Midterm project

The goal of this midterm project is to give you a chance to reflect on what you have been exposed to in this course since the beginning of the semester, including self-reference.

Expectations

You are expected:

- to work in groups, as you did for the weekly handins,
- to share your unit tests, all for one and one for all, and to give credit where credit is due in each of your unit-test functions:

```
let test_something_or_other candidate =
  (* original tests: *)
  (candidate ... = ...)
  &&
  ...
  &&
  (candidate ... = ...)
  &&
  (* my own tests: *)
  (candidate ... = ...)
  &&
  ...
  &&
  (candidate ... = ...)
  &&
  (* Athos's tests: *)
  (candidate ... = ...)
  &&
  ...
  &&
  (candidate ... = ...)
  &&
  (* Porthos's tests: *)
  (candidate ... = ...)
  &&
  ...
  &&
  (candidate ... = ...)
```

```
&&
(* Aramis's tests: *)
(candidate ... = ...)
&&
...
&&
(candidate ... = ...)
(* etc. *);;
```

(If one of Athos's tests helped you debug your code, do mention it somewhere, be it in a comment or in the report.)

- to write an individual report, using your own words.

Throughout, remember to embrace the structure:

1. textual description of a computation,
2. implementation of a unit-test function,
3. inductive specification of the computation,
4. implementation of this inductive specification as a structurally recursive function, and
5. verification that the implementation passes the unit test.

Resources

- The OCaml code for the present midterm project (latest version: 23 Sep 2017).

Part 1

This part is a warmup.

Question 1.1

Using (part of or all of) Aristotle's four causes, describe:

a. a microprocessor
b. an ordinary computer printer
c. a binary tree

Question 1.2

a. Given a processor for x86 programs, an interpreter for Scheme programs written in Python, a self-interpreter for Scheme, a self-interpreter for OCaml, an interpreter for OCaml programs written in Scheme, and a compiler from Python to x86 written in x86, can we execute an OCaml program?

   Why?

b. Given a processor for x86 programs, a compiler from OCaml to x86 written in Python, a Scheme interpreter written in x86, a self-interpreter for Python, and a compiler from Python to Scheme written in Scheme, and a chocolate bar, can we execute an OCaml program?

   Why?

c. Given a processor for x86 programs, a compiler from OCaml to Scheme written in Scheme, and an interpreter for Scheme programs written in x86, can we execute an OCaml program?

   Why?

Question 1.3

The OCaml library function, `Random.int`, when applied to a positive integer, returns a random integer that is strictly smaller than this positive integer and greater or equal to 0:

```
# Random.int 3;;
- : int = 0
# Random.int 3;;
- : int = 1
# Random.int 3;;
- : int = 1
# Random.int 3;;
- : int = 2
# Random.int 3;;
- : int = 0
# Random.int 3;;
- : int = 2
# Random.int 3;;
- : int = 2
#
```

Applying `Random.int` to a non-positive integer is undefined and so an exception is raised:

```
# Random.int 0;;
Exception: Invalid_argument "Random.int".
# Random.int (-1);;
Exception: Invalid_argument "Random.int".
#
```

Assuming that `is_even` denotes a function of type `int -> bool` that computes whether its argument is even,

a. what is the result of evaluating `is_even (2 * (Random.int 5))`?
   1. `true`, always
   2. `false`, always
   3. sometimes `true`, sometimes `false`, but an exception is never raised
   4. sometimes `true`, sometimes `false`, and sometimes no result because an exception is raised
   5. no result because an exception is raised, always
   6. we can't say, because applying `Random.int` might not terminate
b. what is the result of evaluating `is_even (Random.int (2 * 5))`?
   1. `true`, always
   2. `false`, always
   3. sometimes `true`, sometimes `false`, but an exception is never raised
   4. sometimes `true`, sometimes `false`, and sometimes no result because an exception is raised
   5. no result because an exception is raised, always
   6. we can't say, because applying `Random.int` might crash
c. what is the result of evaluating `is_even (2 * (Random.int 6))`?
   1. `true`, always
   2. `false`, always
   3. sometimes `true`, sometimes `false`, but an exception is never raised
   4. sometimes `true`, sometimes `false`, and sometimes no result because an exception is raised
   5. no result because an exception is raised, always
   6. we can't say, because applying `Random.int` returns a, you know, random result
d. what is the result of evaluating `is_even (Random.int (2 * 6))`?
   1. `true`, always
   2. `false`, always
   3. sometimes `true`, sometimes `false`, but an exception is never raised
   4. sometimes `true`, sometimes `false`, and sometimes no result because an exception is raised

5. no result because an exception is raised, always
6. we can’t say in general, because every call to random accelerates the half-life decay of our silicon processor, which is why our computers have gotten slower and slower during this course; fortunately, there is Moore’s law, but then we would need to buy a new computer, and this new computer would also become slower and slower because of all these calls to random; so all in all, we can’t say in general

e. what is the result of evaluating `Random.int (Random.int (6 * 9))`?
   1. a *really* random number, always
   2. sometimes a random number, sometimes no result because an exception is raised
   3. no result because an exception is raised, always
   4. no result because compound randomness is uncomputable
   5. a random answer to life, the universe, and everything

Briefly justify each of your answers.

Part 2

The goal of this part is to study the following alternative data type of binary trees, where the payload (an integer) is not in the leaves but in the nodes:

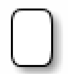
```
type binary_tree' =
  | Leaf'
  | Node' of binary_tree' * int * binary_tree';;
```

The associated induction principle reads as follows:

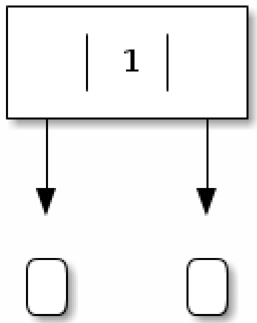
```
                          P(Leaf')    for all binary trees t1 and t2, for all integers n, P(t1) /\ P(t2) => P(Node' (t1, n, t2))
INDUCTION_BINARY_TREE'   ------------------------------------------------------------------------------------------------------
                          for all binary trees t, P(t)
```

For example:

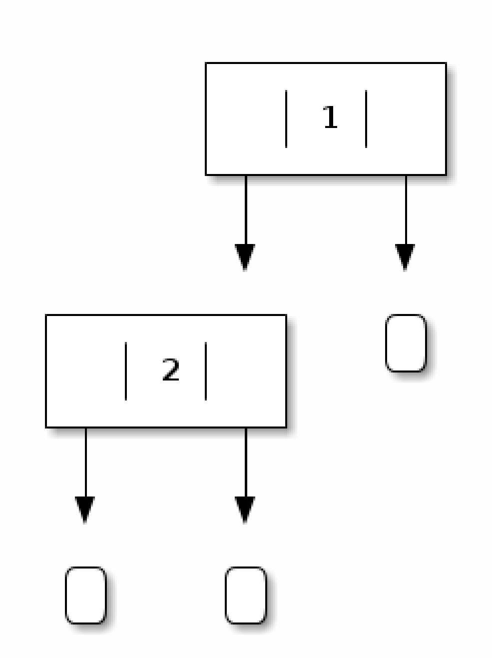
- The binary tree obtained by evaluating `Leaf'` is depicted as follows:



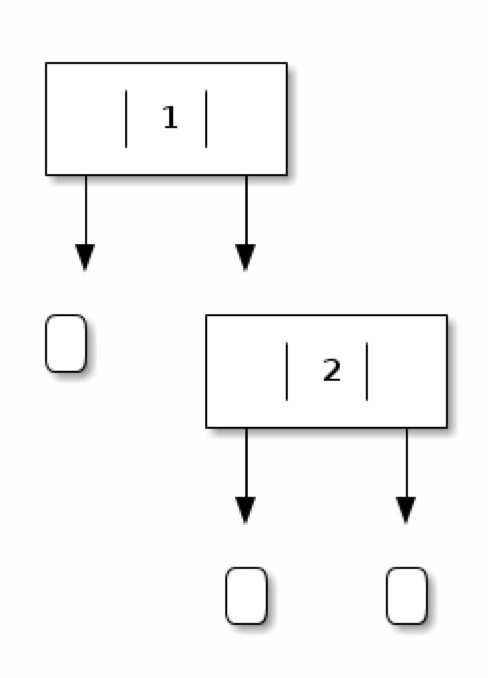
- The binary tree obtained by evaluating `Node' (Leaf', 1, Leaf')` is depicted as follows:



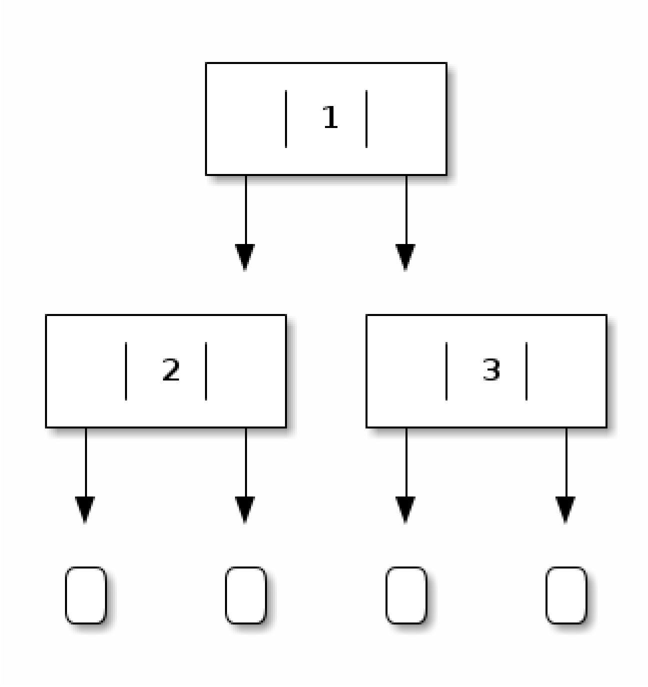
- The binary tree obtained by evaluating `Node' (Node' (Leaf', 2, Leaf'), 1, Leaf')` is depicted as follows:



- The binary tree obtained by evaluating `Node' (Leaf', 1, Node' (Leaf', 2, Leaf'))` is depicted as follows:



- The binary tree obtained by evaluating `Node' (Node' (Leaf', 2, Leaf'), 1, Node' (Leaf', 3, Leaf'))` is depicted as follows:



- etc.

Question 2.1

Implement an OCaml function that counts the number of leaves of a given binary tree.

Question 2.2

Implement an OCaml function that counts the number of nodes of a given binary tree.

Question 2.3

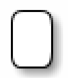
Is there a relation between the number of leaves and the number of nodes of the same binary tree? If so, state it and prove it.

Question 2.4

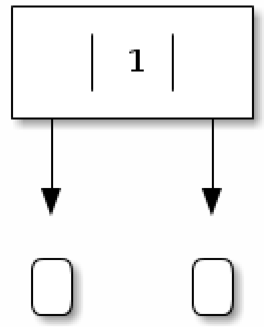
A binary tree is left-balanced if all its right subtrees are leaves.

For example:

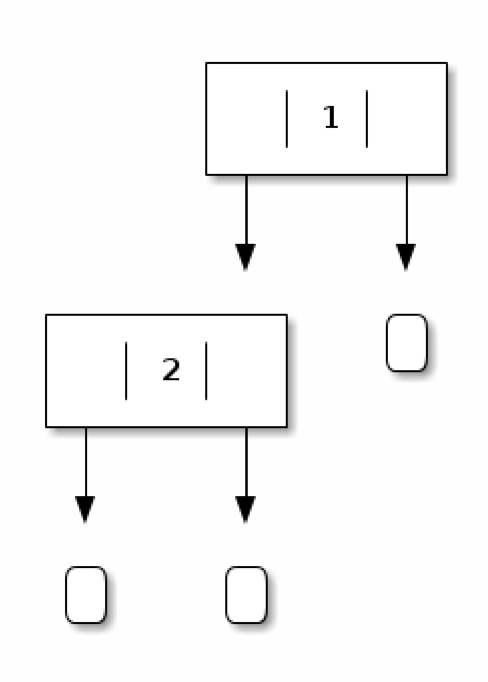
- The binary tree obtained by evaluating `Leaf'` is vacuously left-balanced:



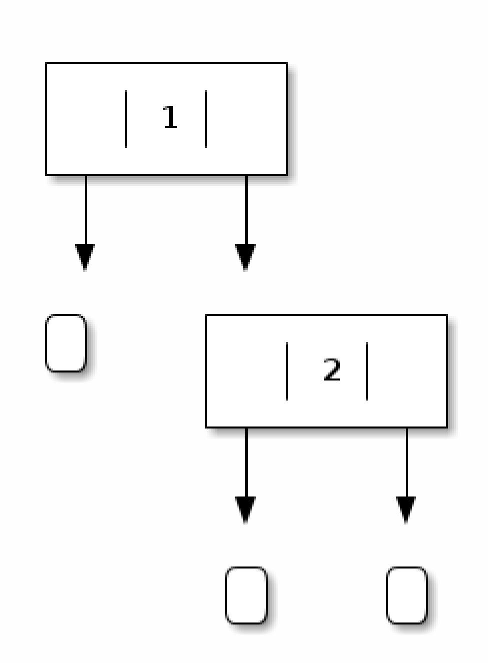
- The binary tree obtained by evaluating `Node' (Leaf', 1, Leaf')` is left-balanced:



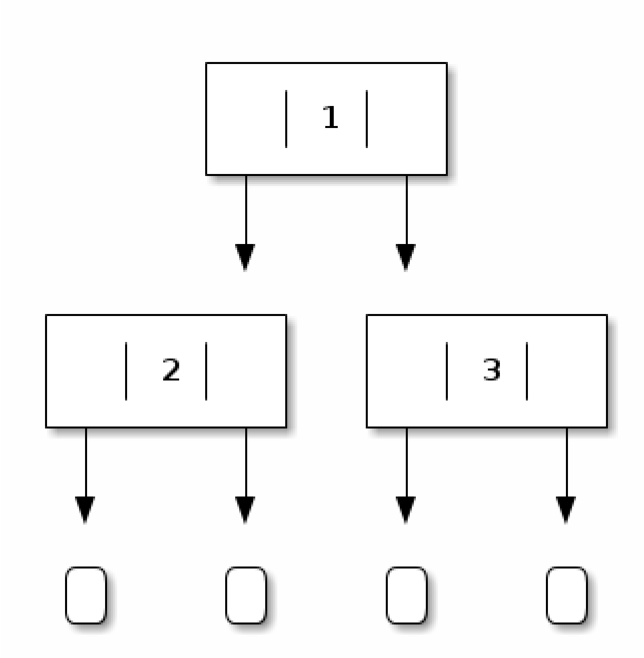
- The binary tree obtained by evaluating `Node' (Node' (Leaf', 2, Leaf'), 1, Leaf')` is left-balanced:



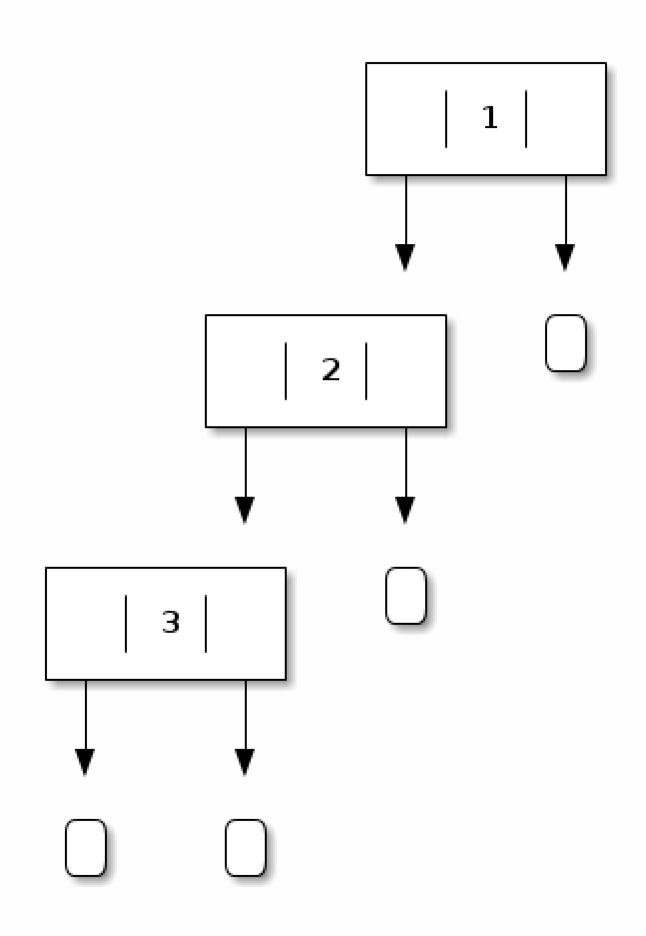
- The binary tree obtained by evaluating `Node' (Leaf', 1, Node' (Leaf', 2, Leaf'))` is not left-balanced:



- The binary tree obtained by evaluating `Node' (Node' (Leaf', 2, Leaf'), 1, Node' (Leaf', 3, Leaf'))` is not left-balanced:



- The binary tree obtained by evaluating `Node' (Node' (Node' (Leaf', 3, Leaf'), 2, Leaf'), 1, Leaf')` is left-balanced:



- etc.

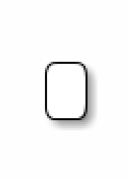
Implement an OCaml function `left_balanced` that tests whether a given binary tree is left-balanced.

Question 2.5

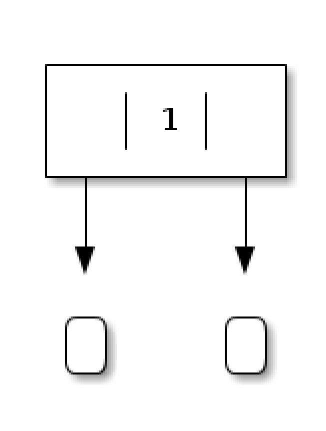
A binary tree is right-balanced if all its left subtrees are leaves.

For example:

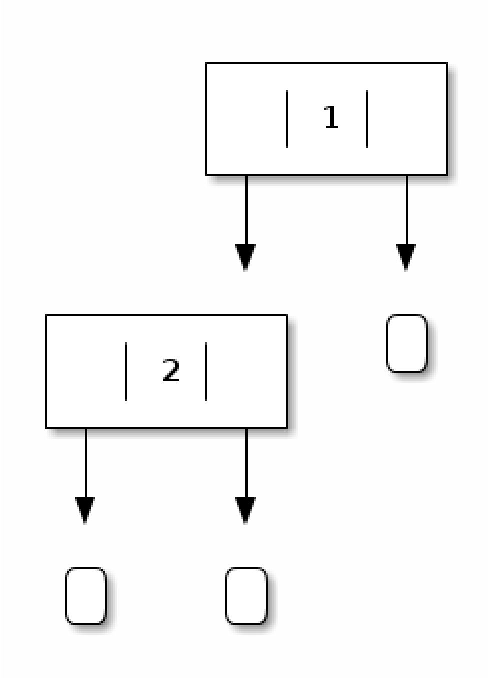
- The binary tree obtained by evaluating `Leaf'` is vacuously right-balanced:



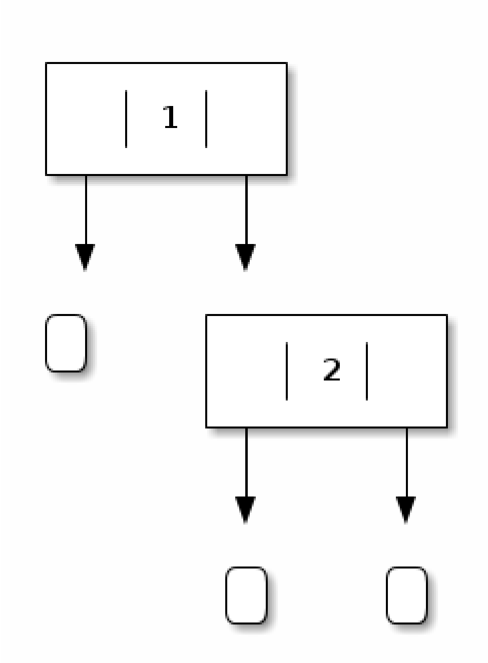
- The binary tree obtained by evaluating `Node' (Leaf', 1, Leaf')` is right-balanced:



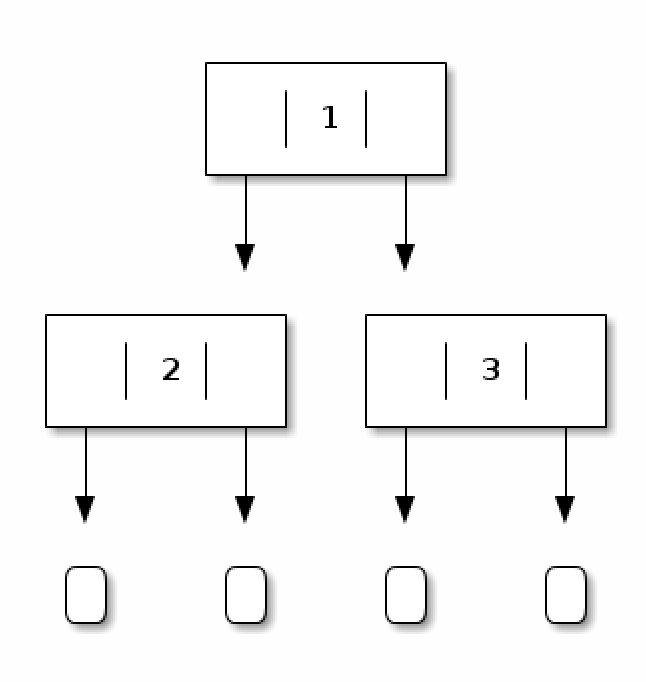
- The binary tree obtained by evaluating `Node' (Node' (Leaf', 2, Leaf'), 1, Leaf')` is not right-balanced:



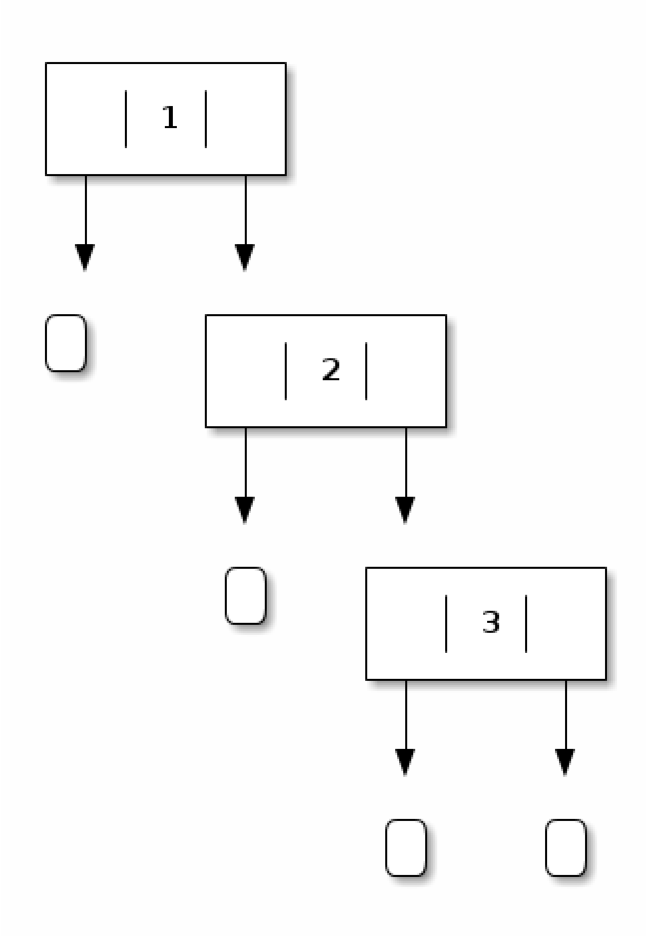
- The binary tree obtained by evaluating `Node' (Leaf', 1, Node' (Leaf', 2, Leaf'))` is right-balanced:



- The binary tree obtained by evaluating `Node' (Node' (Leaf', 2, Leaf'), 1, Node' (Leaf', 3, Leaf'))` is not right-balanced:



- The binary tree obtained by evaluating `Node' (Leaf', 1, Node' (Leaf', 2, Node' (Leaf', 3, Leaf')))` is right-balanced:



- etc.

Implement an OCaml function `right_balanced` that tests whether a given binary tree is right-balanced.

Question 2.6

How many trees are both left-balanced and right-balanced? If there are finitely many (all with the same integer, e.g., `0`, in their nodes), enumerate them.

Part 3

As an alternative to representing left-balanced binary trees as binary trees that satisfy the predicate `left_balanced`, let us implement the following dedicated data type, along with the conversion functions from it to `binary_tree'` and back:

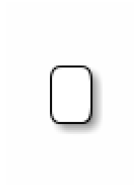
```
type left_binary_tree' =
  | Left_Leaf'
  | Left_Node' of left_binary_tree' * int;;

let rec embed_left_binary_tree'_into_binary_tree' t =
     (* embed_left_binary_tree'_into_binary_tree' : left_binary_tree' -> binary_tree' *)
  match t with
  | Left_Leaf' ->
     Leaf'
  | Left_Node' (t1, n) ->
     Node' (embed_left_binary_tree'_into_binary_tree' t1, n, Leaf');;

type option_left_binary_tree' =
  | Some_left_binary_tree' of left_binary_tree'
  | None_left_binary_tree';;

let rec project_binary_tree'_into_left_binary_tree' t =
     (* project_left_binary_tree'_into_binary_tree' : binary_tree' -> option_left_binary_tree' *)
  match t with
  | Leaf' ->
     Some_left_binary_tree' Left_Leaf'
  | Node' (t1, n, t2) ->
     match t2  with
     | Leaf' ->
        (match project_binary_tree'_into_left_binary_tree' t1 with
         | Some_left_binary_tree' t1' ->
            Some_left_binary_tree' (Left_Node' (t1', n))
         | None_left_binary_tree' ->
            None_left_binary_tree')
     | Node' _ ->
        None_left_binary_tree';;
```

The associated induction principle reads as follows:

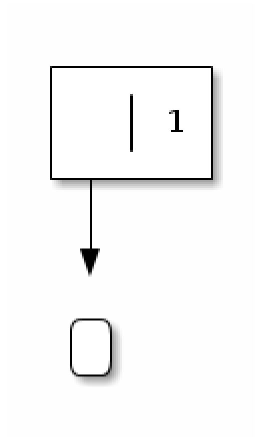
INDUCTION_LEFT_BINARY_TREE'	`L(Left_Leaf')`	`for all left-balanced binary trees t, for all integers n, L(t) => L(Left_Node' (t, n))`
	`for all left-balanced binary trees t, L(t)`	

For example:

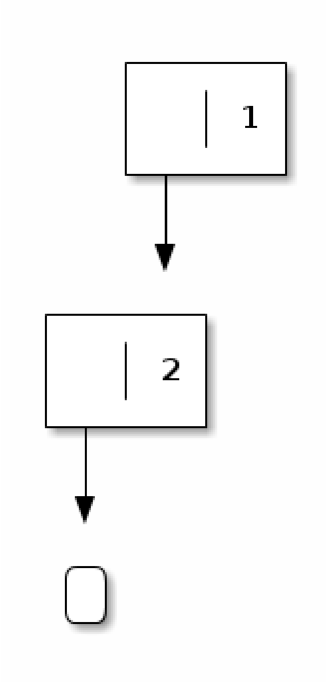
- The binary tree obtained by evaluating `Left_Leaf'` is depicted as follows:



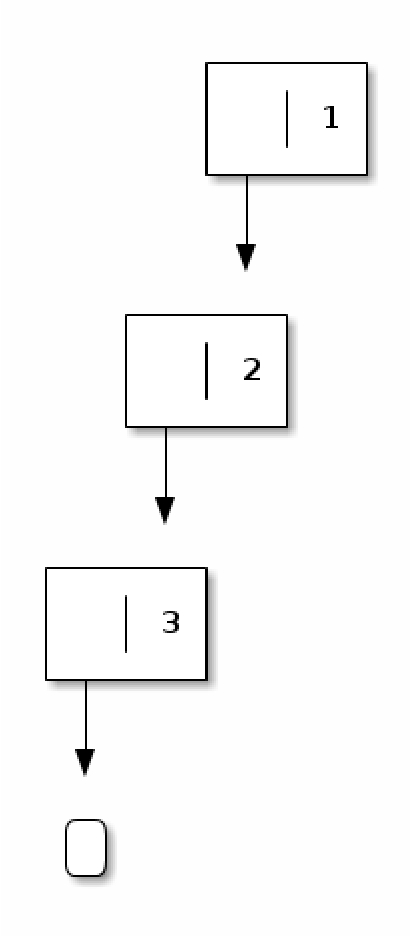
- The binary tree obtained by evaluating `Left_Node' (Left_Leaf', 1)` is depicted as follows:



- The binary tree obtained by evaluating `Left_Node' (Left_Node' (Left_Leaf', 2), 1)` is depicted as follows:



- The binary tree obtained by evaluating `Left_Node' (Left_Node' (Left_Node' (Left_Leaf', 3), 2), 1)` is depicted as follows:



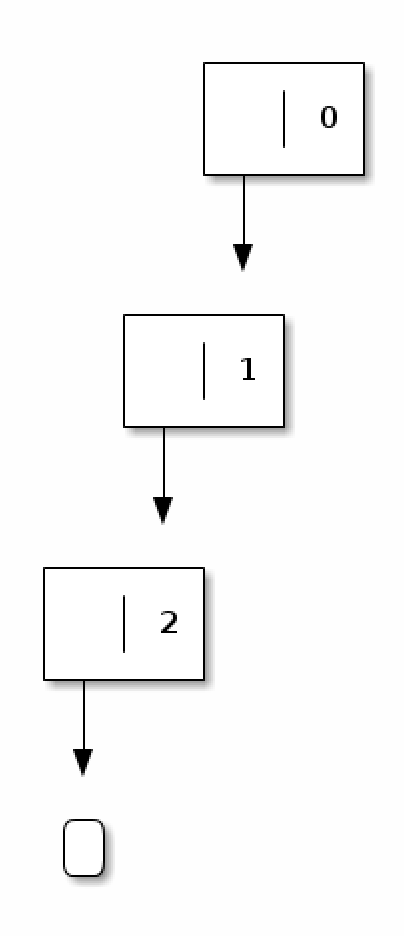
- etc.

Question 3.1

Why do we need an option type in the co-domain of `project_binary_tree'_into_left_binary_tree'`?

Question 3.2 (optional)

Implement an OCaml function `left_nth` that indexes a given left-tree at a given depth. For example, consider the following left-tree:



- indexing this tree at depth 0 yields 0,
- indexing this tree at depth 1 yields 1, and
- indexing this tree at depth 2 yields 2.

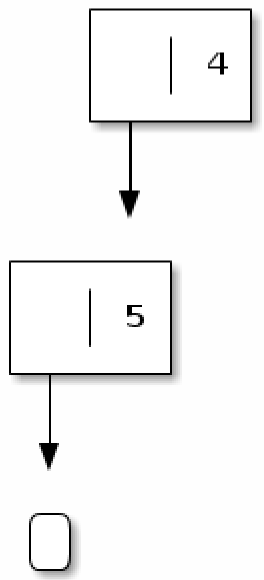
Is your indexing function partial or total?

Question 3.3

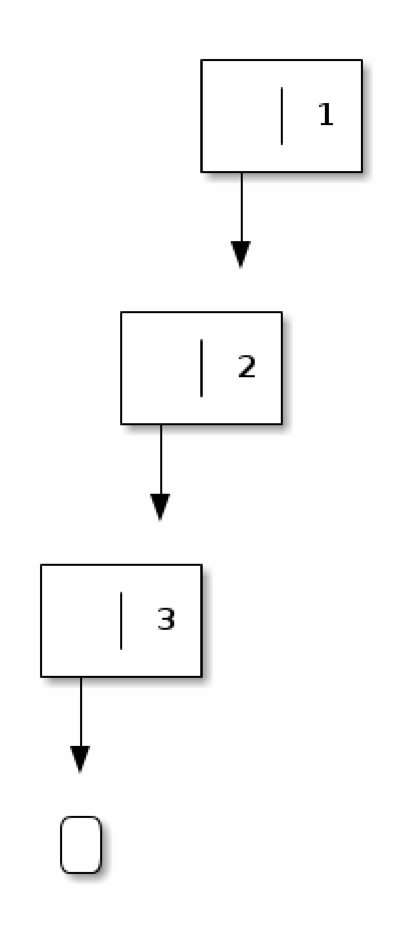
Implement an OCaml function `left_stitch` that stitches together two left-balanced binary trees and that satisfies the following unit tests:

```
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_Node' (Left_Leaf', 2), 1), 0))
   = Left_Node' (Left_Node' (Left_Node' (Left_Node' (Left_Node' (Left_Leaf', 4), 3), 2), 1), 0))
  &&
  (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', 4), 3))
             (Left_Node' (Left_Leaf', 2))
   = Left_Node' (Left_Node' (Left_Node' (Left_Leaf', 4), 3), 2))
  &&
  (candidate (Left_Node' (Left_Node' (Left_Leaf', 4), 3))
             (Left_Leaf')
   = Left_Node' (Left_Node' (Left_Leaf', 4), 3))
  (* etc. *);;
```

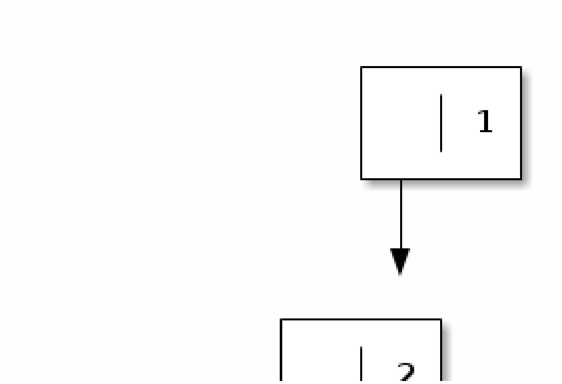
Pictorially, the two left-trees

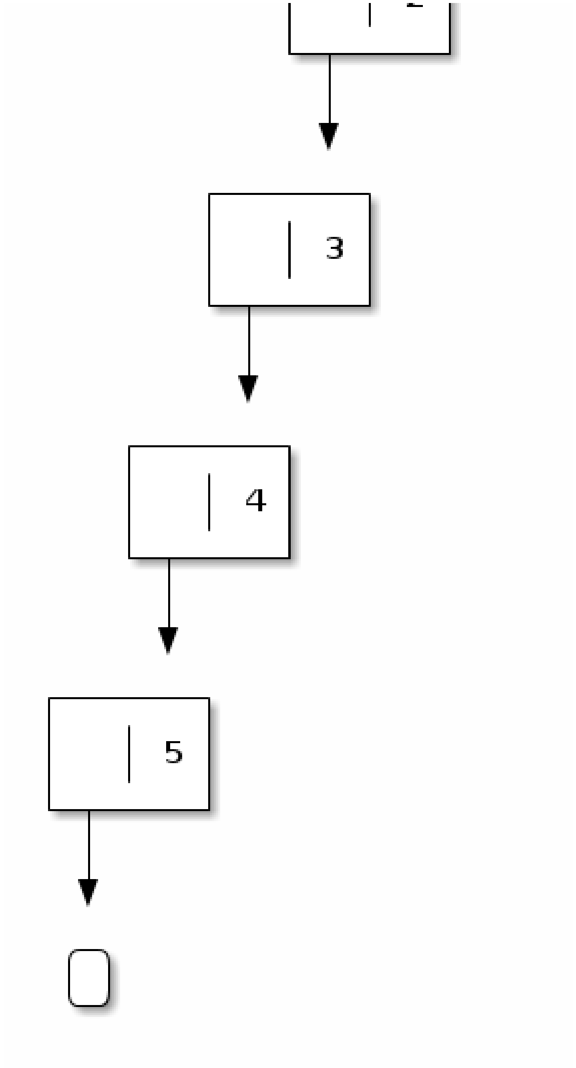


and



are stitched into the following left-tree:





Also, add this pictorial example as a new clause in `test_left_stitch candidate`.

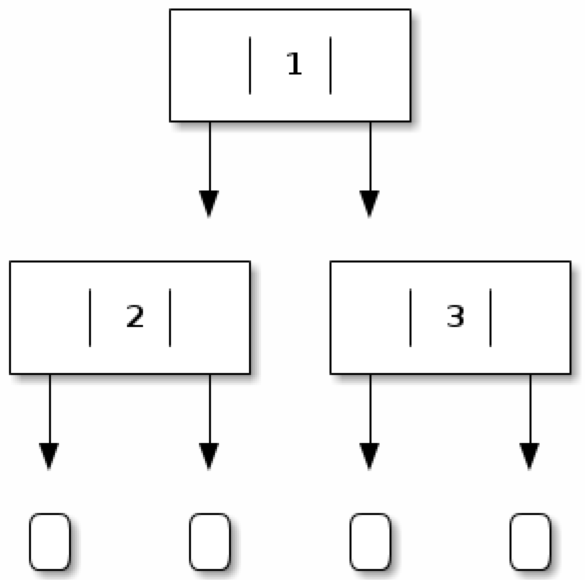
Question 3.4

Implement an OCaml function `left_rotate` that rotates a binary tree to the left and that satisfies the following unit tests:

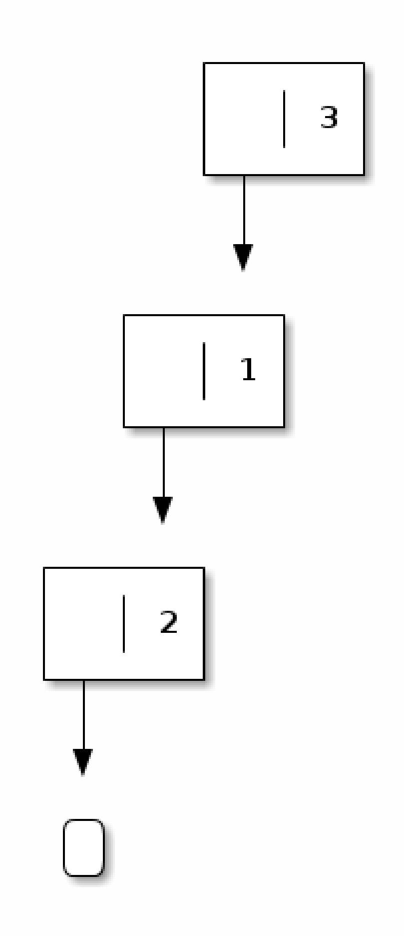
```
let test_left_rotate candidate =
 (* test_left_rotate : (binary_tree' -> left_binary_tree') -> bool *)
```

```
(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. *);;
```

Pictorially, the binary tree obtained by evaluating `Node' (Node' (Leaf', 2, Leaf'), 1, Node' (Leaf', 3, Leaf'))`, i.e.:



is rotated to the left into the left-tree obtained by evaluating `Left_Node' (Left_Node' (Left_Node' (Left_Leaf', 2), 1), 3)`, i.e.:

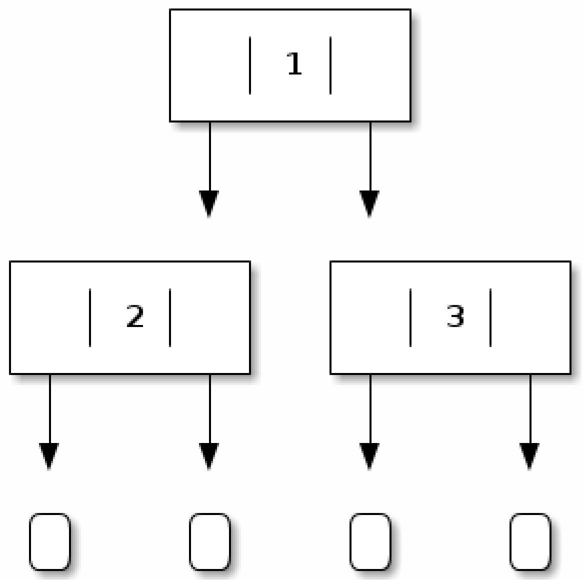


Question 3.5

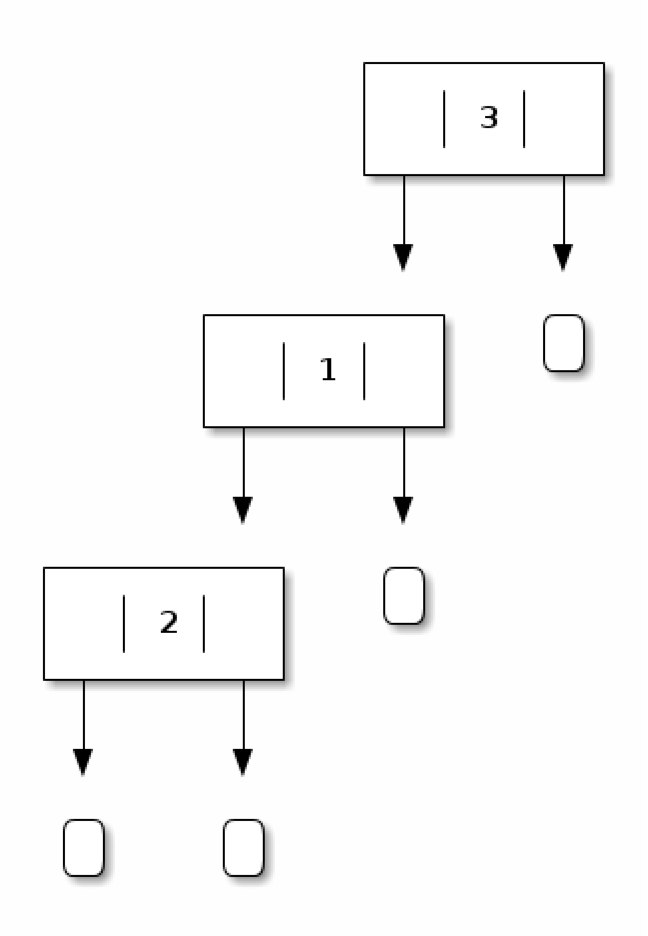
We are now in position to define a function that flattens a binary tree to the left:

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

For example, flattening the binary tree obtained by evaluating `Node' (Node' (Leaf', 2, Leaf'), 1, Node' (Leaf', 3, Leaf'))`, i.e.:



yields the binary tree obtained by evaluating `Node' (Node' (Node' (Leaf', 2, Leaf'), 1, Leaf'), 3, Leaf')`, i.e.:



Justify why `left_flatten` fits the bill and implement a unit-test function for it.

Part 4

As an alternative to representing right-balanced binary trees as binary trees that satisfy the predicate `right_balanced`, let us implement the following dedicated data type, along with the conversion functions from it to `binary_tree'` and back:

```
type right_binary_tree' =
  | Right_Leaf'
  | Right_Node' of int * right_binary_tree';;
```

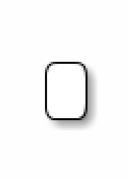
```
let rec embed_right_binary_tree'_into_binary_tree' t =
     (* embed_right_binary_tree'_into_binary_tree' : right_binary_tree' -> binary_tree' *)
  match t with
  | Right_Leaf' ->
     Leaf'
  | Right_Node' (n, t1) ->
     Node' (Leaf', n, embed_right_binary_tree'_into_binary_tree' t1);;

type option_right_binary_tree' =
  | Some_right_binary_tree' of right_binary_tree'
  | None_right_binary_tree';;

let rec project_binary_tree'_into_right_binary_tree' t =
     (* project_binary_tree'_into_right_binary_tree' : binary_tree' -> option_right_binary_tree' *)
  match t with
  | Leaf' ->
     Some_right_binary_tree' Right_Leaf'
  | Node' (t1, n, t2) ->
     match t1  with
     | Leaf' ->
        (match project_binary_tree'_into_right_binary_tree' t2 with
         | Some_right_binary_tree' t2' ->
            Some_right_binary_tree' (Right_Node' (n, t2'))
         | None_right_binary_tree' ->
            None_right_binary_tree')
     | Node' _ ->
        None_right_binary_tree';;
```

The associated induction principle reads as follows:

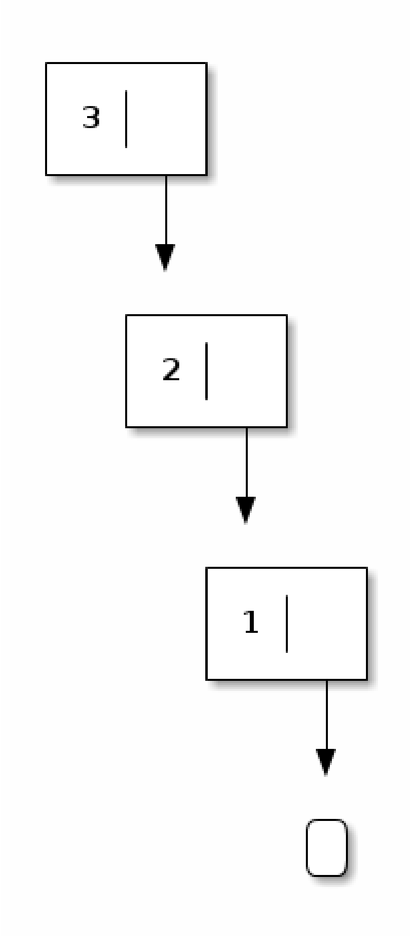
INDUCTION_RIGHT_BINARY_TREE'	`R(Right_Leaf')`	`for all right-balanced binary trees t, for all integers n, R(t) => R(Right_Node' (n, t))`
	`for all right-balanced binary trees t, R(t)`	

For example:

- The right-tree obtained by evaluating `Right_Leaf'` is depicted as follows:



- The right-tree obtained by evaluating `Right_Node' (3, Right_Node' (2, Right_Node' (1, Right_Leaf')))` is depicted as follows:

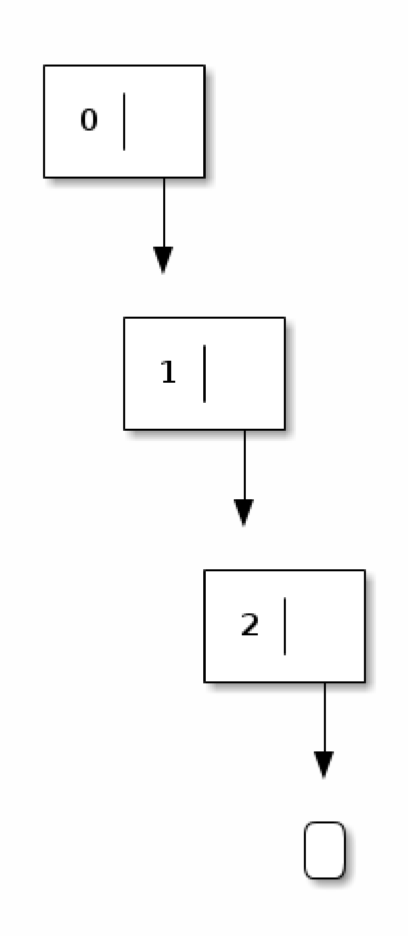


Question 4.1

Why do we need an option type in the co-domain of `project_binary_tree'_into_right_binary_tree'`?

Question 4.2 (optional)

Implement an OCaml function `right_nth` that indexes a given right-tree at a given depth. For example, consider the following right-tree:



- indexing this tree at depth 0 yields 0,
- indexing this tree at depth 1 yields 1, and
- indexing this tree at depth 2 yields 2.

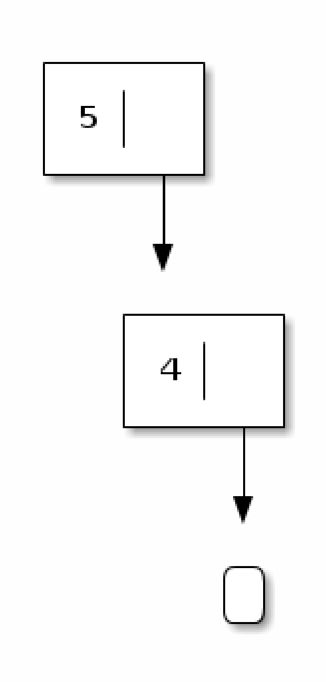
Is your indexing function partial or total?

Question 4.3

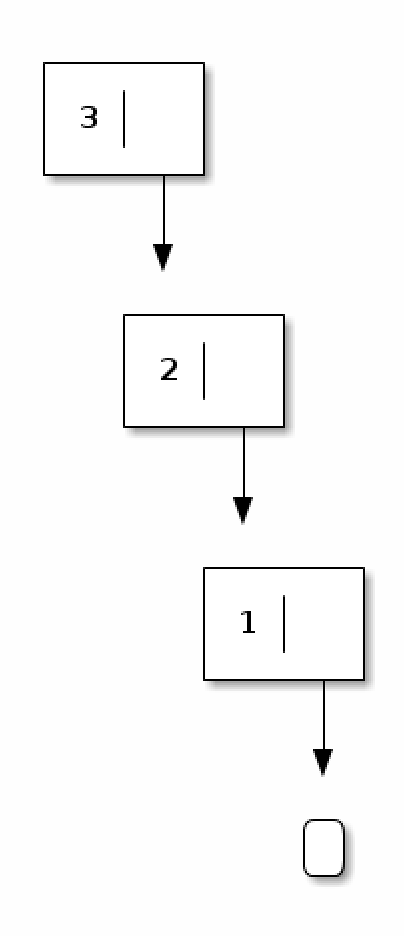
Implement an OCaml function `right_stitch` that stitches together two right-balanced binary trees and that satisfies the following unit tests:

```
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')))))
  (* etc. *);;
```

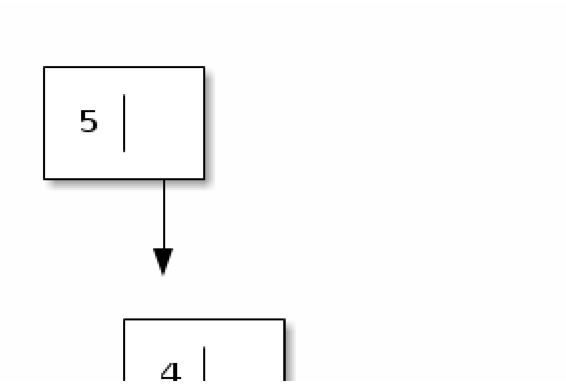
Pictorially, the two right-trees

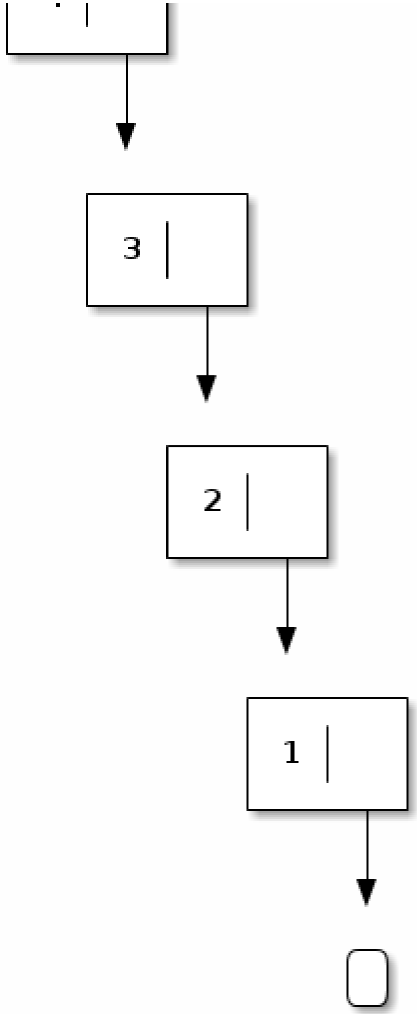


and



are stitched into the following right-tree:





Also, add this pictorial example as a new clause in `test_right_stitch candidate`.

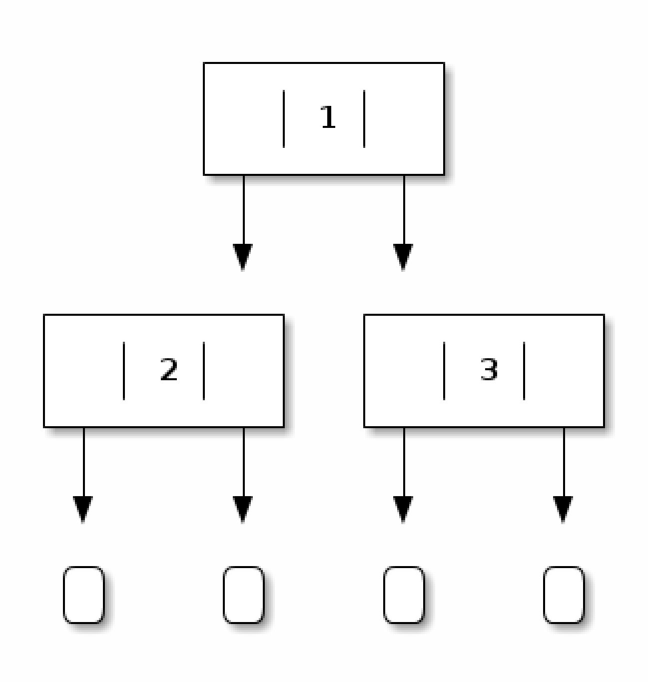
Question 4.4

Implement an OCaml function `right_rotate` that rotates a binary tree to the right and that satisfies the following unit tests:

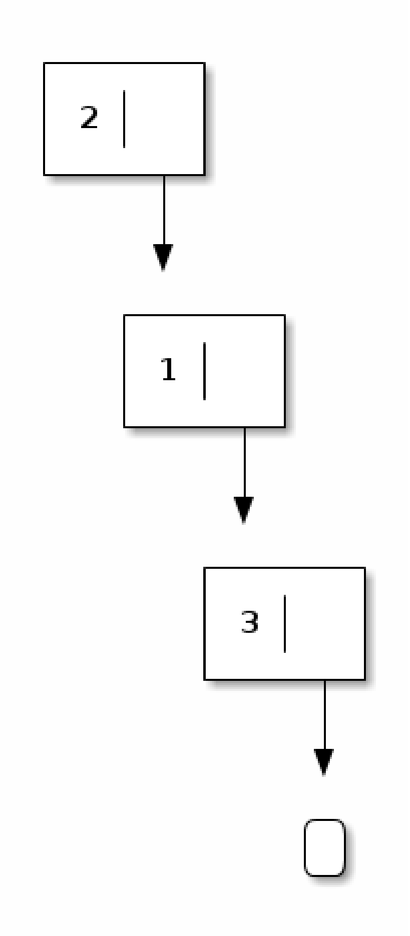
```
let test_right_rotate candidate =
 (* test_right_rotate : (binary_tree' -> right_binary_tree') -> bool *)
```

```
 (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. *);;
```

Pictorially, the binary tree obtained by evaluating `Node' (Node' (Leaf', 2, Leaf'), 1, Node' (Leaf', 3, Leaf'))`, i.e.:



is rotated to the right into the right-tree obtained by evaluating `Right_Node' (2, Right_Node' (1, Right_Node' (3, Right_Leaf')))`, i.e.:

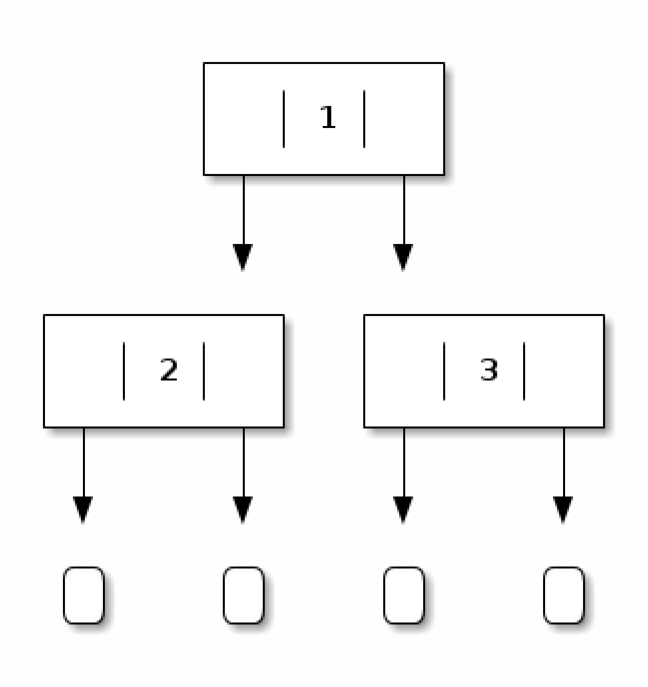


Question 4.5

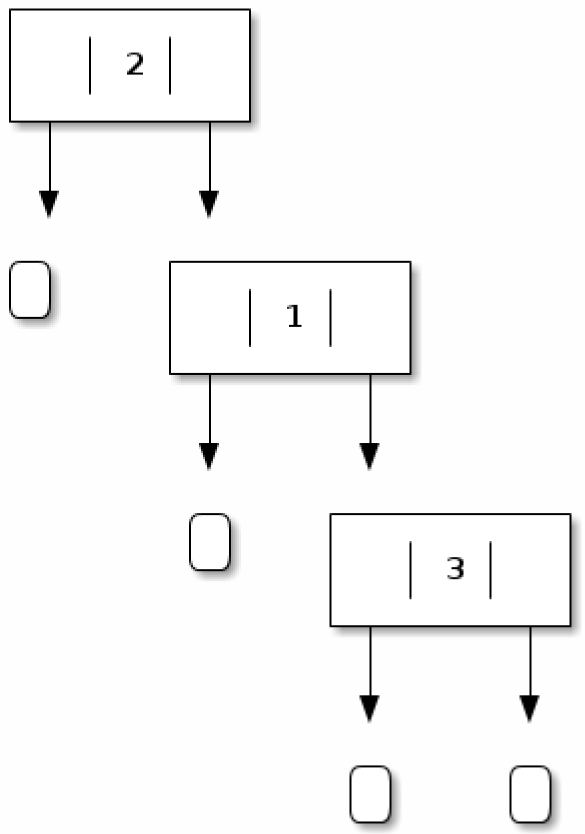
We are now in position to define a function that flattens a binary tree to the right:

```
let right_flatten t =
 (* right_flatten : binary_tree' -> binary_tree' *)
  embed_right_binary_tree'_into_binary_tree' (right_rotate t);;
```

For example, flattening the binary tree obtained by evaluating `Node' (Node' (Leaf', 2, Leaf'), 1, Node' (Leaf', 3, Leaf'))`, i.e.:



yields the binary tree obtained by evaluating `Node' (Leaf', 2, Node' (Leaf', 1, Node' (Leaf', 3, Leaf')))`, i.e.:



Justify why `right_flatten` fits the bill and implement a unit-test function for it.

Part 5

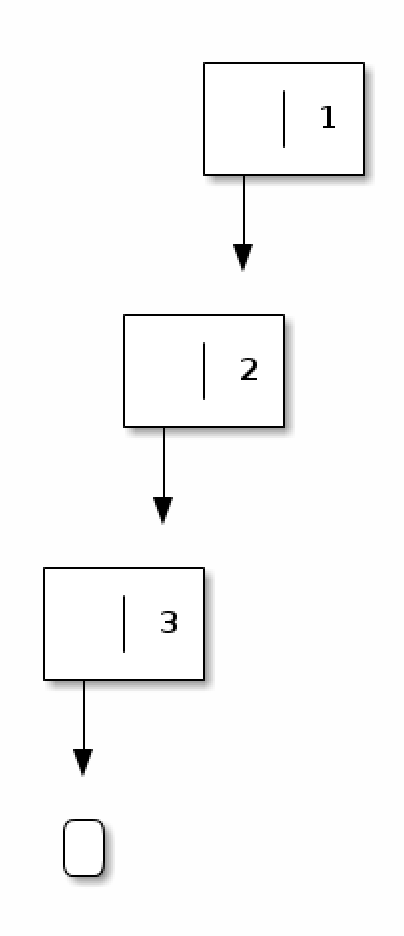
The goal of this part is to study how to rebalance left-trees into right-trees and vice-versa.

Question 5.1

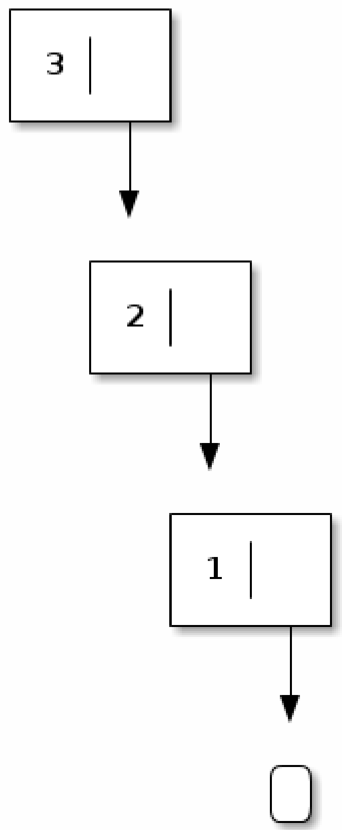
Implement an OCaml function that maps a left-balanced tree to a right-balanced tree and that satisfies the following unit tests:

```
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. *);;
```

For example, the left-tree obtained by evaluating `Left_Node' (Left_Node' (Left_Node' (Left_Leaf', 3), 2), 1)`:



is rebalanced into the right-tree obtained by evaluating `Right_Node' (3, Right_Node' (2, Right_Node' (1, Right_Leaf')))`:



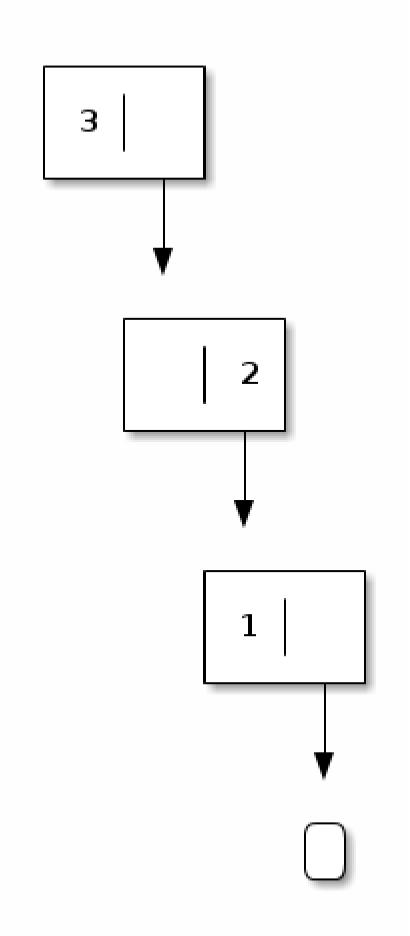
Question 5.2

Implement an OCaml function that maps a right-balanced tree to a left-balanced tree and that satisfies the following unit tests:

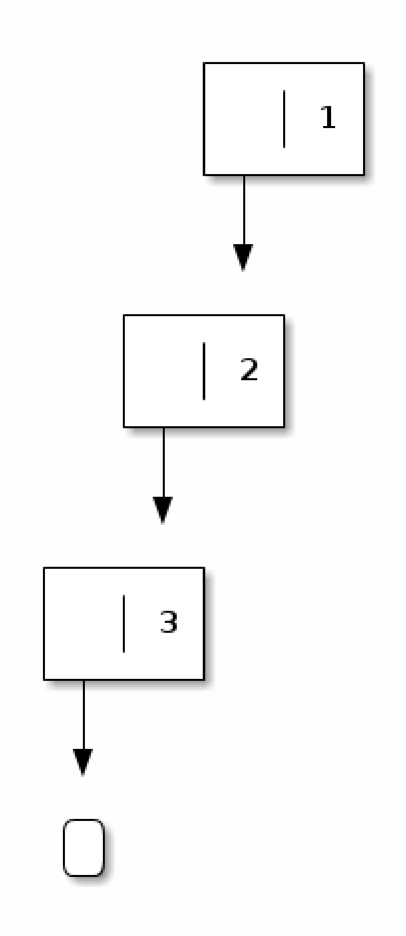
```
let test_right_to_left candidate =
 (* test_right_to_left : (right_binary_tree' -> left_binary_tree') -> bool *)
  (candidate Right_Leaf'
   = Left_Leaf')
  &&
  (candidate (Right_Node' (1, Right_Leaf'))
   = Left_Node' (Left_Leaf', 1))
  &&
```

```
(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. *);;
```

For example, the right-tree obtained by evaluating `Right_Node' (3, Right_Node' (2, Right_Node' (1, Right_Leaf')))`:



is rebalanced into the left-tree obtained by evaluating `Left_Node' (Left_Node' (Left_Node' (Left_Leaf', 3), 2), 1)`:



Question 5.3

Characterize the following two OCaml functions:

```
let foo t
  = left_to_right (right_to_left t);;

let bar t
  = right_to_left (left_to_right t);;
```

For example, what are their types?

Question 5.4

Consider the following 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)));;
```

Does your implementation pass these tests? Why?

The individual report

Your report should include

- a front page with title, name, student number, and date,
- the names of the other members of your group,
- a second page with a table of contents, and
- from the third page and onwards,
  - an introduction,
  - a series of sections and subsections reflecting the structure of the project, and
  - a conclusion where you assess what you did and reflect on how you did it.

Of course you should not paraphrase the code, because what is the point of that.

Pages should be numbered.

An inspiring (and not necessarily humorous, just on topic) quote or three would be welcome.

Resources

- The OCaml code for the present midterm project (latest version: 23 Sep 2017).

Version

Added a handful of clauses in the definition of `test_left_stitch` [29 Sep 2017]

Fixed a typo in Question 2.5, thanks to Shi Tingsheng’s eagle eye [28 Sep 2017]

Fixed a typo in Question 4.5, thanks to Shardul Sapkota’s eagle eye [27 Sep 2017]

Tightened the narrative, thanks to Kira Kutscher’s comments [25 Sep 2017]

Fixed a typo in the notation for `Random.int`, thanks to Chong Woon Han’s eagle eye [24 Sep 2017]

Fine-tuned the narrative [24 Sep 2017]

Created [23 Sep 2017]