National University of Singapore

CS1101S — Programming Methodology

AY2022/2023 Semester 1

Final Assessment

Time allowed: 2 hours

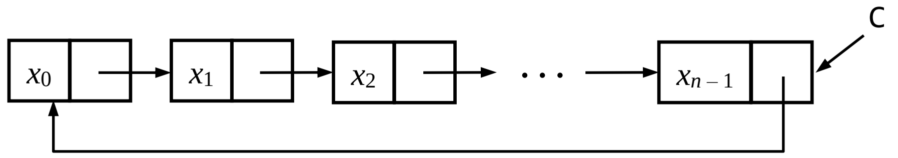
SOLUTIONS

INSTRUCTIONS

1. This **QUESTION PAPER** contains **21** Questions in **7** Sections, and comprises **XX** printed pages, including this page.
2. The **ANSWER SHEET** comprises **XX** printed pages.
3. Use a pen or pencil to **write** your **Student Number** in the designated space on the front page of the **ANSWER SHEET**, and **shade** the corresponding circle **completely** in the grid for each digit or letter. DO NOT WRITE YOUR NAME!
4. You must **submit only** the **ANSWER SHEET** and no other documents. Do not tear off any pages from the ANSWER SHEET.
5. All questions must be answered in the space provided in the **ANSWER SHEET**; no extra sheets will be accepted as answers.
6. Write legibly with a **pen** or **pencil** (do not use red color). Untidiness will be penalized.
7. For **multiple choice questions (MCQ)**, **shade** in the **circle** of the correct answer **completely**.
8. The full score of this assessment is **100** marks.
9. This is a **Closed-Book** assessment, but you are allowed to bring with you one double-sided **A4 / letter-sized sheet** of handwritten or printed **notes**.
10. Where programs are required, write them in the **Source §4** language. A **reference** of some of the **pre-declared functions** is given in the **Appendix** of the Question Paper.
11. In any question, unless it is specifically allowed, your answer **must not use functions** given in, or written by you for, other questions.

Section A: Rear Circular Lists [20 marks]

We represent a non-empty sequence of numbers (x_0, x_1, x_2, ... , x_{n-1}), where $n \geq 1$, in a data structure `C` that has the following box-and-pointer diagram:



Note that `C` is always referring to the pair that has the last element of the sequence.

We call such a data structure a ***Rear Circular List*** (***RCL***). Note that an RCL must have **at least one element**.

Supposed there is a function `make_RCL`, which takes as argument a non-empty list `L` of numbers and returns a RCL that represents the same sequence as the input list, the following shows an example run of `make_RCL`:

```
const L = list(22, 33, 44, 55);
const C = make_RCL(L);
head(C);  // returns 55
head(tail(C));  // returns 22
head(tail(tail(C)));  // returns 33
head(tail(tail(tail(C))));  // returns 44
head(tail(tail(tail(tail(C)))));  // returns 55
```

(1) [3 marks]

Complete the declaration of function `insert_last`, which takes as arguments a RCL `C` and a number `x`, and returns a RCL that is the result of inserting the number `x` as a new element to the end of the sequence represented by the input RCL.

The result RCL must reuse all the existing pairs of the input RCL. Your function must not use the `set_head` function. It must take $O(1)$ time to run.

Example:

```
const L = list(11, 22, 33, 44, 55);
const C = make_RCL(L);
head(C);        // returns 55
head(tail(C));  // returns 11
const D = insert_last(C, 66);
head(D);        // returns 66
head(tail(D));  // returns 11
head(C);        // returns 55
head(tail(C));  // returns 66
```

```
function insert_last(C, x) {

    const new_pair = pair(x, tail(C));
    set_tail(C, new_pair);
    return new_pair;

}
```

(2) [4 marks]

Complete the declaration of function `make_RCL`, which takes as argument a ***non-empty*** list `L` of numbers, and returns a RCL that represents the same sequence as the input list. Your implementation of `make_RCL` must make use of the `insert_last` function from the preceding question in a meaningful way.

Your function must not modify the input list, and must not use the `set_head` function. It must take $O(n)$ time to run, where n is the length of the input list.

Write your answer only in the dashed-line box.

```
function make_RCL(L) {
    let C = pair(head(L), null); // new pair for the first element.
    set_tail(C, C);  // make an RCL for the first element.

    for (let p = tail(L); !is_null(p); p = tail(p)) {
        C = insert_last(C, head(p));
    }
    return C;
}
```

(3) [4 marks]

Complete the declaration of function `append_RCLs`, which takes as arguments two RCLs `C1` and `C2`, and returns a RCL that represents the sequence resulting from appending the sequence of `C2` after the sequence of `C1`.

The result RCL must reuse all the existing pairs of the input RCLs. Your function must not use the `set_head` function. It must take $O(1)$ time to run.

Example:

```
const L1 = list(11, 22);
const L2 = list(33, 44);
const C1 = make_RCL(L1);
const C2 = make_RCL(L2);
const D  = append_RCLs(C1, C2);
head(D);                       // returns 44
head(tail(D));                 // returns 11
head(tail(tail(D)));           // returns 22
head(tail(tail(tail(D))));     // returns 33
```

```
function append_RCLs(C1, C2) {

    const C1_front = tail(C1);
    set_tail(C1, tail(C2));
    set_tail(C2, C1_front);
    return C2;

}
```

(4) [5 marks]

Complete the declaration of function `remove_last`, which takes as argument a RCL `C` that has ***at least 2 elements***, and returns a RCL that is the result of removing the last element of the sequence represented by the input RCL.

All pairs used in the result RCL must come from the existing pairs of the input RCL. Your function must not use the `set_head` function. It must take $O(n)$ time to run, where n is the length of the sequence represented by the input RCL.

Example:

```
const L = list(11, 22, 33, 44);
const C = make_RCL(L);
const D = remove_last(C);
head(D);                       // returns 33
head(tail(D));                 // returns 11
head(tail(tail(D)));           // returns 22
head(tail(tail(tail(D))));     // returns 33
```

Write your answer only in the dashed-line boxes.

```
function remove_last(C) {
    // Find second last pair.
    let second_last = C;
    while (tail(second_last) !== C) {
        second_last = tail(second_last);
    }
    set_tail(second_last, tail(C));
    return second_last;
}
```

(5) [4 marks]

Complete the declaration of function `RCL_to_stream`, which takes as argument a RCL `C`, and returns a finite stream that represents the sequence represented by the input RCL.

Your function must not modify the input RCL. It must take *O*(1) time to run.

Example:

```
const L = list(11, 22, 33, 44);
const C = make_RCL(L);
const S = RCL_to_stream(C);
stream_to_list(S); // returns list(11, 22, 33, 44)
```

Write your answer only in the dashed-line box.

```
function RCL_to_stream(C) {
    function helper(D) {
        return D === C
               ? pair(head(D), () => null)
               : pair(head(D), () => helper(tail(D)));
    }
    return helper(tail(C));
}
```

Section B: Binary Min Trees [19 marks]

A ***binary tree*** is either empty or has an ***entry***, a ***left branch*** and a ***right branch***, where the entry is a number and the left and right branches are binary trees.

The entry of a binary tree *T*, together with the entries of its left and right branches, and all the entries of their left and right branches, and so on, are the ***elements*** of *T*.

A ***binary min tree*** (***BmT***) is a binary tree that has an additional property — **for each tree whose entry is *m*, all elements of its left branch *and* right branch are greater than *m*, and all elements of its left branch are less than all elements of its right branch** (we assume no duplicate elements in the entire tree).

You can access binary min trees **using only** the following functions, which provide the ***binary tree abstraction***:

- **`is_empty_tree(tree)`** — Tests whether the given binary tree `tree` is empty.
- **`is_tree(x)`** — Checks if `x` is a binary tree.
- **`left_branch(tree)`** — Returns the left subtree of `tree` if `tree` is not empty.
- **`entry(tree)`** — Returns the value of the entry of `tree` if `tree` is not empty.
- **`right_branch(tree)`** — Returns the right subtree of `tree` if `tree` is not empty.
- **`make_empty_tree()`** — Returns an empty binary tree.
- **`make_tree(value, left, right)`** — Returns a binary tree with entry `value`, left subtree `left`, and right subtree `right`.

You must use this abstraction for all the questions in this section, and should not make any assumption about how binary trees are represented and implemented.

(6) [3 marks]

Complete the declaration of function `find_2nd_min`, which takes as argument a binary min tree `bmt`, which has ***at least two elements***, and returns the second smallest element of `bmt`. Your solution must make good use of the property of the BmT to efficiently compute the result.

Example:

```
const L = make_tree(33, make_empty_tree(), make_empty_tree());
const R = make_tree(44, make_empty_tree(), make_empty_tree());
const Ta = make_tree(22, L, R);
const Tb = make_tree(22, make_empty_tree(), R);
find_2nd_min(Ta); // returns 33
find_2nd_min(Tb); // returns 44
```

Write your answer only in the dashed-line box.

```
function find_2nd_min(bmt) {
  return !is_empty_tree(left_branch(bmt))
         ? entry(left_branch(bmt))
         : entry(right_branch(bmt));
}
```

(7) [5 marks]

Complete the declaration of function `find_max`, which takes as argument a ***non-empty*** binary min tree `bmt`, and returns the greatest element of `bmt`. Your solution must make good use of the property of the BmT to efficiently compute the result.

Example:

```
const L = make_tree(33, make_empty_tree(), make_empty_tree());
const R = make_tree(44, make_empty_tree(), make_empty_tree());
const Ta = make_tree(22, L, R);
const Tb = make_tree(22, L, make_empty_tree());
find_max(L); // returns 33
find_max(Ta); // returns 44
find_max(Tb); // returns 33
```

Write your answer only in the dashed-line box.

```
function find_max(bmt) {
  return is_empty_tree(left_branch(bmt))
         && is_empty_tree(right_branch(bmt))
         ? entry(bmt)
         : is_empty_tree(right_branch(bmt))
         ? find_max(left_branch(bmt))
         : find_max(right_branch(bmt));
}
```

(8) [6 marks]

Complete the declaration of function `find_x`, which takes as arguments a binary min tree `bmt`, a number `x`, and returns `true` if an element of `bmt` is equal to `x`, and returns `false` otherwise. Your solution must make good use of the property of the BmT to efficiently compute the result.

Example:

```
const L = make_tree(33, make_empty_tree(), make_empty_tree());
const R = make_tree(44, make_empty_tree(), make_empty_tree());
const T = make_tree(22, L, R);
find_x(make_empty_tree(), 99); // returns false
find_x(T, 99); // returns false
find_x(T, 11); // returns false
find_x(T, 38); // returns false
find_x(T, 33); // returns true
```

Write your answer only in the dashed-line box.

```
function find_x(bmt, x) {

  return is_empty_tree(bmt)
         ? false
         : x === entry(bmt)
         ? true
         : x < entry(bmt)
         ? false
         : !is_empty_tree(right_branch(bmt))
               && x >= entry(right_branch(bmt))
         ? find_x(right_branch(bmt), x)
         : find_x(left_branch(bmt), x);

}
```

(9) [5 marks]

Complete the declaration of function `flatten`, which takes as arguments a binary min tree `bmt`, and returns a list of numbers that contains all the elements of `bmt` arranged in ***descending order***. Your solution must make good use of the property of the BmT to efficiently compute the result.

Example:

```
const L = make_tree(33, make_empty_tree(), make_empty_tree());
const R = make_tree(44, make_empty_tree(), make_empty_tree());
const T = make_tree(22, L, R);
flatten(T); // returns list(44, 33, 22)
```

Write your answer only in the dashed-line box.

```
function flatten(bmt) {
  return is_empty_tree(bmt)
         ? null
         : append(flatten(right_branch(bmt)),
                  append(flatten(left_branch(bmt)),
                         list(entry(bmt))));
}
```

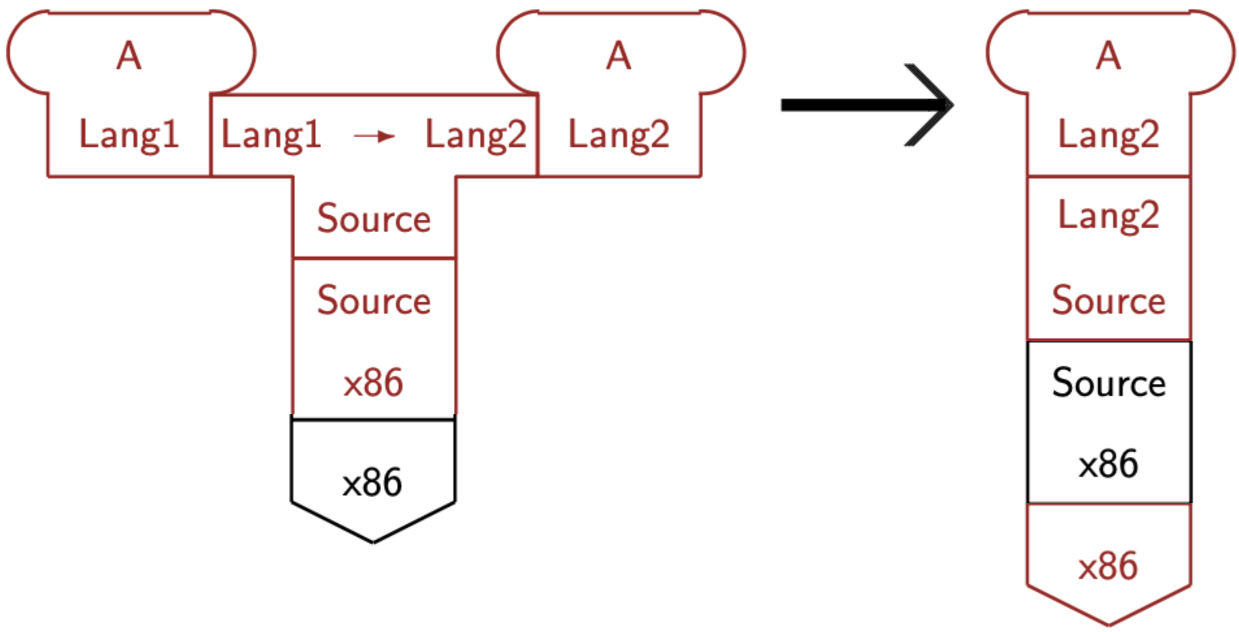
Section C: Tombstone Diagrams [8 marks]

(10) [8 marks]

Consider the following:

- You have a **Lang1** to **Lang2 Compiler** written in **Source**.
- You have a program **A** written in **Lang1**.
- You have a **Source interpreter** written in **x86** and a computer that can run **x86**.
- You also have a **Lang2 interpreter** written in **Source**.
- You would like to run your program **A**.

Draw the tombstone diagrams (T-diagrams) to demonstrate how you would get the program **A** into a state where it can be run, and how you would run it.



Section D: Environment Model [16 marks]

For each of the following Source programs, draw the diagram to show the environment during the evaluation of the program. Show all the frames that are created during the program evaluation (except the global environment frame). Show the final value of each binding.

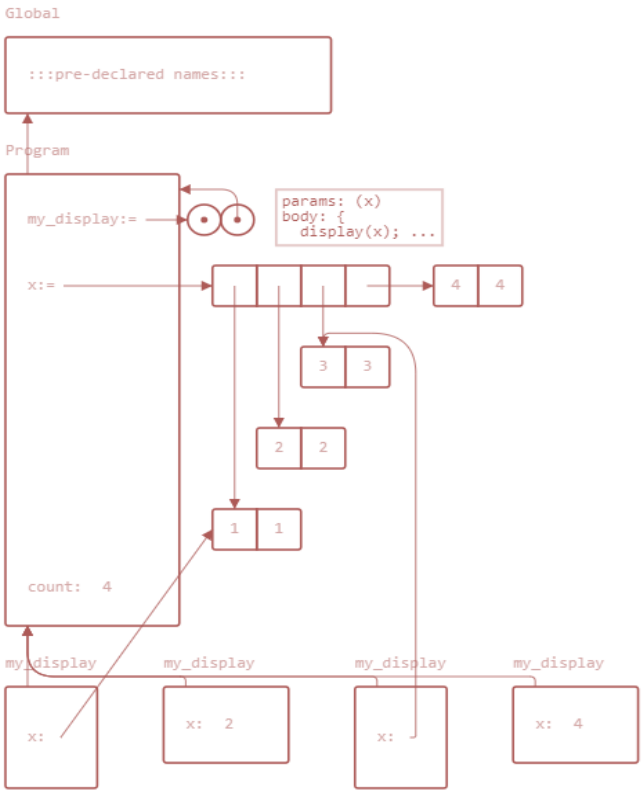
Note that the application of a **primitive** function (e.g. `array_length`, `pair`, `display`, `tail`) does not create a new environment frame.

(11) [8 marks]

Program D1:

```
function my_display(x) {
    display(x);
}

const x = [pair(1,1), pair(2,2), pair(3,3), pair(4,4)];
let count = 0;

while (count < array_length(x)) {
    if (count % 2 === 0) {
        my_display(x[count]);
    } else {
        my_display(tail(x[count]));
    }
    count = count + 1;
}
```

Environment of Program D1:

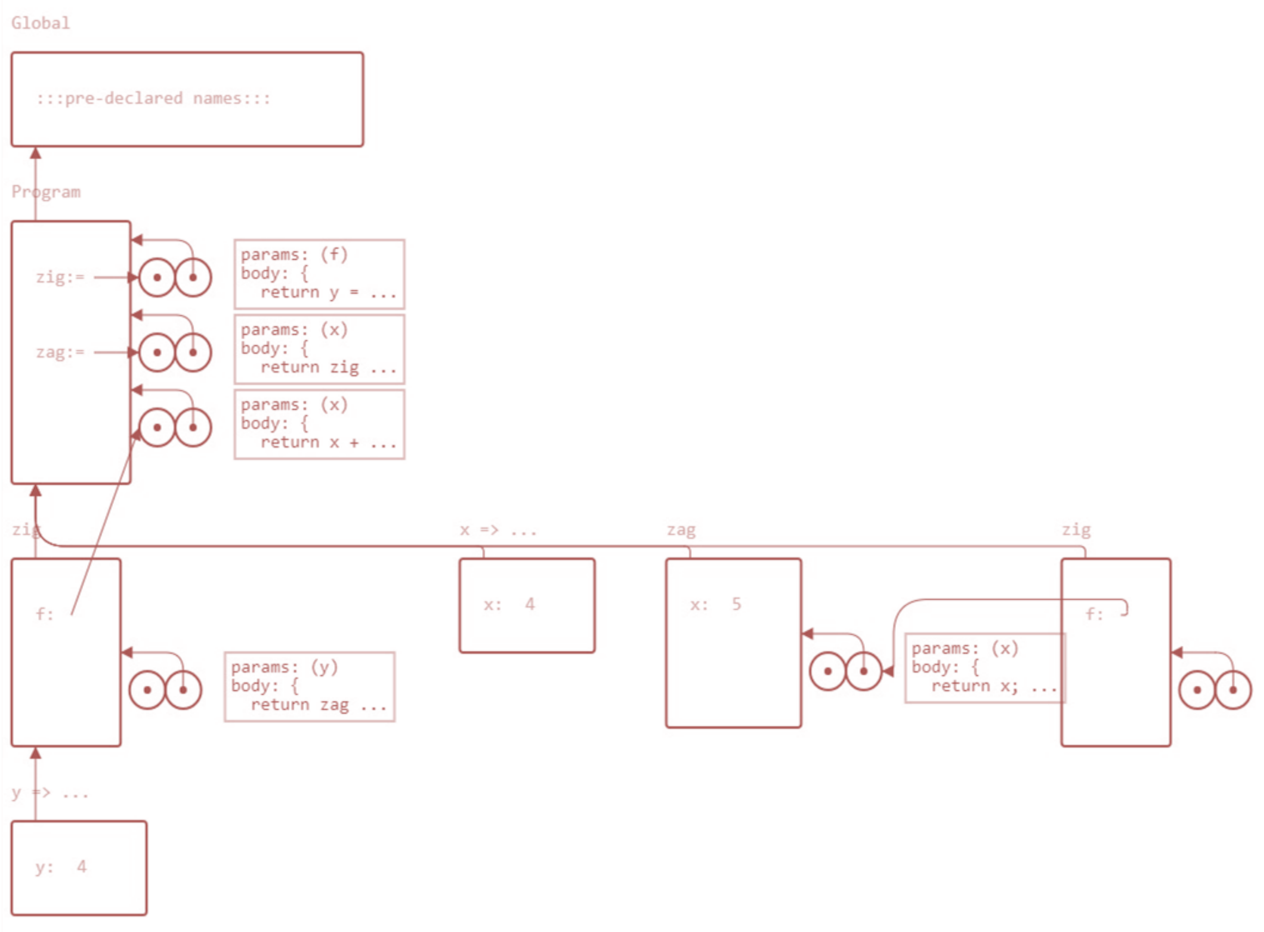


(12) [8 marks]

Program D2:

```
function zig(f) {
    return y => zag(f(y));
}

function zag(x) {
    return zig(x => x);
}

zig(x => x + 1)(4);
```

Environment of Program D2:



Section E: Matrices [6 marks]

(13) [6 marks]

The ***upper triangular mask*** of size n is a $n \times n$ matrix whose elements are all 0, except the elements on and above the major diagonal (from top left to bottom right) are all 1.

We represent the upper triangular mask of size n as an array of n arrays of numbers, where each of the n arrays is of length n. For example, the following is the representation of the upper triangular mask of size 4.

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

Complete the declaration of function `utm`, which takes in a positive integer argument `n` and returns the upper triangular matrix of size `n`.

Examples:

```
utm(1); // returns [[1]]
utm(3); // returns [[1, 1, 1], [0, 1, 1], [0, 0, 1]]
```

Write your answer only in the dashed-line box.

```
function utm(n) {
    const M = [];
    for (let r = 0; r < n; r = r + 1) {
        M[r] = [];
        for (let c = 0; c < n; c = c + 1) {
            if (r <= c) {
                M[r][c] = 1;
            } else {
                M[r][c] = 0;
            }
        }
    }
    return M;
}
```

Section F: Streams [20 marks]

Consider the following **Program F**:

```
const st1 = stream_filter(x => x % 2 === 1, integers_from(0));
eval_stream(st1, 6);
```

(You may refer to the Appendix for the declarations of the predeclared functions `stream_filter`, `eval_stream` and `integers_from`.)

(14) [3 marks] (MCQ)

What is the result of evaluating Program F in *list notation*?

A. `list(0, 1, 3, 5, 7, 9)`

B. `list(1, 3, 5, 7, 9, 11)` (answer)

C. `list(0, 2, 4, 6, 8, 10)`

D. `list(2, 4, 6, 8, 10, 12)`

E. `list(3, 5, 7, 9, 11, 13)`

F. `list(0, 1, 2, 3, 4, 5)`

(15) [3 marks] (MCQ)

How many **pairs** are created during the evaluation of Program F?

A. 6

B. 15

C. 16

D. 18

E. 24

F. 27 (answer)

(16) [3 marks] (MCQ)

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

```
function new_stream(a, b) {
    return pair(a, () => stream_map(x => -x, new_stream(-b, a + 2)));
}

const st2 = new_stream(1, 2);
eval_stream(st2, 8);
```

A. `list(1, 3, -1, -3, 1, 3, -1, -3)`

B. `list(1, 2, 3, 0, -1, -2, -3, 0)`

C. `list(1, -2, 3, 1, -2, 3, 1, -2)`

D. `list(1, 2, -3, 0, 1, 2, -3, 0)`   (answer)

E. `list(1, -1, 1, -1, 1, -1, 1, -1)`

F. `list(1, 2, -3, 1, 2, -3, 1, 2)`

(17) [3 marks] (MCQ)

Recall the function `memo_fun` from the lectures:

```
function memo_fun(fun) {
    let already_run = false;
    let result = undefined;

    function mfun() {
        if (!already_run) {
            result = fun();
            already_run = true;
            return result;
        } else {
            return result;
        }
    }
    return mfun;
}
```

Now, consider the following Source program:

```
function evens() {
    function helper(x) {
        display(x);
        return pair(x, memo_fun(() => helper(x + 2)));
    }
    return helper(2);
}

const st = evens();
eval_stream(st, 3);
eval_stream(st, 6);
```

How many **pairs** are created during the evaluation of the above program?

A. 1

B. 5

C. 6

D. 12

E. 16 (answer)

F. 21

(18) [5 marks]

Variadic functions are functions that can be passed a variable number of arguments. In Source, we have encountered a couple of variadic functions, for example the function `stream`:

```
stream(1, 2, 3)    // returns a stream with the elements 1, 2, and 3
stream(1, 2, 3, 4) // returns a stream with the elements 1, 2, 3, and 4
```

In Source, we can create a variadic function using the ***rest*** syntax. Functions written with this syntax can only have one parameter. Three dots (`...`) are placed in front of that parameter's name, which means that this parameter can accept a variable number of arguments. For example:

```
function variadic(...args) {
    // implementation removed
}

variadic(1, 2, 3);
variadic(1, 2, 3, 4);
```

The binding of `args` that is ultimately passed to the function `variadic` is an array. For example, in the function application `variadic(1, 2, 3, 4)`, `args` will be bound to an array `[1, 2, 3, 4]`.

Complete the declaration of function `stream`, which takes in a variable number of arguments and returns a finite stream containing those elements.

```
function stream(...xs) {
    const n = array_length(xs);

    function helper(i) {
        return i >= n
               ? null
               : pair(xs[i], () => helper(i + 1));
    }
    return helper(0);

}
```

(19) [3 marks]

In this question, we will use an even lazier version of streams, where the head of our stream is also a nullary function. This allows us to delay the evaluation of the element of the stream until we need it. Consider the following function `lazier_stream` which returns a ***lazier stream***, when passed a function `f` and a list of elements `xs`.

```
function lazier_stream(f, xs) {
    return pair(() => f(head(xs)), () => lazier_stream(f, tail(xs)));
}
```

Complete the declaration of function **`lazier_stream_element`**, which that takes as arguments a lazier stream **`s`** and a number **`n`**, and returns the stream element at index **`n`**. It should not evaluate any element apart from the one with index **`n`**. For example:

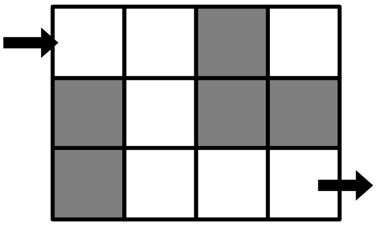
```
const s = lazier_stream(math_sqrt, list(4, 9, 16));
lazier_stream_element(s, 1); // returns 3
```

```
function lazier_stream_element(s, n) {

    return n === 0
           ? head(s)()
           : lazier_stream_element(stream_tail(s), n - 1);


}
```

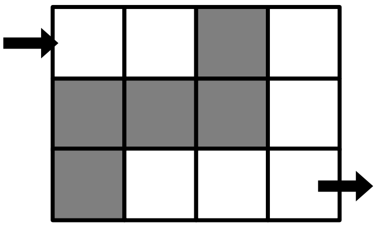
Section G: Solving the Maze [11 marks]

A ***maze*** is a rectangular room with walls and paths. We aim to write a maze solver using the Source language. We enter the maze at the top-left corner, and we need to escape the maze at the bottom-right corner. Our maze solver would return `true` if there is a solution to the maze and `false` if the maze is not solvable.

We represent the maze using a matrix where the value `1` represents a cell of wall and the value `0` represents a cell of unobstructed space. The following diagram shows two mazes and their corresponding matrix representations.



```
const maze_1 = [[0, 0, 1, 0],
                [1, 0, 1, 1],
                [1, 0, 0, 0]];
```



```
const maze_2 = [[0, 0, 1, 0],
                [1, 1, 1, 0],
                [1, 0, 0, 0]];
```

In our coordinate system, the topmost row of the matrix is row 0 and the leftmost column of the matrix is column 0. We use the ordered pair (r, c) to refer to the position of the cell at row r and column c. With this, the above `maze_1` is solvable by following the path (0, 0) – (1, 1) – (2, 1) – (2, 2) – (2, 3). There is no unobstructed path from (0, 0) to (2, 3) in `maze_2`, therefore it is not solvable.

(20) [5 marks]

First, we will write a helper function to identify the next cells we can move to from a given current position. At any given point, we have three possible moves: move 1 step down, move 1 step right, or move 1 step diagonally down and right. A move is not valid if it is moving into a wall cell.

Complete the function `valid_next`, which takes as arguments a current position (as `row` and `col`) and a maze `maze`, and returns an array of valid next positions. Each valid next position is represented as a two-element array `[r, c]`, where `r` is its row and `c` its column.

Examples:

```
valid_next(0, 0, maze_1); // returns [[0, 1], [1, 1]]
valid_next(2, 1, maze_1); // returns [[2, 2]]
valid_next(0, 1, maze_2); // returns []
```

Write your answer only in the dashed-line box.

```
function valid_next(row, col, maze) {

    const n_rows = array_length(maze);     // number of rows
    const n_cols = array_length(maze[0]); // number of columns

    const next = [[row, col+1], [row+1, col], [row+1, col+1]];
    let n_valids = 0;
    const ans = [];

    for (let i = 0; i < array_length(next); i = i + 1) {
        if (next[i][0] < n_rows  &&  next[i][1] < n_cols
                && maze[next[i][0]][next[i][1]] === 0) {
            ans[n_valids] = next[i];
            n_valids = n_valids + 1;
        }

    }

    return ans;
}
```

(21) [6 marks]

Complete the function `is_solvable`, which takes as arguments the start position (as `row` and `col`) and a maze `maze`, and returns `true` if `maze` is solvable and `false` otherwise.

Examples:

```
is_solvable(0, 0, maze_1); // returns true
is_solvable(0, 0, maze_2); // returns false
```

Note that the given start position will always be in an unobstructed cell (not a wall). Your function may make use of the `valid_next` function from the preceding question.

Write your answer only in the dashed-line boxes.

```
function is_solvable(row, col, maze) {
    const n_rows = array_length(maze);     // number of rows
    const n_cols = array_length(maze[0]); // number of cols
    let ans = false;

    if (    row === n_rows - 1  &&  col === n_cols - 1        ) {
        // base case - have reached end
        ans = true;
    } else {
        // have not reached end yet
        const next = valid_next(row, col, maze);
        for (let i = 0; i < array_length(next); i = i + 1) {
         ans = ans || is_solvable(next[i][0], next[i][1], maze);
        }
    }
    return ans;
}
```

—— **END OF QUESTIONS** ——

Appendix

We assume the following pre-declared functions in Source §4 are declared as follows:

```
function map(f, xs) {
    return is_null(xs)
           ? xs
           : pair(f(head(xs)), map(f, tail(xs)));
}

function filter(pred, xs) {
    return is_null(xs)
           ? null
           : pred(head(xs))
           ? pair(head(xs), filter(pred, tail(xs)))
           : filter(pred, tail(xs));
}

function accumulate(f, initial, xs) {
    return is_null(xs)
           ? initial
           : f(head(xs), accumulate(f, initial, tail(xs)));
}

function append(xs, ys) {
    return is_null(xs)
           ? ys
           : pair(head(xs), append(tail(xs), ys));
}


function stream_map(f, s) {
    return is_null(s)
           ? null
           : pair(f(head(s)), () => stream_map(f, stream_tail(s)));
}

function stream_filter(p, s) {
    return is_null(s)
           ? null
           : p(head(s))
           ? pair(head(s), () => stream_filter(p, stream_tail(s)))
           : stream_filter(p, stream_tail(s));
}

function stream_ref(s, n) {
    return n === 0
           ? head(s)
           : stream_ref(stream_tail(s), n - 1);
}
```

```
function enum_stream(low, hi) {
    return low > hi
           ? null
           : pair(low, () => enum_stream(low + 1, hi));
}

function eval_stream(s, n) {
    return n === 0
           ? null
           : pair(head(s), eval_stream(stream_tail(s), n - 1));
}

function integers_from(n) {
    return pair(n, () => integers_from(n + 1));
}
```

—— END OF PAPER ——