NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

MIDTERM QUIZ (ADAPTED TO AY2020/21 IN 9/2020)
Semester 1 AY2016/2017

CS1101S — PROGRAMMING METHODOLOGY

28 September 2016 Time Allowed: **1 Hour 35 Minutes**

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

Your Avenger's Name:

Instructions (please read carefully):

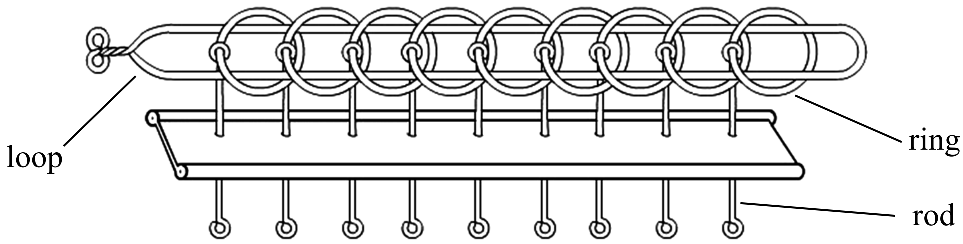
1. Write down your **matriculation number** on the **question paper**. DO NOT WRITE YOUR NAME ON THE QUESTION SET!
2. Write down your Avenger's name in the box provided above.
3. This is an **open-sheet quiz**. You are allowed to bring one A4 sheet of notes (written or printed on both sides).
4. This paper comprises **8 questions** and **TWENTY-THREE (23)** 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!

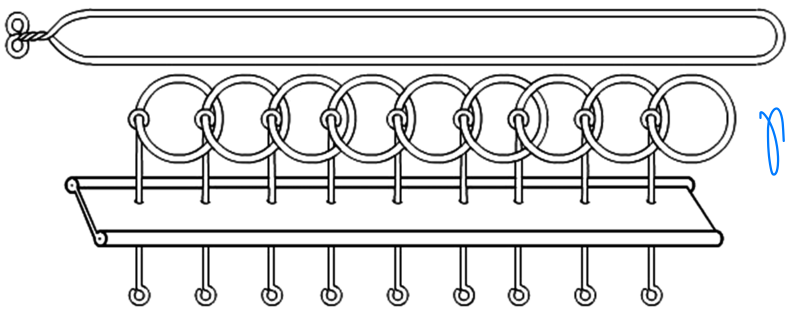
Q#	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Σ**
MAX	8	6	10	8	2	3	7	16	60
SC									

Background: The Nine Rings Puzzle

The *Nine Rings Puzzle* is a puzzle in which a given number n of rings (traditionally 9) are hanging on a wire loop. The rings are connected with each other by a system of rods that severely restricts their movement. The goal of the game is to remove all rings from the loop.



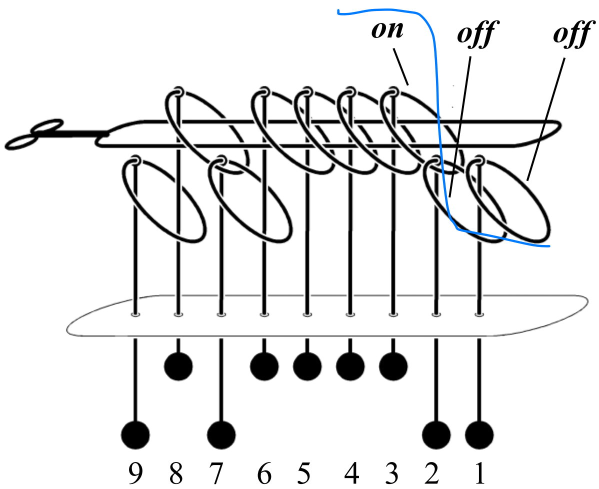
Initial configuration for 9 rings. All rings are ***on*** the loop.


Goal configuration for 9 rings. All rings are ***off*** the loop.

In order to describe the possible movements of the rings, the following definition will be useful:

A configuration of *i* rings is called a *<u>free "on" configuration</u>* if $i = 0$ (empty configuration)**, or the first ring is *on* the loop and *all* rings to its right are *off* the loop.**

For example, the three rightmost rings in the following picture form a ***free "on" configuration***, because the first ring is ***on*** and the two rings to its right are ***off*** the loop.



The three rightmost rings form a ***free "on" configuration***.

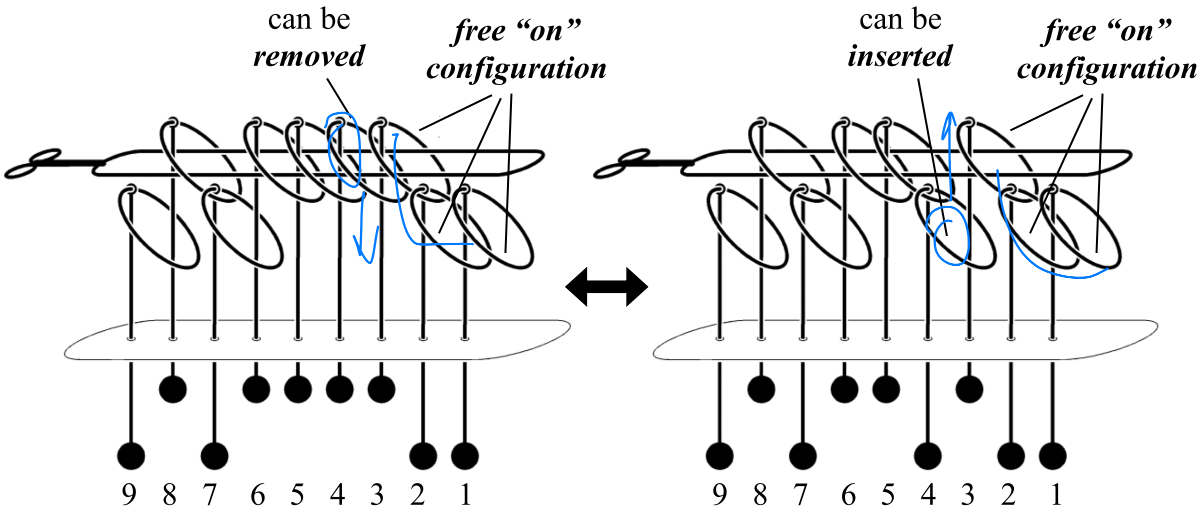
The following corresponding definition will come in handy later:

A configuration of i rings is called a *free "off" configuration* if $i = 0$, or the first ring is *off* the loop and *all* rings to its right are also *off* the loop.

off off

Now we can define the movements of the rings:

A ring can *be inserted* to or *removed* from the loop in one single step, if and only if all rings to its right form a *free "on" configuration*.



In the example above, the fourth ring from the right can be ***removed*** or ***inserted*** in one step, because the three rings to its right form a ***free "on" configuration***. (Don't worry if you cannot imagine how the fourth ring can be physically removed or inserted in one step. You just have to believe it can be done!)

Note that the definition above captures the case where there are no rings to the right. The rightmost ring can be moved freely *on* and *off* the loop.

For easier reference to the individual rings, we assign each ring an *ID* by labeling the rings 1 to n from right to left, i.e. the rightmost ring is Ring 1 and the leftmost Ring n.

n 1

Data Representation of Rings

In the following, we define an abstract data type that represents rings with a given *state* and a given *ID*. The data type defines a constructor `make_ring` and accessor functions `ring_state` and `ring_id`.

```
function make_ring(state, id) {
    return pair(state, id);
}

function ring_state(ring) {
   return head(ring);
}

function ring_id(ring) {
    return tail(ring);
}
```

Example use:

```
const my_ring = make_ring("off", 7);
ring_state(my_ring);   // returns "off"
ring_id(my_ring);      // returns 7

ring_state(make_ring("on", 3));   // returns "on"
```

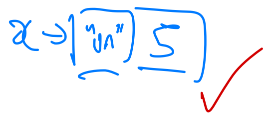
We will use this data representation of rings for all questions in this paper.

Question 1: Box-and-Pointer Diagrams [8 marks]

Using our representation of rings, for each of the following parts, draw the box-and-pointer diagram for the value of x. In each diagram, clearly show where x is pointing to.

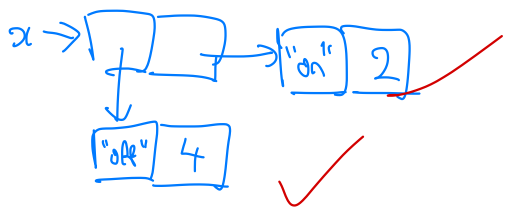
A. [1 mark]

```
const x = make_ring("on", 5);
```



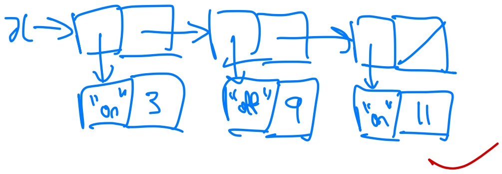
B. [1 mark]

```
const x = pair(make_ring("off", 4), make_ring("on", 2));
```



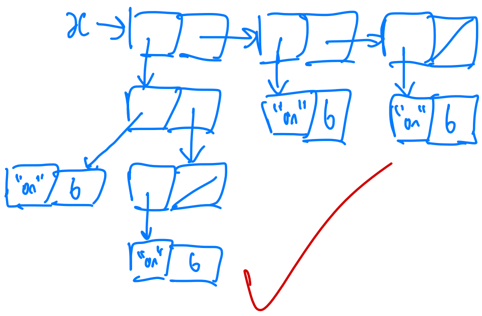
C. [2 marks]

```
const x = list(make_ring("on", 3), make_ring("off", 9),
               make_ring("on", 11));
```



D. [4 marks]

```
const s = make_ring("on", 6);
const t = list(s, s);
const x = list(t, s, make_ring("on", 6));
```



Question 2: Uniform Configurations [6 marks]

A ***uniform configuration*** is a list of rings that either is empty or only consists of elements of the same state whose IDs descend from *n* to 1, where *n* is the length of the list.

A. [4 marks]

Define a function `make_uniform_configuration` that, when given a `state` and a number `n`, constructs a configuration of `n` rings, all of which have the same given `state`. The IDs of the rings are descending from `n` to 1.

Example:

```
make_uniform_configuration("off", 3);
// returns list(make_ring("off", 3),
//              make_ring("off", 2),
//              make_ring("off", 1)).
```

To get the maximum 4 marks, you must make effective use of the **`build_list`** function, otherwise you get at most 2 marks.

```
function make_uniform_configuration(state, n) {
        return    reverse(build_list( n,
                                 x => make_list(state, x) )};

                 build_list(n, i => make_ring(state, n-i));
}
```

B. [1 mark]

Describe the runtime of your function with respect to the given number *n* using Θ notation.

$\Theta(n^2)$ $\Theta(n)$

C. [1 mark]

Describe the space consumption of your function with respect to the given number *n* using Θ notation.

$\Theta(n)$

Question 3: Free Configurations [10 marks]

Recall that a ***free "on" configuration*** is either the empty configuration, or a configuration that starts with an "***on***" ring, followed by a possibly empty uniform configuration of "***off***" rings.

Similarly, a ***free "off" configuration*** is either the empty configuration, or a configuration that starts with an "***off***" ring, followed by a possibly empty uniform configuration of "***off***" rings.

A. [3 marks]

Write a function `make_free_configuration` that takes a state `first_state` and a number `n` as arguments and returns a ***free configuration*** of length `n`. If `n` > 0, the state of the first ring should be the given state, and the IDs of the rings should be descending from `n` to 1.

```
function make_free_configuration(first_state, n) {
    return append( make_ring( first_state, n ),
                   make_uniform_configuration ("off", n-1));

    return n <== 0
           ? null
           : pair ( make_ring (first_state, n),
                    make_uniform_configuration ( "off", n-1)
           );
}
```

B. [7 marks]

Write a function `check_free_configuration` that takes a state `first_state` and a list of rings and returns true if and only if the given list of rings forms a ***free configuration*** and the first ring (if there is any) has the given state. Your function should also verify that the IDs of the rings are descending from *n* to 1, where *n* is the length of the list.

To get the maximum 7 marks, you must make effective use of the **`accumulate`** function, otherwise you get at most 4 marks.

```
function check_free_configuration(first_state, rings) {

    return is_null(rings)
           ? true
           : equals(ring_state(list_ref(rings, 0)), first_state)
               ? length(filter(x => !equals(ring_state(x), "off"), tail(rings))) > 1
                 : false
                 ? accumulate((x, y) => ring_id(x) > y
                                         ? true
                                         : false,
                              0,
                              rings)

           : false;

                    works?
}
```

Question 4: Steps [8 marks]

A. [4 marks]

Write a Source §2 program that defines an abstract data type that makes steps with a given *action* and a given *ring ID*. A step's action can be either "***insert***" or "***remove***". The data type needs to define a constructor `make_step` and accessor functions `step_action` and `step_id`. Furthermore, we need a function `step_to_string` that transforms a given step into a string as described in the example below.

Example:

```
const my_step = make_step("remove", 7);
step_action(my_step);          // returns "remove"
step_id(my_step);              // returns 7
step_to_string(my_step);       // returns "remove ring 7"
step_to_string(
    make_step("insert", 8));   // returns "insert ring 8"
```

```
function make_step(step, number) {
        return pair(step, number);
}
function step_action(step) {
        return head(step);
}
function step_id(step) {
        return tail(step);
}.
function step_to_string(step) {
        return step_action + "ring" + step_id;
}.
```

(step)

(step)

Stringify.

B. [4 marks]

Write a function `steps_to_string` that takes a given list of steps as argument and returns a string that describes the steps.

Example:

```
steps_to_string(list(make_step("remove", 7),
                     make_step("insert", 4)));
```

should return the string

```
remove ring 7
insert ring 4
```

There should be a newline character "\n" after every line in the output string.

To get the maximum 4 marks, you must make effective use of the **`map`** *and* **`accumulate`** functions. By making effective use of only one of `map` and `accumulate`, you get at most 3 marks and without these functions, you get at most 2 marks.

```
function steps_to_string(steps) {
    return accumulate ( (x,y) => append(x, append( list("\n"), y)),
                        null,
                        map(x => list(step_to_string(x)), steps));

                        accumulate( (x,y) => x+y,
                                    "",
                                    map (step => step_to_string(step)
                                         + "\n", steps)
                        );
}
```

Question 5: Flipping Rings [2 marks]

Write a function `flip` that takes a ring as argument and returns a step that flips the ring to the opposite state.

Example:

```
flip(make_ring("on", 7));  // returns make_step("remove", 7)
flip(make_ring("off", 4)); // returns make_step("insert", 4)
```

```
function flip(ring) {
    if ( equal( ring_state( ring) , "on" ) ){
        return make_step( "remove", ring_id(ring)) ;
    } else {
        return make_step( "insert" , ring_id(ring));
    }
}
```

Question 6: Solving the Puzzle [3 marks]

The key to solving the puzzle effectively is a function `steps_to_free_configuration` that takes a state `desired_first_state` and a configuration as argument, and that returns a list of steps needed to turn the configuration into a ***free configuration*** whose first ring has `desired_first_state` as state.

Example:

```
steps_to_free_configuration(
    "on",
    list(make_ring("off", 3),
         make_ring("on", 2),
         make_ring("off", 1))
);
```

returns a list of steps that turns the given configuration into the ***free "on" configuration***

```
list(make_ring("on", 3),
     make_ring("off", 2),
     make_ring("off", 1));
```

Assuming that you have such a function `steps_to_free_configuration`, define a function `solve` that takes a non-negative integer `n` as argument and returns a list of steps that turns a uniform configuration of "***on***" rings into a uniform configuration of "***off***" rings.

Hint: The function `make_uniform_configuration` from Question 2 may come in handy.

```
function solve(n) {
        const uniform_on = make_uniform_configuration("on", n);
        return steps_to_free_configuration("off", uniform_on);
}
```

Question 7: The Centre Piece [7 marks]

Questions 7 and 8B are **challenge questions**. You may want to attempt all other questions first before these.

The key to solving the puzzle effectively is a function `steps_to_free_configuration` that takes a state `desired_first_state` and a configuration as argument, and that returns a list of steps needed to turn the configuration into a ***free configuration*** whose first ring has `desired_first_state` as state.

Define the function `steps_to_free_configuration` described in the previous question. Recall that a ring can only be ***removed*** or ***inserted*** if the configuration on its right is a ***free "on" configuration***.

You can assume that the given list of rings have IDs descending from *n* to 1, where *n* is the length of the list.

```
function steps_to_free_configuration(desired_first_state,
                                     rings) {
```

See down

base.

```
  return is_null(rings)
         // base case: empty sequence is free configuration
         ? null
         : ring_state(head(rings)) === desired_first_state
           // first ring is ok: compute steps to make the
           // rest into a uniform off sequence
           ? steps_to_free_configuration("off", tail(rings))
           // we need to flip the first ring
           // (for this, the second ring must be on
           //  and the rest off)
           : append(steps_to_free_configuration("on",
                                                tail(rings)),
                    pair(flip(head(rings)),
                         steps_to_free_configuration("off",
                            make_free_configuration("on",
                                           length(rings) - 1))
                         )
                    )
         );
```

State of first ring true

make the rest of rings are "off"

free on.

false ?

3 steps

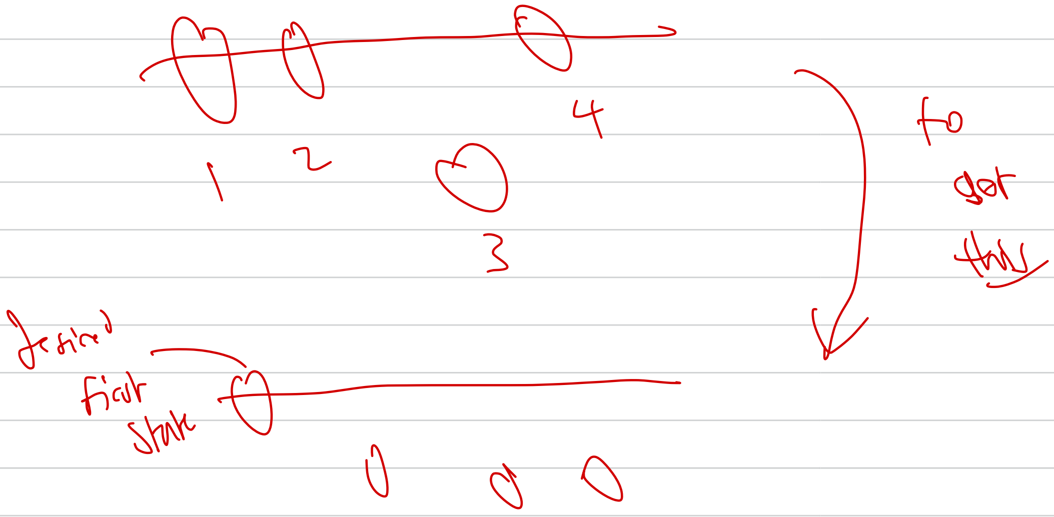
then flip the ring

→ remove the ring.

→ free on state.

make sure the rest of the rings are of "off" config.

(more writing space next page)


1
2
3
4
to get this
desired first state

```
}
```

Question 8: Legal Moves [16 marks]

Question 4 gives a method of describing actions as sequences of steps, where each step says what to do with a specific ring. In this question, we are describing such actions as functions that operate on configurations. In addition, we would like to make sure that we only make changes that are actually allowed, according to the rule stated in the beginning of the paper. This leads us to the following specification:

A ***legal move*** is a function that *takes a configuration and returns a configuration*. It performs a single unique step on the configuration *c* (removing or inserting a specific ring) and returns the resulting configuration, if that step can be performed on *c*. If the step cannot be performed on *c*, the configuration *c* is returned without change.

A. [5 marks]

Complete the following function `step_to_legal_move` that, when given a step, returns a legal move for the given step.

Example:

```
const my_legal_move = step_to_legal_move(my_step);
const new_config = my_legal_move(config);
```

`new_config` is the result configuration of carrying out the given `my_step` (if the step is legal), and is `config` otherwise.

You can assume that every configuration is a list of rings that have IDs descending from *n* to 1, where *n* is the length of the list. Also note that *n* may be smaller than the ID of `my_step`.

```
function step_to_legal_move(step) {

    // is_legal returns true if and only if step can be
    // carried out (is legal) on the configuration cfg.
    function is_legal(cfg) {
        const n = length(cfg);
        if (step_id(step) > n) {
            return false;
        } else {
            const the_ring = list_ref(cfg, n - step_id(step));
            const right = tail(member(the_ring, cfg));
            return (step_action(flip(the_ring)) ===
                     step_action(step)) &&
                    check_free_configuration("on", right);
        }
    }
```

(*more writing space next page*)

function helper(step)

```
    return config =>
        is_legal(config)
        ? map(r => ring_id(r) !== step_id(step)
                   ? r
                   : ring_state(r) === "off"
                     ? make_ring("on", ring_id(r))
                     : make_ring("off", ring_id(r)),
              config)
        : config;
```

}

B. [6 marks]

Questions 7 and 8B are **challenge questions**. You may want to attempt all other questions first before these.

function that takes in config,
return a config with a step applied to it

Define a function `legal_move_to_step` that takes a legal move as argument and returns the step that corresponds to the legal move.

string / tells you what to do.

Example:

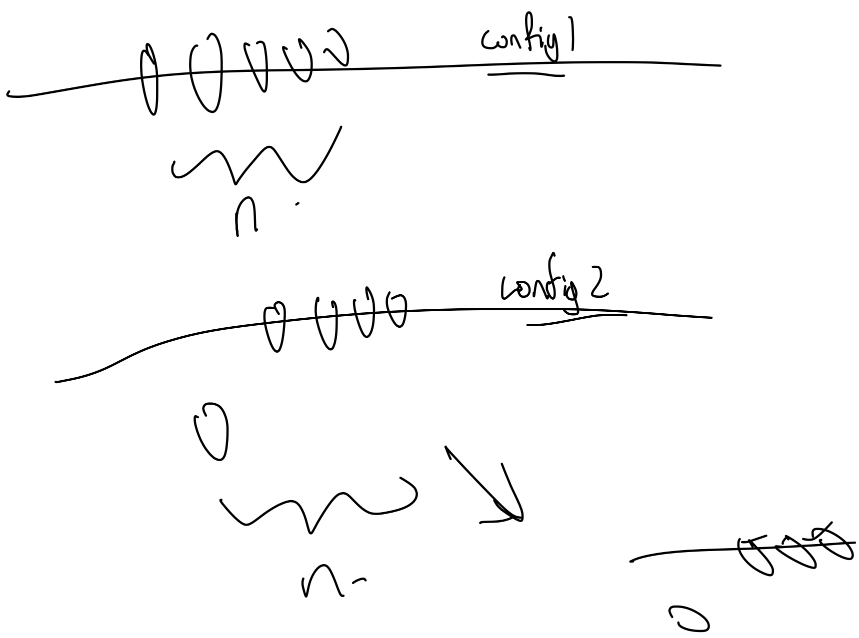
```
const a_legal_move = step_to_legal_move(make_step("remove", 7));
legal_move_to_step(a_legal_move);
// returns make_step("remove", 7)
```

```
function legal_move_to_step(legal_move) {

    function tryout(n) {
        const config1 = pair(make_ring("on", n),
                make_free_configuration("on", n - 1));
        const config2 = pair(make_ring("off", n),
                make_free_configuration("on", n - 1));

        return (legal_move(config1) !== config1)
            ? make_step("remove", n)
            : (legal_move(config2) !== config2)
              ? make_step("insert", n)
              : tryout(n + 1);
    }

    return tryout(1);
```




```
}
```

C. [5 marks]

The new representation of actions using legal moves allows us to verify any solution to Question 6. The idea is to take the result of the solver (a list of steps), transform them to a list of legal moves, and apply the moves to a correct initial configuration. If the result is the correct goal configuration, we can have very high confidence that the solver works.

Write a checker function `check_solver` that takes a number `n` and a `solve` function as argument and verifies the `solve` function using an initial configuration with `n` rings. The checker returns true if the `solve` function passes the test and false otherwise.

Example:

```
check_solver(9, solve);
// returns true for a correct solve function according to Q6
check_solver(9, n => null);
// returns false
```

```
function check_solver(n, solve) {

    const steps = solve(n);
    const legal_moves = map(step_to_legal_move, steps);
    const initial_config = make_uniform_configuration("on", n);
    const final_config = accumulate(
                             (x, y) => x(y),
                             initial_config,
                             reverse(legal_moves));

    return check_free_configuration("off", final_config);

}
```

—— 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. Do not tear off.)

(Scratch Paper. Do not tear off.)

NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

MIDTERM QUIZ (ADAPTED TO AY2020/21 IN 9/2020)
Semester 1 AY2016/2017

CS1101S — PROGRAMMING METHODOLOGY

28 September 2016 Time Allowed: **1 Hour 35 Minutes**

SOLUTION

Matriculation No.:

Your Avenger's Name:

Instructions (please read carefully):

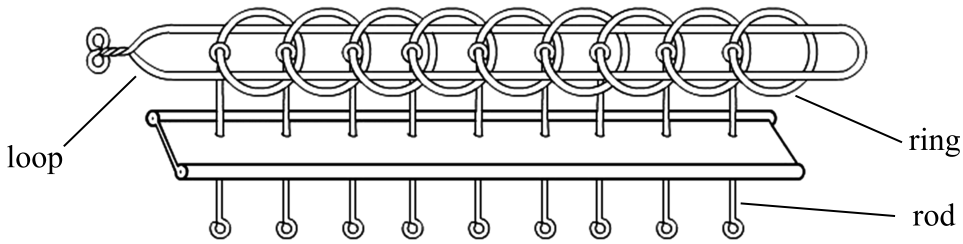
1. Write down your **matriculation number** on the **question paper**. DO NOT WRITE YOUR NAME ON THE QUESTION SET!
2. Write down your Avenger's name in the box provided above.
3. This is an **open-sheet quiz**. You are allowed to bring one A4 sheet of notes (written or printed on both sides).
4. This paper comprises **8 questions** and **TWENTY-THREE (23)** 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!

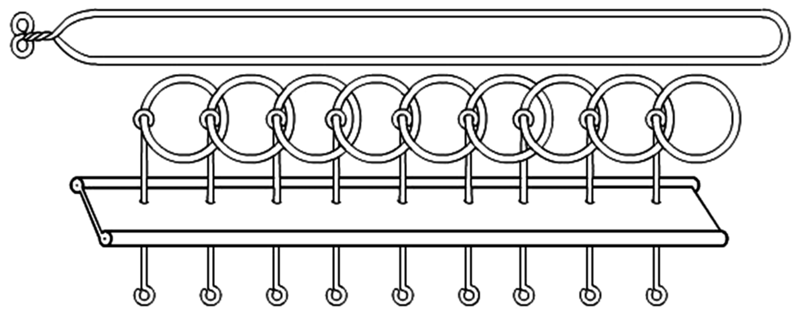
Q#	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Σ**
MAX	8	6	10	8	2	3	7	16	60
SC									

Background: The Nine Rings Puzzle

The *Nine Rings Puzzle* is a puzzle in which a given number *n* of rings (traditionally 9) are hanging on a wire loop. The rings are connected with each other by a system of rods that severely restricts their movement. The goal of the game is to remove all rings from the loop.



Initial configuration for 9 rings. All rings are ***on*** the loop.

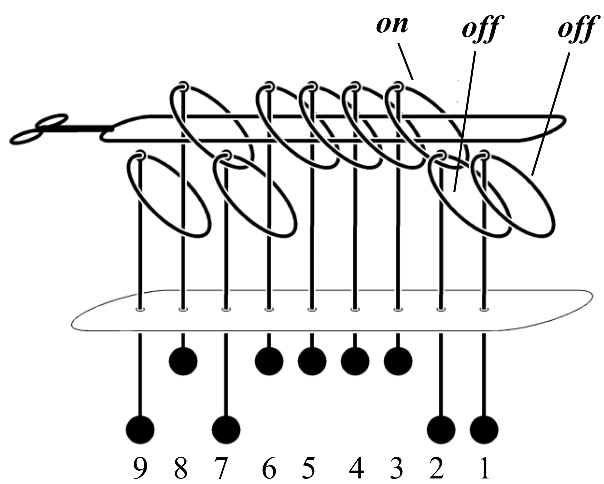


Goal configuration for 9 rings. All rings are ***off*** the loop.

In order to describe the possible movements of the rings, the following definition will be useful:

A configuration of *i* rings is called a *free "on" configuration* if *i* = 0 (empty configuration)**, or the first ring is *on* the loop and *all* rings to its right are *off* the loop.**

For example, the three rightmost rings in the following picture form a ***free "on" configuration***, because the first ring is ***on*** and the two rings to its right are ***off*** the loop.



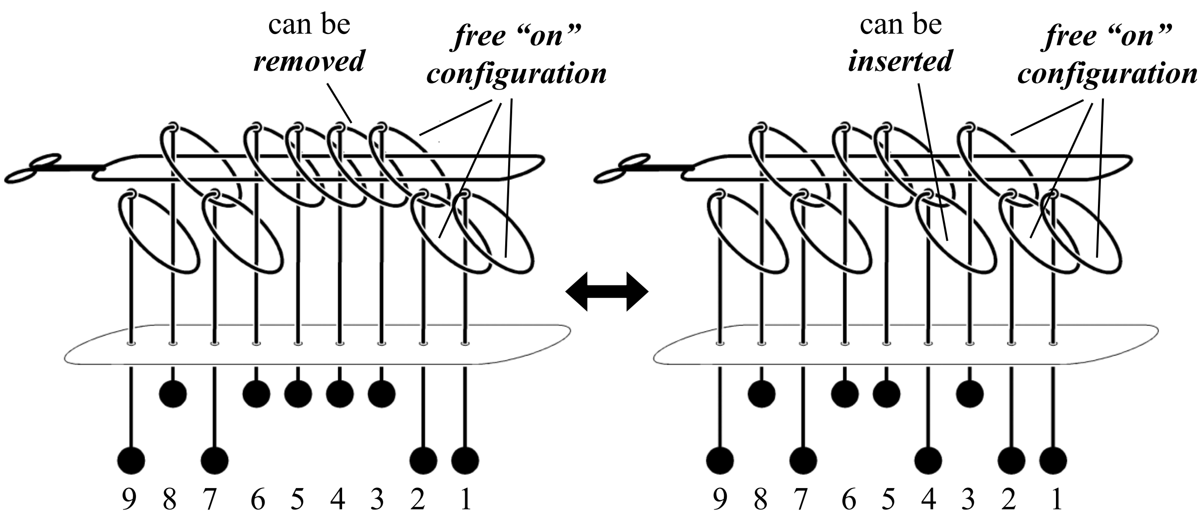
The three rightmost rings form a ***free "on" configuration***.

The following corresponding definition will come in handy later:

A configuration of *i* rings is called a <u>*free "off" configuration*</u> if *i* = 0, or the first ring is *off* the loop and *all* rings to its right are also *off* the loop.

Now we can define the movements of the rings:

A ring can be *inserted* to or *removed* from the loop in one single step, if and only if all rings to its right form a *free "on" configuration*.



In the example above, the fourth ring from the right can be ***removed*** or ***inserted*** in one step, because the three rings to its right form a ***free "on" configuration***. (Don't worry if you cannot imagine how the fourth ring can be physically removed or inserted in one step. You just have to believe it can be done!)

Note that the definition above captures the case where there are no rings to the right. The rightmost ring can be moved freely *on* and *off* the loop.

For easier reference to the individual rings, we assign each ring an *ID* by labeling the rings 1 to *n* from right to left, i.e. the rightmost ring is Ring 1 and the leftmost Ring *n*.

Data Representation of Rings

In the following, we define an abstract data type that represents rings with a given *state* and a given *ID*. The data type defines a constructor `make_ring` and accessor functions `ring_state` and `ring_id`.

```
function make_ring(state, id) {
    return pair(state, id);
}

function ring_state(ring) {
   return head(ring);
}

function ring_id(ring) {
    return tail(ring);
}
```

Example use:

```
const my_ring = make_ring("off", 7);
ring_state(my_ring);  // returns "off"
ring_id(my_ring);     // returns 7

ring_state(make_ring("on", 3));  // returns "on"
```

We will use this data representation of rings for all questions in this paper.

Question 1: Box-and-Pointer Diagrams [8 marks]

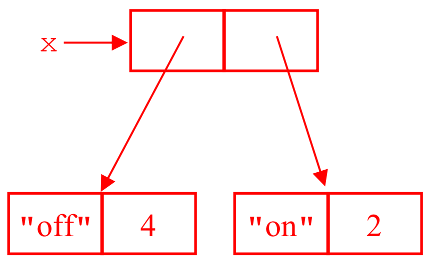
Using our representation of rings, for each of the following parts, draw the box-and-pointer diagram for the value of x. In each diagram, clearly show where x is pointing to.

A. [1 mark]

```
const x = make_ring("on", 5);
```

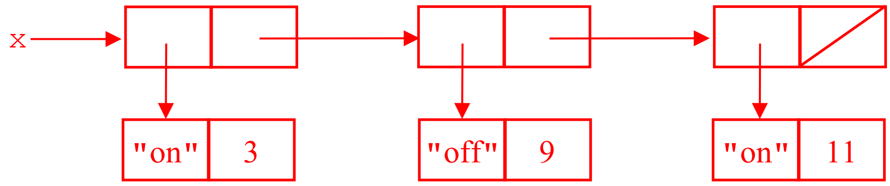
B. [1 mark]

```
const x = pair(make_ring("off", 4), make_ring("on", 2));
```



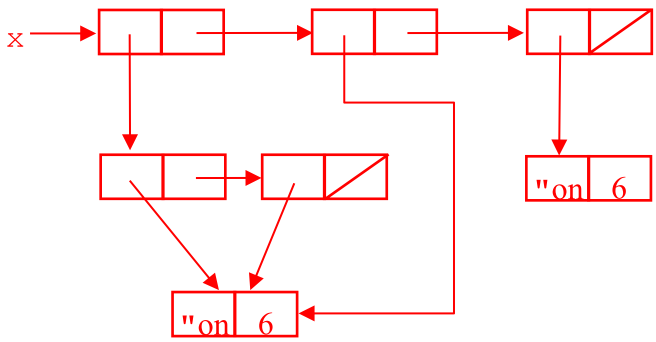
C. [2 marks]

```
const x = list(make_ring("on", 3), make_ring("off", 9),
               make_ring("on", 11));
```

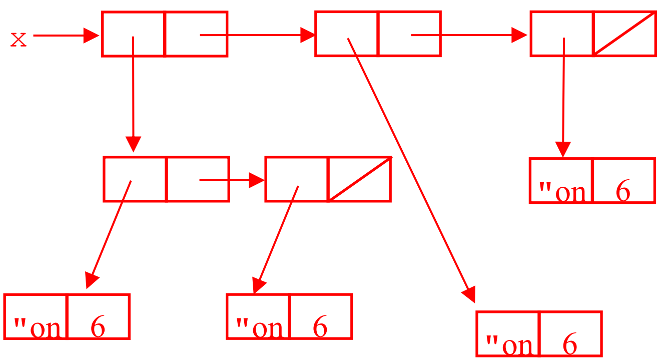


D. [4 marks]

```
const s = make_ring("on", 6);
const t = list(s, s);
const x = list(t, s, make_ring("on", 6));
```



OR



Question 2: Uniform Configurations [6 marks]

A ***uniform configuration*** is a list of rings that either is empty or only consists of elements of the same state whose IDs descend from *n* to 1, where *n* is the length of the list.

A. [4 marks]

Define a function `make_uniform_configuration` that, when given a `state` and a number `n`, constructs a configuration of `n` rings, all of which have the same given `state`. The IDs of the rings are descending from `n` to 1.

Example:

```
make_uniform_configuration("off", 3);
// returns list(make_ring("off", 3),
//                 make_ring("off", 2),
//                 make_ring("off", 1)).
```

To get the maximum 4 marks, you must make effective use of the **`build_list`** function, otherwise you get at most 2 marks.

```
function make_uniform_configuration(state, n) {

    return build_list(n, i => make_ring(state, n - i));

}
```

B. [1 mark]

Describe the runtime of your function with respect to the given number *n* using Θ notation.

$\Theta(n)$

C. [1 mark]

Describe the space consumption of your function with respect to the given number *n* using Θ notation.

$\Theta(n)$

Question 3: Free Configurations [10 marks]

Recall that a ***free "on" configuration*** is either the empty configuration, or a configuration that starts with an "***on***" ring, followed by a possibly empty uniform configuration of "***off***" rings.

Similarly, a ***free "off" configuration*** is either the empty configuration, or a configuration that starts with an "***off***" ring, followed by a possibly empty uniform configuration of "***off***" rings.

A. [3 marks]

Write a function `make_free_configuration` that takes a state `first_state` and a number `n` as arguments and returns a ***free configuration*** of length `n`. If `n` > 0, the state of the first ring should be the given state, and the IDs of the rings should be descending from `n` to 1.

```
function make_free_configuration(first_state, n) {

    return n === 0
           ? null
           : pair(make_ring(first_state, n),
                  make_uniform_configuration("off", n - 1)
                 );

}
```

B. [7 marks]

Write a function `check_free_configuration` that takes a state `first_state` and a list of rings and returns true if and only if the given list of rings forms a ***free configuration*** and the first ring (if there is any) has the given state. Your function should also verify that the IDs of the rings are descending from *n* to 1, where *n* is the length of the list.

To get the maximum 7 marks, you must make effective use of the **`accumulate`** function, otherwise you get at most 4 marks.

```
function check_free_configuration(first_state, rings) {

// Solution 1 [7 marks]:
    return is_null(rings)
        ? true
        : ring_state(head(rings)) === first_state &&
          ring_id(head(rings)) === length(rings) &&
          tail(accumulate(
                  (r, p) => pair(head(p) + 1,
                                 tail(p) &&
                                 ring_state(r) === "off" &&
                                 ring_id(r) === head(p)),
                 pair(1, true),
                 tail(rings)));


// Solution 2 [4 marks]:
    function helper(expected_id, rs) {
        return is_null(rs)
            ? true
            : ring_state(head(rs)) === "off" &&
              ring_id(head(rs)) === expected_id &&
              helper(expected_id - 1, tail(rs));
    }

    return is_null(rings)
        ? true
        : ring_state(head(rings)) === first_state &&
          ring_id(head(rings)) === length(rings) &&
          helper(ring_id(head(rings)) - 1, tail(rings));









}
```

Question 4: Steps [8 marks]

A. [4 marks]

Write a Source §2 program that defines an abstract data type that makes steps with a given *action* and a given *ring ID*. A step's action can be either "***insert***" or "***remove***". The data type needs to define a constructor `make_step` and accessor functions `step_action` and `step_id`. Furthermore, we need a function `step_to_string` that transforms a given step into a string as described in the example below.

Example:

```
const my_step = make_step("remove", 7);
step_action(my_step);          // returns "remove"
step_id(my_step);              // returns 7
step_to_string(my_step);       // returns "remove ring 7"
step_to_string(
    make_step("insert", 8)); // returns "insert ring 8"
```

```
function make_step(action, id) {
    return pair(action, id);
}

function step_action(step) {
   return head(step);
}

function step_id(step) {
    return tail(step);
}

function step_to_string(step) {
    return step_action(step) +
           " ring " +
           stringify(step_id(step));
}
```

B. [4 marks]

Write a function `steps_to_string` that takes a given list of steps as argument and returns a string that describes the steps.

Example:

```
steps_to_string(list(make_step("remove", 7),
                     make_step("insert", 4)));
```

should return the string

```
remove ring 7
insert ring 4
```

There should be a newline character "`\n`" after every line in the output string.

To get the maximum 4 marks, you must make effective use of the **`map`** *and* **`accumulate`** functions. By making effective use of only one of `map` and `accumulate`, you get at most 3 marks and without these functions, you get at most 2 marks.

```
function steps_to_string(steps) {

    return accumulate(
               (x, y) => x + y,
               "",
               map(step => step_to_string(step) + "\n", steps)
           );

}
```

Question 5: Flipping Rings [2 marks]

Write a function `flip` that takes a ring as argument and returns a step that flips the ring to the opposite state.

Example:

```
flip(make_ring("on", 7));  // returns make_step("remove", 7)
flip(make_ring("off", 4)); // returns make_step("insert", 4)
```

```
function flip(ring) {

    return make_step(ring_state(ring) === "off"
                     ? "insert"
                     : "remove",
                     ring_id(ring));

}
```

Question 6: Solving the Puzzle [3 marks]

The key to solving the puzzle effectively is a function `steps_to_free_configuration` that takes a state `desired_first_state` and a configuration as argument, and that returns a list of steps needed to turn the configuration into a ***free configuration*** whose first ring has `desired_first_state` as state.

Example:

```
steps_to_free_configuration(
    "on",
    list(make_ring("off", 3),
         make_ring("on", 2),
         make_ring("off", 1))
);
```

returns a list of steps that turns the given configuration into the ***free "on" configuration***

```
list(make_ring("on", 3),
     make_ring("off", 2),
     make_ring("off", 1));
```

Assuming that you have such a function `steps_to_free_configuration`, define a function `solve` that takes a non-negative integer `n` as argument and returns a list of steps that turns a uniform configuration of "***on***" rings into a uniform configuration of "***off***" rings.

Hint: The function `make_uniform_configuration` from Question 2 may come in handy.

```
function solve(n) {

    const initial_configuration =
        make_uniform_configuration("on", n);

    return steps_to_free_configuration("off",
                                       initial_configuration);







}
```

Question 7: The Centre Piece [7 marks]

Questions 7 and 8B are **challenge questions**. You may want to attempt all other questions first before these.

The key to solving the puzzle effectively is a function `steps_to_free_configuration` that takes a state `desired_first_state` and a configuration as argument, and that returns a list of steps needed to turn the configuration into a ***free configuration*** whose first ring has `desired_first_state` as state.

Define the function `steps_to_free_configuration` described in the previous question. Recall that a ring can only be ***removed*** or ***inserted*** if the configuration on its right is a ***free "on" configuration***.

You can assume that the given list of rings have IDs descending from *n* to 1, where *n* is the length of the list.

```
function steps_to_free_configuration(desired_first_state,
                                     rings) {

  return is_null(rings)
         // base case: empty sequence is free configuration
         ? null
         : ring_state(head(rings)) === desired_first_state
           // first ring is ok: compute steps to make the
           // rest into a uniform off sequence
           ? steps_to_free_configuration("off", tail(rings))
           // we need to flip the first ring
           // (for this, the second ring must be on
           //  and the rest off)
           : append(steps_to_free_configuration("on",
                                                tail(rings)),
                    pair(flip(head(rings)),
                         steps_to_free_configuration("off",
                            make_free_configuration("on",
                                                    length(rings) - 1)
                         )
                    )
             );
```

(*more writing space next page*)

```
}
```

Question 8: Legal Moves [16 marks]

Question 4 gives a method of describing actions as sequences of steps, where each step says what to do with a specific ring. In this question, we are describing such actions as functions that operate on configurations. In addition, we would like to make sure that we only make changes that are actually allowed, according to the rule stated in the beginning of the paper. This leads us to the following specification:

A ***legal move*** is a function that *takes a configuration and returns a configuration*. It performs a single unique step on the configuration *c* (removing or inserting a specific ring) and returns the resulting configuration, if that step can be performed on *c*. If the step cannot be performed on *c*, the configuration *c* is returned without change.

A. [5 marks]

Complete the following function `step_to_legal_move` that, when given a step, returns a legal move for the given step.

Example:

```
const my_legal_move = step_to_legal_move(my_step);
const new_config = my_legal_move(config);
```

`new_config` is the result configuration of carrying out the given `my_step` (if the step is legal), and is `config` otherwise.

You can assume that every configuration is a list of rings that have IDs descending from *n* to 1, where *n* is the length of the list. Also note that *n* may be smaller than the ID of `my_step`.

```
function step_to_legal_move(step) {

    // is_legal returns true if and only if step can be
    // carried out (is legal) on the configuration cfg.
    function is_legal(cfg) {
        const n = length(cfg);
        if (step_id(step) > n) {
            return false;
        } else {
            const the_ring = list_ref(cfg, n - step_id(step));
            const right = tail(member(the_ring, cfg));
            return (step_action(flip(the_ring)) ===
                     step_action(step)) &&
                   check_free_configuration("on", right);
        }
    }
```

(*more writing space next page*)

```
    return config =>
        is_legal(config)
        ? map(r => ring_id(r) !== step_id(step)
                   ? r
                   : ring_state(r) === "off"
                     ? make_ring("on", ring_id(r))
                     : make_ring("off", ring_id(r)),
              config)
        : config;
```

```
}
```

B. [6 marks]

Questions 7 and 8B are **challenge questions**. You may want to attempt all other questions first before these.

Define a function `legal_move_to_step` that takes a legal move as argument and returns the step that corresponds to the legal move.

Example:

```
const a_legal_move = step_to_legal_move(make_step("remove", 7));
legal_move_to_step(a_legal_move);
// returns make_step("remove", 7)
```

```
function legal_move_to_step(legal_move) {

    function tryout(n) {
        const config1 = pair(make_ring("on", n),
                make_free_configuration("on", n - 1));
        const config2 = pair(make_ring("off", n),
                make_free_configuration("on", n - 1));

        return (legal_move(config1) !== config1)
            ? make_step("remove", n)
            : (legal_move(config2) !== config2)
              ? make_step("insert", n)
              : tryout(n + 1);
    }

    return tryout(1);

}
```

C. [5 marks]

The new representation of actions using legal moves allows us to verify any solution to Question 6. The idea is to take the result of the solver (a list of steps), transform them to a list of legal moves, and apply the moves to a correct initial configuration. If the result is the correct goal configuration, we can have very high confidence that the solver works.

Write a checker function `check_solver` that takes a number `n` and a `solve` function as argument and verifies the `solve` function using an initial configuration with `n` rings. The checker returns true if the `solve` function passes the test and false otherwise.

Example:

```
check_solver(9, solve);
// returns true for a correct solve function according to Q6
check_solver(9, n => null);
// returns false
```

```
function check_solver(n, solve) {

    const steps = solve(n);
    const legal_moves = map(step_to_legal_move, steps);
    const initial_config = make_uniform_configuration("on", n);
    const final_config = accumulate(
                             (x, y) => x(y),
                             initial_config,
                             reverse(legal_moves));

    return check_free_configuration("off", final_config);

}
```

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.

- `equal(x, y)`: Returns `true` if `x` and `y` have the same structure (using pairs and `null`), and corresponding leaves are ===, and `false` otherwise.

(Scratch Paper. Do not tear off.)

(Scratch Paper. Do not tear off.)