TP - 2

24 septembre 2014

Aujourd'hui on commence par corriger le TP1. On apprend à écrire un makefile et à exploiter la compilation séparée. On commençera ensuite à coder un petit jeu.

1 Correction TP1

Vous trouverez les 3 fichiers de solutions à l'adresse suivante. Relevez la tête pour écouter la correction :)

Ce qu'il faut retenir, comprendre et appliquer par la suite :

— le code doit être enregistré dans un fichier et édité hors terminal, vous devez le **compiler** et **exécuter** l'exécutable. N'utilisez pas le top level pour autre chose que vous aider sur un type, usage de fonction, ... ;

— le code doit être correctement **indenté** (n'oubliez pas d'appuyer sur TAB pour demander à emacs d'indenter la ligne courante) avec des **retours à la ligne bien placés** pour améliorer la lisibilité (cf. solutions). Le code doit être suffisament **commenté** : un lecteur doit comprendre le rôle de chaque sous fonction et des variables que vous définissez ;

— ayez le réflexe d'écrire une fonction assez**générique** qui répond au problème sans rien afficher puis une fonction de test qui appelle votre première fonction et qui affiche le résultat. Exemple : `fact` ou `parcours_largeur` ne doivent rien afficher. Il faut également essayer de ne pas limiter vos fonctions à des usages trop précis. Un bon exemple est la fonction sur le parcours en largeur. Ce parcours doit pouvoir être utilisé avec n'importe quelle fonction de traitement aux noeuds de type `label -> unit`. Ceci permet de rendre votre code plus **modulaire**. ;

— il faut essayer de découper une grosse fonction en de plus **petites fonctions atomiques** qui ont chacune un rôle disinct. De cette façon, on gagne en lisibilité, modularité et on facilite le débogage. Exemple : la fonction de lexing de l'exo 4 doit faire beaucoup de choses comme des traitements sur la chaînes de caractères puis un parcours sur la chaîne, etc.. Vous pouvez écrire tout ça avec 4 petites fonctions dont on comprend bien le fonctionnement. Cf. correction ;

— évitez de trop mélanger le style impératif et fonctionnel et préférez le **style fonctionnel** quand c'est possible. Pour plusieurs d'entrez vous, vos références peuvent êtres remplacés par des `let ... in ...` ;

— la **librairie standard** contient un grands nombres de modules qui peuvent vous êtres utiles. Une fois que vous avez décidé d'une structure de données (liste, tableau, file, pile, ...), consultez la page correspondante dans la doc de la libraire standard pour connaître les fonctions du module, leur type et leur usage. Des noms en vrac : pour afficher : `Printf`, listes : `List`, tableaux et matrices : `Array`, pile : `Stack`, files : `Queue`, tables de hachages : `Hashtbl`...

2 Compilation séparée

Dans la section 4, nous commencerons à coder le jeu puissance 4. Pour un tel programme, il faut écrire du code pour décrire le plateau de jeu, les règles régissant le jeu, l'afichage d'un plateau et des jetons, une session interactive permettant de jouer, une intelligence artificielle, etc..

On ne va pas écrire tout ça dans un même fichier pour plusieurs raisons. Par exemple, par la suite, on pourrait vouloir changer la fonction d'affichage (pour ajouter des couleurs par exemple) sans toucher au reste du code. Si le code de l'affichage est mélangé au reste on va avoir du mal à isoler les fonctions à modifier. Il faudrait aussi pouvoir recompiler uniquement le code correspondant à l'affichage puisque le reste n'a pas changé. D'autre part, on voudrait pouvoir réutiliser le code d'affichage pour des versions différentes du jeu (humain contre humain et humain contre ordinateur par exemple). Ces exemples montrent que l'on a besoin d'un code modulaire. C'est à dire un code organisé autour de modules [1] ayant des rôles et une interface bien définis que l'on peut charger et compiler séparémment. Par exemple, `List`, `Array`, ... sont des modules. On peut ainsi très bien changer le code de `List.iter` sans avoir à recompiler TOUS les programmes utilisant cette fonction.

En pratique Tout cela est simple à mettre en oeuvre. Dans un dossier, chaque fichier `.ml`, une fois compilé, peut être chargé comme un module dans un autre source. Pour compiler un fichier qui utilise un module, il faut ajouter ce fichier source en argument à la compilation (ex. `$ ocamlopt monModule.ml monProg.ml`).

Exemple Reprenons l'exemple du puissance 4 : dans un fichier `regles.ml` nous définissons les structure de données représentant les joueurs, jetons et plateau de jeu. Nous y définissons également quelques fonctions utilitaires implémentant les règles du jeu. Dans un fichier `affiche.ml` nous chargeons le module `Regles` en écrivant `open Regles` (attention à la majuscule) et nous définissons les fonctions permettant d'afficher le plateau de jeu. Un autre fichier `puissance4.ml` charge `Affiche` et `Regles` puis définit la fonction principale implémentant la session interactive permettant de jouer. Enfin, un dernier ficheir `tests.ml` charge tous les fichiers et contient tous les tests des fonctions.

Pour compiler, il suffit de donner à `ocamlopt` tous les fichiers à la suite dans l'ordre de dépendance (`regles.ml`, `affiche.ml` puis `puissance4.ml`) : `$ ocamlopt -o puissance4.o regles.ml affiche.ml puissance4.ml`.

3 Makefile

Pour le moment vous avez dû taper des commandes du type `$ ocamlopt -o exe.out monSource.ml` pour compiler vos programmes. Mais comme vous l'avez vu dans la section précédente, avec plusieurs fichiers ça commence à être plus compliqué. De plus, on pas forcément envie de tout recompiler dans le cas où l'on change juste un fichier.

Un *Makefile* est un fichier décrivant comment construire un programme à partir de sources. L'exécutable `make` lit ce fichier et construit le programme. Il permet de décrire dans quelle ordre et comment compiler des sources, `make` s'occupe ensuite de détecter quelles fichiers ont été modifiés et doivent être recompilés. C'est un outil très utile que l'on utilise partout (pas seulement pour OCaml bien sûr, on l'utilisera par exemple aussi pour compiler du C).

En pratique Vous pouvez décrire une série de *cibles* avec pour chacune d'elle les commandes à exécuter pour la construire. La syntaxe est la suivante (attention aux tabulations) :

```
nom_cible : composant_1 composant_2 ... composant_n
<<tabulation>>  commande_1
...
<<tabulation>>  commande_l
```

En exécutant `$ make nom_cible`, `make` exécute automatiquement les commandes `commande_1`, ...`commande_l`. De plus, si un composant est aussi une cible du Makefile alors il commence par exécuter cette cible.

1. Les modules constituent un trait essentiel d'OCaml mais le cours de Prog1 n'a pas pour vocation de les étudier dans le détail. Mais sachez que vous les verrez lors du cours de Prog2.

Exemple Pour notre puissance 4 nous allons constuire le Makefile suivant :

```
regles: regles.ml
<<tabulation>>  ocamlc -c regles.ml

affiche: regles.ml affiche.ml
<<tabulation>>  ocamlc -c regles.ml affiche.ml

all: regles.ml affiche.ml puissance4.ml
<<tabulation>>  ocamlc -o puissance4.out regles.ml affiche.ml puissance4.ml

test: regles.ml affiche.ml puissance4.ml tests.ml
<<tabulation>>  ocamlc -o tests.out regles.ml affiche.ml puissance4.ml tests.ml

clean:
<<tabulation>>  rm -f *.cm[iox] *~ .*~ #*#
<<tabulation>>  rm -f *.o
```

La règle clean permet de nettoyer les fichiers générés par la compilation pour ne laisser que les sources. Les autres règles permettent de compiler un fichier particulier pour voir si le fichier compile bien. La règle all permet de construire le programme `puissance4.out` que l'on peut ensuite exécuter. La règle test compile le fichier `tests.ml` et produit l'exécutable `tests.out`.

4 Problème

Comme vous vous en doutez, on va maintenant écrire le code pour les fichiers décrits dans le paragraphe 2. Vous avez sans doute déjà tous joué au puissance 4. Pour information, la grille fait 7 colonnes et 6 lignes et chaque joueur a 25 jetons.

Exercice - 1 *Puissance 4*

Dans un premier temps, l'objectif est de terminer le jeu Humain contre Humain. On verra plus tard comment implémenter une intelligence artificielle pour pouvoir jouer contre l'ordinateur.

Le jeu doit se dérouler dans une fenêtre graphique (en utilisant le module `Graphics`) et être complétement interactif (via la fenêtre graphique et si possible la souris). L'affichage doit représenter la grille, les jetons (de couleurs) et l'état du jeu (à qui de jouer ? gagnant ?). Le programme doit évidemment détecter la victoire.

1- Dans un nouveau dossier, récupérez l'archive à l'adresse suivante et décompressez là avec cette commande : `$ cd votre_dossier;`
`wget http://www.lsv.ens-cachan.fr/~hirschi/enseignements/prog1/TP2/puis4/puis4.tar;`
`tar -xvf puis4.tar`.
2- Si vous n'avez plus bien en tête le rôle de chaque fichier, relisez les deux sections précédentes. Les `(* TODO (type: ...) *)` indiquent une fonction que vous devrez compléter dans les questions suivantes. Vous ne devez pas changer les noms des fonctions existantes. Vous devez respectez les indications de type. Vous êtes encouragés à créer d'autres fonctions auxiliaires pour mieux organiser votre code.
3- Achevez les codes des fonctions dans `regles.ml`. Vous avez le droit (et vous y êtes encouragés) d'écrire des fonctions auxiliaires entre ces fonctions. Ecrivez des tests dans `tests.ml` pour ces fonctions. Testez (`$ make test; ./tests.out`).
4- Achevez d'écrire les fonctions du fichier `affichage.ml` en utilisant des fonctions de la librairie Graphics. Vous avez le droit (et vous y êtes encouragés) d'écrire des fonctions auxiliaires entre ces fonctions. Testez.
5- Ecrivez le jeu interactif dand `puissance4.ml`. Vous avez le droit (et vous y êtes encouragés) d'écrire des fonctions auxiliaires entre ces fonctions. Le code de `getCoup` est déjà écrit. Il utilise des *exceptions*. Ce n'est pas grave si vous ne savez pas ce que c'est[2] , lisez simplement le test correspondant dans `tests.ml` pour voir comment exploiter cette fonction. Jouez ;)

2. Si vous êtes curieux et en avance, lisez ceci.