Projet Mastermind

Licence Informatique 2e année

Université d'Orléans

UE Programmation fonctionnelle

2014-2015

Groupe : Ferre Donovan / Fournier Paul

Chargé de TD : F. Dabrowski / W. Bousdira

Sommaire

On utilisera le code couleur suivant pour améliorer la lisibilité :

Fonction = *fonction*

Paramètre = *paramètre*

Organisation du programme

On pourrait facilement décomposer le programme en 5 grandes parties :

- La partie configuration de la partie : c'est un dialogue avec l'utilisateur pour décider de la version du jeu à laquelle il souhaite jouer , la taille de la combinaisons , ...
- La partie création de la liste de combinaisons : elle crée la liste de toute les combinaisons possible du jeu avant même que l'ordinateur ai fait sa première proposition à l'utilisateur (avec ou sans doublons selon le choix de l'utilisateur)
- La partie déroulement de la partie : cette fois encore c'est un dialogue avec l'utilisateur ; l'ordinateur lui propose une combinaison et l'utilisateur dit en quoi elle diffère ou non de la combinaison qu'il a choisit en début de partie . L'ordinateur traite alors la liste des combinaisons en conséquence (via la partie suivante).
- La partie nettoyage de la liste de combinaisons : grâce aux informations données par l'utilisateur sur la combinaison proposée par l'ordinateur on retire toutes les combinaisons qui sont devenu impossible .
- La partie résolution : après chaque nettoyage de la liste de combinaison l'ordinateur vérifie si celle-ci se retrouve avec uniquement un élément (l'ordinateur a alors deviné la combinaison de l'utilisateur) , si elle est vide (on peux donc en déduire que l'utilisateur triche) et enfin si dans le cas de la limitation de coup de l'ordinateur , ce dernier peux encore joué si ce n'est pas le cas l'utilisateur a gagné sinon on reprend la suite d'opération depuis la partie 3 .

Fonctionnement du programme

On va expliquer le fonctionnement du programme partie par partie :

- La partie configuration de la partie :

En fonction de la version à laquelle on va jouer, on utilise la méthode *configuration_parti* qui prend en paramètre la *taille* de la solution et le booléen *version_2* (true si on joue a la version 2 , false sinon) .

cette méthode appelle ensuite la bonne méthode pour construire la liste des combinaisons.

- La partie création de la liste de combinaisons : on a décidé dans tout les cas de créer la liste avec les combinaisons à doublon et de supprimer les dites combinaisons par la suite .

Pour créer la liste de combinaisons complète on a besoin de trois fonctions : *creer* , *combine* et *creer_combi*.

La méthode principal est la méthode *creer_combi* , c'est elle qui ressort la liste des combinaisons . Elle prend en paramètre la *liste* des couleurs et la *taille* des combinaisons à créer .

Dans le principe elle distingue trois grand cas :

1. Soit on a encore rien créé dans ce cas elle appelle la méthode *creer* .La méthode *creer* prend en paramètre une *liste* à laquelle ajouter les éléments de la deuxième liste passé en paramètre *liste_suite* afin d'obtenir une liste de liste composé de la *liste* a laquelle on a ajouté les différentes couleurs possible de *liste_suite*. Dans notre cas de base on appel donc cette méthode sur la liste vide ([ ]) et avec la liste de couleur nommé *liste* dans *creer_combi* .
2. Soit on a déjà une liste de combinaisons mais d'une taille insuffisante . Dans ce cas on appelle la méthode *combine* . Cette méthode observe le même principe que la méthode *creer* mais au lieu d'avoir une liste en paramètre on a une liste de liste nommé *a_combine* , la liste des couleurs est cette fois nommé *liste* . On applique donc la méthode *creer* a toutes les listes de la liste *a_combine*.
3. Soit la liste de combinaisons est prête, on peux donc la retourner . Ce cas est implicite en réalité on exécute en remontant on fait nos appels récursifs au cas de la taille souhaité , on redescend jusqu'au cas de base de la liste où l'on a rien créé et on remonte en combinant jusqu'à obtenir la liste de combinaison .

Pour retirer les liste contenant des doublons de la liste de combinaisons on utilise la méthode récursive *des_doublons* qui prend en paramètre la *liste* des combinaisons et la *taille* de la solution attendu, elle envoi chaque combinaison a la méthode *suppr_doublons* qui retourne la combinaison seulement si elle ne possède pas de doublons ; pour cela elle prend chaque couleur une par une et regarde grâce a *list.mem* si la couleur n'est pas dans la suite de la combinaison. si la combinaison ne possède pas de

doublons alors on l'ajoute a une liste sinon non , une fois le parcourt de la liste terminé on retourne cette liste.

- La partie déroulement de la partie :

  On utilise la fonction *jouer* qui demande à l'utilisateur si on utilise la version1 (combinaisons sans doublon) ou la version (combinaisons avec doublon) puis si l'ordinateur a un nombre de coup limité ou non. cette fonction appelle une fonction récursive .

  parti qui prend en paramètre *ordi_coups* (le nombre de coup que l'ordinateur peux faire au maximum (1 s'il l utilisateur a choisi de ne pas le faire jouer)) *liste* , la liste de combinaisons encore disponible , *taille* , la taille de la solution et enfin un booléen *ordi* (true si l 'ordinateur a un nombre de coup déterminer par l utilisateur et false sinon)si *ordi* est sur false alors *ordi_coups* sera égale a 1 mais on ne décrément jamais sinon on décrémente a chaque appelle de cette fonction la variable *ordi*. La méthode *parti* demande a l'utilisateur le nombre de pion bien placé et mal placé; elle appelle la méthode

  premiere_descombinaisons qui prend en paramètre *liste* des combinaison et qui renvoi la première des combinaison si la liste n'est pas vide sinon elle renvoi une erreur (tricheur) car toutes les combinaisons auront été supprimé ce qui n est pas possible

- La partie nettoyage de la liste de combinaisons : cette partie du programme est traité avec les méthodes : *supprimer* , *test_possible* , *contient* , *modif* .

Le cœur du nettoyage de la liste est la fonction *supprimer* elle prend 6 paramètres :

1. *combi_propose* qui est la combinaison proposé par l'ordinateur à l'utilisateur
2. *liste_combi* qui est la liste des combinaisons qui reste possible à cet instant du jeu
3. *nb_bienplace* qui est le nombre de pions qui sont bien placé dans la combinaison proposée par l'ordinateur (valeur donnée par l'utilisateur)
4. nb_malplace qui est le nombre de pions de la bonne couleur mais mal placé dans la combinaison proposée par l'ordinateur (valeur donnée par l'utilisateur)
5. *nb_diff* qui est le nombre de pions qui ne devrait pas se trouver dans la combinaison proposée par l'ordinateur (valeur déduite des deux valeurs précédentes)
6. *taille* qui correspond à la taille de la combinaison , la seule utilité de cette valeur est de vérifié que l'utilisateur n'essaye pas de tricher en rentrant des valeurs incohérentes .

L'idée principal de la méthode *supprimer* est de voir combinaison par combinaison si celle-ci est compatible avec les nouvelles informations données par l'utilisateur , pour se faire on appelle la méthode *test_possible* sur chaque combinaison (même paramètre que pour *supprimer* sauf qu'au lieu d'appeler sur la *liste_combi* on l'appelle sur combi_test qui est la combinaison testée) . Si la *test_possible* renvoie false on retire la combinaison de la liste sinon on la garde .

Cette méthode va fonctionner au cas par cas , élément par élément , on a donc beaucoup de conditionnelle , on parcourt la combinaison proposée et la combinaison à tester simultanément. Voici donc les cas à traiter:

1. Le deux éléments sont identiques et différent de "non" (pourquoi "non" ? On y reviendra plus tard) dans ce cas on décrémente le compteur *nb_bienplace* . Si ce compteur est déjà à zéro cela signifie qu'on a déjà trouvé assez d'élément(s) bien placé(s) , la combinaison est donc impossible et on en renvoie donc false (les trois compteurs fonctionnent de la même manière , on se permettra donc des raccourcis par la suite sauf cas particulier).

2. Les deux éléments sont différents , dans ce cas on doit regarder pour les deux pions si on les retrouve dans la suite de la liste de l'autre combinaison . Si on retrouve un pion dans la suite de l'autre combinaisons on le remplace dans la suite de la combinaison par "non" pour indiquer qu'on l'a déjà pris en compte (l'utilisation de "non" est totalement arbitraire on aurait aussi bien pu le remplacer par "déjà pris en compte" mais cela aurai pris plus de temps a écrire) . On fait donc attention en comparant les deux combinaisons que le pion n'est pas un pion bien placé . On peux maintenant décompter ces pions du compteur *nb_malplace* (dans le cas où les deux pions testés sont des pions mal placé on décompte 2 au compteur notre test n'est alors plus compteur=0 mais compteur<2). Pour faire cette opération on utilise la fonction *contient* et *modif* . La fonction *contient* test l'appartenance d'un pion dans la suite de la combinaison de l'autre tout en s'assurant qu'on ai pas le cas d'un pion bien placé , *modif* remplace ce pion trouvé par "non" . La méthode *modif* est donc toujours appelée en réponse à la méthode *contient* .

3. Si enfin les deux pions ne se trouve pas dans la suite de la combinaison de l'autre on décompte alors le bon nombre de pions au compteur *nb_diff* .

Cette méthode est assez lourde il est possible que l'on ai oublié de parler d'un cas . Mais vous avez maintenant tout les outils en main pour comprendre succinctement la méthode en jetant un œil au code source .

- La partie résolution :

  Le but et de montrer à l'utilisateur la première combinaison de la liste des combinaisons pour qu'il puisse rentrer deux valeurs; le nombre de pion bien placé et le nombre de pion mal placé par rapport à ses informations on supprime de la liste des combinaisons toutes les combinaisons qui seront impossible.

Il ne reste plus que trois cas de figure :

1. premier cas, la combinaison proposé par l'ordinateur est la bonne, dans ce cas c est fini le programme s’arrête (le nombre de pion bien placé sera égale a la taille de la solution)

2. deuxième cas, il ne reste qu une seule solution dans ce cas l'ordinateur la donne avec l affirmation que c est la bonne réponse car il ne reste qu'elle.

3. troisième cas, la liste des combinaisons est vide ce qui veut dire que l'utilisateur a un moment ou un autre menti sur le nombre de pion bien placé et/ou le nombre de pion mal placé.

Difficultés rencontrées

On a rencontré les difficultés suivante lors de notre implémentation :

- Lors de la création de la liste de combinaison le fait de devoir avoir une fonction qui gère parfois une liste et d'autre fois une liste de liste , nous a posé de sérieux problème on est donc passé par une représentation plus simple en ajoutant manuellement chaque couleur a chaque liste pour affiner par la suite pour arriver à la fonction finale qui permet au programme de gérer plus facilement l'ajout d'une couleur au jeu .
- Lors du nettoyage de la liste de combinaison le cas du pion mal placé a pris du temps à gérer , le fait de devoir prendre en compte le pion de la liste proposé pour le compte des pions mal placé n'a pas non plus été une évidence au départ.
- Le dialogue avec l'utilisateur , il nous a fallut beaucoup de test avant de comprendre comment utiliser les fonctions de dialogue read_int , print_string .

Répartition du travail

Donovan : première version de la méthode qui génère la liste des combinaison (celle où on créait les combinaisons en rentrant explicitement les couleurs dans la fonction de création);

les méthodes qui se chargent de la suppression des doublons;

les méthodes de dialogue avec l'utilisateur, du lancement de partie et de la configuration de la partie avec la gestion des erreurs de tricherie (liste de combinaison vide).

Paul : version final de la méthode qui génère la liste des solutions;

les méthodes qui s'occupent de supprimer les combinaisons impossible et qui s'occupe également du cas de tricherie (le nombre de pion bien placé + le nombre de pion mal placé + le nombre de pions différents != taille).

Partie test

- Test d'initialisation de la liste des combinaisons en fonction de la version 1 puis 2 (false = sans doublons; true = avec doublons) :

```
# configuration_parti 2 false

- : string list list =
[["noir"; "rouge"]; ["vert"; "rouge"]; ["bleu"; "rouge"]; ["jaune"; "rouge"];
 ["blanc"; "rouge"]; ["orange"; "rouge"]; ["violet"; "rouge"];
 ["rouge"; "noir"]; ["vert"; "noir"]; ["bleu"; "noir"]; ["jaune"; "noir"];
 ["blanc"; "noir"]; ["orange"; "noir"]; ["violet"; "noir"];
 ["rouge"; "vert"]; ["noir"; "vert"]; ["bleu"; "vert"]; ["jaune"; "vert"];
 ["blanc"; "vert"]; ["orange"; "vert"]; ["violet"; "vert"];
 ["rouge"; "bleu"]; ["noir"; "bleu"]; ["vert"; "bleu"]; ["jaune"; "bleu"];
 ["blanc"; "bleu"]; ["orange"; "bleu"]; ["violet"; "bleu"];
 ["rouge"; "jaune"]; ["noir"; "jaune"]; ["vert"; "jaune"]; ["bleu"; "jaune"];
 ["blanc"; "jaune"]; ["orange"; "jaune"]; ["violet"; "jaune"];
 ["rouge"; "blanc"]; ["noir"; "blanc"]; ["vert"; "blanc"]; ["bleu"; "blanc"];
 ["jaune"; "blanc"]; ["orange"; "blanc"]; ["violet"; "blanc"];
 ["rouge"; "orange"]; ["noir"; "orange"]; ["vert"; "orange"];
 ["bleu"; "orange"]; ["jaune"; "orange"]; ["blanc"; "orange"];
 ["violet"; "orange"]; ["rouge"; "violet"]; ["noir"; "violet"];
 ["vert"; "violet"]; ["bleu"; "violet"]; ["jaune"; "violet"];
 ["blanc"; "violet"]; ["orange"; "violet"]]

# configuration_parti 2 true

- : string list list =
[["rouge"; "rouge"]; ["noir"; "rouge"]; ["vert"; "rouge"]; ["bleu"; "rouge"];
 ["jaune"; "rouge"]; ["blanc"; "rouge"]; ["orange"; "rouge"];
 ["violet"; "rouge"]; ["rouge"; "noir"]; ["noir"; "noir"]; ["vert"; "noir"];
 ["bleu"; "noir"]; ["jaune"; "noir"]; ["blanc"; "noir"]; ["orange"; "noir"];
 ["violet"; "noir"]; ["rouge"; "vert"]; ["noir"; "vert"]; ["vert"; "vert"];
 ["bleu"; "vert"]; ["jaune"; "vert"]; ["blanc"; "vert"]; ["orange"; "vert"];
 ["violet"; "vert"]; ["rouge"; "bleu"]; ["noir"; "bleu"]; ["vert"; "bleu"];
 ["bleu"; "bleu"]; ["jaune"; "bleu"]; ["blanc"; "bleu"]; ["orange"; "bleu"];
 ["violet"; "bleu"]; ["rouge"; "jaune"]; ["noir"; "jaune"];
 ["vert"; "jaune"]; ["bleu"; "jaune"]; ["jaune"; "jaune"];
 ["blanc"; "jaune"]; ["orange"; "jaune"]; ["violet"; "jaune"];
 ["rouge"; "blanc"]; ["noir"; "blanc"]; ["vert"; "blanc"]; ["bleu"; "blanc"];
 ["jaune"; "blanc"]; ["blanc"; "blanc"]; ["orange"; "blanc"];
 ["violet"; "blanc"]; ["rouge"; "orange"]; ["noir"; "orange"];
 ["vert"; "orange"]; ["bleu"; "orange"]; ["jaune"; "orange"];
 ["blanc"; "orange"]; ["orange"; "orange"]; ["violet"; "orange"];
 ["rouge"; "violet"]; ["noir"; "violet"]; ["vert"; "violet"];
 ["bleu"; "violet"]; ["jaune"; "violet"]; ["blanc"; "violet"];
 ["orange"; "violet"]; ["violet"; "violet"]]

#
```

- Test de détection des doublons :

```
# des_doublons ["bleu";"vert"]

- : bool = false

# des_doublons ["bleu";"vert";"rouge"]

- : bool = false

# des_doublons ["bleu";"vert";"bleu"]

- : bool = true

# des_doublons["bleu"]

- : bool = false

#
```

- Test première des combinaison et d'une des triche (liste des combinaison vide) :

```
# premiere_descombinaisons [["rouge"];["bleu"]]
- : string list = ["rouge"]
# premiere_descombinaisons [[]]
Exception: Failure "la liste est vide".
# premiere_descombinaisons []
Exception: Failure "la liste est vide".
#
```

- Test de la suppression de combinaisons impossible :

On a créé une liste de combinaisons de deux couleurs avec doublons (pour une question de visibilité) .

```
val combi : string list list =
  [["rouge"; "rouge"]; ["noir"; "rouge"]; ["vert"; "rouge"];
   ["bleu"; "rouge"]; ["jaune"; "rouge"]; ["blanc"; "rouge"];
   ["orange"; "rouge"]; ["violet"; "rouge"]; ["rouge"; "noir"];
   ["noir"; "noir"]; ["vert"; "noir"]; ["bleu"; "noir"]; ["jaune"; "noir"];
   ["blanc"; "noir"]; ["orange"; "noir"]; ["violet"; "noir"];
   ["rouge"; "vert"]; ["noir"; "vert"]; ["vert"; "vert"]; ["bleu"; "vert"];
   ["jaune"; "vert"]; ["blanc"; "vert"]; ["orange"; "vert"];
   ["violet"; "vert"]; ["rouge"; "bleu"]; ["noir"; "bleu"]; ["vert"; "bleu"];
   ["bleu"; "bleu"]; ["jaune"; "bleu"]; ["blanc"; "bleu"];
   ["orange"; "bleu"]; ["violet"; "bleu"]; ["rouge"; "jaune"];
   ["noir"; "jaune"]; ["vert"; "jaune"]; ["bleu"; "jaune"];
   ["jaune"; "jaune"]; ["blanc"; "jaune"]; ["orange"; "jaune"];
   ["violet"; "jaune"]; ["rouge"; "blanc"]; ["noir"; "blanc"];
   ["vert"; "blanc"]; ["bleu"; "blanc"]; ["jaune"; "blanc"];
   ["blanc"; "blanc"]; ["orange"; "blanc"]; ["violet"; "blanc"];
   ["rouge"; "orange"]; ["noir"; "orange"]; ["vert"; "orange"];
   ["bleu"; "orange"]; ["jaune"; "orange"]; ["blanc"; "orange"];
   ["orange"; "orange"]; ["violet"; "orange"]; ["rouge"; "violet"];
   ["noir"; "violet"]; ["vert"; "violet"]; ["bleu"; "violet"];
   ["jaune"; "violet"]; ["blanc"; "violet"]; ["orange"; "violet"];
   ["violet"; "violet"]]
```

Dans ce test on souhaite avoir toutes les combinaisons contenant soit rouge soit bleu mais pas les deux .

```
# supprimer ["rouge";"bleu"] combi 1 0 1 2;;
- : string list list =
[["rouge"; "rouge"]; ["rouge"; "noir"]; ["rouge"; "vert"]; ["noir"; "bleu"];
 ["vert"; "bleu"]; ["bleu"; "bleu"]; ["jaune"; "bleu"]; ["blanc"; "bleu"];
 ["orange"; "bleu"]; ["violet"; "bleu"]; ["rouge"; "jaune"];
 ["rouge"; "blanc"]; ["rouge"; "orange"]; ["rouge"; "violet"]]
```

Dans ce test on a les deux couleurs dans la combinaisons mais elle sont mal placés .

```
# supprimer ["rouge";"bleu"] combi 0 2 0 2;;
- : string list list = [["bleu"; "rouge"]]
#
```

Dans ce test on a les deux couleurs qui ne se trouve pas dans la solution on supprime toutes les combinaisons contenant bleu ou/et rouge .

```
# supprimer ["rouge";"bleu"] combi 0 0 2 2;;
- : string list list =
[["noir"; "noir"]; ["vert"; "noir"]; ["jaune"; "noir"]; ["blanc"; "noir"];
 ["orange"; "noir"]; ["violet"; "noir"]; ["noir"; "vert"]; ["vert"; "vert"];
 ["jaune"; "vert"]; ["blanc"; "vert"]; ["orange"; "vert"];
 ["violet"; "vert"]; ["noir"; "jaune"]; ["vert"; "jaune"];
 ["jaune"; "jaune"]; ["blanc"; "jaune"]; ["orange"; "jaune"];
 ["violet"; "jaune"]; ["noir"; "blanc"]; ["vert"; "blanc"];
 ["jaune"; "blanc"]; ["blanc"; "blanc"]; ["orange"; "blanc"];
 ["violet"; "blanc"]; ["noir"; "orange"]; ["vert"; "orange"];
 ["jaune"; "orange"]; ["blanc"; "orange"]; ["orange"; "orange"];
 ["violet"; "orange"]; ["noir"; "violet"]; ["vert"; "violet"];
 ["jaune"; "violet"]; ["blanc"; "violet"]; ["orange"; "violet"];
 ["violet"; "violet"]]
```

Dans ce test on regarde ce qui se passe si l'utilisateur essaye de tricher en rentrant un nombre de pions supérieur à la taille de la solution.

```
# supprimer ["rouge";"bleu"] combi 2 1 0 2;;
Exception: Failure "tricheur !".
```