CS1101S — Programming Methodology
School of Computing, National University of Singapore

Mid-Term Quiz
(adapted to Source 2021 in 9/2020)

October 2, 2013 **Time allowed:** 1 hour 40 minutes

Matriculation No:

Instructions (please read carefully):

1. Write down your matriculation number on the **question paper**. DO NOT WRITE YOUR NAME ON THE QUESTION SET!
2. This is **an open-sheet quiz**. You are allowed to bring one A4 sheet of notes (written or printed on both sides).
3. This paper comprises 7 questions and **TWENTY (20) pages**. The time allowed for solving this quiz is **1 hour 40 minutes**.
4. The maximum score of this quiz is **60 marks**. The weight of each question is given in square brackets beside the question number.
5. All questions must be answered correctly for the maximum score to be attained.
6. All questions must be answered in the space provided in the answer sheet; no extra sheets will be accepted as answers.
7. The back-sides of the sheets and the pages marked "scratch paper" in the question set may be used as scratch paper.
8. You are allowed to use pencils, ball-pens or fountain pens, as you like (no red color, please).

GOOD LUCK!

Q#	1	2	3	4	5	6	7	Σ
Max	2	6	5	11	8	6	22	60
Sc								

Question 1: Minimalistic Equality [2 marks]

Define a Source function `equal_boolean` that returns `true` if two given boolean values are equal and `false` otherwise. The twist: Only use the the following symbols

```
?   :   x   y   true
```

```
function equal_boolean(x, y) {

    return x ? y : y ? x : true

    ;
}
```

Question 2: Additional overload [6 marks]

In JavaScript (not Source), the operator + can be applied as follows

operator	argument 1	argument 2	result
+	number	number	number
+	string	any	string
+	any	string	string

In the first case, the two given numbers are added according to the IEEE 754 standard. In the other two cases, a type "any" means that any argument is allowed **and that the argument is converted into a string**. The two strings are then concatenated. Assume that you have the following functions given:

- `add_ieee_754(x,y)`: add two numbers `x` and `y`
- `concatenate(x,y)`: concatenate two strings `x` and `y`
- `is_number(x)`: returns `true` if the argument `x` is a number and `false` otherwise
- `is_string(x)`: returns `true` if the argument `x` is a string and `false` otherwise
- `to_string(x)`: converts the argument `x` into a string, regardless of the type of `x`.

Define a Source function `plus` using these functions and without using the operator + that meets the above specification. JavaScript is a bit "lenient" and also produces reasonable results in other cases. **However, your solution should return the string `"error: wrong types"` for argument combinations not listed above.**

```
function plus(x, y) {
    return is_number(x) && is_number(y)
        ? add_ieee_754(x, y)
        : is_string(x) || is_string(y)
          ? concatenate(to_string(x),
                        to_string(y))
          : "error: wrong types";
}
```

Question 3: Elementary, Watson? [5 marks]

In mathematics, the repeated power function, also called *tetration*, is defined as follows:

$$^{n}b := b^{b^{\cdot^{\cdot^{\cdot^{b}}}}}$$

where b appears n times in the power chain, and where n is a positive integer. Recall that x^{y^z} is read as $x^{(y^z)}$ and not as $(x^y)^z$. For example, $^{4}2 = 2^{2^{2^{2}}} = 2^{16} = 65536$.

[4 marks] Define a function `tetrate` such that `tetrate(b,n)` computes ^{n}b. You can use the function `power(b, e)` from the lectures that raises a number `b` to the power of `e`.

```
function tetrate(b, n) {
    return n === 1 ? b
        ? power(b, tetrate(b, n - 1));
}
```

[1 mark] What is the order of growth of the runtime of your function, in O (Big-Oh) notation, with respect to the arguments b and n, assuming that the order of growth of runtime of the function `power(b,e)` is $\Theta($`e`$)$? (no need for explanation)

In the final call of `power`, the exponent is `tetrate(b, n - 1)`. Since the runtime for `power(b,e)` has order of growth $\Theta($`e`$)$, we have a runtime of $\Theta(^{n-1}b)$ for the last call. For the previous $n-2$ calls of `power`, the exponent is smaller, thus we can safely say that the overall runtime is $O(n \times {}^{n-1}b)$.

Question 4: Entangled chains [11 marks]

Consider the following functions:

```
function plus_one(x) {
    return x + 1;
}
function twice(f) {
    return x => f(f(x));
}
function n_times(f,n) {
    return n === 1 ? f
        : x => f(n_times(f, n - 1)(x));
}
function chain(f,n) {
    return n === 1 ? f
        : chain(f, n - 1)(f);
}
```

What is the result of the following Source programs (no need for explanation):

A. [1 mark]

```
0;
```

0

B. [1 mark]

```
plus_one(0);
```

1

C. [1 mark]

```
plus_one(plus_one(0));
```

2

D. [1 mark]

```
twice(plus_one)(0);
```

$0+2=2$

E. [1 mark]

```
n_times(plus_one,4)(0);
```

$0+4=4$

F. [1 mark]

```
twice(twice)(plus_one)(0);
```

$2^2=4$

G. [1 mark]

```
n_times(twice,3)(plus_one)(0);
```

$2^3=8$

H. [1 mark]

This subquestion is quite difficult, and **each of the following is more difficult than the previous one**. Manage your time carefully.

```
twice(twice)(twice)(plus_one)(0);
```

$^{3}2 = 16$

I. [1 mark]

```
n_times(n_times(twice,2),3)(plus_one)(0);
```

$(2^2)^3 = 64$

J. [1 mark]

```
n_times(chain(twice,3),2)(plus_one)(0);
```

$(^{3}2)^2 = 16^2 = 256$

K. [1 mark]

```
chain(twice,4)(plus_one)(0);
```

$^{4}2 = 2^{16} = 65536$

Question 5: Zipping along [8 marks]

A. [5 marks]

Let us say we have two lists `xs` and `ys` of equal length given. We would like to define a function `make_pairs` that constructs a list of the same length as `xs` where each element is a pair of the corresponding elements of `xs` and `ys`. For example,

```
make_pairs(list(1,2,3), list(10,20,30));
```

yields the same list as

```
list(pair(1,10), pair(2,20), pair(3,30));
```

[4 marks] Give the function `make_pairs` in Source without using your solution to the next question.

```
function make_pairs(xs,ys) {
    return is_null(xs)  // is_null(ys) due to equal length
        ? null
        : pair(pair(head(xs), head(ys)),
                    make_pairs(tail(xs), tail(ys)));
}
```

[1 mark] What is the order of growth of the runtime of your function, in Θ notation, with respect to the size n of the given lists? (no need for explanation)

$\Theta(n)$

Does your function give rise to a recursive or iterative process? (no need for explanation)

recursive

B. [3 marks]

Generalize this concept by defining a function `zip` that applies a given function to the corresponding elements. For example,

```
zip( (x, y) => x * y,
    list(1,2,3),
    list(10,20,30));
```

should result in the same list as

```
list(1 * 10, 2 * 20, 3 * 30);
```

```
function zip(f, xs, ys) {
    return is_null(xs) // is_null(ys) due to equal length
        ? null
        : pair(f(head(xs), head(ys)),
               zip(f, tail(xs), tail(ys)));
}
```

Question 6: An accumulating filter [6 marks]

[3 marks] Use the function `accumulate` to define the function `filter`, both of which are described in the Appendix.

```
function filter(pred, xs) {
    return accumulate( (x,acc) => pred(x) ? pair(x,acc) : acc,
                       null,
                       xs);
}
```

[3 marks] Most likely, your solution will give rise to a recursive process. Briefly describe how you can implement the function `filter` such that it gives rise to an iterative process.

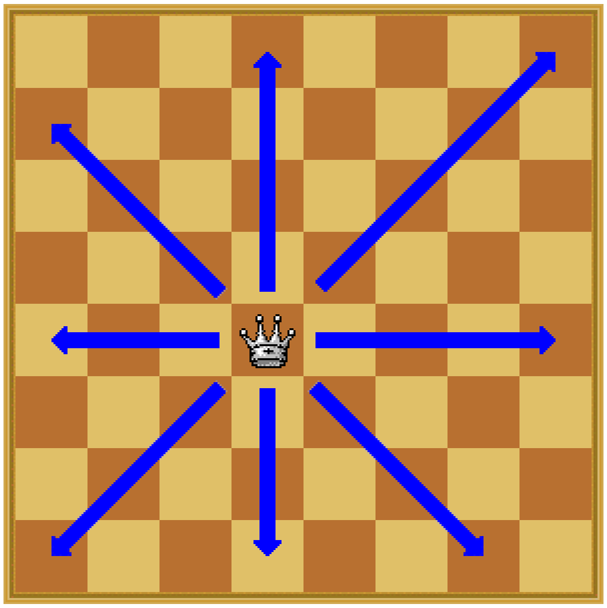
This is done using a function `fold_left` similar to the solution above (described in the recitations), followed by a reverse operation.

```
function fold_left(op, initial, sequence) {
    function iter(result, rest) {
        return is_null(rest)
            ? result
            : iter(op(result, head(rest)), tail(rest));
    }
    return iter(initial, sequence);
}
```

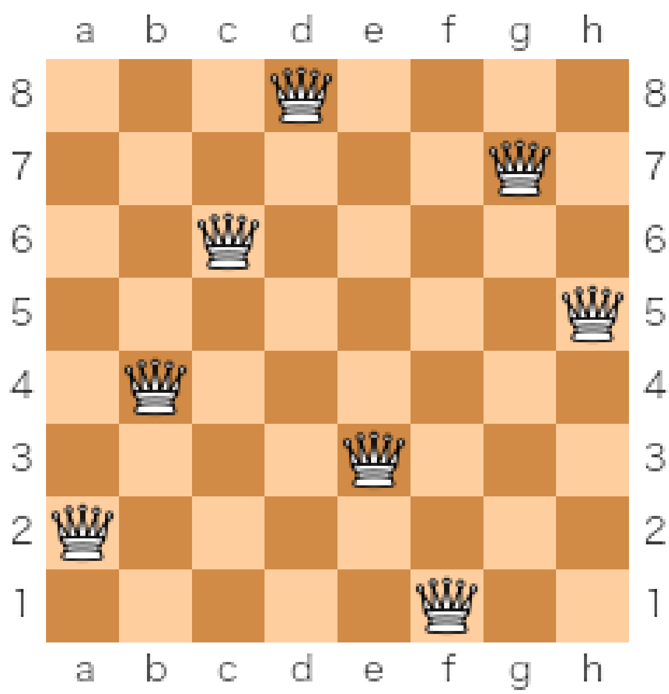
Question 7: Attack of the queens [22 marks]

The n-queens puzzle was devised in 1850 by Franz Nauck. Its task is to place n queens on an $n \times n$ chessboard so that no two queens attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

The following figure illustrates the ways a chess queen can move on an $n \times n$ board: up and down its column, left and right in its row and 45^o diagonally.



A solution to the 8-queens puzzle is given in the following figure.



A. [2 marks]

In order to represent the row and column position of a queen, define a constructor `make_queen` and accessor functions `row` and `column` that work as follows:

```
const q = make_queen(3, 4);
const r = row(q);            // r now refers to 3
const c = column(q);         // c now refers to 4
```

```
const make_queen = pair;
const row = head;
const column = tail;
```

B. [3 marks]

Since the *n*-queens puzzle is about placing queens without attack, we first find out whether two queens can attack each other diagonally. (To prevent horizontal and vertical attacks we shall use different means later.) Define a Source function `attack_each_other_diagonally` that returns `true` if queen `q1` can attack `q2` using a diagonal move and `false` otherwise.

```
function attack_each_other_diagonally(q1, q2) {
    const column_difference = math_abs(column(q1) - column(q2));
    const row_difference    = math_abs(row(q1)
- row(q2)   );
    return column_difference === row_difference;
}
```

C. [4 marks]

It will be useful to have a function that checks whether a given queen `q1` can attack any queen of a given list of queens diagonally. For example,

```
attack_any_diagonally(make_queen(1,1),
    list(make_queen(2,3), make_queen(3,3), make_queen(4,2)));
```

should return `true`, because a queen at position $(1,1)$ can attack a queen at position $(3,3)$ using a diagonal move.

[3 marks] You earn 3 marks for the program, if you use the function `accumulate` in a correct solution, and 2 marks for any other correct solution.

```
function attack_any_diagonally(q1, qs) {
    return accumulate( (q2, acc)
                        => attack_each_other_diagonally(q1, q2)
                           || acc,
                       false,
                       qs);
}
```

[1 mark] What is the order of growth of the runtime of your function, in Θ notation, with respect to the size n of the given list `qs`? (no need for explanation)

The function `accumulate` performs n calls of `attack_each_other_diagonally`, the runtime for each of which has order of growth $\Theta(1)$. Thus overall we have $\Theta(n)$.

D. [4 marks]

We shall follow the principle of "generate and test" in our solution: We first generate a comprehensive set of solution candidates, and then test each of them if any queen can attack any other queen diagonally.

[3 marks] For this test, use the function `attack_any_diagonally` from the previous question in order to define a function `attack_diagonally` that returns `true` if *any two* queens of a given list of queens `qs` can attack each other diagonally, and `false` otherwise.

```
function attack_diagonally(qs) {
    return is_null(qs) ? false
        : attack_any_diagonally(head(qs), tail(qs))
          || attack_diagonally(tail(qs));
}
```

[1 mark] What is the order of growth of the runtime of your function, in Θ notation, with respect to the size n of the given list `qs`? (no need for explanation)

The runtime for the function `attack_any_diagonally` has order of growth $\Theta(n)$ because it uses the function `attack_each_other_diagonally` n times, and the runtime for `attack_each_other_diagonally` has order of growth $\Theta(1)$. The function `attack_diagonally` calls `attack_any_diagonally` n times, with sizes ranging from $n-1$ to 1. The result is a quadratic runtime: $\Theta(n^2)$

E. [6 marks]

Assume given a function `permutations(xs)` that returns a list of all permutations of the elements in `xs`. For example, `permutations(list(1,2,3))` returns the same list as

```
list(list(1,2,3),list(2,1,3),list(2,3,1),
     list(1,3,2),list(3,1,2),list(3,2,1));
```

Our strategy starts with producing the list of all permutations of the numbers from 1 to n. For a particular permutation p, we then generate a solution candidate by placing one queen in each column. The row for the i-th queen is indicated by the i-th element of p. Since we place at most one queen in each column, we can be sure that there are no vertical attacks, and since no two numbers appear twice in a permutation, we can be sure that there are no horizontal attacks.

[5 marks] Define a function `queens` that produces all possible ways to place one queen in each column and in different rows on a $n \times n$ chessboard. The function `queens` should return a list of lists of queens. For example, `queens(3)` should return the same list of lists of queens as

```
list(list(make_queen(1,1),make_queen(2,2),make_queen(3,3)),
     list(make_queen(1,2),make_queen(2,1),make_queen(3,3)),
     list(make_queen(1,2),make_queen(2,3),make_queen(3,1)),
     list(make_queen(1,1),make_queen(2,3),make_queen(3,2)),
     list(make_queen(1,3),make_queen(2,1),make_queen(3,2)),
     list(make_queen(1,3),make_queen(2,2),make_queen(3,1))
    );
```

The order within the lists and between the lists may be different than the one given here. If you use the functions `map` and `zip` (from the previous question) correctly, you earn 5 marks for your program. You earn 3 marks for any other correct program.

```
function queens(n) {
    const ordered = enum_list(1,n);
    const perms = permutations(ordered);
    return map(perm => zip(make_queen,
                           ordered,
                           perm),
               perms);
}
```

[1 mark] What is the order of growth of the runtime of your function, in Θ notation, with respect to **the number of permutations** q of lists of size `n`? You can assume that the runtime of the function `permutations(xs)` has order of growth $\Theta(q)$. (no need for explanation)

> The runtime in terms of q and n has order of growth $\Theta(q \times n)$. Various other explanations are admitted here, since the question is a bit vague.

F. [3 marks]

To put it all together, write a function `solutions` that produces a comprehensive set of solution candidates, using `queens`, and that filters them such that only those candidates are kept in which no diagonal attack occurs. You get 3 marks if you use the function `filter` correctly and 2 marks for any other correct solution.

```
function solutions(n) {
    return filter(qs => ! attack_diagonally(qs),
                  queens(n)
                 );
}
```

Appendix

List Support

Source Week 6 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)`: Can only be applied to the empty list or a pair. Returns `true` if `xs` is the empty list, and `false` if `xs` is a pair.
- `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`.

- `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 `f` takes constant time.

Miscellaneous Functions

- `is_number(x)`: Returns `true` if `x` is a number, and `false` otherwise.

Scratch paper: Tear off, if needed

Scratch Paper: Tear off, if needed

— E N D O F P A P E R —