TP - 1

18 septembre 2014

On commence avec les outils de base que certains connaissent sans doute. L'objectif est d'arriver à mettre en place un environnement de travail pour commencer à programmer dans de bonnes conditions. On s'amusera ensuite avec OCaml. Pour ceux qui n'ont jamais programmé en Ocaml, voici quelques références : un tutoriel et un résumé de la syntaxe. On pourra regarder ça ensemble si vous avez du mal avec les exercices.

1 Shell Unix

La plupart des programmes dont nous aurons besoin n'ont pas d'interface graphique, nous utiliserons donc le shell Unix qui nous servira d'interface avec la machine.

Commandes importantes Ouvrez un terminal et entrez ces commandes (lien ou lien en français vers une liste de commandes et leur description) :

1. `$ pwd`
2. `$ ls`
3. `$ cd ..; pwd`
4. `$ cd ~`
5. `$ mkdir TP_Prog`
6. `$ cd TP_Prog`
7. `$ touch fichire;ls`
8. `$ mv fichierr fichier`
9. `$ cp fichier fichier2`
10. `$ rm hello.ml`

Vous apprendrez à utiliser le Shell sur le tas mais sachez que Bash (le shell que nous utilisons) est un vrai petit langage de programmation (langage de script). Les ressources ne manquent pas sur internet (quelques liens : description en profondeur [en], les commandes et la programmation [fr]). Si vous voulez apprendre à utiliser ces commandes ou tout autre programme, vous pouvez accéder à son "mode d'emploi" en entrant `$ man cmd` (pour le programme "cmd").

Signaux Vous pouvez arrêter, mettre en pause ou reprendre des programmes que vous avez lancés depuis le shell avec les raccouris respectifs : C-c, C-z, fg [% job] (où % job est le numéro attribué au programme mise en pause).

2 Emacs

Si vous maitrisez un éditeur de texte puissant et adapté à la programmation (coloration syntaxique, indentation du code, recherche, remplacement, etc.) vous pouvez passer cette section. Sinon, je vous propose d'apprendre à utiliser Emacs. On lance emacs comme ceci : `$ emacs hello.ml&`. On accède aux fonctionnalités habituelles via des raccourcis (C pour contrôle) :

- C-x C-f pour ouvrir un fichier existant ou nouveau ;
- C-x C-s pour sauver le fichier courant ;
- C-x C-b pour changer de "buffer" (fichier ouvert) ;
- C-x C-c pour quitter Emacs ;
- selection a la souris pour copier, clic du milieu pour coller (ou au clavier) ;
- C-h t pour lancer le tutoriel d'Emacs.

Vous pouvez garder ce cheatsheet sous la main pour mémoriser les raccourcis. Consultez la page des tuteurs pour plus de détails sur Emacs.

3 Au travail

Maintenant que l'on sait naviguer dans le système et éditer des fichiers, on va pouvoir programmer. On va commencer par quelques rappels d'OCaml (et sa compilation).

Exercice - 1 *Compilation*

1- Pour vous montrer comment compiler un fichier source OCaml, écrivez simplement un Hello World dans le fichier `hello.ml`[1].
2- Vérifiez que vous êtes dans le dossier du fichier et entrez dans le shell la commande `$ ocamlopt -o hello.out hello.ml`. Vérifiez que le fichier `hello.out` a bien été créé : c'est l'exécutable généré par la compilation de `hello.ml`.
3- Lancez cette exécutable : `$ ./hello.out`.

C'est de **cette façon** que l'on exécutera nos programmes. On commence par créer un fichier source écrit dans un langage compréhensible pour l'homme (comme `hello.ml`). On compile ce fichier ; autrement dit, on traduit ce programme dans un langage compréhensible pour la machine (essayez de lire `hello.out` en faisant `$ cat hello.out`). Puis on lance le programme. Si vous voulez tester des fonctions rapidement, vous pouvez toujours utiliser l'interpréteur Ocaml (entrez `$ ledit ocaml`).

Exercice - 2 *Remise en forme*

1- Ecrivez un programme qui calcule puis affiche le 25ème terme de la série de Hofstadter Conway et **testez** :
— $a(1) = a(2) = 1$;
— $a(n) = a(a(n-1)) + a(n - a(n-1))$ pour $n \geq 2$.
2- Le même code permet-il de calculer le 1489ème terme[2] ? Modifiez (si besoin) le programme que vous avez écrit afin qu'il puisse calculer ce terme.
3- Si vous avez utilisé `let rec`, réécrivez le programme sans. Si vous avez utilisé `while` ou `for`, réécrivez le programme sans. **Testez**.

• Un bonne pratique que vous devez adopter dès maintenant consiste à séparer le code qui répond au problème du code qui permet de tester ce que vous avez écrit. Il faut également penser à commenter son code (pour soi et pour les autres). Par exemple, pour les questions précédentes, j'attend au moins les fonctions suivantes : `conwaRec` (muni d'un commentaire du type "calcule le i-ème terme de la suite de Conway en style récursif") `conwayImp` (je vous laisse inférer le commentaire) et `testConway` (qui teste les fonctions conway avec du printf).

Exercice - 3 *Polymorphisme*

1- a- Rappelez vous de la notion de curryfication et écrivez une fonction d'addition de type `int` $\to$ (`int` $\to$ `int`).
b- Ecrivez une fonction de "curryfication" pour les fonctions à deux arguments de type (('a $*'$ b) $\to'$ c) $\to'$ a $\to'$ b $\to'$ c.
c- Ecrivez une fonction de "de-curryfication" pour les fonctions à deux arguments de type ('a $\to'$ b $\to'$ c) $\to$ ('a $*'$ b) $\to'$ c.

1. Conseil : vous pouvez trouver la documentation de la librairie standard ici et y chercher `printf`.
2. Faites C-c si vous perdez le contrôle.

2- a- Ecrivez le type d'un arbre d'arité non fixée dont les noeuds sont étiquetés par une valeur de type polymorphe.
b- Ecrivez une fonction qui réalise le parcours en profondeur d'un tel arbre en appliquant une fonction abstraite de type `'a` $\rightarrow$ `unit` où `'a` est le type des valeurs aux noeuds (pensez à aller voir la doc de `List`).
c- (*) Même chose avec un parcours en largeur.
d- (*) Même chose sans utiliser de modules de la librairie standard (excepté Printf).

Exercice - 4 *Amusons-nous avec Pascal*

D'après un exo de David Baelde

1- Écrire une fonction `next : int list -> int list` qui à une ligne du triangle de pascal associe la suivante.
a- Á l'aide de la fonction précédemment définie, affichez le triangle de Pascal, représentant les valeurs paires par un `' '` et les impaires par un `'#'`.

Vous utiliserez les fonctions suivantes :

```
mod : int -> int -> int
print_char : char -> unit
print_newline : unit -> unit
```

b- Affichez ainsi les 31 premières lignes du triangle. Si vous ne reconnaissez pas la figure obtenue, essayez de diminuer la police, plisser les yeux ou afficher plus de lignes.
2- Si ce n'est pas déjà fait, écrivez ces mêmes fonctions en utilisant des listes mais aucune référence, boucle for ou while.
3- Quel type utiliseriez vous pour réécrire la fonction *next* spécialisée pour le calcul des parités ?

Exercice - 5 *Encore plus rigolo : une calculette à l'ancienne*

L'objectif de cet exercice est de réaliser un parseur d'expressions mathématiques écrites en notation polonaise inversée. Vous devez prendre en compte les opérations $+$, $-$, $*$, $-_1$ (i.e. opposé) sur les entiers. A partir d'une expression mathématique donnée sous la forme d'une liste d'entiers et d'opérateurs, vous devez calculer le résultat de l'expression.
1- Ecrivez le type `lexem` des léxèmes (ce sont les éléments de la liste représentant l'expression).
2- a- De quelle structure de données avons-nous besoin pour calculer de telles expressions ?
b- Implémentez cette structure de données de façon polymorphe.
c- Ecrivez la fonction `parse_polonaise` de type `lexem list` $\rightarrow$ `int` qui calcule le résultat.
d- (*) Ecrivez la fonction de lexing (i.e. calcul de la liste de lexèmes à partir d'une chaîne de caractères) de type `string` $\rightarrow$ `lexem list`.
e- (*) Ecrivez une calculette en ligne de commande en notation polonaise inversée.

Pour mercredi prochain <12H: Envoyez-moi vos solutions aux exercices 2, 3 et 4 par mail. Un fichier par exercice de la forme `prenom-nom_TP1_numExo.ml` ou un fichier pour tous les exercices de la forme : `prenom-nom_TP1.ml`. Vos fichiers doivent : compiler sans erreur, être correctement et suffisamment commentés et contenir des tests qui affichent les résultats à l'exécution.