Rapport Niveau 3 – PS5 BRAINF-CK

MIAOU – MIAOU Is An Open Umbrella
Nassim BOUNOUAS - Guillaume CASAGRANDE
Julien LEMAIRE - Pierre-Emmanuel NOVAC

28 novembre 2016

1 Introduction

Nous vous présentons à travers ce rapport notre version de l'interpréteur du langage de programmation exotique Brainf*ck développée à ce jour jusqu'à la fin du niveau 3, suivant le découpage du sujet d'origine. Nous avons donc repris les fonctionnalités dont le développement a été achevé dans les deux précédents niveaux, à savoir :

- La mise en place d'une `Machine` virtuelle et d'une mémoire (`Memory`).
- La prise en charge des différentes `Instructions` proposée par le langage Brainf*ck (`INCR`, `DECR`, `LEFT`, `RIGHT`, `IN`, `OUT`, `JUMP`, `BACK`).
- La redirection des flux d'entrée et de sortie pour que les instructions `IN` et `OUT` puissent respectivement lire et écrire dans des fichiers plutôt que sur l'entrée/la sortie standard.
- La prise en charge des sauts conditionnels dans une implémentation naïve (décrite plus tard) et de la vérification de leur bonne utilisation dans un programme donnée.
- La prise en charge des différentes syntaxes décrites dans le sujet, à savoir la syntaxe image (les instructions étant alors codées sous la forme de couleurs de 3 x 3 pixels), la syntaxe texte dite "longue" (les instructions sont alors écrites ligne par ligne, sous la forme de leur nom suscité) et la syntaxe texte dite "courte" (la syntaxe la plus connue, agrémentée de symboles).
- Le module de traduction (`Translator`) afin de pouvoir passer d'une syntaxe à une autre.

Ces fonctionnalités étant implémentées, nous devions alors y rajouter celles du niveau 3. Celles-ci étaient, entre autres, composées des `Metrics` qui, à chaque lancement d'un programme Brainf*ck, propose des données sur son exécution, ou encore du `Logger` qui, à la demande de l'utilisateur, devait fournir un fichier log, lié au programme exécuté, et détaillant les différentes instructions effectuées avec leur conséquence propre sur la `Machine` et la `Memory`. Le support des commentaires et de l'indentation permettaient de rentre un programme Brainf*ck aussi libre et compréhensible que l'auteur l'aurait souhaité.

En dehors de ces outils, plus de l'ordre de la maintenance d'un programme Brainf*ck et de sa clarté, l'ajout du support des `Macros` était réellement une vraie fonctionnalité supplémentaire pour le développeur Brainf*ck, qui pourrait alors sauvegarder un morceau de code sous l'appellation de son choix. Chaque appel de cette même appellation serait alors remplacé par le code lui-même lors de la lecture du fichier. Enfin, la `JumpTable` est apparue comme une amélioration de notre implémentation naïve des sauts conditionnels, permettant de lier chaque instruction `JUMP` à l'instruction `BACK` associée et inversement.

Proposer une implémentation de ces différentes fonctionnalités était donc notre objectif pour terminer ce niveau 3. Cet objectif étant à présent atteint, nous vous présenterons dans ce rapport nos différents choix d'implémentations pour le réaliser.

2 Nos choix d'implémentations

2.1 Schéma d'implémentation / Diagramme de classes

La Figure 1 décrit une vision globale de l'architecture de notre application en termes de classes, de fonctionnalités et d'interaction entre les classes. Certains détails de ce diagramme seront abordés plus amplement par la suite.

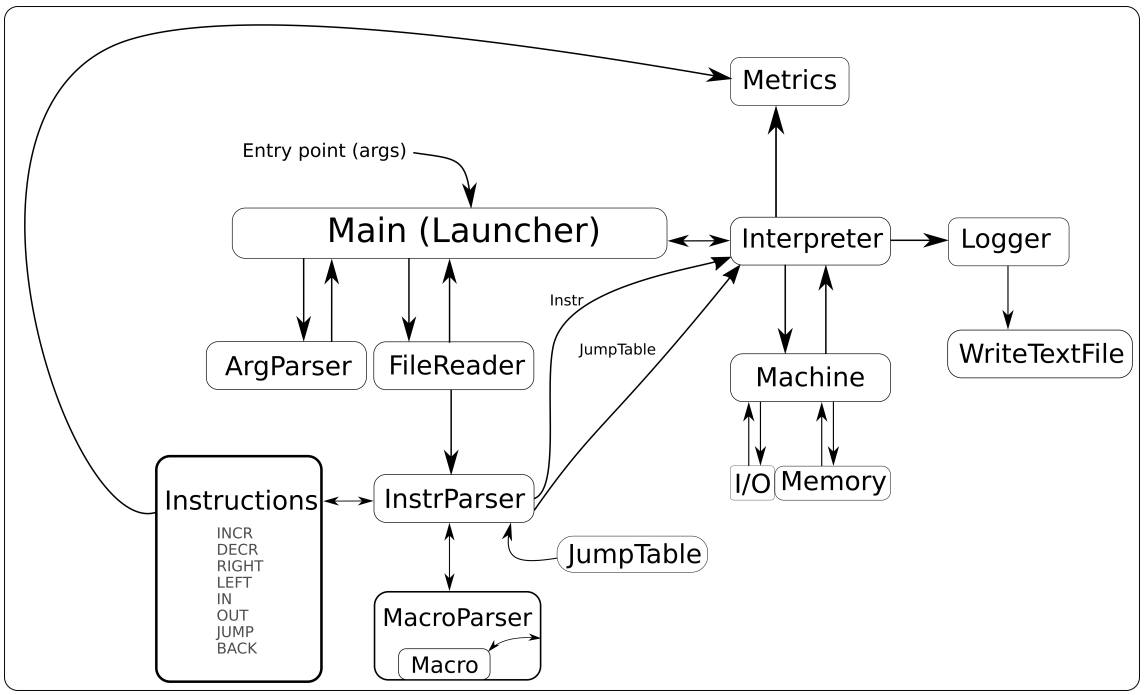

FIGURE 1 – Diagramme de classes

2.2 Metrics

Comme précisé en introduction, les `Metrics` sont des données fournies à chaque interprétation d'un programme BRAINF*CK à propos de celui-ci et de son exécution. Ces `Metrics` sont composés de 6 valeurs, à savoir :

- `PROG_SIZE`, qui contient le nombre d'instructions écrites dans le programme interprété.
- `EXEC_TIME`, qui donne le temps d'exécution du programme en ms.
- `EXEC_MOVE`, qui donne le nombre de fois que le pointeur d'exécution (entendons par là, le pointeur sur les instructions à exécuter) a été déplacé.
- `DATA_MOVE`, qui donne le nombre de déplacements du pointeur sur la mémoire lors de l'exécution du programme.
- `DATA_READ`, qui donne le nombre d'opérations de lecture de la mémoire effectuées par le programme.
- `DATA_WRITE`, qui donne le nombre d'opérations d'écriture dans la mémoire effectuées par le programme.

Ces `Metrics` sont systématiquement affichés à la fin de l'exécution d'un programme BRAINF*CK, comme le décrit la FIGURE 2 :

```
julien@julien-port:~/SI3/Projet/BRAINF-CK_SI3$ ./bfck -p examples/log/1.bf
5
5C0: 1
C1: 53
PROG_SIZE: 12
EXEC_TIME: 2167
EXEC_MOVE: 21
DATA_MOVE: 2
DATA_WRITE: 10
DATA_READ: 6
```

FIGURE 2 – Affichage des `Metrics` à la fin de l'exécution d'un programme BRAINF*CK

D'un point de vue implémentation, les différentes `Metrics` sont tout simplement stockées dans un `enum`. Chacun des éléments de cet `enum` (nommé `Metrics`) possède un attribut de type `long`. Il peut être modifié par diverses méthodes. Ainsi, on peut l'incrémenter, via une méthode nommée `incr`, ou encore directement changer sa valeur en dur par le biais d'une méthode `set` ou finalement l'utiliser comme un chronomètre avec les méthodes `start` et `stop`. Cette pluralité dans les moyens de modifications des `Metrics` sert bien sûr à couvrir tous les comportements possibles (chacune d'entre elles ne se calculant pas de la même façon, `EXEC_TIME` est le seul calculateur de temps par exemple). Toutefois, nous reconnaissons que cette implémentation pose le problème de la liberté du changement de valeur. Il est possible ici de calculer `DATA_WRITE` comme un temps par exemple alors que cela devrait être impossible. Nous avions réfléchi à une meilleure implémentation, plus orientée objet, pour régler ce problème, mais la solution de l'`enum` s'imposait comme la plus simple malgré tout.

Nous avons choisi de diviser les différentes `Instructions` que propose le langage Brainf*ck en différentes catégories. Pour rappel, nous avions implémenté ces `Instructions` sous la forme d'un Command Pattern, c'est-à-dire qu'elles héritaient toutes d'une classe mère `Instruction` définissant une méthode nommée `accept`. Cette méthode est alors surchargée dans les classes filles pour définir le comportement d'une instruction donnée sur la `Machine` et la `Memory`.

Ces mêmes `Instructions` influent sur une partie des `Metrics`, chacune augmentant précisément l'une d'entre elles à chacune de leurs exécutions. En l'occurrence :

- `Left` et `Right` influent sur `DATA_MOVE`.
- `In`, `Incr` et `Decr` influent sur `DATA_WRITE`.
- `Out`, `Jump` et `Back` influent sur `DATA_READ`.

Ces comportements communs entre les différentes instructions nous ont conduits à la création de classes mères de façon à les séparer en fonction de leurs agissements sur les `Metrics`. Elles implémentent donc ces comportements dans leur méthode `accept` à laquelle les classes filles font appel dans leur propre méthode `accept`. Nous avons donc créé les classes `MoveCursor`, `ReadMemory` et `WriteMemory` et les instructions suivent désormais l'architecture décrite dans la Figure 3.

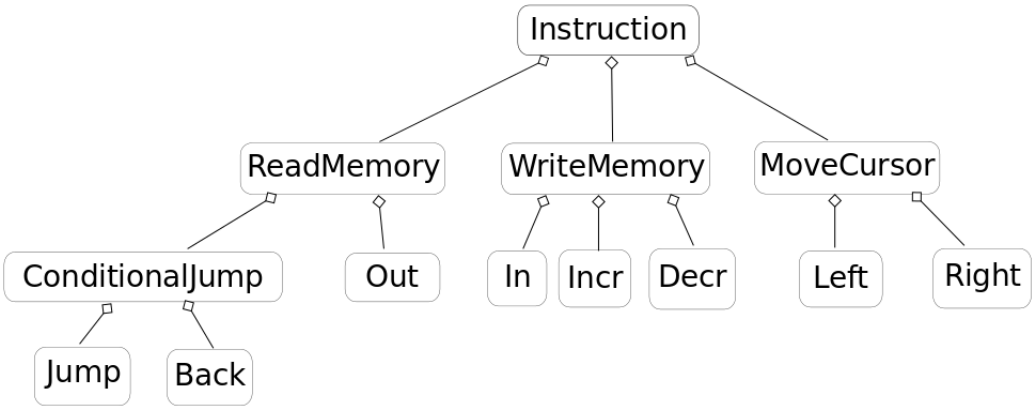

Figure 3 – Architecture des `Instructions` en prenant en compte les `Metrics`.

Si ce choix peut paraitre compliqué pour un simple changement de valeur, il permet de regrouper par l'héritage des comportements communs, plutôt que de recopier ce même comportement dans plusieurs classes différentes. Ainsi, si le code de modification des `Metrics` se voit être complexifié par l'avenir, il pourra être plus facilement modifiable du point de vue des instructions.

Pour ce qui est des `Metrics` restantes, elles sont toutes modifiées au sein même de l'`Interpreter`. `PROG_SIZE` est directement mise à la valeur de la taille de la liste d'instructions tandis qu'`EXEC_TIME` calcule la différence de temps entre le début et la fin de la boucle d'exécution du programme. Enfin, `EXEC_PROG` est incrémentée à chaque tout d'exécution de la boucle (le pointeur d'exécution ayant nécessairement bougé à chaque fois d'une case). La modification de ces `Metrics` est ainsi centralisée et peut être facilement modifiée. La nouvelle boucle d'exécution de l'`Interpreter` peut ainsi être représentée par le morceau de code suivant :

```
int i = 0;
Metrics.PROG_SIZE.set(instructions.size());
Metrics.EXEC_TIME.start();
while (i >= 0 && i < instructions.size()) {
    //old interpreting stuff
    Metrics.EXEC_MOVE.incr();
    i++;
}
Metrics.EXEC_TIME.stop();
```

2.3 Logs

Les logs sont des fichiers retraçant le déroulement de l'exécution d'un programme Brainf*ck. Ils restituent ainsi un certain nombre d'informations récoltées lors de l'exécution de chaque `Instruction` spécifiée dans le

programme Brainf*ck. Un fichier log est créé à chaque fois que l'exécutable de notre application est appelé avec l'option `-trace`, conformément aux spécifications.

La création du fichier de log est alors géré par une classe nommée `Logger`. Un objet `Logger` est créé avec comme paramètre le nom du fichier Brainf*ck actuellement exécuté (qu'il soit sous format texte ou image), ceci dans le but de pouvoir remplacer l'extension du fichier par « `.log` », ce qui nous donne le nom de notre futur fichier de log. Ces opérations sont exécutées dans le constructeur et non pas depuis l'extérieur de la classe, ce qui permet réellement d'isoler ce comportement spécifique au `Logger`. Ceci augmente donc son indépendance vis à vis des autres classes, autrement dit, cela diminue le « coupling » général du programme.

En ce qui concerne l'écriture du fichier lui-même, le `Logger` possède un attribut de type `WriteTextFile` associé au nom de fichier construit précédemment. En conséquence, la classe implémente une méthode `logstep` qui, à chaque appel, écrit une nouvelle étape de log dans le fichier prévu à cet effet. Un tel ajout contiendra un certain nombre d'informations, à savoir :

- Le numéro de l'`Instruction` exécutée, défini par incrémentation (première valeur égale à 1).
- L'emplacement du pointeur d'exécution (pour rappel, le pointeur parcourant les instructions à exécuter).
- Le nom de l'`Instruction` exécutée (sous la forme de sa représentation en syntaxe longue (INCR par exemple)).
- L'emplacement du pointeur de données (la case mémoire sur laquelle on travaille).
- Un affichage entier de la mémoire après exécution de l'instruction.

En outre, une étape de log s'affichera comme suit dans le fichier correspondant :

```
21 - exec 11: INCR on C0
C0: 1
C1: 53
```

La méthode `logstep` du `Logger` est appelée à chaque tour de la boucle d'exécution de l'`Interpreter`. Si le code intégré dans la classe `Logger` aurait ainsi pu être directement intégré à l'`Interpreter`, le choix de l'en séparer nous a semblé être meilleur, toujours dans un but de maximisation de l'indépendance entre les classes et de réduction du « coupling », rendant le code plus maintenable.

2.4 Commentaires et indentation

L'ajout du support des commentaires et de l'indentation avait pour but de permettre à un programmeur Brainf*ck de rendre ses programmes aussi lisibles et compréhensibles qu'il le souhaite. Du point de vue de notre interpréteur, cela se traduisait par le fait d'ignorer (dans le sens de ne pas interpréter) les espaces, les tabulations et les chaines de caractères précédées par un « # ».

Nous décidons de régler ce problème dans l'`InstructionParser`. Pour rappel, cette classe traite le contenu du fichier à interpréter pour l'analyser et en récupérer une liste d'objets `Instruction` compréhensible pour notre interpréteur. C'est pendant ce traitement que sont ignorés les parties commentées, les espaces et les tabulations.

Pour cela, on opère des traitements supplémentaires sur chaque ligne du texte. On regarde d'abord s'il s'agit d'une ligne entièrement commentée (si elle commence par un « # »). Sinon, on cherche à tronquer de la ligne une éventuelle partie commentée, si elle existe. Par la suite, on analyse la ligne sans les espaces et tabulations qui la composent en son début et sa fin. Si l'analyse caractère par caractère est nécessaire (cas d'une ligne qui n'est pas en syntaxe longue, donc), on ignorera les espaces et tabulations.

2.5 Jump Table

Lors de notre première implémentation des sauts conditionnels, nous ne faisions que vérifier si toutes les instructions `JUMP` possédaient bien une instruction `BACK` associée. Dès lors, au sein d'une boucle conditionnelle, lorsqu'une instruction `BACK` est atteinte, nous retrouvions le `JUMP` associé en remontant la liste d'instructions jusqu'à tomber sur ce que nous cherchions. Les instructions ainsi remontées n'étaient pas interprétées, mais l'opération restait malgré tout couteuse.

La `JumpTable` était donc une amélioration de cette implémentation naïve des sauts conditionnels. Nous avons décidé de l'implémenter par le biais d'une classe entière, donc un objet serait instancié dans l'`Interpreter`. Elle relie, par le biais de `HashMap`s, chaque instruction `JUMP` à la position de l'instruction `BACK` correspondante (et vice-versa). La gestion d'une même boucle conditionnelle nécessite ainsi deux fois moins d'opérations qu'auparavant.

Pour montrer l'efficacité de cette amélioration, nous pouvons voir les résultats en termes de `Metrics` d'un même programme une première fois avec notre implémentation naïve des sauts conditionnels et une seconde fois avec la `JumpTable`. Nous avons volontairement choisi un programme conséquent opérant 255 fois une séquence de 255 incrémentations suivies de 255 décrémentations sur la même case mémoire. Nous utilisons donc ici une boucle conditionnelle et un compteur sur la case mémoire précédente.

Après les deux essais effectués sur la même machine, nous obtenons les résultats suivants :

Sans JumpTable :

```
PROG_SIZE: 764
EXEC_TIME: 23
EXEC_MOVE: 257804
DATA_MOVE: 506
DATA_WRITE: 128524
DATA_READ: 254
```

Avec JumpTable :

```
PROG_SIZE: 764
EXEC_TIME: 21
EXEC_MOVE: 129536
DATA_MOVE: 506
DATA_WRITE: 128524
DATA_READ: 254
```

On peut donc bien se rendre compte d'une division par deux du nombre de déplacement du pointeur d'exécution avec la `JumpTable`. De plus, le temps d'exécution se voit diminué de 2ms, environ. Cette différence de temps s'est avérée être plus significative sur un programme comme le classique Hello World pour lequel la différence est d'environ 9 ms. On obtient en effet les résultats affichés en FIGURE 4 et en FIGURE 5.

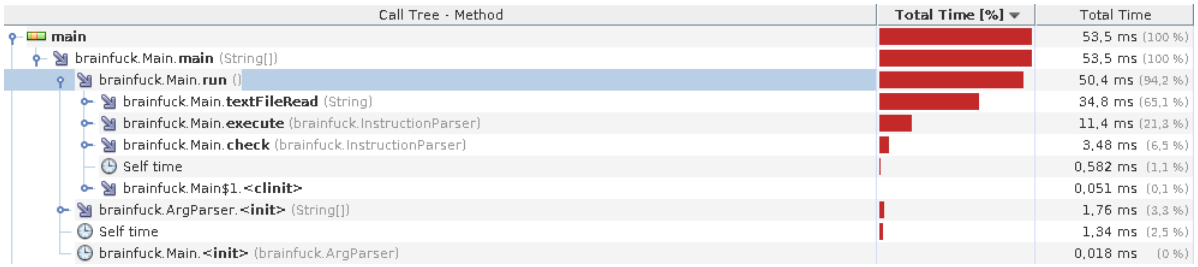

FIGURE 4 – Temps d'exécution du programme Hello World BRAINF*CK sans `JumpTable`.

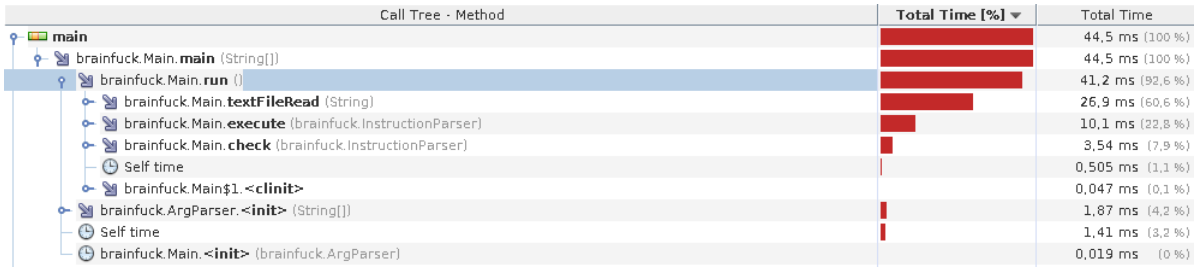

FIGURE 5 – Temps d'exécution du programme Hello World BRAINF*CK avec `JumpTable`.

`TODO: Parler d'implémentation -> Nassim.`

2.6 Macros

2.7 Macros récursives et paramétrées