Lycée Chateaubriand
Informatique MP 2002/03
Mathieu Giraud

TP Caml n° 7 – Corrigé
Exercices et stratégies min-max

1 Quatre exercices

1.1 Exponentiation rapide

a. Le calcul rapide de x^n utilise les deux égalités $x^{2k} = (x^k)^2$ et $x^{2k+1} = x^{2k} \cdot x$.

```
let rec puissance x = function
  | 0 -> 1.
  | n -> if (n mod 2) = 0
         then (puissance x (n/2)) ** 2.
         else (puissance x (n-1)) *. x ;;
```

1.2 Calcul de π

b. Le seul calcul d'une double récurrence sur a_n et b_n ne suffit pas en raison de la somme $\sum_{i=1}^{n} 2^i \left({a_i}^2 - {b_i}^2\right)$. On peut par contre effectuer une triple récurrence simultanée en posant :

$$\pi_n = \frac{4{a_n}^2}{1 - 2S_n} \quad \text{où} \quad \forall n \geq 0 \quad S_n = \sum_{i=1}^{n} 2^i \left({a_i}^2 - {b_i}^2\right)$$

```
let rec calcul_sab n =
  match n with
  | 0. -> 0., 1., 1./.(sqrt 2.)
  | n -> let (s, a, b) = calcul_sab (n -. 1.)
         in let an = ((a +. b) /. 2.)
            and bn = (sqrt (a *. b))
            in let sn =  (s +. 2. ** n *. (an ** 2. -. bn ** 2.))
            in (sn, an, bn) ;;

let salamin n =
  let (s, a, b) = calcul_sab n
  in  4. *. a ** 2. /. (1. -. 2. *. s) ;;

#salamin 4. ;;
- : float = 3.14159265359
```

La triple récurrence permet de faire un algorithme en temps $\mathcal{O}(n)$. Ceci aurait aussi pu se programmer de manière itérative en calculant successivement les a_i, b_i, et S_i, voire en utilisant un tableau pour les conserver.

1.3 Méthode de Newton

c.

```
let epsilon_dér = 1e-6 ;;

let dérivée f x = ((f (x +. epsilon_dér)) -. (f x)) /. epsilon_dér ;;

let rec newton f epsilon x =
  let x1 = x -. (f x) /. (dérivée f x)
  in if (abs_float (x1 -. x)) < epsilon
     then x1
     else newton f epsilon x1 ;;

#newton (fun x -> x *. x -. 2.) 1e-6 1. ;;
- : float = 1.41421356237

#newton (fun x -> cos (x  /. 2.)) 1e-6 1. ;;
- : float = 3.14159265359
```

Vous avez vu au cours du TP que cet exercice demande d'arrêter le calcul non pas au bout de `n` itérations fixées mais lorsqu'une condition est remplie. Nous pouvons donc par cette technique estimer la "limite" d'une suite qui converge. Cette estimation est bien sûr loin d'être parfaite mais utilisable dans de nombreux cas.

1.4 Listes

d. Voici une solution simple en $\mathcal{O}(n^2)$ (qui peut d'ailleurs rendre une liste de sortie avec des éléments en plusieurs exemplaires) :

```
let rec au_moins_deux = function
  | []     -> []
  | n :: l -> let reste = au_moins_deux l in
              if appartient n l
              then n :: reste
              else reste;;
```

Une solution plus rapide, en $\mathcal{O}(n \log n)$, consiste à trier tout d'abord la liste puis a effectuer une recherche linéaire :

```
let rec deux_mais_pas_plus deuxième v = function
  | [] -> []
  | t::q -> if (v != t)
            then (deux_mais_pas_plus true t q)
            else if deuxième
                 then t::(deux_mais_pas_plus false t q)
                 else    (deux_mais_pas_plus false t q) ;;

let au_moins_deux l = match (tri_fusion l) with
  | [] -> []
  | t::q  -> deux_mais_pas_plus true t q ;;
```

2 Problème : stratégies min-max

2.1 Préliminaires

e.

```
let rec trouvemax évalue = function
  | [] -> failwith "trouvemax : liste vide"
  | [p] -> p, (évalue p)
  | t::q -> let et = évalue t
            and (p, ep) = trouvemax évalue q
            in if (et > ep)
               then (t, et)
               else (p, ep) ;;
```

f. On peut bien sûr programmer `trouvemin` en changeant juste de sens l'inégalité `(et > ep)`. Il est plus général de faire un appel du type `(cmp et ep)` où `cmp` est une fonction de comparaison.

```
let rec trouveopt cmp évalue = function
  | [] -> failwith "trouveopt : liste vide"
  | [p] -> p, (évalue p)
  | t::q -> let et = évalue t
            and (p, ep) = trouveopt cmp évalue q
            in if (cmp et ep)
               then (t, et)
               else (p, ep) ;;

let trouvemax = trouveopt (prefix >) ;;
let trouvemin = trouveopt (prefix <) ;;
```

2.2 Stratégie min-max

g. Cette question était une des plus délicates du TP. Considérons l'étape maximisante `maximise pos n` :

- si `n` vaut zéro, on est aux feuilles de l'arbre et on se contente de renvoyer le couple formé de la position courante et de son évaluation;
- sinon on souhaite maximiser (par la fonction `trouvemax`) "le résultat de l'étape suivante minimisante". Sur une position `p`, ce résultat vaut `minimise p (n-1)`, ou, si l'on souhaite garder que le résultat et non le coup qui y mène, `snd (minimise p (n-1))`. On applique donc `trouvemax` sur la fonction `fun p -> snd (minimise p (n-1))` et sur la liste des positions atteignables au coup suivant `coups pos`.

On réalise de même l'étape minimisante, ce qui donne au total :

```
let minmax coups évalue pos n =

  let rec maximise pos = function
    | 0 -> (pos, évalue pos)
    | n -> trouvemax  (fun p -> (snd (minimise p (n-1))))  (coups pos)

  and minimise pos = function
    | 0 -> (pos, évalue pos)
    | n -> trouvemax  (fun p -> (snd (maximise p (n-1))))  (coups pos)

  in maximise pos n ;;
```

La fonction précédente marche correctement, mais s'arrête toujours à une profondeur `n`. On la corrige légèrement pour faire jouer un rôle particulier aux valeurs `+1` et `-1` qui signifient que la partie est terminée et qu'il ne faut pas poursuivre la recherche :

```
let minmax coups évalue pos n =

  let rec maximise pos n =
    let epos = (évalue pos) in match (n, epos) with
      | _, 1.
      | _, -.1.
      | 0, _  -> (pos, epos)
      | _, 1. -> (pos, epos)
      | _, _  -> trouvemax  (fun p -> (snd (minimise p (n-1))))  (coups pos)

  and     minimise pos n =
    let epos = (évalue pos) in match (n, epos) with
      | _, 1.
      | _, -.1.
      | 0, _  -> (pos, epos)
      | _, _  -> trouvemin  (fun p -> (snd (maximise p (n-1))))  (coups pos)

  in maximise pos n ;;
```

Une autre solution serait d'utiliser des "positions puits" `p` telles que `coups p` boucle sur `p`.

2.3 Application : le jeu des allumettes

h. La fonction `coups` donne la "règle du jeu" : les positions décrivent ici le nombre d'allumettes laissées sur la table ainsi que le joueur à qui c'est le tour de jouer.

```
let coups (b, na) = match na with
  | 0 -> []
  | 1 -> [(not b, 0)]
  | 2 -> [(not b, 0); (not b, 1)]
  | _ -> [(not b, na-3); (not b, na-2); (not b, na-1)] ;;
```

i. Pour cet exemple, nous écrivons une fonction `évalue` toute simple, qui se contente de renvoyer `1` ou `-1` lorsque la partie est finie et qui est sinon "indécise" (`0`).

```
let évalue (b, na) = match na with
  | 0 -> if b then (-.1.0) else (1.0)
  | _ -> 0. ;;
```

j.

```
#minmax coups évalue (true, 3) 4 ;;
- : (bool * int) * float = (false, 0), 1.0     (* gagné dans tous les cas *)

#minmax coups évalue (true, 4) 4 ;;
- : (bool * int) * float = (false, 3), -1.0    (* perdu si l'adversaire joue bien *)

#minmax coups évalue (true, 6) 4 ;;
- : (bool * int) * float = (false, 4), 1.0     (* gagné dans tous les cas *)

#minmax coups évalue (true, 7) 4 ;;
- : (bool * int) * float = (false, 4), 1.0     (* gagné dans tous les cas *)

#minmax coups évalue (true, 10) 4 ;;
- : (bool * int) * float = (false, 9), 0.0     (* la profondeur est trop faible *)

#minmax coups évalue (true, 10) 6 ;;
- : (bool * int) * float = (false, 8), 1.0     (* là c'est bon : gagné *)
```

La question sur l'élaguage n'ayant pas été abordé au cours du TP, je vous laisse y réfléchir. N'hésitez pas à m'en parler si besoin est.