

```
permutations(list(1, 2, 3));
// Example result: list(list(1,2,3), list(2,1,3), list(2,3,1),
//                      list(1,3,2), list(3,1,2), list(3,2,1)).
```

To get full marks for this part, your function **must make use of the `insertions` function** from Part A in a correct and meaningful way.

```
function permutations(xs) {

// SOLUTION 1:

    if (is_null(xs)) {
        return list(null);
    } else {
        const s = permutations(tail(xs));
        const t = map(ys => insertions(head(xs), ys), s);
        return accumulate(append, null, t);
    }


// SOLUTION 2:

    return accumulate((x, ps) => accumulate((p, qs) => append(insertions(x, p),
                                                                  qs),
                                            null,
                                            ps),
                      list(null),
                      xs);


// SOLUTION 3:

    return accumulate((x, ps) => accumulate(append,
                                            null,
                                            map(p => insertions(x, p), ps)),
                      list(null),
                      xs);

}
```

——— **END OF QUESTIONS** ———

Appendix

List Support

The following list processing functions are supported:

- `pair(x, y)`: Makes a pair from `x` and `y`.
- `is_pair(x)`: Returns `true` if `x` is a pair and `false` otherwise.
- `head(x)`: Returns the head (first component) of the pair `x`.
- `tail(x)`: Returns the tail (second component) of the pair `x`.
- `is_null(xs)`: Returns `true` if `xs` is the empty list, and `false` otherwise.
- `is_list(x)`: Returns `true` if `x` is a list as defined in the lectures, and `false` otherwise. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of the chain of `tail` operations that can be applied to `x`.
- `list(x1, x2,..., xn)`: Returns a list with `n` elements. The first element is `x1`, the second `x2`, etc. Iterative process; time: $O(n)$, space: $O(n)$, since the constructed list data structure consists of n pairs, each of which takes up a constant amount of space.
- `length(xs)`: Returns the length of the list `xs`. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.
- `map(f, xs)`: Returns a list that results from list `xs` by element-wise application of `f`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `build_list(n, f)`: Makes a list with `n` elements by applying the unary function `f` to the numbers 0 to `n - 1`. Recursive process; time: $O(n)$, space: $O(n)$.
- `for_each(f, xs)`: Applies `f` to every element of the list `xs`, and then returns `true`. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.
- `list_to_string(xs)`: Returns a string that represents list `xs` using the text-based box-and-pointer notation `[...]`.
- `reverse(xs)`: Returns list `xs` in reverse order. Iterative process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`. The process is iterative, but consumes space $O(n)$ because of the result list.
- `append(xs, ys)`: Returns a list that results from appending the list `ys` to the list `xs`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `member(x, xs)`: Returns first postfix sublist whose head is identical to `x` (===); returns `null` if the element does not occur in the list. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.
- `remove(x, xs)`: Returns a list that results from `xs` by removing the first item from `xs` that is identical (===) to `x`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `remove_all(x, xs)`: Returns a list that results from `xs` by removing all items from `xs` that are identical (===) to `x`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `filter(pred, xs)`: Returns a list that contains only those elements for which the one argument function `pred` returns `true`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `enum_list(start, end)`: Returns a list that enumerates numbers starting from `start` using a step size of 1, until the number exceeds (>) `end`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`. For example, `enum_list(2, 5)` returns the list `list(2, 3, 4, 5)`.
- `list_ref(xs, n)`: Returns the element of list `xs` at position `n`, where the first element has index 0. Iterative process; time: $O(n)$, space: $O(1)$, where n is the length of `xs`.

- `accumulate(op, initial, xs)`: Applies binary function `op` to the elements of `xs` from right-to-left order, first applying `op` to the last element and the value `initial`, resulting in r_1, then to the second-last element and r_1, resulting in r_2, etc, and finally to the first element and r_{n-1}, where n is the length of the list. Thus, `accumulate(op, zero, list(1,2,3))` results in `op(1, op(2, op(3, zero)))`. Recursive process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`, assuming `op` takes constant time.

(Scratch Paper. Do not tear off.)

(Scratch Paper. Do not tear off.)