NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

MIDTERM ASSESSMENT SOLUTION

Semester 1 AY2017/18

October 4, 2017 Time allowed: 1 hour 30 minutes

Student number: | | | | | | | | | |

Your Avenger's Name: |

Instructions (please read carefully):

- Write down your student number (the number on your student card) in the space above. DO NOT WRITE YOUR NAME ON THE QUESTION SET!
- Write down your Avenger's name in the box provided above.
- You are allowed to use one A4 sheet of notes (written or printed on both sides).
- This paper comprises 7 questions and **FOURTEEN (14) pages**, including this page. You are given **1 hour 30 minutes** to complete the assessment.
- The maximum score of this assessment is **55 marks**. The weight of each question is given in square brackets beside the question number.
- All questions must be answered correctly for the maximum score to be attained.
- All questions should be answered in the boxes provided in the answer sheet. If you run out of space in the boxes, clearly indicate where the answer is.
- The back-sides of the sheets and the pages marked "scratch paper" may be used as scratch paper.
- You are allowed to use pencils, ball-pens or fountain pens, as you like (no red color, please).
- Write legibly; **UNTIDINESS will be penalized.**

GOOD LUCK!

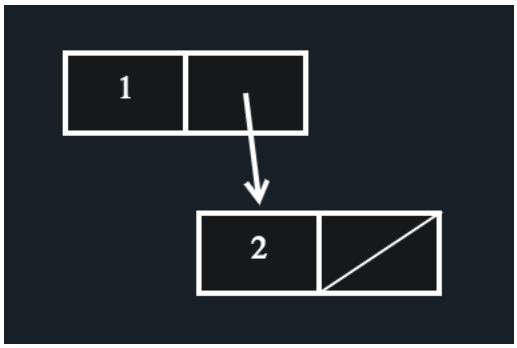
Q#	1	2	3	4	5	6	7	Σ
Max	16	5	12	5	5	7	5	55
Sc								

Question 1: Boxes and Pointers [16 marks]

A. [2 marks]

Assume that the variable `my_list` refers to the value
Draw the box-and-pointer diagram of the value resulting from the following program:

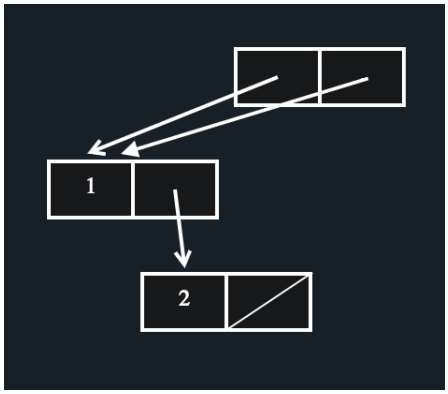
```
const my_list = pair(1, pair(2, null));
my_list;
```



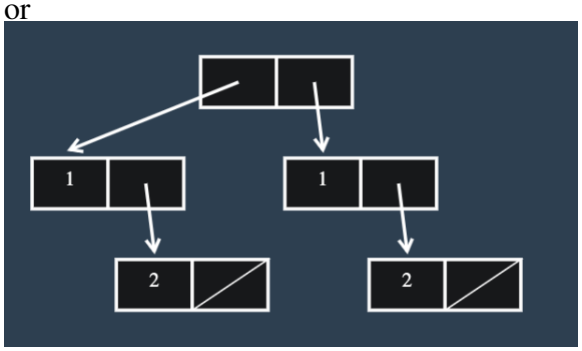
B. [2 marks]

Draw the box-and-pointer diagram of the value resulting from the following program:

```
const my_list = pair(1, pair(2, null));
pair(my_list, my_list);
```



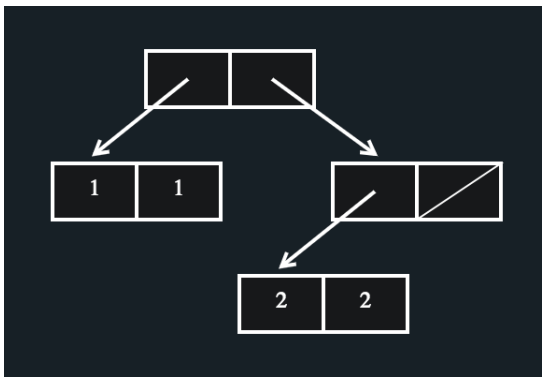
or



C. [3 marks]

Draw the box-and-pointer diagram of the value resulting from the following program:

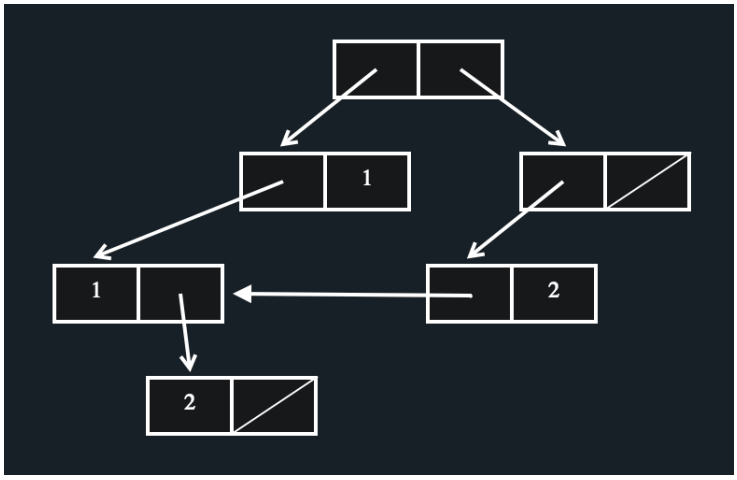
```
const my_list = pair(1, pair(2, null));
map(x => pair(x, x), my_list);
```



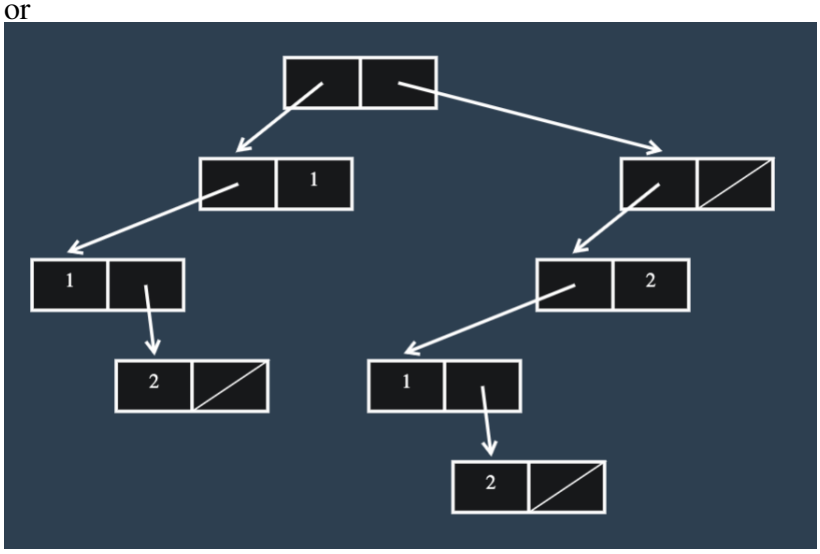
D. [3 marks]

Draw the box-and-pointer diagram of the value resulting from the following program:

```
const my_list = pair(1, pair(2, null));
map(x => pair(my_list, x), my_list);
```



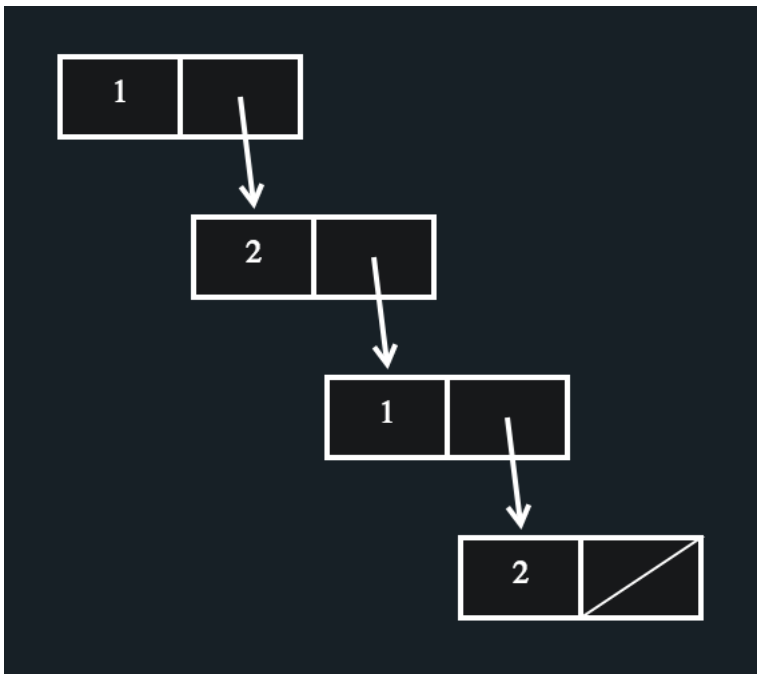
or



E. [3 marks]

Draw the box-and-pointer diagram of the value resulting from the following program:

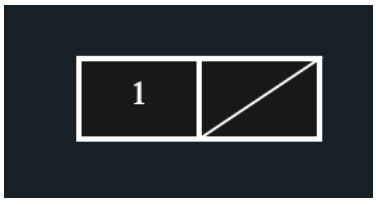
```
const my_list = pair(1, pair(2, null));
accumulate(pair, my_list, my_list);
```



F. [3 marks]

Draw the box-and-pointer diagram of the value resulting from the following program:

```
const my_list = pair(1, pair(2, null));
accumulate(member, reverse(my_list), my_list);
```



Question 2: Function with a Twist [5 marks]

Consider the following function `twist`:

```
function twist(f) {
    return (x, y) => f(y, x);
}
```

A. [2 marks]

What is the result of the following program?

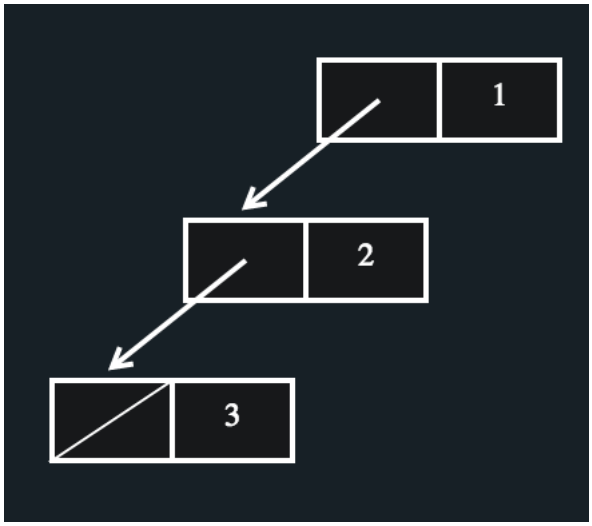
```
(twist(math_pow))(3, 4);
```

64

B. [3 marks]

Draw the box-and-pointer diagram of the result of the following program.

```
accumulate(twist(pair), null, list(1,2,3));
```



Question 3: Variations on an Empty List [12 marks]

This question is obsolete in 2020, due to changes in the module material.

Question 4: Improving Selection Sort [5 marks]

Recall the sorting function `selection_sort` from the lectures:

```
function selection_sort(xs) {
    if (is_null(xs)) {
        return xs;
    } else {
        const x = smallest(xs);
        return pair(x, selection_sort(remove(x, xs)));
    }
}
```

The function `smallest` finds the smallest element `x` of a given list `xs`, whereas the function `remove(x, xs)` returns a list with the same elements as `xs`, but with one less occurrence of `x`. Both of these functions run through the list, so it seems reasonable to try to combine them into one function, called `find_min`.

Specification: The function `find_min` is applied to a non-empty list `xs`. It returns a pair whose head is the smallest element `x` of `xs`, and whose tail is a list with the same elements as `xs`, but with one less occurrence of `x`.

With this function `find_min`, we can implement `selection_sort` easier, as follows:

```
function selection_sort(xs) {
    if (is_null(xs)) {
        return xs;
    } else {
        const xss = find_min(xs);
        return pair(head(xss), selection_sort(tail(xss)));
    }
}
```

Implement the function `find_min` such it **calls the `tail` function at most n times**, when it is applied to a list with n elements.

```
function find_min(xs) {
    function helper(ys, smallest_so_far, acc) {
        return (is_null(ys))
               ? pair(smallest_so_far, acc)
               : (smallest_so_far > head(ys))
                 ? helper(tail(ys), head(ys),
                          pair(smallest_so_far, acc))
                 : helper(tail(ys), smallest_so_far,
                          pair(head(ys), acc));
    }
    return helper(tail(xs), head(xs), null);
}
```

Question 5: Improving Merge Sort [5 marks]

Recall the `merge_sort` algorithm given in the lectures.

```
function merge_sort(xs) {
    if (is_null(xs) || is_null(tail(xs))) {
        return xs;
    } else {
        const mid = middle(length(xs));
        return merge(merge_sort(take(xs, mid)),
                     merge_sort(drop(xs, mid)));
    }
}
```

The two function `take` and `drop` have a lot in common, so it may be a good idea to combine them into one.
Specification: The function `take_drop` is applied to a non-empty list `xs` with n elements and a number `k`, where $0 < \texttt{k} < n$. The call `take_drop(xs, k)` returns a pair whose `head` is a list containing the first `k` elements of `xs` and whose `tail` is a list containing the remaining $n-$ `k` elements.
With such a function `take_drop`, we can simplify `merge_sort` as follows:

```
function merge_sort(xs) {
    if (is_null(xs) || is_null(tail(xs))) {
        return xs;
    } else {
        const td = take_drop(xs, middle(length(xs)));
        return merge(merge_sort(head(td)),
                     merge_sort(tail(td)));
    }
}
```

Implement the function `take_drop` such that `take_drop(xs, n)` **applies the `tail` function at most `n` times.**

```
function take_drop(xs, n) {
    function helper(ys, k, acc) {
         return (k === 0)
             ? pair(acc, ys)
             : helper(tail(ys), k - 1, pair(head(ys), acc));
    }
    return helper(xs, n, null);
}
```

Question 6: Rucer Puzzle [7 marks]

Professor Evis Rucer invented the following puzzle to challenge the Source Academy cadets.
A Rucer Puzzle consists of a sequence of at least two positive numbers. At any point in time, exactly one of the numbers is said to be *in focus*. When a number *n* is in focus, a legal step will move the focus *n* positions to the left or to the right.
For example, consider the following Rucer Puzzle:

```
3 5 8 4 2 7 1 6
```

If the second number (which is the number 5) is in focus, the only legal step will move the focus five numbers to the right. Thus, after the legal step the focus will be on the seventh number (which is the number 1).
If the fifth number (which is the number 2) is in focus, there are two possible legal steps. One legal step will move the focus two positions to the left (and thus to the number 8), and the other legal step will move the focus two positions to the right (and thus to the number 1).
If the third number (which is the number 8) is in focus, there are no legal steps.
In the beginning, the first number is in focus, and the puzzle is considered solved when the last number is in focus. Your goal is to decide if a given Rucer Puzzle is solvable within a given number of legal steps.
For example, the Rucer Puzzle above is solvable within 3 legal steps. In fact, it is solvable within 2 legal steps. In the first step the focus moves from the number 3 to the number 4, and the the second step, the focus moves from the number 4 to the number 6, which is already the last number.
As another example, the following puzzle is not solvable within 3 legal steps:

```
6 1 3 5 2 2 4 3
```

Note that this Rucer puzzle is solvable within 4 legal steps.
Prof Rucer challenges you to provide a function `solvable` that takes a list `xs` of at least two positive numbers and a positive number `n` as arguments. A call `solvable(xs, n)` returns true iff the Rucer Puzzle with the sequence `xs` is solvable within `n` steps.
For example, `solvable(list(3, 5, 8, 4, 2, 7, 1, 6), 3)` should return `true`, whereas `solvable(list(6, 1, 3, 5, 2, 2, 4, 3), 3)` should return `false`.

```
// check if Rucer Puzzle xs is solvable within k steps
function solvable(xs, k) {
    const n = length(xs);

    // check if Rucer Puzzle with focus on f-th number
    // is solvable within i steps
    function solvable_with_focus(i, f) {
        if (i < 0 || f < 1 || f > n) {
            return false;
        } else if (f === n) {
            return true;
        } else {
            // list_ref starts at 0
            const number_in_focus = list_ref(xs, f - 1);
            return solvable_with_focus(i - 1,
                                       f - number_in_focus)
                   ||
                   solvable_with_focus(i - 1,
                                       f + number_in_focus);
        }
    }
    return solvable_with_focus(k, 1);
}
```

Question 7: The Essence of Recursion [5 marks]

Most recursive processes in the lectures resulted from recursive functions, which are functions that call themselves in their body. As an example, consider:

```
function g(x) { return g(g(x)); }
g(0);
```

It is possible to achieve similar behaviour without recursive functions. As an example, consider:

```
(f => f(f))(f => f(f));
```

We can say the function

```
f => f(f)
```

captures *the essence of recursion*.
We shall use this kind of recursion to implement the divide-and-conquer ("wishful thinking") examples in the lectures, *without any recursive calls*. Consider the following (incomplete) Source program:

```
((f => f(f))       // the essence of recursion
 (make_factorial =>
     n => (n === 0) ? 1 : n * (


                make_factorial(make_factorial)


                                )(n - 1)
 )
)(5);
```

Here the essence of recursion is applied to a function that returns a kind of factorial function. Note that recursion applies f to itself, and thus returns a kind of factorial function, that we then apply to the number 5. However, there is a gap in the factorial function! Fill in the gap **without defining any additional function** such that this program returns the factorial of the given number, in this case 120.

— END OF PAPER —