Generating all balanced parentheses

In this short note, we look at the problem of generating all balanced parentheses in a lazy way. In the revision session, we looked at a way to do this by generating all binary sequences and filtering out those that do not represent balanced sequences. A question arose as to whether it is possible to generate only valid balanced sequences. Here, we will see a way for doing this. This exercise might be instructive both for working with lazy lists and also for discrete mathematics.

1 Recursively generating all balanced parentheses

The problem with finding a recursive way to generate all balanced parentheses is that we need to avoid repeating the same sequence twice. For example, if we generate all parentheses of length n by generating all parentheses of length $k = \{0, 2, 4, \ldots n-2\}$ and then for each k merging all sequences of length k with those of length $n-k$, i.e.

$$\texttt{generate\_balanced\_parens}(k) \ @ \ \texttt{generate\_balanced\_parens}(n-k).$$

The problem with this method is that we generate ()()() twice, once for the grouping ()()() and once for the grouping ()()().
It turns out that a small modification to this approach, leads to a correct solution. The "problematic sequences" are the ones that have multiple top-level parentheses, e.g. $(\ldots)(\ldots)\ldots(\ldots)$. We want to generate each of these sequences only once. We do this by fixing the length of the first top-level bracket (note that there always exists one) and say it is k. In order to generate the contents of the first parentheses we use

$$(\texttt{generate\_balanced\_parens}(k-2)) \ @ \ \texttt{generate\_balanced\_parens}(n-k).$$

Hence, we can do this for every $k = \{2, 4, 6 \ldots, n\}$ and get
Connection to the Catalan numbers. The Catalan number C_n is defined as the number of balanced parentheses sequences of length $2n$. It turns out that our approach above, proves the following identify

$$C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$$

This identity can be used to prove (see e.g. here) the formula,

$$C_n = \frac{1}{n+1} \cdot \binom{2n}{n}.$$

2 Example

Note that for $n = 2$, the set of possible parentheses are [()] and for $n = 4$, they are [()(); (())].
For $n = 6$,

- For $k = 2$, the first top-level parenthesis is () and we can merge this with any of the parentheses of length $n - 2 = 4$. This gives the following sequences ()()() and ()(()).
- For $k = 4$, the first top-level parenthesis is going to be (any parentheses seq of lenght 2), so (()) and then followed by any balanced parentheses of length $n - 4 = 2$. This gives only (())().
- For $k = 6$, the first top-level parenthesis is going to be (any parentheses seq of lenght 4), so (()()) or (()) and then followed by any balanced parentheses of length $n - 6 = 0$. This gives (()()) and (()).

So, in total we get these five sequences (and these are all of them).

3 Lazy list operations

In the implementation, we will use the following functions:

1. A function to enumerate all naturals from a to b, i.e. $a+1, a+2, \ldots, b$ (`from_to`).

2. Map function for lazy lists (`mapl`).
3. Flatten function for lazy lists. Given a lazy list of lazy lists (all finite), return a single finite lazy list containing their elements (`flatten`).
4. A function that takes two lazy lists of lists and creates a single lazy list containing lists of the form `x @ y`, where x is an element of the first lazy list and y is an element of the second lazy list (`cross`). We will use this to merge the elements of the two lazy lists obtained recursively. For example, if we have

   ```
   s1 = Cons([‘(’, ‘)’, ‘(’, ‘)’], fun() -> Cons([‘(’, ‘(’, ‘)’, ‘)’], fun () -> Nil))
   ```

   and

   ```
   s1 = Cons([‘a’], fun() -> Cons([‘b’], fun () -> Nil))
   ```

   Then, it should return,

   ```
   ans = Cons([‘(’, ‘)’, ‘(’, ‘)’, ‘a’], fun() -> Cons([‘(’, ‘)’, ‘(’, ‘)’, ‘a’], fun() ->
     Cons([‘(’, ‘(’, ‘)’, ‘)’, ‘b’], fun () -> Cons([‘(’, ‘(’, ‘)’, ‘)’, ‘b’], fun () ->
                                          Nil))))
   ```

5. To test the above implementations, the following functions may be useful.

   ```
   let rec to_list = function
     (Cons(x, xs)) -> x :: to_list (xs ())
   | Nil -> [];;

   let rec from_list = function
     x::xs -> Cons(x, fun () -> from_list xs)
   | [] -> Nil;;
   ```

Try to implement (and test) these on your own before proceeding.

Implementation of `mapl`:

```
type 'a llist = Nil | Cons of 'a * (unit -> 'a llist);;

let rec mapl f = function
  Cons(x, xs) -> Cons(f x, fun () -> mapl f (xs()))
| Nil -> Nil;;
```

Implementation of `flatten`:

```
let rec flatten = function
  Nil -> Nil
| Cons(Nil, xs) -> flatten (xs ())
| Cons(Cons(y, ys), xs) -> Cons(y, fun () -> flatten (Cons(ys(), xs)));;

(* Example *)
let c = from_list [ from_list [1;2;3]; from_list [4;5]; from_list [6;7;8;9] ];;
to_list (flatten c);;
```

Implementation of `cross`:

```
let rec cross_aux seq1 seq2 original_seq2 = match seq1, seq2 with
  Cons(x, xs), Cons(y, ys) -> Cons(x @ y, fun () -> cross_aux seq1 (ys ())
      original_seq2)
| Cons(x, xs), Nil -> cross_aux (xs ()) original_seq2 original_seq2
| Nil, _ -> Nil;;

let cross seq1 seq2 = cross_aux seq1 seq2 seq2;;

(* Example. *)
let a = [ [1]; [2]; [3]; [4] ];;
let b = [ [10]; [20]; [30] ];;

let ans = cross (from_list a) (from_list b);;
to_list ans;;
```

4 Putting it all together

```
let rec generate_balanced_parens n =
  if n = 0 then Cons([], fun () -> Nil) (* Base case *)
  else
     flatten (mapl (fun k -> (* Do this for all even k. *)
        (* Merge the solutions obtained recursively. *)
        cross (mapl (fun y -> '('::y @ [')']) (generate_balanced_parens (2 * k)))
              (generate_balanced_parens (n - 2 * k - 2))) (from_to 0 (n/2 - 1)));;
```

5 Verifying the implementation

We verify correctness with some small tests,

```
to_list (generate_balanced_parens 2);;
to_list (generate_balanced_parens 4);;
to_list (generate_balanced_parens 6);;
(*
# to_list (generate_balanced_parens 2);;
- : char list list = [['('; ')']]
# to_list (generate_balanced_parens 4);;
- : char list list = [['('; ')'; '('; ')']; ['('; '('; ')'; ')']]
# to_list (generate_balanced_parens 6);;
- : char list list =
[['('; ')'; '('; ')'; '('; ')']; ['('; ')'; '('; '('; ')'; ')'];
 ['('; '('; ')'; ')'; '('; ')']; ['('; '('; ')'; '('; ')'; ')'];
 ['('; '('; '('; ')'; ')'; ')']]
*)
```

And for bigger values we just check that the number of solutions matches the Catalan numbers.

```
List.length (to_list (generate_balanced_parens 8));; (* 14 *)
List.length (to_list (generate_balanced_parens 10));; (* 42 *)
List.length (to_list (generate_balanced_parens 12));; (* 132 *)
List.length (to_list (generate_balanced_parens 14));; (* 429 *)
```

We can also check that indeed sequences are generated in a lazy manner by enabling trace for `generate_balanced_parens`:

```
#trace generate_balanced_parens;;
let ans = generate_balanced_parens 14;;
let get_tail (Cons(_, xs)) = xs ();;

let ans = get_tail ans;;
let ans = get_tail ans;;
let ans = get_tail ans;;
let ans = get_tail ans;;

(*
# let ans = get_tail ans;;
generate_balanced_parens <-- 0
generate_balanced_parens --> Cons ([], <fun>)
generate_balanced_parens <-- 2
generate_balanced_parens <-- 0
generate_balanced_parens --> Cons ([], <fun>)
generate_balanced_parens <-- 0
generate_balanced_parens --> Cons ([], <fun>)
generate_balanced_parens --> Cons (['('; ')'], <fun>)
val ans : char list llist =
  Cons
   (['('; ')'; '('; ')'; '('; ')'; '('; ')'; '('; ')'; '('; '('; ')'; ')'],
   <fun>)
# let ans = get_tail ans;;
generate_balanced_parens <-- 2
```

```
generate_balanced_parens <-- 0
generate_balanced_parens --> Cons ([], <fun>)
generate_balanced_parens <-- 0
generate_balanced_parens --> Cons ([], <fun>)
generate_balanced_parens --> Cons (['('; ')'], <fun>)
generate_balanced_parens <-- 2
generate_balanced_parens <-- 0
generate_balanced_parens --> Cons ([], <fun>)
generate_balanced_parens <-- 0
generate_balanced_parens --> Cons ([], <fun>)
generate_balanced_parens --> Cons (['('; ')'], <fun>)
val ans : char list llist =
  Cons
   (['('; ')'; '('; ')'; '('; ')'; '('; ')'; '('; '('; ')'; ')'; '('; ')'],
   <fun>)
# let ans = get_tail ans;;
generate_balanced_parens <-- 0
generate_balanced_parens --> Cons ([], <fun>)
generate_balanced_parens <-- 4
generate_balanced_parens <-- 2
generate_balanced_parens <-- 0
generate_balanced_parens --> Cons ([], <fun>)
generate_balanced_parens <-- 0
generate_balanced_parens --> Cons ([], <fun>)
generate_balanced_parens --> Cons (['('; ')'], <fun>)
generate_balanced_parens <-- 0
generate_balanced_parens --> Cons ([], <fun>)
generate_balanced_parens --> Cons (['('; ')'; '('; ')'], <fun>)
val ans : char list llist =
  Cons
   (['('; ')'; '('; ')'; '('; ')'; '('; ')'; '('; '('; ')'; '('; ')'; ')'],
   <fun>)
# let ans = get_tail ans;;
generate_balanced_parens <-- 0
generate_balanced_parens --> Cons ([], <fun>)
generate_balanced_parens <-- 2
generate_balanced_parens <-- 0
generate_balanced_parens --> Cons ([], <fun>)
generate_balanced_parens <-- 0
generate_balanced_parens --> Cons ([], <fun>)
generate_balanced_parens --> Cons (['('; ')'], <fun>)
val ans : char list llist =
  Cons
   (['('; ')'; '('; ')'; '('; ')'; '('; ')'; '('; '('; '('; ')'; ')'; ')'],
   <fun>)
*)
```