NATIONAL UNIVERSITY OF SINGAPORE

CS1101S — PROGRAMMING METHODOLOGY

(AY2019/2020 SEMESTER 1)

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

Time Allowed: **1 Hour 45 Minutes**

SOLUTIONS

INSTRUCTIONS

1. This assessment paper contains **SIX (6)** 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

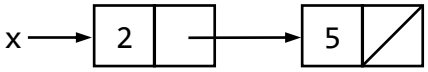
Q#	1	2	3	4	5	6	Σ
MAX	10	19	8	13	11	14	75
SC							

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.

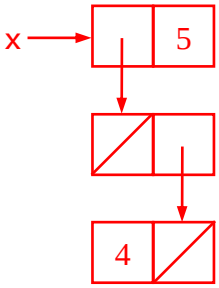
For example, the following program results in the following diagram on the right:

```
const x = pair(2, pair(5, null));
```



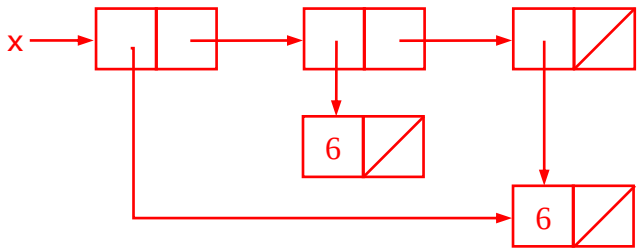
1A. [2 marks]

```
const x = pair(pair(null, pair(4, null)), 5);
```

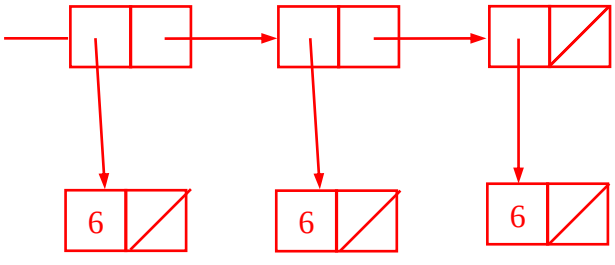


1B. [2 marks]

```
const q = list(6);
const x = list(q, pair(6, null), q);
```

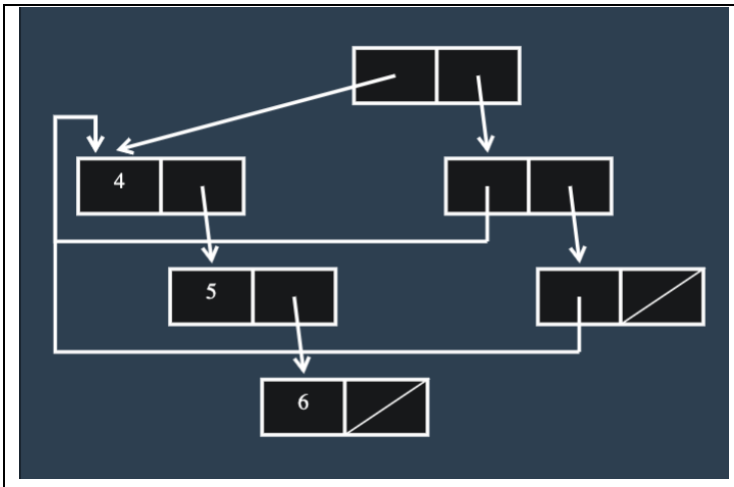


OR

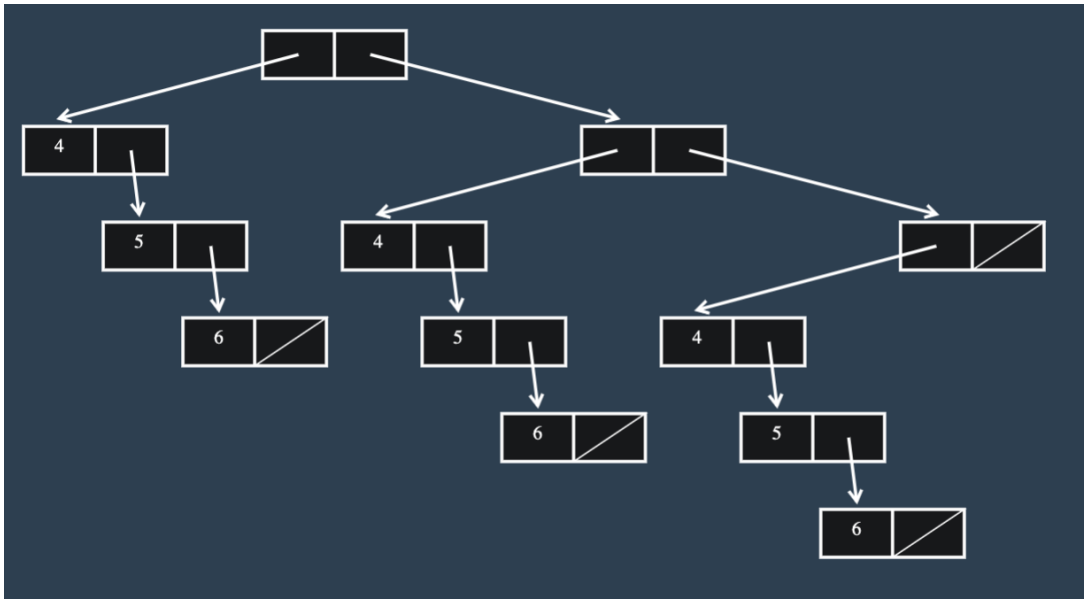


1C. [3 marks]

```
const ys = list(4, 5, 6);
const x = map(x => ys, ys);
```

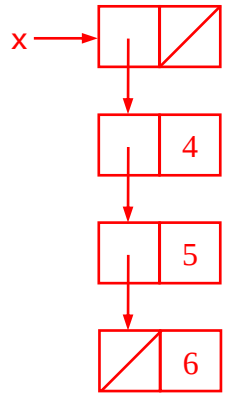


OR



1D. [3 marks]

```
const x = accumulate((x, ys) => map(y => pair(y, x), ys),
                     list(null), list(4, 5, 6));
```



Question 2: List Processing [19 marks]

2A. [4 marks]

THIS QUESTION IS OBSOLETE IN AY2020/21.

2B. [5 marks]

THIS QUESTION IS OBSOLETE IN AY2020/21.

2C. [5 marks]

Complete the `last_comes_first` function, which takes as argument a *non-empty list* `xs`, and returns a list that results from removing the last element from `xs` and adding it as the first element of the result list.

Examples:

```
last_comes_first(list(2));          // returns list(2)
last_comes_first(list(2,5,3,4,5));  // returns list(5,2,5,3,4)
```

```
function last_comes_first(xs) {

    if (is_null(tail(xs))) {
        return xs;
    } else {
        const p = last_comes_first(tail(xs));
        return pair(head(p), pair(head(xs), tail(p)));
    }









}
```

2D. [5 marks]

The `fib_list` function takes as argument an integer `N` ≥ 2 and returns a list containing the first `N` Fibonacci numbers, arranged in *ascending order*.

Examples:

```
fib_list(2);  // returns list(0, 1)
fib_list(7);  // returns list(0, 1, 1, 2, 3, 5, 8)
```

Complete the following implementation of `fib_list`, which must give rise to an **iterative process** and its runtime should have an order of growth of $\Theta(N)$.

```
function fib_list(N) {

    function helper(count, rev) {
        return count === N
            ? rev
            : helper(count + 1,
                     pair(head(rev) + head(tail(rev)), rev));
    }

    return reverse(helper(2, list(1, 0)));









}
```

Question 3: Orders of Growth [8 marks]

3A. [3 marks]

Assume a resource function $r(n)$. Indicate true or false for each of the following:

(i) [1 mark] $r(n)$ has order of growth $\Theta(r(n))$.

Circle one: (True) / False

(ii) [1 mark] $r(n)$ has order of growth $O(r(n))$.

Circle one: (True) / False

(iii) [1 mark] $r(n)$ has order of growth $\Omega(r(n))$.

Circle one: (True) / False

3B. [2 marks]

Assume a resource function $r(n)$ and another function $g(n)$ such that $r(n)$ has order of growth $\Theta(g(n))$. Indicate true or false for each of the following:

(i) [1 mark] $r(n)$ has order of growth $O(g(n))$.

Circle one: (True) / False

(ii) [1 mark] $r(n)$ has order of growth $\Omega(g(n))$.

Circle one: (True) / False

3C. [3 marks]

Assume a resource function $r(n)$ with order of growth $\Theta(n \log n)$. An example would be the runtime for merge sort. Indicate true or false for each of the following:

(i) [1 mark] $r(n)$ has order of growth $\Theta(n^2)$.

Circle one: True / (False)

(ii) [1 mark] $r(n)$ has order of growth $\Omega(n^2)$.

Circle one: True / (False)

(iii) [1 mark] $r(n)$ has order of growth $O(n^2)$.

Circle one: (True) / False

Question 4: Active Lists [13 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 active list, and returns that element in a list of length 1. Note that the first element in an active list is at position 0.

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

Example:

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

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

4A. [3 marks]

Write the function `act_length` that takes as argument an active list `as`, and returns the length of the active list.

Example:

```
const as = make_active_list(list());
const bs = make_active_list(list(8, 3, 5));
act_length(as);  // returns 0
act_length(bs);  // returns 3
```

```
function act_length(as) {

    function iter(k) {
        return is_null(as(k)) ? k : iter(k + 1);
    }
    return iter(0);









}
```

4B. [5 marks]

Write the function `act_append` that takes as arguments two active lists, `as` and `bs`, and returns an active list that results from appending `bs` to `as`.

Example:

```
const as = make_active_list(list(11, 22));
const bs = make_active_list(list(33, 44, 55));
const cs = act_append(as, bs);
act_length(cs);  // returns 5
list(cs(0), cs(1), cs(2), cs(3), cs(4));
    // returns list(list(11), list(22), list(33), list(44), list(55))
```

Your implementation may make use of the `act_length` function from the preceding task.

```
function act_append(as, bs) {

    const len_as = act_length(as);

    return pos => (pos < len_as)
                  ? as(pos)
                  : bs(pos - len_as);









}
```

4C. [5 marks]

Write the function `sum` that takes as arguments an active list `as` and a function `f`, and returns the sum of `f(x)` for every element `x` of the input active list. We assume that all elements of the input active list are numbers.

Example:

```
const as = make_active_list(list(1, 2, 3));
sum(as, x => x * x);  // returns 14  (1*1 + 2*2 + 3*3)
```

Your implementation may use the `act_length` function, and must make use of **at least one of the three functions**: `accumulate`, `map`, `filter`, in a meaningful way, to produce the result.

```
function sum(as, f) {

    return accumulate((x, ys) => f(x) + ys,
                      0,
                      build_list(act_length(as), i => head(as(i))));









}
```

Question 5: Binary Arithmetic Expressions [11 marks]

A ***Binary Arithmetic Expression (BAE)*** is either a *number* or the expression (⟨*bae*⟩ ⟨*op*⟩ ⟨*bae*⟩), where each ⟨*bae*⟩ is a BAE and ⟨*op*⟩ is the binary operator + or *. The followings are examples of BAEs:

- 123
- (56 + 23)
- ((2 + 5) * 100)

BAEs are arithmetic expressions that we are all familiar with, except that in BAEs, a pair of parentheses is always used to surround every binary arithmetic operation. As a result, we do not need to be concerned with operator precedence and associativity.

We represent BAEs in Source in the following way: a BAE is either a *number* or a list that has 3 elements where the first element is a BAE, the second element is a string `"+"` or `"*"`, and the third element is a BAE. The first and third elements are the left and right operands of the binary arithmetic operation, respectively. For example, the BAE ((2 + 5) * 100) has the following representation in Source: `list( list(2, "+", 5), "*", 100 )`.

5A. [5 marks]

Write a function `eval_BAE` that takes as argument a BAE `bae`, and evaluates it to a single numeric value.

Example:

```
const bae1 = 123;
eval_BAE(bae1);  // returns 123
const bae2 = list( list(2, "+", 5), "*", 100 );
eval_BAE(bae2);  // returns 700
```

```
function eval_BAE(bae) {

    if (is_number(bae)) {
        return bae;
    } else {
        const left = eval_BAE(head(bae));
        const right = eval_BAE(head(tail(tail(bae))));
        const op = head(tail(bae));
        return (op === "+") ? left + right : left * right;
    }









}
```

5B. [6 marks]

Write a function `negate_BAE` that takes as argument a BAE `bae`, and returns a BAE whose value is the negation of `bae`. The result BAE must have the same number of `"+"` and `"*"` as the original.

Example:

```
const bae1 = 123;
negate_BAE(bae1);            // returns -123
eval_BAE(negate_BAE(bae1));  // returns -123

const bae2 = list( list(2, "+", 5), "*", 100 );

negate_BAE(bae2);  // returns list( list(-2, "+", -5), "*",  100 )
                   //      or list( list( 2, "+",  5), "*", -100 )

eval_BAE(negate_BAE(bae2));  // returns -700
```

```
function negate_BAE(bae) {

    if (is_number(bae)) {
        return -bae;
    } else {
        const op = head(tail(bae));
        const left = head(bae);
        const right = head(tail(tail(bae)));
        return (op === "*")
            ? list(negate_BAE(left), op, right)
            : list(negate_BAE(left), op, negate_BAE(right));
    }









}
```

Question 6: Functions [14 marks]

6A. [6 marks]

Consider the following two functions:

```
const twice = f => (x => f(f(x)));
const thrice = f => (x => f(f(f(x))));
```

What is the result of each of the following statements?

(i) [2 marks] `(twice(x => 2 * x))(1);`

2^2 = 4

(ii) [2 marks] `(thrice(twice(x => 2 * x)))(1);`

(2^2)^3 = 64

(iii) [2 marks] `((thrice(twice))(x => 2 * x))(1);`

2^(2^3) = 256

6B. [4 marks]

What is the result of evaluating the following program?

```
function mystery(f, x) {
    return x === 0
        ? f(x)
        : mystery(x => f(x + 1), x - 1);
}
mystery(x => 7 * x, 8);
```

56

6C. [4 marks]

THIS QUESTION IS OBSOLETE IN AY2020/21.

——— END OF QUESTIONS ———

Appendix

The following **list processing** 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`. 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.

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.

(Scratch Paper. Do not tear off.)

(Scratch Paper. Do not tear off.)