CS1101S — Programming Methodology

AY2022/2023 Semester 1

Midterm Assessment

SOLUTIONS

Time allowed: 1 hour and 30 minutes

INSTRUCTIONS

1. This **QUESTION PAPER** contains **22** Questions in **8** Sections, and comprises **XX** printed pages, including this page.

2. The **ANSWER SHEET** comprises **XX** printed pages.

3. Use a pen or pencil to **write** your **Student Number** in the designated space on the front page of the **ANSWER SHEET**, and **shade** the corresponding circle **completely** in the grid for each digit or letter. DO NOT WRITE YOUR NAME!

4. You must **submit only** the **ANSWER SHEET** and no other documents. Do not tear off any pages from the ANSWER SHEET.

5. All questions must be answered in the space provided in the **ANSWER SHEET**; no extra sheets will be accepted as answers.

6. Write legibly with a **pen** or **pencil** (do not use red color). Untidiness will be penalized.

7. For **multiple choice questions (MCQ)**, **shade** in the **circle** of the correct answer **completely**.

8. The full score of this assessment is **100** marks.

9. This is a **Closed-Book** assessment, but you are allowed to bring with you one double-sided **A4 / letter-sized sheet** of handwritten or printed **notes**.

10. Where programs are required, write them in the **Source §2** language. A **reference** of some of the **pre-declared functions** is given in the **Appendix** of the Question Paper.

11. In any question, unless it is specifically allowed, your answer **must not use functions** given in, or written by you for, other questions.

Section A: List and Box Notations [6 marks]

(1) [3 marks] (MCQ)

What is the result of evaluating the following Source program in *box notation*?

```
pair(list(1, 2), null);
```

A. `[[1, 2], null]`

B. `[[[1, 2], null], null]`

C. `[[1, 2, null], null]`

D. `[[1, [2, [null]]], null]`

E. `[[1, [2, null]], null]` (answer)

F. It is impossible to be done in box notation

(2) [3 marks] (MCQ)

What is the result of evaluating the following Source program in *list notation*?

```
const lst = list(pair(1, 2), list(3));
pair(head(lst), lst);
```

A. `list([1, 2], list(1, 2, list(3)))`

B. `list([1, 2], [1, 2, 3])`

C. `list([1, 2], [1, 2], list(3))` (answer)

D. `list([1, 2], lst)`

E. It is impossible to be done in list notation because it is a pair, not a list

F. It is impossible to be done in list notation because the pair does not end with a `null`

Section B: List Processing [11 marks]

You are tasked to develop a program to record all space recruit information in Source Academy. Each space recruit's record has the following data: an identification number ID (integer), the recruit type (boolean — `true` for Cadet and `false` for Avenger) and the age (integer). The records are stored as elements of a ***recruit-list***, where each record is a list of three elements: ID, recruit type and age. The following example shows one such recruit-list:

```
const recruit_info = list(list(150, true, 19),
                          list(101, false, 21), list(122, false, 20));
// Three recruits: 1 Cadet and 2 Avengers, aged 19, 21 and 20 respectively
```

(3) [5 marks]

Complete the implementation of function `average_age`, which takes as argument a non-empty recruit-list, and returns the average age of all the recruits in the recruit-list. The following shows an example usage of this function.

```
average_age(recruit_info);  // returns 20
```

```
function average_age(recruits) {

    return accumulate((x, y) => head(tail(tail(x))) + y, 0, recruits)
           / length(recruits);









}
```

(4) [6 marks]

Complete the implementation of function `split_type`, which takes as argument a recruit-list and returns a pair, where the head of the pair is a recruit-list that contains only the Cadets, while the tail of the pair is a recruit-list that contains only the Avengers. The elements in the two recruit-lists in the result can be in any order. The following shows an example usage of this function.

```
split_type(recruit_info);
// returns [ list( list(150, true, 19) ),
//           list( list(101, false, 21), list(122, false, 20) ) ]
```

Write your answer only in the dashed-line boxes.

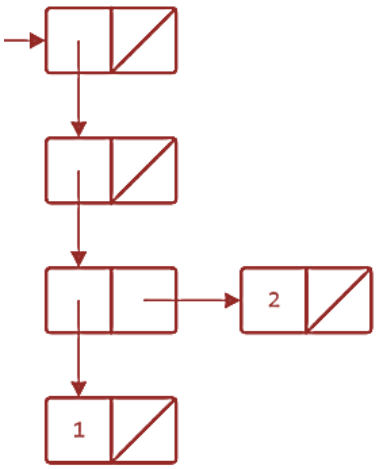
```
function split_type(recruits) {
    return accumulate(
        (x, y) => head(tail(x))
            ? pair( pair(x, head(y)), tail(y) )

            : pair( head(y), pair(x, tail(y)) ),

        pair(null, null),
        recruits);
}
```

Section C: Boxes and Pointers [14 marks]

(5) [4 marks]

Draw the box and pointer diagram of the result of evaluating the following program.

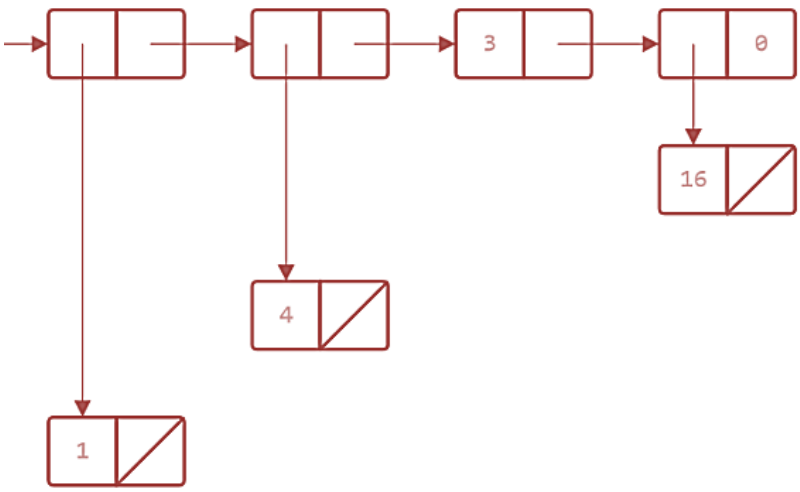
```
list(list(pair(list(1), list(2))));
```



(6) [5 marks]

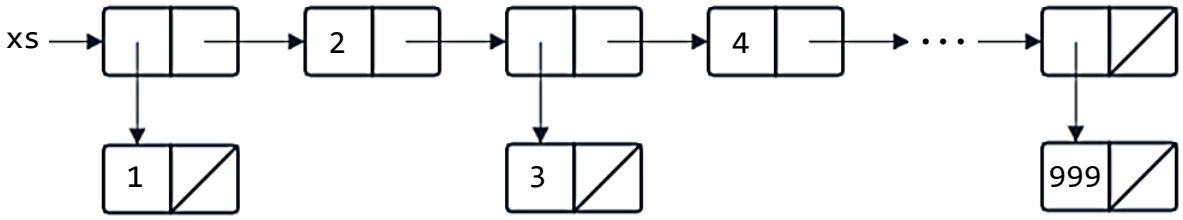
Draw the box and pointer diagram of the result of evaluating the following program.

```
accumulate((x, y) => pair(x, y), 0,
           map(x => x % 3 === 0 ? x : list(x * x), enum_list(1, 4)));
```



(7) [5 marks]

For the following box-and-pointer diagram, write a Source program such that at the end of the evaluation of your program, the name `xs` will have the value as shown in the diagram. You must not use any ellipsis (...) in your program. Hint: You can create a list of integers from 1 to 999 using `enum_list(1, 999)`.




```
const xs = map(x => x % 2 === 0 ? x : list(x), enum_list(1, 999));
```

Section D: Orders of Growth [12 marks]

What is the **order of growth** of the **running time** of each of the following functions in terms of N using the Θ notation? Note that N is a positive integer value.

(8) [3 marks] (MCQ)

```
function fun(N) {
    return N <= 1  ?  1  :  N * fun(N - 1000000);
}
```

A. $\Theta(1)$

B. $\Theta(\log N)$

C. $\Theta(N)$ **(answer)**

D. $\Theta(N \log N)$

E. $\Theta(N^2)$

F. $\Theta(N^2 \log N)$

G. $\Theta(N^3)$

H. $\Theta(k^N)$ where k is some constant greater than 1

(9) [3 marks] (MCQ)

```
function fun(N) {
    return N <= 1  ?  1  :  1000000 * N * fun(N * 0.99);
}
```

A. $\Theta(1)$

B. $\Theta(\log N)$ **(answer)**

C. $\Theta(N)$

D. $\Theta(N \log N)$

E. $\Theta(N^2)$

F. $\Theta(N^2 \log N)$

G. $\Theta(N^3)$

H. $\Theta(k^N)$ where k is some constant greater than 1

(10) [3 marks] (MCQ)

```
function fun(N) {
   if (N <= 1) {
      return 1;
   } else {
      const len = length(filter(x => x > N, enum_list(1, math_floor(N))));
      return fun(N / 2) + fun(N / 2) + len;
   }
}
```

A. $\Theta(1)$

B. $\Theta(\log N)$

C. $\Theta(N)$

D. $\Theta(N \log N)$ **(answer)**

E. $\Theta(N^2)$

F. $\Theta(N^2 \log N)$

G. $\Theta(N^3)$

H. $\Theta(k^N)$ where k is some constant greater than 1

(11) [3 marks] (MCQ)

What is the order of growth of the running time of the following `selection_sort` function in terms of N, where N is the length of the input list `xs`?

```
function largest(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 = largest(xs);
        return append(selection_sort(remove(x, xs)), list(x));
    }
}
```

A. $\Theta(1)$

B. $\Theta(\log N)$

C. $\Theta(N)$

D. $\Theta(N \log N)$

E. $\Theta(N^2)$ **(answer)**

F. $\Theta(N^2 \log N)$

G. $\Theta(N^3)$

H. $\Theta(2^N)$

Section E: Vector Operations [18 marks]

Consider two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ both with the same length n:

$$\boldsymbol{x}: [x_1, x_2, \ldots, x_n]$$
$$\boldsymbol{y}: [y_1, y_2, \ldots, y_n]$$

The ***dot product*** of these two vectors is defined as

$$\boldsymbol{x} \bullet \boldsymbol{y} = x_1y_1 + x_2y_2 + \ldots + x_ny_n$$

The ***Euclidean distance*** between these two vectors is defined as

$$D(\boldsymbol{x}, \boldsymbol{y}) = ((y_1 - x_1)^2 + (y_2 - x_2)^2 + \ldots + (y_n - x_n)^2)^{1/2}.$$

(12) [6 marks]

Now, consider two vectors $\boldsymbol{v}_1$: [1, 2, 3] and $\boldsymbol{v}_2$: [1, 2, 3, 4]. These vectors can themselves be represented using Lists of Numbers:

```
const v1 = list(1, 2, 3);
const v2 = list(1, 2, 3, 4);
```

Two vectors are equal if all their elements are equal, and they are of the same length:

```
equal_vec(v1, v2); // evaluates to false
equal_vec(v1, v1); // evaluates to true
```

Complete the implementation of the `equal_vec` function, whose arguments `x` and `y` are vectors. You **must not use any pre-declared functions** except `is_null`, `head`, `tail`, and `is_pair`.

```
function equal_vec(x, y) {

    return is_pair(x) && is_pair(y)
           ? head(x) === head(y) && equal_vec(tail(x), tail(y))
           : is_null(x) && is_null(y);









}
```

(13) [6 marks]

Now, consider the dot product of two vectors:

```
const v3 = list(1, 2, -3);
const v4 = list(2, 0,  3);

dot_product(v3, v4); // evaluates to -7
```

Complete the implementation of the `dot_product` function, whose arguments `x` and `y` are vectors of equal length. You **must not use any pre-declared functions** except `is_null`, `head`, `tail`, and `is_pair`.

```
function dot_product(x, y) {

    return is_null(x)
           ? 0
           : head(x) * head(y) + dot_product(tail(x), tail(y));


}
```

(14) [6 marks]

Now, consider the Euclidean distance between two vectors:

```
const v5 = list(1, 5, 3);
const v6 = list(4, 1, 3);

euclidean_distance(v5, v6); // evaluates to 5
euclidean_distance(v5, v5); // evaluates to 0
```

Complete the implementation of the `euclidean_distance` function, whose arguments `x` and `y` are vectors of equal length. You **must not use any pre-declared functions** except `is_null`, `head`, `tail`, `is_pair`, and `math_sqrt`.

```
function euclidean_distance(x, y) {

    function helper(x, y) {
        return is_null(x)
               ? 0
               : (head(y) - head(x)) * (head(y) - head(x))
                 + helper(tail(x), tail(y));
    }
    return math_sqrt(helper(x, y));

    // Alternative solution:
    const sqr = x => x * x;
    return is_null(x)
           ? 0
           : math_sqrt( sqr(head(y) - head(x))
                        + sqr(euclidean_distance(tail(x), tail(y))) );
}
```

Section F: Binary Search Trees [9 marks]

In this section, we will use the ***binary tree abstraction*** that you have seen before:

- `is_empty_tree(tree)` — Tests whether the given binary tree `tree` is empty.
- `is_tree(x)` — Checks if `x` is a binary tree.
- `left_branch(tree)` — Returns the left subtree of `tree` if `tree` is not empty.
- `entry(tree)` — Returns the value of the entry of `tree` if `tree` is not empty.
- `right_branch(tree)` — Returns the right subtree of `tree` if `tree` is not empty.
- `make_empty_tree()`— Returns an empty binary tree.
- `make_tree(value, left, right)` — Returns a binary tree with entry `value`, left subtree `left`, and right subtree `right`.

(15) [5 marks] (MCQ)

Consider the tree `bst` created as the result of evaluating the following Source program. Select the correct values for `A`, `B`, `C`, `D`, and `E` to make the tree `bst` a binary search tree.

```
const t1 = make_tree(A, make_empty_tree(), make_empty_tree());
const t2 = make_tree(B, make_empty_tree(), make_empty_tree());
const t4 = make_tree(C, make_empty_tree(), make_empty_tree());
const t3 = make_tree(D, t1, t2);
const bst = make_tree(E, t3, t4);
```

A. `A=1 B=2 C=3 D=4 E=5`

B. `A=1 B=3 C=5 D=4 E=2`

C. `A=1 B=3 C=5 D=2 E=4` `(answer)`

D. `A=1 B=2 C=4 D=3 E=5`

E. `A=5 B=4 C=3 D=2 E=1`

F. None of the above options will work

(16) [4 marks] (MCQ)

Consider the tree `bst` created as the result of evaluating the following Source program. Is this tree a binary search tree or not? If not, why not?

```
const t1 = make_tree(1, make_empty_tree(), make_empty_tree());
const t2 = make_tree(2, t1, make_empty_tree());
const t3 = make_tree(3, t2, make_empty_tree());
const t4 = make_tree(4, t3, make_empty_tree());
const bst = make_tree(5, t4, make_empty_tree());
```

A. Yes, it is a BST. **(answer)**

B. No, it is not a BST as the values in the left subtree are not smaller that the value at the root.

C. No, it is not a BST as the values in the left subtree are not larger that the value at the root.

D. No, it is not a BST as the values in the right subtree are not smaller that the value at the root.

E. No, it is not a BST as the values in the right subtree are not larger that the value at the root.

F. No, it is not a BST as the tree is not balanced.

Section G: Sorting [17 marks]

For all questions in this section, consider the following implementation of the ***insertion sort*** algorithm.

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

function insert_sort(xs, cmp) {
    return is_null(xs)
           ? xs
           : insert(head(xs), insert_sort(tail(xs), cmp), cmp);
}
```

(17) [4 marks]

Rewrite the `insert_sort` function using the `accumulate` function. Your function may call the given `insert` function.

Write your answer only in the dashed-line boxes.

```
function insert_sort(xs, cmp) {
    return accumulate(
               (x, ys) => insert(x, ys, cmp)   ,
               null                            ,
               xs
           );
}
```

(18) [4 marks] (MCQ)

The function `sort_descend` takes a list of numbers as argument and returns a new list that contains the input numbers sorted in descending order. The following are three different implementations of `sort_descend`:

(X)
```
function sort_descend(xs) {
    const cmp = (x, y) => x <= y;
    return reverse(insert_sort(xs, cmp));
}
```

(Y)
```
function sort_descend(xs) {
    const cmp = (x, y) => x <= y;
    return insert_sort(reverse(xs), cmp);
}
```

(Z)
```
function sort_descend(xs) {
    const cmp = (x, y) => x <= y;
    return reverse(insert_sort(reverse(xs), cmp));
}
```

Which of the above implementations is/are correct?

A. None of them is correct
B. Only (X) is correct
C. Only (Y) is correct
D. Only (Z) is correct
E. Only (X) and (Y) are correct
F. Only (X) and (Z) are correct **(answer)**
G. Only (Y) and (Z) are correct
H. All (X), (Y) and (Z) are correct

(19) [4 marks]

A ***pair-list*** is a list of pairs where each pair has a number in the head and a number in the tail. For example, `list(pair(3,3), pair(1,2), pair(5,3), pair(0,5))` is a pair-list.

Complete the implementation of function `sort_pairs_by_sum`, which takes as argument a pair-list `PL`, and returns a pair-list whose pairs are from `PL` and are sorted in ascending order by the sum of the head and tail of each pair. When two pairs have the same sum, the two pairs can be in any order relative to each other in the result. For example,

```
const plist = list( pair(3,3), pair(1,2), pair(5,3), pair(0,5) );
sort_pairs_by_sum(plist); // returns list([1,2], [0,5], [3,3], [5,3])
```

Write your answer only in the dashed-line boxes.

```
function sort_pairs_by_sum(PL) {
    return insert_sort(PL,
                       (x, y) => head(x) + tail(x)
                                 <= head(y) + tail(y)
                      );
}
```

(20) [5 marks] (MCQ)

A ***pair-list*** is a list of pairs where each pair has a number in the head and a number in the tail.

The function `sort_pairs` takes as argument a pair-list `PL`, and returns a pair-list whose pairs are from `PL` and are ordered in ascending order by their head values, and for pairs that have the same head value, they are ordered in descending order by their tail values. For example,

```
const L = list(pair(3,4), pair(6,2), pair(3,2), pair(4,2),
               pair(3,7), pair(6,5), pair(6,4));
sort_pairs(L); // returns list([3,7],[3,4],[3,2],[4,2],[6,5],[6,4],[6,2])
```

Which of the following is a correct implementation of `sort_pairs`?

A.	`function sort_pairs(PL) {`
	`  const S = insert_sort(PL, (x, y) => head(x) < head(y));`
	`  return insert_sort(S, (x, y) => tail(x) > tail(y));`
	`}`

B.	`function sort_pairs(PL) {`
	`  const S = insert_sort(PL, (x, y) => tail(x) > tail(y));`
	`  return insert_sort(S, (x, y) => head(x) < head(y));`
	`}`

C.	`function sort_pairs(PL) {`
	`  const S = insert_sort(PL, (x, y) => head(x) <= head(y));`
	`  return insert_sort(S, (x, y) => tail(x) >= tail(y));`
	`}`

D.	`function sort_pairs(PL) {`
	`  const S = insert_sort(PL, (x, y) => tail(x) >= tail(y));`
	`  return insert_sort(S, (x, y) => head(x) <= head(y));`
	`}` **(answer)**

E.	`function sort_pairs(PL) {`
	`  return insert_sort(PL, (x, y) => head(x) <= head(y)`
	`  && tail(x) >= tail(y));`
	`}`

F.	`function sort_pairs(PL) {`
	`  return insert_sort(PL, (x, y) => head(x) < head(y)`
	`  && tail(x) > tail(y));`
	`}`

Section H: The Benefits of Being Sorted [13 marks]

(21) [6 marks]

Complete the implementation of function `filter_by_pos`, which takes as arguments a list of numbers `xs`, a list of non-negative integers `pos_list`, and returns a list of all the elements of `xs` whose positions in `xs` are elements of `pos_list`. Note that the first element of a list is at position 0. Therefore, all elements of `pos_list` are in the range from 0 to `length(xs) - 1`, and they are all unique and arranged in increasing order in `pos_list`. **The running time of your function must have an order of growth of *O*(`length(xs)`).**

Examples:

```
const L = list(10, 11, 12, 13, 14, 15, 16, 17, 18, 19);
filter_by_pos(L, list(1, 4, 5, 8)); // returns list(11,14,15,18)
filter_by_pos(L, list()); // returns null
```

Write your answer only in the dashed-line boxes.

```
function filter_by_pos(xs, pos_list) {
    function select(ys, curr_pos, ps) {
        return is_null(ps)
            ? null
            : curr_pos < head(ps)
            ? select(tail(ys), curr_pos + 1, ps)

            : pair(head(ys),
                   select(tail(ys), curr_pos + 1, tail(ps)));
    }
    return select(xs, 0, pos_list);
}
```

Solution: https://share.sourceacademy.nus.edu.sg/xeo9c

(22) [7 marks]

We represent a ***set of numbers*** as an *ordered list of numbers*, where all the elements are **unique** and are arranged in **ascending order**.

The ***set difference*** of set *A* and set *B* (denoted *A* – *B* in mathematics), is the set of elements in *A* but not in *B*.

Complete the implementation of function `diff`, which takes as arguments two sets of numbers, `A` and `B`, and returns the set difference of `A` and `B`. For example,

```
diff(list(2,4,5,6,9,10), list(4,6,7,10,11)); // returns list(2,5,9)
```

The running time of your function must have an order of growth of *O*(*n*), where *n* is the total length of the two input lists.

Write your answer only in the dashed-line boxes.

```
function diff(A, B) {

    if (is_null(A)) {
        return A;
        // Alternative solution:  return null;
    } else if (is_null(B)) {
        return A;

    } else if (head(A) === head(B)) {
        return diff(tail(A), tail(B));

        // Alternative solution:  return diff(tail(A), B);
    } else if (head(A) < head(B)) {
        return pair(head(A), diff(tail(A), B));


    } else {
        return diff(A, tail(B));


    }
}
```

Solution: https://share.sourceacademy.nus.edu.sg/xwovi

——— END OF QUESTIONS ———

Appendix

The following **LISTS** functions are supported in Source §2:

- `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`. Iterative process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`.
- `build_list(f, n)`: Makes a list with `n` elements by applying the unary function `f` to the numbers 0 to `n - 1`. Iterative 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`.
- `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`. Iterative 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`. Iterative 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`. Iterative 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`. Iterative 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`. Iterative 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)))`. Iterative process; time: $O(n)$, space: $O(n)$, where n is the length of `xs`, assuming `op` takes constant time.

Some other functions supported in Source §2:

- `is_boolean(x)`: Returns `true` if `x` is a boolean value, and `false` otherwise.
- `is_number(x)`: Returns `true` if `x` is a number, and `false` otherwise.
- `is_string(x)`: Returns `true` if `x` is a string, and `false` otherwise.
- `display(v)`: Displays the value `v` in the REPL.
- `stringify(v)`: Returns a string that represents the value `v`. For example, `stringify(123)` returns the string `"123"`, and `stringify(false)` returns the string `"false"`.
- `math_floor(x)`: Returns the largest integer that is equal to or less than the number `x`.
- `math_sqrt(x)`: Returns the square root of the number `x`.

——— END OF QUESTION PAPER ———