NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

MIDTERM QUIZ

ADAPTED TO SOURCE 2021 IN 9/2020

Semester 1 AY2015/2016

CS1101S — PROGRAMMING METHODOLOGY

30 September 2015 Time Allowed: **1 Hour 35 Minutes**

SOLUTION

Matriculation No.: | | | | | | | | | |

Instructions (please read carefully):

1. Write down your **matriculation number** on the **question paper**. DO NOT WRITE YOUR NAME ON THE QUESTION SET!
2. Write down your Avenger's name in the box provided above.
3. This is an **open-sheet quiz**. You are allowed to bring one A4 sheet of notes (written or printed on both sides).
4. This paper comprises **5 questions** and **EIGHTEEN (18)** printed pages.
5. The maximum score of this quiz is **60 marks**. The weight of each question is given in square brackets beside the question number.
6. All questions must be answered correctly for the maximum score to be attained.
7. All questions must be answered in the space provided in the question paper; no extra sheets will be accepted as answers.
8. The pages marked "scratch paper" in the question paper may be used as scratch paper.
9. You are allowed to use pencils or pens, as you like (no red color, please).
10. Write legibly; **UNTIDINESS will be penalized**.

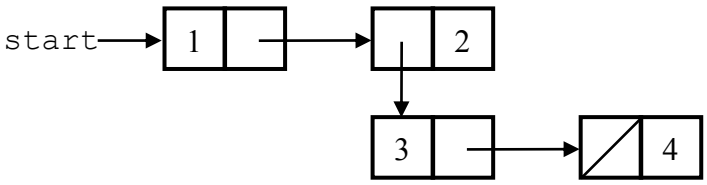
GOOD LUCK!

Q#	1	2	3	4	5	Σ
MAX	5	15	13	12	15	60
SC						

Question 1: Boxes and Pointers [5 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 name `start` must refer to the pair as shown in the diagram. If the structure cannot be constructed using Source §2, you must answer "not possible."

A. [1 mark]

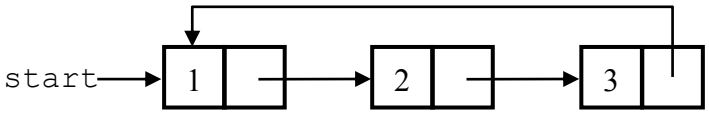



```
const start = pair(1, pair(pair(3, pair(null, 4)), 2));
```

B. [2 marks]

[not relevant in 2020, due to changes in the material]

C. [2 marks]



Not possible. (Because Source §2 does not allow the contents in a pair to be modified after its creation.)

Question 2: Lists and Trees [15 marks]

A. [3 marks] Sharing Pairs

[not relevant in 2020, due to changes in the material]

B. [4 marks] Lists of Numbers

A list of numbers is null or a pair whose head is a number and whose tail is a list of numbers. For example,

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

Write a function `is_list_of_numbers` such that `is_list_of_numbers(x)` returns `true` if `x` is a list of numbers and `false` in all other cases. Your function is not supposed to give rise to an error, regardless of the argument value.

You get the maximum 4 marks only if you *do not* use any of the functions `filter`, `map` and `accumulate` in your correct solution, otherwise you get at most 2 marks.

```
function is_list_of_numbers(x) {

// Solution 1 [4 marks]
    return is_null(x)
        ? true
        : is_pair(x)
          ? is_number(head(x)) &&
            is_list_of_numbers(tail(x))
          : false;

// Solution 2 [2 marks]
    return is_list(x) &&
           accumulate( (a, b) => is_number(a) && b,
                       true, x);









}
```

C. [5 marks] Trees of Numbers

A tree of numbers is a list whose elements are numbers or trees of numbers. For example,

```
const tree_of_num = list(list(1, list(2,3)), 4, 5, list(6, 7));
```

Write a function `is_tree_of_numbers` such that `is_tree_of_numbers(x)` returns `true` if `x` is a tree of numbers and `false` in all other cases. Your function is not supposed to give rise to an error, regardless of the argument value.

You get 5 marks if you use at least one of the functions `filter`, `map` and `accumulate` in a correct and meaningful way, and 3 marks for any other correct solution.

```
function is_tree_of_numbers(x) {

// Solution 1 [5 marks]
    return is_list(x) &&
           accumulate( (a, b) => (is_number(a) ||
                                  is_tree_of_numbers(a)) && b,
                       true, x);

// Solution 2 [3 marks]
    return is_null(x)
        ? true
        : is_pair(x)
          ? (is_number(head(x)) ||
             is_tree_of_numbers(head(x)))
            &&
            is_tree_of_numbers(tail(x))
          : false;









}
```

D. [3 marks] `my_filter`

Write a Source §2 function `my_filter` that behaves like the built-in function `filter`, but you must implement it using the `accumulate` function.

```
function my_filter(pred, xs) {

    return accumulate( (a, b) => pred(a) ? pair(a, b) : b,
                       null, xs);









}
```

Question 3: Processing Digital Images [13 marks]

A grayscale 2D digital image is a rectangular table of pixel brightness values. Here, we represent a digital image as a list of lists of numbers. For example, the digital image

1	2	3	4
5	6	7	8
9	10	11	12

is represented as

```
list(list(1, 2, 3, 4), list(5, 6, 7, 8), list(9, 10, 11, 12));
```

The top-most row of the image is row 0, and the row number increases from top to bottom. The left-most column of the image is column 0, and the column number increases from left to right.

A. [6 marks] `make_image`

Write a function `make_image` that takes two *positive* integers `rows` and `columns` as the first and second arguments, respectively, and a function `func` as the third argument, such that `make_image` returns an image with `rows` rows and `columns` columns, and each pixel at row `r` and column `c` has the value `func(r, c)`.

```
function make_image(rows, columns, func) {

// Solution 1
    return build_list(rows,
                r => build_list(columns,
                                c => func(r, c)));


// Solution 2
    const row_nums = build_list(rows, x => x);
    const col_nums = build_list(columns, x => x);

    return map(r => map(c => func(r, c),
                        col_nums),
                row_nums);









}
```

B. [2 marks] `flip_image_vertically`

Write a function `flip_image_vertically` that takes an image as the only argument, and returns an image that is the vertical flip of the input image. For example, the vertical flip of the digital image

1	2	3	4
5	6	7	8
9	10	11	12

is the digital image

9	10	11	12
5	6	7	8
1	2	3	4

```
function flip_image_vertically(image) {

    return reverse(image);









}
```

C. [2 marks] `flip_image_horizontally`

Write a function `flip_image_horizontally` that takes an image as the only argument, and returns an image that is the horizontal flip of the input image. For example, the horizontal flip of the digital image

1	2	3	4
5	6	7	8
9	10	11	12

is the digital image

4	3	2	1
8	7	6	5
12	11	10	9

```
function flip_image_horizontally(image) {

    return map(reverse, image);









}
```

D. [3 marks] `rotate_image_180`

Write a function `rotate_image_180` that takes an image as the only argument, and returns an image that is equal to the input image rotated 180 degrees.

```
function rotate_image_180(image) {

    return flip_image_horizontally(
               flip_image_vertically(image));









}
```

Question 4: Exhausting Time and Space [12 marks]

According to the substitution model of execution, a process can be said to *exhaust all time resources* if it keeps evaluating sub-expressions and never reaches any result value.

A process can be said to *exhaust all space resources* if it keeps growing while it evaluates sub-expressions, i.e. for any natural number n, it will reach the size n (in the number of sub-expressions) during evaluation.

Example:

```
function loop(x) {return loop(x);}
loop(0);
```

This program exhausts all time resources, but not all space resources.

Example:

```
function loop2(x) {return loop2(loop2(x));}
loop2(0);
```

This program exhausts all time resources and all space resources.

Consider the following programs and indicate whether they exhaust all time resources and whether they exhaust all space resources. **You are awarded 1 mark for each correct answer, 0 mark for no answer, and −½ mark for an incorrect answer.** The minimum total marks for the entire Question 4 will be kept at 0.

A. [2 marks]

```
(x => x)(x => x);
```

Exhausts all time resources? (Yes/No): No

Exhausts all space resources? (Yes/No): No

B. [2 marks]

```
(x => x(x))(x => x);
```

Exhausts all time resources? (Yes/No): No

Exhausts all space resources? (Yes/No): No

C. [2 marks]

```
(x => x(x))(x => x(x));
```

Exhausts all time resources? (Yes/No): Yes

Exhausts all space resources? (Yes/No): No

D. [2 marks]

```
(x => x(x)(x))(x => x(x));
```

Exhausts all time resources? (Yes/No): Yes

Exhausts all space resources? (Yes/No): No

E. [2 marks]

```
(x => x(x(x)))(x => x(x));
```

Exhausts all time resources? (Yes/No): Yes

Exhausts all space resources? (Yes/No): No

F. [2 marks]

```
(x => x(x))(x => x(x(x)));
```

Exhausts all time resources? (Yes/No): Yes

Exhausts all space resources? (Yes/No): Yes

Question 5: Combinations and Permutations [15 marks]

A. [8 marks] Checking Permutations

[6 marks] Write a function `are_permutation` that takes two lists of numbers `xs1` and `xs2` as the first and second arguments, respectively, and returns `true` if `xs1` and `xs2` are permutations of each other. Note that `xs1` (and `xs2`) is allowed to have duplicate numbers.

For example,

- `are_permutation(list(1, 2, 2, 5, 4), list(4, 2, 5, 1, 2))` returns `true`;
- `are_permutation(list(1, 2, 2, 5, 4), list(1, 5, 5, 2, 4))` returns `false`.

```
function are_permutation(xs1, xs2) {

   return is_null(xs1) && is_null(xs2)
       ? true
       : !is_null(xs1) && !is_null(xs2)
         ? !is_null(member(head(xs1), xs2)) &&
           are_permutation(tail(xs1), remove(head(xs1), xs2))
         : false;

}
```

[2 marks] Let the length of `xs1` and `xs2` be *m* and *n*, respectively. What would be the order of growth of runtime of `are_permutation`, in Θ notation, with respect to *m* and *n*? (No need for explanation.)

Θ(*mn*).

B. [7 marks] *k*-Combinations

A *k-combination* of a set *S* is a subset of *k* elements of *S*.

Write a function `combinations` that takes a list of *n distinct* numbers `xs` as the first argument and an integer `k` as the second argument, and returns a list containing all the *k*-combinations of the numbers in `xs`. Each *k*-combination is represented as a list of *k* numbers. We can assume $0 \leq k \leq n$.

For example, `combinations(list(1, 2, 3, 4), 2)` returns `list(list(1, 2), list(1, 3), list(1, 4), list(2, 3), list(2, 4), list(3, 4))`.

Note that the numbers within each *k*-combination can be in any order, and the *k*-combinations within the returned list can be in any order too.

```
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);
    }









}
```

——— END OF QUESTIONS ———

Appendix

List Support

Source §2 supports the following list processing functions:

- `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.
- `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 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.

Miscellaneous Functions

- `is_number(x)`: Returns `true` if `x` is a number, and `false` otherwise.

(Scratch Paper)

(Scratch Paper)

(Scratch Paper)

(Scratch Paper)