TP 6— INTERPRÈTE

http://www.gallium.inria.fr/~scherer/teaching/colles/2012/tp6.pdf

Les questions en rapport avec les TP, Caml ou la programmation en général sont les bienvenues et peuvent être envoyées à gabriel.scherer@gmail.com. N'hésitez pas !

Un code source est (généralement) un fichier texte dans un langage bien défini. C'est très bien, mais souvent on n'écrit pas les codes sources uniquement pour les lire, mais aussi pour "faire quelque chose", "lancer" le programme qui lui correspond. Il y a deux grandes méthodes pour faire quelque chose :

- on peut utiliser un *interpréteur*, un programme déjà lancé sur la machine qui va lire le code source qu'on lui donne et effectuer les calculs correspondants
- on peut utiliser un *compilateur*, un programme qui va lire le code source et produire un autre programme écrit en *langage machine*, un langage très simple que notre ordinateur peut alors exécuter directement

Il est nettement plus simple d'écrire un interpréteur, et c'est ce qu'on va faire ici pour un langage de programmation très simple.

▷ **Question 1.** Écrire un programme `valide : string -> bool` qui prend une chaîne contenant uniquement des '(' et des ')', et indique si elle est bien parenthésée. Par exemple, `"()"`, `"(())"` et `"(()())()"` sont bien parenthésées, mais pas `"())("`. ◁

1 LE LANGAGE BF

Le BF est un langage très simple, qui a été écrit dans le but d'être le plus simple à compiler possible. Un programme BF dispose d'une "bande" mémoire découpée en N cases contenant chacune un entier, et d'un "pointeur" placé sur l'une des cases de la bande, et qui peut se déplacer vers les deux cases voisines. Le langage dispose de 8 instructions, représentées par 8 lettres :

- '>' et '<' déplacent le pointeur vers la case de droite ('>') ou de gauche ('<') de la case courante
- '+' et '-' modifient la valeur de la case courante : '+' ajoute 1, '-' enlève 1
- '.' affiche la valeur de l'entier contenu dans la case courante
- ',' lit un entier, et le stocke dans la case courante
- '[' saute à l'instruction suivant le ']' correspondant si la case courante vaut 0
- ']' revient à l'instruction suivant le '[' correspondant si la case courante ne vaut pas 0

Les crochets fonctionnent comme des parenthèses : un crochet fermant correspond au dernier crochet ouvrant qui n'a pas déjà été refermé.

1.1 MISES EN APPÉTIT

▷ **Question 2.** Étant donné une bande de départ, que font les quatre programmes suivants ?
`,+.` `[-.]` `[[-]>]` `[-]>,[>,]<[.<]` ? ◁

▷ **Question 3.** Écrire un programme BF qui lit deux entiers naturels et affiche leur somme. ◁

▷ **Question 4.** Écrire un programme BF qui lit deux entiers naturels et affiche leur produit. ◁

1.2 Parsing

On va maintenant essayer de *parser* le langage BF : étant donné une chaîne de caractères représentant un programme Brainfuck, on voudrait obtenir une valeur `Caml`qui représente la structure du programme.

On définit pour représenter un programme BF deux types, `instr` et `prog`, représentant respectivement une instruction brainfuck (instruction simple ou boucle délimitée par des crochets) et un bloc d'instructions.

```
type prog == instr list
and instr =
| Move of int       (* déplacement du pointeur *)
| Change of int     (* modification de la case *)
| Read | Print      (* lecture/ecriture *)
| Loop of prog      (* boucle : [...] *)
```

Il faut que vous recopiez cette déclaration dans votre code pour pouvoir l'utiliser ensuite. Vous pouvez remarquer que ces deux types sont mutuellement récursifs : chacun utilise l'autre dans sa propre définition.

▷ **Question 5.** Que fait le programme `,[.-]` . ?
Écrire la valeur de type `prog` correspondant. ◁

On cherche à écrire une fonction `bf_parse : string -> prog` qui prend une chaîne et renvoie la valeur du type `prog` correspondante.

Notre type étant récursif, on peut s'attendre à ce que la fonction `bf_parse` utilise de la récursivité (elle ne sera pas récursive elle-même mais contiendra une sous-fonction récursive). Plus précisément, la récursion dans la définiton de `instr` est dans le cas `Loop` (les blocs délimités par des crochets), donc on peut s'attendre à un appel récursif servant à parser l'intérieur d'une boucle. Il faut donc que notre fonction récursive :

- puisse commencer à une position arbitraire (après le '[' dans le cas de l'appel récursif)
- puisse s'arrêter avant la fin de la chaîne (au ']' correspondant dans notre cas)

On utilisera donc le squelette suivant :

```
exception Erreur of int;;

let bf_parse str =
  let ajoute x (xs, y) = (x :: xs, y) in
  let rec parse i =
     (* votre code ici ... *)
  in
  let (prog, fin) = parse 0 in
  if fin < string_length str then raise (Erreur fin)
  else prog
```

L'exception Erreur sert à indiquer la position où se trouvent des erreurs de syntaxe. Par exemple, le programme `[]][` est incorrect à partir de la 3^e lettre donc renverrait l'exception `Erreur 2`.

Le type de `parse` est `int -> prog * int` : il prend en paramètre l'indice à partir duquel il doit commencer à lire dans la chaîne `str`, et renvoie le bout de programme qu'il a lu, et l'indice auquel il s'est arrêté. Pour parser toute la chaîne on le lance au début (indice 0) et on vérifie qu'il ne s'est pas arrêté avant la fin (sinon c'est une erreur).

▷ **Question 6.** À votre avis ? Compléter le squelette. ◁

1.3 Évaluation

Maintenant, vous allez devoir vous débrouiller seuls.

▷ **Question 7.** Écrire une fonction `bf_eval : prog -> unit` qui évalue le code BF donné. Utiliser deux sous-fonctions mutuellement récursives, `eval_prog : prog -> unit` et `eval : instr -> unit`. ◁

▷ **Question 8.** Question libre : écrire des fonctions travaillant sur le type `prog` pour produire des représentations du même programme en moins d'instructions. Par exemple on peut réunir deux instructions `Move` consécutives en une seule. On pourra même envisager de modifier le type `prog`. ◁