



Licence 2 Informatique

Rapport du projet IF01

## **Jeu de Mastermind**

Realise par :

**MEKHILEF Wissame, RETY Martin**

16 avril 2015

## Résumé

# Table des matières

# Table des figures

# Chapitre 1

## Developpement general

### 1.1 Evolution du projet

Nous avons débuté ce projet en réfléchissant aux différentes manières d'aborder la création des combinaison des couleurs. Deux choix étaient possible, créer les combinaisons à partir de type string ou bien de créer nous même un type couleur. Nous avons alors opté pour créer un type somme couleur avec comme constructeur toutes les couleurs proposées dans le sujet.

Après avoir créé ce type couleur nous avons directement commencer à faire la fonction de construction de la liste de combinaison avec redondance, nous avons dans l'idée de créer la liste sans redondance avec cette liste et une fonction de comparaison. Nous avons alors commencé à programmer ces fonctions en binome. Martin s'est occupé de faire la fonction de création de la liste avec redondances et Wissame celle sans redondances, nous avons fait cela tout en gardant un oeil sur le travail de l'autre pour comprendre les fonctions et pouvoir les utiliser.

Ensuite nous nous sommes concentrer sur la réalisation de la fonction d'élagage de la liste grâce aux indications. Nous avons tester plusieurs façons de la réaliser mais la plupart ne permettaient pas la gestion de tous les cas, nous sommes donc partis sur une fonction d'élagage qui compare les indications de l'utilisateur et les indications de la combinaison à tester faite par d'autres fonctions. Pour faire ces fonctions nous avons travailler en binome.

Pour finir, nous avons créer la fonction qui permet de jouer au Mastermind ainsi

que le menu de jeu. Nous avons directement implanter la gestion des exceptions de tricherie et d'erreurs de saisies. Ensuite nous avons souhaiter améliorer la projet en ajoutant un module, contenant toutes les fonctions définies qui sont utilisées dans la fonction jouer. Pour se faire, nous avons réparti le travail, Martin s'est chargé de la fonction jouer et du module, quant à Wissame, il s'est occupé du menu, de la gestion des exceptions et de quelques améliorations.

## 1.2 Outils utilise

Pour faire ce projet nous avons utilisé :

- Emacs avec le mode Tuareg pour programmer et tester nos fonctions
- GitHub pour partager nos fichiers et faire avancer notre code
- Skype pour discuter et travailler sur le projet
- LaTeX pour écrire notre rapport
- Internet pour faire des recherches sur les manières de résolutions d'un mastermind et de la documentation sur Ocaml.

# Chapitre 2

## Analyse des fonctions

```
module ListCouleur :
  sig
    type couleur =
      Rouge
      | Bleu
      | Vert
      | Noir
      | Jaune
      | Orange
      | Violet
      | Blanc
  val listeCouleur : couleur list list
  val construire_ListR : int -> couleur list list -> couleur list list
  val compList : 'a list -> bool
  val construire_ListSR : 'a list -> ('a -> bool) -> 'a list
  val suppression : 'a -> 'a list -> 'a list
  val pions_communs : 'a list -> 'a list -> int
  val pions_bien_places : 'a list -> 'a list -> int
  val indications : 'a list -> 'a list -> int * int
  val elagage : 'a list -> int * int -> 'a list list -> 'a list list
  val print_list : couleur list -> unit
end
```

### 2.1 Construction des listes



```

let rec construire_ListR taille listC=
  let rec construire_aux list=
    match list with
    | [] -> []
    | h::t->(Rouge::h)::(Bleu::h)::(Vert::h)::(Noir::h)::(Jaune::h)::(Orange::h)::(construire_aux list)
  in match taille with
  | 1 -> listC
  | _-> if taille >= 1 then
    construire_aux (construire_ListR (taille - 1) listC)
  else
    failwith "Erreur_taille_trop_petite"

let rec compList list =
  match list with
  | [] -> true
  | h::t-> not(mem h t) && compList t

let rec construire_ListSR list comp =
  let rec aux l res=
    match l with
    | [] -> res
    | h::t-> if not(comp h) then
      aux t res
    else
      aux t (h::res)
  in aux list []

```

### 2.1.1 Création de la liste avec redondance

La fonction `construire_ListR` permet de construire la liste des combinaisons avec redondances, celle-ci possède 2 paramètres :

- `taille`, la taille des combinaisons
- `listC`, la liste des couleurs possibles

Elle possède une fonction auxiliaire `construire_aux` qui prend en paramètre une liste. La fonction principale et auxiliaire sont toutes deux récursives, la récursion de la fonction principale permet l'accumulation des appels de `construire_aux` en fonction de la taille (grâce à l'appel : `construire_aux( construire_ListR (taille-1) listC) ).` Quand à la fonction auxiliaire, celle-ci permet d'ajouter chacune des couleurs possibles sur tous les éléments de la liste placée en paramètre ( `(Rouge : h) :: (Bleu : h) ...`). Grâce





à cette double récursion la création de la liste est très efficace malgré les 32768 cas à produire pour une combinaison de taille 5.

### 2.1.2 Exemple :

```
construire_ListR 1 listeCouleur;;
```

renvoie la listC

```
construire_ListR 2 listeCouleur;;
```

```
renvoie [[Rouge;Rouge];[Bleu;Rouge];[Vert;Rouge];[Noir;Rouge];[Jaune;Rouge];[Orange;Rouge];[V
```

```
construire_ListR 3 listeCouleur;;
```

renvoie la liste construite par constuire\_aux (construire\_aux(construire\_list 1 listeCouleur))

### 2.1.3 Création de la liste sans redondance

Pour créer cette liste nous avons besoin d'une fonction de comparaison compList prenant en paramètre une liste et qui renvoie un booléen :

- Vrai, si l'élément comparé de la liste n'apparaît pas dans le reste de cette liste
- Faux, sinon

Ensuite pour la création de la liste, la fonction prend 2 paramètres :

- list, qui correspond à la liste copmlète des combinaisons possibles
- comp, le comparateur défini juste avant

Cette fonction utilise une fonction auxiliaire aux prenant en paramètre une liste et le résultat des précédentes exécutions. Elle consiste à tester chaque élément de la liste avec le comparateur et ensuite les rajouter ou non dans la liste res.

## 2.2 Elagage

Pour faire cette fonction d'élagage nous avons besoin de plusieurs fonctions :



- suppression, prenant en paramètre un élément  $e$  et une liste  $l$ . cette fonction permet de supprimer la 1ère occurrence de l'élément  $e$  de la liste  $l$
- pions\_communs, prenant en paramètre 2 listes  $l1$  et  $l2$ . Elle permet de renvoyer le nombre de pions communs entre ces 2 listes. Cette fonction utilise suppression pour supprimer les éléments de  $l1$  appartenant aussi à  $l2$ , ensuite renvoie la longueur de la liste  $l2$  moins la longueur de  $l1$  avec application de la fonction auxiliaire
- pions\_biens\_places, prenant 2 listes  $l1$  et  $l2$  en paramètre. Elle compare les éléments de même ordre dans les 2 listes et renvoie le nombre des éléments identiques de ces 2 listes.
- indications, prenant 2 listes  $l1$  et  $l2$ , celle-ci appelle les 2 fonctions précédentes et renvoie le couple  $(bp, mp)$  (sachant qu'il faut réduire  $mp$  avec  $bp$ )

Avec toutes ces fonctions, il suffit de faire la fonction `elagage`, prenant en paramètre `comb` (la combinaison de comparaison), `ind` (les indications de l'utilisateur) et `l` (la liste des combinaisons à réduire). Cette fonction renvoie la liste de combinaison qui respecte le prédicat suivant :  $(indications\ comb\ h) = ind$ , c'est-à-dire que l'élément  $h$  à tester possède le même couple  $(bp, mp)$  que `comb`. Cette fonction est assez efficace quand `bp` ou `mp` est assez élevé mais prend plus de temps quand ceux-ci valent 0 ou 1 (la fonction doit alors inspecter plus en détails la liste).

## 2.3 Fonction joue

**exception** Tricherie

```
val jouer : int -> ListCouleur.ListCouleur.couleur list list -> unit
val menu :
  ListCouleur.ListCouleur.couleur list list ->
  ListCouleur.ListCouleur.couleur list list -> unit
val listeComplete : ListCouleur.ListCouleur.couleur list list
val listeSR : ListCouleur.ListCouleur.couleur list list
val main : unit -> unit
```

```
let jouer nbcoup liste =
  let rec aux n l =
    match (n, l) with
    | _, [] -> raise Tricherie
    | x, [h] ->
      let prop = hd l in
      ListCouleur.print_list prop;
      print_string "est la bonne combinaison !\n";
      print_newline();
      | x, _ when x > nbcoup && nbcoup < 0 ->
```



```

    print_string "Plus_d'essai:_PERDU!\n";
    print_newline();
|x,_->
    let prop=hd l in
    print_string "Essai_";
    print_int x;
    print_string "_:_";
    ListCouleur.print_list prop;
    print_string "\n";
    print_string "Nombre_de_pion(s)_bien_placé(s):\n";
    try
        let bp=read_int() in
        print_string "Nombre_de_pions(s)_mal_placé(s):\n";
        let mp=read_int() in
        print_newline();
        aux (x + 1) (ListCouleur.elagage prop (bp,mp) l)
    with Failure("int_of_string") ->print_string "Erreur_de_saisie!\n"; au

```

La fonction jouer est seulement l'application de diverse fonction, exception ou d'affichage en fonction des paramètre quelle possède, ceux-ci sont le nombre de coup disponible et la liste sur laquelle on travaille. Cette fonction utilise une autre fonction auxiliaire permettant simplement l'ajout du nombre coup n exécuter pour le moment. Les différents cas de la fonction jouer sont :

- `_,[]->` ce cas correspond à une tricherie car la liste n'est vide seulement dans le cas où l'utilisateur rentre de mauvaise indication, l'ordinateur supprime alors tous les cas existants
- `x,[h] ->` affiche la bonne combinaison car il ne reste plus qu'un seul élément dans la liste
- `x,_` quand `x` est supérieur à `nbcoup` et différent de 0 -> celle-ci indique que l'ordinateur a perdu car il n'a plus de coup disponible
- `x,_ ->` le dernier cas qui permet l'affichage de la proposition de l'ordinateur, l'entrée des informations de l'utilisateur (tout en ayant une gestion d'erreur de saisie) et l'application de l'élagage sur la liste avec les indications de l'utilisateur et la dernière proposition de l'ordinateur ( `aux (x+1) (ListCouleur.elagage prop (bm,mp) l)` )

### 2.3.1 Fonction restante

Il reste quelque fonction à définir :



- `print_list`, celle-ci permet simplement l’affichage sur terminal d’une liste de couleur
- `menu`, prenant en paramètre les 2 types de liste de combinaison (avec/sans redondance). Celle-ci permet le choix du mode (`matching`), la gestion des exceptions d’erreur de saisie et de tricherie (`try/with`) et une récursion permettant la rejouabilité du programme
- `main()`, faisant juste l’appel du menu avec les 2 types de liste

## Chapitre 3

# Exemples et Conclusion

### 3.1 Quelques exemples