YSC1212: Introduction to Computer Science

Midterm: A Curious Encounter with Computation

Liu Zhang, A0190879J

Created [1 March 2020]

Other team members:
Jincong Chu, A0156290E
Zhi Yi Yeo, A0156253H
Chengpei Liu, A0136151R

Contents

1 Introduction

What is computation? This question seems straight-forward enough to be dismissed. However, it is not easy to fully unpack and appreciate the hidden intricacies of computation. Through the exercises in this project, we have realized the significance of understanding and being mindful of these intricacies and we have developed some sense about the underlying mechanism of computation. This is as much a re-learning experience as it is a learning experience, for it enables us to see computation in a new light.

This midterm project is a culmination of what we have learned so far in YSC1212 and consists of four main mini-projects and bonus questions. In these projects, we have explored many aspects of computation, including but not limited to computation on recursive structures, order of computation, simplifying programs (which is essentially the process of computation), equivalence (and hence difference) in computation, verifying/testing computation. We have also put the knowledge into application by solving two interesting case studies, the Fibonacci numbers and the Palindromes.

2 A Miscellany of Recursive Programs

2.1 Overview

In this mini-project, we explore the nature of computation from the point of view of recursion. Through the exercises in this mini-project, we realized the power of recursion in implementing computation and also the profound connections between the different recursive structures, in particular expressing other recursive functions in terms of the basic recursive structures, **fold_right_nat** and **parafold_right_nat**, as well as expressing these two structures in terms of each other.

2.2 Exercise 3: Expressing the factorial function using fold_right_nat

Define the factorial function using fold_right_nat.

In this exercise, we are asked to define the factorial function using **fold_right_nat**. We first include a unit test function for the factorial function.

```
let test_fact candidate =
  let b0 = (candidate 0 = 1)
  and b1 = (candidate 1 = 1)
  and b2 = (candidate 2 = 2)
  and b3 = (candidate 3 = 6)
  and b4 = (let n = 1 + Random.int 20 in candidate (succ n) = candidate n * (succ n))
  in b0 && b1 && b2 && b3 && b4;;
```

In our group code, we have come up with a working solution for the problem (and elegant enough too). The idea behind this solution is that at each step of the recursion, we "remember" a pair of integers, (ind, factor). The first entry stores the index indicating whose factorial we are calculating in the current step of recursion, i.e., n. The second entry stores the factorial so far (up to the current index), i.e., $n!$. The inductive specification is as follows,

- Base case, i.e., **zero_case**: (0,1) because when n = 0, n! = 1.
- Induction step: given a pair of integers (ind,factor), the **succ_case** (the next step) should return (ind + 1,factor*(ind + 1));

With the inductive specification, we can implement the function in terms of fold_right_nat:

```
let fact_v0 n_given =
  let () = assert (n_given >= 0) in
  let succ_case ih =
    let (ind, factor) = ih
      in (ind + 1, factor * (ind + 1))
  in let (index, factorial) = fold_right_nat (0, 1) succ_case n_given
  in factorial;;

let () = assert(test_fact fact_v0);;
```

We can further simplify the above function by unfolding the declaration of the inductive hypothesis, ih:

```
let fact_v0_1 n_given =
  let () = assert (n_given >= 0) in
  let succ_case (ind, factor) = (ind + 1, factor * (ind + 1))
  in let (index, factorial) = fold_right_nat (0, 1) succ_case n_given
  in factorial;;

let () = assert(test_fact fact_v0_1);;
```

Similarly, we can further simplify the function by unfolding the declaration of succ_case:

```
let fact_v0_2 n_given =
  let () = assert (n_given >= 0) in
  let (index, factorial) = fold_right_nat (0, 1) (fun (ind, factor) -> (ind + 1, factor * (ind
  in factorial;;

let () = assert(test_fact fact_v0_2);;
```

We verify that all the three versions pass our unit tests. This is an effective solution but it seems that the implementation is too similar to using a parafold_fold_nat function, which is essentially a fold_right_nat function with an additional entry storing the index in the induction step. Thus I offer another solution using a different underlying motivation: we "remember" the two preceding factorials at each step and make use of this to generate the next pair. In other words, we can think of the computation of factorial as recursively define a sequence of factorials, $a_n = (n!)$ where $n = 0, 1, 2, \ldots$. This "sequential view of factorial numbers" is inspired by the Fibonacci function in the mini-project **"The case of the Fibonacci numbers"**(4.2).

```
let fact_v1 n_given =
  let () = assert (n_given >= 0) in
  let (fact_n, fact_succ_n) =
    fold_right_nat (1, 1) (fun ih -> let(fact_n', fact_n) = ih in
        (fact_n, fact_n * (fact_n/fact_n' + 1))) n_given
  in fact_n;;

let () = assert(test_fact fact_v1);;
```

Now, we briefly compare these two solutions.

- In fact_v0, we "remember" the current term and its corresponding factorial; in fact_v1, at every step we "remember" the two nearest factorials.
- Both functions have same computational complexity, to be specific, both run with constant time, O(n).

Based on Prof Danvy's input, we are inspired to further characterize the first solution, fact_v0. The graph of the function is the following set of pairs, (x, y) — y = f(x), i.e., if the function has type nat -¿ nat, it is (0, f(0)), (1, f(1)), (2, f(2)), ... We can then think of this recursion as a mathematical function, in which the index parameter is the independent variable x and the factorial parameter is the dependent variable y. This points out why we return fact_n in the end: that is the value of y when x = n. We can also think of the induction step in terms of the gradient of the graph (as x increases by one, it causes y to increase by a factor of (x+1)). This could be a topic on its own to pursue further.

2.3 Exercise 4: The sumtorial function, revisited

As a continuation of Exercise 14 in Week 03, implement two sumtorial functions: one should make as many recursive calls as the natural number it is applied to, and the other one should not even be recursive. Verify that both functions pass your unit test, and express the first using parafold_right_nat.

N.B.: You will need to start from scratch, i.e., with the inductive specification of the sumtorial function. (Hint: mimic the specification of the factorial function.)

We first include the unit test covering both the simple cases and random cases.

```
let test_sumtorial candidate =
  let b0 = (candidate 0 = 0)
  and b1 = (candidate 1 = 1)
  and b2 = (candidate 2 = 3)
  and b3 = (let n = Random.int 20
                in candidate (succ n) = (succ n) + candidate n)
  in b0 && b1 && b2 && b3;;
```

This unit test is limited because we can write a fake sumtorial function that passes the unit test but is not the sumtorial function we want because this function would only work for $0 < n < 20$ and not otherwise.

```
let fake_sumtorial n =
  if n > 0 && n <= 20 then (0 + n) * (n + 1) / 2
  else 0;;

let () = assert (test_sumtorial fake_sumtorial);;
```

We give the inductive specification:

- Base case: summing up 0 through 0 yields 0.
- Induction step: given a number n' such that summing up 0 through n' yields ih, summing up 0 through succ n' should yield (succ n') + ih.

Based on this inductive specification, we can implement the first version of sumtorial and verify that it passes the unit test previously defined.

```
let sumtorial_v1 n_given =
  let () = assert (n_given >= 0) in
  let rec visit n =
    if n = 0
    then 0
    else let n' = pred n
         in let ih = visit n'
            in (succ n') + ih
  in visit n_given;;

let () = assert (test_sumtorial sumtorial_v1);;
```

However, we notice that recursive function is not necessary for computing the sum from 0 to n. An easier way is to use the known mathematical formula,

$$\sum_{k=1}^{n} k = \frac{(1+n)}{2},$$

which is implemented as follows. This version has passed the unit test.

```
let sumtorial_v2 n_given =
  (1 + n_given) * n_given / 2;;

  let () = assert (test_sumtorial sumtorial_v2);;
```

The last version of sumtorial is to use the pre-defined parafold_right_nat. This version has the same induction specification as the first version. The only difference is that instead of writing the recursive structure in sumtorial function, we simply pass the base case, induction hypothesis and the given n to the pre-defined parafold_right_nat function. After that, we can count on the recursive structure already defined in parafold_right_nat to do the computation. Eventually, parafold_right_nat will return the sum. We have verified that this version also passes the unit test.

```
let sumtorial_v3 n_given =
  let () = assert (n_given >= 0) in
  parafold_right_nat 0 (fun n' ih -> succ(n') + ih) n_given;;

  let () = assert (test_sumtorial sumtorial_v3);;
```

Using similar line of reasoning, we can also define sumtorial using structural recursion fold_right.

```
let sumtorial_v4 n_given =
  let () = assert (n_given >= 0) in
  let (index, sumtorial) = fold_right_nat (1,0) (fun (ind, sum) -> (ind + 1, sum + ind)) n_giv
  in sumtorial;;

let () = assert (test_sumtorial sumtorial_v4);;
```

2.4 Exercise 5: Specifying and implementing the Sigma (sum) notation

(a) *Compose a unit-test function for sum, based on its inductive specification.*

First and foremost, we give the inductive specification for the unit test. Based on the definition of the summation operator, i.e., sum f n = f(0) + f(1) + f(2) + ... + f(n-1) + f(n), we can give the mathematical definition of the base case and inductive hypothesis:

- Base case: f(0) = f(0).
- Induction step: given f(n-1), f(n) = f(n-1) + f(n).

Now we include the above in the unit test for the sum function below.

```
let test_sum candidate f =
  let b0 = (candidate f 0 = f 0)
  and b1 = (candidate f 1 = candidate f 0 + f 1)
  and b2 = (candidate f 2 = candidate f 1 + f 2)
  and b3 = (let n = 1 + Random.int 20
                        in (candidate f n = candidate f (n-1) + f n))
  in b0 && b1 && b2 && b3;;
```

In the unit test function, **b0 = (candidate f 0 = f 0)** tests for the base case; and **b3** tests for the induction step. The unit test is limited since it does not cover all natural numbers n. However, it suffices our purpose for now by allowing us to have a preliminary verification on the correctness of the programs written.

(b) *Implement this specification as a structurally recursive function expecting a function from int to int and a non-negative integer.*

We give the inductive specification for the implementation:

- Base case: summing up 0 through 0 yields f(0).
- Induction step: given a number n' such that summing up f(0) through f(n') yields ih, the result for succ n' should yield f(succ n') + ih.

Based on this inductive specification, we can implement the summation operator.

```
let sum_v1 f n_given =
  let () = assert(n_given >= 0) in
  let rec visit f n =
    if n = 0
    then f 0
    else let n' = pred n
             in let ih = visit f n'
            in ih + f (succ n')
  in visit f n_given;;
```

(c) *Verify that this function passes your unit test.*

We write a silent test right after defining the unit test function and the sum function.

```
let () = assert(test_sum sum_v1 sumtorial_v1);;
```

If the program passes the test, it will be silent (does not print any message); and otherwise, it will generate an error message. Our sum function passes the silent test.

(d) *Express your implementation using either fold_right_nat or parafold_right_nat, your choice. Justify this choice, and verify that your new implementation passes your unit test.*

We will implement a second version of sum function using parafold_right_nat. This version has the same induction specification as the first version. The only difference is that instead of writing the recursive structure in sum function, we simply pass the base case, induction hypothesis and the given n to the pre-defined parafold_right_nat function. After that, we can count on the

recursive structure already defined in parafold_right_nat to do the computation. We also verify that this version also passes the unit test.

```
let sum_v2 f n_given =
  let () = assert(n_given >= 0) in
  parafold_right_nat (f 0) (fun  n' ih -> (f (succ n')) + ih) n_given;;

let () = assert(test_sum sum_v2 sumtorial_v1);;
```

(e) *Revisit the sumtorial function and express it using sum.*

The sum function is essentially a higher-order abstraction of the sumtorial function. In other words, by having the term f(x) instead the term x in the summation, our structure now has greater generalizability, effectively increasing its capability for computation. To put it more concretely, the sumtorial function is only capable of summing up the sequence of natural numbers themselves, but the sum function is now capable of summing up any specific mapping of the natural number, which we can define freely based on our specific needs.

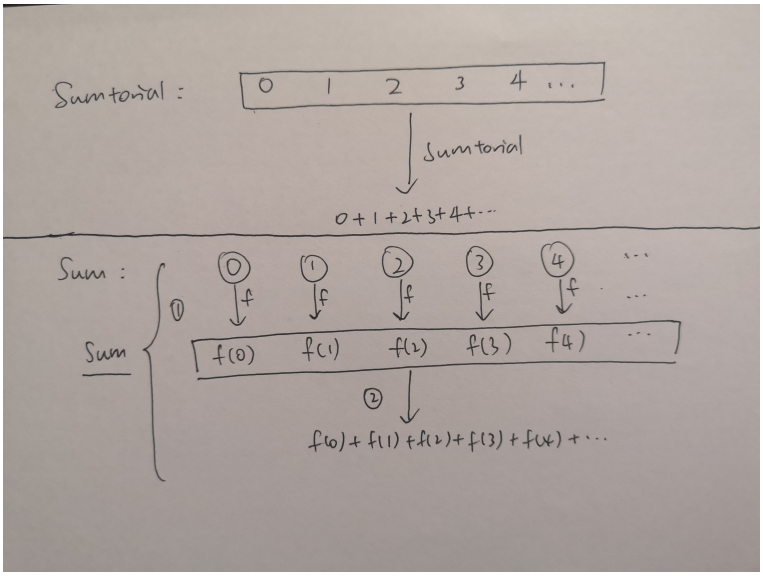

Figure 1: Visualizing the similarities and differences between sum and sumtorial functions.

The key idea that this figure aims to illustrate is that the sum function can be thought of as a two-step process: 1) apply some function f; 2) apply sumtorial function. With this theoretical understanding, it is easy to see that if we apply the identity function to sumtorial function, the final effect of the two-step process is equivalent to the second step alone, which is to apply sumtorial function. Hence, we just need to pass the identity function (fun x -¿ x) to the first argument of the sum function, concretely:

```
let sumtorial_v4 n_given =
  let () = assert (n_given >= 0) in
  sum_v2 (fun x -> x) n_given;;
```

```
let () = assert(test_sumtorial sumtorial_v4);;
```

(f) *Revisit Exercise 15 in Week 03 and implement three OCaml functions that given a non-negative integer n, compute the sum of the n+1 first odd numbers. The first should be structurally recursive. The second should use either fold_right_nat or parafold_right_nat, your choice. And the third should operate in constant time.*

The first version of oddsum is structurally recursive. We give the inductive specification for the implementation:

- Base case: when n = 0, computing the sum of the (n+1=1) first odd number yields 1.
- Induction step: given a number n such that computing the sum of the (n+1) first odd number yields ih, the result for succ n should yield ih + n*2+1. Note here that the (n+1)-th odd number is n*2+1.

Based on this inductive specification, we can implement the oddsum function and its unit test function.

```
let test_oddsum candidate =
  let b0 = (candidate 0 = 1)
  and b1 = (candidate 1 = 4)
  and b2 = (candidate 2 = 9)
  and b3 = (candidate 3 = 16)
in b0 && b1 && b2 && b3;;

let oddsum_v1 n_given =
  let () = assert (n_given >=0) in
  let rec visit n =
    if n = 0 then 1
    else let ih = visit (pred n)
             in ih + n*2+1
  in visit n_given;;

let () = assert (test_oddsum oddsum_v1);;
```

We will now implement a second version of the oddsum function using parafold_right_nat. This version has the same induction specification as the first version. The only difference is that instead of writing the recursive structure in sum function, we simply pass the base case, induction hypothesis and the given n to the pre-defined parafold_right_nat function. We also verify that this version also passes the unit test.

```
let oddsum_v2 n_given =
  parafold_right_nat 1 (fun n' ih -> 2* (succ n') +1 + ih) n_given;;

let () = assert (test_oddsum oddsum_v2);;
```

Similar to what we did for sumtorial, we can also express the oddsum function in terms of parafold_right_nat, and this version passes the unit tests:

```
let oddsum_v4 n_given =
  let () = assert (n_given >= 0) in
  let (index, oddsum) = fold_right_nat (0,1) (fun (ind, oddsum) -> (ind + 1, oddsum + (ind
  in oddsum;;

let () = assert (test_oddsum oddsum_v4);;
```

The last version is based on a known mathematical formula, the sum of first (n+1) odd numbers is

$$\sum_{k=0}^{n} 2k + 1 = (n+1)^2.$$

```
let oddsum_v3 n_given =
  (n_given + 1)^2;;

let () = assert (test_oddsum oddsum_v3);;
```

(g) Observing that for any non-negative integer n,

$$1 + F_0 + F_1 + F_2 + ... + F_n + = F_{n+2}.$$

(where F_0, F_1, F_2, etc. are Fibonacci numbers) verify that this identity holds and beef up your unit-test function for sum with it.

First of all, we can verify the identity by making use of the existing unit test function and the most efficient version of the fib function, fib_v5, which is given in the other mini-project, ***The case of the Fibonacci numbers***. Now, we first write a "naive" unit test with a few simple cases and the random cases.

```
let test_fibrule candidate =
  let b0 = (1 + sum_v1 candidate 0 = candidate 2)
  and b1 = (1 + sum_v1 candidate 1 = candidate 3)
  and b2 = (1 + sum_v1 candidate 2 = candidate 4)
  and b3 = (1 + sum_v1 candidate 3 = candidate 5)
  and b4 = (let n = Random.int 20
                          in 1 + sum_v1 candidate n = candidate (n+2))
  in b0 && b1 && b2 && b3 && b4;;

let () = assert(test_fibrule fib_v5);;
```

We can also verify this identity mathematically by a proof by induction. Base case is easily verified because when $n = 0, 1 + F_0 = 1 = F_2$ is true. For the inductive hypothesis, suppose that the identity is true for n. This implies that $1 + F_0 + F_1 + F_2 + ... + F_n + = F_{n+2}$. Then,

$$1 + F_0 + F_1 + F_2 + ... + F_n + Fn + 1 = F_{n+2} + F_{n+1} = F_{n+3},$$

which means the identity is also true for $n + 1$. Hence, we prove that the identity holds.

Note that verification by mathematical proving is stronger than the testing because testing is incomplete in the sense that it is not generalized to all the natural numbers (unlike proving). We are once again reminded of Dijkstra's quote:

> "Program testing can be used to show the presence of bugs, but never to show their absence!"

Despite that, let us not despair. At least we can now go back and improve the unit test for sum function by including our new discovery on the identity:

```
let test_sum_v1 candidate f =
  let b0 = (candidate f 0 = f 0)
  and b1 = (candidate f 1 = candidate f 0 + f 1)
  and b2 = (candidate f 2 = candidate f 1 + f 2)
  and b3 = (let n = 1 + Random.int 20
                        in 1 + candidate fib_v5 n = fib_v5 (n+2))
  in b0 && b1 && b2 && b3;;

let () = assert(test_sum_v1 sum_v1 fib_v5);;
```

2.5 Exercise 6: fold_right_nat and parafold_right_nat

In this exercise, we explore the connections between the two basic recursive function, fold_right_nat and parafold_right_nat. The aim is to define one in terms of the other.

(a) *Define fold_right_nat in terms of parafold_right_nat.*

Before we implement anything, let us write a unit test function for fold_right_nat.

```
let test_fold candidate =
  let b0 = (candidate 3 (fun ih -> succ ih) 5 = 8)
  and b1 = (candidate 10 (fun ih -> pred ih) 5 = 5)
  and b2 = (candidate 0 (fun ih -> 9 + ih) 5 = 45)
  and b3 = (candidate 5 (fun ih -> ih) 6 =5)
  in b0 && b1 && b2 && b3;;

let ()= assert(test_fold fold_right_nat);;
```

This is a relatively easier task than the second one. fold_right_nat is essentially "contained," or a part of, parafold_right_nat. Therefore the inductive specification is simply:

- Base case: zero_case
- Induction step: (fun x ih → succ_case ih). Given the index x and the inductive hypothesis ih, we discard the index and preserve only the succ_case ih, which gives the induction step of a fold_right_nat.

With this inductive specification we implement fold_right_nat in terms of parafold_right_nat. And it successfully passed the silent test.

```
let fold_right_nat_para zero_case succ_case n =
  let ()= assert(n >=0 ) in
 parafold_right_nat zero_case (fun x ih -> succ_case ih) n;;

let ()= assert(test_fold fold_right_nat_para);;
```

(b) *Define parafold_right_nat in terms of fold_right_nat.*

Before we implement anything, let us write a unit test function for parafold_right_nat.

```
let test_para candidate =
  let b0 = (candidate 1 (fun n' ih -> 2* (succ n') +1 + ih) 5 = 36)
  and b1 = (candidate 1 (fun n' ih -> 2* (succ n') +1 + ih) 10 = 121)
  and b2 = (candidate 1 (fun n' ih -> succ n' * ih) 10 = 3628800)
  and b3 = (candidate 1 (fun n' ih -> succ n' * ih) 0 = 1)
  and b4 = (candidate 0 (fun n' ih -> succ n' + ih) 0 = 0)
  and b5 = (candidate 0 (fun n' ih -> succ n' + ih) 5 = 15)
  in b0 && b1 && b2 && b3 && b4 && b5;;

let ()= assert(test_para parafold_right_nat);;
```

This is a bit more complicated but not undoable and it follows the same intuition but the other way around. parafold_right_nat is essentially fold_right_nat with one extra argument, the index. This motivates us to make use of the pair structure. Therefore the inductive specification is simply:

- Base case: (0,zero_case), i.e., we append the index 0 to form the base case.
- Induction step: (fun (n', ih) → (succ n', succ_case n' ih)), i.e., given a pair (n', ih) where n' is the index and ih is the inductive hypothesis, the induction step should be defined such that it returns us (succ n', succ_case n' ih), where succ n' is the next index and succ_case n' ih is the next corresponding succ_case.
- Also note that parafold_right_nat returns only the second entry i.e., the succ_case, and thus we use the pair_2 function to take out and return the second entry of the pair.

With this inductive specification we implement parafold_right_nat in terms of fold_right_nat. And it successfully passed the silent test.

```
let pair_2 (a,b) = b;;

let parafold_right_nat_fold zero_case succ_case n =
  pair_2 (fold_right_nat (0,zero_case) (fun (n', ih) -> (succ n', succ_case n' ih)) n);;

let () = assert (test_para parafold_right_nat_fold);;
```

To sum up, we can think of parafold_right_nat as an "indexed version" of fold_right_nat. The underlying thought process is the essentially same as the one behind expressing the factorial function in Exercise 3 (or any other recursive function) in terms of parafold_right_nat. This intuition motivates the bonus questions 7 and 8, since we can also think of String.mapi as an "indexed version" of String.map.

2.6 Exercise 9: Mutual recursion on the ternary.

Generalizing from 2 to 3, a number divisible by 3 is said to be ternary, the successor of a ternary number is said to be post-ternary, and the predecessor of a positive ternary number is said to be pre-ternary. Write predicates about whether a given non-negative integer is ternary, pre-ternary, or post-ternary, in the same fashion as for the predicates about a given non-negative integer being even or odd.

In this exercise, we use a mutual recursion to determine if a given non-negative integer is ternary, pre-ternary, or post-ternary.

Observe that if a number is ternary, its predecessor is pre-ternary, and its successor is post-ternary. Based on this observation, we first come up with the inductive specification by defining the three together:

- Base case: 0 is ternary and 0 is not pre-ternary and 0 is not post-ternary.
- Induction step: given a number n' such that its being ternary is the Boolean ih_ter, that its being pre-ternary is the Boolean ih_pre, and that its being post-ternary is the Boolean ih_post. The predicate of succ n' being ternary is n' being pre-ternary, i.e., ih_pre; the predicate of succ n' being pre-ternary is n' being post-ternary, i.e., ih_post; and the predicate of succ n' being post-ternary is n' being ternary, i.e., ih_ter.

Since the three predicates are defined together, we can test them together:

```
let test_ternary ter pre post =
      (* an instance of the base cases: *)
  let b0 = (ter 0 = true)
  and b1 = (pre 0 = false)
  and b2 = (post 0 = false)
      (* an instance of the induction steps: *)
  and b3 = (let n' = Random.int 1000
            in ter(3*n') = pre(3*n'-1))
  and b4 = (let n' = Random.int 1000
            in ter(3*n') = post(3*n'+1))
  (* etc. *)
  in b0 && b1 && b2 && b3 && b4;;
```

Similarly, since the two predicates are defined together, we can implement them together, i.e., in a mutually recursive way:

```
let rec ter n =
  if n = 0
  then true
  else let n' = n - 1
       in pre n'
and pre n =
  if n = 0
  then false
```

```
  else let n' = n - 1
       in post n'
and post n =
  if n = 0
  then false
  else let n' = n - 1
       in ter n';;

let () = assert (test_ternary ter pre post);;
```

To condense this process, I have come up with a visualization to show the inner working of this mutual recursion. It is also more clear with the diagram that this will be a tail recursion: whether we end up in the solid red circle or the empty red circle will give us the final answer as to whether the given number is indeed a ternary number.

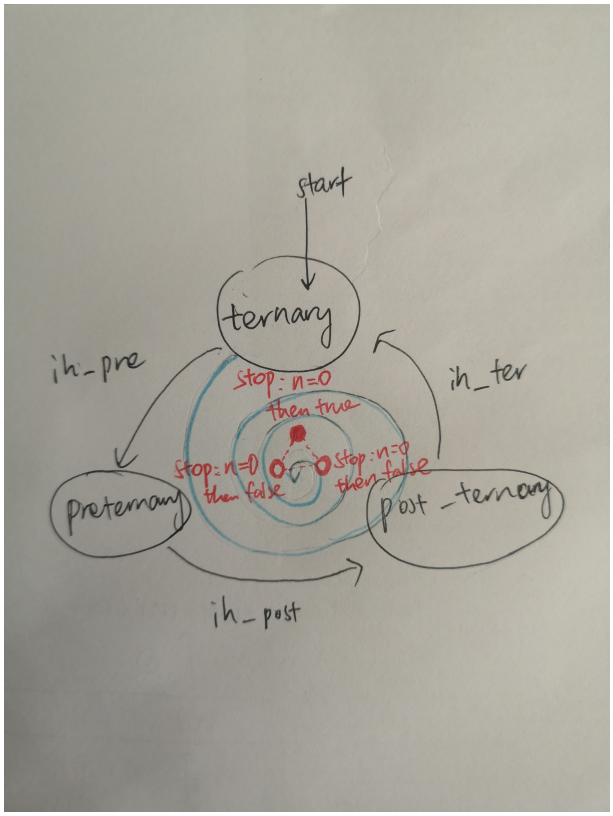

Figure 2: Visualizing the mutual recursion for ternary function.

Further reflecting on this visualization, I am reminded of the unmistakably similar "rotating structures," particularly modulus and the rotate function (in the "(almost) forgotten" exercises 7 and 8 in Week 6). My conjecture is that all structures having this periodic/rotating/going-back behavior can be expressed in terms of mutual recursion. This behavior is as opposed to (but also related to) a growing behavior involving the index, for example the Fibonacci numbers, which is usually more easily solved by structures like fold_right_nat and parafold_right_nat. The latter we will further explore in 4.

Given the comments from Prof Danvy, we can further explore how to define a singly recursive version of each predicate.

We will start with implementing the predicate for ternary number.

First, we include a unit test function:

```
let test_ternary_v2 ter =
      (* an instance of the base cases: *)
  let b0 = (ter 0 = true)
  and b1 = (ter 1 = false)
  and b2 = (ter 2 = false)
      (* an instance of the induction steps: *)
  and b3 = (let n' = Random.int 1000
            in ter(3*n') = true)
  and b4 = (let n' = Random.int 1000
            in ter(3*n' + 1) = false)
  and b5 = (let n' = Random.int 1000
             in ter(3*n' + 2) = false)
  (* etc. *)
  in b0 && b1 && b2 && b3 && b4 && b5;;
```

We realize that for n = 0, 1, ..., n, the respective Boolean value (for whether the number is ternary or not) is TRUE, FALSE, FALSE, TRUE, FALSE, FALSE, TRUE, TRUE... Given this pattern, we can define the inductive specification the same way as the Fibonacci numbers (Fibonacci is everywhere indeed!), i.e., the true value of each term depends on the truth values of the previous terms. When one is TRUE and the other FALSE, the third is FALSE; when both are FALSE, the third is TRUE. Thus, we have the inductive specification:

- Base case: when n = 0, we have a pair of Boolean values (TRUE, FALSE).
- Induction step: given a pair of Booleans (ter_n', ter_n), the next step would be another Boolean pair (ter_n, not(ter_n' —— ter_n)).

With this, we can define the ternary function:

```
let ter_v1 n_given =
  let () = assert (n_given >= 0) in
  let rec visit n =
    if n = 0
    then (true, false)
    else let n' = n - 1
         in let (ter_n', ter_n) = visit n'
            in (ter_n,  not(ter_n' || ter_n))
  in let (ter_n, ter_succ_n) = visit n_given
     in ter_n;;

let () = assert(test_ternary_v2 ter_v1);;
```

Similar to all previous questions, we can also define this on fold_right_nat:

```
let ter_v2 n_given =
      (* base cases: *)
  let () = assert (n_given >= 0) in
  let (ter_n, ter_succ_n) = fold_right_nat (true, false) (fun (ter_n', ter_n) -> (ter_n, not(t
in ter_n;;

let () = assert(test_ternary_v2 ter_v2);;
```

Both pass the unit tests. The predicates for the pre-ternary and post-ternary are exactly the same as the ternary function except that the base case for pre-ternary is (FALSE, TRUE) and for post-ternary is (FALSE, FALSE).

I also generalized this to m-nary, i.e., to determine whether a number is divisible by m. I now include the unit test and the m-nary predicate:

```
let test_mnary m mnary =
      (* an instance of the base cases: *)
  let b0 = (mnary m 0 = true)
  and b1 = (mnary m 1 = false)
  and b2 = (mnary m 2 = false)
      (* an instance of the induction steps: *)
  and b3 = (let n' = Random.int 1000
            in mnary m (m * n') = true)
  and b4 = (let n' = Random.int 1000
            in let r = Random.int m
               in mnary m (m * n' + r) = false)
  (* etc. *)
  in b0 && b1 && b2 && b3 && b4;;

let mnary m n_given =
  let () = assert (n_given >= 0) in
  let rec visit (i, n) =
    if n = 0
    then true
    else begin
      if i = 1 then not(visit (m, pred n))
      else if i = m then not(visit (pred i, pred n))
      else visit (pred i, pred n)
      end
  in visit (m, n_given);;
```

(Note: there seems to be some difficulties in running the unit test, but a few examples verify that the predicate is in fact correct.)

3 The Underlying Determinism of OCaml

3.1 Overview

In this mini-project, we look at computation from a more language-specific point of view. In particular, for two expressions e1 and e2, we are concerned about whether they are observationally equivalent, i.e., whether expressions produce an output given the same input. In the case that they are equivalent, we can think of them as pure expressions that do not produce side effect; otherwise, they are impure expressions that produce side effects. To conclude whether the given expressions are indeed equivalent, we need to figure out the order of computation by scrutinizing the intricacies and eccentricities of OCaml.

3.2 Question 2: Evaluating order for a function and an argument

In a function application, are the two sub-expressions (namely the one in position of a function, on the left, and the one that is the actual parameter of the function, i.e., the argument, on the right) evaluated from left to right or from right to left?

We use the functions provided by Prof Danvy to conduct a simple test.

```
a_function an_int 5;;

processing a function...
processing 5...
- : int = 5
```

In the case of the identity function, the function is processed first, i.e., the expressions are evaluated from left to right.

However, in the case of other functions, we see that the argument is processed first, i.e., the expressions are evaluated from right to left.

```
let simple_function x = 5;;

a_function simple_function (a_char 'c');;

processing a function...
processing 7...
- : int = 7
```

Subsidiary question: is this order compatible with the answer to Question 1? Why?

This is compatible with question 1, because in question 1 we see that operands (arguments) are evaluated before the operation (function) takes place.

```
(* Test for the identity function *)
let identity x = x;;

a_function identity an_int 7;;

processing a function...
```

```
processing 7...
- : int = 7

a_function identity an_int (succ (an_int 6));;

processing 6...
processing a function...
processing 7...
- : int = 7
```

3.3 Question 3: Evaluating order for definienses evaluated in a let expression

In a (local or global) let-expression declaring several bindings at once, i.e., with and, are the definienses evaluated from left to right or from right to left?

In both cases, the definienses are evaluated from left to right.

```
(* Declare bindings for global let-expression *)
let n = an_int 1 and m = an_int 2;;

processing 1...
processing 2...
val n : int = 1
val m : int = 2

(* Declare bindings for local let-expression *)
let n = an_int 1 and m = an_int 2 in n + m;;

processing 1...
processing 2...
- : int = 3
```

Food for thought

- *When a tuple is constructed, in which order are its components evaluated?*

  We consult OCaml because we can. A quick consultation reveals that for a tuple the components are evaluated from right to left.

  ```
  (an_int 6, an_int 5);;

  processing 5...
  processing 6...
  - : int * int = (6, 5)

  (an_int 5, an_int 6);;

  processing 6...
  processing 5...
  - : int * int = (5, 6)
  ```

- *Given two expressions of type int, e1 and e2, how do the two following expressions compare: let x1 = e1 and x2 = e2 in x1 + x2;;*

  let (x1, x2) = (e1, e2) in x1 + x2;;

  We verify this by doing a few experiments in OCaml:

```
let x1 = an_int 5 and x2 = an_int 6 in x1 + x2;;

processing 5...
processing 6...
- : int = 11

let (x1, x2) = (an_int 5, an_int 6) in x1 + x2;;

processing 6...
processing 5...
- : int = 11
```

  The key difference between the two expressions is the order in which they are processed. For the first expression, since the let-expression evaluates its definienses from left to right, $x1 = e1$ (first definiens) is evaluated first before $x2 = e2$ (second definiens). However, in the second expression, there is only one definiens in the let-expression. This definiens is thus evaluated, and since it is a tuple, it will be evaluated from right to left. This means that $e2$ is evaluated before $e1$, making it different from the first expression.

3.4 Question 4: Evaluating order for conjuncts evaluated in a Boolean conjunction

In a Boolean conjunction (&& in infix notation), are the conjuncts (i.e., the arguments of &&, of type bool) evaluated from left to right or from right to left? To answer this question completely, make sure to consider the 4 possible cases. Why does this design make sense? Here are our experiments in OCaml:

```
(* the && conjunction *)

a_bool true && a_bool false;;

processing true...
processing false...


a_bool true && a_bool true;;

processing true...
processing true...


a_bool false && a_bool true;;
```

```
processing false...

a_bool false && a_bool false;;
processing false...
```

From testing the 4 cases, we see that the conjuncts are evaluated from left to right. And when the first argument is false, the second argument will never be evaluated. We will now explain both in details.

In the case of *true && false* and *true && true*, both conjuncts are evaluated from left to right. This returns the Boolean $false$ for the former case and the Boolean *true* for the latter case.

However, in the case of *false && true* and *false && false*, while the conjuncts are also evaluated from left to right, we see that only the left conjunct (false) is evaluated. A Boolean $false$ is returned right after the first conjunct ($false$) is evaluated. This is in fact a very efficient design on OCaml's part, as we know that for any logical expression with the Boolean operator &&, the statement should evaluate to $false$ as long as any one conjunct is $false$. Thus, OCaml does not even need to evaluate the following conjunct(s) when it detects a $false$ conjunct in the operation.

Out of curiosity's sake, we also do the same evaluation for the *or*/|| expression.

In the case of the || operator, we see the same processing order (from left to right) and the same efficient design: when the first argument is *true*, the second argument will never be evaluated. This is because for an *or* expression, any expression with the *true* conjunct evaluates to *true*. Thus, when the *true* conjunct is placed on the left, a Boolean of *true* is returned immediately without the right conjunct being evaluated.

Our experiments for the || conjunction:

```
(* the || conjunction *)

a_bool true || a_bool true;;

processing true...
- : bool = true

a_bool true || a_bool false;;

processing true...
- : bool = true

a_bool false || a_bool true;;

processing false...
processing true...
- : bool = true

a_bool false || a_bool false;;

processing false...
processing false...
- : bool = false
```

In retrospect, based on Prof Danvy's comments, we know that this phenomenon is called "short-circuit evaluation." Upon further research, I realize that there is actually a lot to unpack. I will start with the definition I found. When evaluating conditions involving **and** and **or**, the language is smart enough to stop evaluating when the rest of the expression becomes unimportant. [2] This is closely related to the evaluation order. There are two kinds of evaluation order:

(i) Applicative-Order Evaluation: evaluates function arguments before passing them to a function

(ii) Normal-Order Evaluation: passes arguments unevaluated to a function

Normal-order evaluation will allow us to short-circuit the evaluation as soon as one of the conditional expressions causes the outcome to become known. [3] This means that the expressions are not evaluated immediately; they are passed to a function first, after which they might be short-circuited before they are evaluated at all. We can implement mechanisms that allow such short-circuit. The following is an example:

```
let f_and e1 e2 =
if e1 then e2
else false;;
```

The above function is effectively implements conjunct operator. Only if e1 is true will the expression e2 be evaluated because otherwise (if e1 is false) we will go to the other branch and outputs FALSE right away. Lazy evaluation does not evaluate an expression until its value is actually needed. Normal order evaluation is essentially lazy evaluation. [3]

This is a significant discovery because now we finally see (perhaps) why for arithmetic operations OCaml will always evaluate from right to left: both expressions are needed aSnd it is necessary to evaluate both before doing the operations. This necessity is more clear when we are doing division because we have to check that the second expression does not evaluate to zero before we start doing the division. In other words, we shall be "lazy" whenever we are sure it is safe to do so; but there are cases where it is wise not to be "lazy." (In fact I discovered during the quick research that there are critics against short-circuit evaluation for Boolean operators, Dijkstra included, perhaps not surprisingly.)

3.5 Question 5: warmup!

This question is a warmup for the following two questions. Implement an OCaml function that given 3 characters, yields a string containing these 3 characters.

We use the *String.make* function to create a string from the character input and concatenate them together.

```
let warmup c0 c1 c2 =
  String.make 1 c0 ^ String.make 1 c1 ^ String.make 1 c2;;
```

In the question prompt, we are given a basic unit test:

```
let test_warmup candidate =
  (candidate 'a' 'b' 'c' = "abc");;
```

We then write another unit test that expands the unit test given above in order to make it more complete.

```
let test_warmupv2 candidate =
  let b0 = (candidate 'a' 'b' 'c' = "abc")
  and b1 = (candidate '1' '2' '3' = "123")
  and b2 = (candidate '1' ' ' 'a' = "1 a")
  and b3 = (candidate ' ' ' ' ' ' = "   ")
  and b4 = (candidate '$' '!' '&' = "$!&")
  in b0 && b1 && b2 && b3 && b4;;
```

We test our simple string concatenation function with the two unit test functions respectively and it passed both of the unit test functions, as expected:

```
let () = assert (test_warmup warmup);;
let () = assert (test_warmupv2 warmup);;
```

Finally, for lucidity's sake, we create a function that passes the unit test but goes not behave as we would like it to. This shows the limitation of testing: that it can alert us when the bug is *present*, but it cannot guarantee that there is no bug in our program.

```
let warmup_bugged c0 c1 c2 =
  if int_of_char c0 >= 32 && int_of_char c1 >= 32 && int_of_char c2 >= 32
  then String.make 1 c0 ^ String.make 1 c1 ^ String.make 1 c2
  else "Gotcha!";;

let () = assert (test_warmup warmup_bugged);;
let () = assert (test_warmup warmup_bugged);;
```

3.6 Question 6: implementing String.map with a simulated for-loop

(a) *In which order are the characters accessed in the given string?*

To figure out what order the characters are assessed in a given string for String.map, we write a traced function *next_letter* that can be passed to String.map. From this traced function, we see that the characters in the string are accessed from left to write in String.map.

```
let to_next c0 = a_char (char_of_int ( int_of_char(c0) + 1));;
String.map to_next "abcd";;

processing 'b'...
processing 'c'...
processing 'd'...
processing 'e'...
- : string = "bcde"
```

(b) *Using a recursive function that operates over a natural number, implement a left-to-right version of String.map.*

I will first explain and show the group code. We include the unit test function first.

```
let test_string_map candidate =
  let b0 = (candidate (fun _ -> 'a') "abcde" = "aaaaa")
  and b1 = (candidate (fun c0 -> char_of_int (int_of_char c0 + 2)) "abcde" = "cdefg")
  and b2 = (candidate (fun _ ->  ' ') "1234" = "    ")
  and b3 = (let n = Random.int 10 in
            candidate (fun _ -> 'p') (String.make n 'a') = (String.make n 'p'))
  in b0 && b1 && b2 && b3;;
```

Then, to apply the function to the string from the character starting from the smallest index, we in fact "start" the recursion from the length of the string (biggest index + 1). We then recursively visit the character of the next smaller index. The base case starts when we reach the character in the smallest index, and we return an empty string as the base case. This starts a series of returns where we create a string of length 1 using the character which has the input function applied to it. Each successive return is concatenated to the right to form the string to be returned. Using this reasoning, we implemented the function and it passed the unit test previously defined.

```
(* implement a left-to-right version of String.map *)
(* group code: a working recursion*)
let string_map_left_right_v0 f s =
  let () = assert (String.length s >= 0) in
  let rec visit n =
    if n = 0
    then ""
    else let n' = n - 1
         in let ih = visit n'
            in ih ^ String.make 1 (f (String.get s n'))
  in visit (String.length s);;

let () = assert (test_string_map string_map_left_right_v0);;
```

I will now make two improvements to the group code. The first is to "pull out" the String.length expression from the recursion to reduce the number of times that this expression is evaluated.

```
(* First improvement *)
let string_map_left_right_v1 f s =
  let n = String.length s
  in let () = assert (n >= 0) in
     let rec visit n =
       if n = 0
       then ""
       else let n' = n - 1
            in let ih = visit n'
               in ih ^ String.make 1 (f (String.get s n'))
  in visit (n);;

let () = assert (test_string_map string_map_left_right_v1);;
```

The second improvement is to simplify the the code by unfolding ih, the inductive hypothesis, as follows.

```
(* Second improvement: substitute the definien visit (n-1) for the variable ih*)
let string_map_left_right_v2 f s =
  let n = String.length s
  in let () = assert (n >= 0) in
     let rec visit n =
       if n = 0
       then ""
       else visit (n - 1) ^ String.make 1 (f (String.get s (n-1)))
     in visit (n);;

let () = assert (test_string_map string_map_left_right_v2);;
```

I will be using the last version from now on.

We can also express this map left to right function using fold:

```
let  string_map_left_right_v3 f s =
  (* base cases: *)
  let n = String.length s
  in let () = assert (n >= 0)
     in let (index,str_n) = fold_right_nat (0,"") (fun (ind,str_n') -> (ind+1, str_n' ^ S
     in str_n;;

let () = assert (test_string_map string_map_left_right_v3);;
```

One key to solving this is to include the index in the recursion, otherwise we will keep using the one position (the last element) in the string to apply the f function on it. The index helped us "proceed" along the string.

(c) *Using a recursive function that operates over a natural number, implement a right-to-left version of String.map.*

In this case, we want to apply the function to the string starting from 0. As opposed to the previous case, we start the recursion from the smallest index and recursively visit the character of the next larger index. The "end" case starts when we reach the length of the string (biggest index + 1), which returns an empty string. Similar to the previous implementation, we recursively visit the character of the next larger index, returning an empty string as the "end" case. This starts a series of returns where we create a string of length 1 using the character which has the input function applied to it. Each successive return is concatenated to the left to form the string to be returned.

Thus, we simply modify the base case and induction step according to the reasoning above. Our program successfully passed the unit test.

```
let string_map_right_left_v2 f s =
  let len = String.length s
```

```
  in let () = assert (len >= 0) in
     let rec visit n =
       if n = len
       then ""
       else String.make 1 (f (String.get s n)) ^ visit (n + 1)
     in visit (0);;

let () = assert (test_string_map string_map_right_left_v2);;
```

We can also express the map function using fold, from the previous folded map left to right function, the only change is in the subscript:

```
let  string_map_right_left_v3 f s =
  (* base cases: *)
  let n = String.length s
  in let () = assert (n >= 0)
     in let (index,str_n) = fold_right_nat (n,"") (fun (ind,str_n') -> (ind-1, String.mak
     in str_n;;

let () = assert (test_string_map string_map_right_left_v3);;
```

To end off, we write unit tests that tests our functions and write a bugged function that passes our unit tests but does not behave as expected.

```
(* For lucidity's sake *)

let string_map_bugged f s =
  if String.length s <= 10
  then string_map_up f s
  else "Gotcha!";;

let () = assert (test_string_map string_map_bugged);;
```

(We are now once again asked to recall Dijkstra's quote...)

3.7 Question 7: implementing String.mapi with a simulated for-loop

(a) *In which order are the characters accessed in the given string?*

We begin again by writing a traced function (a traced version of the *blam* function used in the notes) to determine in which direction characters are assessed in the string. Using this traced function, we found out that similar to String.map, characters within the string are accessed from left to right.

```
let blam_traced i _ = a_char (char_of_int (i + int_of_char '0'));;

let blam i _ = char_of_int (i + int_of_char '0');;
```

```
Stringy.mapi blam_traced "test";;

(*
OCaml output:
processing '0'...
processing '1'...
processing '2'...
processing '3'...
- : string = "0123"
*)
```

(b) *Using a recursive function that operates over a natural number, implement a left-to-right version of String.mapi.*

We include the unit test first, which will be used for parts b and c.

```
let test_string_mapi candidate =
  let b0 = (candidate blam "abcde" = "01234")
  and b1 = (candidate (fun i c0 -> char_of_int (int_of_char c0 + i)) "abcde" = "acegi")
  and b2 = (candidate (fun i _ ->  ' ') "1234" = "    ")
  and b3 = (let n = Random.int 10 in
            candidate (fun i _ -> ('p')) (String.make n 'a') = (String.make n 'p'))
  in b0 && b1 && b2 && b3;;
```

Then mirroring what we have done for question 6, we implement the left-to-right String.mapi and verify that it passes the unit test. The only difference is that we now add an index (n-1) in the induction step.

```
let string_mapi_left_right_v2 f s =
  let n = String.length s
  in let () = assert (n >= 0) in
     let rec visit n =
       if n = 0
       then ""
       else visit (n - 1) ^ String.make 1 (f (n-1) (String.get s (n-1)))
     in visit (n);;

let () = assert (test_string_mapi string_mapi_left_right_v2);;
```

We can also express the mapi left to right function using fold, the only new thing is the added parameter in f:

```
let  string_mapi_left_right_v3 f s =
  (* base cases: *)
  let n = String.length s
  in let () = assert (n >= 0)
     in let (index,str_n) = fold_right_nat (0,"") (fun (ind,str_n') -> (ind+1, str_n' ^ S
     in str_n;;
```

```
let () = assert (test_string_mapi string_mapi_left_right_v3);;
```

(c) *Using a recursive function that operates over a natural number, implement a right-to-left version of String.mapi.*

Similarly, mirroring what we have done for Question 6c, we can implement the right-to-left version of String.mapi and verify that it passes the unit test previously defined. We also write a bugged function for String.mapi function to indicate the limitation of the unit test.

```
(* Part C: implement a right-to-left version of String.mapi *)
let string_mapi_right_left_v2 f s =
  let len = String.length s
  in let () = assert (len >= 0) in
     let rec visit n =
       if n = len
       then ""
       else String.make 1 (f n (String.get s n)) ^ visit (n + 1)
     in visit (0);;

let () = assert (test_string_mapi string_mapi_right_left_v2);;

(* For lucidity's sake *)
let string_mapi_bugged f s =
  if String.length s <= 10
  then string_mapi_up f s
  else "Gotcha!";;

let () = assert (test_string_mapi string_mapi_bugged);;
```

Similarly, we express the mapi right to left function using fold, the only difference is in the subscript of the string:

```
let  string_map_right_left_v3 f s =
  (* base cases: *)
  let n = String.length s
  in let () = assert (n >= 0)
     in let (index,str_n) = fold_right_nat (n,"") (fun (ind,str_n') -> ((ind-1), String.ma
     in str_n;;

let () = assert (test_string_mapi string_map_right_left_v3);;
```

3.8 Question 8: determining the validity of a handful of simplifications

(a) *For any pure expression v of type int, would it be valid to simplify v * 0 into 0?*

Since we are assuming that v is type correct (in this case v is of type int, for any pure expression v it would be valid to simplify $v * 0$ into 0. This is because a pure expression v would not incur

any side effect, raise errors, or diverge. In the case of multiplying by 0, any number multiplied by 0 will be 0. Thus, for a pure expression 0 it is valid to simplify $v * 0$ to 0.

(b) *For any potentially impure expression e of type int, would it be valid to simplify e * 0 into 0?*

In the case of a potentially impure expression e, it would not be valid to simplify $e * 0$ to 0. This is because a potentially impure expression can produce a trace when evaluated. To prove this, consider the example:

```
an_int 5 * 0;;

processing 5...
- : int = 0
```

In the example above, the expression, while returning a result of 0, also returns a trace *processing 5...*. In contrast, the expression 0 will simply return 0 of the int type without the trace.

```
0;;

- : int = 0
```

Thus, the two expressions $e * 0$ and 0 are not observationally equivalent.

(c) *For any pure expression v of type int, would it be valid to simplify v * 1 into v?*

Similar to part a, we are assuming that v is type correct. Based on that assumption, it is valid to simplify $v * 1$ into v because anything multiplied by one will return itself. In this case, 1 is an identity element for the multiplication operator

(d) *For any potentially impure expression e of type int, would it be valid to simplify e * 1 into e?*

Similar to part b, it would not be valid to simplify $e * 1$ into e. Consider a similar example:

```
an_int 5 * 1;;
processing 5...
- : int = 5

an_int 5;;
processing 5...
- : int = 5

an_int 5 * an_int 1;;
processing 1...
processing 5...
- : int = 5

an_int 5 * 1;;
processing 5...
- : int = 5
```

The example above returns a trace *processing 5...*, showing that the two expressions $e * 1$ is not the same as e.

3.9 Question 9: determining the equivalence of let expressions involving impure expressions

(a) *Are the two following expressions equivalent:*

let x1 = e1 and x2 = e2 in (x1, x2);;

let x2 = e2 and x1 = e1 in (x1, x2);;

To answer this question, let us consider the following expressions:

```
let x1 = an_int 5 and x2 = an_int 6 in (x1, x2);;
processing 5...
processing 6...
- : int * int = (5, 6)

let x2 = an_int 6 and x1 = an_int 5 in (x1, x2);;
processing 6...
processing 5...
- : int * int = (5, 6)
```

While the two expressions return similar outputs (a pair, $(5, 6)$), the order in which the definienses *x1 = an_int 5* and *x2 = an_int 6* are processed between the two expressions above is different (recall that the definienses within a let-expression are evaluated from left to right). Since the expressions *x1 = an_int 5* and *x2 = an_int 6* are traced, this means that the two expressions being considered in this question are not equivalent.

(b) *Are the two following expressions equivalent:*

let x1 = e1 in let x2 = e2 in (x1, x2);;

let x2 = e2 in let x1 = e1 in (x1, x2);;

To answer this question, consider a (similar) example of the following expressions:

```
let x1 = an_int 5 in let x2 = an_int 6 in (x1, x2);;
processing 5...
processing 6...
- : int * int = (5, 6)

let x2 = an_int 6 in let x1 = an_int 5 in (x1, x2);;
processing 6...
processing 5...
- : int * int = (5, 6)
```

From the above example, we see that we run into the same issues as with part a, in that the order in which the different definienses within the let-expressions are processed are different between the two expressions, thus rendering them not equal.

Yet, an interesting pattern is observed here. We see that for these two expressions:

```
let x1 = e1 and x2 = e2 in (x1, x2);;
let x1 = e1 in let x2 = e2 in (x1, x2);;
```

we get the same results using the tests we used above. Upon closer thought, these two expressions can be said to be equivalent because 1) They return the same results and 2) They are processed in the same order (left to right in both cases).

3.10 Question 10: determining the equivalence of let expressions involving pure expressions

(a) *Are the two following expressions equivalent:*

let x1 = v1 and x2 = v2 in (x1, x2);;

let x2 = v2 and x1 = v1 in (x1, x2);;

Yes, the two expressions are similar. In the case of a pure expression $v1$ and $v2$, we can expect that there will be no errors or trace that will be returned as the expression is evaluated. This means that if the output is the same in the two expressions (in this case $(x1, x2)$), then the two expressions can be said to be equivalent. In this case, since the two expressions return the same output, they are equivalent.

(b) *Are the two following expressions equivalent:*

let x1 = v1 in let x2 = v2 in (x1, x2);;

let x2 = v2 in let x1 = v1 in (x1, x2);;

This example is tricky. Although the output is the same $((x1, x2))$ for both, the two expressions are not observationally equivalent, i.e., they are impure expressions that will produce side effects. The side effect is produced because nesting let-expressions create new blocks and as a result, new lexical scopes. This belongs to the type of side effects bounded in the OCaml runtime. [4]

To sum up, we can define a pure function by two criteria.

(a) Given the same arguments, the return value will always be the same.

(b) Evaluating the function doesn't result in observable side effects.

As we have seen in this project, it is important for a computer scientist to be mindful about these details, including the order of computation, observational equivalence of expressions (and hence the cases where they are not equivalent any more), subscripts of strings, limitations of unit test... The list goes on, and thus we shall all remember: constant vigilance!

notion of equivalence: two expressions are equivalent if evaluating them carries out the same computation.

4 The case of the Fibonacci numbers

4.1 Overview

This mini-project is the first case study that integrates ... into action. Given a specific problem, how to utilize what we've learned in the past to define new recursive function. Also, the process of simplifying function -¿ computation

4.2 Exercise 1: Expressing a Fibonacci function using fold_right_nat

The Fibonacci function is essentially a double recursion (note: not "mutual recursion"), where we have two base cases and two induction hypotheses. In this exercise, we aim to express the Fibonacci function using fold_right_nat.

Before implementing the Fibonacci function, we first include the unit test.

```
let test_fib candidate =
      (* base cases: *)
  let b0 = (candidate 0 = 0)
  and b1 = (candidate 1 = 1)
      (* intuitive numbers: *)
  and b2 = (candidate 2 = 1)
  and b3 = (candidate 3 = 2)
  and b4 = (candidate 4 = 3)
  and b5 = (candidate 5 = 5)
  and b6 = (candidate 6 = 8)
      (* instance of the induction step: *)
  and b7 = (let n = Random.int 25
            in candidate n + candidate (n + 1) = candidate (n + 2))
  (* etc. *)
  in b0 && b1 && b2 && b3 && b4 && b5 && b6 && b7;;
```

We then start from the given function fib_v5, and re-express the inductive specifications in terms of fold_right_nat.

```
let fib_v5 n_given =
  let () = assert (n_given >= 0) in
  let rec visit n =
    if n = 0
    then (0, 1)
    else let n' = n - 1
         in let (fib_n', fib_n) = visit n'
            in (fib_n, fib_n' + fib_n)
  in let (fib_n, fib_succ_n) = visit n_given
     in fib_n;;
```

The pre-defined function fold_right_nat takes in three arguments, namely the zero case, the successor case (the induction hypothesis), and the given n. It is clear from the above fib_v5 function that

- The zero case is (0,1) (when n = 0).
- The induction step: $fun(fib\_n', fib\_n) \rightarrow (fib\_n, fib\_n' + fib\_n)$, i.e., given a pair (fib_n', fib_n), the induction case should return a new pair which is the given pair shifted right by one step in the Fibonacci sequence, thus giving us $(fib\_n, fib\_n' + fib\_n)$.

We write these inductive specifications above into the new function fib_v6.

```
let fib_v6_0 n_given =
      (* base cases: *)
  let () = assert (n_given >= 0) in
  let (fib_n, fib_succ_n) = fold_right_nat (0, 1) (fun ih -> let (fib_n', fib_n) = ih in (fib_
in fib_n;;

let () = assert(test_fib fib_v6_0);;

(* Further simplifying: *)

let fib_v6 n_given =
    let () = assert (n_given >= 0) in
  let (fib_n, fib_succ_n) =
  fold_right_nat (0, 1) (fun (fib_n', fib_n) -> (fib_n, fib_n' + fib_n)) n_given
  in fib_n;;

let () = assert(test_fib fib_v6);;
```

Our program passed the unit test. This exercise shows that the recursive structure of the Fibonacci sequence can be expressed using fold_right_nat. This is a significant realization because it allows us to generalize: a large number of functions can now be defined by re-writing them into the form that can be "recognized" by the pre-defined fold_right_nat. This is particularly desirable because we do not have to build recursive function from scratch every time we encounter a new function. We only need to pass on the information specific to the new function (the zero case, succ case, and given n), and fold_right_nat will take care of the rest of the recursion.

4.3 Exercise 2: Analyzing the traces of two Fibonacci functions

Loki has spotted that the accompanying resource file also contains a traced version of fib_v4, he has tried it, and he has observed that the trace it emits is not the same as the trace emitted by traced_fib_v0.

(a) Confirm Loki's observation.

fib_v0 and fib_v4 indeed emit different traces, as illustrated below with a sample input of 3.

```
trace_fib_v0 3;;
fib_v0 3 ->
  visit 3 ->
    visit 1 ->
    visit 1 <- 1
    visit 2 ->
      visit 0 ->
      visit 0 <- 0
      visit 1 ->
      visit 1 <- 1
    visit 2 <- 1
  visit 3 <- 2
fib_v0 3 <- 2
- : int = 2

traced_fib_v4 3;;
fib_v4 3 ->
  visit 3 ->
    visit 2 ->
      visit 1 ->
      visit 1 <- 1
      visit 0 ->
      visit 0 <- 0
    visit 2 <- 1
    visit 1 ->
    visit 1 <- 1
  visit 3 <- 2
fib_v4 3 <- 2
- : int = 2
```

(b) Characterize the difference between the two traces.

We make use of the tree structure to visualize the computational steps undergone by the two Fibonacci functions.

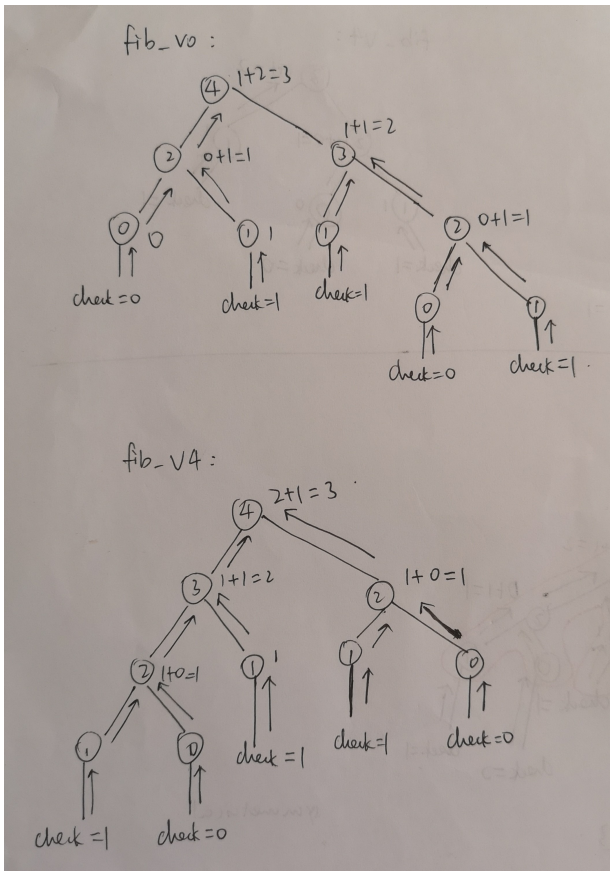

Figure 3: Representation of the recursive structure for each version.

We can first gain some intuition by observing the two recursive structures in the above figure. The two structures are laterally symmetrical and they have the same number of nodes.

(c) Does this difference matter? Using the characterization deduced in the previous question, we compare these two structures in terms of space and time complexities and the computations.

- Same time and space complexity. This is because the two functions undergo the same number of computational steps, shown by the same number of nodes in the two tree structures above.
- Order of evaluating is the crucial reason why the two versions give rise to these laterally symmetrical structures. We recall from the mini-project ***The underlying determinism of OCaml*** that in OCaml, arithmetic operations are evaluated from right to left whereas definienses in a let expression are evaluated from left to right. In fib_v4, the induction step visit(n-1) is executed before the induction step visit(n-2). This is why we observed that after we visited n=4, we visited n=3, n=2, and then n=1. However, in fib_v0, the expression **visit ih" = n"(i+1) where n" = n -2**, is executed before the the expression **ih' = visit n' (i + 1) where n ' = n -1**. This is why the trace function for fib_v4 showed that the sequence of visits were n=4, n = 2, and then n = 0.

- Probing further, we realize that the final results are still the same for fib_v0 and fib_v4 because the recursive relation only involves addition and addition is associative. The result might not be the same any more if the recursive relation involves subtraction or division. This prompts us to make the connection to the notion of pure and impure functions discussed in details in Question 10 (3.11) of the mini-project ***The underlying determinism of OCaml***. The final results are same (except the computational steps and the tracing outputs) because both ih" and ih' are pure expressions. They do not leave traces or raise errors (hopefully) or diverge. If we are using a traced version of fib (as shown in this question), then the results would be different because of the tracing. In this case, ih" and ih' are no longer pure.

5 Palindromes, string concatenation, and string reversal

5.1 Overview

The final mini-project is to apply what we already know about computation to solving palindromes and strings. This project is more interesting because one can approach the problem in many different ways (more freedom); but also more dangerous because there are more hidden pitfalls. The latter realization is prompted in retrospect by the quote of Alan Perils that "[The string] is a perfect vehicle for hiding information."

5.2 Question 1: Concatenate With String.init

We are tasked to code a function that takes two strings and then concatenates them, using String.init. Effectively, we need to code a function that performs the role of the OCaml pervasive, ^ i.e., the string concatenation operator.

We first write the unit test function. We have purposefully included empty strings (either two empty strings, or one empty string with one non-empty string) to test the border case of the function.

```
let test_string_append t =
  let b0 = (t "" "" = "")
  and b1 = (t "" "a" = "a")
  and b2 = (t "a" "" = "a")
  and b3 = (t "intro" "CS" = "introCS")
  and b4 = (t "Happy" "Recess" ="HappyRecess")
  (* etc. *)
  in b0 && b1 && b2 && b3 && b4;;
```

We then test the function that uses the string concatenation operator, ^ with our unit tests. As expected, it passes our unit tests.

```
let () = assert (test_string_append (fun s1 s2 -> s1^s2));;
```

We then implement our own concatenation function. Note String.init has the type $int \rightarrow (int \rightarrow char) \rightarrow string$. String.init n f returns a string of length n, with character i initialized to the result of f i (called in increasing index order), where i is of the type *int* and f is of the type $(int \rightarrow char)$.

It is clear that n should be equal to the combined length of given strings (s1 and s2). Let len1 be the length of the first string. Function f should return the ith character from first string if i¡len1, while returning (i-len1)th character from the second string if otherwise. In OCaml language, we have:

```
let f i =
    if i<len1
    then String.get s1 (i)
    else String.get s2 (i-len1);;
```

Therefore, after renaming f to a meaningful name, we combine the ideas above and code the function as shown below:

```
let string_conca s1 s2 =
  let len1 = String.length s1
```

```
  in let len = len1 + String.length s2
      in let get_character i =
           if i<len1
           then String.get s1 (i)
           else String.get s2 (i-len1)
  in String.init len get_character;;
```

Our function passes the unit tests as expected.

Note that we have been working on coding our own functions that perform the same functions as $+$, $\times$, and etc. Now, we have worked on basic functions on strings through this question.

5.3 Question 2: Reversing a string

We are tasked to code two functions of type *string* $\rightarrow$ *string* that reverse a string. By reversing a string, it means, for example, to map from "abc" to "cba."

We first code some unit tests! Note again, we would love to include the empty case so that border case is included (here we include the empty string). Note that if we reverse any string twice, we should get back the same string. As a result, we include a test to test whether this property holds (in this case b7 below).

```
let test_string_reverse t =
  let b0 = (t "" = "")
  and b1 = (t "a" = "a")
  and b2 = (t "aba" = "aba")
  and b3 = (t "abba" = "abba")
  and b4 = (t "123456" ="654321")
  and b5 = (t "fakeYale" ="elaYekaf")
  and b6 = (t "ABCDEFGHIJKLMN" = "NMLKJIHGFEDCBA")
  and b7 = (t (t "ASDF") = "ASDF")
  (* etc. *)
  in b0 && b1 && b2 && b3 && b4 && b5 && b6 && b7;;
```

a. The first function needs to use String.mapi.

The first case is slightly easier. Let us first recall how String.mapi works. It has the type (*int* $\rightarrow$ *char* $\rightarrow$ *char*) $\rightarrow$ *string* $\rightarrow$ *string* (always to understand the type in OCaml!). String.mapi f s calls f with each character of s and its index (in increasing index order) and stores the results in a new string that is returned. Therefore, f has type (*int* $\rightarrow$ *char* $\rightarrow$ *char*) and s has type *string*, while String.mapi returns a *string*.

Therefore, let len be the length of the string given. What f needs to do is to take in the index i, and return the (len-i-1)th character. We use a String.get function to achieve that. The reason why (len-i-1 is) is because the index of the string starts with 0 and ends with (len-1). For instance index 0 corresponds with index (len-0-1). The combined codes are shown below:

```
let string_reverse_mapi s =
  let len = String.length s
  in String.mapi (fun i c -> String.get s (len-i-1)) s;;
```

b. The second function needs to use recursion over a non-negative integer.

We need to identify the base case and successive case. The base case is when $n = 0$, and we return an empty string. The successive case is to concatenate the (len-n)th character to the end of the ih. The reason why it is (len-n) here instead of (len-n-1) is because we are starting the induction from len instead of (len-1). Note that we have used String.make as well to convert the character to a string of length 1. We then combine the ideas above to the code below:

```
let string_reverse_rec s =
  let len = String.length s
  in let rec visit n =
    if n = 0
    then ""
    else let n' = n - 1
         in let ih = visit n'
            in  ih ^ String.make 1 (String.get s (len-n))
  in visit len;;
```

Note that effectively we are looping from the end of the string to the start.

Both our functions pass our unit tests. We can see from this exercise that mapi is essentially a kind of recursion. This recursion acts on each of the index on the string.

5.4 Question 3: Reversing a concatenated string

Let the reverse function of Question 2 to be represented as string_reverse. Let any strings denoted by s1 and s2. We are tasked to investigate the relationship between string_reverse s1, string_reverse s2, and s1^s2.

Our conjecture is that string_reverse (s1 ^ s2) = (string_reverse s2) ^ (string_reverse s1).

We use a unit test to test it. This is a bit different from the unit tests before. We are taking only two strings in the unit tests, and we claim that for each these two strings s1 and s2 (no matter what they are), the relationship in our conjecture holds.

Note that we are conveniently using the function that we have coded in Question 2 here. We use both versions, one for mapi and one for rec. Both versions hold for all the unit tests we have coded.

```
(*unit tests*)
let test_string_reverse_for_question_3 t =
  let b0 = (t ""  "")
  and b1 = (t "a"  "asdf")
  and b2 = (t "aba"  "wefs")
  and b3 = (t "abba"  "123")
  and b4 = (t "123456" "654321")
  and b5 = (t "fakeYale" "elaYekaf")
  and b6 = (t "ABCDEFGHIJKLMN"  "OPQRSTUVWXYZ")
  (* etc. *)
  in b0 && b1 && b2 && b3 && b4 && b5 && b6;;

(*two versions of testing*)
```

```
let test_string_reverse_mapi s1 s2 =
  string_reverse_mapi (s1^s2) = string_reverse_mapi s2 ^ string_reverse_mapi s1;;
let () = assert (test_string_reverse_for_question_3 test_string_reverse_mapi);;

let test_string_reverse_rec s1 s2 =
  string_reverse_rec (s1^s2) = string_reverse_rec s2 ^ string_reverse_rec s1;;
let () = assert (test_string_reverse_for_question_3 test_string_reverse_rec);;
```

We can also implement Here, we modify from the map_right_left function. In order to reverse, we just make sure we traverse from right to left, but concatenate at the front of the existing string (on the left), hence str_n' Ŝtring.make 1 (String.get s (ind-1)). Another thing is that since we are not applying any additional function to the characters, i.e., they stay what they are, we can just remove the parameter f (note that this is equivalent of applying an identity function).

```
let  string_map_right_left_v3 s=
  (* base cases: *)
  let n = String.length s
  in let () = assert (n >= 0)
     in let (index,str_n) =
     fold_right_nat (n,"") (fun (ind,str_n') ->
     (ind-1, str_n' ^ String.make 1 (String.get s (ind-1)))) n
     in str_n;;

let () = assert(test_string_reverse string_map_right_left_v3 );;
```

5.5 Question 4: Generating palindrome

We first define what a palindrome is. A palindrome is a string such that enumerating its characters from left to right and from right to left give the same enumeration. So for example, "aba" and "abba" are palindromes, but "abc" isn't. (Definition from Professor Danvy)

We are tasked to code a palindrome generator of type $string \rightarrow string$. This function needs to map a string to a palindrome that contains this string. We come up with five ways of making palindrome.

(i) First one is to attach s to the reverse of s. Effectively, we are generating *abba* from *ab*. We use the string_reverse function that we have coded in Question 2 for the purpose here too.

```
let make_palindrome_v1 s =
  s ^ (string_reverse_mapi s);;
```

(ii) The second one is similar to the first one, except that the last letter of s is not repeated twice. Therefore, we will generate *aba* from *ab*. This is still a palindrome. In order to do so, we use the String.sub function to take the last character from string s. Note that String.sub s start len returns a fresh string of length len, containing the substring of s that starts at position start and has length len. Therefore, let len be the length of the string. To get the last character from string s and make it into a string, we use String.sub s (len-1) 1. To get all string except the last character, we use String.sub s 0 (len-1).

We then reverse the string (except the last character) and concatenate accordingly. The codes are:

```
let make_palindrome_v2 s =
  let len = String.length s
  in let short_string = String.sub s 0 (len-1)
  in  short_string ^ (String.sub s (len-1) 1) ^( string_reverse_mapi short_string);;
```

(iii) The third one is just to repeat the same string s twice, before we run it through the first method.

```
let make_palindrome_v3 s =
  make_palindrome_v1 (s^s);;
```

(iv) The fourth one is to repeat the first character twice and then apply it to version 2. Note that this method only works if the string is not empty (i.e. at least have one character). Therefore, we assert that the string is not empty first before proceeding. We use String.get to get the first character, and then use String.make to convert the character to a string, before we concatenate the character to the original string s.

```
let make_palindrome_v4 s=
  let () = assert (s != "")
    in  make_palindrome_v2 ((String.make 1 (String.get s 0) ) ^ s)
;;
```

(v) The fifth one is to generate a random lowercase letter to attach at the back of the string, and then generate the palindrome via the version 1.

In order to generate the random lowercase number, we generate a random number between 0 and 25, and then add to the ASCII code of 'a', before converting the random code to the corresponding character.

```
let make_palindrome_v5 s =
  let a = Char.chr (Char.code 'a' + Random.int 26)
    in make_palindrome_v1 (s^ String.make 1 a);;
```

In fact, there can be a lot more ways of generating palindromes. We can add any random strings at the back or before the original string. We can also repeat the original string a few times. As long as we pass the changed string to version 1 or version 2 above, we can get a palindrome that contains the original string.

We give a few examples of the results of our palindrome generators:

(i) "Yale-NUS"

v1: "Yale-NUSSUN-elaY"

v2: "Yale-NUSUN-elaY"

v3: "Yale-NUSYale-NUSSUN-elaYSUN-elaY"

v4: "YYale-NUSUN-elaYY"

v5: "Yale-NUSggSUN-elaY"

(ii) "a"

v1: "aa"

v2: "a"

v3: "aaaa"

v4: "aaa"

v5: "aaaa"

(iii) "Chengpei"

v1: "ChengpeiiepgnehC"

v2: "ChengpeiepgnehC"

v3: "ChengpeiChengpeiiepgnehCiepgnehC"

v4: "CChengpeiepgnehCC"

v5: "ChengpeijjiepgnehC"

The interesting thought from this exercise is that we should be mindful of whether we can use previous functions we have coded to shorten the current function under construction. For instance, we utilized the string_reverse function a lot in this exercise, which significantly reduces our work. In addition, we also use the v1 and v2 versions of palindrome makers in other versions.

5.6 Question 5: Detecting Palindrome

We are tasked to code a palindrome detector of type *string* → *bool*. If the input string is a palindrome, the function should output true. If otherwise, the function should output false.

We have input some unit tests of palindromes and some unit tests of non-palindromes here.

a. Implement using String.mapi.

The central idea of using String.mapi is to compare whether the reverse of the given string is the same as the given string. If and only if this condition is satisfied, the given string is a palindrome.

We get the reverse of the string similar to what we have done in Question 2. We then compare the reverse with the original string.

```
let palindromep_mapi s =
  let len = String.length s
  in let reverse = String.mapi (fun i c -> String.get s (len-i-1)) s
  in reverse = s
;;
```

b. Implement using recursion over a non-negative integer.

The central idea of this function is to compare each character of the string with the corresponding character of the "flipped" side of the string. For instance, we compare the index 0 with index (len-1) character, where len is the length of the string. In general, we are comparing index (n-1) and (len-n). Hence, the successive case is that all the previous comparisons must be true and the current comparison is also true, i.e.:

```
ih && (String.get s (n-1) = String.get s (len-n))
```

The base case can technically be n=0. However, we do not necessarily need to run the recursion until n=0, as we only need to compare until n is the middle index of the string. Any further comparison is unnecessary. Therefore, we let the recursion base case be true, and stops at (n¡=len/2), where len is the length of the string. We also start the recursion at len.

```
let palindromep_rec s =
  let len = String.length s
  in let rec visit n =
    if n <= len /2
    then true
    else let n' = n - 1
         in let ih = visit n'
            in  ih && (String.get s (n-1) = String.get s (len-n))
   in visit len;;
```

Our functions pass the unit tests. Note that this question again shows us that mapi and recursion are effectively the same. Mapping process is effectively recursion through the index of the string.

5.7 Question 6: Reverse Palindrome

We need to implement a function that reverses a palindrome.

The reversion of palindrome is just itself. Here, we need to test whether the given string is indeed palindrome. We use assert function with the test that we have coded in question 5. Then we just output the original string if it passes the assert.

```
let reverse_palindrome s =
  let ()= assert (palindromep_mapi s)
  in s;;
```

Note that we can also use the reverse function coded in question 2, since palindrome is just a special case of string. However, our function above is obviously much faster (if the given string is indeed palindrome). Therefore, putting a little bit thought into coding for the special cases can save computational time and resources.

Based on Prof Danvy's comments, a improved version of palindromeep_rec function is shown below:

```
(* Improving the palindromep_rec function: simplifying, reducing number of evaluations *)

let palindromep_rec_v1 s =
  let len = String.length s
  in let rec visit n =
    if n = len /2
    then true
    else (String.get s (n-1) = String.get s (len-n)) && visit(n - 1)
     in visit len;;

let () = assert (test_palindromep palindromep_rec_v1);;
```

The improvements include:

- Unfold the declaration of ih, hence simplifying the program.
- Exploiting the efficiency of the short-circuit evaluation (lazy evaluation) explored in previous exercise, we can move the recursive trigger after the **and** conjunct. This way, the comparison will stop at the first characters that differ.
- The base case can just be n = len / 2 because len/2 is the maximum number of comparisons we need to make before we can conclude whether a string is a palindrome (that is, even when assuming the worst case).

5.8 Question 7: Fibonaccize a string

Implement an OCaml function of type *string* → *string* that, given a string containing the successive characters c0, c1, c2, etc., constructs a string that contains the following successive characters:

- ' ', i.e., the space character,
- F0 occurrences of c0, where F0 is the first Fibonacci number,
- F1 occurrences of c1, where F1 is the second Fibonacci number,
- F2 occurrences of c2, where F2 is the third Fibonacci number,
- F3 occurrences of c3, where F3 is the fourth Fibonacci number,
- etc.

We coded some unit tests with increasing number of characters. Note that the space character needs to be included in all. Also we are using the Fibonacci function in the previous section of the mid-term for this question.

We are using a recursion here. Our base case stops at n=len, where len is the length of the string. In this case, there are no more characters to work on, and we will return an empty string.

We here start with index zero and successfully add one to the index (this is slightly different from our usual practice of reducing one). For each index i, we include the fib(i+1) number of this character with the help of the function String.make. And then we move on to the next index.

```
let fibonaccize s =
  let len = String.length s
  in let rec visit n =
    if n = len
    then ""
    else let n' = n + 1
         in let ih = visit n'
            in  String.make (fib_v5 (n+1)) (String.get s n) ^ih
     in " " ^ (visit 0)
;;
```

We claim that the length of the resulting string equals fib(len + 2). We know this because of the relation $\sum_0^n fib(i) = fib(n+2)$. We use a unit test to test, in which we pass in a random string of random length. We first fibonaccize the string and get the length of the resulting string with the

function String.length. We then calculate fib(len+2), where len is the length of the original string. We compare the two results afterwards.

```
let len_fibonaccize s =
  (String.length (fibonaccize s)) = fib_v5 ((String.length s) + 2);;
```

We are glad to see that it passes our unit test! To generalize the way we solve this problem, we first look at some simple examples with string lengths between 0 and 4. We then try to generalize a pattern from these examples. We then conjecture and realize that the resulting string length is connected with $\sum_0^n fib(i) = fib(n+2)$, as shown in our previous answer of "A miscellany of recursive programs" (2.4). We then are able to solve this question.

Therefore, when we face difficult problems, it is great to try with some simple examples. Then try to generalize, and then test. Note that this is exactly what we have learnt in Scientific Inquiries about the scientific methods!

6 More for The Road

6.1 Overview

As hinted in the section on expressing fold_right_nat and parafold_right_nat (2.5), we will now implement two bonus functions String.map and String.mapi. The solution is inspired by realizing the analogical relation between the duo String.map vs String.mapi and the duo fold_right_nat vs parafold_right_nat. We also make use of skills and insights gained in the mini-project, ***The underlying determinism of OCaml***(3)

6.2 Exercise 7: Define *String.map* using *String.mapi*

We are tasked to define *String.map* using *String.mapi*.

As always, we start with the unit tests. Here, the type of String.map is $(char \rightarrow char) \rightarrow string \rightarrow string$. As a result, the input of the unit tests include a function, which takes a character to a character, and a string.

b0 and b1 are changing whole string to 'a' and 'b'. b2 and b4 are converting between lower are upper case. b3 is to move to character corresponding to ASCII code plus one. We expect that String.map passes our unit tests and indeed it does.

```
let test_string_map t =
  let b0 = (t (fun c-> 'a') "aasdfasdf" = "aaaaaaaaa")
  and b1 = (t (fun c-> 'b') "aasdfasdf" = "bbbbbbbbb")
  and b2 = (t Char.lowercase_ascii "ABC" = "abc")
  and b3 = (t (fun c->  Char.chr((Char.code c ) +1 )) "abcde" = "bcdef")
  and b4 = (t Char.uppercase_ascii "asdfg" = "ASDFG")
  (* etc. *)
  in b0 && b1 && b2 && b3 && b4;;

let ()=assert(test_string_map String.map);;
```

The function is rather simple. We just need to define a function g that can take in the index (with type integer). Since the index does not do anything, we can just pass this through and get the result. The function passes our unit tests.

```
let string_map f s =
  let g = fun n c -> f c
    in String.mapi g s;;

let ()=assert(test_string_map string_map);;
```

The ease of converting *String.mapi* to *String.map* makes sense given that *String.map* can be seen as a special case of *String.mapi* where the index n is not used.

6.3 Exercise 8: Defining String.mapi using String.map and String.init

We are tasked to define *String.mapi* using *String.map* and *String.init*.

Again, the unit tests are tricky. b0 and b1 are changing letters based on the index. So the first letter does not get changed, the first second letter gets to change by one ASCII code, and the third

letter gets to change by two ACSII code (i.e. 'a' becomes 'c' or the other way round), and so on. b2 and b4 are not really utilizing the index and just converting between lower and upper case. b3 is generating a string of the equal length with s but replacing the content with numbers equal to the respective index.

Indeed, as we expect, our unit tests pass if we input String.mapi.

```
let test_string_mapi t =
  let b0 = (t  (fun n c -> Char.chr((Char.code c) +n)) "abcd" = "aceg")
  and b1 = (t (fun n c -> Char.chr((Char.code c) -n)) "aceg" = "abcd")
  and b2 = (t (fun n c -> Char.lowercase_ascii c) "ABC" = "abc")
  and b3 = (t (fun n c -> Char.chr(n + Char.code '0')) "wsfg" = "0123")
  and b4 = (t (fun n c -> Char.uppercase_ascii c) "asdfg" = "ASDFG")
  (* etc. *)
  in b0 && b1 && b2 && b3 && b4;;

let ()=assert (test_string_mapi String.mapi);;
```

The function itself is rather tricky and we spent substantial amount of time on it. We first define g function which takes in an integer n and returns $String.map\,(f\,n)\,s$. What exactly this does is: if we assume that every single letter is of index n, the result on string s is $String.map\,(f\,n)\,s$. However, we know that this is not what we want. There is only one character in this entire resulted string that is correct, i.e. the nth position. Hence, we define a function h, which takes in an integer n. We use *String.get* to obtain the correct resulted character from the entire string output from $g\,n$. Hence, what h does is: if we input any index n, h would return a correct character of corresponding to the index n.

By now, we have finished our set up. Based on the guide of the question, we know we still need to use *String.init*. Let us look at what *String.init* does again: $String.init\,n\,f$ returns a string of length n, with character i initialized to the result of $f\,i$ (called in increasing index order). We realize that it is what exactly we need. n corresponds to the length of the string s, which we can get by *String.length*. f is exactly our h.

```
let string_mapi f s =
  let g n = String.map (f n) s
    in let h n = String.get (g n) n
       in String.init (String.length s) h;;

let ()=assert (test_string_mapi string_mapi);;
```

Note the type needs to match (which amazingly does here). This is, in fact, the nice thing about OCaml. A lot of times, the errors in the program are related to the types. As a result, debugging on OCaml is slightly easier, because we do not need to worry about one whole category of errors (i.e. type mismatch), since OCaml tells us that we are wrong!

In addition, using OCaml makes us a lot more conscious of the issues with types. That might in fact aid us in solving a programming problem. For instance, in this question, the way we thought of the solution really came from looking at the type. We realized that *String.mapi* has type $(int \to char \to char) \to string \to string$. Therefore, the two inputs f and s has type $int \to char \to char$ and *string* respectively. The difficulties come with f, because *String.map* takes a function of the

type $char \to char$, while *String.init* takes a function of the type $int \to char$. They are all different! Therefore, we thought, the only way for *String.map* to take the function f is if we add input n into f, i.e. to form $f\,n$. This function has the type we need: $char \to char$. This became the point of entry for the solution of the problem!

Finally, we would like to draw a parallel between this question with Exercise 9 (2.5) of "A miscellany of recursive programs". In Exercise 9, we ask to convert between parafold and fold. In this question, we need to convert between *String.map* and *String.mapi*. *String.map* corresponds to fold since it does not allow index input, while *String.mapi* corresponds to parafold. However, the parallel is not very strong, as we need to use another function *String.init* to assist us: so in some sense, *String.mapi* and *String.map* are not equivalent, unlike parafold and fold. This is because of the type restriction on the module String. We cannot change the input into type pair like in fold and parafold. The rigidity is not too surprising, if we understand that *String.mapi* and *String.map* are special cases of parafold and fold, like what we have done in Exercise 6 (3.6) and Exercise 7 (3.7 of "The underlyng determinism of OCaml".

7 Conclusion

This mid-term project ties together many fundamental ideas in the first half of the course and there are a few that particularly speak to me, which I have been reflecting upon.

- We can use finite rules to define potentially infinite steps of computation. This is especially salient in recursion. At each step, we do not need to remember all of the preceding information. We just need to "remember" the information that is necessary for us to get to the next step (induction step); and of course, when to stop (base case). What information is necessary then depends on the nature of the problem we are dealing with, for example the even/odd predicate and the Fibonacci sequence clearly have different recursive nature and hence different things we want to "remember" at each step. This idea reminds one of the concept syntax and grammar that we have learned in the first few lessons, where we use finite grammatical rules to define a potentially infinite number of expressions. I remember reading somewhere about the analogy of measuring a long paper extending from the ground to the moon, with only a piece of A4 paper. We (both us and computers) have limited memory space, finite like the A4 paper. Recursion is thus powerful because it frees us from remembering every piece of information in all the previous steps, which could either be infinite or exponentially large.
- We have observed many isomorphisms in this project (and computer science in general). The expressions can have different syntax but are observationally equivalence. Another intuition that stayed with me was gained from extending the sumtorial function to sum function. The latter has really similar structure to the former but is a higher abstraction because it is generalized to the sum of any functions taking the sequence of natural numbers, as opposed to just the sum of the sequence of natural numbers. This is one way we can extend our computational power. Another relevant observation is that we can express recursive structures in terms of each other, by tweaking and playing around with the inductive specification - same underlying structure but different manifestations.

While I was ruminating over the project, a curious structure in the Yale-NUS Library collection caught my eyes. It is a metal sculpture depicting a ball on the top of a half-folded strip of paper that extends from a pyramid base.

Figure 4: A curious structure, found at Yale-NUS Library.

I started to fold a paper imitating its shape and it was not as easy as it seemed. Then I realized it was just repeatedly folding a triangle - a recursion! fold_right_triangle! Then I realized that the distance between two triangles are closer at the lower end, that the ball was dangerously close to the corner, ... I could go on. But the point is that not only does this structure provoked thoughts about recursion and computation, but the very process of discovering patterns and spotting potential logical loopholes mirrored the discovery process during this project.

Like many people, I passed by this sculpture multiple times every day, blind to its curious structure. Unfortunately, sometimes I find myself guilty of being oblivion to the fascinating structures of mathematical objects unfolding right in front of my eyes (due to an insufficient contemplation, good insight etc.) It is therefore my mission in the next half of the course and beyond to train my mind to be sensitive and appreciate such things with curiosity and vigilance.

8 Acknowledgements

I would like to give credits to my group mates, Chengpei Liu, Zhi Yi Yeo, and Jincong Chu, who contributed to the completion of this project. They have generously share their good ideas and their meticulousness helped me avoid many pitfalls. To be specific, Chengpei was primarily responsible for the palindrome project, bonus questions, and code-checking, Zhi Yi was primarily responsible for the Ocaml project, and I was primarily responsible for the Fibonacci project. In order to maximize my learning, I tried to re-work and re-write the projects that I was not primarily responsible for and challenged myself by trying to think of new explanations or new solutions, but I cannot measure how much I've learned just by reading my teammates' work. I am also grateful for the opportunity to attend YSC1212: Introduction to Computer Science, 2019/20 Rendition, taught by Professor Olivier Danvy who is ever so encouraging and wise.

References

[1] O. Danvy, *Introduction to Computer Science (YSC1212)*, 2019, Week 01-07.

[2] https://www.cs.drexel.edu/ jpopyack/Courses/GovSchool/2005/Sp04/lectures/08.1_compound_conditionals/lazy.html?CurrentSlide=5.

[3] http://web.eecs.utk.edu/ bvanderz/teaching/cs365Sp12/notes/other-functional-issues.html