TP4: langage C

Lucca Hirschi

5 et 6 octobre 2015

L'objectif de ce TP est de programmer quelques fonctions simples en C pour vous familiariser avec sa syntaxe. Vous aurez besoin des points de syntaxes donnés dans le cours de Jean Goubault. Je vous met également un fichier à disposition ici (et ici en html) qui contient de petits exemples de programmes C utilisant les constructions `if`, `for`, `while`, `switch` ainsi que des tableaux statiques.

Dans ce TP, vous n'avez pas et vous ne devez pas utiliser de *pointeurs* (ou du moins en déclarer un). On laisse ça pour la semaine prochaine.

Exercice - 1 *Compilation*

Comme pour le dernier TP, on commence par apprendre à compiler et exécuter un simple Hello World. Il n'y a pas de Top Level pour le C alors cette fois on est obligé de passer par là. C'est un argument supplémentaire pour que vous compreniez qu'il faut *vraiment* écrire vos programmes OCaml dans un source pour le compiler ensuite !

1- Ouvrez un fichier `hello.c` et recopiez ce programme :

```
#include <stdio.h>

int main () {
  printf("Coucou\n");
  return 0;
}
```

2- Dans un terminal, entrez `$ gcc -o hello.out hello.c` pour compiler votre programme vers l'exécutable `hello.out`.

3- Lancez l'exécutable avec `$ ./hello.out`.

Exercice - 2 *On commence doucement*

1- Ecrivez une fonction qui calcule le maximum de 3 entiers. Testez.

2- Ecrivez une fonction qui calcule le reste modulaire de deux entiers sans utiliser l'opérateur $\%$. Test...

3- Ecrivez une fonction qui calcule le maximum d'un nombre quelconque d'entiers donnés sous la fome d'un tableau d'entiers. Son *prototype* doit être :

```
int max_t (int tab[], int taille)
```

- Le type `int tab[]` dénote un argument `tab` correspondant à un tableau d'entiers.

4- Testez cette fonction en l'appelant sur les arguments de l'exécutable (ex. `$ ./exo2.out 10 -18 14` doit renvoyer 14).

- Le prototype typique d'une fonction main est : `void main (int argc, char *argv[])`. L'argument `argc` (pour *argument count*) contient le nombre de paramètres passés à l'exécutable (ex. avec `$ ./a.out param1 param2`). L'argument `argv` (pour *argument values*) est un tableau contenant les `argc` paramètres sous la forme de chaînes de caractères. Vous pouvez transformer cette chaîne en entier avec `atoi()` (pour *ascii to int*). Attention, l'exécutable prend toujours un premier paramètre implicite correspondant au nom de l'exécutable.

Exercice - 3 *Cribles*

1- En utilisant le crible d'Erastostène, affichez tous les nombres premiers plus petits que 1000.

2- (*) Affichez un diagramme (en bâton ?) représentant la proportion de nombres premiers par tranches de 1000 entre 2 et 10000.

Exercice - 4 *Bitmap*

Dans cet exercice, on va exploiter les *bitmap* pour résoudre quelques problèmes simples. Un *bitmap* est une structure de données permettant de stocker de façon très compacte des tableaux de booléens. L'idée est simplement de représenter un tableau de bits par un entier dont la représentation binaire correspond au tableau. L'objectif est de recoder quelques fonctions de base pour les bitmap. Vos fonctions doivent être très efficaces (la plupart du temps en un cycle CPU). Conseil : pensez aux opérateurs bits à bits.

1- Ecrivez une fonction qui change la valeur d'une case d'un bitmap.

2- Ecrivez une fonction qui affecte un certain bit pour une certaine position dans un bitmap.

3- Ecrivez une fonction qui concatène deux bitmap.

4- Ecrivez une fonction qui transforme un tableau de bool en un bitmap et inversement.

5- Reecrivez le crible d'Erastostène avec un bitmap.

6- Mettez en valeur le gain en temps de cette version. A votre avis, d'où vient ce gain ?

Exercice - 5 *Automates cellulaires* *(d'après David Baelde)*

On considère des automates cellulaires uni-dimensionnels à états booléens. Concrètement, on s'intéresse à un système de "cellules" organisé selon une ligne bi-infinie orientée, où chaque cellule a un voisin à gauche et un voisin à droite, et est dans un état pris dans $\{0, 1\}$. L'évolution du système est synchrone et locale : à chaque étape, toutes les cellules changent d'état en même temps, en fonction de son propre état et de ceux de leurs deux voisins immédiats. Si s, s_g et s_d sont respectivement les états d'une cellule, de sa voisine de gauche et de celle de droite, le prochain état de s est donné par $f(s_g, s, s_d)$.

Par exemple, si $f(x, y, z) = 1$ ssi $x + y + z = 1$, alors on a la transition suivante :

t	…	0	0	1	1	…
$t+1$	…	?	1	0	?	…

La fonction f est la table de transition ou règle de calcul de l'automate. Elle a $2^3 = 8$ entrées possibles, et renvoie un booléen : on peut donc coder ces fonctions sur un octet (et pas moins). Précisemment, si n est un entier 8 bits, la fonction associée sera celle qui à (s_g, s, s_d) associe le bit de poids 2^k de n, où k s'écrit $s_g s s_d$ en binaire, avec s_g bit de poids fort. Cette notation a été popularisée par le mathématicien Wolfram, qui a étudié les diverses règles possibles en fonction du type de comportements qu'elles engendrent.

1- Ecrivez la fonction qui calcule la fonction de transition associée à un octet. Son prototype devra être le suivant :

```
bool transition(char rule, bool left, bool here, bool right);
```

Pour utiliser le type `bool`, qui n'est pas prédéfini en C, il faudra inclure l'en-tête `stdbool.h`. Pour tester, on incluera les lignes suivantes dans la fonction `main`, après avoir inclus l'en-tête `assert.h` (il déclare la fonction `assert()`, qui sert à indiquer une erreur à l'exécution quand son argument est faux) :

```
assert(transition(4,false,true,false));
assert(transition(7,false,true,false));
assert(transition(64+4,true,true,false));
assert(!transition(64+4,true,true,true));
assert(transition(128,true,true,true));
assert(!transition(128,true,true,false));
```

2- Nous allons simuler des automates sur un anneau de taille N. La taille sera fixée à la compilation, ce qui permettra de n'utiliser que des tableaux alloués statiquement. Pour pouvoir changer la valeur de N facilement, nous allons cependant le définir au début du fichier par le biais d'une directive du préprocesseur :

```
#define N 16
```

Définissez ensuite une fonction qui prend un tableau de taille N représentant l'état précédent de l'automate, un autre tableau destiné à stocker l'état suivant, et remplit ce deuxième tableau selon la règle de calcul passée en premier argument. Votre fonction devra respecter le prototype suivant :

```
void step(int rule, bool prev[N], bool next[N]);
```

3- Ecrivez une fonction pour afficher un état du système, comme une ligne de caractères `'0'` et `'1'` séparés par des espaces :

```
void print_line(bool cur[N]);
```

4- Dans votre fonction `main()`, déclarez deux tableaux de N booléens, initialisez le premier avec un seul 1 au centre du tableau, et testez les fonctions précédentes sur une règle simple, comme 0, 255 ou 1.

5- Modifiez votre fonction principale pour utiliser comme règle un entier fourni par l'utilisateur en premier argument sur la ligne de commande. Pour cela on donnera à `main` son prototype général :

```
int main(int argc, char** argv);
```

Le premier argument donne le nombre de paramètres sur la ligne de commande, en prenant en compte le nom du programme lui-même. Le second argument est un tableau de chaînes de caractères, de taille `argc`. Par exemple, si on tape dans le shell `./monprog` alors `argc` vaudra 1 et `argv` sera `{"./monprog"}` et si l'on tape `./monprog 12` alors on aura `argc==2` et `argv` vaudra `{"./monprog","12"}`. Afin de convertir une chaîne en entier, on utilisera `atoi`. Que se passe-t-il quand votre programme est lancé sans argument sur la ligne de commande ? Essayez de gérer "gracieusement" ce cas, en informant l'utilisateur du bon usage de votre programme.

6- Modifiez votre fonction principale pour afficher N états successifs de votre automate. Pour cela, on utilisera seulement deux tableaux, en utilisant la fonction `transition()` pour calculer alternativement l'un à partir de l'autre et l'autre à partir de l'un.

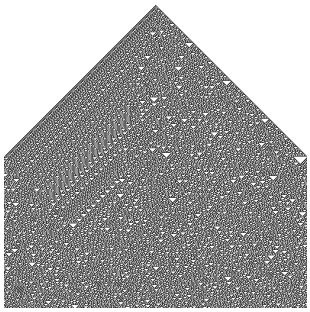
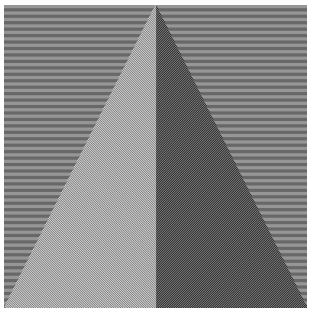
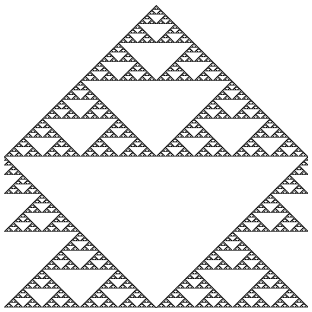
Exercice - 6 *Diagrammes espace-temps*

Ce qui est bien avec les automates 1D, c'est qu'on peut visualiser en 2D leur diagramme espace-temps : il s'agit bêtement de la suite de lignes que votre programme calcule. Mais pour y voir quelquechose d'intéressant, il faut augmenter N, et pour que cela soit joli on va produire une image.

1- Avant vos N lignes de N `'0'` et `'1'` séparés par des espaces, affichez `"P1"` sur une première ligne, puis deux fois N (en notation décimale usuelle) séparés par un espace sur une deuxième ligne. Vous avez produit un fichier au format *plain* pbm, que vous pouvez visualiser avec un afficheur d'image quelconque. (La seconde ligne indique simplement les dimensions de l'image, et la suite indique la succession de pixels noirs et blancs.)

```
$ ./automate 12 > 12.pbm # produire l'image
$ head -n 2 12.pbm       # verifier l'en-tete
P1
16 16
$ ristretto 12.pbm       # afficher l'image    % $
```

C'est l'occasion d'augmenter la taille de votre simulation, et visualiser les diagrammes obtenus pour différentes règles. Ci-dessous, quelques exemples de ce qu'on peut trouver.

Le format d'image *plain* `pbm`, basé sur la notation ASCII, a l'avantage d'être facile à produire, et facile à lire pour un être humain. Mais les fichiers obtenus sont inutilement gros : un bit d'information est codé sur deux octets. Le format d'image `pbm` bit à bit améliore ceci. Il s'obtient en remplaçant l'en-tête `P1` par `P4`, en laissant la seconde ligne inchangée, et en remplaçant les lignes suivantes par un codage au niveau du bit.

Chaque ligne est codée comme une succession d'octets. Les pixels de la ligne, lus de gauche à droite, sont codés comme les bits de la suite d'octets, lus de gauche à droite et dont les bits sont lus du poids fort au poids faible. (En général, le dernier octet peut contenir des bits non significatifs, si la longueur des lignes n'est pas un multiple de huit, mais on pourra prendre cette hypothèse simplificatrice ici.) Les lignes sont simplement concaténées, sans caractère de retour à la ligne entre deux lignes.

2- Adaptez votre programme pour produire des images au format `pbm` compact.

Exercice - 7 *Puissance 4 (Bonus)*

Une grille de puissance 4 est composée de 6 lignes et 7 colonnes. On se propose de représenter les positions d'un joueur de puissance 4 sur 32 bits de la façon suivante :

```
.  .                 .
5 12                47
4 11   etc.         46
3 10       etc.     45
2  9                44
1  8 15 22          43
0  7 14 21 28 35 42
```

On indique ici, à une position dans la grille, le poids du bit utilisé pour coder cette position. On notera que les bits de poids $7k - 1$ (notés par des `.` ci-dessus) ne sont pas utilisés : on devra laisser ces bits à zéro.

On commencera par poser la définition suivante, s'appuyant sur le fait qu'un entier `long long` est codé sur au moins 64 bits — ce qu'on pourra vérifier en affichant ou testant la valeur de `typeof(board)` :

```
typedef
  unsigned long long
  board;
```

1- Comment tester l'existence d'un alignement vertical de quatre positions dans un `board` ?

2- Comment tester l'existence d'un alignement horizontal de quatre positions dans un `board` ?

3- Comment tester l'existence d'un alignement diagonal de quatre positions dans un `board` ?

4- Coder une fonction indiquant si un joueur est gagnant, étant donné le `board` représentant ses positions.

5- Coder un jeu de puissance 4 interactif. On pourra utiliser la fonction scanf pour lire un caractère sur la ligne de commande — documentation sur le web ou `man 3 scanf`.

6- Pour aller plus loin…

— Coder un algorithme min-max pour le puissance 4. La fonction prendra une borne sur la longueur des branches à explorer. Pour évaluer les feuilles de l'arbre exploré, dans le cas d'une paire de grille où personne ne gagne, on pourra prendre la différence des nombre d'alignements de taille 3 des deux joueurs.

- Adaptez votre fonction pour implémenter l'algorithme alpha-beta.
- Exploitez les résultats de John Tromp, qui a "résolu" le jeu : http://tromp.github.io/c4/c4.html. On trouve notamment sur cette page un fichier contenant l'évaluation parfaite (quel joueur gagne) de toutes les positions du jeu après 8 coups.

Si vous voulez approfondir ce langage par vous même, je vous conseille de piocher ce que vous voulez dans ce cours.