YSC1212: Introduction to Computer Science

Term Project: From Recursion to Data Structure

Jincong Chu, A0156290E
Zhi Yi Yeo, A0156253H
Chengpei Liu, A0136151R
Liu Zhang, A0190879J

Created [5 May 2020]
Edited based on Professor Danvy's comments [6 May 2020]

What if belly fat can recurse its way back...

Contents

1 Introduction

The term project marks the end of a semester long journey in YSC1212, a journey that will hopefully continue and follow us as we all proceed with our future semesters (and for some of us, life after college). In this term project, we review all that we have learnt over the past semester, focusing on the new knowledge gained in the second half of the semester.

We begin the final leg of our journey by examining the use of accumulators in programming with OCaml, specifically in the context of multiplying integers within binary trees. This is followed up with exercises that seek to enforce understanding in the traversal of binary trees and understanding how that relates to queues. We see this two concepts further play out in the next exercise with indexing different data structures, where we will make use of accumulators in some instances and also investigate how the traversal of binary trees relate to indexing it. Finally, we implement a series of functions that further deepen our understanding of OCaml (and programming in general), with the exercises on computing the width of a binary tree, implementing a sieve, the knapsack problem, and implementing language processors for arithmetic expressions.

2 Multiplying integers in a tree using an accumulator

In this section, we practiced implementing tree traversal with Depth First Search (DFS) and with the application of the concepts of accumulators and exceptions to make the DFS algorithm more efficient. Before we proceed with the exercises, we first clarify the concepts behind programming accumulator and exception. The goal of this project is to multiply integers stored in a given binary tree.

An accumulator is like a counter - both are variables that gather values as the program proceeds. Unlike counter which increments by one at each step, an accumulator allows more flexibility in that we can define the computational rule that governs how the accumulator increments. This enables us to compute partial results as and when the program proceeds. At the end of this project, we will discuss how by using accumulator the program becomes more efficient.

Programming with exception is a technique which is useful especially when we want to temporarily "bypass" certain cases (exceptions) and treat those cases later in the program. For example, programming with exception can allow the program to proceed nonetheless even when it encounters some error (such as division by zero).

2.1 Explain the unit test

Explain, in your own words, the structure and the rationale of the unit-test function in the accompanying file.

The unit-test function includes several Boolean variables which are our test cases. Within each test case, we pass the parameters to the function **test_mult_one**. These parameters include: name, our function to be implemented and tested, the name of the current test case, the correct result (of multiplying all integers in the tree), and the binary tree for testing in the current test case. In the function **test_mult_one** itself, we compare the result generated by our function, i.e., **actual_result** and the correct result, i.e., **expected_result**. If the two results do not match, we will print the message showing that the function failed the test. Otherwise, the unit-test function stays silent. We repeat this process for several test cases, i.e., to test our **mult** function with several different binary trees.

This way, we have achieved a silent unit-test for the mult function that we will be implementing in the following exercises.

2.2 Expand the unit test

Expand the unit-test function so that it does not just test the candidate function on one random binary tree of integers, but on a given number of random binary trees of integers.

To expand this unit-test function, we define a new test case, b8. In this test case, we randomly generate 4 (it can be any number as desired) different binary trees and test our mult function with them. This is achieved by using the repeat function. Note that thunk here is used to delay evaluation.

```
b8 = let thunk = (fun () -> let t = generate_random_binary_tree_int 10
                                    in test_mult_one name candidate_mult "b8" (mult_witness t) t
           in repeat 4 thunk
```

2.3 With an accumulator, without exceptions

(i) Implement one more version of the multiplication function over binary trees, mult_v7, using an accumulator and no exception.

In this part of the exercise, we rewrite the function using an accumulator and without exceptions. The accumulator has an initial value of 1 and it computes the result of multiplying numbers in the given binary tree.

Formally, the `mult` function is specified inductively, following the structure of binary trees. The visit function has two parameters: the first is the subtree that is yet to be traversed and the second is the accumulator computed thus far (that carries information about the tree that has been traversed and computed).

- **Base case:** The Leaf case is a number n. If the leaf is non-zero, the accumulator can be computed by a * n. If the Leaf is 0, then the value of the accumulator will become 0 regardless of its former value.
- **Induction step:** Given any subtree t1 and given any subtree t2, node(t1,t2) (the node connecting two subtrees t1 and t2) carries two pieces of information: **t1**, that is, the subtree that is yet to be traversed and **a**, the accumulator computed thus far after traversing the subtree t2. Note that this means that the traversal order is from right to left, which matches the mult_witness. More intermediate steps and explanations will be given later.

With the above inductive specifications, we implemented the `mult_v7` function with an accumulator and verified that it passed the unit-tests:

```
(* using an accumulator and no exception: *)

let mult_v7 t_given =
  let rec visit t a =
    match t with
    | Leaf n ->
       a * n
```

```
    | Node (t1, t2) ->
       visit t1 (visit t2 a)
  in visit t_given 1;;

let () = assert (test_mult "mult_v7" mult_v7);;
```

After being given the trace functions by Professor Danvy, we were prompted to observe and think about the difference between our initial `mult_v7` function and `mult_witness` function:

```
let mult_witness_tracing t_given =
  let rec visit t =
    Printf.printf "%s ->\n" (show_binary_tree show_int t);
    match t with
    | Leaf n ->
       n
    | Node (t1, t2) ->
       visit t1 * visit t2
  in visit t_given;;

let mult_v7_tracing t_given =
  let rec visit t a =
    Printf.printf "%s ->\n" (show_binary_tree show_int t);
    match t with
    | Leaf n ->
       if n = 0
       then 0
       else a * n
    | Node (t1, t2) ->
       visit t1 (visit t2 a)
  in visit t_given 1;;

let t = generate_random_binary_tree_int 4;;
mult_witness_tracing (t);;
mult_v7_tracing (t);;
```

And the traces produced were:

```
# mult_witness_tracing (t);;
Node (Node (Leaf (-1), Node (Leaf (-2), Node (Leaf 2, Leaf 4))), Node (Leaf (-1), Node (N
Node (Leaf (-1), Node (Node (Leaf 4, Leaf 2), Node (Leaf 3, Leaf 4))) ->
Node (Node (Leaf 4, Leaf 2), Node (Leaf 3, Leaf 4)) ->
Node (Leaf 3, Leaf 4) ->
Leaf 4 ->
Leaf 3 ->
Node (Leaf 4, Leaf 2) ->
Leaf 2 ->
```

```
Leaf 4 ->
Leaf (-1) ->
Node (Leaf (-1), Node (Leaf (-2), Node (Leaf 2, Leaf 4))) ->
Node (Leaf (-2), Node (Leaf 2, Leaf 4)) ->
Node (Leaf 2, Leaf 4) ->
Leaf 4 ->
Leaf 2 ->
Leaf (-2) ->
Leaf (-1) ->
- : int = -1536

-----------
(*
Trace when the node case in our mult_v7 is:
 | Node (t1, t2) ->
       visit t2 (visit t1 a)
*)

# mult_v7_tracing (t);;
Node (Node (Leaf (-1), Node (Leaf (-2), Node (Leaf 2, Leaf 4))), Node (Leaf (-1), Node (N
Node (Leaf (-1), Node (Leaf (-2), Node (Leaf 2, Leaf 4))) ->
Leaf (-1) ->
Node (Leaf (-2), Node (Leaf 2, Leaf 4)) ->
Leaf (-2) ->
Node (Leaf 2, Leaf 4) ->
Leaf 2 ->
Leaf 4 ->
Node (Leaf (-1), Node (Node (Leaf 4, Leaf 2), Node (Leaf 3, Leaf 4))) ->
Leaf (-1) ->
Node (Node (Leaf 4, Leaf 2), Node (Leaf 3, Leaf 4)) ->
Node (Leaf 4, Leaf 2) ->
Leaf 4 ->
Leaf 2 ->
Node (Leaf 3, Leaf 4) ->
Leaf 3 ->
Leaf 4 ->
- : int = -1536
```

An important realization is that the model function `mult_witness` traverses the tree from right to left whereas our initial `mult_v7` traverses from left to right, as illustrated by the figure below.

Based on this realization, we modified out mult_v7 by changing the order of t1 and t2 to match the order of traversal observed in the mult_witness. And we verified that this change resulted in the desired outcome, which is to produce the same trace as the mult_witness function:

```
(*
Trace after changing the order of t1 and t2, i.e., when the node case for our mult_v7 is:
```

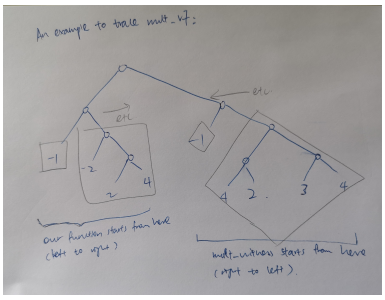

Figure 1: Comparing our initial mult_v7 with mult_witness

```
 | Node (t1, t2) ->
       visit t1 (visit t2 a)
*)
# mult_v7_tracing (t);;
Node (Node (Leaf (-1), Node (Leaf (-2), Node (Leaf 2, Leaf 4))), Node (Leaf (-1), Node (N
Node (Leaf (-1), Node (Node (Leaf 4, Leaf 2), Node (Leaf 3, Leaf 4))) ->
Node (Node (Leaf 4, Leaf 2), Node (Leaf 3, Leaf 4)) ->
Node (Leaf 3, Leaf 4) ->
Leaf 4 ->
Leaf 3 ->
Node (Leaf 4, Leaf 2) ->
Leaf 2 ->
Leaf 4 ->
Leaf (-1) ->
Node (Leaf (-1), Node (Leaf (-2), Node (Leaf 2, Leaf 4))) ->
Node (Leaf (-2), Node (Leaf 2, Leaf 4)) ->
Node (Leaf 2, Leaf 4) ->
Leaf 4 ->
Leaf 2 ->
Leaf (-2) ->
Leaf (-1) ->
- : int = -1536
```

(ii) Express your implementation using `fold_binary_tree` (which is defined in the accompanying file).

In this part of the exercise, we express the `mult_v7` function in terms of the recursive structure for binary trees defined in `fold_binary_tree`. By instantiating `leaf_case` to be `(fun n a -> a * n)` and `node_case` to be `(fun ih1 ih2 a -> ih2 (ih1 a))`, we make `fold_binary_tree` emulate the process of multiplying integers in any given binary tree:

```
let mult_v7_alt t_given =
  fold_binary_tree
    (fun n a -> a * n)
    (fun ih1 ih2 a -> ih1 (ih2 a))
```

```
    t_given 1;;
```

The detailed steps involved in arriving at the above are as follows:

First, we observed that the `fold_right_binary` allows only one parameter which is t_given. Thus, we need to find a way to put the accumulator inside the recursion and not as a parameter. Recall that the following two functions are equivalent (including being type-equivalent).

```
# let add x y = (x + y);;
val add : int -> int -> int = <fun>

# let add x = (fun y -> (x + y));;
val add : int -> int -> int = <fun>
```

Using this fact, we can rewrite the function such that the accumulator is inside the recursion by defining a function:

```
let mult_v7 t_given =
let rec visit t =
(fun a ->
  match t with
    | Leaf n ->
       a * n
    | Node (t1, t2) ->
        let ih1 = visit t1
       and ih2 = visit t2
        in ih1 (ih2 a)
  in visit t_given) 1;;
```

Then it is easier to see how this structure can be instantiated into the base case and the induction step of the `fold_right_binary`. We simply "distribute" the function and match the structure of the `fold_right_binary`:

```
let mult_v7 t_given =
let rec visit t =
  match t with
    | Leaf n ->
       (fun a -> a * n)
    | Node (t1, t2) ->
        let ih1 = visit t1
       and ih2 = visit t2
        in (fun a -> ih1 (ih2 a))
  in visit t_given 1;;

let fold_mult_v7 t_given =
        fold_binary_tree
        (fun n -> (fun a -> a * n))
        (fun ih1 ih2 -> (fun a -> ih1 (ih2 a)))
        t_given 1;;
```

Finally, for concision's sake, we can simplify this expression using the same fact as in the first step (but the backward implication). And this becomes our final solution for the `fold_right_binary` equivalent of the function `mult_v7`. We have also verified that this function passed the unit-tests:

```
let mult_v7_alt t_given =
  fold_binary_tree
    (fun n a -> a * n)
    (fun ih1 ih2 a -> ih1 (ih2 a))
    t_given 1;;

let () = assert (test_mult "mult_v7_alt" mult_v7_alt);;
```

2.4 With accumulators and exceptions

(i) Implement one more version of the multiplication function over binary trees, mult_v8, using an accumulator and an exception.

This version of the `mult` function has the same inductive specification as the previous exercise. The difference is that in this version programming with exceptions is involved.

The additional feature of programming with exceptions in this case is to raise an exception ("Zero") when encountering 0 in the leaf case.

For any given tree denoted by t_given, evaluating mult t completes (and yields an integer) if this tree does not contain 0. If this tree contains 0, the exception Zero is raised, caught at the initial call to visit, and then 0 is returned.

By raising the exception, the program becomes more time efficient. This is because once we encounter one integer zero anywhere in the recursion, we do not need to proceed any further because we would already know that the final product will be 0. Hence it saves time if we can directly exit the recursion and return the value 0.

We then verified that this function passed the unit-tests.

```
(* using an accumulator and an exception *)

let mult_v8 t_given =
  let rec visit t a =
    try
      match t with
      | Leaf n ->
          if n = 0
          then raise Zero
          else a * n
      | Node (t1, t2) ->
          visit t1 (visit t2 a)
    with
    | Zero -> 0
  in visit t_given 1;;
```

```
let () = assert (test_mult "mult_v8" mult_v8);;
```

(ii) Express your implementation using `fold_binary_tree` (which is defined in the accompanying file).

Similar to the previous exercise, we can rewrite the `mult` function in terms of the recursive structure defined in `fold_binary_tree`. This is achieved simply by instantiating the leaf case to be the base case for `fold_binary_tree` and the node case to be the induction step for `fold_binary_tree` using the same inductive specifications. The additional feature of programming with exception is included in the base case (leaf case), since our exception (encountering zeros in the binary tree) only concern what happen at the leaf of the given tree. We also verified that this function passed the unit-tests.

```
(* the fold_binary_tree version, using an accumulator and an exception *)

let mult_v8_alt t_given =
  try fold_binary_tree
        (fun n a -> if n = 0 then raise Zero else a * n)
        (fun ih1 ih2 a -> ih1 (ih2 a))
        t_given 1
  with
  |Zero -> 0;;

let () = assert (test_mult "mult_v8_alt" mult_v8_alt);;
```

2.5 Conclusion

We have seen a simple yet interesting application of accumulator and exception. Through these exercises we have learned to appreciate some of the motivations behind programming with accumulators and exceptions. We will now summarize our reflections.

The motivations behind using programming with accumulators include:

- It is more computationally efficient. Normal recursion will have to go step by step to the base case and compute the value only when it is going back up. Accumulator, on the other hand, allows us to compute simultaneously as the program proceeds.
- It is easier to control and program.

The motivations behind using programming with exceptions include:

- In the case where we have a known scenario where we are guaranteed a final answer before completing the entire recursion, programming with exceptions is useful to provide an short-cut exit from the recursion.
- We can group exceptions of the same types together and handle them together. It also makes it easier to recognize different types of exceptions,

3 Depth-first and breadth-first traversals

In this section, we investigate the depth first or breadth first traversals of our binary trees. We try to understand the relationships between them and our well-known fold binary tree functions. Eventually, we also apply functors to code modules that can conduct these traversals.

3.1 Extensional question 1: Analyze our examples

We are tasked to analyze the output of these tree-traversing functions ("traverse foo life", "traver foo fifo", "traverse bar lifo", and "traverse bar fifo") on the four given trees, t1 (3.1), t2 (3.1), t3 (3.1), and t4 (3.1).

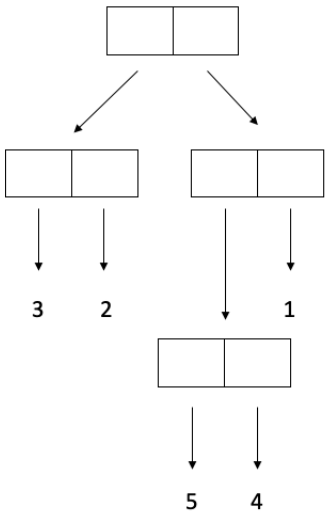

Figure 2: Tree t1

The output of the functions including the traced versions are in the submitted .ml file, and we shall not repeat here. Note that each traversal produces an "ordered" list (i.e. from 1 to 5) on one and only one tree. "traverse foo lifo" on t2. "traverse foo fifo" on t1. "traverse bar lifo" on t3. "traverse bar fifo" on t4. (How elegant!)

Let us then discuss the specifics on how each traversal passes trees.

(i) "traverse foo lifo": from left to right, disregarding the depth. Therefore, on t1, we have 3 before 2, 2 before 5, and 4 before 1. (Note that the last relationship holds despite 4 is of greater depth than 1.)

(ii) "traverse foo fifo": from right to left, the leaves with smaller depth first. Therefore, we have 1 before 2, and 4 before 5. However, 3 is before 4, because 3 is of smaller depth than 4, even though 4 is towards the right of 3.

(iii) "traverse bar lifo": from right to left, disregarding the depth. Reasoning similar as above.

(iv) "traverse bar fifo": from left to right, the leaves with smaller depth first. Reasoning similar as above.

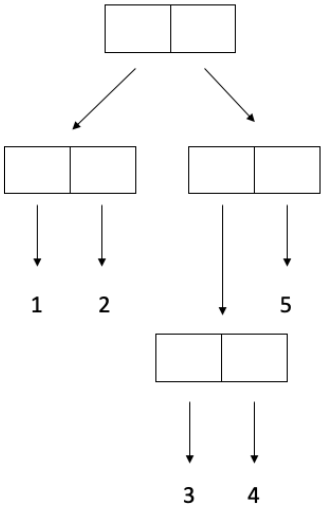

Figure 3: Tree t2

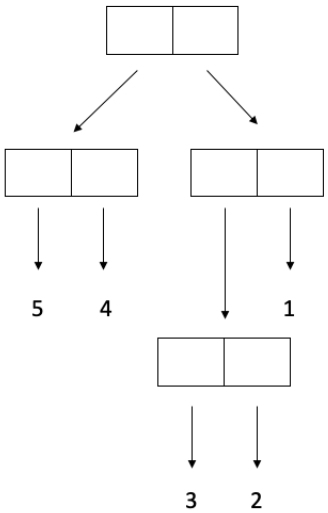

Figure 4: Tree t3

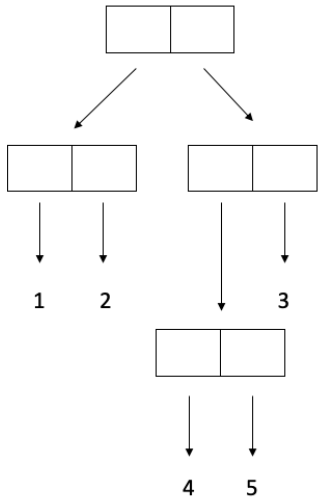

Figure 5: Tree t4

3.2 Extensional question 2: Depth first, breadth first, left to right, and right to left?

Based on our explanations in the previous question, we can categorize as follows:

(i) "traverse foo lifo": left to right and depth first.

(ii) "traverse foo fifo": right to left and breadth first.

(iii) "traverse bar lifo": right to left and depth first.

(iv) "traverse bar fifo": from left to right and breadth first.

3.3 Extensional question 3: More examples

The four examples are of the graphs below: t5 (3.3), t6 (3.3), t7 (3.3), and t8 (3.3).

For t5, we expect right-to-left depth first and right-to-left breadth first yield the same results $[4, 3, 2, 1]$. However, we do not yield the same result for the other two traversals (surprisingly). If we go left-to-right breadth first, we switch 1 and 2, while leaving 4 and 3 intact, i.e. $[4, 3, 1, 2]$. For last remaining one (left-to-right depth first), we have $[1, 2, 3, 4]$. Indeed, the results from OCaml are as expected.

For t6, we find it interesting to see that 5 will always be returned first on breadth first search, because 5 is the closest to the root. The rest of the numbers are returned depending on whether left-to-right or right-to-left.

For t7, the tree is obtained by rotating every single node of t5. Therefore, we see something similar here. The results for left-to-right breadth first and for left-to-right depth first are exactly the same. While the results for right-to-left breadth first and right-to-left depth first are different.

For t8, we construct it such that all traversals will return different results!

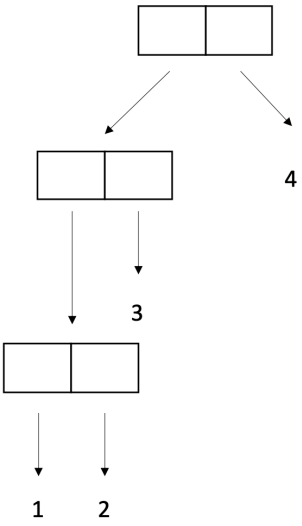

Figure 6: Tree t5

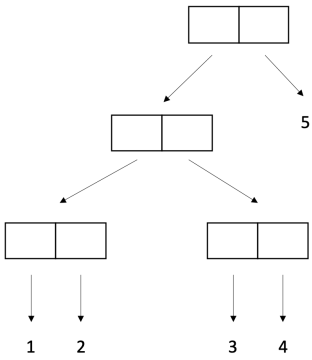

Figure 7: Tree t6

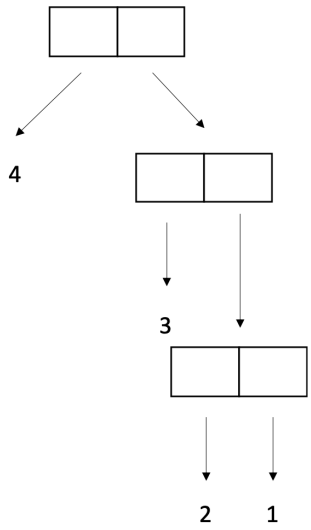

Figure 8: Tree t7

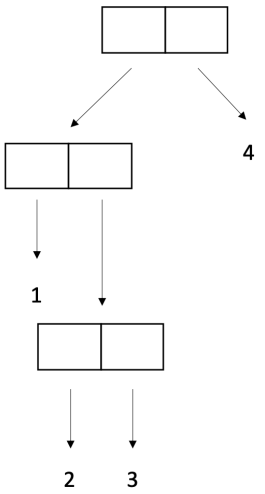

Figure 9: Tree t8

3.4 Subsidiary question 1: Why not structural recursive

In order for a function to be structurally recursive and written by fold binary tree, it must be described as a base case and then a successive case. However, if we look closely at traverse bar or traverse foo, we realize that we cannot successfully define the base and successive case.

The program first dequeue the queue, depending on whether we are at leaf or tree, we either process the leaf or enqueue the respective elements in the node. The presence of an unknown queue prevents us from defining the successive case meaningfully.

3.5 Subsidiary question 2: Modify traversals in structural recursive form

To answer all of the sub-questions together, it is not hard to see that fold binary tree is a depth-first operation, just that the depth-first character is hidden in the way OCaml processes our recursive function. We take the example of traverse foo lifo on t1, which we know is a left-to-right depth-first traversal.

```
val traced_traverse_foo_lifo_int : int binary_tree -> int list = <fun>
process_foo [Node (Node (Leaf 3, Leaf 2), Node (Node (Leaf 5, Leaf 4), Leaf 1))] ->
process_foo [Node (Leaf 3, Leaf 2); Node (Node (Leaf 5, Leaf 4), Leaf 1)] ->
process_foo [Leaf 3; Leaf 2; Node (Node (Leaf 5, Leaf 4), Leaf 1)] ->
  process_foo [Leaf 2; Node (Node (Leaf 5, Leaf 4), Leaf 1)] ->
    process_foo [Node (Node (Leaf 5, Leaf 4), Leaf 1)] ->
    process_foo [Node (Leaf 5, Leaf 4); Leaf 1] ->
    process_foo [Leaf 5; Leaf 4; Leaf 1] ->
      process_foo [Leaf 4; Leaf 1] ->
        process_foo [Leaf 1] ->
          process_foo [] ->
          process_foo [] <- []
        process_foo [Leaf 1] <- [1]
      process_foo [Leaf 4; Leaf 1] <- [4; 1]
    process_foo [Leaf 5; Leaf 4; Leaf 1] <- [5; 4; 1]
  process_foo [Leaf 2; Node (Node (Leaf 5, Leaf 4), Leaf 1)] <- [2; 5; 4; 1]
process_foo [Leaf 3; Leaf 2; Node (Node (Leaf 5, Leaf 4), Leaf 1)] <- [3; 2; 5; 4; 1]
- : int list = [3; 2; 5; 4; 1]
```

We see from the trace that OCaml processes the root note by breaking it down into two nodes. And the next step is to break down further on the left node (i.e. in this case Leaf 3 and Leaf 2). Therefore, leaf 3 and leaf 2 are evaluated earlier then the rest of the leaves. This reminds us of the fold binary tree operation, such that the respective value of the function is evaluated earlier before the function. Note that OCaml is call by value. Therefore, indeed, we can use the fold binary tree to represent traverse foo lifo. Similarly, we can do that for traverse foo lifo. The ideas are similar here that our base case is to process the leaf, while our success case is to connect the resulting list from the subtree.

We note that our programs pass the unit trees.

It is a lot more difficult to code breadth first using OCaml. Given the set up of our node, it does not seem possible to us. One idea we have to potentially do so is to record the depth of the leaf in each leaf beyond just the payload. In this way, we can recursively return the leaf with depth 1 and

then depth 2, until we cannot find any more leaves. However, given the current setup, we do not think it is possible.

3.6 Closing question

This question is relatively straightforward once we follow the definition of Functors as required. We just need to make sure that once Lifo or Fifo is applied to the functor, we can return a module that uses the respective traverse foo or traverse bar function.

```
module Traversal_maker =
  functor (M : QUEUE) ->
  struct
    let foo t =
  traverse_foo M.empty M.enqueue M.dequeue t
    let bar t =
      traverse_bar M.empty M.enqueue M.dequeue t
  end;;

module Traversal_lifo = Traversal_maker (Lifo);;
module Traversal_fifo = Traversal_maker (Fifo);;
```

We also coded the unit tests for each bar lifo, foo lifo, bar fifo, and foo fifo. Our module passes the unit tests.

3.7 Conclusion

We are delighted to have experience in coding functors towards the end of the section. Now we are equipped with modules that can run traversals of various types as we like. In this section, we gained a deeper understanding of depth-first and breadth-first traversals. We are left wondering under what conditions we should use which traversal? Is there one superior than the other in some cases?

Addition thanks to Professor Danvy's comments: Depth or width first depending on the nature of the problem at hand. For instance, to compute the maximum depth (i.e., the height) of a tree, i.e., the length of the longest path from the root of the tree to a leaf, depth-first is the natural choice. (Note the longest requirement here.) On the other hand, to compute the minimum depth of a tree, i.e., the length of the shortest path from the root of the tree to a leaf, breadth-first is the natural choice. (Note the shortest requirement here. We can stop the program once we found the first path in breadth-first search.) Also, when we want to compute the width of a tree, we need to use breadth-first search. If we want to find the left most leaf, then we need to use depth-first search.

4 Indexing different data structures

4.1 Indexing Strings

1. a) To index a string using String.get and String.length that returns type char option, we can use String.length to check for certain conditions before using String.get to obtain the indexed character. The conditions that we want to check for are 1) that n_given is not negative and 2) that n_given is shorter than the length of the string -1 (in other words, n_given **strictly less than** String.length of s_given; we refer to *Exercise 0* from Week 5) as to why). If these conditions are satisfied we return *Some* character as indexed using String.get, otherwise, we return *None*. The function passes the unit test.

   b) To index a string using String.get and an exception, we simply use the String.get function to index the string and catch the exception with the keywords *try* and *with*. If an exception is raised, we return *None*.

2. a) The idea here is similar to 1a. However, we need to input the index differently into String.get since we are indexing from right to left. The checks remain the same since the index should still not exceed String.length - 1. However, for the index input to String.get, we input String.length s_given - 1 - n_given instead of simply n_given. This is because we are indexing from the right - the first character from the right is essentially the last character from the left, the second character from the right is the second last character from the left (and so on). Hence, we take String.length s_given - 1 (because the indexing starts from 0 and not 1) - n_given while indexing from the right. Note that we first define len = String.length s_given in order to prevent multiple traversals of the string.

   b) The idea here is akin to combining parts 1b and 2a. We write a function similar in form to the function from 1b, but simply changed the way the string is index similar to the way described in 2a.

4.2 Indexing Arrays

All in all, the functions used to index arrays work in the same way as the functions used to index strings.

4.3 Indexing Lists

1. a) We note that this is similar to parts a of the previous sections.

   b) This question is also similar to 1b for strings and arrays, except that there are now 2 exceptions to account for indexing lists using List.nth as compared to only 1 exception for strings/arrays. For strings/arrays, the error for when n_given = -1 and n_given more than List.length - 1 is the same, but with lists these result in different errors that need to be accounted for in the *try* and *with* expression.

   c) We had multiple iterations of the function to index a list that is structurally recursive on the list. To start, we note that the base case and the successive cases had to return an option for the function to return the element we want. Hence, in most cases, the base case should return None. Then, for the first iteration of the function, we note that we can use the difference between List.length of the original list and List.length of the list of the

recursive call (vs) as a check to decide when to return the element of the list. However, we challenged ourselves to find a more elegant solution.

The next iteration uses an index ('a') that counts how many times the recursive function is called, starting from n_given decreasing each time the recursive function is called and returning the element when the index reaches 0. However, we note that we cannot write this version of the function using fold_right_list because an additional argument is used in the recursive function. We thus experiment with the use of pairs to include the index as one of the elements within a pair, with the other element of the pair being the output of the list (which we hope to obtain). We note here that this implementation requires the use of List.length, as we need to start the index from the length of the list and decrease it with each call of the recursive function. This is in contrast with the previous iteration, where we can start the index at n_given and decrease it with each iteration. This is because the prior function (v3) starts indexing the function starting from the first call to the recursive function (visit vs_given n_given), while the indexing in v4 only starts after the base case in the recursive function is called. This essentially means that if we were to start with an index of 0 or n_given, v3 starts indexing the list from left to right, whereas v4 starts indexing the list from right to left. Thus, we need to start the index at the length of the list for v4.

Hence, we endeavour to find a better solution that can still be expressed with fold_right_list. We note that in fact, the index (a) from v3 can be expressed within the recursive function visit itself, thus allowing it to be expressed in terms of fold_right_list.

d) We express v4 and v5 of our functions from part c using fold_right_list.

e) We note that to express the function recursively on the index, we would require the use of List.hd and List.tl such that the list being indexed changes together with the index (otherwise we would not be able to find the element that we want). The recursive function starts with the given index and the list, and decreases the index for each iteration of the recursive call. If the index has decreased to zero, we take the head of the List (which would be the element we want), otherwise the tail of the list is used for the next recursive call (so that the list, and also the head of that list, changes).

However, we note that this implementation is unable to account for when n_given is less than 0 or greater than the length of the list - 1. Thus, we include those checks at the start of the function. Since we are once again traversing the list twice (once with our recursive function and once with List.length), we note that there is definitely a more elegant way to do this and continue to adapt the function. In the next version of the function, we first check if the list we are indexing is the empty list before we proceed to take the head or tail of the list. We note the fact that if the list is an empty list before n reaches zero signifies that n is greater than the length of the list. If n was a negative number, the recursion would continue without the n = 0 condition being met, and will terminate once the list becomes an empty list. Thus, this removes the need to conduct checks on n_given outside of the recursive function. Finally, we note that the version of the recursive function still takes in two arguments, and thus adapt it to express the function that takes only one argument for the recursive function.

f) We express the function from part e using fold_right_nat. However, because the fold_right_nat function requires n to be positive, we check that n is positive outside of the expressed indexing function before proceeding.

2. a) We note that the solution here is the same as 2a of indexing strings and arrays.
   b) We note that indexing the list from right to left is the same as indexing the reverse of the same list from left to right. Thus, we call index_list_left_to_right on the reversed list to index the original list from right to left.
   c) Similarly, we start the with a version of the function that uses List.length to check when to return the output. In the recursive call, we note that when the length of the list within the recursive call (vs') is equal to n_given, then the element of the list (v) is what we want to return. We challenged ourselves to implement a function that does not make use of List.length. The next iteration of this function thus makes use of a pair, similar to 1c. The key difference is that the index starts from 0 at the base case, and increases with each call of the recursive function. As noted above, this is because the function traverses the list from left to right, with the index only initialized at the end of the traversal (which is at the right of the list).

      Another creative solution as proposed by Prof Danvy was to construct a new type 'a intermediate_result, which consists of the construct 'Found' of 'a and 'Still_looking' of int. Doing so allowed us to make use of an index (with the construct 'Still_looking') while also allowing us to return the value we want (with the construct 'Found') once the index fulfils the conditions. The parsimony of this solution (no need to use pairs where one element of the pair is not needed, no need for an additional index), makes it the most elegant one by far.
   d) We express the function from part c using fold_right_list.

   Subsidiary question: We note that since fold_right_list traverses the list from left to right while fold_left_list traverses the list from right to left, expressing the functions using fold_left_list instead of fold_right_list will result in a mirrored function. That is to say, a function that indexes from left to right will now index from right to left, and vice versa. We use the unit tests written for indexing lists to put this hypothesis to test in our code, and found it to be true.

4.4 Indexing Lazy Lists

1. a) We note that the implementation of this function is similar to that of a regular list, and provide 2 implementations. One where the recursive function takes 2 arguments, and the other where the index is expressed within the recursive function itself.
   b) We express the functions from part a using fold_right_llist.
   c) The implementation of the function is similar to question 1e for indexing lists. However, since there is no pre-defined function for accessing the head and tail of a lazy list, we start the function by writing an accessor to access the head and tail of the lazy list (similar to those used to access a stream).
   d) We express the function from 1c using fold_right_nat.

2. a) We note that the solution here is similar to question 2c of indexing lists, whereby we use a pair (where one element of the pair is an index and the other is the element we want returned from the lazy list) to index the lazy list.

      However, we note that it is possible for a lazy list to be infinite (like a stream). Thus, this implementation is possible with the added assumption that the lazy list to be indexed is finite in nature.

b) We express the function from part a using fold_right_llist.

4.5 Indexing Streams

1. In this case, we first begin by writing our own unit tests. We test our indexing function by creating a stream consisting of the natural numbers (From Week 10 of the lecture notes). The solution to index a stream is similar to that used to index a lazy list, where we use an index to count the number of times the recursive function is called and return the element (using a stream accessor) when the index reaches n_given. However, note that because a stream goes on infinitely, we need to check that n_given is positive before applying the function, otherwise the function would continue forever.

2. We note that since a stream goes on infinitely, we would not be able to index it from the right.

4.6 Indexing Binary trees

We start by writing unit tests for the different indexing functions in this section. To do this, we use the following binary tree (4.6):

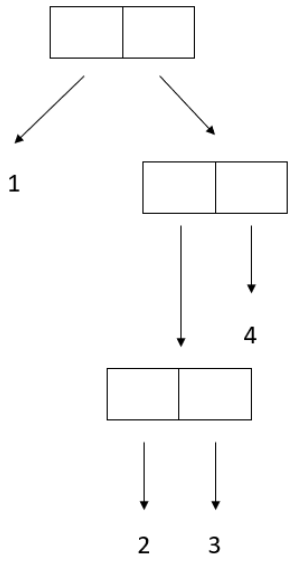

Figure 10: Binary tree for unit test

1. To index a tree depth first from left to right, we first need to understand what the output should be such that we can test it. Based on the exercise on queues and traversals, we know that the output for indexing the above tree depth first from left to right should return Some 1 for n_given 0, return Some 2 for n_given 1, return Some 3 for n_given 2, and return Some 4 for n_given 3. We code a unit test as such.

   We note in our first implementation that the most basic way to index the tree is to first traverse it to return a list (based on the traversal functions from the previous exercise), and then to index

the list as such. Since we want to index it depth first from left to right, we thus traverse it depth first from left to right.

However, we of course note that is simply a mechanism, not a solution. To provide a solution that is firstly not structurally recursive, we look to the traverse function from the previous exercise such that it not only traverses it, but also indexes it. Thus, instead of returning a list, it should return an option. We also note that the indexing methods should mirror the traversal methods - that is to say the enqueuing and dequeuing functions used should be last in, first out, if we were to index the tree depth first from left to right (same as the traversal method). Finally, we deduced that since the elements to be returned exist in the Leaf of a tree, the Leaf is where we should use an if-else statement to check for certain conditions before returning the output. The Nodes should thus serve as areas to continue the recursive call (and thus, continue traversing the tree until we reach a Leaf). Within the Leaf, we then use a counter (similar to previous implementations of indexing functions) to check whether a Leaf should be returned or whether a the function should continue traversing the tree.

Following feedback from Prof Danvy, we note that a more elegant solution is possible without the use of pairs, with the newly defined type intermediate_result. The underlying logic to a function written using intermediate_result is similar to the previous solution, apart from the fact that instead of returning a pair, we return an element with the construct 'Found' when the index conditions are met, while returning the index (and decreasing it) with the construct 'Still_looking' when conditions are not met.

Finally, we implement the function using an exception, "Found_it" of int. This exception allows us to further reduce the match conditions within the function, and is also more efficient as it stops the function once the element we are trying to index is found (as opposed to traversing the whole tree in previous cases).

The structurally recursive method to index a tree is somewhat different then simply traversing it. We first note that again, we would require the use of a counter to indicate when the element of the tree should be returned. In this case, we require the use of a pair, with the first element being the counter and the second element being the element from the leaf. In the leaf case, we check the counter to see if it is the index that we want to return, and if so, return the element of the Leaf, otherwise return None. In the Node case, we first traverse the left-most node and check if the output of the Leaf has been returned. If the output of the Leaf (Some v) has been returned, we can stop traversing the tree. Otherwise (None) is returned, and we continue to traverse the tree until a Leaf element is returned.

Finally, we express the structurally recursive version of the function using fold_binary_tree. All the functions pass the unit test.

2. To index a tree depth first from right to left, we note that the output returned should be Some 4 for n_given 0, Some 3 for n_given 1, Some 2 for n_given 2, and Some 1 for n_given 3. We code a unit test as such.

   We note that the implementation for the function is very similar to the previous function where we index the tree depth first from left to right apart from 1 difference. This difference being that as we index the tree from right to left, we enqueue the right-most node of the tree first instead of the left-most node of the tree. We include versions of the function that uses

methods from index_tree_depth_first_left_to_right (using the type intermediate_result, and using intermediate_result with an exception) as well.

Similarly, the structurally recursive function works in the same way as part 1 above, except that the right-most node is visited before the left-most node. We also express the function using fold_binary_tree. All the functions pass the unit test.

3. To index a tree breadth first from right to left, we note that the output returned should be Some 1 for n_given 0, Some 4 for n_given 1, Some 2 for n_given 2, and Some 3 for n_given 3. We code a unit test as such.

   We note that again, the implementation of the function is very similar to previous iterations. The key difference is the way in which the tree is traversed. In this case, since we are indexing the tree breadth-first from left to right we note that is similar to the traverse_bar_fifo function from the previous section, meaning that the Fifo (first in first out) module is used to enqueue and dequeue the tree, and that the right-most node is traversed first before the left-most node. The function passes the unit test.

4. To index a tree breadth first from right to left, we note that the output returned should be Some 1 for n_given 0, Some 4 for n_given 1, Some 3 for n_given 2, and Some 2 for n_given 3. We code a unit test as such.

   The version of the indexing function is similar to traverse_foo_fifo from the previous section, meaning that the Fifo module is used to enqueue and dequeue the tree, and the left most node is traversed before the right most node. The function passes the unit test.

4.7 Conclusion

From this section, we were able to learn the many different ways in which we can index a data structure, and the many different types of data structures we have encountered throughout our Intro to CS journey. We note that while there is a plethora of data structures, there are in many ways similar strands of logic behind indexing these different structures. An example - a common theme within indexing data structures is the need to traverse them, although the method of traversing them could differ between different data structures.

As such, this mini project has allowed us to greater appreciate the similarities and differences between these different structures, and help us put them together to form a more comprehensive understanding of the universe of OCaml (and programming, in general).

5 Computing the width of a binary tree

In this mini-project, we are tasked to code a function that computes the width of a binary tree. The width of a binary tree is the largest number of leaves and nodes at the same depth.

5.1 Exercise 1: Expand unit tests

We are using two characteristics of the width of the binary tree to expand our unit tests. First is its invariance given mirroring. Second is its invariance given change in payload. We conveniently used three functions that we have coded in the lecture notes coupled with a randomly generated binary tree as our unit tests. The three functions are: mirror (mirror the entire binary tree), distorted mirror (mirror the entire binary tree and make some random numbers negative), map binary tree (specifically, we are adding 1 to every single payload). The unit tests are shown below:

```
and b8 = (let t = generate_random_binary_tree_int 10
          in test_width_int_one
               name
               candidate_width
               "b8"
               (candidate_width (mirror t))
               t
         )
and b9 = (let t = generate_random_binary_tree_int 10
          in test_width_int_one
               name
               candidate_width
               "b9"
               (candidate_width (distorted_mirror_int t))
               t
         )
and b10 = (let t = generate_random_binary_tree_int 10
           in test_width_int_one
                name
                candidate_width
                "b10"
                (candidate_width (map_binary_tree succ t))
                t
          )
```

5.2 Exercise 2: Recursive width function

It is not easy to directly calculate the width of a tree recursively. In order to know the width of a tree, we need to know the maximum width of each layer of the tree. Therefore, we need to find a way to calculate the width of each layer recursively first, before we can find the maximum width.

We then realize that in the recursive function, we have to return a list of integers that keeps the width of each layer. When we reach a node, we can sum the two lists together to get the width of

each layer of this sub-tree (a tree with the node as its root). By doing so recursively, we can then get the width of each layer of the entire tree. The implementation is as follows.

We define the base case, i.e. the leaf case, as a list [1]. This denotes the fact that there is only one layer, and this layer has width 1 (only one leaf).

We define the successive case in two steps. First we take the two lists from each sub-trees and sum them pairwise. For instance, if we have two lists [1, 2 ,3] and [4, 5, 6, 7], we shall get [1+4, 2+5, 3+6, 7]. After summing the lists pairwise to get one list, this one list denotes the width of each layer up until the node case we are at now. We then insert an 1 at the beginning of the list to denote that at this node width position, we are only one (i.e. this node).

Once we finish this recursive function as we name it "visit". We then code another recursive function to get the maximum number in the generated list. We use similar ideas as the "array max" function that Professor provided us.

Hence, we get our function as shown below, and happily, it passes our unit tests.

```
let width_v1 t =
  (* width_v1 : 'a binary_tree -> int *)
  let rec sum_queue x1 x2 =
      match x1 with
      | [] ->
         x2
      | v1 :: vs1 ->
         match x2 with
         | [] ->
            x1
         | v2 :: vs2 ->
            v1+v2 :: (sum_queue vs1 vs2)
  in let rec visit t =
    1 :: match t with
    | Leaf v ->
       []
    | Node (t1, t2) ->
       sum_queue (visit t1) (visit t2)
  in let rec list_max max m =
    match m with
    | [] ->
       max
    | i :: m' ->
       if i>max then list_max i m' else list_max max m'
    in list_max 1 (visit t);;
```

5.3 Exercise 3: width function using fold binary tree

Given the previous exercise, this exercise is not difficult at all. We just change the syntax sugar to use the fold binary tree function. Thanks to our well-crafted thoughts in defining base and successive case from the previous exercise, this exercise is straightforward: we just change base case to leaf case and successive case to node case.

```
let width_v2 t =
 (* width_v2 : 'a binary_tree -> int *)
  let node_case x1 x2 =
    let rec sum_queue x1 x2 =
      match x1 with
      | [] ->
         x2
      | v1 :: vs1 ->
         match x2 with
         | [] ->
            x1
         | v2 :: vs2 ->
            v1+v2 :: (sum_queue vs1 vs2)
    in 1 :: (sum_queue x1 x2)
  in let rec list_max max m =
    match m with
    | [] ->
       max
    | i :: m' ->
       if i>max then list_max i m' else list_max max m'
    in list_max 1 (fold_binary_tree (fun x -> [1])  node_case t);;
```

Happily, this function passes our unit tests too!

5.4 Exercise 4: width function using array

This imperative implementation work in a similar way as our recursive function. The imperative implementation first aims to find the width of each layer in the entire tree. Then it finds the maximum width across different layers.

The difference between our implementations and Professor's implementation is in how to generate the width of each layer. Here, the imperative methods work in recording the depth i of each leaf or node, so that the program can add 1 to the respective depth. The process works as if we are finding the frequency table of a bunch of random numbers. We look through the random numbers, and once we see a number, we add 1 to the respective box in the frequency number. Eventually, we will get the number of nodes or leafs of that specific layer. The last step in finding the maximum is then pretty straightforward.

5.5 Conclusion

In this mini-project, we explored how to calculate the width of a binary tree. We learnt to think recursively in dealing with binary trees again. The key to code recursive functions, as we have learnt in this project, is to think about the base case and successive case. Usually the base case is not difficult, but we need to be careful with how we can get what we want in the successive case. Here, base case refers to the leaf case, while the successive case refers to the node case.

6 Implementing a sieve

6.1 Strike out the $(n+2)^{th}$ element of a stream

We start with a unit test that includes the six scenarios provided in the question. They include operating on streams of natural numbers, positive natural numbers. For each of the scenario, we use *prefix_stream* to check the first part of the outcome stream with the expected outcome.

In the main function, we first test if the input is valid. Then we have a counter i in our recursion. The counter starts at $n+2$, the frequency at which we want to strike. When the counter hits 1, it gets back to $n+2$ and carries on forever. When the counter is not 1, we records the head of the current stream and move on by reducing the counter by 1 and operating on the tail of the stream.

```
let strike_out s n =
  let () = assert (n >= 0) in
  let rec visit i (Scons (v, s')) =
    if i = 1 then visit (n+2) (Lazy.force s')
    else Scons(v, lazy(visit (pred i) (Lazy.force s')))
  in visit (n+2) s;;
```

6.2 Exercise 2: Construct a stream of sums

For later use, we construct a few standard streams including alternating 1 and -1, alternating 1 and 0, squares, alternative 1 and 0, all ones and all zeros. Next, we write a unit test that includes the 4 scenarios provided in the notes.

To use an accumulator, in the main function that we call *partsum*, we design a recursion function that passes 2 streams. The first stream is the input stream, the second stream is the accumulator stream that starts from all zeros. v is the head of the input stream. After processing v, we will discard it from the stream. h is the current rightmost sum. In each iteration, we always add the head of the new stream with h.

```
let partsum s =
  let rec visit (Scons (v, s')) (Scons (h, t)) =
    Scons(v + h, lazy(visit(Lazy.force s') (make_stream (v+h) (fun i -> 0))))
  in visit s zeros;;
```

6.3 Exercise 3: Combine the last 2 parts

We modify the unit test from the last exercise slightly and feed it with expected answers provided by the note and manually calculated.

The main function is a simple call of the two components connected by *in*.

```
let test_composite candidate =
  let b0 = (prefix_stream (candidate posneg1 0) 20 = prefix_stream posnats 20)
  and b1 = (prefix_stream (candidate onezeros 5) 20 = prefix_stream ones 20)
  and b2 = (prefix_stream(candidate ones 3) 20 = prefix_stream posnats 20)
  and b3 = (prefix_stream (candidate odds 0) 5 = [1;6;15;28;45])
  in b0 && b1 && b2 && b3;;
```

```
let composite s n =
  let a = strike_out s n
  in partsum a;;

let () = assert (test_composite composite);;
```

6.4 Exercise 4: Paring an onion

We write a unit test that includes both calculated answers and analogue answers derived by from writing out the onion without using recursion. By observing the resulting streams when the initial stream s is one followed by zeros and $k+1$ is the number of layers of the union (inclusive of the 0^{th} layer), we propose that $sievesk = [1^k, 2^k, 3^k, 4^k ...]$. We test this proposition with a randomized scenario in the unit test. For the randomized scenario, we prepare a function that produces a list with a given length of natural numbers raised to the power of a given number.

Let us bring in the concept of code coverage. Code coverage refers to how much of our source code is tested. It could be measured by the lines of code, the number of functions or the number of branches (e.g. if statements) covered. Besides our main function *sieve*, this unit test covers functions including $prefix\_stream$, *composite*, $strike\_out$, *scan* and *powers*. In other words, all the pre-defined functions in this mini-project are covered. The input streams tested include *onezeros* and *posneg*1 with varying length. We note that *Sieve* does not have branches. To summarize, our unit test has good code coverage in terms of functions covered, but could improve on the variation of the input streams. Having said that, this improvement could be manual and laborious. It's more of an art than science to find the balance between enough input variations and coding efficiency.

The main function is simple. We have a counter that counts from 0 to k and adds one layer to the onion for each iteration.

```
let powers m n =
  let rec visit base power =
    match base with
    | 0 -> []
    | i -> (pow (m+1-i) power):: visit (i-1) power
  in visit m n;;

let test_sieve candidate =
  let b0 = (prefix_stream (candidate onezeros 0) 20 =prefix_stream (composite onezeros 0) 20)
  and b1 = (prefix_stream (candidate onezeros 1) 20 =prefix_stream posnats 20)
  and b2 = (prefix_stream (candidate onezeros 2) 20 =prefix_stream squares 20)
  and b3 = (prefix_stream (candidate posneg1 3) 20 =prefix_stream (composite (composite (compo
  and b4 = (prefix_stream (candidate onezeros 4) 10 = [1; 16; 81; 256; 625; 1296; 2401; 4096;
  and b5 = let n = Random.int 100 in (prefix_stream (candidate onezeros n) n = powers n n)
in b0 && b1 && b2 && b3 && b4 && b5;;

let sieve s k =
  let () = assert (k >= 0) in
  let rec visit s i =
    if i = k
```

```
    then composite s i
    else composite (visit s (succ i)) i
  in visit s 0;;

let () = assert (test_sieve sieve);;
```

6.5 Exercise 5: 3 cases

a. Using the following unit case, we propose that *sieve* (1,1,1...) n = *sieve* (1,0,0...) $(n+1)$

```
let test_sieve_a candidate =
  let b1 = (prefix_stream (candidate ones 0) 20 =
    prefix_stream posnats 20)
  and b2 = (prefix_stream (candidate ones 1) 20 =
    prefix_stream squares 20)
  and b3 = (prefix_stream (candidate ones 2) 20 =
    prefix_stream (sieve onezeros 3) 20)
  and b4 = (prefix_stream (candidate ones 3) 10 =
    [1; 16; 81; 256; 625; 1296; 2401; 4096; 6561; 10000])
  and b5 = (prefix_stream (candidate posneg1 3) 20 =
    prefix_stream (composite (composite (composite (composite posneg1 3) 2)
    1)0) 20)
  and b6 = let n = Random.int 100 in (prefix_stream (candidate ones n) 20 =
    prefix_stream (sieve onezeros (n+1)) 20)
  in b1 && b2 && b3 && b4 && b5 && b6;;

let () = assert (test_sieve_a sieve);;
```

b. Using the following unit case, we propose that *sieve posnats* n = *sieve onezeros* $(n+2)$.

```
let test_sieve_b candidate =
  let b0 = (prefix_stream (candidate posnats 0) 10 =
    prefix_stream (candidate onezeros 2) 10)
  and b1 = (prefix_stream (candidate posnats 1) 10 =
    prefix_stream (candidate onezeros 3) 10)
  and b2 = (prefix_stream (candidate posnats 2) 10 =
    [1; 16; 81; 256; 625; 1296; 2401; 4096; 6561; 10000])
  and b3 = (prefix_stream (candidate posneg1 3) 20 =
    prefix_stream (composite (composite (composite (composite posneg1 3) 2)
    1) 0) 20)
  and b4 = let n = Random.int 100 in (prefix_stream (candidate posnats n) 20 =
    prefix_stream (sieve onezeros (n+2)) 20)
in b0 && b1 && b2 && b3 && b4;;

let () = assert (test_sieve_b sieve);;
```

c. From part b, we know that *sieve posnats* 0 produces a stream of square numbers. Therefore, operating *sieve* on the square stream is equivalent to adding another layer of *sieve* to *sieve*

posnats 0. We also know that *sieve posnats* 0 = *sieve onezeros* 2. Thus, we propose *sieve squares* n = *sieve*(*sieve onezeros* 2) n. We verify the proposition using the following unit test.

```
let test_sieve_c candidate =
  let b0 = (prefix_stream (candidate squares 0) 10 =
    prefix_stream (candidate(candidate onezeros 2) 0) 10)
  and b1 = (prefix_stream (candidate squares 1) 10 =
    prefix_stream (candidate(candidate onezeros 2) 1) 10)
  and b2 = (prefix_stream (candidate squares 2) 10 =
    prefix_stream (candidate(candidate onezeros 2) 2) 10)
  and b3 = let n = Random.int 100 in (prefix_stream (candidate squares n) 10 =
    prefix_stream (candidate(candidate onezeros 2) n) 10)
  in b0 && b1 && b2 && b3;;

let () = assert (test_sieve_c sieve);;
```

6.6 Exercise 6: 3 more cases

Using the following unit test, we propose:

- sieve tenzeros n scales each element of "sieve onezeros n" by 10
- sieve twozeros n" scales each element of "sieve onezeros n" by 2
- sieve xzeros n" scales each element of "sieve onezeros n" by x

```
let test_sieve_6 candidate =
  let b0 = (prefix_stream (candidate tenzeros 0) 10 =
    tentimes(prefix_stream (candidate onezeros 0) 10))
  and b1 = (prefix_stream (candidate tenzeros 1) 10 =
    tentimes(prefix_stream (candidate onezeros 1) 10))
  and b2 = (prefix_stream (candidate twozeros 2) 10 =
    twice(prefix_stream (candidate onezeros 2) 10))
  and b3 = (prefix_stream (candidate hundzeros 3) 10 =
    hundtimes(prefix_stream (candidate onezeros 3) 10))
  in b0 && b1 && b2 && b3;;

let () = assert (test_sieve_6 sieve);;
```

6.7 Exercise 7: Another 3 more cases

Using the following unit test, we propose:

- Consider in (1,3,5,7...), the index of each element starts from 0, then
- sieve onetwos n scales each element of (1,3,5,7...) by the index raised to the power of n.
- Consider in (1,11,21,31...), the index of each element starts from 0, then

- sieve onetens n scales each element of (1,11,21,31...) by the index raised to the power of n.
- Consider in (1,101,201,301...), the index of each element starts from 0, then
- sieve onehunds n scales each element of (1,101,201,301...) by the index raised to the power of n.

```
let onetwos = make_stream 1 (fun i-> 2);;
let onetens = make_stream 1 (fun i-> 10);;
let onehunds = make_stream 1 (fun i -> 100);;

let test_sieve_7 candidate =
  let b0 = (prefix_stream (candidate onetwos 0) 10 =
    prefix_stream odds 10)
  and b1 = (prefix_stream (candidate onetwos 1) 10 =
    List.mapi (fun n a-> (n+1)*a )(prefix_stream odds 10))
  and b2 = (prefix_stream (candidate onetwos 2) 10 =
    List.mapi (fun n a -> (n+1)*(n+1)*a)(prefix_stream odds 10))
  and b3 = (prefix_stream (candidate onetwos 3) 10 =
    List.mapi (fun n a -> a* pow(n+1) 3)(prefix_stream odds 10))
  and b4 = (prefix_stream (candidate onetwos 4) 10 =
    List.mapi (fun n a -> a* pow(n+1) 4)(prefix_stream odds 10))
  and b5 = (prefix_stream (candidate onetens 4) 10 =
    List.mapi (fun n a -> a* pow(n+1) 4)(prefix_stream (make_stream 1 (fun i -> i+10)) 10))
  and b6 = (prefix_stream (candidate onehunds 4) 10 =
    List.mapi (fun n a -> a* pow(n+1) 4)(prefix_stream (make_stream 1 (fun i -> i+100)) 10))
  in b0 && b1 && b2 && b3 && b4 && b5 && b6;;

let () = assert (test_sieve_7 sieve);;
```

6.8 Exercise 8: Generalize the pattern

Using the following unit test, we can propose a generalization: when the input stream consists of one followed by ks, consider in a stream (1, 1+k, 1+2k, 1+3k...), the index of each element starts from 0, then sieve s n scales each element of (1, 1+k, 1+2k, 1+3k...) by the index raised to the power of n.

```
let test_sieve_8 candidate =
      let b0 = (prefix_stream (candidate (make_stream 1 (fun i-> 5)) 3) 10 =
        List.mapi (fun n a -> a* pow (n+1) 3) (prefix_stream (make_stream 1 (fun i -> i+5))
        10))
      and b1 = (prefix_stream (candidate (make_stream 1 (fun i-> 6)) 9) 10 =
        List.mapi (fun n a -> a* pow (n+1) 9) (prefix_stream (make_stream 1 (fun i -> i+6))
        10))
      and b2 = (prefix_stream (candidate (make_stream 1 (fun i-> 4)) 5) 10 =
        List.mapi (fun n a -> a* pow (n+1) 5) (prefix_stream (make_stream 1 (fun i -> i+4))
        10))
      in b0 && b1 && b2;;

    let () = assert (test_sieve_8 sieve);;
```

7 Knapsack Problem

7.1 Without *fold_right_list*

We start by converting the 4 examples provided in the notes into a unit test. The unit test works for both *minimize_assessed_powerset_v*0 and *minimize_assessed_powerset_v*1.

```
let test_minimax candidate =
   let b0 = (candidate 2 [("diamond", 10, 100); ("ruby1", 7, 70); ("ruby2", 7, 70);
    ("jade", 3, 30)]  = (0, [(0, 0, [])]))
   and b1 = (candidate 5 [("diamond", 10, 100); ("ruby1", 7, 70); ("ruby2", 7, 70);
    ("jade", 3, 30)] = (30, [(3, 30, [("jade", 3, 30)])]))
   and b2 = (candidate 8 [("diamond", 10, 100); ("ruby1", 7, 70); ("ruby2", 7, 70);
    ("jade", 3, 30)] = (70, [(7, 70, [("ruby1", 7, 70)]); (7, 70, [("ruby2", 7, 70)])]))
   and b3 = (candidate 10 [("diamond", 10, 100); ("ruby1", 7, 70); ("ruby2", 7, 70);
    ("jade", 3, 30)] = (100, [(10, 100, [("diamond", 10, 100)]); (10, 100,
    [("ruby1", 7, 70); ("jade", 3, 30)]); (10, 100, [("ruby2", 7, 70); ("jade", 3, 30)])]))
   in b0 && b1 && b2 && b3;;
```

Given the code template, our job is to produce a new *minimize_assessed_powerset_v*1. We will modify *minimize_assessed_powerset_v*0 using a maximum weight constraint.

We use the same concept as *minimize_assessed_powerset_v*0 by also employing an outer recursion and an inner recursion. The outer recursion separates the current object from the constrained sets made from the objects on the right. The inner recursion separates the the first set on the right from the other sets.

We start with determining the base case for the outer recursion. Then, for other cases, we separate the first object. We get the legal sets from objects on the right by $outer\_ih = outer\_visit\ vs'$.

Now, we are ready to enter the inner recursion. The legal sets are the input for the inner recursion which separates the first set from the rest. In the base case, when the remaining legal sets are empty, we return to the outer recursion. In other cases, we check if we can add the current object to the first set, using the weight constraint. If adding it meets the weight constraint, we will also update the total weight and total value. Note that the attempt to add the current object to sets doesn't stop here. We use *inner_ih* to continue the inner recursion and attempt adding the current object to all legal sets. When all is done, we return to the outer recursion and work on the next object.

The rest is easy, we find the legal package with the best value by calling *maximize_the_monetary_value* in the main function.

Our main function passes the unit test.

```
let minimize_assessed_powerset_v1  given_weight given_vs =
  let rec outer_visit vs =
    match vs with
    | [] ->
       [(0,0,[])]
    | (v,weight1,monetary_value1) :: vs' ->
       let outer_ih = outer_visit vs'
       in let rec inner_visit wss =
            match wss with
```

```
            | [] ->
               outer_ih
            | (weight2,monetary_value2, ws) :: wss' ->
               let inner_ih = inner_visit wss'
               in if (weight1+weight2 ) <=given_weight
                  then ( weight1+ weight2, monetary_value1+monetary_value2,
                    (v,weight1,monetary_value1) :: ws) :: inner_ih
                  else inner_ih
          in inner_visit outer_ih
  in outer_visit given_vs;;


let minimax_v1 given_weight given_triples =
  maximize_the_monetary_value
    (minimize_assessed_powerset_v1 given_weight given_triples);;

let () = assert(test_minimax minimax_v1);;
```

7.2 With *fold_right_list*

Let us take another look to refresh our memory of *fold_right_list*. It takes the base case, a list, continuation case, a function, and an initial input, a list. When the input list is recursed all the way to the bottom, we will receive the base case; otherwise, we will continue going down by feeding the head of the current list to the *ih* of the rest of the list.

```
let fold_right_list nil_case cons_case vs_init =
 (* fold_right_list : '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_init;;
```

Now, we move on to placing *fold_right_list* into *minimize_assessed_powerset*. It is probably the first time that we have 2 layers of recursion and need to use *fold_right_list*. We know for sure that for the outer layer, the base case for *fold_right_list* should be [(0,0,[ ])] and the initial input list will be *given_vs*. The challenge is in *cons_case*1. *cons_case*1 should take *v ih*, as we have observed in *fold_right_list*. *v* is the first triple in the list of triples. We proceed to the *ih*′ layer. Here we again need to separate the first triple from the rest of the list. Then we check if the first triple in the outer layer can be added to the knapsack. If not, *ih*′ doesn't change. If yes, we add the triple to the current package.

```
let minimize_assessed_powerset_v2  given_weight given_vs =
  let cons_case1 v ih =
```

```
  let (v1, weight1, monetary_value1)=v
  in let cons_case2 v' ih' =
       let (weight2, monetary_value2,v2)=v'
       in if (weight1+weight2 ) <=given_weight
                then (weight1+ weight2, monetary_value1+monetary_value2,(v1,weight1,
                monetary_value1) :: v2) :: ih'
                else ih'
     in fold_right_list ih cons_case2 ih
in fold_right_list [(0,0,[])] cons_case1 given_vs;;
```

8 Three language processors for arithmetic expressions

8.1 Introduction

In this mini-project, we fulfilled an earlier anticipation for the module: to build an interpreter and compiler for arithmetic expressions. In what follows we will explain our implementations and share some interesting observations.

8.2 Task 1: Implement the source language interpreter

Implement an interpreter for the source language as an OCaml function that satisfies the specification in the project descriptions.

The first task is to implement an interpreter for the source language. In this process, we make use of the match expressions to implement the given specifications of the interpreter. Note that our implementation mirrors the specification (almost exactly) and each of the induction steps have the same structure. Concretely, we evaluate the input expression `e` by matching it with the following cases:

- **Base case**: for any integer n (noting its syntactic representation as n), evaluating Literal n yields the integer n. And our implementation is:

```
| Literal n -> Expressible_int n
```

- **Induction step for additions:** for any arithmetic expression e1 that was evaluated as the expressible value ev1 (which is the first induction hypothesis) and any arithmetic expression e2 that was evaluated as the expressible value ev2 (which is the second induction hypothesis). For this our implementation is:

```
| Plus (e1, e2) ->
       (match evaluate e1 with
        | Expressible_int n1 ->
           (match evaluate e2 with
            | Expressible_int n2 -> Expressible_int (n1 + n2)
            | Expressible_msg s -> Expressible_msg s)
        | Expressible_msg s -> Expressible_msg s)
```

- **Induction step for subtractions:** mirroring the structure for the induction step for additions, we have

- **Induction step for quotients:** the implementation, again, mirrors the structure for additions and subtractions, except an additional step to determine if there is an error of division by zero (if there is, we will generate an error message; otherwise we will proceed with the usual computation).

- **Induction step for remainders:** this induction step mirrors that of the quotient, in which we will generate the error message of division by zero.

```
| Remainder (e1, e2) ->
       (match evaluate e1 with
```

```
            | Expressible_int n1 ->
               (match evaluate e2 with
                | Expressible_int n2 ->
                   if n2 = 0 then
                     (Expressible_msg ("remainder of " ^ string_of_int n1 ^ " over 0"))
                   else Expressible_int (n1  mod n2)
                | Expressible_msg s -> Expressible_msg s)
            | Expressible_msg s -> Expressible_msg s)
```

Finally, we verified that our function passed the unit-test function:

```
let () = assert (int_test_interpret "interpret" interpret);;
let () = assert (msg_test_interpret "interpret" interpret);;
```

Subsidiary questions:

(i) Does your interpreter evaluate from left to right or from right to left?

We experimented with Ocaml and discovered that our interpreter evaluates from left to right:

```
interpret (Source_program (Plus (Literal (an_int 1), Literal (an_int 0))));;
processing 0...
processing 1...
- : expressible_value = Expressible_int 1

interpret (Source_program (Plus(Quotient(Literal (an_int 10), Literal (an_int 0)),
    Quotient(Literal (an_int 100), Literal (an_int 0)))));;

processing 0...
processing 100...
processing 0...
processing 10...
- : expressible_value = Expressible_msg "quotient of 10 over 0"
```

In the above we need to make a distinction between the evaluation order for arithmetic operations in OCaml (which is from right to left) and the evaluation order of our interpreter (which is from left to right). This is made clear by the second example, where the message is printed for the first instance of division by zero, which means it evaluates the expression on the left `(Quotient(Literal (an_int 10), Literal (an\_int 0)))` before the expression on the right `Quotient(Literal (an_int 100), Literal (an_int 0))`.

(ii) Does it make any observable difference whether your interpreter evaluates from left to right or from right to left?

As illustrated by the first example, when the expression is pure (i.e., does not produce any side effects), the evaluation order of the interpreter does not make any observable difference, for instance, the case where the expression evaluates to an output of the type `Expressible_int`. As illustrated by the second example, when the expression is impure (i.e., produces any side effects), the evaluation order of the interpreter will make observable difference, for instance, the case where the expression evaluates to an output of the type `Expressible_msg`.

8.3 Task 2: Implement a one byte-code instruction processor

Implement a processor for one byte-code instruction as an OCaml function that satisfies the specification in the project descriptions.

In this task, we implemented a function that can process the byte-code instructions. Our problem-solving approach was the same as the previous exercise: follow the instructions, that is, the structure of our implementation mirrored that of the given specifications. The function was specified by cases, following the structure of byte-code instructions:

- **Push n**, where n represents the integer n: for any data stack ds (represented as ds), processing the byte-code instruction Push n consists in pushing n on top of ds and returning the resulting stack, i.e., n :: ds. Our implementation is as follows:

```
| Push n -> OK ( n:: ds)
```

- **Add.** The implementation for this case follows from the byte-code instructions for addition.

```
| Add -> (match ds with
          |  ( n1 ::  n2 :: ds') -> OK ( (n1+n2) :: ds')
          | _ -> KO "stack underflow for Add")
```

- **Sub.** The implementation for this case follows from the byte-code instructions for subtraction.

```
| Sub -> (match ds with
          | (n1 :: n2 :: ds') -> OK( (n2 - n1) :: ds')
          | _ -> (KO "stack underflow for Sub"))
```

- **Quo.** The implementation for this case follows from the byte-code instructions for quotient.

```
| Quo -> (match ds with
           | ( n1 ::  n2 :: ds') ->
              if n1 = 0 then KO ( "quotient of " ^ string_of_int n2 ^ " over 0")
              else OK( (n2 / n1) :: ds')
           | _ -> (KO "stack underflow for Quo"))
```

- **Rem.** The implementation for this case follows from the byte-code instructions for remainder.

```
| Rem -> (match ds with
           | ( n1 ::  n2 :: ds') ->
              if n1 = 0 then KO ( "remainder of " ^ string_of_int n2 ^ " over 0")
              else OK( (n2 mod n1) :: ds')
           | _ -> (KO "stack underflow for Rem"))
```

Finally, we verified that our function passed the unit-test function:

```
let () = assert (test_decode_execute "decode execute" decode_execute);;
```

8.4 Task 3: Implement the target language interpreter

Implement an interpreter for the target language (i.e., a virtual machine) as an OCaml function that satisfies the specification just above.

(Solution is given for this task.)

8.5 Task 4: Implement a compiler

Implement a compiler from the source language to the target language as an OCaml function that satisfies the specification in the project descriptions.

In this exercise, we used list concatenation to implement each case in the function.

- **Base case:** for any integer n (noting its syntactic representation as n), translating Literal n yields the singleton list containing the byte-code instruction**Push n**. Following this specification, our implementation is as follows:

  ```
  | Literal n -> [Push n]
  ```

- **Induction step for additions:** for any arithmetic expression e1 (represented as e1) that was translated as the list of byte-code instructions bcis1 (which is the first induction hypothesis) and any arithmetic expression e2 (represented as e2) that was translated as the list of byte-code instructions bcis2 (which is the second induction hypothesis), translating Plus (e1, e2) yields the list of byte-code instructions that begins with the instructions in bcis1, continues with the instructions in bcis2, and ends with the **Add** instruction.

  In our implementation, we simply append the translated expressions to the list, with the **add** instruction. Concretely,

  ```
  | Plus (e1, e2) ->
    List.append (List.append (translate e1)  (translate e2) [Add]
  ```

- **Induction step for subtractions:** for any arithmetic expression e1 (represented as e1) that was translated as the list of byte-code instructions bcis1 (which is the first induction hypothesis) and any arithmetic expression e2 (represented as e2) that was translated as the list of byte-code instructions bcis2 (which is the second induction hypothesis), translating Minus (e1, e2) yields the list of byte-code instructions that begins with the instructions in bcis1, continues with the instructions in bcis2, and ends with the **Sub** instruction. Our implementation reflected this specification:

  ```
  | Minus (e1, e2) ->
    List.append (List.append (translate e1)  (translate e2)) [Sub]
  ```

- **Induction step for quotients:** for any arithmetic expression e1 (represented as e1) that was translated as the list of byte-code instructions bcis1 (which is the first induction hypothesis) and any arithmetic expression e2 (represented as e2) that was translated as the list of byte-code instructions bcis2 (which is the second induction hypothesis), translating Quotient (e1, e2) yields the list of byte-code instructions that begins with the instructions in bcis1, continues with the instructions in bcis2, and ends with the **Quo** instruction. Our implementation reflected this specification:

```
| Quotient (e1, e2) ->
  List.append(List.append (translate e1)  (translate e2)) [Quo]
```

- **Induction step for remainders:** for any arithmetic expression e1 (represented as e1) that was translated as the list of byte-code instructions bcis1 (which is the first induction hypothesis) and any arithmetic expression e2 (represented as e2) that was translated as the list of byte-code instructions bcis2 (which is the second induction hypothesis),

  translating Remainder (e1, e2) yields the list of byte-code instructions that begins with the instructions in bcis1, continues with the instructions in bcis2, and ends with the **Rem** instruction. Our implementation reflected this specification:

```
| Remainder (e1, e2) ->
  List.append(List.append (translate e1)  (translate e2)) [Rem]
```

Finally, we verified that our function passed the unit-test function:

```
let () = assert (test_compile "compile" compile);;
```

Subsidiary questions:

- Does your compiler translate from left to right or from right to left?

  The compiler translates from left to right because of the use of List.append(). To verify, we use the same examples as in Task 1. It is clear from the output that the compiler translates the source program to a list of byte-code instructions (BCI) from left to right.

```
compile (Source_program (Plus (Literal (an_int 1), Literal (an_int 0))));;
processing 0...
processing 1...
- : target_program = Target_program [Push 1; Push 0; Add]

compile (Source_program (Plus(Quotient(Literal (an_int 10), Literal (an_int 0)),
    Quotient(Literal (an_int 100), Literal (an_int 0)))));;

processing 0...
processing 100...
processing 0...
processing 10...
- : target_program =
Target_program [Push 10; Push 0; Quo; Push 100; Push 0; Quo; Add]
```

- Does it make any observable difference whether your compiler translates from left to right or from right to left? Yes, it does. If the order of translation is changed, the order in which the corresponding BCI will be changed. This will cause `decode_execute` and `run` to generate different output. Therefore, the order of translation makes observable difference.

8.6 Task 5: Implement an alternative compiler

Presumably your compiler (in Task 4) uses list concatenation (i.e., List.append). List concatenation, however, incurs a linear cost since it prepends its first argument onto its second by copying it.

Program an alternative version of your compiler that does not use list concatenation (and verify that it passes the unit tests).

The goal for this exercise is to bypass the use of List.append in order to avoid the linear cost. Our solution took inspiration from the mini-project about multiplying integers in a tree with an accumulator. In this exercise, instead of "accumulating" the product after repeated multiplication, we are "accumulating" the byte-code instruction after translation.

- Base case: `Literal n` is analogous to the `Leaf` case in the binary tree. Whenever we encounters `Literal n`, we yield a list by adding the BCI `Push n` into the existing BCIs and returns the updated accumulator. This is our base case.
- Induction step: Each of `Plus`,`Minus`,`Quotient`,`Remainder` is analogous of the `Node` case in the binary tree. The only difference (and hence the challenge) is that we now have "four distinct types of nodes" and we need to specify that by adding the corresponding BCI (`Add`, `Sub`,`Quo`,`Rem`) after specifying the inductive hypothesis. Concretely, our implementation is as follows.

```
let compile_v2 (Source_program e) =
  let rec translate e a =
    match e with
    | Literal n ->
       [Push n] @ a
    | Plus (e1, e2) ->
       (let a = [Add] @ a
        in translate e1 (translate e2 a))
    | Minus (e1, e2) ->
       (let a = [Sub] @ a
        in translate e1 (translate e2 a))
    | Quotient (e1, e2) ->
       (let a = [Quo] @ a
        in translate e1 (translate e2 a))
    | Remainder (e1, e2) ->
       (let a = [Rem] @ a
        in translate e1 (translate e2 a))
  in Target_program(translate e []);;
```

We verified that this version of the compiler also passed the unit-test.

```
let () = assert (test_compile "compile_v2" compile_v2);;
```

8.7 Task 6: Unit-test to verify interpreter and the compiler yield the same output

Implement a unit-test function to verify that interpreting source programs yields the same result as compiling them and running the resulting target programs.

Food for thought:

The resource file for the present lecture note contains a generator of random arithmetic expressions, generate_random_arithmetic_expression that, given a non-negative integer n, yields a random arithmetic expression of depth at most n.

Inspired by the food for thought above, we made use of the `generate_random_arithmetic_expression` function to generate random arithmetic expressions to make up the source program and use it to build our unit-test function. While solving this exercises, we were reminded and inspired by exercise 2 in the mini-project about multiplying integers in a binary tree, where we expanded the unit-test function for one particular tree and then used a random binary tree generator to conduct the test several times to improve its robustness. Thus, to define the commutativity test, we first define the commutativty test for one source program:

```
let commutativity_test_one candidate_interpret candidate_compile candidate_run p =
  candidate_interpret p = candidate_run (candidate_compile p);;
```

With this, we expanded the unit test function:

```
let commutativity_test candidate_interpret candidate_compile candidate_run =
   (* commutativity_test : (source_program -> expressible_value) ->
                           (source_program -> target_program) ->
                           (target_program -> expressible_value) ->
                           source_program ->
                           bool *)
  let b0 = let thunk = (fun () ->
               let p = Source_program (generate_random_arithmetic_expression 10)
               in commutativity_test_one
                candidate_interpret candidate_compile candidate_run p)
           in repeat 10 thunk
  in b0;;
```

Subsidiary question:
What is the impact of

- evaluating from left to right vs. from right to left, and of
- translating from left to right vs. from right to left on the commuting diagram? Does it always commute, no matter the order? If not, can you exhibit a counter-example? What is the consequence of this commutation or of this non-commutation?

  We made use of the unit-test function that we have just defined to test for commutativity:

  ```
  let () = assert(commutativity_test interpret compile run);;
  let () = assert(commutativity_test interpret compile_v2 run);;
  ```

  It was observed that both tests were passed. This means that if the evaluation order of the interpreter and the translation order of the compiler are the same, then the two commute. To be more precise, the compiler combined with the virtual machine is commutative with the interpreter. For simplicity, we may use `run(compile)` to refer to the compiler combined with the virtual machine which is defined in the function `run`.

We confirmed our answer by printing out the unit-test function, using the function provided in the accompanying file. All the tests showed that the output result of the interpreter and compiler are the same. Below we have shown one of such examples.

```
Target_program [Push ~-59; Push ~-79; Push 70; Push 82; Quo; Push 22; Push 28; Add; Add;
Push ~-36; Push 68; Push 49; Rem; Rem; Add; Sub; Add]
Source_program (Plus (Literal ~-59, Minus (Literal ~-79, Plus (Plus (Quotient (Literal 70
Remainder (Literal ~-36, Remainder (Literal 68, Literal 49))))))
Interpret result -171
Run Compile result -171
```

This showed that since the interpreter evaluates from left to right and the compiler translates from left to right, `interpret` generates the same output as `run(compile)`. We also verified that it is also the case when the target program has the type `Expressible_msg`:

```
interpret (Source_program (Plus(Quotient(Literal (an_int 10), Literal (an_int 0)),
    Quotient(Literal (an_int 100), Literal (an_int 0)))));;
processing 0...
processing 100...
processing 0...
processing 10...
- : expressible_value = Expressible_msg "quotient of 10 over 0"

run (compile (Source_program (Plus(Quotient(Literal (an_int 10), Literal (an_int 0)),
    Quotient(Literal (an_int 100), Literal (an_int 0))))));;

processing 0...
processing 100...
processing 0...
processing 10...
- : expressible_value = Expressible_msg "quotient of 10 over 0"
```

8.8 Task 7: The fold-right version of the interpreter and the compiler

Define the fold-right function associated with arithmetic_expression and use it to express both interpret (in Task 1) and compile (in Task 4).

First, we defined the fold_right arithmetic taking the the five cases as its parameters, namely, literal_case (the base case for the interpreter), plus_case (induction step for additions), minus_case (induction step for subtractions), quotient_case (induction step for quotients), remainder_case (induction step for remainders). Concretely,

```
let fold_right_arithmetic literal_case plus_case minus_case quotient_case remainder_case
    (Source_program e) =
  let rec evaluate e =
    match e with
    | Literal n ->
       literal_case n
```

```
    | Plus (e1, e2) ->
       plus_case (evaluate e1) (evaluate e2)
    | Minus (e1, e2) ->
       minus_case (evaluate e1) (evaluate e2)
    | Quotient (e1, e2) ->
       quotient_case (evaluate e1) (evaluate e2)
    | Remainder (e1, e2) ->
       remainder_case (evaluate e1) (evaluate e2)
  in evaluate e;;
```

With the `fold_right_arithmetic` defined, we can rewrite the interpreter in terms of the `fold_right_arithmet` that is, with a structure parallel to the five cases so that each can be passed to `fold_right_arithmetic` as a parameter. Notice that the following implementation is the same as the previous version of interpreter, except that now we define each case *explicitly* with let-expressions and at the end of the program pass the five cases to `fold_right_arithmetic`.

```
let interpret_fold_right (Source_program e) =
  let literal_case n = Expressible_int n
  in let plus_case e1 e2 =
       match e1 with
       | Expressible_msg s ->
          Expressible_msg s
       | Expressible_int n1 ->
          (match e2 with
           | Expressible_msg s ->
              Expressible_msg s
           | Expressible_int n2 ->
              Expressible_int (n1+n2))
     in let minus_case e1 e2 =
       match e1 with
       | Expressible_msg s ->
          Expressible_msg s
       | Expressible_int n1 ->
          (match e2 with
           | Expressible_msg s ->
              Expressible_msg s
           | Expressible_int n2 ->
              Expressible_int (n1 - n2)
          )
        in let quotient_case e1 e2 =
             match e1 with
             | Expressible_msg s ->
                Expressible_msg s
             | Expressible_int n1 ->
                (match e2 with
                 | Expressible_msg s ->
```

```
                Expressible_msg s
              | Expressible_int n2 ->
                if n2=0 then
                  (Expressible_msg ("quotient of " ^ string_of_int n1 ^ " over 0") )
                else Expressible_int (n1 / n2)
            )
        in let remainder_case e1 e2 =
            match e1 with
            | Expressible_msg s ->
               Expressible_msg s
             | Expressible_int n1 ->
               (match e2 with
                | Expressible_msg s ->
                   Expressible_msg s
               | Expressible_int n2 ->
                  if n2=0 then
                    (Expressible_msg ("remainder of " ^ string_of_int n1 ^ " over 0") )
                  else Expressible_int (n1  mod n2)
               )
          in fold_right_arithmetic literal_case plus_case minus_case quotient_case
            remainder_case (Source_program e);;
```

Finally, we verified that our fold-right version passed the unit-test function for interpreter.

```
let () = assert (int_test_interpret "interpret_fold_right" interpret_fold_right);;
let () = assert (msg_test_interpret "interpret_fold_right" interpret_fold_right);;
```

Using exactly the same logic behind how we have rewritten the interpreter, it is rather straightforward to write the first version of our compiler in terms of `fold_right_arithmetic` by explicitly defining the cases (namely, `literal_case`, `minus_case`, `quotient_case`, `remainder_case` and pass them to the previously defined `fold_right_arithmetic` function. Our implementation is as follows:

```
let compile_v1_fold_right (Source_program e) =
  (* compile_v2 : source_program -> target_program *)
  let rec translate e =
let literal_case n = [Push n] in
    let plus_case e1 e2 =
        List.append (List.append(translate  e1)(translate  e2))[Add]
    in let minus_case e1 e2 =
        List.append (List.append(translate e1)(translate  e2))[Sub]
    in let quotient_case e1 e2 =
        List.append (List.append(translate  e1)(translate  e2))[Quo]
    in let remainder_case e1 e2 =
        List.append (List.append(translate e1)(translate e2))[Rem]
    in fold_right_arithmetic literal_case plus_case minus_case quotient_case
        remainder_case (Source_program e)
in Target_program (translate e);;
```

Finally, we verified that our fold-right compiler passed the unit-test function for compiler, and that the fold-right compiler combined with virtual machine (i.e., the function `run`) commutes with the fold-right interpreter:

```
let () = assert (test_compile "compile_v1_fold_right" compile_v1_fold_right);;
let () = assert (commutativity_test interpret_fold_right compile_v1_fold_right run);;
```

8.9 Task 8: The fold-right version of the alternative compiler

As a followup to Task 7 (optional), use the fold-right function associated with arithmetic_expression to express the compiler from Task 5 (optional).

Rewriting the accumulator version of the compiler required some extra thoughts and considerations. The solution was, again, inspired by the mini-project about multiplying integers in a binary tree. We now need to remove the accumulator from the parameters, but we also need to define and initialize it. To resolve this issue, we return a function instead.

```
let compile_v2_fold_right (Source_program e) =
  let rec translate e =
    let literal_case n = (fun a -> [Push n] @ a)
    in let plus_case e1 e2 =
         (fun a -> (let a = [Add] @ a
                    in e1 (e2 a)))
       in let minus_case e1 e2 =
            (fun a -> (let a = [Sub] @ a
                       in e1 (e2 a)))
          in let quotient_case e1 e2 =
               (fun a -> (let a = [Quo] @ a
                          in e1 (e2 a)))
             in let remainder_case e1 e2 =
                  (fun a -> (let a = [Rem] @ a
                             in e1 (e2 a)))
                in fold_right_arithmetic literal_case plus_case minus_case quotient_case
                     remainder_case (Source_program e)[]
  in Target_program (translate e);;
```

Finally, we verified that our second version of fold-right compiler passed the unit-test function for compiler, and that the this second version of fold-right compiler also commutes with the fold-right interpreter:

```
let () = assert (test_compile "compile_v2_fold_right" compile_v2_fold_right);;
let () = assert (commutativity_test interpret_fold_right compile_v2_fold_right run);;
```

8.10 Extension: the world seen from right to left

We were led to wonder: can we implement a right-to-left interpreter/compiler?

We implemented the right-to-left version by essentially - at the risk of oversimplifying of course - swapping e1 and e2. To be specific, in the right-to-left interpret function, we always `(evaluate e2)`

and then match `(evaluate e1)` (the opposite of what has been done in the left-to-right version), for example:

```
 | Plus (e1, e2) ->
       (match evaluate e2 with
        | Expressible_int n2 ->
           (match evaluate e1 with
            | Expressible_int n1 ->
               Expressible_int (n1 + n2)
            | Expressible_msg s ->
               Expressible_msg s
           )
        | Expressible_msg s ->
          Expressible_msg s)
```

Similarly, for `compile`, we swap the order of `translate`, for example:

```
(* in the list append version of compile *)
 | Plus (e1, e2) ->
       List.append (List.append (translate e2)  (translate e1))  [Add]

(* in the accumulator version of compile*)
 | Plus (e1, e2) ->
       (let a = [Add] @ a
        in translate e2 (translate e1 a))
```

The same logic applies to the right-to-left version for the folded interpret and compile, and more implementation details can be found in the code files. Note that other functions are not changed, including `fold_right_arithmetic`, `decode_execute`, `run`, since we are only changing the order of two actions, namely **evaluate** and **translate**.

Our implementation for the right-to-left version is symmetrical ("laterally") to the left-to-right version. To test the new right-to-left version, we modified the unit-test function in the following parts.

Making use of the right-to-left implementation, we experimented with the different possible combinations of different versions of `interpret` and `run(compile)`: (Note that in order to run both left-to-right and right-to-left language processors in one single OCaml session, we renamed the functions in a way that is quite self-explanatory.)

```
interpret_ltor (Source_program (Plus(Quotient(Literal 10, Literal 0),
    Quotient(Literal 100, Literal 0))));;
- : expressible_value = Expressible_msg "quotient of 10 over 0"

run (compile_v1_ltor (Source_program (Plus(Quotient(Literal 10, Literal 0),
    Quotient(Literal 100, Literal 0)))));;
- : expressible_value = Expressible_msg "quotient of 10 over 0"

interpret_rtol (Source_program (Plus(Quotient(Literal 10, Literal 0),
    Quotient(Literal 100, Literal 0))));;
```

```
- : expressible_value = Expressible_msg "quotient of 100 over 0"

run (compile_v1_rtol (Source_program (Plus(Quotient(Literal 10, Literal 0),
    Quotient(Literal 100, Literal 0)))));;
- : expressible_value = Expressible_msg "quotient of 100 over 0"
```

From the above results, we can see that when the order of evaluation for the interpreter is the same as the order of translation for the compiler, i.e., either both are left-to-right or both are right-to-left, `interpret` and `run(compile)` are commutative.

Otherwise, we have two cases:

(i) If the final `expressible_value` is of the type `Expressible_msg`, they are not commutative when (unless by coincidence the message from `interpret` and the message from `run(compile)` *appear* the same).

(ii) If the final `expressible_value` is of the type `Expressible_int`, they are still commutative because they are observationally equivalent despite the difference in order.

8.11 Conclusion

One thing that surprised us was the parallel structure in these implementations, which perhaps echoed an earlier metaphor of "the same elephant." More generally, in this project, we observed many pieces of the module coming together: the basics of computation, fold_right recursion, order of evaluation, list constructor, stack, just to name a few. We started the course by discussing the use of interpreter and compiler on an abstract level. Now, we were finally able to implement them and compare them in more technical details.

9 Putting it all together

Taking the last project (about) as a departure point, we have attempted to summarize the structure of the interpreter, the compiler and the virtual machine, using the I-diagram and the T-diagram learned back in Week 01. In the figure, we have organized the information on three levels: high-level (as abstract concepts learned at the beginning of the course), low-level (with implementation details), and links to other mini-projects in this term project.

As we have highlighted in the diagram, the source program expression essentially has the structure of a binary tree (as shown in the example on the top right in Figure 11.) And the process of evaluating (in the case of the interpreter), translating (in the case of the compiler), and fetching and decoding BCIs (in the case of the virtual machine) are essentially the process of traversing the tree. This then links to the other mini-projects: we traverse the same binary tree structure through which we transform one data structure to another.

Now that we have realized this process of abstracting the specific data structures into a generic pattern/structure of binary trees, a natural follow-up investigation is then to find out the common properties shared by these types of data structures that make them fit for the binary tree representation. We have included some observations:

(i) They all have some direction, for example, arithmetic expressions will have an order.

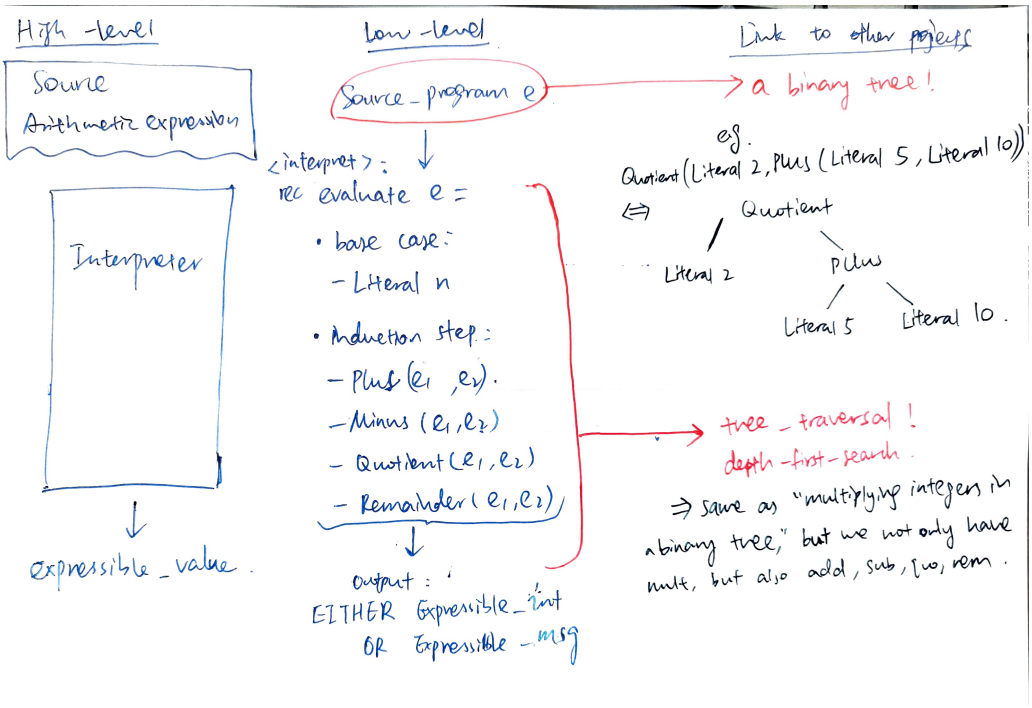

Figure 11: Summary of the Interpreter

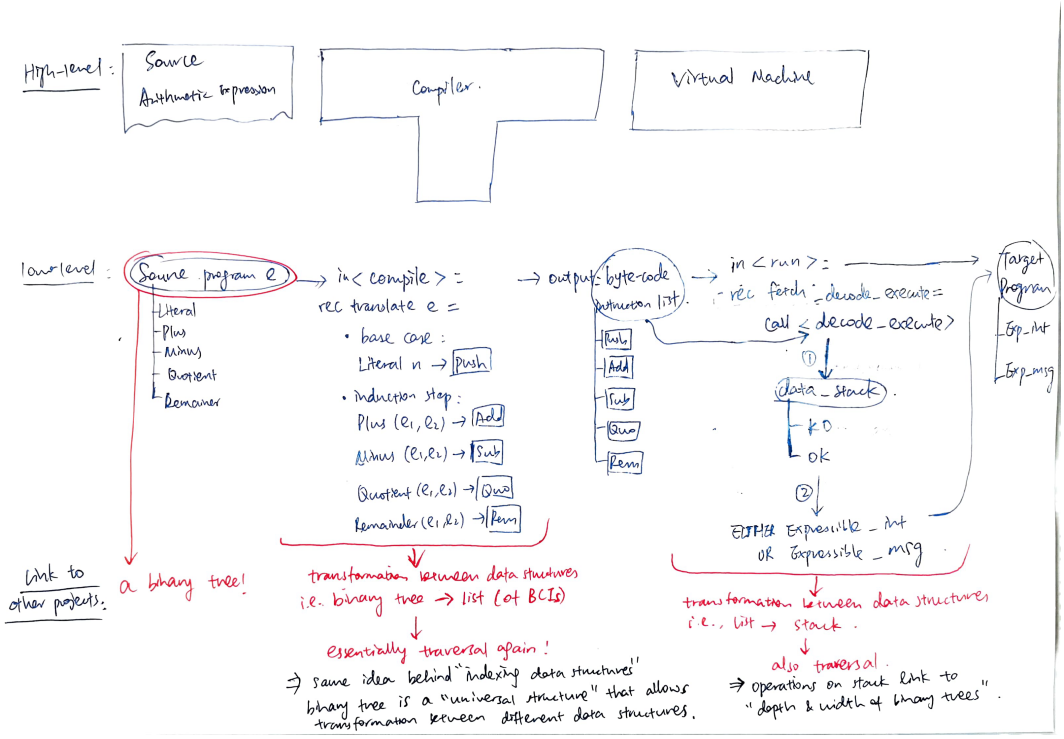

Figure 12: Summary of the Compiler (with Virtual Machine)

(ii) They are finite data structures. A counter example is seen in the mini-project about indexing data structures - stream. It is not possible to index a stream from right to left because in such a direction it will become infinite.

(iii) Two and only two things are involved at a time.

Another closely related structure that also shared these properties is the regular expression which we have learned in week 02. And we can now appreciate the fact that regular expressions can be represented in an abstract-syntax tree. In fact, arithmetic expressions is a subset of the regular expressions, and hence what we have implemented can be considered a partially functional parser/unparser. Another conjecture is that since proof tree and abstract syntax tree are equivalent methods to parse, and since the Boolean operator is also a binary operator, would it be possible to implement the proof tree with a similar approach.

Through this project we have used the basic building block, recursion, to work with different data structures and appreciate how it makes the process of abstraction much neater - indeed a graceful leap from recursion to data structure.