Feuille d'exercices n°13 - Jeux

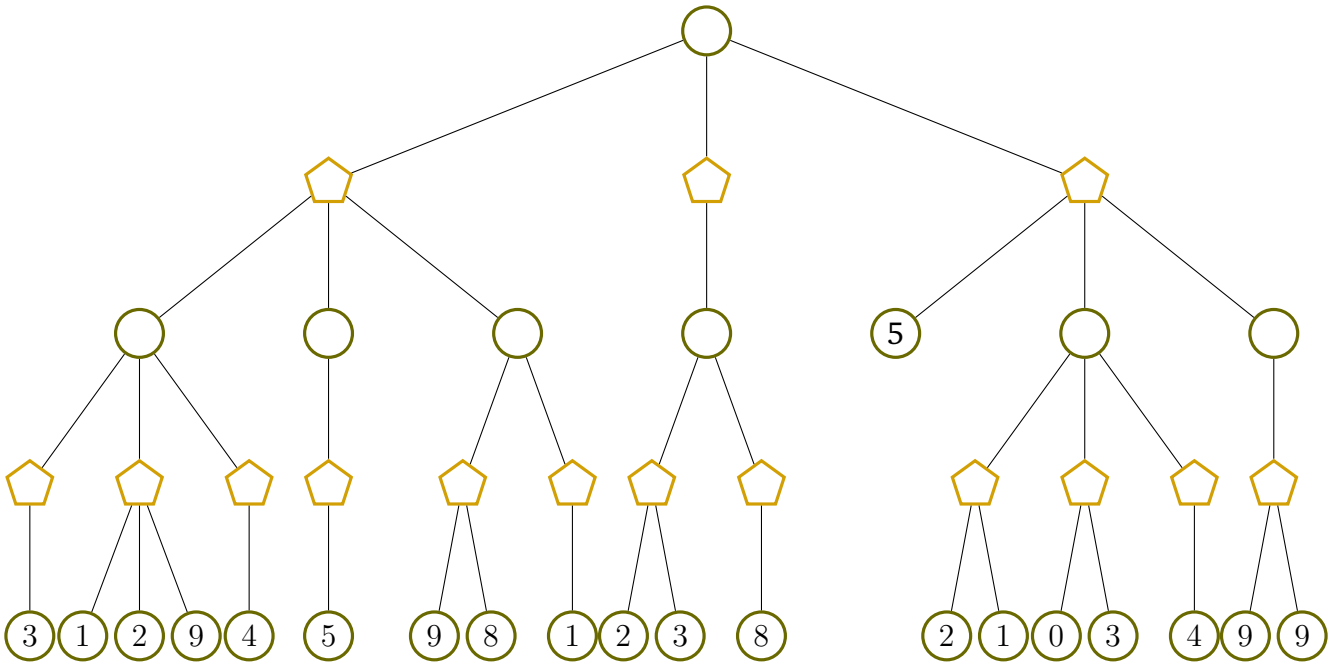
Notions abordées

- jeu de Chomp, jeu des allumettes généralisé
- calcul des attracteurs
- algorithme MinMax (dont une version avec mémoïsation)
- mesurer le temps en OCaml
- manipuler un dictionnaire en OCaml

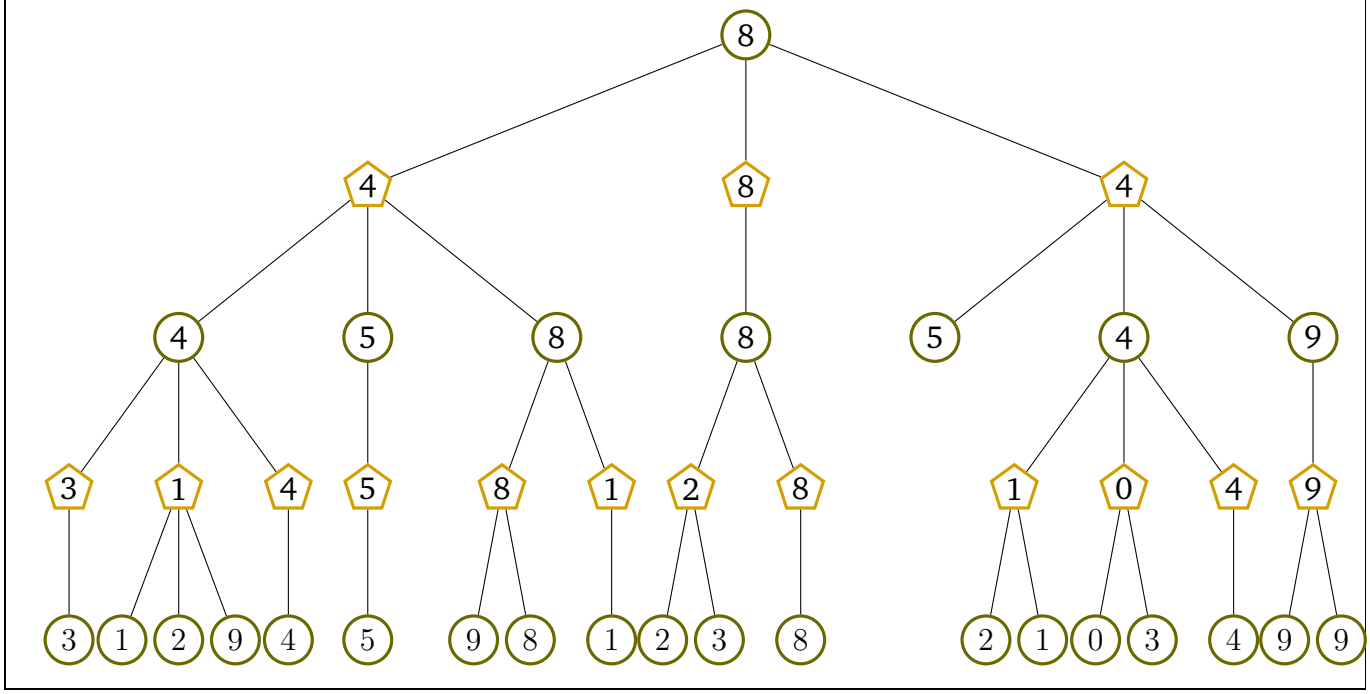
Exercice 1 : Calcul de min-max

On prend comme convention dans cet exercice que les nœuds ronds représentent un état du jeu où c'est à Alice de jouer, tandis que ceux pentagonaux représentent un état où c'est à Bob de jouer. De plus on suppose que les scores sont donnés de sorte qu'Alice cherche à maximiser le score et Bob cherche à le minimiser. Autrement dit un score élevé représente une situation favorable à Alice.

Q. 1 Compléter l'arbre ci-dessous au moyen de l'algorithme du min-max.



Solution



On souhaite fournir une implémentation en OCaml de l'algorithme mis en œuvre dans la question précédente. On définit donc le type suivant pour représenter les arbres de min-max.

```
type joueur = Alice | Bob

type mm_tree =
  | Leaf of joueur * int
  | Node of joueur * mm_tree list (* liste non vide *)
```

On remarque que dans de tels arbres les nœuds internes ne contiennent pas de valeurs. En effet seules les feuilles sont annotées avec une valeur, que l'on peut imaginer être celle fournie par une fonction d'heuristique. On souhaite "compléter" l'arbre, autrement dit associer une valeur à chaque nœud interne de l'arbre. On définit donc le type suivant pour représenter les arbres de min-max complétés.

```
type mm_tree_completed =
  | LeafC of joueur * int
  | NodeC of joueur * int * mm_tree_completed list (* liste non vide *)
```

À titre d'exemple le sous-arbre de hauteur 2 le plus à gauche de l'arbre de la question 1 est représenté au moyen de la valeur OCaml ci-dessous.

```
let ex =
  Node(Alice,[
      Node(Bob,[
          Leaf(Alice, 3)]);
      Node(Bob,[
          Leaf(Alice, 1); Leaf(Alice, 2); Leaf(Alice, 9)]);
      Node(Bob,[
          Leaf(Alice, 4)])])
```

Sa version complétée est quant à elle représentée par la valeur suivante.

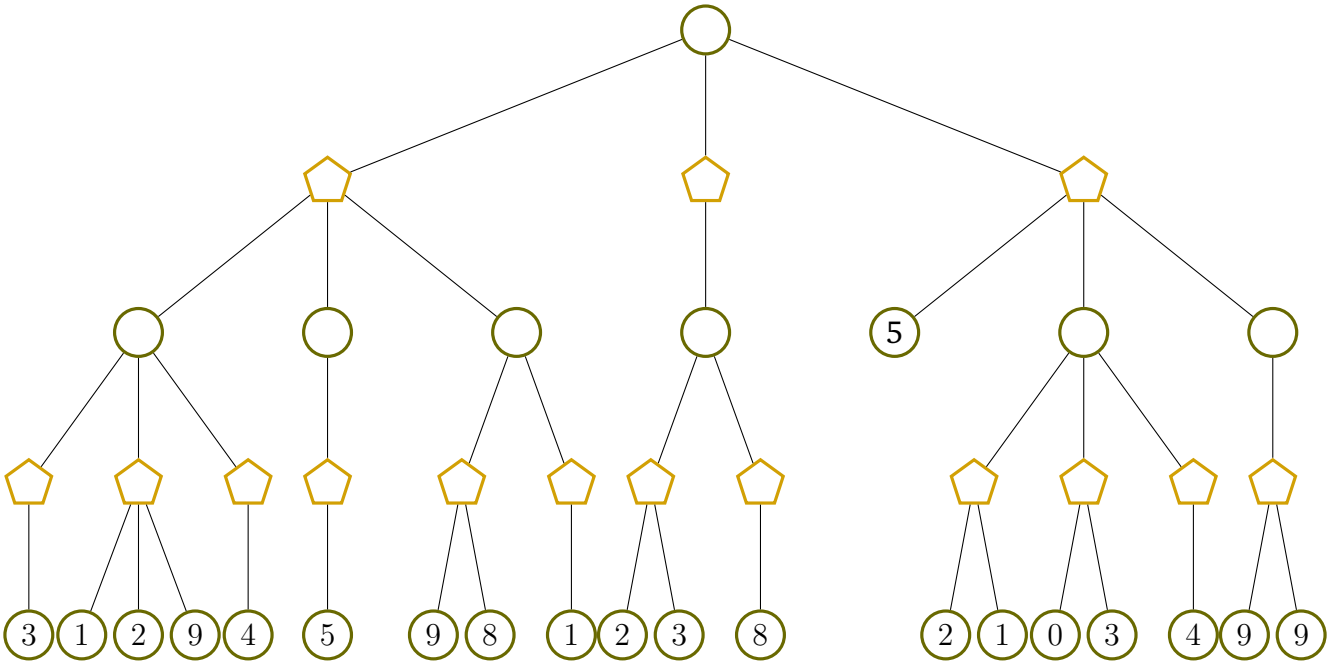
```
let ex_c =
  NodeC(Alice, 4, [
      NodeC(Bob, 3, [
          LeafC(Alice, 3)]);
      NodeC(Bob, 1, [
          LeafC(Alice, 1); LeafC(Alice, 2); LeafC(Alice, 9)]);
      NodeC(Bob, 4, [
          LeafC(Alice, 4)])])
```

Q. 2 Proposer une fonction `complete_mm : joueur -> mm_tree -> mm_tree_completed` prenant en arguments un joueur j et un arbre de min-max, et qui retourne une version complétée de cet arbre lorsque c'est le joueur j qui cherche à maximiser son score.

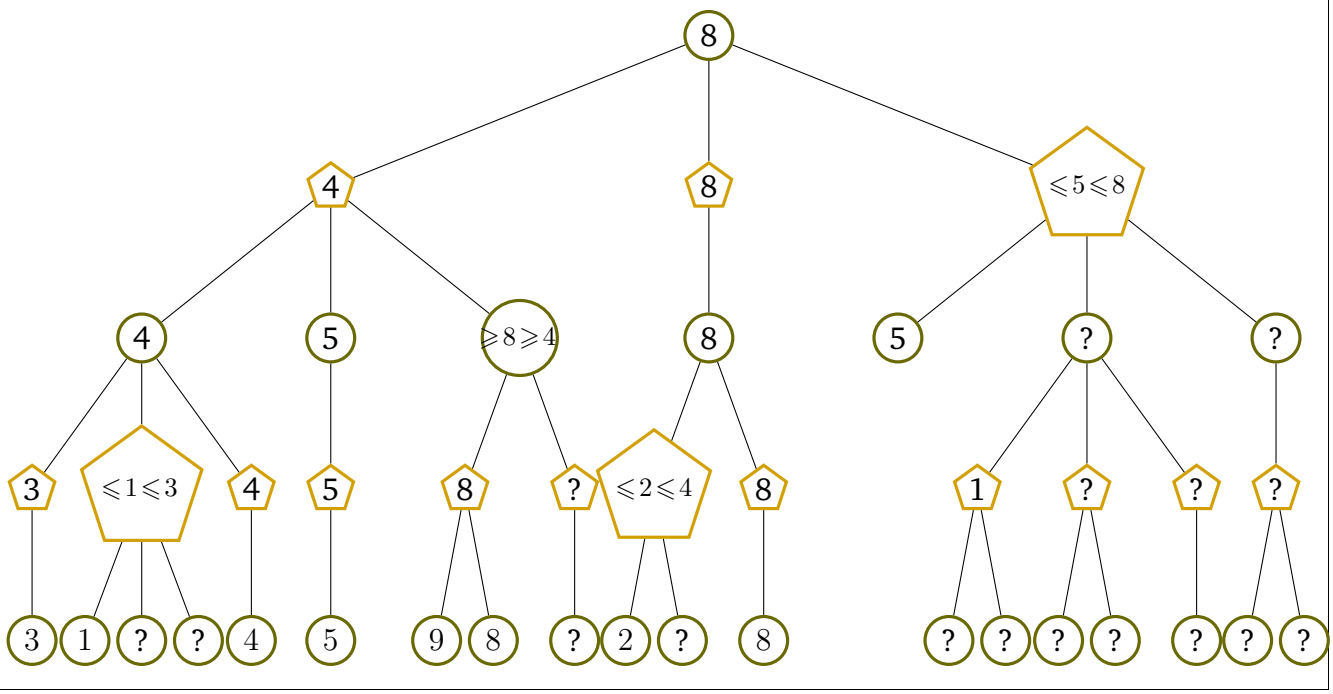
Solution

```
let score (mtc : mm_tree_completed) : int =
  match mtc with
  | LeafC(jj, s) -> s
  | NodeC(jj, s, ls) -> s


let rec complete_arbre (j: joueur) (mt: mm_tree) : mm_tree_completed =
  match mt with
  | Leaf(jj, s)  -> LeafC(jj,s)
  | Node(jj, ls) ->
    let ls_c = complete_liste j ls in
    match ls_c with
    | []             -> failwith "invariant liste non vide"
    | first :: reste ->
      let score = List.fold_left
          (fun acc x -> (if jj=j then max else min) (score x) acc)
          (score first)
          reste
      in NodeC(jj, score, ls_c)

and complete_liste (j: joueur) (l: mm_tree list) : mm_tree_completed list =
  match l with
  | [] -> []
  | t::q -> (complete_arbre j t) :: (complete_liste j q)
```

Q. 3 Compléter l'arbre ci-dessous au moyen de l'algorithme du min-max **avec élagage** $\alpha - \beta$.



Solution



Q. 4 Proposer une fonction `lazy_score_tree : joueur -> mm_tree -> int` prenant en arguments un joueur j et un arbre de min-max et calculant, de manière **paresseuse grâce à l'élagage** $\alpha - \beta$, le score de cet arbre lorsque c'est le joueur j qui cherche à maximiser son score.

Solution

On met d'abord en place des types et petites fonctions pour gérer les bornes inférieures et supérieures.

```
type b_inf =
  | Minfini
  | Binf of int

let val_b_inf (alpha: b_inf) : int =
  match alpha with
  | Minfini -> failwith "max vide"
  | Binf a  -> a

let best_b_inf (s: int) (a: b_inf) : b_inf =
  match a with
  | Minfini -> Binf s
  | Binf aa -> Binf (max aa s)
```

```
type b_sup =
  | Bsup of int
  | Pinfini

let val_b_sup (beta: b_sup) : int =
  match beta with
  | Pinfini -> failwith "min vide"
  | Bsup b  -> b

let best_b_sup (s: int) (b: b_sup) : b_sup =
  match b with
  | Pinfini -> Bsup s
  | Bsup bb -> Bsup (max bb s)
```

On définit ensuite la fonction suivante, qui calcule le score d'un arbre à partir de trois fonctions mutuellement récursives : une qui agit sur les arbres, une (resp. la dernière) qui agit sur les listes de fils d'un nœud où on cherche à maximiser (resp. minimiser) le score.

```
let lazy_score (j: joueur) (mt: mm_tree) : int =
  let rec aux_tree (mt: mm_tree) (alpha:b_inf) (beta:b_sup) : int =
    (* calcule le score exact de mt s'il est ds  [alpha,beta], sinon     *)
    (* si c'est à j de jouer, le score retourné va être pris dans un min *)
    (* on peut se permettre de renvoyer alpha, sinon on renvoie beta     *)
     match mt with
    | Leaf(jj,s) -> s
    | Node(jj,ls) ->
      (if jj = j then aux_list_max else aux_list_min) ls alpha beta
  and aux_list_max (l:mm_tree list) (alpha:b_inf) (beta:b_sup) : int =
    match l with
    | [] -> val_b_inf alpha
    | t::q -> let score_t = aux_tree t alpha beta in
              if (best_b_sup score_t beta) = beta (* score_t >= beta *)
              then val_b_sup beta
              else aux_list_max q (best_b_inf score_t alpha) beta
  and aux_list_min (l:mm_tree list) (alpha:b_inf) (beta:b_sup) : int =
    match l with
    | [] -> val_b_sup beta
    | t::q -> let score_t = aux_tree t alpha beta in
              if (best_b_inf score_t alpha)=alpha (* score_t <= alpha *)
              then val_b_inf alpha
              else aux_list_min q alpha (best_b_sup score_t beta)
  in aux_tree mt Minfini Pinfini
```

En ajoutant à la fonction précédente des instructions d'affichage bien choisies, on obtient l'affichage suivant sur l'arbre ex, on peut vérifier que cela correspond au déroulé mené sur le sous-arbre de hauteur 2 le plus à gauche dans la question 3.

```
ici aux_tree
  ici aux_liste_max
    ici aux_tree
      ici aux_liste_min
        ici aux_tree sur une feuille
      ici aux_liste_min sur liste vide
```

```
ici aux_liste_max
  ici aux_tree
    ici aux_liste_min
      ici aux_tree sur une feuille
ici aux_liste_max
  ici aux_tree
    ici aux_liste_min
      ici aux_tree sur une feuille
    ici aux_liste_min sur liste vide
ici aux_liste_max sur liste vide
```