Informatique, TP 1
Découverte d'OCaml

Thomas Blanc

11 octobre 2012

1 La suite de Fibonacci

1.1 Définition

Soit $(F_n)_{n\geq 0}$ La suite définie comme suit :

$$F_0 = 0$$
$$F_1 = 1$$
$$\forall n > 1; F_n = F_{n-1} + F_{n-2}$$

On veut écrire une fonction *fib : int -> int* telle que *fib n* renvoie F_n.

1.2 Fonction naïve et fonction linéaire

Écrire une fonction récursive *fib_naive* qui reprend directement la formule de F_n, quelle est sa complexité ?

Améliorer l'algorithme de façon à ce que la fonction *fib* ne soit jamais appelée deux fois sur le même indice. Quelle est la nouvelle complexité ?

Est-il possible de rendre l'algorithme récursif terminal ? Si oui, écrire l'algorithme. Si non, expliquer pourquoi.

1.3 Fonctions encore plus rapide

Un ami mathématicien m'a montré que $\forall n \in \mathbb{N}$:

$$F_{2n} = (2F_{n-1} + F_n)F_n$$
$$F_{2n+1} = F_{n+1}^2 + F_n^2$$

Écrire alors un algorithme logarithmique qui calcule fibonacci. Peut-on le rendre récursif terminal ? Si oui, écrire l'algorithme. Si non, expliquer pourquoi.

2 Résolution d'un Sudoku

2.1 Règles

On veut résoudre le jeu du Sudoku à l'aide d'OCaml. Pour rappel, on a une grande grille de 9x9 découpée en 9 carrés de 3x3. Il faut mettre chacuns des chiffres (sauf 0) dans chaque ligne, chaque colonne et chaque carré.

2.2 Définition des types

Écrire la définition suivante dans l'interpréteur :

```
type chiffre = Un | Deux | Trois | Quatre
| Cinq | Six | Sept | Huit | Neuf
and case = chiffre option
and grille = case array array
```

Écrire alors une fonction *creer_grille : (int*int*chiffre) list -> grille* qui crée une grille avec une liste de chiffres initiaux.

Écrire ensuite une fonction *chiffres_autorises : grille -> int -> int -> chiffre list* qui renvoie la liste des chiffres autorisés sur une case. Attention, une case déjà remplie n'autorise aucun autre chiffre que le sien !

Écrire de même une fonction *valide : grille -> bool* qui vérifie qu'une grille pleine n'est pas en contradiction avec les règles.

2.3 Résolution

Écrire une fonction *resoudre_brute_force : grille -> unit* qui résout le sudoku en testant toutes les possibilités. Attention, vous aurez besoin de sauvegarder les modifications faîtes sur la grille pour pouvoir revenir en arrière.

Donner des idées d'améliorations pour l'algorithme, si possible les mettre en œuvre.

3 Compiler un fichier OCaml

3.1 Le cas de base

Avec l'éditeur de texte, créons un fichier **test.ml**, qui serait, disons à la racine de votre dossier personnel. Ce fichier va contenir les commandes qu'autrement nous aurions rentré dans le programme **ocaml** (qu'on appelle toplevel).

Maintenant, nous allons écrire la ligne suivante dans ce fichier :

```
print_endline "Hello world !";;
```

Il n'y a plus qu'à sauver et ouvrir un terminal, en vérifiant bien que nous sommes dans le dossier où test.ml a été enregistré. Si ce n'est pas le cas, les commandes **ls cd** et **pwd** peuvent être utiles. Il suffit alors de compiler en utilisant **ocamlc** (le "c" veut dire compilateur). Cette opération crée (entre autres) un fichier a.out qui exécutera ce qu'on veut :

```
ocamlc test.ml
./a.out
```

3.2 Invoquer une bibliothèque extérieure

La fonction *print_endline* fait partie de la bibliothèque standard, du cœur du langage. Si nous voulons par exemple utiliser des threads, nous aurons besoin d'utiliser des fonctions qui ne sont pas accessibles par défaut. Pour cela, nous allons appeler une bibliothèque tierce : thread.

Reprenons déjà notre fichier de la semaine dernière, appelons le **test2.ml** :

```
let f n = for i = 1 to n do print_char 'A' done;;
let g n = for i = 1 to n do print_char 'B' done;;

let n = 100;;

let ta = Thread.create f n;;
let tb = Thread.create g n;;

Thread.join ta;;
Thread.join tb;;
print_newline ();;
```

On remarque que pour appeler les fonctions non définies de bases, il faut les précéder du nom du *module* que l'on veut appeler. Nous devons maintenant expliquer au compilateur que la bibliothèque de threads est nécessaire. Pour cela, il y a trois choses à faire :

1. Prévenir le compilateur que ce programme utilisera les threads, avec l'option **-thread**.
2. Indiquer au compilateur où trouver les fonctions de threads, qui sont dans le fichier **threads.cma**.
3. Enfin, la bibliothèque threads a elle même besoin de la bibliothèque unix pour fonctionner. On ajoute donc **unix.cma**.

```
ocamlc -thread unix.cma threads.cma test2.ml
./a.out
```

Il est important de bien mettre **unix.cma** avant **threads.cma** et **test2.ml** à la fin !

3.3 Compiler plusieurs fichiers ensemble

Dans beaucoup de gros projet, mettre toutes les définitions dans le même fichier devient vite ingérable. On voudrait donc pouvoir définir plusieurs fichiers, et de réunir leur code à la compilation.

Par exemple, nous allons essayer de séparer le code précédent entre les définitions de f et g et le programme en soit. Nous allons donc définir deux fichiers **a.ml** et **b.ml** comme suit :

```
(* a.ml *)
let f n = for i = 1 to n do print_char 'A' done;;
let g n = for i = 1 to n do print_char 'B' done;;

let n = 42;;

(* b.ml *)
let ta = Thread.create f n;;
let tb = Thread.create g n;;

Thread.join ta;;
Thread.join tb;;
print_newline ();;
```

Nous avons donc ici deux fichiers a.ml et b.ml qui vont chacun définir un module. Nous pouvons maintenant les compiler ensemble. Comme c'est b.ml qui va appeler des fonctions de a.ml, on doit le mettre après a.ml sur la ligne de commande.

```
ocamlc -thread unix.cma threads.cma a.ml b.ml
```

Bon, d'accord, ça ne marche pas. Mais c'est déjà un bon début. On sait que chaque fichier définit un module, et donc f ne fait plus partie du module de b.ml. Il faut expliciter l'appel extérieur, exactement comme avec *Thread.create*. Pour cela, rien de plus simple : le fichier **azerty.ml** donne une fois compilé le module *Azerty*. En suivant les mêmes règles, nous avons maintenant un module *A* qui contient ce que l'on désire.

Voilà donc le nouveau code de b.ml :

```
let ta = Thread.create A.f A.n;;
let tb = Thread.create A.g A.n;;

Thread.join ta;;
Thread.join tb;;
print_newline ();;
```

Cela fait beaucoup de "A." et de "Thread.", et comme tout bon informaticien cherche à en faire un minimum, on va dire à OCaml "si tu ne trouves pas cette fonction, va chercher dans A ou dans Thread, elle y est sûrement". Pour cela, on va utiliser la directive *open* :

```
open Thread;;
open A;;

let ta = create f n;;
let tb = create g n;;

join ta;;
join tb;;
print_newline ();;
```

Ce nouveau b.ml effectue exactement le même calcul, mais il faut admettre qu'il est beaucoup plus lisible n'est-ce pas ? Attention toutefois, abuser de *open* peut parfois engendrer des confusions sur quelle fonction on appelle. C'est pour cela qu'il vaut mieux utiliser *Nom_de_module.nom_de_fonction* si l'on veut éviter des bugs difficiles à détecter.

3.4 Ocamlbuild, l'outil magique de compilation

Ocamlbuild est un outil assez magique qui va grandement simplifier la compilation :

- Il suffit de lui indiquer une fois que l'on va utiliser la bibliothèque thread.
- Il va lui même chercher a.ml lorsque l'on essaye de compiler b.ml
- Il ne recompile pas deux fois un fichier qui n'a pas été modifié.

Il faut d'abord expliquer à ocamlbuild que l'on veut utiliser la bibliothèque de threads. Pour cela on va écrire une ligne pour renseigner ocamlbuild dans le fichier **_tags**, pour cela, nous allons écrire dans le terminal :

```
echo '<b.*>: use_unix, thread' >_tags
```

Ensuite, il n'y a plus qu'à appeler ocamlbuild. Le programme prend simplement en argument le nom du fichier en remplaçant le ".ml" par le type de fichier que l'on veut. Ici, nous voulons obtenir du bytecode et nous allons donc taper :

```
ocamlbuild b.byte
./b.byte
```

Il est à noter que cette fois-ci, le programme créé ne s'appelle pas a.out. Ocamlbuild choisit de nommer le programme de façon plus intuitive pour faciliter la compréhension.

4 Aller plus loin avec les modules

4.1 Écrire des interfaces en .mli

Lorsque l'on écrit un module, on a pas forcément besoin que toutes les fonctions qu'il fabrique ne soient accessibles depuis l'extérieur. De plus, on aimerait s'assurer que le module décrit correspond à un moule nommé interface.

Les informations d'interface sont les informations de typage que nous donne le toplevel. On peut aussi en obtenir grâce à la commande "ocamlc -i" :

```
ocamlc -thread -i unix.cma threads.cma test2.ml
```

On peut voir de cette façon toutes les valeurs définies par **test2.ml** avec leur type.

Essayons maintenant d'enregistrer l'interface de a.ml :

```
ocamlc -i a.ml >a.mli
```

Exercice Faire diverses modifications à a.mli et expliquer leurs impacts. (rappel : les commentaires en OCaml sont entre des *(* *)*)

4.2 Des modules dans les modules

Créons un fichier **test_modules.ml** et écrivons le texte suivant :

```
module C = struct
let f () = print_endline "module C"
end

let f () = print_endline "module Test_module"

module D = struct
let f () = print_endline "module D"
 module E = struct
 let f () = print_endline "module E"
 end
end
```

Au passage, il n'y a aucun double point virgule. Ces doubles points virgules sont optionnels dès que l'on effectue une assignation avec *let* ou avec *module* (non valable dans le toplevel). L'intérieur d'une déclaration de module est du code OCaml qui n'a pas le droit de contenir des doubles points virgules.

Exercice Faire appel à toutes les fonctions f définies dans Test_modules.

4.3 Les foncteurs et l'outil Set

Les foncteurs sont des sortes de fonctions qui prennent en argument un module et qui renvoient un autre module. Par exemple, nous allons voir le foncteur *Set.Make* de la librairie standard.

```
module Int = struct
type t = int
let compare = compare
(* on utilise la fonction de comparaison de la lib standard *)
end

module Ints = Set.Make(Int)
```

On a alors créé un module *Ints* dont l'interface est la même que celle qu'on peut trouver à l'adresse **http ://caml.inria.fr/pub/docs/manual-ocaml/libref/Set.S.html**.

Exercice À partir de ce module, créer une fonction qui prend en entrée une liste d'entiers et renvoie la même liste triée.

5 La bibliothèque OCaml

5.1 La bibliothèque standard

La bibliothèque standard peut toujours être appelée, sans devoir rien spécifier au compilateur. Elle contient les opérations pour manipuler les structures de données les plus basiques et quelques outils qui servent toujours.

5.1.1 Array et List

Pas besoin de revenir en détails sur ces deux modules, il faut se souvenir que les listes sont plus souples à l'usage que les tableaux. À noter : les opérations *map*, *iter* et *fold_left* qui permettent de parcourir ces données très facilement.

5.1.2 Buffer, Char, Printf, Scanf, String

Ces modules servent de boîte à outils pour tout ce qui concernent les chaînes de caractères :

Buffer : Ce module produit des strings optimisées pour la concaténation. Peut être utile de temps en temps.

Char : Pour manipuler directement un caractère. Indispensable dans certains cas très précis, inutile sinon.

Printf : Permet d'imprimer dans la sortie standard ou dans une string de façon très souple. Hyper-utile.

Scanf : Permet de lire très facilement. Très utile si l'on sait s'en servir.

String : Des tas de fonctions utiles sur les chaînes de caractères !

Printf et *Scanf* utilisent des "chaînes préformatées" qui sont des chaînes de caractères auxquelles on ajoute des arguments introduits par %. Par exemple, le code suivant :

```
Printf.printf
 "Entier: %d, Flotant: %f, Chaine %s"
  42
  13.37
  "Hello world";;
```

Remarquez que les fonctions de *Printf* et *Scanf* ont des types bizarres. Oui, c'est comme ça. Ce type très bizarre leur permet de prendre un nombre variable d'arguments.

Exercice Écrire une fonction qui demande à quelqu'un son âge et qui affiche le double du nombre indiqué.

5.1.3 Hashtbl, Queue, Stack, Set, Map

Ces modules procurent d'autres types de données conteneurs efficaces.

Hashtbl : Une sorte de tableau de taille modifiable tout restant extrêmement rapide.

Queue : Une file de type FIFO.

Stack : Une pile de type LIFO.

Set : Un foncteur pour manipuler des ensembles.

Map : Un foncteur pour manipuler des dictionnaires.

Hashtbl, *Queue* et *Stack* sont des outils impératifs : ils fonctionnent par effet de bord.
Dans l'ensemble, *Map* est un peu moins efficace que *Hashtbl* mais reste moins lourd et plus souple d'utilisation.

5.1.4 Filename, Random, Sys

Ici, on a quelques autres outils utiles.

Filename : Jouer avec les noms de fichiers, leurs extensions. Sert rarement.

Random : Fabriquer des nombres aléatoires.

Sys : Utiliser les informations système de base. Très important.

Exercice Écrire un programme qui prend en argument sur la ligne de commande un nombre n, tire un dé à n faces et imprime le résultat dans un fichier temporaire dont le nom termine par ".output".

D'autres modules peuvent être utiles dans la bibliothèque standard. Attention cependant à ne pas utiliser le module *Digest* dont l'algorithme n'est pas assez sécurisé.

5.2 La bibliothèque unix

Cette bibliothèque est une espèce de couteau suisse de l'interaction avec le système.

5.2.1 Environnement

Une variable très importante nommée variable d'environnement permet d'obtenir diverses informations comme le type de terminal, la langue de l'utilisateur où encore la localisation de divers outils. On peut accéder aux différentes valeurs avec *getenv* et les modifier avec *putenv*.

Exercice Écrire un programme qui détecte la langue de l'utilisateur, et selon cette langue affiche bonjour dans toutes les langues que vous connaissez.

5.2.2 Exécution

Cette partie de la librairie contient *fork* et les fonctions permettant d'appeler d'autres programmes. On peut attendre la fin d'un autre processus.

Exercice Écrire un programme qui appelle **ls**, écrit "au revoir" et quitte.

5.2.3 Lire et écrire

Inutile d'utiliser cette partie là : entre les *channel* définit de base, *Printf* et *Scanf*, on a pas besoin de ces fonctions de bas niveau. Il reste toujours utile de savoir que cela existe, c'est à la base ces outils qui sont en fait utilisés.

À noter un peu plus bas l'opération *select* qui peut faire le café, sauver des vies et renverser des gouvernements.

5.2.4 Opération sur les fichiers et sur les dossiers

Avec ces fonctions, on a les opérations de bases pour supprimer, déplacer, changer les droits... En général, un simple script shell est bien plus rapide mais c'est toujours utile de pouvoir gérer ça directement avec votre programme.

5.2.5 Signaux et temps

Avec ces fonctions on peut commander l'exécution des programmes avec des signaux. Un programme peut être mis en pause, redémarré. On peut aussi mettre en place un chronomètre, gérer des dates, etc.

5.2.6 Internet

Tout ce qui est indispensable à la gestion d'internet peut se trouver ici. À retenir les trois fonctions de haut niveau *open_connection*, *shutdown_connection* et *establish_server* qui font tout ce qui est fait au dessus en plus simple.

5.3 La bibliothèque threads

Comme on l'a vu plus haut, la bibliothèque des threads permet de faire du calcul en parallèle. Nous allons maintenant voir toutes les options que nous avons à disposition.

5.3.1 Créer et gérer des threads

On crée donc le thread avec *create*. On attend donc la fin d'un thread avec *join*.

On peut ensuite demander aux threads de patienter avant de lire sur un descripteur de fichier par exemple, grâce à l'instruction *select* notamment.

5.3.2 Les Mutex, les Condition et les Events

Ces trois outils permettent de coordonner les threads et d'éviter qu'ils ne se marchent sur les pieds.

Mutex : Définit des «verrous» qui permettent d'éviter d'accéder deux fois en même temps à la même variable.

Condition : Permet de demander aux threads d'attendre puis de repartir à un certain momment.

Event : Permet des communications un peu plus complexes entre threads.

Exercice Écrire un programme composé de deux threads, l'un ne pouvant écrire que A, l'autre ne pouvant écrire que B, et qui écrit ABABABABABA-BABA... autant de fois que désiré.