Programmation - TP1: Introduction

Simon Halfon

September 16, 2015

Ce document est basé sur les TP rédigés par Lucca Hirschi.

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, vous trouverez un tutoriel sur http://mirror.ocamlcore.org/ocaml-tutorial.org/. Vous pouvez aussi vous reportez à la fiche **Ocaml Cheatsheet**. Si vous découvrez Ocaml, je vous conseille de faire les exercices de cette fiche. Pour toute questions, vous pouvez me contacter à l'adresse: simon.halfon@lsv.fr

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. Référez vous à la fiche **Unix Shell** pour une rapide description du système de fichier Unix.

Commandes importantes Ouvrez un terminal et testez ces commandes

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 trouverez une liste des commandes et leur description à l'adresse http://ss64.com/bash/, ou en français http://fr.wikibooks.org/wiki/Programmation_Bash/Commandes_shell, ou encore en utilisant la commande `$ man cmd` ("cmd" désigne le nom de la commande dont vous voulez la description).

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). Vous apprendrez à l'utiliser plus en détail dans le cours de Systèmes et architectures.

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: http://punchcard.files.wordpress.com/2010/10/emacs2.png

3 Environnement de 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`

Aide: pour l'affichage, utilisez la fonction Printf.printf. Nous verrons plus tard les librairies Ocaml plus en détail.

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. Pour tester des fonctions rapidement, on peut lancer l'interpréteur Ocaml (Top Level) dans le terminal avec la commande `$ ocaml` (*cf* exerercice suivant).

Remarques sur la compilation: la commande `$ ocamlopt` appelle le compilateur "optimisé" de ocaml. On peut utiliser `$ ocamlc` à la place.

Exercice - 2 *Top Level*

Pour lancer le Top Level, tapez `$ ocaml` dans le terminal. Le Top Level d'Ocaml est particulièrement pénible à utiliser: remarquez que vous ne pouvez pas revenir sur ce que vous avez écrit sans l'effacer. Quelques commandes utiles:

- `#Quit` pour quitter (ou C-c)
- `#use "monfichier.ml"` pour charger un fichier ml
- `#load "lib.cma"` pour charger une librairie (*e.g.* graphics)

1- Essayez ces commandes (ce n'est pas fondamental, passez cet exercice si vous êtes en retard. Vous aurez les quelques commandes utiles si nécessaire plus tard)

Exercice - 3 *Tuareg-mode dans Emacs*

Pour pallier à l'effroyable inefficacité de l'interpréteur Ocaml, vous pouvez utiliser le mode Tuareg sous Emacs (ceux qui sont fidèles à un autre éditeur devront compiler à la main dans le terminal). C'est très pratique pour les petits exercices que nous feront en TP, cela permet de tester rapidement vos programmes. C'est déconseillé pour le projet.

Sur les machines de la 411, Tuareg se lance tout seul lorsque vous ouvrez un fichier `.ml` dans Emacs (C-x C-f). Les commandes propres à Tuareg dont vous aurez besoin sont les suivantes:

- C-c C-c: compile le fichier ;;)
- C-x C-e: evalue la phrase (*i.e.* la région entre deux
- C-c C-e: idem
- C-c C-b: evalue tout le fichier
- C-c C-q: indente la phrase
- C-M-\: indente la région

Vous trouverez tous les raccourcis à l'adresse ocamlpro.com/files/tuareg-mode.pdf.

Exercice - 4 *Librairies Ocaml*

Certains d'entre vous ont peut être appris en CPGE le langage "Caml-Light". Comme son nom l'indique, il s'agit d'une version allégée de Ocaml. Malheureusement, ce langage vous donne quelques mauvaises habitudes:

- Les librairies `List` et `Array` y sont "ouvertes" par défaut
- Les fonctions de la librairie `Array` sont renommées avec le préfixe "vect" et en utilisant une syntaxe qui n'existe pas en Ocaml: par exemple `make_vect`

Dans Ocaml, il n'y a pas de fonction `hd`. Par contre, il y a une librairie `List` qui contient une fonction `hd`. Pour l'utiliser dans votre code, vous pouvez:

- Soit utiliser `List.hd`
- Soit ajouter la ligne `open List` au début de votre fichier, puis utiliser `hd`.

Une documentation des librairies Ocaml (fonctions de la librairie et leur description) est disponible sur le site de l'INRIA. En pratique, tapez "Ocaml Lib" dans un moteur de recherche (Lib = nom de la librairie). Vous serez ammenés à utiliser les librairies suivantes: `List`, `Array`, `Printf`, `Scanf`, `String`, ... Vous pouvez aussi consulter la librairie `Pervasives` qui contient les fonctions de bases d'Ocaml (elle n'a pas besoin d'être ouverte !).
1- Ecrivez une fonction qui transforme une liste en un tableau.
2- Si ce n'est pas déjà le cas, réécrivez la sans boucle `for` ou `while`.
3- Ecrivez une fonction qui transforme un tableau en liste.
4- Si ce n'est pas déjà le cas, réécrivez la sans boucle `for` ou `while`.
5- Ecrivez une fonction

```
make_matrix: int -> int -> int array array
```

qui crée une matrice de tailles données en entrée remplie de 0 (interdiction d'utiliser `Array.make_matrix`).
6- Créez une matrice de taille 4×4 et mettez à jour la case $(2, 1)$. Afficher la matrice.

4 Exercices

Exercice - 5 *Remise en forme*

1- Ecrivez un programme qui calcule puis affiche le 25ème terme de la série de Hofstadter Conway (http://mathworld.wolfram.com/Hofstadter-Conway10000-DollarSequence.html) 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 ?
Aide: faites C-c si vous perdez le contrôle.
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**.

Exercice - 6 *Amusons-nous avec Pascal*

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 - 7 *Les exceptions*

1- Sans utiliser la librairie Array, écrivez une fonction qui cherche si un élément est dans un tableau.
2- Réécrivez votre fonction sans utiliser de récursion, ni de boucle `while`. La fonction ne doit pas pour autant parcourir inutilement le tableau (si vous ne voyez pas comment faire, regardez le titre de l'exercice).
3- Modifiez votre fonction pour qu'elle renvoie l'indice de première occurence de l'élément dans le tableau.
4- Implémentez une fonction de recherche dans un arbre dont les noeuds ne sont pas étiquetés et les feuilles sont étiquetées par des entiers.
5- Réécrire cette fonction en utilisant une exception.
6- En utilisant la fonction `time` suivante, comparez les temps d'exécution des deux fonctions sur arbres binaires complets de grande profondeur dont toutes les feuilles sont étiquetées par 0. Qu'observez vous ? Comment l'expliquez vous ?

```
let time f x =
  let t = Unix.gettimeofday () in
  let res = f x in
  Printf.printf "Execution time: %fs\n" (Unix.gettimeofday () -. t ;
  res ;;
```

Exercice - 8 *Polymorphisme*

1- 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 -> unit` où `'a` est le type des valeurs aux noeuds.
c- Même chose avec un parcours en largeur (vous pouvez utiliser la librairie `Queue`).
d- Réécrivez la fonction précédente sans utilisez la librairie `Queue`.
2- a- Ecrire deux fonctions:

```
f: 'a * 'b -> 'b * 'a
g: 'b * 'a -> 'a * 'b
```

telles que $f \circ g$ et $g \circ f$ soient l'identité. On dit que les types `'a * 'b` et `'b * 'a` sont isomorphes. **b-c-** Montrer que les types

```
('a * 'b) -> 'c
'a -> ('b -> 'c)
```

sont isomorphes, en écrivant les fonctions associées. C'est ce que l'on appelle la *curryfication*.
d- Les types `'a` et `'a * 'a` sont-ils isomorphes ?
e- Et les types `int` et `int * int` ?

5 Compilation séparée

Exercice - 9 *Réalisation d'une calculatrice à l'ancienne*

Dans cet exercice, nous abordons le problème du *parsing*: un compilateur reçoit en argument un fichier, qui n'est autre qu'une grosse chaine de caractère. La première étape est donc de retrouver la structure *arborescente* du programme à l'origine. Vous ne connaissez pas encore les outils théoriques utilisés pour faire du *parsing*, vous verrez cela au cours de Langages Formels au second semestre (et oui, les automates que vous avez vu en prépa, en fait, ça sert à quelque chose !). On va donc travailler avec des expressions très simples, et faire du parsing "à la main". Vous allez réaliser un parseur d'expressions mathématiques écrites en *notation polonaise inverse* (https://fr.wikipedia.org/wiki/Notation_polonaise_inverse). Il faut prendre en compte les opérations entières `+`, `-`, `*`, `/`.

Cet exercice est aussi l'occasion de parler de la compilation séparée. Pour un projet de taille importante, il est fondamental d'organiser son code. Il convient notamment de séparer le code en plusieurs fichier `.ml`. Cela présente plusieurs avantages:

- On peut ne recompiler qu'une partie du projet
- Un module peut être réutilisé
- Clarté du code

Il faut alors "lier" les fichiers. C'est la même chose que pour une librairie (type `List`): on ajoute la ligne `open Module` en début de fichier (ou on écrit `Module.f` à chaque utilisation d'une fonction `f` du module). Pour compiler le tout, il suffit de mettre les fichiers `.ml` dans le bon ordre.
1- Récupérer les sources de la calculatrice (demandez moi la clé USB) et utilisez la commande `$ ocamlc -o Calc graphics.cma parser.ml calculator.ml interface.ml` pour compiler le tout.

Imaginons que le fichier `parser.ml` ne soit pas de vous, et que vous soyez sûr de ne pas le modifier. Il est alors inutile de le recompiler à chaque fois. Vous pouvez le compiler une seule fois indépendément avec `$ ocamlc -o parser.ml` puis utiliser le *fichier objet* (.cmo) généré avec `$ ocamlc -o Calc graphics.cma parser.cmo calculator.ml interface.ml`. C'est la même chose pour les librairies Ocaml: dans ce cas vous ne disposez même pas du fichier `.ml` à recompiler, vous ne pouvez que lui fournir le fichier objet. Cette fois ci cependant, l'extension est `.cma` pour indiquer au compilateur d'aller chercher le fichier dans les librairies standards et pas dans le dossier courant. Tous ces aspects module sont l'une des force d'Ocaml (*e.g.* les signatures, les foncteurs, etc). Vous les verrez en détail dans le cours de programmation 2.

Vous vous dites sans doute que ça commence à être pénible de taper des lignes aussi longue pour compiler, et notre petit projet ne contient que trois fichiers ! Pour cette raison il existe un outil particulièrement pratique: le `Makefile`. Ce n'est pas l'objet de ce TP de comprendre en détail comment cela fonctionne, pour l'instant, retenez simplement que pour compiler il vous suffit de taper `$ make`, et cela crée (ou met à jour) l'exécutable du projet, que j'ai nommé `Calc`. Pour lancer la calculette, tapez donc `$ ./Calc`.
2- Faites le, et jouez un peu avec la calculette. Vous allez vite vous rendre compte de ce qui n'est pas codé.

Les fichiers à compléter sont `parser.ml` et `calculator.ml`, mais il se pourrait que vous ayez besoin d'aller consulter `interface.ml` pour savoir comment doivent se comporter les fonctions `parse` et `compute` (notamment vis à vis des exceptions).

3- écrire le type `arith` qui vous permettra de représenter les expressions arithmétiques décrites ci dessus
4- écrire la fonction `parse`, qui produit la représentation de type `arith` d'une expression arithmétique sous forme de chaine de caractère, ou produit l'exception `Parsing_error` si la chaine reçue n'est pas correcte. Il suffit pour cela de créer une pile puis de parcourir la chaine de gauche à droite, si l'on voit un nombre on l'empile, si l'on voit un opérateur, on dépile deux expressions de la pile et on empile l'expression obtenue en appliquant l'opérateur aux deux expressions dépilées.
5- écrire la fonction (triviale) qui prend une expression arithmétique et calcule sa valeur entière. Notez que l'on aurait pu court-circuiter le type `arith`, il est très simple de modifier la fonction `parse` pour qu'elle effectue ce calcul directement.