NATIONAL UNIVERSITY OF SINGAPORE

**CS1101S — PROGRAMMING METHODOLOGY**

**CURATED VERSION OF 14/11/2020 (CORRECTED ON 22/11/2020)**

(Semester 1 AY2017/2018)

Time Allowed: **2 Hours**

---

**INSTRUCTIONS TO STUDENTS**

1. This assessment paper contains **FIVE (5)** questions and comprises **TWENTY-FIVE (25)** printed pages, including this page and two sheets of scratch paper at the end.
2. The full score of this paper is **80 marks**.
3. This is a **CLOSED BOOK** assessment, but you are allowed to use **ONE** A4 sheet of notes.
4. Answer **ALL** questions **within the space provided** in this booklet.
5. Where programs are required, write them in the **Source Week 11** language.
6. Write legibly with a **pen or pencil**. **UNTIDINESS will be penalized**.
7. Do not tear off any pages from this booklet.
8. Write your **Student Number** below **USING A PEN**. Do not write your name.

**Student No.:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |

---

This portion is for examiner's use only

| Question | Marks | Question | Marks |
|---|---|---|---|
| **Q1 (17 marks)** | | **Q4 (14 marks)** | |
| **Q2 (18 marks)** | | **Q5 (14 marks)** | |
| **Q3 (17 marks)** | | **TOTAL (80 marks)** | |

# Question 1: Pairs, lists and trees [17 marks]

Recall from the lectures the definition of trees:

> *A tree of certain data items is a list whose elements are such data items, or trees of such data items.*

In this question, we restrict ourselves to *trees of numbers*. A tree of numbers thus is a list whose elements are numbers or trees of numbers.

## A. Box-and-pointer diagrams of trees of numbers [6 marks]

Draw the box-and-pointer diagrams of results of evaluating the following Source statements.

### A.1. [1 mark]

```
pair(4, null);
```

### A.2. [1 mark]

```
list(5);
```

**A.3. [2 marks]**

```
list(list(4, 5, 6), 2, 3, list(8, 9));
```

**A.4. [2 marks]**

```
pair(null, pair(null, pair(null, null)));
```

## B. A notion of similarity [4 marks]

Recall the function `equal` from the lectures. When applied to two trees of numbers, `equal` returns `true`, if and only if they have exactly the same structure (same pairs and empty lists) and the same numbers at their leaves.

In this question, we would like to be a bit more liberal, and define two trees to be *similar*, if they have the same structure and if the corresponding number leaves differ by at most 1.

**Examples:**

```
similar(list(4, list(5,6)),
        list(4,null,list(5,6))); // false (not same structure)

similar(list(4, null, list(5,6)),
        list(5, null, list(4,7))); // true

similar(list(4, list(5,6)),
        list(5, list(3,7)));     // false (too big difference)
```

Define the function `similar` such that `similar(tn1, tn2)` returns `true` if `tn1` and `tn2` are similar and `false` otherwise. You can assume that both `tn1` and `tn2` are trees of numbers. Make your function as simple and clear as possible. You can assume that the function `is_number` is available, with the obvious meaning.

## C. Counting differences [4 marks]

For two similar trees, the question arises at how many places they differ. Write a function `differences` such that `differences(tn1, tn2)` returns the number of places where `tn1` and `tn2` have different numbers. You can assume that `tn1` and `tn2` are *similar* trees of numbers.

**Examples:**

```
differences(list(4, null, list(4,6), 8),
            list(5, null, list(4,7), 8)); // returns 2
differences(list(4,5,6,7),
            list(4,5,6,7));               // returns 0
```

Make your function as simple and clear as possible. You can assume that the function `is_number` is available, with the obvious meaning.

## D. Incrementing trees of numbers [3 marks]

Recall the function `map_tree` from the lectures.

```
function map_tree(f, tree) {
    return map(sub_tree => ! is_list(sub_tree)
                           ? f(sub_tree)
                           : map_tree(f, sub_tree),
               tree
              );
}
```

Use `map_tree` to define a function increment such that `increment(tn)` returns a tree that is *similar* to the given tree `tn`, where each number is incremented by 1. You can assume that `tn` is a tree of numbers.

**Example:**

```
increment(list(4, null, list(4,6), 8)); // returns the same as

          list(5, null, list(5,7), 9);
```

## The following question is not relevant for CS1101S as of 2019/20.

## ~~Question 2: Key-value stores [18 marks]~~

~~Key-value stores are databases that use lists of strings as keys and strings as values. Today, databases that use key-value stores such as Oracle's NoSQL are popular for Big Data applications.~~

### ~~A. Drawing stores [6 marks]~~

~~We shall use literal Source objects in order to implement key-value stores. For example, the following Source object can be seen as a key-value store.~~

```
var my_key_value_store =
{"NUS": {"students": {"Chris" : {"mobile": "98765432",
                                 "home"  : "65432109" },
                      "Sam"   : {"mobile": "44556677" } },
         "staff"    : {"Peter" : {"office": "65166632" } } },
 "CERN":              {"Tim"   : {"office": "59843726" } }    };
```

~~Draw the environment diagram of the environment after evaluating the program above.~~

## ~~B. Stores and the prototype chain [4 marks]~~

~~The string "__proto__" enjoys special treatment in Source (and JavaScript) objects. Consider the following Source object.~~

```
var your_key_value_store =
 { "a"           : { "b"           : { "c" : "1", "e": "2" },
                     "__proto__"   : { "f" : "3", "g": "4" } },
   "__proto__"   : { "h"           : { "i" : "5", "j": "6" } } };
```

~~What is the result of the following operations:~~

### ~~B.1. [1 mark]~~

```
your_key_value_store.a.b.c;
```

### ~~B.2. [1 mark]~~

```
your_key_value_store.a.f;
```

### ~~B.3. [1 mark]~~

```
your_key_value_store.h.b;
```

### ~~B.4. [1 mark]~~

```
your_key_value_store.h.j;
```

**~~We shall assume henceforth that "__proto__" does not occur in keys of key-value stores.~~**

## ~~C. Storing values [4 marks]~~

~~A key-value store is initially empty, and thus is represented by the empty object.~~

```
function make_empty_key_value_store() { return {}; }
```

~~In order to add entries to the key-value store, we require a function `put` that takes a *store* as first argument, followed by a *key*, which is a list of strings, followed by the *new value* that we want to enter. The string elements of the key are used to successively access the properties of the respective object, starting with the first element (also called the major key) and the whole store. The process will create objects whenever necessary, on the path to where the value belongs.~~

~~After the following session, `my_key_value_store_2` will refer to an object with the same structure and values as the object `my_key_value_store` in Part **A**.~~

```
var my_key_value_store_2 = make_empty_key_value_store();
put(my_key_value_store_2,
    list("NUS", "students", "Chris", "mobile"), "98765432");
put(my_key_value_store_2,
    list("NUS", "students", "Chris", "home"  ), "65432109");
put(my_key_value_store_2,
    list("NUS", "students", "Sam",   "mobile"), "44556677");
put(my_key_value_store_2,
    list("NUS", "staff",    "Peter", "office"), "65166632");
put(my_key_value_store_2,
    list("CERN",            "Tim",   "office"), "59843726");
```

~~Define the function `put`. The call `put(store, key, new_value)` should always succeed in writing the string `new_value` to any `store`, such that it can later be retrieved with the list of strings given in `key`. If the `store` already has a value (which might even be an object and not a string) under the given `key`, that value is replaced by `new_value`.~~

## ~~D. Retrieving values [4 marks]~~

~~In order to access a key-value-store, we need a function `get` that is applied to a store and a key. The function `get(store, key)` should always succeed for a given store `my_store` and a given list of strings `key`. If `store` contains the value `val` under `key`, the function `get` should return `val`. If there is **no string value** under `key`, the function `get` should return the value `undefined`. You can assume there are functions `is_string` and `is_object` to check if a given value is a string or an object, respectively.~~

~~**Examples:**~~

```
get(my_key_value_store_2,
    list("NUS", "students", "Chris", "mobile")); // "98765432"
get(my_key_value_store_2,
    list("NUS", "students"));                    // undefined
get(my_key_value_store_2,
    list("SFU", "students", "Chris", "mobile")); // undefined
```

# Question 3: Counting Sort [17 marks]

The sorting functions discussed in the lectures have in common that they use comparisons to determine the order of the elements in the resulting sorted array. The worst-case runtime for any such comparison-based sorting algorithms has order or growth $\Omega(n \log n)$, where $n$ is the size of the array to be sorted, if no additional information is known about the elements in the array or their order.

One way to do better than $\Omega(n \log n)$ is to use operations other than comparisons. For example, if we know that all elements of a given array are *integers* ranging from 0 to a relatively small maximal value, we could use the integers as *indices* into an array, and use array assignment and access as operations:

```
my_array[unsorted_array[i]] = …;
```

In this question, we are given an unsorted array of numbers `unsorted`, and a number `max`, with the additional information that all elements are integers between 0 and `max`.

Possible values for `unsorted` and `max` could be:

```
var unsorted = [5, 1, 10, 2, 1, 5, 7, 3];
var max = 12;
```

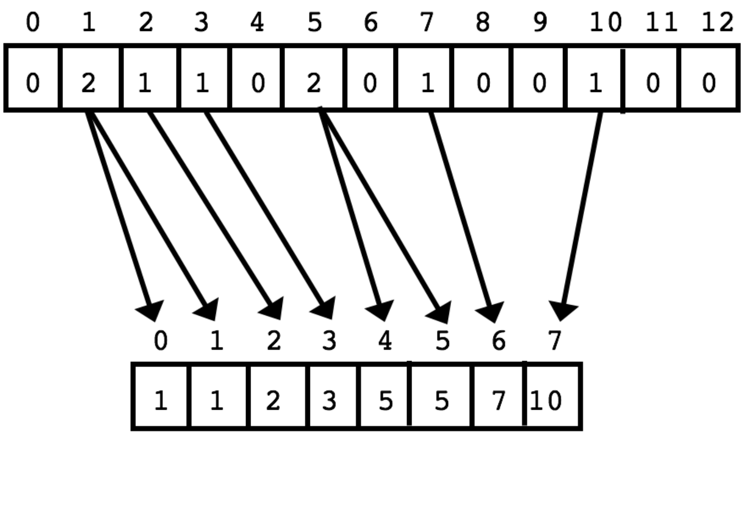
Our strategy for sorting such arrays is to build an array that represents a *histogram* for the values of the original array. The histogram indicates for each possible value 0 to `max`, how many times the value occurs in `unsorted`. In the example above, the histogram would look as follows:

```
var histogram = [0, 2, 1, 1, 0, 2, 0, 1, 0, 0, 1, 0, 0];
```

From such a histogram, we shall then generate the sorted array by placing as many copies of the histogram indices into the target array as indicated by the histogram:

```
var sorted = [1, 1, 2, 3, 5, 5, 7, 10];
```

The diagram below illustrates how the sorted array can be obtained from the histogram.



## A. Making arrays of zeros [3 marks]

It will be convenient to start with a histogram whose elements are all 0. Write a function `array_with_zeros` such that a call `array_with_zeros(n)` returns an array of length `n` whose values at indices 0 to `n - 1` are all 0.

For example,

```
array_with_zeros(5);
```

should return the array `[0, 0, 0, 0, 0]`.

## B. Making a histogram [4 marks]

Write a function `make_histogram` that takes an array of non-negative integers and a number `max` as arguments. You can assume that all integers in the array are less than or equal to `max`. The call `make_histogram(arr, max)` should return an array of length `max` + 1 that indicates for each number 0 to `max` the number of times the element occurs in `arr`. For example, the resulting histogram for the array `unsorted` on Page 11 and `max = 12` is the array:

```
[0, 2, 1, 1, 0, 2, 0, 1, 0, 0, 1, 0, 0]
```

Define your function as simply and clearly as possible, and make use of the function `array_with_zeros` in Part **A** if you think it helps.

## C. Entering copies [3 marks]

To fill the result array, we may need a function `enter_copies` such that `enter_copies(arr, n, v, start)` enters into `arr` exactly `n` copies of a given value `v`, starting at index `start`. For example, consider the array

```
var some_array = [1, 1, 1, 1, 1, 1, 1, 1, 1];
```

The call

```
enter_copies(some_array, 4, 8, 2);
```

should enter 4 copies of the value 8 into `some_array`, starting at position 2. Thus, after the call, `some_array` should have the form `[1, 1, 8, 8, 8, 8, 1, 1, 1].`

## D. Generating sorted array [5 marks]

We now generate a sorted array from a given histogram. For each index *i* of the histogram, we enter the given number of copies of *i* into the sorted array, starting from 0. For example,

```
   generate_sorted([0, 2, 1, 1, 0, 2, 0, 1, 0, 0, 1, 0, 0]);
```

should return the array `[1, 1, 2, 3, 5, 5, 7, 10]`.

## E. Complexity of counting sort [2 mark]

We can now define the overall sorting function

```
function counting_sort(unsorted, max) {
    return generate_sorted(make_histogram(unsorted, max));
}
```

The basic operation for this algorithm is array access and assignment, which we will call *array operations*. We define the *size of a sorting problem* as the maximum of the length of the given array `unsorted` and `max`, which is the given limit for the largest value of the array. Characterise the number of array operations for running `counting_sort(unsorted, max)`, using Θ-notation.

***The number of array operations for solving a sorting problem of size n using counting sort has order of growth:***

## The following question is not relevant for CS1101S as of 2019/20.

## ~~Question 4: Digital Orrery [14 marks]~~

~~An orrery is a mechanical clockwork model of the solar system, or parts of it. The image on the right shows an orrery at the Museum of the History of Science in Oxford. Jerry Sussman – one of the original authors of the CS1101S textbook – created a digital orrery, and used it in 1988 to predict the behaviour of the solar system for 845 million years, to study if it exhibits chaotic behaviour. In this question, we will lay the foundation for a simple digital orrery using Source.~~

### ~~A. [4 marks]~~

~~First, we shall define a coordinate system relative to which we position the bodies in the solar system. We limit ourselves to a classical, 3-dimensional, Euclidean space and choose a *heliocentric ecliptic rectangular coordinate system*, where the centre of the sun has coordinates (*x: 0, y: 0, z: 0*), and the Earth always has (*z: 0*). As units, we will use million kilometres. A possible location for the center of the Earth could be characterized by the Source object:~~

```
({ x: 145.9, y: 76.4, z: 0.0 });
```

~~Define a function `Body` that allows us to create such objects, using~~

```
var earth = new Body(145.9, 76.4, 0.0);
```

~~Add methods `get_x`, `get_y`, `get_z` that enable access to the coordinates as in~~

```
earth.get_x();  // returns 145.9
```

**~~B. [6 marks]~~**

~~Now we could define a few bodies in our digital orrery as follows:~~

```
var earth  = new Body(145.9, 76.4, 0.0  );
var moon   = new Body(145.3, 76.1, 0.001);
var sun    = new Body(  0.0,  0.0, 0.0  );
var system = list(earth, moon, sun);
```

~~Draw the environment diagram for this program, including the function `Body`, its methods, and the frames that result from calling `Body`.~~

## ~~C. [4 marks]~~

~~For our orrery to work, we need to keep track of the speed of the bodies, and not just their position. For this we assume we can use 'speed' objects.~~ ~~**Define a function `MovingBody`**~~ ~~that allows you to create new bodies using~~

```
var moving_earth = new MovingBody(145.9, 76.4, 0.0,
                                  initial_speed_of_earth);
// initial_speed_of_earth is given object stating speed of earth
```

~~Your function `MovingBody` should inherit from the function `Body` and make use of `Body` in its definition.~~

~~Also **define a method `get_speed`** such that we can retrieve the current speed of `moving_earth` by~~

```
var current_earth_speed = moving_earth.get_speed();
```

# Question 5: The Source metacircular evaluator [14 marks]

## A. Repeated parameters [5 marks]

JavaScript insists that function parameters are unique. For example, a function

```
const power = (x, x) => x === 0 ? 1 : x * power(x, x - 1);
```

lists `x` twice as parameter of `power`. Unlike JavaScript, the metacircular evaluator accepts this program although this is clearly a programming error and the meaning of the program is not clear.

Modify the implementation of lambda expressions in the metacircular evaluator such that any lambda expression where the parameters are not unique displays an error and returns `undefined`. For example, the evaluation of the function definition above will lead to the display of the string
"`parameters in function definition not unique`"
and bind the name `power` to the value `undefined`.

**Hint:** Recall that lambda expressions are implemented in the metacircular evaluator as follows:

```
function evaluate(component, env) {
   return is_literal(component)
          ? literal_value(component)
          …
          : is_lambda_expression(component)
          ? make_function(lambda_parameter_symbols(component),
                          lambda_body(component), env)
          …
          : error(component, "Unknown syntax -- evaluate");
}
```

## The following question is not relevant for CS1101S as of 2019/20.

### ~~B. Property assignment to primitive values [3 marks]~~

~~JavaScript allows property assignment to primitive values such as boolean values, numbers and strings. Any such attempts are just ignored by the JavaScript evaluator. For example,~~

```
true["aaa"]  = 1;    // has no effect
"bbb" . ccc  = 2;    // has no effect
123["ddd"]   = 3;    // has no effect
```

~~All these property assignments succeed in JavaScript, but have no effect. Most of the time, such operations will be a programming error, and the author intended something else. JavaScript's strict mode therefore generates an exception in these situations.~~

~~Modify property assignment such that error messages are displayed, in the examples above:~~

```
"expecting array/object/function value before [, found value true"
"expecting array/object/function value before [, found value 'bbb'"
"expecting array/object/function value before [, found value 123"
```

~~**Hint:** Recall the current implementation of evaluation property assignment for MetaSource as follows.~~

```
function evaluate_property_assignment(stmt,env) {
    var obj = evaluate(object(stmt), env);
    var prop = evaluate(property(stmt), env);
    var val = evaluate(value(stmt), env);
    obj[prop] = val;
    return val;
}
```

~~You can assume that the builtin functions `is_object`, `is_array` and `is_function` are available and return **`true`** if and only if their argument is a Source object, Source array or Source function, respectively.~~

## C. The Essence of evaluation? [6 marks]

Let us assume we want to support a function `evaluate` in the interpreted language that takes a string as argument, parses it, evaluates the parsed program, and returns the result. This question explores what environment can or should be used for the evaluation.

Recall that primitive functions such as `pair`, `head` and `tail` are defined in the metacircular evaluator as follows:

```
// the global environment has bindings for all
// primitive functions, including the operators
const primitive_functions = list(list("pair", pair),
                                 list("head", head),
                                 list("tail", tail),
                                 …
                                );

const primitive_function_symbols =
        map(head, primitive_functions);

const primitive_function_objects =
        map(fun => list("primitive", head(tail(fun))),
            primitive_functions);

function setup_environment() {
    return extend_environment(
               append(primitive_function_symbols,
                      primitive_constant_symbols),
               append(primitive_function_objects,
                      primitive_constant_values),
               the_empty_environment);
}

const the_global_environment = setup_environment();

function parse_and_evaluate(input) {
    const program = parse(input);
    const implicit_top_level_block = make_block(program);
    return evaluate(implicit_top_level_block,
                    the_global_environment);
}
```

Our approach to provide an `evaluate` function in MetaSource shall be to add a line in the definition of `primitive_functions` as follows:

```
let primitive_functions
            = list(list("pair",          pair),
                   list("head",          head),
                   list("tail",          tail),
                   list("evaluate",      parse_and_evaluate),
                   …
                  );
```

Using this modification of the evaluator, what are the results of the following Source programs?

### C.1. [2 marks]

```
let x = 1;
parse_and_evaluate(
    "evaluate('let x = 2; x + x;');");
```

### C.2. [2 marks]

```
let x = 1;
parse_and_evaluate(
    `let x = 2;
     evaluate('function f(x) { return x + x; } f(3);');`);
```

### C.3. [2 marks]

```
let x = 1;
parse_and_evaluate(`let x = 2;
                    function f(x) {
                        return evaluate('x + x;');
                    }
                    f(3);`);
```

(Scratch paper. Do not tear off.)

(Scratch paper. Do not tear off.)

**——— END OF PAPER ———**