NATIONAL UNIVERSITY OF SINGAPORE

CS1101S — PROGRAMMING METHODOLOGY

(Semester 1 AY2018/2019)

CURATED VERSION OF 16/11/2020 (LAST CORRECTED ON 8/12/2021)

Time Allowed: **2 Hours**

INSTRUCTIONS TO STUDENTS

1. This assessment paper contains **EIGHT (8)** questions and comprises **TWENTY-FOUR (24)** printed pages, including this page.
2. The full score of this paper is **100 marks**.
3. This is a **CLOSED BOOK** assessment, but you are allowed to use **ONE** A4 sheet of notes.
4. Answer **ALL** questions **within the space provided** in this booklet.
5. Where programs are required, write them in the **Source §4** 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.

(write with a pen)

Student No.:

This portion is for examiner's use only

Q#	1	2	3	4	5	6	7	8	Σ
MAX	12	9	9	9	18	15	12	16	100
SC									

Question 1: Box-and-Pointer Diagrams [12 marks]

For each of the following Source programs, show the box-and-pointer diagram for **`input` and `result`** at the end of the program evaluation. Clearly show where `input` and `result` are pointing to. For pairs that are *identical* (===), only one box must be shown. Pairs that are *not identical* must be drawn as separate boxes. To represent an empty list in the tail of a pair, use **a single slash through the tail part** of the box for the pair.

1A. [3 marks]

```
const input = list(list(2), 3, pair(4, 5));
const result = map(x => x, input);          // see Appendix for map
```

1B. [3 marks]

```
const input = list(1, 2, 3);
const result = append(input, input);    // see Appendix for append
```

1C. [3 marks]

```
function my_append(xs, ys) {
    if (is_null(xs)) {
        return ys;
    } else {
        set_tail(xs, my_append(tail(xs), ys));
        return xs;
    }
}
const input = list(1, 2, 3);
const result = my_append(input, input);
```

1D. [3 marks]

```
function remove(v, xs) {
    return is_null(xs)
        ? null
        : v === head(xs)
            ? tail(xs)
            : pair(head(xs), remove(v, tail(xs)));
}
const input = list(1, 2, 3, 4, 5);
const result = remove(3, input);
```

Question 2: List Replacement Procedure [9 marks]

Consider the following function `replace`.

```
function replace(a, b, xs) {
    return is_null(xs)
        ? null
        : a === head(xs)
          ? pair(b, replace(a, b, tail(xs)))
          : pair(head(xs), replace(a, b, tail(xs)));
}
```

2A. [2 marks]

What is the result of evaluating the following program?

```
replace(1, 0, list(3, 1, 5, 0, 4, 1, 2, 1));
```

Enter your answer here (no explanation needed):

```
list(    ,    ,    ,    ,    ,    ,    ,    )
```

2B. [2 marks]

Does the function `replace` give rise to an iterative or a recursive process?
Circle the correct answer (no explanation needed):

```
iterative        recursive
```

2C. [2 marks]

What is the order of growth in runtime for applying `replace` to a list `xs` of length n? Characterise the runtime using Θ-notation.

$\Theta(\quad)$

2D. [3 marks]

Use the function `accumulate` to write a function `replace_2`, which produces the same results as the function `replace` above.

```
function replace_2(a, b, xs) {
    return accumulate(
        (x, ys) => 

        null,
        xs);
}
```

Question 3: Data Abstraction [9 marks]

We consider the **stack** data structure. For the subsequent parts of this question, you **must** make use of the **stack abstraction**, consisting of the following functions:

- `make_stack()` — Returns an empty stack.
- `is_empty(stack)` — Tests whether the stack `stack` is empty.
- `push(stack, x)` — Adds `x` to the top of the stack `stack`.
- `pop(stack)` — Removes and returns the top element of the stack `stack` if `stack` is not empty.

Do not break this stack abstraction in your programs.

3A. [3 + 1 marks]

Define the function `insert_to_bottom(stack, new_elem)` that takes as arguments a stack `stack` and a value `new_elem`, and inserts `new_elem` to the bottom of the stack. Your function `insert_to_bottom` should return `undefined`. You earn 3 marks for a correct solution. One extra mark is given for **a correct solution that does not use any additional data structure (array, pair, stack, etc)**, apart from the arguments of `insert_to_bottom`.

Example:

```
const S = make_stack();  push(S, 3);  push(S, 4);
insert_to_bottom(S, 9);
pop(S);  // returns 4
pop(S);  // returns 3
pop(S);  // returns 9
```

```
function insert_to_bottom(stack, new_elem) {







}
```

3B. [3 + 2 marks]

Define the function `reverse_stack(stack)` that takes as argument a stack `stack` and reverses it (i.e. the top element becomes the bottom, and so on). Your function `reverse_stack` should return `undefined`. Your function may call the function defined in Part A. You earn 3 marks for a correct solution. Two extra marks are given for **a correct solution that does not use any additional data structure (array, pair, stack, etc)**, apart from the argument of `reverse_stack`.

Example:

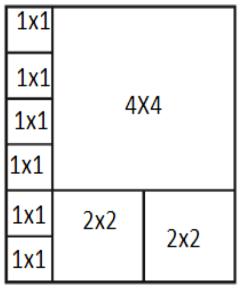
```
const S = make_stack();  push(S, 1);  push(S, 2);  push(S, 3);
reverse_stack(S);
pop(S);  // returns 1
pop(S);  // returns 2
pop(S);  // returns 3
```

```
function reverse_stack(stack) {



}
```

Question 4: Wishful Thinking [9 marks]



Given a rectangular floor area of size $L \times W$, where integer L is the length and integer W is the width, we want to find the minimum number of tiles of size $2^k \times 2^k$, where k can be any non-negative integer, that are needed to cover the floor exactly. For example, given the floor area of 5×6, as shown in the diagram on the right, the minimum number of $2^k \times 2^k$ tiles needed is **9** (6 tiles of 1×1, 2 tiles of 2×2, and 1 tile of 4×4).

Complete the following function `min_tiles(L, W)` that takes as arguments non-negative integers `L` and `W` as the length and width of the rectangular floor area, and returns the minimum number of $2^k \times 2^k$ tiles needed to cover the floor exactly. You can make use of the given function `closest_two_power`.

```
// Returns the largest 2^k (where k is an integer) that is less than
// or equal to x. x must be a positive integer.
function closest_two_power(x) {
    return math_pow(2, math_floor(math_log2(x)));
}

function min_tiles(L, W) {
    if (L === 0 || W === 0) {

    } else if (L === 1 && W === 1) {

    } else {

    }
}
```

Question 5: Sorting and Reordering [18 marks]

5A. [4 marks]

The following function, `bubblesort_array`, implements the Bubble Sort algorithm to sort an array of numbers into ascending order:

```
function bubblesort_array(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 (A[j] > A[j + 1]) {
                const temp = A[j];
                A[j] = A[j + 1];   // Line 7
                A[j + 1] = temp;
            } else { }
        }
    }
}
```

5A.1. [3 marks]

How many times will **Line 7** be reached during the evaluation of the following program?

```
const my_array = [3, 5, 2, 4, 1];
bubblesort_array(my_array);
```

Write your answer below (no explanation needed):

5A.2. [1 mark]

How many additional arrays, **not counting `your_array`,** will the following program create?

```
const your_array = [3, 1, 2, 4];
bubblesort_array(your_array);
```

Write your answer below (no explanation needed):

5B. [6 marks]

Complete the following function, `bubblesort_list` that takes as argument a **list** of numbers and uses the Bubble Sort algorithm to sort the list into ascending order. Your function **must not create any new pair or array**, and **must not use the function `set_tail`**.

Example:

```
const LL = list(3, 5, 2, 4, 1);
bubblesort_list(LL);
LL; // should show [1, [2, [3, [4, [5, null]]]]]
```

```
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) {



        }
    }
}
```

5C. [8 marks]

Given an array `A` of *N* distinct integers, and an array `T` of *N* distinct integers in the range [`0, N−1`], we want to reorder the elements in `A`, according to their target locations specified in `T`. More specifically, the element `A[i]` must be moved to location `T[i]` of array `A`, for each `i = 0, ..., N−1`.

5C.1. [3 marks]

Complete the following function, `reorder1(A, T)`, that takes as arguments the arrays `A` and `T` and modifies `A` such that every element `A[i]` is moved to location `T[i]` of array `A`.

Example:

```
const A = [78, 23, 56, 12, 99];
const T = [ 3,  1,  4,  0,  2];
reorder1(A, T);
A;  // should show [12, 23, 99, 78, 56]
```

```
function reorder1(A, T) {
    const N = array_length(A);
    const B = [];

    // copy B to A
    for (let i = 0; i < N; i = i + 1) {
        A[i] = B[i];
    }
}
```

5C.2. [5 marks]

Complete the following function, `reorder2(A, T)`, that produces the same output array `A` as `reorder1(A, T)`, but **must not use any additional array or list** other than the input arrays `A` and `T`. Your function can modify both arrays `A` and `T`. (Hint: use only swaps to perform the reordering.)

```
function swap(A, i, j) {
    const temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}

function reorder2(A, T) {
    const N = array_length(A);

}
```

Question 6: Tic-Tac-Toe [15 marks]

The game of *tic-tac-toe* (also called *noughts and crosses* or *Xs and Os*) is played on a 3×3 grid whose 9 slots are initially free. The game proceeds by players `X` and `O` taking turns, entering their names in the grid. The player wins by being the first to enter his/her name three times in one row, column or diagonal. The only rule of the game: At each turn, a player can enter his/her name only in a free slot.

6A. [2 marks]

We represent the grid by arrays. Each slot of the grid is initially *free*, represented by a single-character string, containing one *underscore* character `"_"`, as given here:

```
const grid_1 = [ ["_", "_", "_"],
                 ["_", "_", "_"],
                 ["_", "_", "_"] ];
```

After two rounds of play between players `X` and `O`, the grid might look like this:

```
const grid_2 = [ ["_", "X", "O"],
                 ["_", "O", "_"],
                 ["X", "_", "_"] ];
```

Eventually, one of the players may be able to enter his/her name three times in one row, column or diagonal, in which case he/she wins.

```
const grid_3 = [ ["X", "_", "O"],
                 ["X", "_", "X"],
                 ["X", "O", "O"] ];
```

How many arrays get created when the program *just above*, declaring the constant **`grid_3`**, gets evaluated? Write your answer below (no explanation needed):

6B. [2 marks]

In order to print grids, we are declaring a function `grid_to_string` that takes a grid as argument and returns a string representation of it. Fill the correct indices so that the resulting string represents the grid similar to the way grids are written in the programs above.

```
function grid_to_string(grid) {
    return "Current grid:\n" +
        grid[ ][ ] + grid[ ][ ] + grid[ ][ ] + "\n" +
        grid[ ][ ] + grid[ ][ ] + grid[ ][ ] + "\n" +
        grid[ ][ ] + grid[ ][ ] + grid[ ][ ] + "\n" ;
}
```

6C. [2 marks]

After every game, we would like to give the players the option to play again, by resetting the strings in the grid. Write a function `free_grid`, **using nested for-loops**, that resets all strings to `"_"`. You can assume that the given grid `grid` is a 3×3 arrangement as shown above. Your function should change `grid` and **not create any new arrays**.

```
function free_grid(grid) {

}
```

6D. [3 marks]

Write a function `replace_string` that

- enters the given string `new_string` in the slot at the given row `r` and column `c` of the given grid `g` and returns **`true`**, if that slot currently holds the given string `expected_string`, or
- leaves the grid unchanged and returns **`false`**, if there is a string other than `expected_string` in the slot.

You can assume that the grid is represented as in the examples above, that `r` and `c` are numbers 0, 1 or 2, and that `new_string` and `expected_string` are strings.

```
function replace_string(new_string, r, c, g, expected_string) {

}
```

6E. [6 marks]

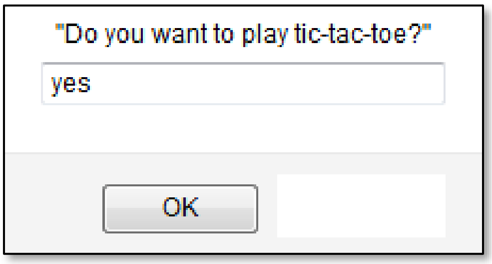
In order to program the game, you can use the following function `check_winner` that checks if a given player, represented by the string `p`, has won the game, according to the current state of the given grid `g`.

```
function check_winner(g, p) {
    return (
       // checking diagonals
       (g[0][0] === p && g[1][1] === p && g[2][2] === p) ||
       (g[0][2] === p && g[1][1] === p && g[2][0] === p) ||
       // checking columns
       (g[0][0] === p && g[1][0] === p && g[2][0] === p) ||
       (g[0][1] === p && g[1][1] === p && g[2][1] === p) ||
       (g[0][2] === p && g[1][2] === p && g[2][2] === p) ||
       // checking rows
       (g[0][0] === p && g[0][1] === p && g[0][2] === p) ||
       (g[1][0] === p && g[1][1] === p && g[1][2] === p) ||
       (g[2][0] === p && g[2][1] === p && g[2][2] === p) );
}
```

Assume a given built-in function `prompt` that takes a string as argument. It pops up a window that displays the string and has an input field, where the user can enter a new string. If the user presses an `OK` button, the function returns the entered string. For example, when evaluating the function application expression

```
prompt("Do you want to play tic-tac-toe?")
```

a window pops up as follows:



When the user presses `OK`, the function application returns the entered string, in this case `"yes"`.

You need to complete the game below and meet the following requirements:

- At every turn, a player chooses a row `r` and column `c` that you can assume to be the integer 0, 1 or 2.
- If the grid already has a player name in the given slot, you need to let the player try again until she hits a free slot, and display the grid each time.
- When a player has won the game, you need to **display the final grid** and **announce the correct player to be the winner**, using the `prompt` function. After that you need to ask again if the players “want to play tic-tac-toe”, using the given outer while-loop.
- Make sure that players `X` and `O` **take turns** playing the game.
- All user interaction must happen through the function `prompt`. Do not use `display` or any other output function.

Complete the program below to meet the requirements.

```
function play_tic_tac_toe() {
    const grid = [["_","_","_"],
                  ["_","_","_"],
                  ["_","_","_"]];
    while (prompt("Do you want to play tic-tac-toe?") === "yes") {
        free_grid(grid);
        let current_player = "X";  // X always starts first
        while (current_player !== "GAME_OVER") {
            const r = parse_int(prompt(grid_to_string(grid) +
                                  "\nPlayer " + current_player +
                                   ": enter row (0-2): "), 10);
            const c = parse_int(prompt(grid_to_string(grid) +
                                  "\nPlayer " + current_player +
                                  ": enter col (0-2): "), 10);

        }
    }
    prompt("Hope you had a nice time playing tic-tac-toe!");
}
```

Question 7: Environment Model [12 marks]

For each of the following Source programs, draw the diagram to show the environment during the evaluation of the program. Show all the frames that are created during the program evaluation, but do not draw empty frames. Show the final value of each binding.

7A. [5 marks]

```
function fun(n) {
    if (n <= 1) {
        return n;
    } else {
        let xx = fun(n - 1);
        let yy = fun(n - 2);
        return xx + yy;
    }
}
fun(3);
```

7B. [7 marks]

```
function dap(fun, xs) {
    if (! is_null(xs)) {
        const h = head(xs);
        set_head(xs, fun(h));
        dap(fun, tail(xs));
    } else { }
}
const LL = list(3);
dap(x => y => x + y, LL);
```

Question 8: Convoluted Extensions [16 marks]

A *binary number operation* is a function that takes two numbers as arguments and returns a number. For example the built-in function `math_pow` is a binary number function.

Similarly, a *binary stream operation* is a function that takes two streams as arguments and returns a stream. For example the pre-declared function `stream_append` (see Appendix) is a binary stream operation.

This question develops abstractions that deal with **infinite streams of numbers**. This means you can assume that no stream is ever the empty list and that the `head` of any stream is a number.

To visualize the beginning of the given stream `s`, we use the following function `show_stream`, which displays its first `n` elements as a string.

```
function show_stream(s, n) {
    for_each(display, eval_stream(s, n));
}
```

We shall use the streams `ones` and `integers` as examples below.

```
const ones = pair(1, () => ones);
const integers = pair(1, () => stream_map(x => x + 1, integers));
show_stream(ones, 6);      // displays "1 1 1 1 1 1 "
show_stream(integers, 6);  // displays "1 2 3 4 5 6 "
```

8A. [2 marks]

Does the function `show_stream` give rise to an iterative or a recursive process when applied to an infinite stream and a positive integer? Circle the correct answer (no explanation needed):

```
iterative          recursive
```

8B. [2 marks]

Consider the following binary stream operation `stream_zip`:

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

What sequence of numbers is displayed as the result of evaluating the following program?

```
show_stream(stream_zip(ones, integers), 9);  // note the nine!
```

8C. [5 marks]

Write a function `extend` that takes a binary number operation `bno` as argument and returns a binary stream operation `bso`. When `bso` is applied to two argument streams, each element of the resulting stream in position `p` is the result of applying `bno` to elements of the argument streams in position `p`.

We illustrate this concept with two examples.

Example 1:

```
const add_streams = extend((x, y) => x + y);
show_stream(add_streams(ones, ones), 6);
    // displays "2 2 2 2 2 2 "
```

Example 2:

```
const mult_streams = extend((x, y) => x * y);
show_stream(mult_streams(integers, integers), 6);
    // displays "1 4 9 16 25 36 "
```

```
function extend(bno) {

}
```

8D. [7 marks]

In this part, we use the following function `convolve` that takes a binary stream operation `bso` and returns a unary stream operation, which applies `bso` to its argument, using a position offset of one for the second argument.

```
function convolve(binary_stream_operation) {
    function unary_stream_operation(stream) {
        const convolved = pair(head(stream),
                               () => binary_stream_operation(
                                         stream, convolved));
        return convolved;
    }
    return unary_stream_operation;
}
```

8D.1. [2 marks]

What sequence of numbers is displayed as the result of evaluating the following program, where `stream_zip` is as in Part B and `integers` is given in the beginning of Question 8?

```
const convolving_zip = convolve(stream_zip);
show_stream(convolving_zip(integers), 9);     // note the nine!
```

8D.2. [2 marks]

What sequence of numbers is displayed as the result of evaluating the following program, where the function `extend` is as described in Part C?

```
show_stream(convolve(extend((x, y) => x * y))(integers), 6);
```

8D.3. [3 marks]

Assume that the function `repeat` is defined as follows:

```
function repeat(f, n) {
    return x => n === 0 ? x : f(repeat(f, n - 1)(x));
}
```

What sequence of numbers is displayed as the result of evaluating the following program, where the function `add_streams` is as described in Part C?

```
const convolving_add = convolve(add_streams);
show_stream(repeat(convolving_add, 0)(ones), 6);  // (a)
show_stream(repeat(convolving_add, 1)(ones), 6);  // (b)
show_stream(repeat(convolving_add, 2)(ones), 6);  // (c)
show_stream(repeat(convolving_add, 3)(ones), 6);  // (d)
show_stream(repeat(convolving_add, 4)(ones), 6);  // (e)
show_stream(repeat(convolving_add, 5)(ones), 6);  // (f)
```

(a):

(b):

(c):

(d):

(e):

(f):

END OF QUESTIONS

Appendix

Built-in Functions

The following are some of the built-in functions in Source §4:

- `display(a)` — Displays the given value `a` on the screen.
- `parse_int(s, b)` — Returns the number represented by the string `s` using base `b`.
- `prompt(s)` — Pops up a window that displays `s` and returns a string entered by the user.
- `math_pow(b, e)` — Returns base number `b` to the power of exponent number `e`.
- `math_floor(n)` — Returns the largest integer that is smaller than or equal to number `n`.
- `math_log2(n)` — Returns the logarithm to the base of 2 of number `n`.
- `pair(x, y)`
- `is_pair(x)`
- `head(x)`
- `tail(x)`
- `is_null(xs)`
- `list(x1, x2,..., xn)`
- `set_head(p, x)`
- `set_tail(p, x)`
- `array_length(x)`
- `stream_tail(x)`
- `is_stream(x)`
- `stream(x1, x2,..., xn)`

Pre-declared Functions

Some of the pre-declared functions in Source §4 are declared as follows:

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

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

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

function for_each(f, xs) {
    if (is_null(xs)) {
        return true;
    } else {
        f(head(xs));
        return for_each(f, tail(xs));
```

```
    }
}

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

```
function stream_append(xs, ys) {
    return is_null(xs)
        ? ys
        : pair(head(xs), () => stream_append(stream_tail(xs), ys));
}

function eval_stream(s, n) {
    return n === 0
        ? null
        : pair(head(s), eval_stream(stream_tail(s), n - 1));
}

function stream_map(f, s) {
    return is_null(s)
        ? null
        : pair(f(head(s)), () => stream_map(f, stream_tail(s)));
}
```

(Blank page. Do not tear off.)

END OF PAPER