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

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

October 3, 2012 **Time allowed:** 1 hour 45 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 6 questions and **TWENTY (20) pages**. The time allowed for solving this quiz is **1 hour 45 minutes**.
4. The maximum score of this quiz is **88 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	Σ
Max	17	5	8	7	14	37	88
Sc							

Question 1: Minimal Conditionality [17 marks]

In the spirit of minimality (remember the lambda calculus) we could define conditional expressions in Source through if-statements.
Specifically, we could "translate" every expression of the form

```
( A ) ? B : C
```

where `A`, `B`, and `C` are Source expressions, to a version that applies a function `conditional` as follows:

```
conditional( A, () => B, () => C)
```

A. [4 marks]

Give a definition of the function `conditional` that uses an if-statement but not `...?...:...` such that the two versions behave exactly the same.

```
function conditional(a, b, c) {

    if (a) {
        return b();
    } else {
        return c();
    }

}
```

B. [4 marks]

With the same translation scheme, consider the following implementation of `conditional` that does not even use if-statements:

```
function conditional(a, b, c) {
    return a(b, c);
}
```

Note that this implementation would not work if we use Source's boolean values `true` and `false` as condition expressions before the "?" symbol. Give definitions of `True` and `False` such that `conditional(True, 1, 2)` returns 1 and `conditional(False, 1, 2)` returns 2.

```
const True = (x, y) => x;
```

```
const False = (x, y) => y;
```

C. [4 marks]

Alternatively, we could consider an even simpler translation of

`( A ) ? B : C`

namely to

`simple_conditional( A , B, C)`

Give a definition of the function `simple_conditional`, also without using `...?...:...`, such that the two versions produce the same results when `B` and `C` are numbers.

```
function simple_conditional(a, b, c) {

    if (a) {
        return b;
    } else {
        return c;
    }

}
```

D. [5 marks]

What is the problem with the translation using `simple_conditional`? Answer using complete English sentences, and illustrate your answer using example expressions `A`, `B`, `C` that behave differently in the two cases.

The problem is that both expressions `B` and `C` get evaluated.

`(true) ? display("yes") : display("no")`

Here both `"yes"` and `"no"` will be displayed.

Question 2: Mystery with a difference [5 marks]

Consider the following function:

```
function mystery(x) {
    return y => z => y(z)(x);
}
```

Define `diff` such that `mystery(a)(diff)(b)` returns the difference between `a` and `b`.
Example: `mystery(21)(diff)(17)` should return 4.

```
const diff = z => x => x - z;
```

Question 3: Two Famous Composers [8 marks]

In this question, we shall use the following two functions in examples:

```
function square(x) {
    return x * x;
}
function add_one(x) {
    return x + 1;
}
```

Consider the following two ways of composing functions:

```
function compose1(f, g) {
    return x => f(g(x));
}
function compose2(f, g) {
    return f(g);
}
```

A. [3 marks]

Write a Source expression that uses `square` **and** `add_one` **and** `compose1` in order to compute the square of the result of adding one to 7.

```
compose1(square,add_one)(7);
```

B. [5 marks]

Write a Source expression that uses `square` **and** `add_one` **and** `compose2` in order to compute the square of the result of adding one to 7.

```
compose2(square,add_one(7));
```

Question 4: Heads or tails? [7 marks]

We find it often convenient to use list discipline when processing lists. Lists are defined as follows:

> A *list* of a certain type is either the empty list `null` or a pair whose head is of that type and whose tail is a list of that type.

There is no particular reason why the head contains the data item and the tail the rest of the list an not vice versa. If we turn the definition around, we get the following:

> A *tsil* of a certain type is either the empty list `null` or a pair whose *tail* is of that type and whose *head* is a tsil of that type.

All our list processing functions can be converted to tsil processing functions. For example, the function `map` becomes:

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

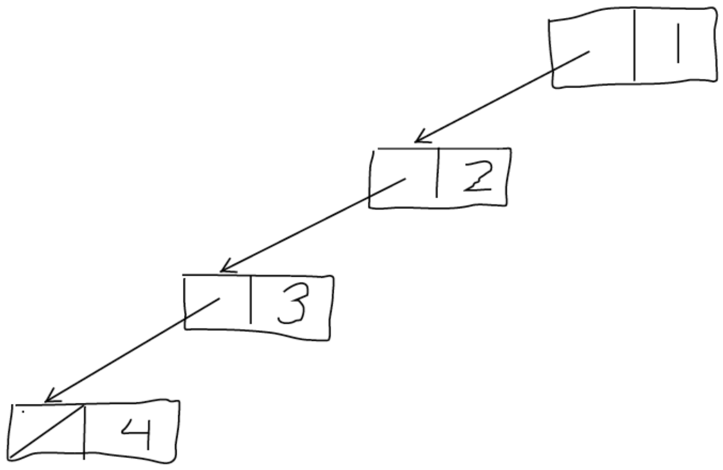
A. [4 marks]

Write a function `list_to_tsil` that translates a given list to a tsil. The order of elements should not change, in a sense that the first element of given list should be accessible in $O(1)$ in the tsil, and the last element of a given list will require $O(n)$ access time in the tsil, where n is the length of the list. Thus, `tail(list_to_tsil(list(1,2,3)))` should evaluate to 1.

```
function list_to_tsil(xs) {
    if (is_null(xs)) {
        return null;
    } else {
        return pair(list_to_tsil(tail(xs)), head(xs));
    }
}
```

B. [3 marks]

Draw the box-and-pointer diagram for `list_to_tsil(list(1,2,3,4))`.



Question 5: Multiple maps [14 marks]

(this subquestion is not relevant any longer in 2020, due to changes in the course material)

A. [4 marks]

Consider the following function `sum_of_list`:

```
function sum_of_list(xs) {
    if (is_null(xs)) {
        return 0;
    } else {
        return head(xs) + sum_of_list(tail(xs));
    }
}
```

This function results in a recursive process. Give a version of `sum_of_list` that results in an iterative process.

```
function sum_of_list(xs) {
    function sum(still_to_process, sum_so_far) {
        if (is_null(still_to_process)) {
            return sum_so_far;
        } else {
            return sum(tail(still_to_process),
                           sum_so_far + head(still_to_process));
        }
    }
    return sum_of_list(xs, 0);
}
```

B. [6 marks]

A limitation of our version of `map` is that it can be applied to only one argument list. What if we want to apply a function element-wise to a number of given lists? More specifically, we want a function `multi_map` that can be applied to a function `f` and a list of lists `xss`. The function `multi_map` will apply `f` first to the list of all first elements of the lists in `xss`, resulting in the first element of the result, then to the list of all second elements, resulting in the second element of the result, etc. We assume that all lists in `xss` have the same length, which is also the length of the result.
Example:

```
multi_map(sum_of_list, list(list(1, 2, 3),
                            list(4, 5, 6),
                            list(7, 8, 9)));
```

returns the list with the three elements 12, 15 and 18. Give an implementation of `multi_map` in Source.

```
function multi_map(f, xss) {
    if (is_null(head(xss))) {
        return null;
    } else {
        return pair( f(map(head, xss)),
                     multi_map(f, map(tail, xss)) );
    }
}
```

Question 6: Sudoku Checker [37 marks]

Sudoku is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids, called *boxes* that compose the grid, contains all of the digits from 1 to 9. Here is an example of a solution to a Sudoku puzzle.

9	5	4	1	6	2	3	8	7
2	6	8	7	4	3	1	5	9
3	7	1	5	9	8	2	6	4
7	9	3	6	8	4	5	1	2
5	8	6	2	1	7	4	9	3
4	1	2	3	5	9	6	7	8
6	3	9	4	7	5	8	2	1
1	2	7	8	3	6	9	4	5
8	4	5	9	2	1	7	3	6

A Sudoku grid can be represented in Source as a list of rows, as follows:

```
const solution = list(list(9, 5, 4, 1, 6, 2, 3, 8, 7)
                      list(2, 6, 8, 7, 4, 3, 1, 5, 9)
                      list(3, 7, 1, 5, 9, 8, 2, 6, 4)
                      list(7, 9, 3, 6, 8, 4, 5, 1, 2)
                      list(5, 8, 6, 2, 1, 7, 4, 9, 3)
                      list(4, 1, 2, 3, 5, 9, 6, 7, 8)
                      list(6, 3, 9, 4, 7, 5, 8, 2, 1)
                      list(1, 2, 7, 8, 3, 6, 9, 4, 5)
                      list(8, 4, 5, 9, 2, 1, 7, 3, 6));
```

Your task in this question will be to write a function `test_sudoku(grid)` which returns `true` if and only if the given `grid` is a solution to the Sudoku puzzle.

We shall distinguish the following concepts:

Cell: A particular slot of the grid. A Sudoku grid has 81 cells.

Box: A 3×3 sub-grid as shown in bold in the example above.

Coordinates: x- and y-coordinates (row and column) of a given cell, where we start counting at 0. The cell with the coordinates 0 and 4 in the grid above contains the number 6.

Coordinates list: A list of coordinates, each specifying a cell.

List of coordinates list: A list, each element of which is a coordinates list.

Our strategy will be to build a list of coordinates lists, each specifying a particular set of cell addresses, all whose entries need to be different.

A. [3 marks]

We need to represent the coordinates of a given cell. For this, specify a constructor `make_coordinates` and two access functions `get_x` and `get_y`. Define these functions in the space below.

```
function make_coordinates(row, column) {
    return pair(row, column);
}
function get_x(coordinates) {
    return head(coordinates);
}
function get_y(coordinates) {
    return tail(coordinates);
}
```

B. [4 marks]

Use these three functions and any functions in Source §2 to write a function `access(coordinates, grid)` that returns the value in the grid cell at the x- and y-coordinates of row and column, each starting at 0. Example:

`access(make_coordinates(0, 4), solution)`

should return 6.

```
function access(coordinates, grid) {
    return list_ref(list_ref(grid,
                             get_x(coordinates)),
                             get_y(coordinates));
}
```

C. [3 marks]

The function `access` of the previous question can be applied to a grid with *n* rows and columns, not just 9. Give the order of growth for the *runtime* your solution as *n* grows, using "big Theta" notation:

runtime(n) has order of growth $\Theta(\ n\ )$.

D. [5 marks]

Write a function `all_different(xs)` that returns `true` if and only if all elements of the list `xs` are different.

Hint: Remember that the function `member(x, xs)` returns an empty list if and only if `x` does not occur in `xs`.

```
function all_different(xs) {
    if (is_null(xs)) {
        return true;
    } else {
        return is_null(member(head(xs), tail(xs)))
                &&
                all_different(tail(xs));
    }
}
```

E. [3 marks]

The function `all_different` of the previous question can be applied to lists of length *n*, not just 9. Give the order of growth for the *runtime* your solution as *n* grows, using "big Theta" notation:

runtime(n) has order of growth $\Theta(\ n^2\ )$.

F. [5 marks]

For each row, we need to generate a list of lists of coordinates of the entries of that row. This is the job of the function `make_row_coordinates_list(row)`. For example, `make_row_coordinates_list(4)` should return the list of coordinates (4,0), (4,1), ... (4,8).

Implement the function `make_row_coordinate_list` using the space below.

```
function make_row_coordinates_list(row) {
    return build_list(9, col => make_coordinates(row,col);
}
```

G. [5 marks]

Write a function `test_coordinates_list(grid,coordinates_list)` that makes sure that the entries in the `grid` with the given `coordinates_list` are all different.

Example:

```
test_coordinates_list(solution, make_row_coordinates_list(0))
```

should return `true` because all entries in the first row of the given grid are different.

```
function test_coordinates_list(grid,coordinates_list) {
    const xs = map(coordinates => access(coordinates, grid),
                   coordinates_list);
    return all_different(xs);
}
```

H. [6 marks]

Assume that we have a function `make_col_coordinates_list` similar to `make_row_coordinates_list`.
For the boxes of the grid, we can use the following function:

```
function make_box_coordinates_list(row_1, row_2, col_1, col_2) {
    const rows = build_list(row_2 - row_1 + 1,
                            x => x + row_1);
    const coordinates_list_list_list =
        map(row =>
            build_list(col_2 - col_1 + 1,
                       y => make_coordinates(row, y + col_1)),
            rows);
    return accumulate(append,null,coordinates_list_list_list);
}
```

With that, we can generate the list of coordinates lists to be checked, as follows:

```
function make_sudoku_coordinates_list_list() {
    const row_coordinates_list_list =
    build_list(9, row => make_row_coordinates_list(row));
    const col_coordinates_list_list =
    build_list(9, col => make_col_coordinates_list(col));
    const box_coordinates_list_list =
    list(make_box_coordinates_list(0,2,0,2),
         make_box_coordinates_list(0,2,3,5),
         make_box_coordinates_list(0,2,6,8),
         make_box_coordinates_list(3,5,0,2),
         make_box_coordinates_list(3,5,3,5),
         make_box_coordinates_list(3,5,6,8),
         make_box_coordinates_list(6,8,0,2),
         make_box_coordinates_list(6,8,3,5),
         make_box_coordinates_list(6,8,6,8));
    return append(row_coordinates_list_list,
                  append(col_coordinates_list_list,
                         box_coordinates_list_list));
}
```

Putting it all together, write a function `test_sudoku(grid)` which checks if a given `grid` contains a solution to a Sudoku puzzle.

```
function test_sudoku(grid) {
    return accumulate(
               (coordinates_list, sofar) =>
                   sofar &&
                   test_coordinates_list(grid,
                                         coordinates_list),
               true,
               make_sudoku_coordinates_list_list());
}
```

I. [3 marks]

The function `test_sudoku` of the previous question can be applied to any Sudoku grid, not just the given `solution`. Give the order of growth for the *runtime* your function.

This question does not make sense. All Sudoku grids have the same size. There is no parameter that grows and thus there is no order of growth to speak of.

Appendix

List Support

Source §2 supports the following list 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.
- `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`.
- `map(f, xs)`: Returns a list that results from list `xs` by element-wise application of `f`.
- `build_list(n, f)`: Makes a list with `n` elements by applying the unary function `f` to the numbers 0 to `n - 1`.
- `for_each(f, xs)`: Applies `f` to every element of the list `xs`, and then returns `true`.
- `list_to_string(xs)`: Returns a string that represents list `xs` using the `[...]` notation.
- `reverse(xs)`: Returns list `xs` in reverse order.
- `append(xs, ys)`: Returns a list that results from appending the list `ys` to the list `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.
- `remove(x, xs)`: Returns a list that results from `xs` by removing the first item from `xs` that is identical (===) to `x`.
- `removeAll(x, xs)`: Returns a list that results from `xs` by removing all items from `xs` that are identical (===) to `x`.
- `filter(pred, xs)`: Returns a list that contains only those elements for which the one-argument function `pred` returns `true`.
- `enum_list(start, end)`: Returns a list that enumerates numbers starting from `start` using a step size of 1, until the number exceeds (>) `end`.
- `list_ref(xs, n)`: Returns the element of list `xs` at position `n`, where the first element has index 0.

- `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)))`.

Miscellaneous Functions

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

Scratch Paper

— E N D   O F   P A P E R —