NATIONAL UNIVERSITY OF SINGAPORE

CS1101S — PROGRAMMING METHODOLOGY

(AY2021/2022 SEMESTER 1)

MIDTERM ASSESSMENT

Time Allowed: **1 Hour 30 Minutes**

SOLUTIONS

INSTRUCTIONS

1. This assessment contains **19 Questions** in **7 Sections**.
2. The full score of this assessment is **70 marks**.
3. Answer **all questions**.
4. This is a **Closed-Book** assessment, but you are allowed one double-sided **A4 / foolscap / letter-sized sheet** of handwritten or printed **notes**.
5. You are allowed to use up to **4 sheets** of **blank A4 / foolscap / letter-sized** paper as **scratch paper**.
6. Where programs are required, write them in the **Source §2** language. You are allowed access to these online reference pages:
   - **Source §2 pre-declared constants and functions** at https://docs.sourceacademy.org/source_2/global.html
   - **Specification of Source §2** at https://docs.sourceacademy.org/source_2.pdf
7. In any question, unless it is specifically allowed, your answer **must not use functions given in, or written by you for, other questions**.
8. **Follow the instructions of your invigilator or the module coordinator to submit your answers**.

Section A: List Processing [14 marks]

(1) [4 marks]

Complete the function `repeat` that takes as arguments a value `v`, and a non-negative integer `k`, and returns a list such that `v` occurs exactly `k` times in the result list.

```
function repeat(v, k) {
    return /* YOUR SOLUTION */
}
```

Examples:

```
repeat(10, 5);     // returns list(10, 10, 10, 10, 10)
repeat(10, 0);     // returns null
repeat("abc", 3);  // returns list("abc", "abc", "abc")
```

In the following space, write your solution only for the part that is marked `/* YOUR SOLUTION */`.

```
// Solution 1:
build_list(x => v, k);

// Solution 2:
map(x => v, enum_list(1, k));

// Solution 3:
k === 0 ? null : pair(v, repeat(v, k - 1));
```

(2) [5 marks]

Complete the function `expand_list` that takes as arguments a list `L`, and a non-negative integer `k`, and returns a list such that each element of `L` occurs exactly `k` times consecutively in the result list. Your solution **must make use** of the `repeat` function from the preceding question.

```
function expand_list(L, k) {
    return accumulate( /* YOUR SOLUTION */ );
}
```

Examples:

```
expand_list(list(7, 8), 3); // returns list(7,7,7, 8,8,8)
expand_list(list(7, 8), 0); // returns null
expand_list(null, 3);       // returns null
```

In the following space, write your solution only for the part that is marked `/* YOUR SOLUTION */`.

```
(x, y) => append(repeat(x, k), y), null, L
```

(3) [5 marks]

An *R* x *C* matrix is represented in Source as a list of *R* lists of numbers and each list of numbers is of length *C*. Note that *R* and *C* are positive integers. For example, the following 2 x 3 matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

is represented in Source as `list(list(1, 2, 3), list(4, 5, 6))`.

Complete the function `expand_matrix` that takes as arguments an `R x C` matrix `M` and a positive integer `k`, and returns a `(k*R) x (k*C)` result matrix, where every element of `M` is replicated into a `k x k` submatrix in the result matrix. For example,

```
expand_matrix(list( list(1, 2, 3),
                    list(4, 5, 6) ), 3);
// returns
// list( list(1, 1, 1,  2, 2, 2,  3, 3, 3),
//       list(1, 1, 1,  2, 2, 2,  3, 3, 3),
//       list(1, 1, 1,  2, 2, 2,  3, 3, 3),
//       list(4, 4, 4,  5, 5, 5,  6, 6, 6),
//       list(4, 4, 4,  5, 5, 5,  6, 6, 6),
//       list(4, 4, 4,  5, 5, 5,  6, 6, 6) )
```

Your solution **must make use** of the `expand_list` function and/or `repeat` function from the preceding questions.

```
function expand_matrix(M, k) {
    return /* YOUR SOLUTION */
}
```

In the following space, write your solution only for the part that is marked ***/* YOUR SOLUTION */***.

```
expand_list( map(r => expand_list(r, k), M), k );
```

Section B: Orders of Growth [10 marks]

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

(4) [2 marks]

```
function fun(N) {
    return N < 1  ?  0  :  (N / 2) + fun(N - 1000);
}
```

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. $\Theta(2^N)$

(5) [2 marks]

```
function fun(N) {
    return N < 1  ?  0  :  (N * N) + fun(N - 1) + fun(N – 1);
}
```

A. $\Theta(1)$
B. $\Theta(\log N)$
C. $\Theta(N)$
D. $\Theta(N \log N)$
E. $\Theta(N^2)$
F. $\Theta(N^2 \log N)$
G. $\Theta(N^3)$
H. $\Theta(2^N)$ **(answer)**

(6) [2 marks]

```
function fun(N) {
    return N <= 1  ?  1  :  N * fun(N / 4);
}
```

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. $\Theta(2^N)$

(7) [2 marks]

```
function fun(N) {
    return N < 1  ?  null  :  pair(enum_list(1, N), fun(N - 1));
}
```

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. $\Theta(2^N)$

(8) [2 marks]

```
function fun(N) {
    return N < 1  ?  null  :  append(enum_list(1, N), fun(N - 1));
}
```

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 C: Processes [9 marks]

Given a list of numbers that is **already sorted in ascending order**, the function `unique` returns a list with all the duplicates removed and the remaining numbers are sorted in ascending order. For example,

```
unique( list(1, 1, 1, 2, 3, 3, 4, 4, 5, 6, 6, 6) );
// returns list(1, 2, 3, 4, 5, 6)
```

Complete the following two different implementations of `unique`. **The runtime of all implementations must have an order of growth of $\Theta(n)$, where n is the length of the input list.**

(9) [5 marks]

Complete the following implementation of `unique`, which must give rise to a **recursive process**. You **must not use** any of the pre-declared `map`, `filter`, and `accumulate` functions in your answer.

```
function unique(xs) {
    if (is_null(xs) || is_null(tail(xs))) {
        return xs;
    } else {
        /* YOUR SOLUTION */
    }
}
```

In the following space, write your solution only for the part that is marked `/* YOUR SOLUTION */`.

```
// Solution 1:
const wish = unique(tail(xs));
return head(xs) !== head(wish) ? pair(head(xs), wish) : wish;

// Solution 2:
return head(xs) !== head(tail(xs))
       ? pair(head(xs), unique(tail(xs)))
       : unique(tail(xs));
```

(10) [4 marks]

Complete the following implementation of `unique`, which gives rise to an **iterative process**.

```
function unique(xs) {
    function iter(ys, result) {
        if (is_null(ys)) {
            return result;
        } else {
            return iter(tail(ys), /* YOUR SOLUTION */ );
        }
    }
    return iter(reverse(xs), null);
}
```

Choose the correct expression from the following for the part that is marked ***/* YOUR SOLUTION */***.

A.
```
is_null(result) || head(ys) !== head(result)
? pair(head(ys), result)
: result
```
(answer)

B.
```
head(ys) !== head(result) || is_null(result)
? pair(head(ys), result)
: result
```

C.
```
is_null(result) || head(ys) !== head(result)
? result
: pair(head(ys), result)
```

D.
```
head(ys) !== head(result) || is_null(result)
? result
: pair(head(ys), result)
```

E.
```
is_null(result) || head(ys) !== head(result)
? pair(result, list(head(ys)))
: result
```

F.
```
head(ys) !== head(result) || is_null(result)
? pair(result, list(head(ys)))
: result
```

Section D: Active Lists [9 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 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` returns `null`.

The ***length*** of an active list `alist` is defined as the smallest non-negative integer `k` such that `alist(k)` is `null`. The function `active_length` takes as argument an active list and returns the length of the active list.

(11) [4 marks]

Complete the function `remove_elem` that takes as arguments an active list `A`, and a non-negative integer `pos`, and returns an active list that has the same elements as `A` but with `A(pos)` removed. The value of `pos` is assumed less than the length of `A`. Do not use the `make_active_list` function in your solution.

```
function remove_elem(A, pos) {
    return /* YOUR SOLUTION */
}
```

Example:

```
const as = make_active_list(list(8, 3, 5, 7));
const bs = remove_elem(as, 2);
active_length(bs); // returns 3
list(bs(0), bs(1), bs(2), bs(3));
// returns list(list(8), list(3), list(7), null)
```

In the following space, write your solution only for the part that is marked ***/* YOUR SOLUTION */***.

```
p => p < pos ? A(p) : A(p + 1);
```

(12) [5 marks]

Complete the function `zip` that takes as arguments two active lists `A` and `B` of the **same length**, and returns an active list in which the elements of `A` and `B` are interleaved. For example,

```
const as = make_active_list(list(11, 22, 33));
const bs = make_active_list(list(44, 55, 66));
const cs = zip(as, bs);
active_length(cs); // returns 6
list(cs(0), cs(1), cs(2), cs(3), cs(4), cs(5));
// returns list(list(11), list(44), list(22), list(55), list(33), list(66))
```

Do not use the `make_active_list` function in your solution.

```
function zip(A, B) {
    return /* YOUR SOLUTION */
}
```

In the following space, write your solution only for the part that is marked ***/* YOUR SOLUTION */***.

```
p => p % 2 === 0 ? A(p / 2) : B((p - 1) / 2);
```

Section E: Find the Power [7 marks]

(13) [7 marks]

In JavaScript and Source, we can represent positive integers from the interval [1, 2^{1023}], but not all these integers can be represented precisely. Only a subset of these integers are ***distinguishable*** from their successors, i.e. `i !== i + 1` is `true`.

It is also a fact that if a positive power of two is **distinguishable** from its successor, then all smaller positive powers of two are distinguishable from their successors.

Complete the following program to find the **exponent** of the **largest power of two** in [1, 2^{1023}] that is **distinguishable** from its successor. It is also given that 2^0 is distinguishable from its successor, and 2^{1023} is not. Write your program such that it requires no more than 10 calls to the `distinguishable` function. You are allowed to use the Source functions `math_pow`, `math_floor`, and `math_ceil`.

```
function distinguishable(i) {
    return i !== i + 1;
}
function find(lo, hi) {
    if (hi - lo <= 1) {
        return lo;
    } else {
        /* YOUR SOLUTION */
    }
}
find(0, 1023);
```

If `find(0, 1023)` returns k, then 2^k is distinguishable from its successor, but 2^{k+1} is not distinguishable from its successor.

In the following space, write your solution only for the part that is marked `/* YOUR SOLUTION */`.

```
const mid = math_floor((lo + hi) / 2);
const pot = math_pow(2, mid);
return distinguishable(pot)
       ? find(mid, hi)
       : find(lo, mid);
```

Section F: Tree Processing [9 marks]

Consider the following function `accumulate_tree`, and the tree of numbers `T`.

```
function accumulate_tree(f1, f2, initial, tree) {
    return is_null(tree)
           ? initial
           : f2( is_list(head(tree))
                    ? accumulate_tree(f1, f2, initial, head(tree))
                    : f1(head(tree)),
                 accumulate_tree(f1, f2, initial, tail(tree)));
}

const T = list(1,
               list(2, 3, list(4, 5)),
               6,
               list(list(7, 8), 9),
               list(10)
              );
```

(14) [3 marks]

What is the result of evaluating the following Source statement?

```
accumulate_tree( x => x, (x, y) => x > y ? x : y, 7.5, T );
```

A. `1`
B. `7`
C. `7.5`
D. `8`
E. `10` **(answer)**
F. None of the other options is the correct answer

(15) [3 marks]

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

```
accumulate_tree( x => list(x), (x, y) => append(y, x), null, T );
```

A. `list(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`
B. `list(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)` **(answer)**
C. `list(1, list(2, 3, list(4, 5)), 6, list(list(7, 8), 9), list(10))`
D. `list(list(10), list(9, list(8, 7)), 6, list(list(5, 4), 3, 2), 1)`
E. None of the other options is the correct answer

(16) [3 marks]

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

```
accumulate_tree( x => x, (x, y) => pair(x, y), null, T );
```

A. `list(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`
B. `list(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)`
C. `list(1, list(2, 3, list(4, 5)), 6, list(list(7, 8), 9), list(10))` **(answer)**
D. `list(list(10), list(9, list(8, 7)), 6, list(list(5, 4), 3, 2), 1)`
E. None of the other options is the correct answer

Section G: Sorted Trees of Numbers [12 marks]

A ***sorted tree of numbers*** (***STon***) is either `null` or a pair whose head is a number or a **non-null** STon, and whose tail is a STon, such that all number elements from the head are smaller than all number elements from the tail.

For example, the following is a STon:

```
const my_ston = list( list(1, 2, list(3, 4),5), 6, list(7), 8, 9,
                      list( list(10), 11, list(12, 13, list(14, 15)) ) );
```

(17) [4 marks]

Complete the function `smallest` that takes as argument a non-null STon, and returns the smallest number element in the STon. Your solution must make good use of the sorted property of the STon to efficiently find the required element.

For efficiency reasons, your solution **should not use** the Source function `is_list`, and instead it **may use** either the `is_pair` or `is_number` function.

```
function smallest(ston) {
    return is_pair(head(ston)) /* YOUR SOLUTION */
}
smallest(my_ston);  // returns 1
```

In the following space, write your solution only for the part that is marked ***/* YOUR SOLUTION */***.

```
? smallest(head(ston)) : head(ston);
```

(18) [4 marks]

Complete the function `largest` that takes as argument a non-null SToN, and returns the largest number element in the SToN. Your solution must make good use of the sorted property of the SToN to efficiently find the required element.

For efficiency reasons, your solution **should not use** the Source function `is_list`, and instead it **may use** either the `is_pair` or `is_number` function.

```
function largest(ston) {
    return !is_null(tail(ston)) /* YOUR SOLUTION */
}
largest(my_ston);  // returns 15
```

In the following space, write your solution only for the part that is marked `/* YOUR SOLUTION */`.

```
? largest(tail(ston))
: is_pair(head(ston)) ? largest(head(ston)) : head(ston);
```

(19) [4 marks]

Complete the function `find` that takes as arguments a SToN and a number `x`, and returns `true` if `x` is one of the elements in the SToN, otherwise it returns `false`. Your solution must make good use of the sorted property and the tree structure of the SToN to efficiently compute the result.

Your solution **must make use** of the `smallest` function and/or `largest` function from the preceding questions.

For efficiency reasons, your solution **should not use** the Source function `is_list`, and instead it **may use** either the `is_pair` or `is_number` function.

```
function find(ston, x) {
    if (is_null(ston)) {
        return false;
    } else if (is_null(tail(ston))) {
        return is_pair(head(ston)) ? find(head(ston), x) : x === head(ston);
    } else {
        /* YOUR SOLUTION */
    }
}
find(my_ston, 12);   // returns true
find(my_ston, 3.5);  // returns false
```

In the following space, write your solution only for the part that is marked `/* YOUR SOLUTION */`.

```
return x >= smallest(tail(ston)
       ? find(tail(ston), x)
       : is_pair(head(ston)) ? find(head(ston), x) : x === head(ston);
```

——— END OF PAPER ———