NATIONAL UNIVERSITY OF SINGAPORE

CS1101S — PROGRAMMING METHODOLOGY

(AY2020/2021 SEMESTER 1)

FINAL ASSESSMENT

Time Allowed: **2 Hours**

INSTRUCTIONS

1. This assessment contains **16 Questions** in **5 Sections**.
2. The full score of this assessment is **66 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 §4** language. You are allowed access to these online reference pages:
   - ***Source §4 pre-declared constants and functions*** at `https://source-academy.github.io/source/source_4/global.html`
   - ***Specification of Source §4*** at `https://source-academy.github.io/source/source_4.pdf`
7. In any question, your answer may use **functions given in, or written by you for,** any **preceding question**. You can assume a correct solution from the preceding question as given, even if your solution from that preceding question was not correct.
8. **Follow the instructions of your invigilator or the module coordinator to submit your answers**.

Section A: Processes and Lists [12 marks]

(1) [2 marks]

The following function `last_pair` returns the last pair of a given non-empty list:

```
function last_pair(xs) {
    return is_null(tail(xs))
           ? xs
           : last_pair(tail(xs));
}
```

Does it give rise to an iterative or a recursive process?

A. Iterative

B. Recursive

C. Neither iterative nor recursive

A

(2) [2 marks]

We can make a copy of a given list using the `map` function:

```
function copy(xs) {
    return map(x => x, xs);
}
```

Does it give rise to an iterative or a recursive process?

A. Iterative

B. Recursive

C. Neither iterative nor recursive

B

(3) [2 marks]

Consider the following `mystery` function.

```
function mystery(x) {
    return x === 0
        ? true
        : mystery(x - 1) ? true : false;
}
```

Does it give rise to an iterative or a recursive process when applied to positive integers?

A. Iterative

B. Recursive

C. Neither iterative nor recursive

B.

(4) [6 marks]

The pre-declared function `accumulate` can be applied to "fold" a given list from right to left, starting from a given initial value, each time applying a given binary function.

Example:

```
accumulate( (x, y) => x / y, 2, list(24, 16, 8) )
```

evaluates to

```
24 / (16 / (8 / 2)) = 6
```

The function `accumulate` as given in the lectures gives rise to a recursive process. Write a function `accumulate_iter` that computes the same result as `accumulate`, but that gives rise to an *iterative process*. **Additional requirement: No pairs must be created by `accumulate_iter` when used instead of `accumulate` in the example above.**

still O(n) space ⇒ stored in diff. location

```
function accumulate_iter(f, init, xs) {
    /* YOUR SOLUTION */
}
```

(Write the *entire function declaration* of `accumulate_iter` in the space provided below.)

```
function accumulate_iter(f, init, xs){
    function helper(xs, acc){
        return is_null(xs)
            ? acc
            : helper(tail(xs), f(head(xs), acc));
    }
    return helper(f, reverse(xs), init);
}

function accumulate_iter(f, init, xs){
    function acc(ys, cont){
        return is_null(ys)
            ? cont(init)
            : acc(tail(ys), x => cont(f(head(ys), x)));
    }
    return acc(xs, x => x);
}
```

CPS

10(

```
fn fact(n)
  if n === 0
    ↳ 1
  else
    ↳ n * fact(n-1)

          wish = fact(n-1)
            ↳ n * wish

                               return

fn fact(n, ret)
  if (n === 0)
    ↳ ret(1)
  else
    ↳ fact(n-1, wish
         => ret(n * wish))

fact(5, res => ____ )
```

no explicit deferred ops.

wish

100ms.

click → server

when the server gives an answer => ~~do~~ so f(answer)

1) callback

Ajax

get('url', (ans)

=> f(ans))

get('url1', (ans1)

=> get('url2', (ans2)

shrek.

Promises

get('url1') pair ans1 get(url, 2)

.then((ans1) =>)

get('url2')

.then(ans2 => pair

(ans1, ans2))

.then

ans1 = await get('url1')

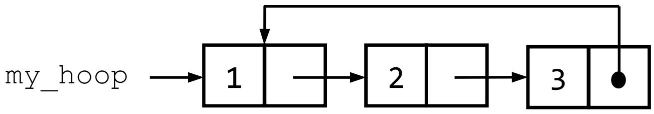
ans2 = await get('url2')

↳ pair(ans1, ans2)

Section B: Hoops [14 marks]

Definition: A ***hoop*** is a pair whose head is a number and whose tail is a hoop.

For example, the data structure `my_hoop` in the box-and-pointer diagram below is a hoop.



(5) [2 marks]

With the name `my_hoop` referring to the hoop depicted above (in the beginning of the section), what is the result of evaluating the following program where `length` is as given in Source §4?

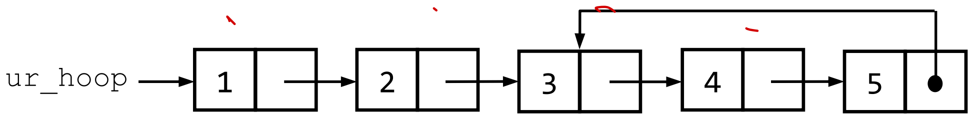
```
length(my_hoop);
```

A. 0
B. 2
C. 3
D. The number value `Infinity`
E. Error resulting from too many deferred operations
F. Error resulting from the `tail` function applied to a non-pair

E.

(6) [2 marks]

Consider the following hoop.



What is the result of evaluating the following program?

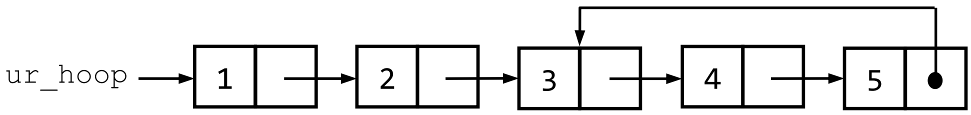
```
list_ref(ur_hoop, 10);
```
Starts from 0.

A. 5
B. 4
C. 3
D. The number value `Infinity`
E. Error resulting from too many deferred operations
F. Error resulting from the `tail` function applied to a non-pair

A.

(7) [4 marks]

Let us consider the results of accessing a hoop using the function `list_ref` applied to an index. Some results occur for a (practically) unbounded number of different indices. We say that a hoop *contains a number* `n` *infinitely often* if `n` results from applying `list_ref` to an unbounded number of different indices. For example, the following `ur_loop` contains 1 and 2 only finitely often, but 3, 4 and 5 infinitely often.



Write a function `hoopify` that takes a (finite) non-empty list of numbers `xs` as argument and returns a hoop that contains all elements of `xs` infinitely often. **Your function must make use of, in a meaningful way, the `last_pair` and `copy` functions from Section A.** Also make sure the original list `xs` is not changed by your function `hoopify`.

```
function hoopify(xs) {
    /* YOUR SOLUTION */
}
```

Example: `hoopify(list(1, 2, 3))` should return a hoop whose box-and-pointer diagram is the same as that of `my_hoop` as depicted in the beginning of the section.

(Write the *entire function declaration* of `hoopify` in the space provided below.)

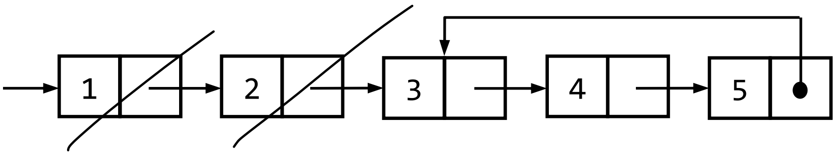
```
function hoopify(xs){
    let ys = copy(xs);
    set_tail( last_pair(ys), ys);
    return ys;
}
```

(8) [6 marks]

Write a function `partially_hoopify` that takes a list `xs` of `n` *distinct* numbers as first argument and a non-negative integer `m` as second argument, where `m` < `n`, and returns a hoop that contains all elements of `xs`. It should contain the first `m` elements of `xs` finitely often and the remaining elements of `xs` infinitely often. Make sure the original list `xs` is not changed by your function `partially_hoopify`.

```
function partially_hoopify(xs, m) {
    /* YOUR SOLUTION */
}
```

Example: `partially_hoopify(list(1, 2, 3, 4, 5), 2)` should return a hoop whose box-and-pointer diagram is as depicted below.



(Write the *entire function declaration* of `partially_hoopify` in the space provided below.)

```
function partially-hoopify (xs, m) {
    let ys = copy (xs);
    let p = ys;
    while (m > 0 ) {
        p = tail(p);
        m = m-1;
    }
    set_tail( last-pair (ys), p);
    return ys;
}.
```

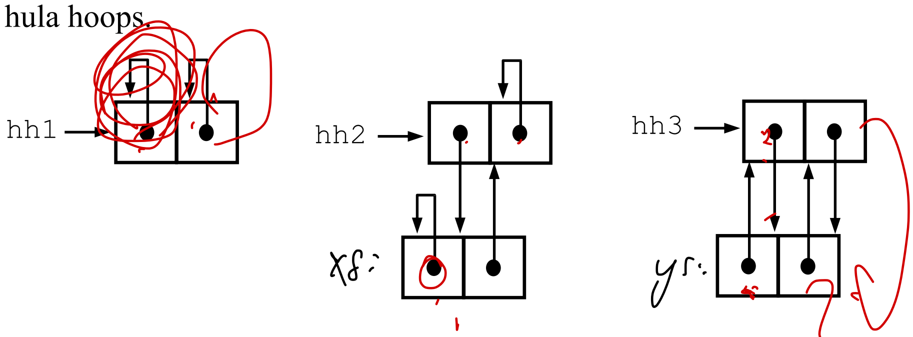
member

1. not a pair → false
2. seen before → true
3. f(head) && f(tail)

Section C: Hula Hoops [13 marks]

Definition: A ***hula hoop*** is a pair whose head and tail are hula hoops.

For example, the three data structures `hh1`, `hh2` and `hh3` indicated in the following box-and-pointer diagrams are hula hoops.



(9) [6 marks]

Complete the following program such that after evaluating the program, the names `hh1`, `hh2` and `hh3` refer to data structures depicted in the beginning of the section.

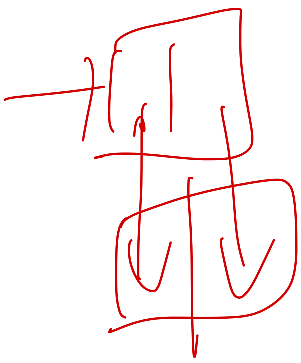
```
const hh1 = pair(undefined, undefined);
const hh2 = pair(undefined, undefined);
const hh3 = pair(undefined, undefined);
/* YOUR SOLUTION */
```

X

false

(Continue the program in the space provided below. You do not need to write again the statements provided above.)

```
set_head(hh1, hh1);
set_tail(hh1, hh1);
let xs = pair( xs , hh2);
set_head(hh2, xs);
set_tail(hh2, hh2);
let ys = pair( hh3, hh3 );
set_head(hh3, ys );
set_tail(hh3, ys );
```



(10) [7 marks]

Recall the *definition*: A ***hula hoop*** is a pair whose head and tail are hula hoops.

Interesting fact: No hoop is a hula hoop.

Write a predicate function `is_hula_hoop` that returns `true` if its argument is a hula hoop and `false` if it is not a hula hoop. Your function must **terminate for any non-pair input, and for any pair structure whose box-and-pointer diagram has a finite number of boxes**, as do all examples shown in the beginning of the section.

```
function is_hula_hoop(x) {
    /* YOUR SOLUTION */
}
```

(Write the *entire function declaration* of `is_hula_hoop` in the space provided below.)

```
function is_hula_hoop(x){
    let pairs = null;
    function check(y) {
        if (is_pair(y)) {
            if (! is_null(member(y, pairs))) {
                return true;
            } else {
                pairs = pair(y, pairs);
                return check(head(y)) && check(tail(y));
            }
        } else {
            return false;
        }
    }
    return check(x)
}
```

list of every

∃

not pair

Section D: Loops and Arrays [15 marks]

(11) [5 marks]

An ***identity matrix*** of size `n` is represented as an array of `n` arrays of numbers and each array of numbers is of length `n`. Elements in the identity matrix are all 0, except those in the major diagonal (from top-left to bottom-right) are all 1.

For example, the following is (the representation of) the identity matrix of size 4:

```
[[1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 1, 0],
 [0, 0, 0, 1]]
```

Write a function `identity` that takes a positive integer argument `n` and returns an identity matrix of size `n`.

```
function identity(n) {
    /* YOUR SOLUTION */
}
```

Examples:

```
identity(1); // returns [[1]]
identity(2); // returns [[1, 0], [0, 1]]
```

(Write the *entire function declaration* of `identity` in the space provided below.)

```
function identity(n) {
    let arr = [];
    for ( let i = 0; i < n; i = i+1){
        arr[i] = [];
        for (let j = 0; j < n; j = j+1){
            if (i === j) {
                arr[i][j] = 1;
            } else {
                arr[i][j] = 0;
            }
        }
    }
    return arr;
}
```

(12) [5 marks]

Write a function `zip_array` that takes in two arrays of numbers of *equal length*, and returns an array of numbers such that

- its first element is the first element of the first input array;
- its second element is the first element of the second input array;
- its third element is the second element of the first input array;
- its fourth element is the second element of the second input array;
- and so on.

```
function zip_array(arr1, arr2) {
    /* YOUR SOLUTION */
}
```

The result array must be the only data structure created in your function. Do not use lists. Also make sure the original input arrays `arr1` and `arr2` are not changed by your function.

Examples:

```
zip_array([], []); // returns []
zip_array([1, 2, 3], [10, 20, 30]); // returns [1, 10, 2, 20, 3, 30]
```

(Write the *entire function declaration* of `zip_array` in the space provided below.)

```
function zip_array(arr1, arr2){
    let N = array_length(arr1) + array_length(arr2);
    let x = 0;
    let y = 0;
    let arr = [];
    for (let i = 0; i < N; i = i+1){
        if (i % 2 === 0){
            arr[i] = arr1[x];
            x = x+1;
        } else {
            arr[i] = arr2[y];
            y = y+1;
        }
    }
    return arr;
}
```

(13) [5 marks]

Write a function `unzip_array` that takes in an array of numbers of even length, and returns a pair whose head is an array containing every even-indexed (0 is considered even) element of the input array, and whose tail is an array containing every odd-indexed element of the input array. The result arrays must contain their elements in the order in which they appear in the input array.

```
function unzip_array(arr) {
    /* YOUR SOLUTION */
}
```

The result arrays and the returned pair must be the only data structures created in your function. Do not use lists. Also make sure the original input array `arr` is not changed by your function.

Examples:

```
unzip_array([]); // returns [[], []]
unzip_array([1, 10, 2, 20, 3, 30]); // returns [[1, 2, 3], [10, 20, 30]]
```

(Write the *entire function declaration* of `unzip_array` in the space provided below.)

```
function unzip_array(arr){
    let N = array_length(arr);
    let x = []; let c_x = 0;
    let y = []; let c_y = 0;

    for (let i = 0; i < N; i = i+1){
        if(i % 2 === 0){
            x[c_x] = arr[i];
            c_x = c_x+1;
        } else {
            y[c_y] = arr[i];
            c_y = c_y+1;
        }
    }
    return pair(x, y);
}
```

Section E: Screams [12 marks]

"I'm a bit bored with infinite streams," says Pixel. "Why do their tails need to be *nullary* functions? Can't they have a couple of parameters? What if we make sure that their first parameter refers to a pair in the stream and their second parameter to an integer?" Pixel screams in delight. "I'll call them *screams*!"

Definition: A ***scream*** `s` is a pair whose tail is a binary function that returns a scream.

The function `scream_ref` takes a scream `s` and a non-negative integer `n` as arguments, and returns the element at position `n` in scream `s`.

Here is the implementation of the function `scream_ref`:

```
function scream_ref(s, n) {
    function helper(s, i, k) {
        return k === 0
               ? head(s)
               : helper(tail(s)(s, i + 1), i + 1, k - 1);
    }
    return helper(s, 0, n);
}
```

fn (prev, num)

position

Important fact: `scream_ref` applies the scream tails such that the first argument is the previous pair and the second argument is the position of the next pair in the scream.

Some example screams:

```
// the scream 1, 1, 1, 1, ...
const ones = pair(1, (ignore1, ignore2) => pair(1, tail(ones)));
scream_ref(ones, 200); // returns 1


// the scream 0, 1, 2, 3, ...
const integers = pair(0, (ignore, i) => pair(i, tail(integers)));
scream_ref(integers, 200); // returns 200


// the scream 0, 1, 2, 3, ... in an alternative way
const integers_alt =
    pair(0, (s, ignore) => pair(head(s) + 1, tail(integers_alt)));

scream_ref(integers_alt, 200); // returns 200
```

(14) [4 marks]

Complete the following declaration of the **factorials scream**, which must contain the elements 0!, 1!, 2!, 3!, 4!, ... (i.e. 1, 1, 2, 6, 24, ...), when using `scream_ref` to access the scream.

1 1 2x1 3x2x1 4x3x2x1

```
const factorials =
    pair(1, (s, i) =>
          // fill the following line, only using the names
          // pair, head, tail, factorials, s, i
          /* YOUR SOLUTION */
        );

scream_ref(factorials, 3); // returns 6
scream_ref(factorials, 5); // returns 120
```

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

(s,i) => pair(i * head(s), tail(factorial));

(15) [4 marks]

Consider the following ***pi-square series***:

$$\pi^2 = \frac{6}{1^2} + \frac{6}{2^2} + \frac{6}{3^2} + \frac{6}{4^2} + \cdots \approx 9.869604401089359$$

Complete the following declaration of the **pi_square_series scream**, which must contain the elements 0, $(6/1^2)$, $(6/1^2 + 6/2^2)$, $(6/1^2 + 6/2^2 + 6/3^2)$, $(6/1^2 + 6/2^2 + 6/3^2 + 6/4^2)$, ... when using `scream_ref` to access the scream.

```
const pi_square_series =
    pair(0, (s, i) =>
          // fill the following line, only using the names
          // pair, head, tail, pi_square_series, s, i
          /* YOUR SOLUTION */
        );

scream_ref(pi_square_series, 1); // returns 6
scream_ref(pi_square_series, 2); // returns 7.5
scream_ref(pi_square_series, 2000000); // returns 9.869601401089872
```

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

pair(6 /(i*i) + head(s) , tail(pi-square-series));

(16) [4 marks]

Complete the following declaration of the **`fibonacci` scream**, which must contain the elements 0, 1, 1, 2, 3, 5, 8, 13, ..., when using `scream_ref` to access the scream.

```
const fibonacci =
    pair(0,
         (s1, ignore) =>
             pair(1,
                  (s2, ignore) =>
                      pair(head(s1) + head(s2),
                           (s3, ignore) =>
                               // fill the following line,
                               // only using the names
                               // tail, s1, s2, s3
                               /* YOUR SOLUTION */
                          )
                 )
        );
scream_ref(fibonacci, 7); // returns 13
```

use this addition

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

pair(head(s3) + head(s2), tail(s3))

tail(tail(s1)(s2, 0))(s3, 0)