National University of Singapore
School of Computing
CS1101S: Programming Methodology
Semester I, 2016/2017

Discussion Group Exercises Week 9

Problems:

1. (a) Consider the following function definition:

```
function change(x, new_value) {
    x = new_value;
}
```

Consider the following statements:

```
var x = 0;
change(x, 1);
```

What is the value of x after the above statements are evaluated? Explain your answer using the environment model.

(b) From recitation week 8, you are given the function definition `make_stack` to create a new tagged data structure called stack and `clean` to empty an existing stack of any elements it may contain. Ben Bitdiddle wonders why a stateful stack needs to be a tagged data structure and proposes the following untagged version:

```
function make_stack() {
    return [];
}

function clean(stack) {
    stack = [];
}
```

Using the environment model and appropriate examples, help explain to Ben why such a tagged data structure is necessary.

2. Given the following program:

```
var a = 10;

function foo(x) {
    var b = 0;

    function goo(x) {
        var a = 30;
        // Breakpoint #1
        if (x <= 2) {
            a = a + x;
            b = b + x;
            // Breakpoint #2
        } else {
            goo(x - 1);
        }
    }

    // Breakpoint #3
    a = a + x;
    b = b + x;
    // Breakpoint #4
    goo(3);
    // Breakpoint #5
}

// Breakpoint #6
foo(1);
// Breakpoint #7
```

Evaluate the program and draw the environment structure produced at each break point.

3. SICP, Exercise 3.2

   In software testing applications, it is useful to be able to count the number of times a given function is called during the course of a computation. Write a function `make_monitored` that takes as input a function, `f`, that itself takes one input. The result returned by `make_monitored` is a third function, say `mf`, that keeps track of the number of times it has been called by maintaining an internal counter. If the input to `mf` is the string "get_count", then `mf` returns the value of the counter. If the input is the string "reset_count", then `mf` resets the counter to zero. For any other input, mf returns the result of calling `f` on that input and increments the counter. For instance, we could make a monitored version of the `Math.sqrt` function:

```
    var s = make_monitored(Math.sqrt);

    s(100);
    > 10
    s("get_count");
    > 1
    s("reset_count");
    s("get_count");
    > 0
```

4. The following procedure is quite useful, although obscure:

```
function mystery(x) {
    function loop(x, y) {
        if (is_empty_list(x)) {
            return y;
        } else {
            var temp = tail(x);
            set_tail(x, y);
            return loop(temp, x);
        }
    }
    return loop(x, []);
}
```

`loop` uses the temporary variable `temp` to hold the old value of the tail of `x`, since the `set_tail` on the next line destroys the tail. Explain what mystery does in general.

Suppose `v` is defined by:

```
var v = list(1, 2, 3, 4);
```

Draw the box-and-pointer diagram that represents the list to which `v` is bound. Suppose that we now evaluate:

```
var w = mystery(v);
```

Draw box-and-pointer diagrams that show the structures `v` and `w` after evaluating this expression. What would be printed as the values of `v` and `w`?

5. SICP, Exercises 3.16 and 3.17

   Ben Bitdiddle decides to write a procedure to count the number of pairs in any list structure. ”Its easy,” he reasons. ”The number of pairs in any structure is the number in the head plus the number in the tail plus one more to count the current pair.” So Ben writes the following function:

```
function count_pairs(x) {
    if (!is_pair(x)) {
        return 0;
    } else {
        return 1 + count_pairs(head(x)) + count_pairs(tail(x));
    }
}
```

   Show that this function is not correct. In particular, draw box-and-pointer diagrams representing list structures made up of exactly three pairs for which Bens procedure would return 3, return 4, return 7, or never return at all.

   Devise a correct version count-pairs that returns the number of distinct pairs in any structure.

   (Hint: Traverse the structure, maintaining an auxiliary data structure that is used to keep track of which pairs have already been counted.)

6. (Modified from 2011 Midterm) You should have by now been familiar with list and its derived data structure stack and queue. Typically, we can only traverse a list in one direction by using the accessor functions `head` and `tail`. In this question, we will introduce a general data structure called *doubly-linked list* which allows us to traverse the list in both direction. A doubly-linked list consists of nodes, each of which can either be an empty list or a list of three entries. One entry is a data item and the other two entries are the previous and next nodes. The first and last nodes of a doubly-linked list should have empty nodes i.e. empty lists as their previous and next node respectively.

   (a) Define the constructor function `make_node` and accessor functions `get_data`, `get_prev` and `get_next` which satisfy the following contracts:

   - `get_data(make_node(data, prev, next))` returns `data`
   - `get_prev(make_node(data, prev, next))` returns `prev`
   - `get_next(make_node(data, prev, next))` returns `next`

```
function make_node(data, prev, next) {
    // Your answer
}

function get_prev(node) {
    // Your answer

}

function get_next(node) {
    // Your answer
}

function get_data(node) {
    // Your answer

}
```

Also define the functions `empty_node` and `is_empty_node` to create and check for an empty node in a doubly-linked list.

```
function empty_node() {
    // Your answer

}

function is_empty_node() {
    // Your answer

}
```

(b) Define the mutator functions `set_data`, `set_previous` and `set_next` that take in a node `n` and a value `v` and set the data entry, previous entry and next entry of `n` to value `v` respectively.

```
function set_data(n, v) {
    // Your answer

}

function set_previous(n, v) {
    // Your answer

}

function set_next(n, v) {
    // Your answer

}
```

(c) Define the functions `insert_before` and `insert_after` that take in two nodes `n1` and `n2` and insert `n2` before and after `n1` respectively in the *doubly-linked list* that contains `n1`. Also define another function `remove` that take in a node `n` and remove that node from the *doubly-linked list* that it resides in.

```
function insert_before(n1, n2) {
    // Your answer

}

function insert_after(n1, n2) {
    // Your answer

}

function remove(n) {
    // Your answer

}
```

(d) Define a function `has_loop` that takes in a doubly-linked list and determines if it contains a loop. If there is a loop, the function returns `true`, otherwise the function returns `false`.

```
function has_loop(dlist) {
    // Your answer

}
```

(e) Define a function `dlist_to_list` that takes in a doubly-linked list and returns a regular list with elements in the same order

```
function dlist_to_list(dlst) {
    // Your answer

}
```

(f) Define a function `list_to_dlist` that takes in a regular list and returns a doubly-linked list with elements in the same order. Your function should return the first node of the doubly-linked list

```
function list_to_dlist(lst) {
    // Your answer

}
```

(g) Define a function `reverse_dlist` that takes in a doubly-linked list and return a doubly-linked list with the elements in reversed order.

```
function reverse_dlist(dlst) {
    // Your answer

}
```