OCaml: Binary Tree

Liao Jianglong

October 2, 2017

Table of Contents

Introduction

This midterm project mainly revisits what we have done in the past, including definitions, self-reference, unit-test functions, inductive specification of the computation and structurally recursive function.

The project mainly contains 5 parts:

1. Revisitng accurate definition using Aristotle's four causes; revisiting microprocessor, interpreter, compiler, and self-reference
2. Revisiting induction and recursion through alternative types of binary tree
3. Further implementing structurally recursive functions to transform left-binary-tree
4. Further implementing structurally recursive functions to transform right-binary-tree
5. Rebalancing left-trees into right-trees and vice-versa.

More details will be shown in the following parts and conclusion.

Part 1

Question 1.1

1. A microprocessor
    - Microprocessor operates (efficient cause)
    - programs written in symbolic machine code which is (material cause)
    - implemented in hardware (formal cause)
    - to execute the programs (final cause)
2. An ordinary computer printer
    - After taking instructions from computer (formal cause)
    - a printer converts (efficient cause)
    - graphics or/and text on computer (material cause)
    - to physical graphics or/and text on paper (final cause)
3. A binary tree
    - A binary arranges (efficient cause)
    - data (material cause)
    - such that each node has at most two leaves (formal cause)
    - so data can be stored and processed efficiently (final cause)

Question 1.2

a. Yes, we can.

1. Using the compiler from Python to x86 written in x86, compile the interpreter for Scheme written in Python. The result is an interpreter for Scheme written in x86.
2. The x86 microprocessor executes the interpreter for Scheme written in x86, this interpreter executes the interpreter for OCaml written in Scheme, and this interpreter executes the OCaml program, as shown in the figure below.

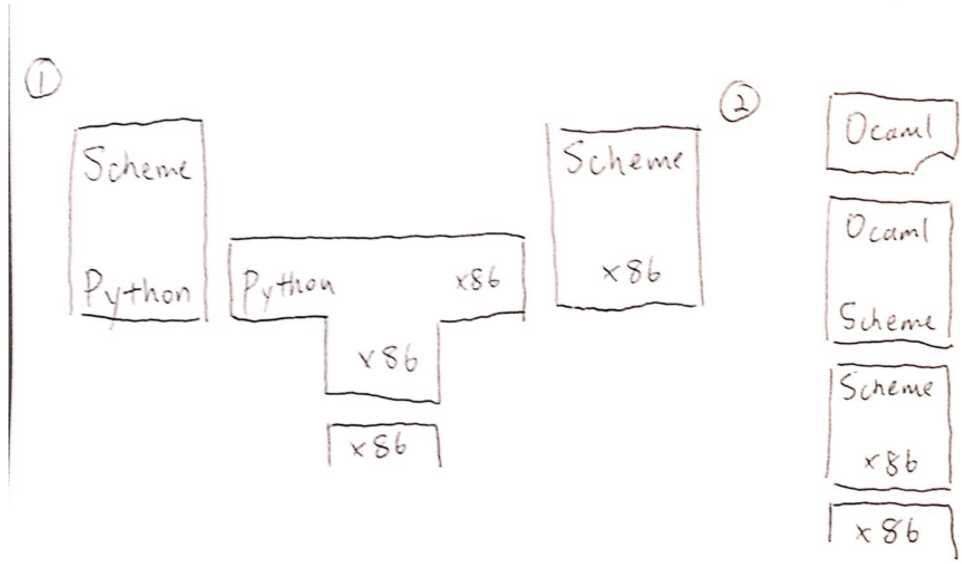

Figure 1 Question 1.2.a

b. Yes, we can.

1. The x86 microprocessor executes the interpreter for Scheme written in x86, this interpreter executes the compiler from Python to Scheme written in Scheme, which compiles the interpreter for Python written in Python. The result is an interpreter for Python written in Scheme.
2. The x86 microprocessor executes the interpreter for Scheme written in x86, this interpreter executes the interpreter for Python written in Scheme, and this interpreter executes the compiler from OCaml to x86 written in Python, which compiles the program written in OCaml into a program written in x86.
3. The x86 microprocessor executes this program written in x86, as shown in the figure below.

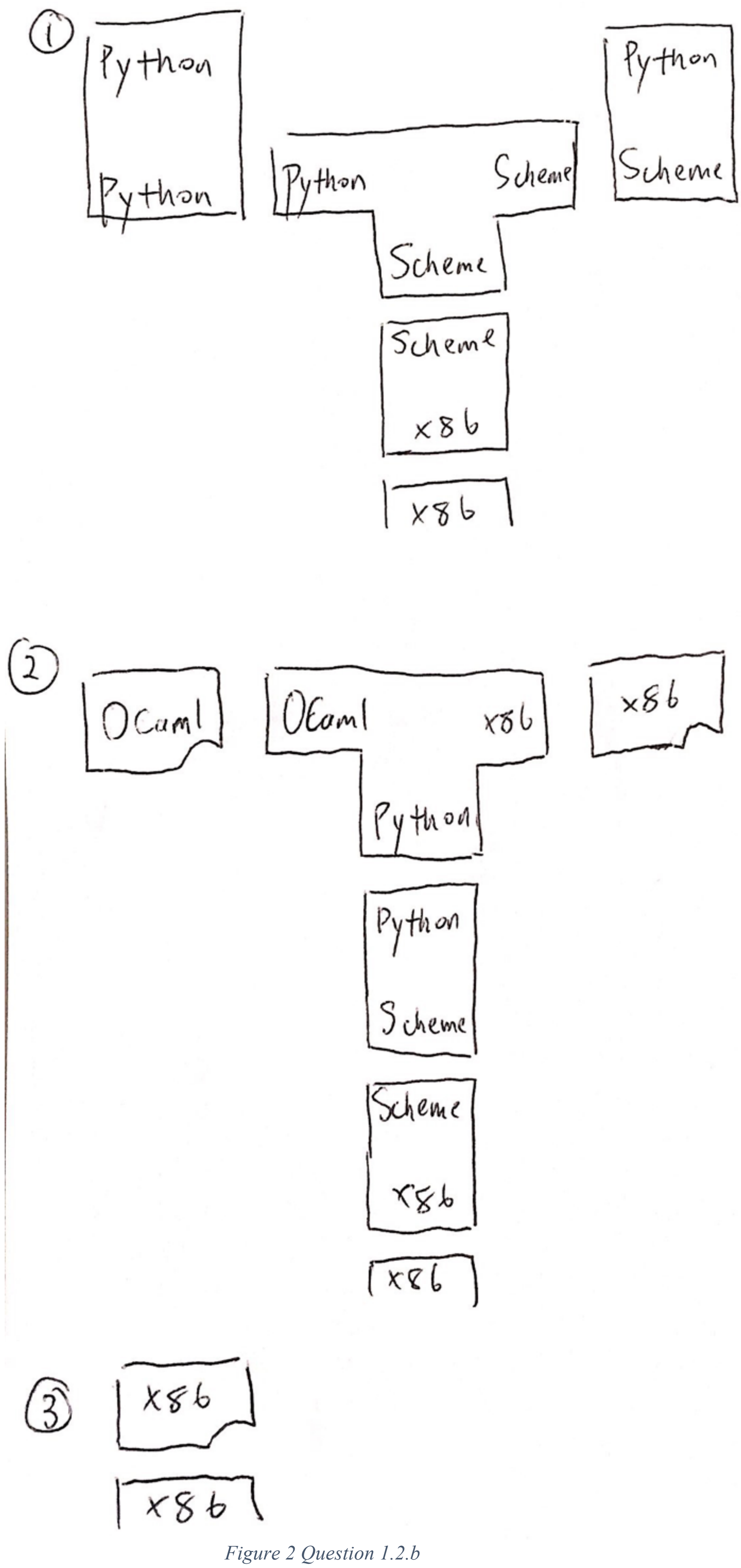

Figure 2 Question 1.2.b

c. Yes, we can.
    1. The x86 microprocessor executes the interpreter for Scheme written in x86, this interpreter executes the compiler from OCaml to Scheme written in Scheme, which compiles the program written in OCaml. The result is a program written in Scheme.
    2. The x86 microprocessor executes the interpreter for Scheme written in x86, this interpreter executes the program written in Scheme, as shown in the figure below.

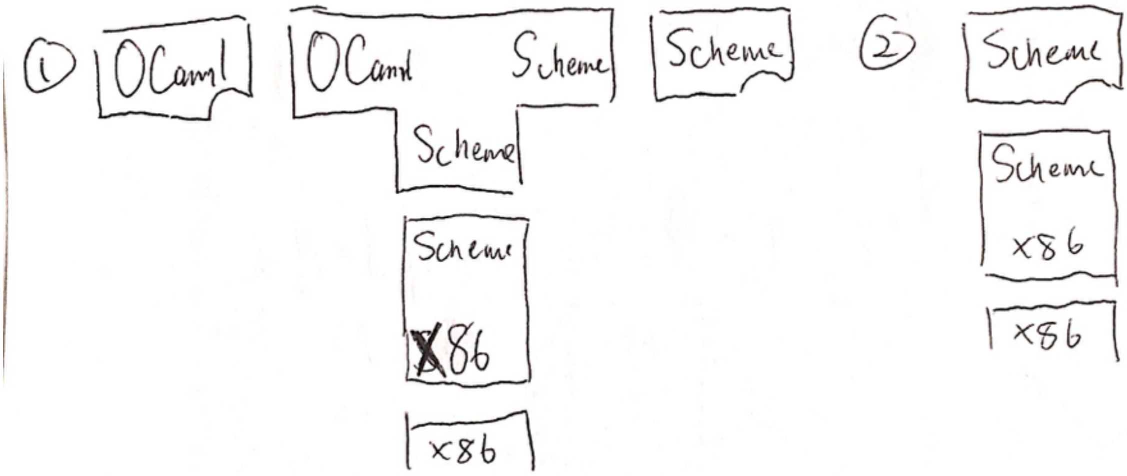

Figure 3 Question 1.2.c

Question 1.3

a. 1. true, always, 2 times an integer from 0 to 4 will always give an even number.
b. 3. sometimes true, sometimes false, but an exception is never raised, because 2 * 5 = 10, an integer from 0 to 9 is either an odd or even number.
c. 1. true, always, because 2 times an integer from 0 to 5 will always give an even number.
d. 3. sometimes true, sometimes false, but an exception is never raised, because 2 * 6 = 12, an integer from 0 to 11 is either an odd or even number.
e. 2. sometimes a random number, sometimes no result because an exception is raised, because Random.int (6 * 9) will give an integer from 0 to 53, and the outer Random.int (Random.int (6 * 9)) will give an integer from 0 to the integer that Random.int (6 * 9) gives, but if Random.int (6 * 9) gives 0, then Random.int 0 gives no result because applying Random.int to a non-positive integer is undefined and so an exception is raised.

Part 2

Let's declare some data type before we start answering questions below. For this part, we mainly need to declare an data type called binary_tree'. This binary_tree' is inductively constructed as follows:

- in the base case, a leaf is written as Leaf'; and
- in the inductive case, given two binary trees t1, an integer n and t2, a node grouping these two trees (as subtrees) is written as Node' (t1, n, t2).

Question 2.1

Textual description

The binary_tree' is constructed recursively, so the function that accounts the number of leaves is constructed by recursively adding the number of the leaves in sub trees.

Construction of unit-tests

We have included different candidate binary trees that range from 1 to 6 leaves. Both Toby and I have constructed different binary trees of 1 to 4 leaves, and Kota constructed an outlier, a binary tree of 6 leaves to ensure the validity of unit-test function.

```
let test_count_leaf candidate_leaf =
    (* test_count_leaf : (binary_tree' -> int) -> bool *)
        (candidate_leaf (Leaf')
        = 1)
    &&
        (candidate_leaf (Node' (Leaf', 1, Leaf'))
        = 2)
    &&
        (candidate_leaf (Node' (Node' (Leaf', 2,
                                        Leaf')
                                        , 1, Leaf'))
        = 3)
    &&
        (candidate_leaf (Node' (Leaf', 1,
                                (Node' (Leaf', 2,
                                        Leaf'))))
        = 3)
    &&
        (candidate_leaf (Node' (Node' (Leaf', 2,
                                        Leaf'), 1,
                                (Node' (Leaf', 3,
                                        Leaf'))))
        = 4)
    &&
        (candidate_leaf (Node'(Node'(Node'(Leaf', 4, Leaf'), 2,
                                           Leaf' ),1,
                                    Node'(Leaf', 3,
                                           Node'(Leaf', 5,
                                                  Leaf'))))
        = 6)
        (* etc. *);;
```

Inductive specification

Base case:

the number of leaves in Leaf' is 1

Inductive case:

For any binary tree' t1 (resp. t2) whose number of leaves is n1 (resp. n2), the number of leaves in Node' (t1, int, t2) is the sum of n1 and n2 to account for the leaves in t1 and in t2.

We then construct the recursive function according to this specification.

```
let rec count_leaf t =
    (* count_leaf : binary_tree' -> int *)
    match t with
    | Leaf' ->
       1
    | Node' (t1, n, t2) ->
       let n1 = count_leaf t1
       and n2 = count_leaf t2
    in n1 + n2;;
```

Implementation of unit-tests

The count_leaf function successfully passed the unit-tests. No alert showed.

```
let () = assert(test_count_leaf count_leaf);;
```

Question 2.2

Textual description

Similar to the function above, this function accounts the number of nodes by recursively adding the number of the node in sub trees, plus 1 to account for the node that hold two sub trees together.

Construction of unit-tests

Again, we have included different candidates that ranges from 1 to 5 nodes. Both Toby and I have constructed different binary trees of 1 to 3 nodes, and Kota constructed a binary tree of 5 nodes.

```
let test_count_node candidate_node =
    (* test_count_leaf : (binary_tree' -> int) -> bool *)
        (candidate_node (Leaf')
        = 0)
    &&
        (candidate_node (Node' (Leaf', 1, Leaf'))
        = 1)
    &&
        (candidate_node (Node' (Node' (Leaf', 2,
                                        Leaf')
                                      , 1, Leaf'))
        = 2)
    &&
        (candidate_node (Node' (Leaf', 1,
                                (Node' (Leaf', 2,
                                        Leaf'))))
        = 2)
    &&
        (candidate_node (Node' (Node' (Leaf', 2,
                                        Leaf'), 1,
                                 (Node' (Leaf', 3,
                                        Leaf'))))
        = 3)
    &&
        (candidate_node (Node'(Node'(Node'(Leaf', 4,
                                             Leaf'), 2,
                                                Leaf' ),1,
                                      Node'(Leaf', 3,
                                              Node'(Leaf', 5,
                                                     Leaf'))))
    = 5 )
        (* etc. *);;
```

Inductive specification

Base case:

the number of nodes in Leaf' is 0

Inductive case:

for any binary tree' t1 (resp. t2) whose number of nodes is n1 (resp. n2),

> the number of nodes of Node (t1, int, t2) is the sum of n1 and n2 to account for the nodes in t1 and t2,
>
> plus 1 to account for the node that holds t1 and t2 together.

We then construct this function according to this specification.

```
let rec count_node t =
    (* count_node : binary_tree' -> int *)
    match t with
    | Leaf' ->
        0
    | Node' (t1, n, t2) ->
        let n1 = count_node t1
        and n2 = count_node t2
    in n1 + n2 + 1;;
```

Implementation of unit-tests

The count_node function successfully passed the unit-tests.

```
let () = assert(test_count_node count_node);;
```

Question 2.3

There seems to be a relationship between the number of leaves and node:

Number of leaves = Number of node + 1

Thus this observation becomes our hypothesis:

for any tree, its number of leaves is its number of nodes, plus 1

Base case:

given a Leaf, the number of a node is 0, and the number of leaves is 1.

Indeed 1 = 0 + 1, so the hypothesis is true for the base case.

Inductive case:

Given two trees t1 and t2 such as

t1 contains l1 leaves and n1 nodes, and l1 = n1 + 1(inductive hypothesis), and

t2 contains l2 leaves and n2 nodes, and l2 = n2 + 1(inductive hypothesis)

Given a tree whose left subtree is t1 and right subtree t2,

this tree contains l1 + l2 leaves, which is n1 + 1 + n2 + 1, and

this tree contains n1 + n2 + 1 nodes,

so, the number of leaves of this tree, l1 + l2, is its number of nodes, i.e., n1 + n2 + 1, plus 1

Because the hypothesis is true for base case and inductive case, the hypothesis is true for all trees.

Question 2.4

Textual description

This left_balanced' function will test whether the binary tree is left-balanced, i.e. all its right sub-trees are leaves.

Construction of unit-tests

We included different candidate binary trees. Some are left-balanced while the others are not. Because the function outputs in Booleans, the result will either be true or false.

```
let test_left_balanced' candidate =
    (candidate Leaf'
     = true )
  &&
    (candidate (Node' (Leaf',
                       1,
                       Leaf'))
     = true )
  &&
    (candidate (Node' (Node' (Leaf',
                              2,
                              Leaf'),
                       1,
                       Leaf'))
     = true )
  &&
    (candidate (Node' (Node' (Node' (Leaf',
                                     3,
                                     Leaf'),
                              2,
                              Leaf'),
                       1,
                       Leaf'))
     = true )
  &&
    (candidate (Node' (Node' (Node' (Node' (Leaf',
                                            3,
                                            Leaf'),
                                     2,
                                     Leaf'),
                              1,
                              Leaf'),
                       0,
                      Leaf'))
     = true )
  &&
    (candidate (Node' (Node' (Leaf',
                              1,
                              Leaf'),
                       3,
                       Node' (Leaf',
                              0,
                              Leaf')))
     = false )
    &&(* addition test by our team *)
    (candidate (Node' (Node' (Leaf', 2,
                    Leaf'), 1,
           (Node' (Leaf', 3,
                  Leaf'))))
     = false)
  (* etc. *);;
```

Inductive specification

Base case:

the binary tree obtained by evaluating Leaf' is vacuously left-balanced.

Inductive case:

Given two binary trees t1 and t2,

1) if t2 is a leaf, then Node (t1, t2) is left-balanced

whenever t1 is left-balanced,

2) if t2 is a node, then Node (t1, t2) is not left-balanced

In OCaml, this function is implemented as a *structurally recursive function* to check whether all right sub-trees are leaves.

```
let rec left_balanced' t =
  (* left_balanced' : binary_tree' -> bool *)
  match t with
  | Leaf' ->
    true
  | Node' (t1, n, t2) ->
    let b1 = left_balanced' t1
    and b2 = (match t2 with
          | Leaf' ->
          true
          | Node' _ ->
          false)
    in b1 && b2;;
```

Implementation of unit-tests

The left_balance function successfully passed the unit-tests.

```
let() = assert (test_left_balanced' left_balanced');;
```

Question 2.5

Textual description

Similar to above function, the right_balanced' function will test whether the binary tree is right-balanced, i.e. all its left sub-trees are leaves.

Construction of unit-tests

Similar to the unit-test function for left_balanced, some candidate binary trees are right-balanced while the others are not.

```
let test_right_balanced' candidate_right =
  (* test given by instruction *)
  (candidate_right (Leaf')
    = true)
  &&
  (candidate_right (Node' (Leaf', 1,
                    Leaf'))
    = true)
  &&
  (candidate_right (Node' (Leaf', 1,
                          (Node' (Leaf', 2,
                          Leaf'))))
    = true)
  &&
  (candidate_right (Node' (Leaf', 1,
                          (Node' (Leaf', 2,
                                (Node' (Leaf', 3,
                                      Leaf'))))))
    = true)
  &&
  (candidate_right (Node' (Leaf', 1,
                          (Node' (Leaf', 2,
                                (Node' (Leaf', 3,
                                      (Node' (Leaf', 4,
                                            Leaf'))))))))
    = true)
  &&
  (candidate_right (Node' (Node' (Leaf', 2,
                           Leaf'), 1,
                  (Node' (Leaf', 3,
                        Leaf'))))
    = false)
  (* etc *);;
```

Inductive specification

Base case:

the binary tree obtained by evaluating Leaf' is vacuously right-balanced.

Inductive case:

Given two binary trees t1 and t2,

1) if t1 is a leaf, then Node (t1, t2) is right-balanced whenever t1 is right-balanced,

2) if t1 is a node, then Node (t1, t2) is not right-balanced

In OCaml, this function is implemented as a *structurally recursive function* to check whether all left sub-trees are leaves.

```
let rec right_balanced' t =
  (* right_balanced' : binary_tree' -> bool *)
  match t with
  | Leaf' ->
    true
  | Node' (t1, n, t2) ->
    let b2 = right_balanced' t2
    and b1 = (match t1 with
              | Leaf' ->
              true
              | Node' _ ->
              false)
    in b1 && b2;;

let() = assert(test_right_balanced' right_balanced');;
```

Implementation of unit-tests

The right_balance function successfully passed the unit-tests.

```
let() = assert(test_right_balanced' right_balanced');;
```

Question 2.6

For a binary tree to be both left and right balanced, the tree's sub-trees should be both leaves. So only two trees are both left and right balanced

these trees are:

1. Leaf' (vacuously true)
2. Node' (Leaf', n, Leaf') for any integer n

Part 3

"Believe you can and you're halfway there". - Theodore Roosevelt

Question 3.1

This project_binary_tree'_into_left_binary_tree' function is partial: it is not defined on some of its input (ie.all binary trees). For example, input a not left-balanced binary tree is undefined, because the function only inputs left-balanced binary trees. So to make the function total, we need an option type. In this way, inputing a not left-balanced binary tree will return None_left_binary_tree'.

Question 3.3

Textual description

The stitching function stitches the top of the left_binary_tree t1 to the bottom of the left_binary_tree t2.

Construction of unit-tests

The unit-test function is given by the instruction and we have added the additional pictorial example to the unit test.

```
let test_left_stitch candidate =
  (* test_left_stitch : (left_binary_tree' -> left_binary_tree' -> left_binary_tree') -> bool *)
  (candidate Left_Leaf'
             Left_Leaf'
   = Left_Leaf')
  &&
  (candidate (Left_Node' (Left_Leaf', 1))
             Left_Leaf'
   = Left_Node' (Left_Leaf', 1))
  &&
  (candidate Left_Leaf'
             (Left_Node' (Left_Leaf', 1))
   = Left_Node' (Left_Leaf', 1))
  &&
  (candidate (Left_Node' (Left_Node' (Left_Leaf', 2), 1))
             Left_Leaf'
   = Left_Node' (Left_Node' (Left_Leaf', 2), 1))
  &&
  (candidate Left_Leaf'
             (Left_Node' (Left_Node' (Left_Leaf', 2), 1))
   = Left_Node' (Left_Node' (Left_Leaf', 2), 1))
  &&
  (candidate (Left_Node' (Left_Node' (Left_Leaf', 4), 3))
             (Left_Node' (Left_Node' (Left_Leaf', 2), 1))
   = Left_Node' (Left_Node' (Left_Node' (Left_Node' (Left_Leaf', 4), 3), 2), 1))
  &&
(candidate (Left_Node' (Left_Node' (Left_Leaf', 5), 4))
             (Left_Node' (Left_Node' (Left_Node' (Left_Leaf', 3), 2), 1))
   = Left_Node' (Left_Node' (Left_Node' (Left_Node' (Left_Node' (Left_Leaf', 5), 4), 3), 2), 1))
  (* etc. *);;
```

Inductive specification

Base case:

given a left_binary_tree' t1 and Left_Leaf' t2

stitching t1 and t2 gives t1

Induction case:

given any left_binary_tree' t1 and left_binary_tree' t2'

such that stitching t1 and t2' gives t3 in Left_Node' (t3, n)

In OCaml, this function is implemented as a structurally recursive function taking two left-binary trees and mapping to a stitched left-binary tree:

```
let rec left_stitch t1 t2 =
  (* left_stich: left_binary_tree' -> left_binary_tree' -> left_binary_tree' *)
match t2 with
| Left_Leaf' ->
  t1
| Left_Node' (t2', n) ->
    let t3 = left_stitch t1 t2'
  in Left_Node' (t3, n);;
```

Implementation of unit-tests

The left-stitch function successfully passed the unit-tests.

```
let() =assert(test_left_stitch left_stitch);;
```

Question 3.4

Textual description

Intuitively, the OCaml function left_rotate recursively rotates the left-most sub-tree to the bottom and transform to a left_binary_tree.

Construction of unit-tests

The unit-test function is given by the instruction.

```
let test_left_rotate candidate =
 (* test_left_rotate : (binary_tree' -> left_binary_tree') -> bool *)
  (candidate Leaf'
   = Left_Leaf')
  &&
  (candidate (Node' (Leaf',
                     1,
                     Leaf'))
   = Left_Node' (Left_Leaf',
                 1))
  &&
  (candidate (Node' (Node' (Leaf',
                            2,
                            Leaf'),
                     1,
                     Node' (Leaf',
                            3,
                            Leaf')))
   = Left_Node' (Left_Node' (Left_Node' (Left_Leaf',
                                         2),
                             1),
                 3))
  &&
  (candidate (Node' (Node' (Node' (Leaf',
                                   4,
                                   Leaf'),
                            2,
                            Node' (Leaf',
                                   5,
                                   Leaf')),
                     1,
                     Node' (Node' (Leaf',
                                   6,
                                   Leaf'),
                            3,
                            Node' (Leaf',
                                   7,
                                   Leaf'))))
   = Left_Node' (Left_Node' (Left_Node' (Left_Node' (Left_Node' (Left_Node' (Left_Node' (Left_Leaf',
                                                                                         4),
                                                                             2),
                                                                 5),
                                                     1),
                                         6),
                             3),
                 7))
  (* etc. *);;
```

Inductive specification

Base case:

left_rotate rotates Leaf' to become Left_Leaf'

Induction case:

Given left sub tree of binary_tree' t1, interger n and sub tree of binary_tree' t2

such that left_rotate t1 gives left_binary_tree' lt1

and left_rotate t2 gives left_binary_tree' lt2

so that left_rotate Node' (t1, n, t2) gives left_stitich Left_Node' (lt1, n) and lt2

In OCaml, this function is implemented as a structurally recursive function mapping to a binary tree to a left-binary tree:

```
let rec left_rotate t =
  match t with
  | Leaf' ->
    Left_Leaf'
  | Node' (t1, n, t2) ->
    let lt1 = left_rotate t1
    and lt2 = left_rotate t2
  in left_stitch (Left_Node' (lt1, n)) lt2
;;
```

Implementation of unit-tests

The left-rotate function successfully passed the unit-tests.

```
let () = assert (test_left_rotate left_rotate);;
```

Question 3.5

The function left_flatten works because a given binary_tree' t is left rotated to a left_binary_tree'. For example:

Left rotate

Node' (Node' (Leaf', 2, Leaf'), 1, Node' (Leaf', 3, Leaf'))

Gives

Left_Node' (Left_Node' (Left_Node' (Left_Leaf', 2), 1), 3)

Then embed_left_binary_tree'_into_binary_tree' converts Left_Leaf' into a Leaf' and Left_Node' (t1, n) into Node' (t1, n, Leaf'). For example:

embed_left_binary_tree'_into_binary_tree'

Left_Node' (Left_Node' (Left_Node' (Left_Leaf', 2), 1), 3)

Gives

Node' (Node' (Node' (Leaf', 2, Leaf'), 1, Leaf'), 3, Leaf')

Therefore, left_flatten fits the bill.

Construction of unit-tests

Intuitively, the unit-test function is simple to write – for any given binary tree, we first left_rotate the tree into a left_binary_tree', and then add transform it to a binary tree.

```
let test_left_flatten candidate =
(* test_left_flatten : (binary_tree' -> binary_tree') -> bool *)
  (candidate Leaf'
   = Leaf')
 &&
 (candidate (Node' (Leaf',
                    1,
                    Leaf'))
  = Node' (Leaf',
           1, Leaf'))
 &&
  (candidate (Node' (Node' (Leaf',
                            2,
                            Leaf'),
                     1,
                     Node' (Leaf',
                            3,
                            Leaf')))
  = Node' (Node' (Node' (Leaf',
                         2, Leaf'),
                         1, Leaf'),
                  3, Leaf'))
 &&
 (candidate (Node' (Node' (Node' (Leaf',
                                  4,
                                  Leaf'),
                           2,
                           Node' (Leaf',
                                  5,
                                  Leaf')),
                    1,
                    Node' (Node' (Leaf',
                                  6,
                                  Leaf'),
                           3,
                           Node' (Leaf',
                                  7,
                                  Leaf'))))
  = Node' (Node' (Node' (Node' (Node' (Node' (Node' (Leaf',
                                                     4, Leaf'),
                                              2, Leaf'),
                                       5, Leaf'),
                                1, Leaf'),
                         6, Leaf'),
                  3, Leaf'),
           7, Leaf'))

 (* etc. *);;
```

Implementation of unit-tests

The left-flatten function successfully passed the unit-tests.

```
let () = assert (test_left_flatten left_flatten);;
```

Part 4

Question 4.1

This project_binary_tree'_into_right_binary_tree' function is partial: it is not defined on some of its input (ie.all binary trees). For example, input a not right-balanced binary tree is undefined, because the function only inputs right-balanced binary trees. So to make the function total, we need an option type. In this way, inputing a not right-balanced binary tree will return None_right_binary_tree'.

Question 4.3

Textual description

The stitching function stitches the bottom of the right_binary_tree t1 to the top of the right_binary_tree t2.

Construction of unit-tests

The unit-test function is given by the instruction and we have added the additional pictorial example to the unit test.

```
let test_right_stitch candidate =
(* test_right_stitch : (right_binary_tree' -> right_binary_tree'_binary_tree' -> right_binary_tree') -> bool *)
 (candidate Right_Leaf'
            Right_Leaf'
  = Right_Leaf')
 &&
 (candidate (Right_Node' (1, Right_Leaf'))
            Right_Leaf'
  = Right_Node' (1, Right_Leaf'))
 &&
 (candidate Right_Leaf'
            (Right_Node' (1, Right_Leaf'))
  = Right_Node' (1, Right_Leaf'))
 &&
 (candidate (Right_Node' (1, Right_Node' (2, Right_Leaf')))
            Right_Leaf'
  = Right_Node' (1, Right_Node' (2, Right_Leaf')))
 &&
 (candidate Right_Leaf'
            (Right_Node' (1, Right_Node' (2, Right_Leaf')))
  = Right_Node' (1, Right_Node' (2, Right_Leaf')))
 &&
 (candidate (Right_Node' (1, Right_Node' (2, Right_Leaf')))
            (Right_Node' (3, Right_Node' (4, Right_Leaf')))
  = Right_Node' (1, Right_Node' (2, Right_Node' (3, Right_Node' (4, Right_Leaf')))))
 &&
 (candidate (Right_Node' (1, Right_Node' (2, Right_Leaf')))
            (Right_Node' (3, Right_Node' (4, Right_Leaf')))
  = Right_Node' (1, Right_Node' (2, Right_Node' (3, Right_Node' (4, Right_Leaf')))))

 (* etc. *);;
```

Inductive specification

Base case:

given a Right_Leaf' t1and right_binary_tree' t2 stitching t1 and t2 gives t1

Induction case:

given any right_binary_tree' t1' and right_binary_tree' t2

such that stitching t1' and t2 gives t3 in Right_Node' (n, t3)

In OCaml, this function is implemented as a structurally recursive function taking two right-binary trees and mapping to a stitched right-binary tree:

```
let rec right_stitch t1 t2 =
match t1 with
| Right_Leaf' ->
  t2
| Right_Node' (n, t1') ->
    let t3 = right_stitch t1' t2
  in Right_Node' (n, t3);;
```

Implementation of unit-tests

The right-stitch function successfully passed the unit-tests.

```
let () = assert (test_right_stitch right_stitch);;
```

Question 4.4

Textual description

Intuitively, the OCaml function right_rotate recursively rotates the right-most sub-tree to the bottom and transform to a right_binary_tree.

Construction of unit-tests

The unit-test function is given by the instruction.

```
let test_right_rotate candidate =
 (* test_right_rotate : (binary_tree' -> right_binary_tree') -> bool *)
  (candidate Leaf'
   = Right_Leaf')
  &&
  (candidate (Node' (Leaf',
                     1,
                     Leaf'))
   = Right_Node' (1,
                  Right_Leaf'))
  &&
  (candidate (Node' (Node' (Leaf',
                            2,
                            Leaf'),
                     1,
                     Node' (Leaf',
                            3,
                            Leaf')))
   = Right_Node' (2,
                  Right_Node' (1,
                               Right_Node' (3,
                                            Right_Leaf'))))
  &&
  (candidate (Node' (Node' (Node' (Leaf',
                                   4,
                                   Leaf'),
                            2,
                            Node' (Leaf',
                                   5,
                                   Leaf')),
                     1,
                     Node' (Node' (Leaf',
                                   6,
                                   Leaf'),
                            3,
                            Node' (Leaf',
                                   7,
                                   Leaf'))))
   = Right_Node' (4,
                  Right_Node' (2,
                               Right_Node' (5,
                                            Right_Node' (1,
                                                         Right_Node' (6,
                                                                      Right_Node' (3,
                                                                                   Right_Node' (7,
                                                                                                Right_Leaf'))))))))
  (* etc. *);;
```

Inductive specification

Base case:

right_rotate rotates Leaf' to become Right_Leaf'

Induction case:

Given right sub tree of binary_tree' t1, interger n and left sub tree of binary_tree' t2

such that right_rotate t1 gives right_binary_tree' rt1

and right_rotate t2 gives right_binary_tree' rt2

so that right_rotate Node' (t1, n, t2) gives right_stitich rt1 and Right_Node' (n, rt2)

In OCaml, this function is implemented as a structurally recursive function mapping to a binary tree to a right-binary tree:

```
let rec right_rotate t =
  match t with
  | Leaf' -> Right_Leaf'
  | Node' (t1, n, t2) ->
    let rt1 = right_rotate t1
    and rt2 = right_rotate t2
   in right_stitch rt1 (Right_Node' (n, rt2));;
```

Implementation of unit-tests

The left-rotate function successfully passed the unit-tests.

```
let () = assert (test_right_rotate right_rotate);;
```

Question 4.5

The function right_flatten works because a given binary_tree' t is right rotated to a right_binary_tree'. For example:

Right rotate

Node' (Node' (Leaf', 2, Leaf'), 1, Node' (Leaf', 3, Leaf'))

Gives

Right_Node' (2, Right_Node' (1, Right_Node' (3, Right_Leaf')))

Then embed_right_binary_tree'_into_binary_tree' converts Right_Leaf' into a Leaf' and Right_Node' (n, t2) into Node' (Leaf', n, t2). For example:

embed_right_binary_tree'_into_binary_tree'

Right_Node' (2, Right_Node' (1, Right_Node' (3, Right_Leaf')))

Gives

Node' (Leaf', 2, Node' (Leaf', 1, Node' (Leaf', 3, Leaf')))

Therefore, right_flatten fits the bill.

Construction of unit-tests

Intuitively, the unit-test function is simple to write – for any given binary tree, we first right_rotate the tree into a right_binary_tree', and then add transform it to a binary tree.

```
 let test_right_flatten candidate=
 (candidate Leaf' = Leaf')
 &&
 (candidate (Node'(Leaf', 1, Leaf'))=
(Node'(Leaf', 1, Leaf')))
 &&
 (candidate (Node'(Node'(Leaf',2, Node'(Leaf', 5,Leaf')), 1,
  Node'(Leaf',3, Node'(Leaf', 4, Leaf'))))
=  (Node'(Leaf', 2, Node'(Leaf', 5, Node'(Leaf', 1, Node'(Leaf', 3, Node'(Leaf',4, Leaf'))))))
    (* etc *)    ;;
```

Implementation of unit-tests

The right_flatten function successfully passed the unit-tests.

```
let () = assert (test_right_flatten right_flatten);;
```

Part 5

Question 5.1

Textual description

Intuitively, this left_to_right function rotates the left-tree bottom node to the top and transforms to a right-tree.

Construction of unit-tests

```
let test_left_to_right candidate =
  (* test_left_to_right : (left_binary_tree' -> right_binary_tree') -> bool *)
   (candidate Left_Leaf'
    = Right_Leaf')
   &&
   (candidate (Left_Node' (Left_Leaf', 1))
    = Right_Node' (1, Right_Leaf'))
   &&
   (candidate (Left_Node' (Left_Node' (Left_Leaf', 2), 1))
    = Right_Node' (2, Right_Node' (1, Right_Leaf')))
   &&
   (candidate (Left_Node' (Left_Node' (Left_Node' (Left_Leaf', 3), 2), 1))
    = Right_Node' (3, Right_Node' (2, Right_Node' (1, Right_Leaf'))))
   (* etc. *);;
```

Specification

This time, we do not need to recursively construct the function from ground-up, but combine the function we have previous define. To maps a left-balanced tree to a right-balanced tree, all we need to do is first embed the left-tree into a binary tree, and then right rotate the tree into a right-tree.

Thus this OCaml function maps a binary tree to a binary tree:

```
let left_to_right t =
  (* left_flatten : binary_tree' -> binary_tree' *)
   right_rotate (embed_left_binary_tree'_into_binary_tree' t);;
```

Implementation of unit-tests

The left_to_right function successfully passed the unit-tests.

```
let () = assert (test_left_to_right left_to_right);;
```

Question 5.2

Textual description

Similarly, this right_to_left function rotates the right-tree from bottom node to the top and transforms to a left-tree.

Construction of unit-tests

```
let test_right_to_left candidate =
 (* test_right_to_left : (right_binary_tree' -> left_binary_tree') -> bool *)
  (candidate Right_Leaf'
   = Left_Leaf')
  &&
  (candidate (Right_Node' (1, Right_Leaf'))
   = Left_Node' (Left_Leaf', 1))
  &&
  (candidate (Right_Node' (2, Right_Node' (1, Right_Leaf')))
   = Left_Node' (Left_Node' (Left_Leaf', 2), 1))
  &&
  (candidate (Right_Node' (3, Right_Node' (2, Right_Node' (1, Right_Leaf'))))
   = Left_Node' (Left_Node' (Left_Node' (Left_Leaf', 3), 2), 1))
  (* etc. *);;
```

Specification

Similarly, we do not need to recursively construct the function from ground-up, but combine the function we have previous define. To maps a right-balanced tree to a left-balanced tree, all we need to do is first embed the right-tree into a binary tree, and then left rotate the tree into a left-tree.

Thus this OCaml function maps a binary tree to a binary tree:

```
 let right_to_left t =
(* left_flatten : binary_tree' -> binary_tree' *)
 left_rotate (embed_right_binary_tree'_into_binary_tree' t);;
```

Implementation of unit-tests

The left_to_right function successfully passed the unit-tests.

```
let () = assert (test_right_to_left right_to_left);;
```

Question 5.3

The right_to_left and left_to_right functions above are inverse functions to each other. When left_to_right is applied to right_to_left function with input t and vice versa, we obtain the starting input t as the output.

For example, implement right_to_left function to a right_binary_tree' gives a left_binary_tree’. Then implement left_to_right function to this tree gives the original input - the right_binary_tree’, vice versa.

Question 5.4

Both implementations past unit tests.

```
let () = assert (test_left_rotate
  (fun t -> right_to_left (right_rotate t)));;

let () = assert (test_right_rotate
  (fun t -> left_to_right (left_rotate t)));;
```

1.Because a binary_tree t is first right-rotated into a right_binary_tree'

Then, right_to_left turns right_binary_tree' into a left_binary_tree'.

right_to_left (right_rotate t) has the same transformative effect as left_rotate

So fun t -> right_to_left (right_rotate t) passes test_left_rotate.

2. Because a binary_tree t is first left-rotated into a left_binary_tree'

Then, left_to_right turns left_binary_tree' into a right_binary_tree'.

left_to_right (left_rotate t) has the same transformative effect as right_rotate

So fun t -> left_to_right (left_rotate t) passes test_right_rotate.

Conclusion

I have finally come to this part. Up to this point, we have successfully solved all the questions in this project and gain more insights than we did in weekly hand-in. I have made several progress since the start of mid-term project.

1. With the help of Aristotle's four causes, we are now able to clearly give a definition about an item or concept.
2. Through continuously building and transforming different types of binary trees, I have become more crafted in recursive function. Also with the help of inductive specification of implementation, I have a better understanding of the train of thoughts under the structurally recursive function.
3. I have learn how to construct simple unit-test function, and use that to test our program.

4. After understanding Part 3, we can quickly apply what we have learned to Part 4, which is quite similar, but structurally opposed to Part 3.
5. I gradually become more and more fascinated to the beauty of recursion, which can represent large amount of computation with minimum of code.

Future Goals

In this project, there are still several things that I would like to advance in the next few weeks before the end of semester:

1. Try to develop more complex and comprehensive unit tests using such as random number generator.
2. Practice more structurally recursive implementation so I can become more proficient in it.

This project helps to refresh my knowledge of functional programming, and I think through this long process I have greatly improved my crafts.

(Word Count: 3124)