| **List, Caution: Exploring functions involving lists**  **Yale-NUS College**  **Week 10**  **7 April 2022** |
| --- |

**Table of Contents**

[**Introduction**](#_8voy1km2delu) **2**

[**Exercise 01: Implementing Comparison functions**](#_n974d41cu0my) **2**

[**Exercise 02: list\_length**](#_lyjegjm0oeqe) **5**

[**Exercise 03: list\_append**](#_7yhyohcnvc4y) **6**

[**Exercise 04: list\_map**](#_9nqygz56esg9) **7**

[**Exercise 05: list\_reverse**](#_5trcze21wu1a) **7**

[**Exercise 06 [Exercise 02 from Week 10: about right-to-left lists (i.e., tsils)]**](#_1hwlm3lsock4) **8**

[**Exercise 07 [Exercise 06 from Week 08: about list\_map]**](#_jn7wzgy477hj) **19**

[**Exercise 08 [Exercise 09 from Week 08: about list\_mapi]**](#_4h4r3cal279m) **20**

[**Exercise 09 [Exercise 02 from Week 09: about function equivalences]**](#_s3cjwhs2wz0v) **23**

[**Exercise 10 [Exercise 10 from Week 09: about list\_fold\_right]**](#_6d7gf3yinddz) **26**

[**Exercise 11 [Exercise 16 from Week 09: about list\_fold\_left]**](#_1puwciqijx2i) **28**

[**Conclusion**](#_fgljknfjvo9) **31**

# Introduction

This week, we explore all about lists.

But first, we practise implementing two comparison functions using each other, allowing us to try match-clauses. Then, we implement four key functions for lists: (i) list\_length to compute the length of a list, (ii) list\_append to concatenate two lists, (iii) list\_map to map a given function over the elements of a given list and that returns the list of the results, and (iv) list\_reverse to reverse a list. We have found these four functions to be analogous to corresponding functions for strings instead of lists.

Next, we define a new type -- tsils, which are essentially the reverse of a list, in that new elements are always added on the right, not the left. We practise implementing the same four aforementioned functions but for tsils instead of lists. Additionally, we use list\_map on a function and implement and use list\_mapi.

Further, we consider the equivalence of functions involving a different combination of the four aforementioned key list functions, and use commuting diagrams to illustrate equivalence.

We then are introduced to the analogues of nat\_fold\_right and nat\_fold\_left — list\_fold\_right and list\_fold\_left. These generalised functions for recursive functions for lists enable us to implement a wide range of familiar functions like finding the list length, reversing lists, appending lists, and returning the same list, etc. using the same generalised function instantiated with specific nil\_case and base\_case. In list\_fold\_left, we also abstract the accumulator, an object that allows us to construct results as we go, without having to wait for the current recursive call to be completed, reducing time and space- complexity of the functions we write. This gives us another layer or toolkit for writing better recursive functions. It makes us wonder if accumulators can be used in our recursive functions for previous data types, like strings and integers.

Finally, we take a brief look at binary trees by implementing a unit test to test an interesting relationship between leaves and nodes.

# Exercise 01: Implementing Comparison functions

Introduction

1. *Implement make\_comparison in terms of compare.*
2. *Implement compare in terms of make\_comparison.*

In the lecture note, we compared two ordered values v1 and v2 by distinguishing the following three cases:

1. v1 < v2
2. v1 = v2
3. v1 > v2

And we have defined a sum type that accounts for this comparison to carry out the task. In this exercise, we make use of the OCaml built-in function compare to implement the same make\_comparison function as well as compare\_make\_comparison function that is functionally equivalent to the compare function using make\_comparison function.

| **type** comparison = Lt | Eq | Gt (\* Make Comp \*) **let** make\_comparison v1 v2 =  (\* make\_comparison : 'a -> 'a -> comparison \*)  **if** v1 < v2 **then** Lt **else** **if** v1 = v2 **then** Eq **else** Gt;; |
| --- |

Solution

First, we implement unit-test functions.

| **let** test\_compare candidate =  **let** b0 = (candidate 3 4 = ~-1)  **and** b1 = (candidate 4 4 = 0)  **and** b2 = (candidate 0 0 = 0)  **and** b3 = (candidate ~-1 6 = ~-1)  **and** b4 = (candidate 8 4 = 1)  **in** b0 && b1 && b2 && b3 && b4;;   **let** test\_make\_comparison candidate =  **let** b0 = (candidate 3 4 = Lt)  **and** b1 = (candidate 4 4 = Eq)  **and** b2 = (candidate 0 0 = Eq)  **and** b3 = (candidate ~-1 6 = Lt)  **and** b4 = (candidate 8 4 = Gt)  **in** b0 && b1 && b2 && b3 && b4;; |
| --- |

1. *Implement make\_comparison in terms of compare.*

The OCaml built-in compare function will, given v1 and v2, return -1 if v1 < v2, 0 if v1 = v2, or 1 if v1 > v2. We use the match-with expression to get the desired output. Though the compare function would only return -1, 0, or 1, we let the last condition be \_ to avoid the warning like the following:

| Warning 8 [partial-**match**]: this pattern-matching is not exhaustive. Here is an example **of** a case that is not matched: 2 |
| --- |

| **let** make\_comparison\_compare v1 v2 =  **match** compare v1 v2 **with**  | -1 -> Lt  | 0 -> Eq  | \_ -> Gt;; |
| --- |

*b. Implement compare in terms of make\_comparison*

In the same manner as the previous part, we implement the compare function.

| **let** compare\_make\_comparison v1 v2 =  **match** make\_comparison v1 v2 **with**  | Lt ->  ~-1  | Eq ->  0  | Gt ->  1;; |
| --- |

We test the functions using the unit test functions.

| **let** () = **assert**(test\_compare compare) **let** () = **assert**(test\_compare compare\_make\_comparison) **let** () = **assert**(test\_make\_comparison make\_comparison) **let** () = **assert**(test\_make\_comparison make\_comparison\_compare) |
| --- |

It passed the unit tests.

Conclusion

In this exercise, we used a custom type that informs of the result of comparing 2 values. In this context, the grammar for the comparison type ( Lt, Eq and Gt ) are used to tell us if the first value is lesser, equal or greater than the second value.

It is useful to note that Lt, Eq and Gt only gain this meaning because this is how they are used in make\_comparison. We could just as easily use Lt, Eq and Gt to imply something entirely different. Hypothetically, for example, we could define Lt as the output of 1 + 1, Eq as the output of 3 + 3, and Gt as the output of any other expression.

In this exercise, we also gained more experience with indirect implementations of functions. To illustrate, we implemented make\_comparison directly using if-statements, but we also implemented it using compare by matching each int output of compare to an appropriate comparison output.

# 

# 

# Exercise 02: list\_length

*Implement a polymorphic function, list\_length : 'a list -> int, that computes the length of a list.*

Solution

* Week 8 notes has the [list\_length\_v0](https://delimited-continuation.github.io/YSC1212/2021-2022_Sem2/week-08_polymorphic-lists.html#example-computing-the-length-of-a-list) function:

Given the unit-test

| **let** test\_list\_length\_int candidate =  *(\* the base case: \*)*  **let** bc = (candidate [] = 0)  *(\* instance of the induction step: \*)*  **and** is = (**let** n = **Random**.int 1000  **in** **let** ns = atoi\_v0 n  **in** candidate (n :: ns) = succ (candidate ns))  *(\* a few handpicked lists: \*)*  **and** b1 = (1 = candidate [1])  **and** b2 = (2 = candidate [2; 1])  **and** b3 = (3 = candidate [3; 2; 1])  **and** b4 = (4 = candidate [4; 3; 2; 1])  *(\* a few automatically generated lists: \*)*  **and** b5 = (candidate (atoi\_v0 5) = 5)  **and** b6 = (candidate (atoi\_v0 10) = 10)  **and** b7 = (candidate (atoi\_v0 100) = 100)  *(\* etc. \*)*  **in** bc && is && b1 && b2 && b3 && b4 && b5 && b6 && b7;; |
| --- |

The function given in the lecture notes, Week 08 “Polymorphic lists in OCaml” is

| # **let** list\_length\_v3 vs\_given =  *(\* list\_length\_v3 : 'a list -> int \*)*  **let** **rec** visit vs =  **match** vs **with**  | [] ->  0  | v :: vs' ->  succ (visit vs')  **in** visit vs\_given;; **val** list\_length\_v3 : 'a list -> int = <**fun**> # |
| --- |

Which passes the unit test:

| # **let** () = **assert**(test\_list\_length\_int list\_length\_v3) # |
| --- |

# Exercise 03: list\_append

*Implement a polymorphic function, list\_append : 'a list -> 'a list -> 'a list, that concatenates two lists*

Solution

* Week 8 notes, under the section Polymorphic lists in OCaml, continued has [list\_append](https://delimited-continuation.github.io/YSC1212/2021-2022_Sem2/week-08_polymorphic-lists-continued.html#example-concatenating-two-lists):

Given the unit test in the lecture notes,

| **let** test\_list\_append\_int candidate =  *(\* two handpicked lists: \*)*  **let** b0 = (candidate [1; 2; 3] [4; 5] = [1; 2; 3; 4; 5])  *(\* an instance of the base case: \*)*  **and** b1 = (candidate [] [1; 0] = [1; 0])  *(\* successive instances of the induction step: \*)*  **and** b2 = (candidate [2] [1; 0] = [2; 1; 0])  **and** b3 = (candidate [3; 2] [1; 0] = [3; 2; 1; 0])  **and** b4 = (candidate [4; 3; 2] [1; 0] = [4; 3; 2; 1; 0])  *(\* other handpicked lists: \*)*  **and** b5 = (candidate [10; 20] [] = [10; 20])  *(\* etc. \*)*  **in** b0 && b1 && b2 && b3 && b4 && b5;; |
| --- |

The function given is:

| # **let** list\_append\_v1 xs\_given ys\_given =  **let** **rec** visit xs =  **match** xs **with**  | [] ->  ys\_given  | x :: xs' ->  x :: visit xs'  **in** visit xs\_given;; **val** list\_append\_v1 : 'a list -> 'a list -> 'a list = <**fun**> # |
| --- |

And this passes the unit test:

| # **let** () = **assert**(test\_list\_append\_int list\_append\_v1) # |
| --- |

# Exercise 04: list\_map

*Implement a polymorphic function, list\_map : ('a -> 'b) -> 'a list -> 'b list, that maps a given function over the elements of a given list and that returns the list of the results.*

Solution

* Week 8 notes has [list\_map](https://delimited-continuation.github.io/YSC1212/2021-2022_Sem2/week-08_list-map.html#generalization-the-map-function-for-lists).

| **# let** test\_list\_map candidate =  **let** b0 = (candidate succ [3; 2; 1; 0] = [4; 3; 2; 1])  **and** b1 = (candidate pred [2; 1; 0] = [1; 0; ~-1])  **and** b2 = (candidate (**fun** n -> n \* n) [3; 2; 1] = [9; 4; 1])  **in** b0 && b1 && b2;; |
| --- |

The function given is:

| # **let** list\_map f xs\_given =  *(\* list\_map : ('a -> 'b) -> 'a list -> 'b list \*)*  **let** **rec** visit xs =  **match** xs **with**  | [] ->  []  | x :: xs' ->  **let** ih = visit xs'  **in** f x :: ih  **in** visit xs\_given;; **val** list\_map : ('a -> 'b) -> 'a list -> 'b list = <**fun**> # |
| --- |

# 

| **# let** () = **assert**(test\_list\_map list\_map) |
| --- |

It passes the unit tests.

# 

# Exercise 05: list\_reverse

*Implement a polymorphic function, list\_reverse : 'a list -> 'a list, that reverses a list*

Solution:

* Week 9 notes, Reversing lists has 4 versions of list\_reverse. Here is [version 1](https://delimited-continuation.github.io/YSC1212/2021-2022_Sem2/week-09_list-reversal.html):

Unit test given:

| **let** test\_reverse candidate =  **let** b0 = (candidate [] = [])  **and** b1 = (candidate [0] = [0])  **and** b2 = (candidate [1; 0] = [0; 1])  **and** b3 = (candidate [2; 1; 0] = [0; 1; 2])  **and** b4 = (**let** ns = atoi 100  **in** candidate (candidate ns) = ns)  *(\* etc. \*)*  **in** b0 && b1 && b2 && b3 && b4;; |
| --- |

The function is:

| # **let** reverse\_v1 vs\_init =  **let** **rec** visit vs =  **if** vs = []  **then** []  **else** **let** ih = visit (**List**.tl vs)  **in** **List**.append ih [**List**.hd vs]  **in** visit vs\_init;;  **val** reverse\_v1 : 'a list -> 'a list = <**fun**> # |
| --- |

Which passes the unit test:

| # test\_reverse reverse\_v1;; - : bool = true # |
| --- |

# Exercise 06 [[*Exercise 02*](https://delimited-continuation.github.io/YSC1212/2021-2022_Sem2/week-10_declaring-new-types-in-ocaml.html#exercise-right-to-left-polymorphic-lists) from Week 10: about right-to-left lists (i.e., tsils)]

Introduction

In this exercise, we play around with defining our own types in OCaml. Specifically, we use a type called tsil, which is a reverse of a list in that new elements are always added on the right, not the left. We begin by defining the type tsil as follows:

| **type** 'a tsil = Lin | Snoc **of** 'a tsil \* 'a;; |
| --- |

We proceed to define a few useful functions for tsil, such as tsil\_length, tsil\_of\_list and so on.

Solution

Part a: Implementing tsil\_length.

Here, we implement a function that computes the length of a tsil. Here is a unit test:

| **let** test\_tsil\_length candidate =  **let** n1 = Random.int 5 **and** n2 = Random.int 1000  **in** **let** b0 = (candidate Lin = 0)  **and** b1 = (candidate (Snoc(Lin, 3)) = 1)  **and** b2 = (candidate (Snoc(Snoc(Lin, 3), 5)) = 2)  **and** b3 = (candidate (Snoc(Snoc(Snoc(Lin, n1), n2), 5)) = 3)  **in** b0 && b1 && b2 && b3;; |
| --- |

Our solution uses an accumulator. For each recursive call where one element is taken off a tsil, we add one to the accumulator. Therefore, the accumulator is initialized with 0. We stop once the tsil is deconstructed to a Lin, and check the accumulator at that point. Our implementation follows.

| **let** tsil\_length ts\_given =  **let** **rec** visit ts a =  **match** ts **with**  | Lin ->  a  | Snoc(c, d) ->  visit c (succ a)  **in** visit ts\_given 0;; |
| --- |

It also passes its unit test:

| # test\_tsil\_length tsil\_length;; - : bool = true |
| --- |

Part b: Implementing tsil\_append.

In this section, we implement a function which can append two tsils together. This is our unit test:

| **let** test\_tsil\_append candidate =  **let** n1 = Random.int 1000 **and** n2 = Random.int 500  **in** **let** b0 = (candidate Lin Lin = Lin)  **and** b1 = (candidate Lin (Snoc(Lin, 3)) = (Snoc(Lin, 3)))  **and** b2 = (candidate (Snoc(Snoc(Lin, 3), 4)) (Snoc(Lin, 5)) = (Snoc(Snoc(Snoc(Lin, 3), 4), 5)))  **and** b3 = (candidate (Snoc(Lin, n1)) (Snoc(Lin, n2)) = (Snoc(Snoc(Lin, n1), n2)))  **in** b0 && b1 && b2 && b3;; |
| --- |

For our solution, we take inspiration from earlier examples of recursive append functions for lists. In those functions, the base case is the list on the right, and a new list is gradually constructed on the left to result in a new appended list. Our implementation is analogous, except that in this case we take note of the fact that Snoc takes a tsil on the left and adds new elements on the right, and so we must use the tsil on the left as our base case.

| **let** tsil\_append ts1\_given ts2\_given =  **let** **rec** visit ts =  **match** ts **with**  | Lin ->  ts1\_given  | Snoc(c,d) ->  Snoc(visit c, d)  **in** visit ts2\_given;; |
| --- |

Our implementation passes the unit test:

| # test\_tsil\_append tsil\_append;; - : bool = true |
| --- |

Part c: Implementing tsil\_map.

For this portion, we are asked to implement a mapping function for a tsil that applies a function to each element. It is analogous to List.map. Here is a unit test:

| **let** test\_int\_tsil\_map candidate =  **let** n1 = Random.int 500 **and** n2 = Random.int 400  **in** **let** b0 = (candidate (**fun** x -> x/2) (Snoc(Snoc(Lin,4), 5)) = (Snoc(Snoc(Lin, 2), 2)))  **and** b1 = (candidate (**fun** x -> x + 1) (Snoc(Snoc(Snoc(Lin, 4), n1), n2)) = (Snoc(Snoc(Snoc(Lin, 5), succ(n1)), succ(n2))))  **and** b2 = (candidate (**fun** x -> x) Lin = Lin)  **and** b3 = (candidate (**fun** x -> x) (Snoc(Lin, n1)) = (Snoc(Lin, n1)))  **and** b4 = (candidate (**fun** x -> x/2) (Snoc(Snoc(Lin,n1), n2)) = (Snoc(Snoc(Lin, n1/2), n2/2)))  **in** b0 && b1 && b2 && b3 && b4;; |
| --- |

On to our solution. Essentially, our implementation here constructs a new tsil after recursively visiting each element of ts\_given and applying f to it. To that end, it needs to take in two inputs, f and ts\_given. Our code follows.

| **let** tsil\_map f ts\_given =  **let** **rec** visit ts =  **match** ts **with**  | Lin ->  Lin  | Snoc(c,d) ->  Snoc(visit c, f d)  **in** visit ts\_given;; |
| --- |

Our implementation passes the unit test:

| # test\_int\_tsil\_map tsil\_map;; - : bool = true |
| --- |

Part d: Implementing tsil\_reverse.

In this section, we implement a function that can reverse a tsil. We do so recursively. First, a unit test:

| **let** test\_tsil\_reverse candidate =  **let** n1 = Random.int 400 **and** n2 = Random.int 600   **in** **let** b0 = (candidate Lin = Lin)  **and** b1 = (candidate (Snoc(Lin, n1)) = (Snoc(Lin, n1)))  **and** b2 = (candidate (Snoc(Snoc(Snoc(Lin, n1), n2), 67)) = (Snoc(Snoc(Snoc(Lin, 67), n2), n1)))  **in** b0 && b1 && b2;; |
| --- |

On to our implementation. For each application of visit, we decompose a given tsil into a smaller tsil, a , and an element b. We then put b on the left, and append our induction hypothesis on the right. The process is repeated until Lin. This will result in a reversed tsil.

| **let** tsil\_reverse ts\_given =  **let** **rec** visit ts =   **match** ts **with**  | Lin ->  Lin  | Snoc(a,b) ->  tsil\_append (Snoc(Lin, b)) (visit a)  **in** visit ts\_given;; |
| --- |

This implementation passes the unit test:

| # test\_tsil\_reverse tsil\_reverse;; - : bool = true |
| --- |

Part d(2): Implementing tsil\_reverse using an accumulator.

In this section, we use an accumulator to reverse a tsil. Our accumulator should begin from Lin, and build a reversed tsil as we recursively deconstruct the given tsil, ts\_given. When ts\_given is deconstructed all the way to Lin, the accumulator matches the reverse of ts\_given and is returned. Our implementation follows.

| **let** tsil\_reverse\_acc ts\_given =  **let** **rec** visit ts a =  **match** ts **with**  | Lin ->  a  | Snoc(c, d) ->  visit c (Snoc(a, d))  **in** visit ts\_given Lin;; |
| --- |

This implementation passes the unit test for tsil\_reverse (the unit test is found above):

| # test\_tsil\_reverse tsil\_reverse\_acc;; - : bool = true |
| --- |

Part e: Implementing tsil\_of\_list and list\_of\_tsil.

We begin with the unit tests for tsil\_of\_list and list\_of\_tsil, already provided to us.

| **let** test\_tsil\_of\_list candidate =  **let** b0 = (candidate [] = Lin)  **and** b2 = (candidate (1 :: 0 :: []) = Snoc (Snoc (Lin, 0), 1))  (\* etc. \*)  **in** b0 && b2;;  **let** test\_list\_of\_tsil candidate =  **let** b0 = (candidate Lin = [])  **and** b2 = (candidate (Snoc (Snoc (Lin, 0), 1)) = 1 :: 0 :: [])  (\* etc. \*)  **in** b0 && b2;; |
| --- |

Our solution is below. We implement both functions recursively.

| **let** tsil\_of\_list ls =  **let** **rec** visit vs =   **match** vs **with**  | [] ->  Lin  | v :: vs' ->  Snoc(visit vs', v)  **in** visit ls;; |
| --- |

For tsil\_of\_list, we match Nil to Lin if the list provided is empty. Otherwise, we apply the tsil equivalent of construction (Snoc) to our induction hypothesis and v. Recursive calls will be made until visit vs' (our induction hypothesis) matches the empty list, upon which Lin will be returned and Snoc will be applied to the appropriate vs, returning successive outputs for visit vs' until we have the output of visit ls, which is the tsil of our list.

| ​​​​**let** list\_of\_tsil ts =  **let** **rec** visit vs =   **match** vs **with**  | Lin ->  []  | Snoc(c,d) ->  d :: visit c  **in** visit ts;; |
| --- |

The way we implement list\_of\_tsil is analogous to how we implemented tsil\_of\_list. We match Lin to Nil if the tsil provided is empty. Otherwise, we apply Cons :: to d and our induction hypothesis. Recursive calls will be made until visit c (our induction hypothesis) matches Lin, upon which Nil will be returned and :: will be applied to the appropriate ds, returning successive outputs for visit c until we have the output for visit ts, which is the list that we want.

Our implementations pass the unit test:

| # test\_list\_of\_tsil list\_of\_tsil;; - : bool = true # test\_tsil\_of\_list tsil\_of\_list;; - : bool = true |
| --- |

Part f

In this part, we make use of the functions we just defined above to indirectly implement functions for tsils using functions for lists, and vice versa. tsil\_of\_list and list\_of\_tsil come in very handy because they allow us to use both types interchangeably. For example, we can take a list, turn it into a tsil, apply a function that acts on tsils, and then convert the resulting tsil back into a list (if a tsil is returned after applying said function).

1. Implementing tsil\_length in terms of list\_length

For this first problem, we don’t need to convert a list back into a tsil since we just need to convert a tsil into a list, and return the length of that list. Here is the unit test for our implementation:

| **let** test\_tsil\_length candidate =  **let** n1 = Random.int 5 **and** n2 = Random.int 1000  **in** **let** b0 = (candidate Lin = 0)  **and** b1 = (candidate (Snoc(Lin, 3)) = 1)  **and** b2 = (candidate (Snoc(Snoc(Lin, 3), 5)) = 2)  **and** b3 = (candidate (Snoc(Snoc(Snoc(Lin, n1), n2), 5)) = 3)  **in** b0 && b1 && b2 && b3;; |
| --- |

And this is our implementation:

| **let** tsil\_length\_using\_list ts =  List.length (list\_of\_tsil ts);; |
| --- |

Which passes the unit test:

| # test\_tsil\_length tsil\_length\_using\_list;;  - : bool = true |
| --- |

1. Implementing list\_length in terms of tsil\_length

The solution for this is entirely analogous to the previous problem. We just convert a list into a tsil, and apply tsil\_length. First, this is the unit test we will use, which is already provided for us in the notes:

| **let** atoi\_v0 n =  **let** () = **assert** (n >= 0) **in**  **let** **rec** visit i =  **if** i = 0  **then** []  **else** **let** i' = pred i  **in** **let** ih = visit i'  **in** i' :: ih  **in** visit n;;  **let** test\_list\_length\_int candidate =  (\* the base case: \*)  **let** bc = (candidate [] = 0)  (\* instance of the induction step: \*)  **and** is = (**let** n = Random.int 1000  **in** **let** ns = atoi\_v0 n  **in** candidate (n :: ns) = succ (candidate ns))  (\* a few handpicked lists: \*)  **and** b1 = (1 = candidate [1])  **and** b2 = (2 = candidate [2; 1])  **and** b3 = (3 = candidate [3; 2; 1])  **and** b4 = (4 = candidate [4; 3; 2; 1])  (\* a few automatically generated lists: \*)  **and** b5 = (candidate (atoi\_v0 5) = 5)  **and** b6 = (candidate (atoi\_v0 10) = 10)  **and** b7 = (candidate (atoi\_v0 100) = 100)  (\* etc. \*)  **in** bc && is && b1 && b2 && b3 && b4 && b5 && b6 && b7;; |
| --- |

Our implementation is below:

| **let** list\_length\_using\_tsil ls =  tsil\_length (tsil\_of\_list ls);; |
| --- |

And our implementation passes the unit test:

| # test\_list\_length\_int list\_length\_using\_tsil;; - : bool = true |
| --- |

1. Implementing tsil\_map using list\_map

For this section, we convert a tsil into a list, then apply list\_map and convert the list back into a tsil. First, here is list\_map, which recursively applies a function to each element of a given list:

| **let** list\_map f xs\_given =  (\* list\_map : ('a -> 'b) -> 'a list -> 'b list \*)  **let** **rec** visit xs =  **match** xs **with**  | [] ->  []  | x :: xs' ->  **let** ih = visit xs'  **in** f x :: ih  **in** visit xs\_given;; |
| --- |

Here is our unit test:

| **let** test\_int\_tsil\_map candidate =  **let** n1 = Random.int 500 **and** n2 = Random.int 400  **in** **let** b0 = (candidate (**fun** x -> x/2) (Snoc(Snoc(Lin,4), 5)) = (Snoc(Snoc(Lin, 2), 2)))  **and** b1 = (candidate (**fun** x -> x + 1) (Snoc(Snoc(Snoc(Lin, 4), n1), n2)) = (Snoc(Snoc(Snoc(Lin, 5), succ(n1)), succ(n2))))  **and** b2 = (candidate (**fun** x -> x) Lin = Lin)  **and** b3 = (candidate (**fun** x -> x) (Snoc(Lin, n1)) = (Snoc(Lin, n1)))  **and** b4 = (candidate (**fun** x -> x/2) (Snoc(Snoc(Lin,n1), n2)) = (Snoc(Snoc(Lin, n1/2), n2/2)))  **in** b0 && b1 && b2 && b3 && b4;; |
| --- |

And our implementation is below:

| **let** tsil\_map\_using\_list f ts =  tsil\_of\_list (list\_map f (list\_of\_tsil ts));; |
| --- |

This implementation passes the above unit test.

| # test\_int\_tsil\_map tsil\_map\_using\_list;; - : bool = true |
| --- |

1. Implementing list\_map using tsil\_map

Our solution for this problem is analogous to our solution for the previous problem.

Here are our unit tests. Since lists are polymorphic, we use 3 different unit tests to beef up our overall testing. Here they are below.

| **let** test\_int\_list\_map candidate =  **let** n1 = Random.int 20 **and** n2 = Random.int 30  **in** **let** b0 = (candidate (**fun** x -> x+1) [] = List.map (**fun** x -> x+1) [])  **and** b1 = (candidate (**fun** x -> x/2) [1;2;3;4] = List.map (**fun** x -> x/2) [1;2;3;4])  **and** b2 = (candidate (**fun** x -> x) [1;3;5] = List.map (**fun** x -> x) [1;3;5])  **and** b3 = (candidate (**fun** x -> x\*x) [n1;n2] = List.map (**fun** x -> x\*x) [n1;n2])  **in** b0 && b1 && b2 && b3;;  **let** test\_int\_opt\_list\_map candidate =  **let** b0 = (candidate (**fun** x -> None) [Some 1;Some 3;Some 4] = List.map (**fun** x -> None) [Some 1;Some 3;Some 4])  **and** b1 = (candidate (**fun** x -> None) [] = List.map (**fun** x -> None) [])  **and** b2 = (candidate (**fun** x -> Some x) [Some 1;Some 2; Some 3] = List.map (**fun** x -> Some x) [Some 1;Some 2;Some 3])  **and** b3 = (candidate (**fun** x -> Some x) [] = List.map (**fun** x -> Some x) [])  **in** b0 && b1 && b2 && b3;;  **let** test\_char\_list\_map candidate =  **let** b0 = (candidate (**fun** x -> x) [] = [])  **and** b1 = (candidate (**fun** x -> x) ['a';'b'] = ['a'; 'b'])  **and** b2 = (candidate (**fun** i -> (char\_of\_int(int\_of\_char i - int\_of\_char 'a' + int\_of\_char 'A'))) ['a'; 'b'; 'c'] = ['A'; 'B'; 'C'])  **in** b0 && b1 && b2;; |
| --- |

The 3 unit tests do not cover all possible types of lists (it would be impossible to do so), but they provide a more rigorous test to ensure that our function doesn’t only work for one kind of list.

Here is our implementation:

| let list\_map\_using\_tsil f ls =  list\_of\_tsil(tsil\_map f (tsil\_of\_list ls));; |
| --- |

Which passes all the unit tests:

| # test\_int\_opt\_list\_map list\_map\_using\_tsil;; - : bool = true # test\_int\_list\_map list\_map\_using\_tsil;; - : bool = true # test\_char\_list\_map list\_map\_using\_tsil;; - : bool = true |
| --- |

Part g: tsil\_fold\_left and tsil\_fold\_right

For this section, we implement tsil\_fold\_left and tsil\_fold\_right, the counterparts of list\_fold\_left and list\_fold\_right. tsil\_fold\_left offers a skeletal structure for tail recursion with an accumulator on tsils, while tsil\_fold\_right offers a skeletal structure for recursion on tsils. In that sense, tsil\_fold\_left and tsil\_fold\_right can be used to implement some of the functions we implemented earlier in this exercise. After sharing our implementation, we demonstrate some examples of tsil\_fold\_left and tsil\_fold\_right in action.

Our implementations draw inspiration from the examples of list\_fold\_left and list\_fold\_right found in the notes, reproduced here to make the report self-contained:

| **let** list\_fold\_right nil\_case cons\_case vs\_given =  *(\* list\_fold\_right : 'a -> ('b -> 'a -> 'a) -> 'b list -> 'a \*)*  **let** **rec** traverse vs =  *(\* traverse : 'b list -> 'a \*)*  **match** vs **with**  | [] ->  nil\_case  | v :: vs' ->  **let** ih = traverse vs'  **in** cons\_case v ih  **in** traverse vs\_given;; |
| --- |

| **let** list\_fold\_left nil\_case cons\_case vs\_given =  *(\* list\_fold\_left : 'a -> ('b -> 'a -> 'a) -> 'b list -> 'a \*)*  **let** **rec** traverse vs a =  *(\* traverse : 'b list -> 'a -> 'a \*)*  **match** vs **with**  | [] ->  a  | v :: vs' ->  traverse vs' (cons\_case v a)  **in** traverse vs\_given nil\_case;; |
| --- |

We just adapt the code to fit the grammar for the tsil type, and remember that Snoc takes a tsil on the left and constructs a new tsil by adding a new element to the right, which means that we need to flip the order in which the snoc\_case function takes in inputs.

| **let** tsil\_fold\_right lin\_case snoc\_case ts\_given =  **let** **rec** traverse ts =  **match** ts **with**  | Lin ->  lin\_case  | Snoc(c,d) ->  **let** ih = traverse c  **in** snoc\_case ih d  **in** traverse ts\_given;; |
| --- |

And here is tsil\_fold\_left:

| **let** tsil\_fold\_left lin\_case snoc\_case ts\_given =  **let** **rec** traverse ts a =  **match** ts **with**  | Lin ->  a  | Snoc(c,d) ->  traverse c (snoc\_case d a)  **in** traverse ts\_given lin\_case;; |
| --- |

Now, here are some cases where we can use these functions. We can use tsil\_fold\_left to reverse a tsil (essentially copying the code for tsil\_reverse\_acc), and we can also use it to calculate the length of a tsil (also copying the code for tsil\_length).

| # tsil\_fold\_left Lin (**fun** d a -> Snoc(a,d)) (Snoc(Snoc(Lin, 3), 4));; - : int tsil = Snoc (Snoc (Lin, 4), 3) |
| --- |

| # tsil\_fold\_left 0 (**fun** c a -> succ a) (Snoc(Snoc(Lin,3), 4));; - : int = 2 |
| --- |

Here is a case where we can use tsil\_fold\_right to reverse a tsil:

| # tsil\_fold\_right Lin (**fun** ih d -> tsil\_append (Snoc(Lin,d)) ih) (Snoc(Snoc(Lin, 3), 4));; - : int tsil = Snoc (Snoc (Lin, 4), 3) |
| --- |

Conclusion

In this exercise, we defined a new type and implemented familiar functions that act on it. These functions are familiar in the sense that we have previously seen counterparts for tasks like appending, reversing, and mapping for strings and lists. This demonstrates the versatility of custom types, and opens up a whole world of options for us in terms of creating new expressions and useful functions to act on those expressions.

Something else which we got to deepen our familiarity with was with indirect implementation. We had our first taste with using int\_of\_char and char\_of\_int to change a character. We’ve encountered it again with custom types in this week’s report.

# Exercise 07 [[*Exercise 06*](https://delimited-continuation.github.io/YSC1212/2021-2022_Sem2/week-08_list-map.html#exercise-list-swap) from Week 08: about list\_map]

Introduction

In this exercise, we exhibit an expression ('a \* 'b) list -> ('b \* 'a) list, which, given a pair in a list, inverts the pair and returns a new list with these values.

Solution

One way is to define a function that inverts a pair in a list using the match-expression:

| **let** f l =   **match** l **with**   | [] -> []  | (a, b)::tl -> [(b, a)];;  **val** f : ('a \* 'b) list -> ('b \* 'a) list = <**fun**> |
| --- |

Or, using list\_map defined as:

| **let** list\_map f xs\_given =  *(\* list\_map : ('a -> 'b) -> 'a list -> 'b list \*)*  **let** **rec** visit xs =  **match** xs **with**  | [] ->  []  | x :: xs' ->  **let** ih = visit xs'  **in** f x :: ih  **in** visit xs\_given;; |
| --- |

The following function inverts the elements of each pair in the list, and returns a new list with these values.

| # **let** mirror vs\_given =  list\_map (**fun** (a, b) -> (b, a)) vs\_given;;  **val** mirror : ('a \* 'b) list -> ('b \* 'a) list = <**fun**> # |
| --- |

| # mirror [('a',2); ('c',3)];; - : (int \* char) list = [(2, 'a'); (3, 'c')] # |
| --- |

Conclusion

In this exercise, we make use of list\_map, which we used in Exercise 04, to help us map a function that inverts elements within a pair, onto a list of pairs. It’s important to recall that each pair within a list has to follow the same type. For example, our unit test used a list of (int \* char) pairs.

# Exercise 08 [[*Exercise 09*](https://delimited-continuation.github.io/YSC1212/2021-2022_Sem2/week-08_list-map.html#exercise-list-mapi) from Week 08: about list\_mapi]

Introduction

In this exercise, we implement a list analogue of String.mapi and use that function to implement some polymorphic functions that, given a list, applies a given function to each element of the list and returns a new list.

Solution:

1. *Implement the list analogue of String.mapi so that*

| (\* example \*) # list\_mapi (**fun** n v -> (n, v)) ['a'; 'b'; 'c'];; - : (int \* char) list = [(0, 'a'); (1, 'b'); (2, 'c')] |
| --- |

Let us implement a unit-test function for int:

| # **let** test\_int\_list\_mapi candidate =  **let** b0 = (candidate (**fun** i \_ -> 4) [] = [])  **and** b1 = (candidate (**fun i** \_ -> 5) [1;2;3;4;5] = List.mapi (**fun** i \_ -> 5) [1;2;3;4;5])  **and** b2 = (candidate (**fun** i \_ -> i + 5) [3;4;5] = List.mapi (**fun** i \_ -> i + 5) [3;4;5])  **and** b3 = (candidate (**fun** i c -> **if** i = 3 **then** 3 **else** c) [3;4;5;6;7;8] = List.mapi (**fun** i c -> **if** i = 3 **then** 3 **else** c) [3;4;5;6;7;8])  **in** b0 && b1 && b2 && b3;; **val** test\_int\_list\_mapi :  ((int -> int -> int) -> int list -> int list) -> bool = <**fun**> |
| --- |

First, we assign the length of the given list to len. If the list is of length 0, i.e., an empty list, the function should yield an empty list. Otherwise, this function will create a pair of the element and its index and add the pair to the traverse\_list in the recursive manner.

| **let** list\_mapi f lst =   **let** len = **List**.length lst **in**   **let** **rec** traverse\_list i lst2 =  **if** i = len  **then** []  **else** (f i (**List**.hd lst2)) :: traverse\_list (succ i) (**List**.tl lst2) **in** traverse\_list 0 lst;;  **val** list\_mapi : (int -> 'a -> 'b) -> 'a list -> 'b list = <**fun**> |
| --- |

| list\_mapi (**fun** n v -> (n, v)) ['a'; 'b'; 'c'];; - : (int \* char) list = [(0, 'a'); (1, 'b'); (2, 'c')] |
| --- |

This implementation passes the unit test:

| # test\_int\_list\_mapi list\_mapi;; - : bool = true |
| --- |

*b) Use this analogue to implement the polymorphic identity function over lists.*

Solution:

| **let** test\_charlistidentity candidate =  **let** b0 = (candidate [] = [])  **and** b1 = (candidate ['a'] = ['a'])  **and** b2 = (candidate ['b';'a'] = ['b';'a'])  **and** b3 = (candidate ['c'; 'b'; 'a'] = ['c'; 'b'; 'a'])  **in** b0 && b1 && b2 && b3;; |
| --- |

We use a function that, for each element of the given list, returns its identical.

| **let** identity lst = list\_mapi (**fun** u v -> v) lst;; **val** identity : 'a list -> 'a list = <**fun**> |
| --- |

| identity ['a'; 'b'; 'c'];; - : char list = ['a'; 'b'; 'c'] |
| --- |

| # test\_charlistidentity identity;; - : bool = true # |
| --- |

*c) Use this analogue to implement a polymorphic function that maps a list to a matching list of increasing indices.*

| **let** test\_incindex candidate =  **let** b0 = (candidate [] = [])  **and** b1 = (candidate ['c'] = [0])  **and** b2 = (candidate ['b'; 'c'] = [0; 1])  **and** b3 = (candidate ['a'; 'b'; 'c'] = [0; 1; 2])  **in** b0 && b1 && b2 && b3;; |
| --- |

In the similar manner as the last question, we use a function that, for each element of the given list, returns its index in the list.

| **let** increasing\_func lst = list\_mapi (**fun** u v -> u) lst;; **val** increasing\_func : 'a list -> int list = <**fun**> |
| --- |

| increasing\_func ['a'; 'b'; 'c'];; - : int list = [0; 1; 2] |
| --- |

This implementation passes the unit test.

| # test\_incindex increasing\_func;; - : bool = true # |
| --- |

*d) Use this analogue to implement a polymorphic function that maps a list to a matching list of decreasing indices.*

| **let** test\_decindex candidate =  **let** b0 = (candidate [] = [])  **and** b1 = (candidate ['c'] = [0])  **and** b2 = (candidate ['b'; 'c'] = [1; 0])  **and** b3 = (candidate ['a'; 'b'; 'c'] = [2; 1; 0])  **in** b0 && b1 && b2 && b3;; |
| --- |

Similarly to how our string functions in the midterm project, we manipulate the list length and index of elements to achieve our objective. We want the function to return the last index of the given list given the first element of the list, for example. Since the last index of a list is len -1, we can get the decreasing indices by len - u - 1 where u is the index of the element in the given list.

| **let** decreasing\_func lst =   **let** len = **List**.length lst **in**  list\_mapi (**fun** u v -> (len - u - 1)) lst;; **val** decreasing\_func : 'a list -> int list = <**fun**> |
| --- |

| decreasing\_func ['a'; 'b'; 'c'];; - : int list = [2; 1; 0] |
| --- |

This implementation passes the unit test.

| # test\_decindex decreasing\_func;; - : bool = true # |
| --- |

Conclusion

This exercise reminds us of the recursive functions we have implemented in the midterm project. We have implemented the list analogue of String.mapi and noted that elements in the list are analogue of the characters in a string; hence we are able to approach the exercises in the same way as how we wrote a recursive function for palindromes, for example. This concept of elements in a list being analogue of the characters in a string comes in handy for the next exercise as well.

# Exercise 09 [[*Exercise 02*](https://delimited-continuation.github.io/YSC1212/2021-2022_Sem2/week-09_list-reversal.html#exercise-equivalence-of-various-functions-over-lists) from Week 09: about function equivalences]

Introduction:

*Given the standard definitions of List.append (to concatenate two lists), List.rev (to reverse a list), and List.map (to map a function over a list, homomorphically), consider the following 10 functions:*

1. *fun xs ys -> List.append (List.rev xs) (List.rev ys)*
2. *fun xs ys -> List.append (List.rev ys) (List.rev xs)*
3. *fun xs ys -> List.rev (List.append ys xs)*
4. *fun xs ys -> List.rev (List.append xs ys)*
5. *fun f xs -> List.map f (List.rev xs) assuming that f will always denote a pure and total function*
6. *fun f xs -> List.rev (List.map f (List.rev xs)) assuming that f will always denote a pure and total function*
7. *fun f xs -> List.rev (List.map f xs) assuming that f will always denote a pure and total function*
8. *fun f xs -> List.map f xs assuming that f will always denote a pure and total function*
9. *fun f xs ys -> List.map f (List.append xs ys) assuming that f will always denote a pure and total function*
10. *fun f xs ys -> List.append (List.map f xs) (List.map f ys) assuming that f will always denote a pure and total function*

*In reference to* [*Exercise 08*](https://delimited-continuation.github.io/YSC1212/2021-2022_Sem2/week-08_list-map.html#exercise-listlessness-map) *in Week 08, which of these functions are equivalent? Briefly justify each answer, e.g., by drawing commuting diagrams, possibly using the following swap arrow.*

For this exercise, we revisit the recurring concept of functional equivalence, by examining the output of each function. For those functions that are equivalent, we are able to illustrate equivalence using commuting diagrams.

Solution:

For the following diagrams,

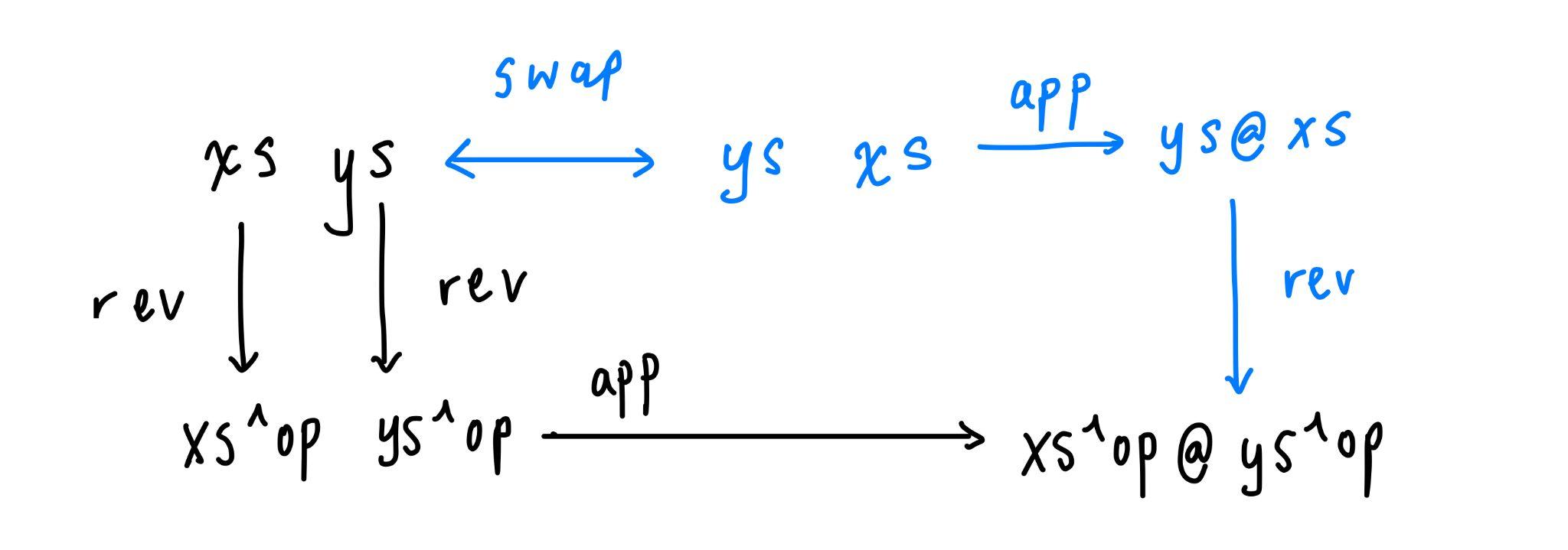
* vs^op = List.rev vs

Functions 0 and 2 are equivalent.

0. fun xs ys -> List.append (List.rev xs) (List.rev ys)

2. fun xs ys -> List.rev (List.append ys xs)

Commuting diagram:

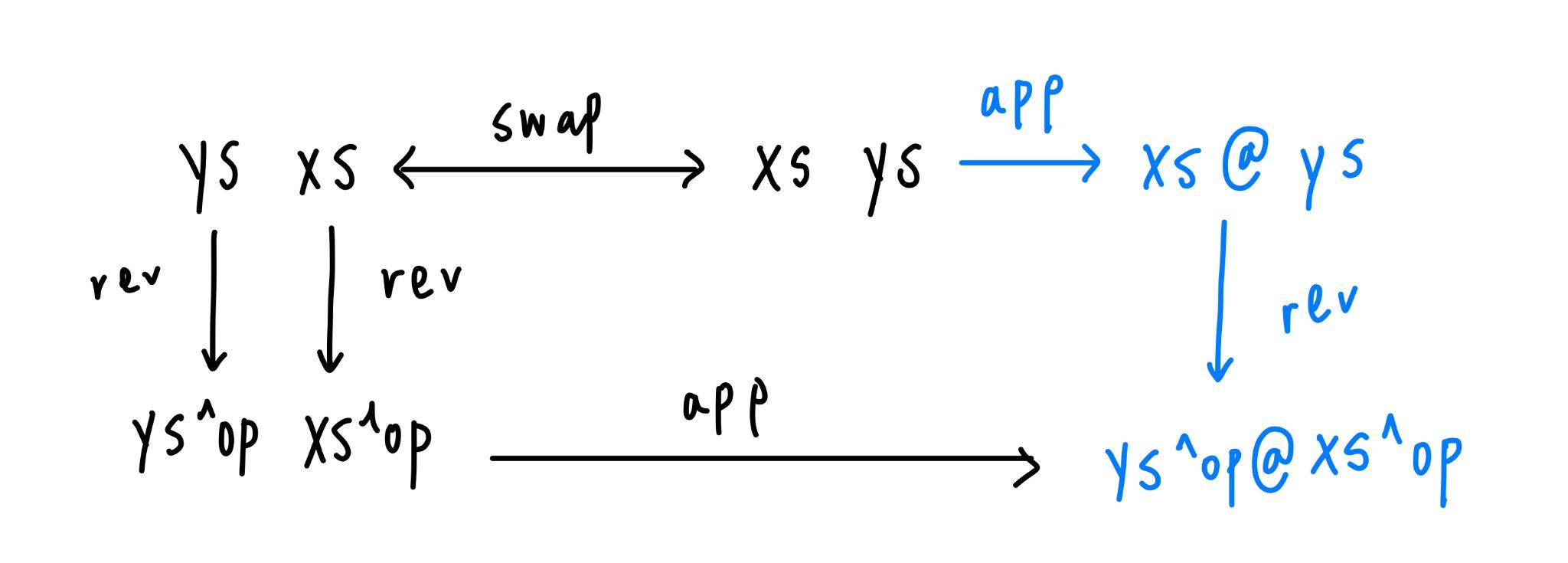


Functions 1 and 3 are equivalent.

1. fun xs ys -> List.append (List.rev ys) (List.rev xs)

3. fun xs ys -> List.rev (List.append xs ys)

Commuting diagram:

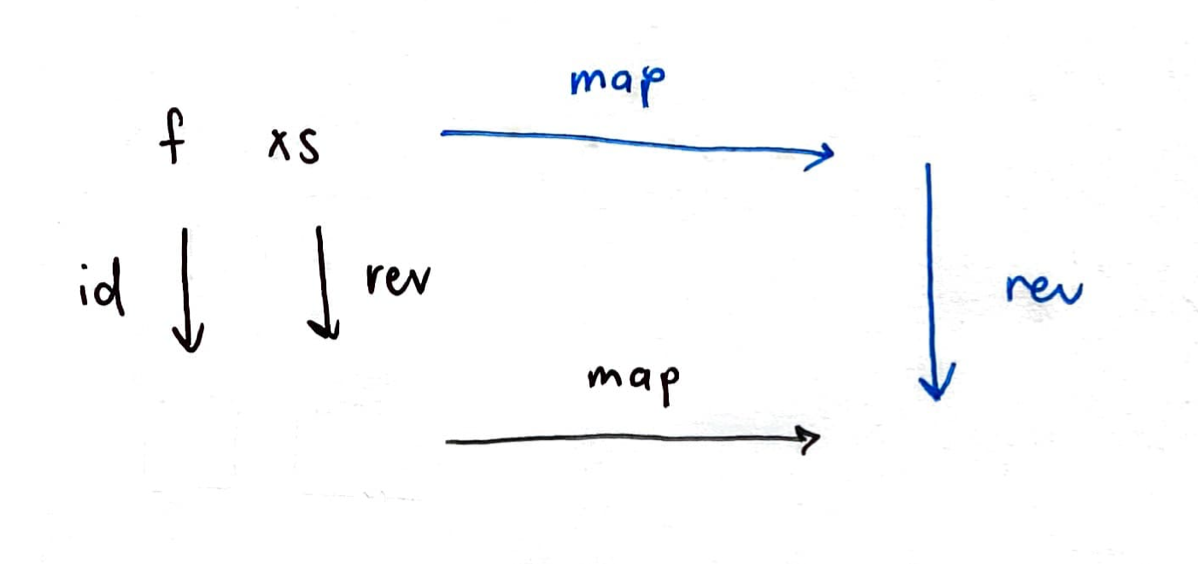


Functions 4 and 6 are equivalent.

4. fun f xs -> List.map f (List.rev xs)

6. fun f xs -> List.rev (List.map f xs)

Commuting diagram:

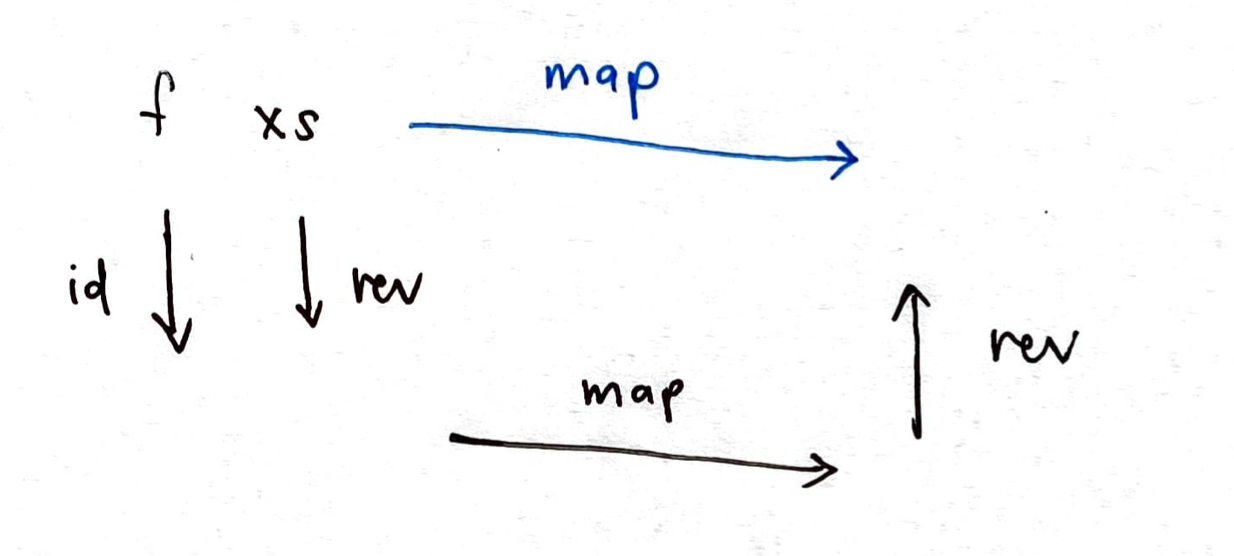


Functions 5 and 7 are equivalent.

5. fun f xs -> List.rev (List.map f (List.rev xs))

7. fun f xs -> List.map f xs

Commuting diagram:

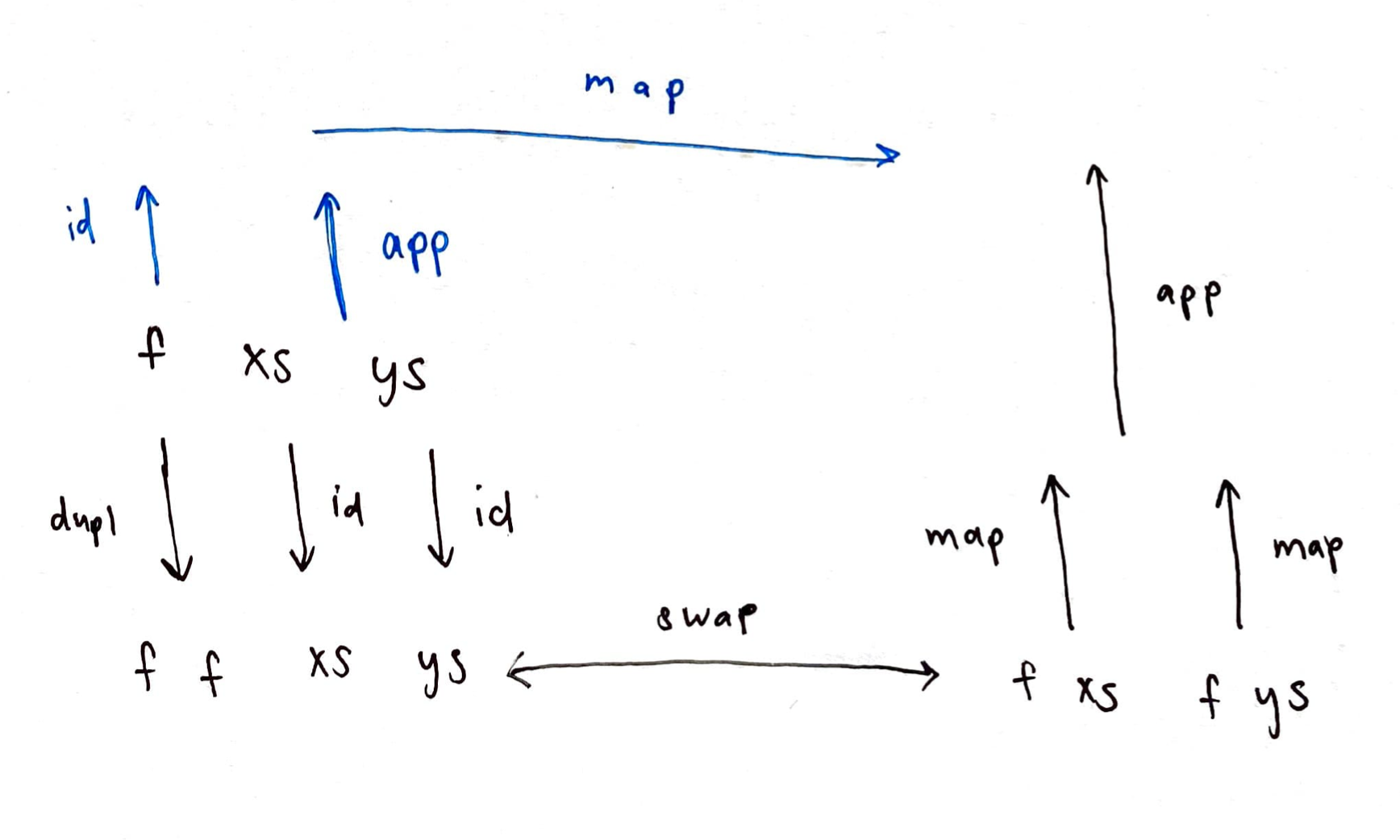


Functions 8 and 9 are equivalent.

8. fun f xs ys -> List.map f (List.append xs ys)

9. fun f xs ys -> List.append (List.map f xs) (List.map f ys)

Commuting diagram:



*Subsidiary question, assuming that String.append denotes a string-concatenation function and String.rev denotes a string-reversal function:*

*If we were to replace List. by String. in the functions above, would your answers be the same? Why?*

If List. were to be replaced with String., our answers would be the same, since characters in a string is an analogue of elements in a list. In the corner case of the empty string, reversing an empty string returns an empty string, while concatenating an empty string to another string returns the other string, just like for the empty list.

Conclusion

This exercise takes us back to functional equivalence from as far back as Week 03, and our midterm project on polynomials and determinism. By considering the outputs of our functions, we evaluate the bodies of these functions, which are just parameterised expressions. Since the bodies of our functions give rise to the same output when given the same inputs, we conclude that the functions are equivalent. We extend our notion of functional equivalence to functions on a new type we have learnt, lists.

We note that the assumption that f will always denote a pure and total function to be a key part of the equivalence of the functions. This allows us to not have to consider: (i) if there is a trace left by f, which would cause some functions to be not equivalent; (ii) if the function does not map every element of its domain to one element of its codomain.

# Exercise 10 [[*Exercise 10*](https://delimited-continuation.github.io/YSC1212/2021-2022_Sem2/week-09_generic-programming-with-lists.html#exercise-list-fold-right-nil-and-cons) from Week 09: about list\_fold\_right]

Introduction:

*Given a list, what happens if we instantiate nil\_case to be [] and cons\_case to be (fun x ih -> x :: ih), i.e., List.cons? Which function does list\_fold\_right emulate then?*

*In other words, you are asked to characterize the following OCaml function:*

| **let** the\_following\_OCaml\_function xs = list\_fold\_right [] (**fun** x ih -> x :: ih) xs;; |
| --- |

Solution:

Instantiating this into the structure of list\_fold\_right:

| **let** list\_fold\_right [] (**fun** x ih -> x :: ih) xs =  *(\* list\_fold\_right : 'a -> ('b -> 'a -> 'a) -> 'b list -> 'a \*)*  **let** **rec** traverse xs =  *(\* traverse : 'b list -> 'a \*)*  **match** vs **with**  | [] ->  []  | x :: xs' ->  **let** ih = traverse xs'  **in** (**fun** x ih -> x :: ih) x ih  **in** traverse xs;; |
| --- |

In the base case (nil\_case argument), for an empty list, the recursive function traverse returns the empty list. For a non-empty list, In the induction step, cons\_case concatenates the head of the list, x, which is the first argument, to the induction hypothesis. Essentially then, given a list e.g. [x1; x2; x3; x4; x5], the\_following\_OCaml\_function yields the result of evaluating x1 :: (x2 :: (x3 :: (x4 :: (x5 :: []), 5 nested applications of the (**fun** x ih -> x :: ih).

Thus, we conclude that it emulates the polymorphic identity function from Question 8b above. The following code provides an example to illustrate:

| the\_following\_OCaml\_function ['a'; 'b'; 'c'];; - : char list = ['a'; 'b'; 'c'] |
| --- |

We can check our claim more robustly, using a unit test.

Adding a unit test for an identity function for the int list type,

| # **let** test\_intlistidentity candidate =  **let** b0 = (candidate [] = [])  **and** b1 = (candidate [0] = [0])  **and** b2 = (candidate [1;0] = [1;0])  **and** b3 = (candidate [2; 1; 0] = [2; 1;0])  **in** b0 && b1 && b2 && b3;;  **val** test\_intlistidentity : (int list -> int list) -> bool = <**fun**> |
| --- |

We verify that this function passes both int list and char list identity function’s unit tests:

| # **let** () = **assert** (test\_intlistidentity the\_following\_OCaml\_function);; # **let** () = **assert** (test\_charlistidentity the\_following\_OCaml\_function);; # |
| --- |

Conclusion:

Just like in nat\_fold\_right, the list analogue list\_fold\_right allows us to parameterise recursive function on lists, and implement a wide range of functions on lists using a single generic function.

# Exercise 11 [[*Exercise 16*](https://delimited-continuation.github.io/YSC1212/2021-2022_Sem2/week-09_generic-programming-with-lists.html#exercise-list-fold-left-nil-and-cons) from Week 09: about list\_fold\_left]

Introduction:

*Given a list xs, what happens if we instantiate nil\_case to be [] and cons\_case to be (fun x a -> x :: a), i.e., List.cons? Which function does list\_fold\_left emulate then?*

*In other words, you are asked to characterize the following other OCaml function:*

| **let** the\_following\_other\_OCaml\_function xs =  list\_fold\_left [] (**fun** x a -> x :: a) xs;; |
| --- |

Solution:

Instantiating the given nil\_case and cons\_case into the structure of list\_fold\_left,

| **let** list\_fold\_left [] (**fun** x a -> x :: a) xs =  *(\* list\_fold\_left : 'a -> ('b -> 'a -> 'a) -> 'b list -> 'a \*)*  **let** **rec** traverse xs a =  *(\* traverse : 'b list -> 'a -> 'a \*)*  **match** xs **with**  | [] ->  a  | x :: xs' ->  traverse xs' ((**fun** x a -> x :: a) x a)  **in** traverse xs [];; |
| --- |

When xs denotes the empty list, the function yields the empty list. If xs is a non-empty list, traverse traverses the given list and at the same time initialises the empty list with a as []. In each subsequent recursive call, each first element of the list is cons’ed to the accumulator. As a result, this gives us the reverse function. Here is an example to demonstrate:

| the\_following\_other\_OCaml\_function ['a'; 'b'; 'c'];; - : char list = ['c'; 'b'; 'a'] |
| --- |

We verify that it passes the unit test for reverse from the earlier exercise, given in the lecture notes:

| # **let** () = **assert** (test\_reverse the\_following\_other\_OCaml\_function);; # |
| --- |

Conclusion:

Like in the previous exercise, lift\_fold\_left gives us yet another way of implementing functions over lists. This exercise, when viewed alongside Exercise 10, is a good reminder that tail recursion is structurally different (and leads to different results) than recursion. When thinking about tail recursion, the ‘transferring a stack of plates from stack A to an empty stack B’ example mentioned in class comes in particularly handy, and helps us intuitively understand why the\_following\_other\_OCaml\_function reverses a list.

# Exercise 12 [[*Exercise 06*](https://delimited-continuation.github.io/YSC1212/2021-2022_Sem2/week-10_polymorphic-binary-trees.html#exercise-comparing-the-number-of-leaves-and-the-number-of-nodes-of-a-random-binary-tree-of-integers) from Week 10: about testing random binary trees of integers]

Introduction:

*Implement a predicate that verifies that for any random binary tree of integers, its number of leaves is equal to its number of nodes, plus 1.*

In this exercise, we are going to create a unit test function to verify a property of a randomly generated binary tree of integers: that the number of leaves will be 1 greater than the number of nodes.

Solution:

Here is the unit test function. We use the function number\_of\_leaves\_v1 to get the number of leaves of a tree and number\_of\_nodes\_v1 to get the number of nodes. Then, we use a conditional statement to check if the desired condition is fulfilled. To increase the coverage of our unit test function, we verify this condition multiple times.

| **let** test\_random\_binary\_tree candidate =  **let** b0 = (**let** i = **Random**.int 5 **in**  **let** tree = candidate i **in**  **let** n\_leaves = number\_of\_leaves\_v1 tree **and**  n\_nodes = number\_of\_nodes\_v1 tree **in**  n\_leaves = (n\_nodes + 1))    **and** b1 = **let** i = **Random**.int 10 **in**  **let** tree = candidate i **in**  **let** n\_leaves = number\_of\_leaves\_v1 tree **and**  n\_nodes = number\_of\_nodes\_v1 tree **in**  n\_leaves = n\_nodes + 1    **and** b2 = **let** i = **Random**.int 15 **in**  **let** tree = candidate i **in**  **let** n\_leaves = number\_of\_leaves\_v1 tree **and**  n\_nodes = number\_of\_nodes\_v1 tree **in**  n\_leaves = n\_nodes + 1    **and** b3 = **let** i = **Random**.int 20 **in**  **let** tree = candidate i **in**  **let** n\_leaves = number\_of\_leaves\_v1 tree **and**  n\_nodes = number\_of\_nodes\_v1 tree **in**  n\_leaves = n\_nodes + 1    **and** b4 = **let** i = **Random**.int 25 **in**  **let** tree = candidate i **in**  **let** n\_leaves = number\_of\_leaves\_v1 tree **and**  n\_nodes = number\_of\_nodes\_v1 tree **in**  n\_leaves = n\_nodes + 1    *(\* etc.\*)*  **in** b0 && b1 && b2 && b3 && b4;;  **val** test\_random\_binary\_tree : (int -> 'a binary\_tree) -> bool = <**fun**> |
| --- |

rHere is the implementation function which, given any integer n, generates a random binary tree of that height.

| **let** generate\_random\_binary\_tree\_int n =  **let** () = **assert** (n >= 0) **in**  **let** **rec** visit n =  **if** n = 0  **then** **Leaf** (**Random**.int 10)  **else** **match** **Random**.int 3 **with**  | 0 ->  **Leaf** (pred (- (**Random**.int 5)))  | \_ ->  **let** n' = pred n  **in** **Node** (visit n', visit n')  **in** visit n;; |
| --- |

We check if the implementation function passes the unit test.

| **let** () = **assert**(test\_random\_binary\_tree generate\_random\_binary\_tree\_int);; |
| --- |

It passes the unit test.

Conclusion:

This exercise makes us realise an interesting property about randomly generated binary trees. Its number of leaves will always be 1 greater than its number of nodes. Since a randomly generated binary tree can take any binary tree form, we can conclude that for any binary tree this property holds. This relation between number of leaves and nodes is intuitive as well: Ignoring the topmost leaf, every other node is accompanied with a leaf at its tail; only the topmost leaf does not have an accompanied node on top of it.

That said, it is also helpful to bear in mind that our unit test does not fully cover all possible binary trees. Strictly speaking, we have not rigorously proven that this relationship is true for every single binary tree. If we want a fully rigorous proof, we may have to rely on mathematics, since unit tests cannot practically cover every possible binary tree.

# Conclusion

This weekly handin covers vast ground and essential concepts from weeks 8 to 10, where we are introduced to lists, options, and binary trees and their accompanying functions and concepts. Implementing familiar functions like reversal, length computation to a new data type strengthens our perceptiveness towards thinking about the inductive specification of our data and how we can write structurally recursive functions to compute operations on them. Lists and tsils also extend our notions of data types and how we can translate between them, implementing functions for one or the other, by analysing their structures comparatively, which will be useful to us in our future programming journey as we venture into more complexity in data types that we face.