Projet Semestre 5

Rapport niveau 2 : Padawan

Café ou Thé

Eric Ung
Thomas Greaux
Florian Feraud
Fabien Durando

24 Novembre 2016

Table des matières

I. Contexte

Ce rapport a pour but de décrire les solutions apportées par notre équipe afin de doter à l'interpréteur Brainf*ck les nouvelles fonctionnalités telles que :

- l'interprétation de la syntaxe abrégée et des images Brainf*ck
- la réécriture en syntaxe abrégée à partir du programme donné en entrée
- la représentation image des instructions
- la gestion des entrées/sorties
- la gestion des boucles

A partir de notre base de travail présentée lors de l'évaluation 1, nous avons effectué quelques modifications au niveau de l'architecture des classes pour assurer une bonne évolutivité à notre interpréteur.

Les détails de ces changements seront décrits dans la partie suivante. Il traite principalement de la mise en place de packages et de la création d'une classe CommandFactory permettant d'éviter les redondances dans notre code.

Les démonstrations dans la partie finale vont permettre de valider les solutions qui répondent aux fonctionnalités attendues par le client.

II. Modifications apportées à l’architecture du code

- **Gestion des nouvelles syntaxes**

Afin d'exploiter au mieux les classes de commandes, la syntaxe abrégée a été ajoutée dans en tant qu’attribut de classe. Par exemple pour la commande INCR nous avons défini la variable nameShort qui a pour valeur "+".

Cela permet à ce que l'instruction et sa syntaxe abrégée soient traitées de la même manière grâce à la classe CommandFactory.

De la même manière, nous avons géré la lecture des images en ajoutant à chaque classe de commande le code couleur qui lui correspond par la variable colorRGB/Hexa.

- **Séparation reader en ReadFile et ReadImage**

La classe ReadFile a pour but de séparer le traitement d’un fichier .bf et d’une image. En effet, le processus pour récupérer chacune des commandes étant différent, pour ne pas créer des conditions dans le Reader nous avons dû avoir recours à cette séparation.

Après le ReadImage, les commandes sont alors transmises au reader (comme pour le readFile), ce qui permet de ne pas avoir de duplication de code.

- **Utilisation de package**

Le code n’étant plus aussi restreint que pour le premier niveau, nous avons choisi d’utiliser des packages pour « mettre en ordre » les différentes classes.

III. Mise en place des slices et démonstration

- **Slice 5 : Support de la syntaxe abrégée**

Le support de la syntaxe abrégée repose sur le fait que notre reader va lire chaque ligne argument par argument (si l'on lui envoie des commandes dans la syntaxe abrégée), soit ligne par ligne si on lui envoie une couleur ou des commandes en syntaxe longue.

```
Commande :  -p files/Level2/slice5_shortened.bf
Fichier :   ++->++>><+
Résultat :  C0: 1
            C1: 2
            C2: 1
```

Cette nouvelle syntaxe est bien entendu « combinable » avec l'ancienne :

```
Commande :  -p files/Level2/slice5_mixed.bf
Fichier :   ++
            INCR
            -
Résultat :  C0: 2
```

- **Slice 6 : Réécriture**

La réécriture est dans notre code une simple méthode appelée en cas de --rewrite, affichant sur la sortie standard la version abrégée d'une instruction. Elle fait appel à la méthode getNameShort, qui associe syntaxe longue et courte de chaque commande. Les commandes n'étant pas exécutées, l'on n'obtient donc pas d'erreur à l'exécution, même si le code est censé ne pas s'exécuter.

```
Commande :  --rewrite -p files/Level2/slice6_long.bf
Fichier :   INCR
            DECR
            INCR
            INCR
            RIGHT
            DECR
            DECR
Résultat :  +-++>--
```

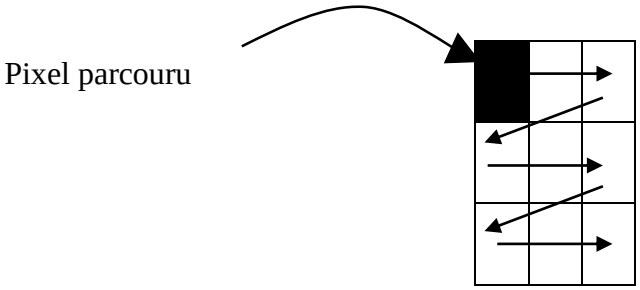
Si une commande est déjà en syntaxe abrégée, elle ne sera donc pas modifiée :

```
Commande :  --rewrite -p files/Level2/slice6_long.bf
Fichier :   INCR
            DECR
```

```
            ++>
            DECR
            DECR
Résultat :  +-++>--
```

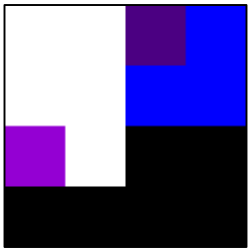
- **Slice 7 : Utilisation d'images**

La classe ReadImage va parcourir l'image en analysant la couleur de chaque pixel supérieur gauche étant donné que chaque instruction possède une zone de 3x3 pixels. En effet, comme évoqué dans le sujet, chacune des zones de 3x3 pixels ne possède qu'une seule couleur.



Représentation du parcours

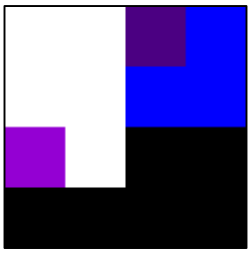
```
Commande :  -p files/Level2/slice5_shortened.bmp
Fichier :   Voir image ci-dessous
Résultat  : C0: 1
            C1: 2
            C2: 1
```



- **Slice 8 :**

Pour ce slice, nous allons cette fois créer l'image à partir d'un fichier brainf*ck. Le principe est simple : grâce à la classe CommandFactory l'on associe les commandes à leur couleur, et l'on colorie en suite une zone de 3x3 pixels correspondant à la couleur de la commande.

```
Commande :  -p files/Level2/slice5_shortened.bf
Fichier :   ++->++>><+
Résultat :
```



- **Slice 9 : Support d'in et out**

Le commandes in et out sont des codées comme étant des commandes comme les autres. In va prendre en entrée le code ASCII de ce qu'il reçoit, donner à la case courante cette valeur. Out va afficher le caractère ASCII correspondant à la valeur de la cellule courante.

```
Commande :  -p files/Level2/slice9_inandout.bf
Fichier :   +>+,.
On rentre : a
Affiché :   a
Résultat :  C0: 1
            C1: 97
```

Ils ont été implémentés pour que lorsqu'un fichier est donné suite à un -i ou -i, pouvoir prendre en entrée un fichier ou en sortie écrire dans un fichier.

```
Commande :  -p files/Level2/slice9_inandout.bf -i files/Level2/in -o
files/Level2/out
IN :        z
Fichier :   +>+,.
Résultat :  C0: 1
                  C1: 122
Contenu du fichier out : z
```

- **Slice 10 : Programme bien écrit**

Le check est, comme pour la réécriture une simple méthode appelée en cas de --check. Elle va vérifier que le programme brainf*ck en entrée ne comporte pas d'erreurs au niveau des parenthèses. Pour ce faire, nous parcourons les commandes, et dès qu'un symbole « [» est rencontré, une variable compteur sera incrémenté. A l'inverse si «] » est rencontré, le compteur sera décrémenté. Ainsi si le compteur est bien à 0 à la fin de l'exécution, et n'a jamais été en dessous de 0 c'est qu'il n'y a pas de problème.

```
Commande :  --check -p files/Level2/slice9_jumpBackFail.bf
Fichier :   +++[++[->-][<+]>
```

```
Résultat :  Process finished with exit code 4
```

Si le code est mal formé, le code de sortie est 4, sinon ce sera 0.

```
Commande :  --check -p files/Level2/slice9_jumpBack.bf
Fichier :   +++[++[->-][<+]>]
Résultat :  Process finished with exit code 0
```

- ## Slice 11 : Exécution des boucles

Cette fonctionnalité a été implémentée de la manière suivante, les différentes commandes à lancer sont stockées. Si la prochaine commande à exécuter est un JUMP, on vérifie si la case mémoire courante est à 0. Si c'est le cas, l'on reprend l'exécution des commandes au BACK correspondant. Si c'est un BACK, l'on vérifie que la case mémoire courante est différente de 0. On partira reprend l'exécution des commandes au JUMP correspondant.

La recherche du back correspondant repose sur le même principe que le --check, à travers de l'utilisation d'un compteur, décrémenté à chaque instruction recherchée et incrémentée si la commande est identique (JUMP suite à un JUMP par exemple).

```
Commande :  -p files/Level2/slice10_jumpBack.bf
Fichier :   +++>+++>++[-]
Résultat :  C0: 3
            C1: 3
```

Affichage de Hello Word :

```
Commande :  -p files/Level2/slice10_jumpBack.bf
Fichier :
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.+++++++..+++.>++.<<+++++++
++++++++.>.+++.------.--------.>+.>.
Affiché :   Hello World!
Résultat :  C1: 87
            C2: 100
            C3: 33
            C4: 10
```

IV. Keep calm and take a step back

L'ajout d'instructions d'entrée / sortie n'a pas eu de « grands » impacts sur le code : il nous a simplement fallu créer les deux nouvelles commandes et les intégrer au CommandFactory pour pouvoir les utiliser.

Les nouvelles représentations telles que les images ont, elles posé plus de problèmes. En effet, la lecture de celles-ci différait d'un fichier brainf*ck avec lequel nous avions l'habitude de travailler. Nous avons donc dû séparer le Reader en deux : ReaderImage et ReaderFile héritant toutes deux de Reader. Une fois cette modification faite, il suffisait d'envoyer les commandes sous forme d'image ou de couleur au reader, qui les interprètera dans chacun des cas grâce au CommandFactory.

Concernant la modularité, nous avons fait en sorte que les futures slices (notamment le niveau 3) ne pose pas de problème avec l'implémentation actuelle. Cependant, il risque d'y avoir tout de même des modifications à apporter car tant que nous n'avons pas commencé à coder, l'on ne sait jamais vraiment si le modèle auquel l'on pensait est véritablement le bon.

La réécriture est une application surjective. En effet, cette option prend en entrée les 3 syntaxes et renvoie la syntaxe abrégée ce qui est important car une telle fonctionnalité doit être capable de travailler sur toutes les représentations des instructions Brainf*ck.

L'algorithme itératif pour vérifier la bonne formation du programme serait le suivant : l'on parcourt toutes les commandes données en entrée. Chaque fois que l'on observe un « [»on incrémente un compteur (initialisé à 0). S'il y a un «] » on décrémente ce même compteur. Si le compteur passe négatif durant le parcours ou qu'il n'est pas à 0 à la fin des commandes, c'est que le code n'est pas bien formé.

Un algorithme récursif permettrait une diminution du temps d'exécution et serait donc plus intéressant que la version itérative.