Module Canonical

1. module P = *Permutation*

Type décrivant une tresse représentée sous la forme $\Delta^r A_1 ... A_n$ où $A_1, ..., A_n$ sont des tresses simples. Les tresses simples sont ici décrites par leurs permutations associées.

type $braid\_permlist = \{bpl\_size : int;\ delta\_power : int;\ permlist : P.permutation\ list\}$

Tresses de base : Δ et la tresse vide

let $delta\ n = \{bpl\_size = n;\ delta\_power = 1;\ permlist = [\,]\}$

let $empty\ n = \{bpl\_size = n;\ delta\_power = 0;\ permlist = [\,]\}$

2. Conversion d'une tresse décrit par un mot sur l'alphabet des générateurs en une représentation sous forme de liste de permutations.

Algorithme : Si l'on rencontre un σ_i alors on ajoute à la liste des permutations la transposition correspondante. Si l'on rencontre un ${\sigma_i}^{-1}$, on utilise le fait que ${\sigma_i}^{-1} = \Delta^{-1}\Delta{\sigma_i}^{-1}$ et que $\Delta{\sigma_i}^{-1}$ est une tresse positive. On l'ajoute à la liste et on fait remonter le Δ^{-1} en composant par τ .

On simplifie les calculs en se rappelant que $\tau^2 = id$.

```
let get_permlist_decomposition (b : Braid.braid) =
    let n = b.Braid.size in
    (* Calcule Δ × σ⁻¹ que l'on sait être un facteur canonique *)
    let delta = P.make_delta n in
    let neg_to_perm x = P.compose_transpose_left delta (−x − 1) (−x) in
    (* Applique l'algorithme, en mémorisant la puissance de τ à appliquer *)
    let (delta_pow, perm_stack) =
        List.fold_left (fun (delta_pow, perm_stack) x →
                            if x > 0 then
                              (delta_pow,
                              (P.make_transpose (x − 1) x n, 0) :: perm_stack)
                            else
                              (delta_pow − 1,
                               (neg_to_perm x, 0) ::
                               (List.map (fun (p, tau_pow) → (p, tau_pow + 1)) perm_stack)))
                         (0, []) b.Braid.word in
    (* On applique la puissance de τ si nécessaire *)
    let perm_list =
        List.rev_map (fun (perm, tau_pow) → if (tau_pow mod 2) ≠ 0
                                                then P.tau perm
```

```
                                        else perm)
                    perm_stack in
  {bpl_size = n; delta_power = delta_pow; permlist = perm_list}
```

3. Mise sous forme maximale à gauche (left-weighted) d'une liste de permutations (et non d'une tresse complète : voir canonicize pour celà).

Algorithme : On considère tour à tour des couples de tresses simples. On calcule alors le «finishing set» de la première et le «starting set» de la seconde. On fait la différence ensembliste entre le «starting set» et le «finishing set» et l'on ajoute les éléments de la différence à la première tresse, et on les retire de la deuxième (concrètement, une composition de permutations).

```
let make_left_weighted start_pl =
  let continue = ref true in
```

trouver la décomposition maximale à gauche de A_1A_2, de permutations associées respectives p1 et p2

```
  let rec make_lw_pair p1 p2 =
    let s2 = P.starting_set p2 and f1 = P.finishing_set p1 in
    if s2 = [] (* p2 = id *) then [p1]
    else if f1 = [] (* p1 = id *) then [p2] (* peu probable *)
    else match P.set_difference s2 f1 with
      | [] → [p1; p2] (* p1 facteur maximal *)
      | i :: _ → continue := true;
                 let p1′ = P.compose_transpose_left p1 (i − 1) i in
                 let p2′ = P.compose_transpose_right p2 (i − 1) i in
                 make_lw_pair p1′ p2′
  in

  let make_lw_pair′ p1 p2 =
    let b = P.meet (P.compose (P.inv p1) (P.make_delta (Array.length p1))) p2 in
    if ¬ (P.is_id b) then (
      continue := true;
      List.filter (fun x → ¬ (P.is_id x)) [P.compose p1 b; P.compose (P.inv b) p2]
    ) else (
      List.filter (fun x → ¬ (P.is_id x)) [p1; p2]
    )
  in
```

On réduit la liste des facteurs à partir de la fin

```
let rec reduce = function
  | [] → []
  | p1 :: q → match reduce q with
      | [] → [p1]
      | p2 :: q′ → make_lw_pair′ p1 p2 @ q′
in
let current_pl = ref start_pl in
while !continue do
  continue := false;
  current_pl := reduce !current_pl
done;
!current_pl
```

4. Met une tresse décrite sous forme de liste de permutations sous forme canonique.

Algorithme : Rend la liste de tresses simples maximale à gauche, et collecte les tresses simples égales à Δ en tête de liste (s'il y a un Δ, il est forcément en tête de liste). Incrémente le champ delta_power en conséquence.

```
let canonicize bpl =
  let dp = ref bpl.delta_power
  and pl = ref (make_left_weighted bpl.permlist)
  and ok = ref false in
  while ¬ !ok ∧ !pl ≠ [] do
    if P.is_delta (List.hd !pl) then (
      dp := !dp + 1;
      pl := List.tl !pl
    ) else (
      ok := true
    )
  done;
  {bpl_size = bpl.bpl_size; delta_power = !dp; permlist = !pl}
```

Met une tresse décrite par un mot de tresse sous forme canonique.

```
let canonical_form b = canonicize (get_permlist_decomposition b)
```

5. Tests d'égalité.

Revient à mettre sous forme canonique, et les comparer.

```
let braids_equal b1 b2 = (canonical_form b1) = (canonical_form b2)
```

```
let permlists_equal bpl1 bpl2 = (canonicize bpl1) = (canonicize bpl2)
```

6. Opérations algébriques sur les tresses sous forme de listes de permutations.
Produit.

Formule : $(\Delta^p A_1 ... A_l)(\Delta^q B_1 ... B'_l) = \Delta^{p+q} \tau^q(A_1) ... \tau^q(A_l) B_1 ... B_l$

```
let product a b =
  {bpl_size = a.bpl_size;
   delta_power = a.delta_power + b.delta_power;
   permlist = (if b.delta_power mod 2 ≠ 0 then List.map P.tau a.permlist else a.permlist)
              @ b.permlist }

let (< * >) = product
```

Inverse.

Formule : $(\Delta^r A_1 ... A_l)^{-1} = \Delta^{-r-l} \tau^{-r-l}(A'_l) ... \tau^{-r-1}(A'_1)$ avec $A'_i = A_i^{-1} \Delta$ en voyant Δ et A_i en tant que permutations (??)

```
let inverse b =
  let l = List.length b.permlist and q = b.delta_power in
  let delta = P.make_delta b.bpl_size in
  let (_, pl') =
    List.fold_left (fun (parity, acc) p →
                          let p' = P.compose (P.inv p) delta in
                          ((parity + 1) mod 2,
                           (if parity = 0 then p' :: acc else (P.tau p') :: acc)))
                       ((abs q) mod 2, []) b.permlist in
  { bpl_size = b.bpl_size;
    delta_power = − q − l;
    permlist = pl'
  }
```

Conjugué.

Conjugué de A par B : $B^{-1}AB$

```
let conjugate a b = (inverse b) < * > a < * > b
```

7. Convertit une tresse sous forme canonique en sa permutation correspondante. Il s'agit d'un antimorphisme (souvent noté π).

```
let braid2perm b =
    let n = b.bpl_size in
    let perm = (if b.delta_power mod 2 = 0 then
                    P.make_id n
                else
                    P.make_delta n
               ) in
```

```
    let temp = Array.make n 0 in
    List.iter (fun p -> P.compose ~dest:temp p perm; Array.blit temp 0 perm 0 n) b.permlist;
    perm
```

8. Génère une tresse sous forme de liste de permutations aléatoire. À appeler avec l = DOUZE, n de l'ordre de 80 (en gros, $10^{1\ ou\ 2}$)

```
let random_braid_sequence n l =
  let rec loop acc = function
    | 0 -> acc
    | i -> loop (P.random_permutation n :: acc) (i - 1)
  in
  {bpl_size = n; delta_power = l - (Random.int (3 × l)); (* tout à fait arbitraire *)
   permlist = loop [] l}
```

9. Affiche une tresse sous forme de permlist : (delta_power | Permutations)

```
let print_braid_permlist bpl =
  Printf.printf "(%d | " bpl.delta_power;
  List.iter (fun p -> P.print_permutation p; print_string " ") bpl.permlist;
  print_string ")"
```