Projet Semestre 5

RAPPORT NIVEAU 3
JEDI KNIGHT

Café ou Thé

Eric Ung
Thomas Greaux
Florian Feraud
Fabien Durando

14/12/2016

Table des matières

1. Contexte

1.1 Précédemment

Au démarrage de ce projet, nous avons implémenté au travers du premier niveau les instructions de base du langage Brainf*ck. Cela comprenait l'incrémentation de la case mémoire courante, la décrémentation de celle-ci, le déplacement dans la mémoire vers la droite (incrémentation de l'index) et vers la gauche (décrémentation de l'index). Les exceptions relatives à ses différentes instructions ont elles aussi été faites.

Au cours du deuxième niveau, nous avons mis en place de nouvelles fonctionnalités plus avancées. Ces nouvelles fonctionnalités sont assez diverses :

- **Permettre l'utilisation de nouveaux moyens d'exécuter des instructions. Tout d'abord au travers de l'utilisation d'une nouvelle syntaxe du code (la syntaxe abrégée), ainsi que d'une nouvelle représentation : avec des images (format bmp).**
- **La traduction de fichiers Brainf*ck dans un autre format. Avec la génération d'images grâce à du code Brainf*ck (--translate) ainsi que le passage de la syntaxe longue ou de l'image à la syntaxe abrégée (--rewrite). Chacune de ces options n'exécute pas le code.**
- **L'intégration de nouvelles instructions, ayant deux à deux un fonctionnement similaire :**
  - **Les instructions in et out. In permet d'enregistrer dans la case courante la valeur ASCII du caractère reçu et out affiche le caractère ASCII correspondant à la valeur de la cellule courante.**
  - **Les instructions jump et back : chaque jump est lié à un back (et inversement), suivant le même fonctionnement que des parenthèses dans des formules. Jump permet de sauter les instructions jusqu'au back correspondant, si la cellule courante est égale à 0. Back permet de revenir au jump correspondant, si la cellule courante n'est pas égale à 0.**
- **Et enfin, l'implémentation d'un vérificateur de syntaxe. Il permet de lever une exception si les jump/back sont mal formés : si on obtient un nombre de back supérieur au nombre de jump, ou si on termine avec un jump non fermé par un back. Le vérificateur de syntaxe est appelé systématiquement avant d'exécuter le code. Il est également possible de l'appeler en utilisant l'option --check, dans ce cas le code n'est pas exécuté.**

1.2 Nouvelles fonctionnalités

Dans ce nouveau niveau, nous avons amélioré l'interpréteur pour prendre en charge de nouvelles fonctionnalités utiles à l'utilisateur de notre interpréteur Brainf*ck.

Dans un premier temps, les exécutions sont surtout utiles au confort du client :

Les métriques servant à afficher le temps d'exécution du fichier utilisé, et donc à se rendre compte d'éventuels problèmes d'optimisation.

La nouvelle option (--trace) permet la génération d'un fichier de logs contenant les instructions exécutées (ainsi que des informations relatives à la mémoire) pour déboguer le code.

Le support de l'indentation et des commentaires, indispensable pour que le code puisse être aéré et documenté.

L'introduction des macros, permettant au client d'utiliser des instructions de haut niveau. Elles lui permettent d'éviter la duplication de code : au lieu d'utiliser plusieurs fois une même suite d'instructions, il peut donc la déclarer en tant que macro et l'utiliser quand il le souhaite.

Nous avons également optimisé l'interprétation des jump et back de manière à ne plus vérifier chaque instruction lorsque cette commande est appelée, mais à récupérer directement l'emplacement du jump ou back correspondant.

1.3 Diagramme de classe

Voir Annexe 1 (p.11).

2. Mise en place des slices

2.1 Slice 12 : Indicateurs de performance

2.1.1 Besoin utilisateur

Il est intéressant pour un développeur d'avoir accès à différents indicateurs de performance. À présent, après l'exécution du programme, l'interpréteur affiches ces indicateurs. Ils peuvent être désactivés en utilisant l'option --moff (pour metrics off).

2.1.2 Solution apportée

Pour cela, nous avons décidé de créer une nouvelle classe : Metrics. Elle regroupe tous ces indicateurs, en accès statique et la visibilité est limitée au package. Cela permet de les modifier facilement dans le programme, tout en ne permettant pas de les modifier en dehors du programme.

A priori, ces indicateurs doivent être modifiés a beaucoup d'endroits différents.

Or, ceci n'est pas souhaitable : il est préférable d'avoir un code le plus orthogonal possible au code client.

Pour cela, nous avons changé la modélisation des pointeurs mémoire et d'exécution. Anciennement modélisé par de simples variables, nous avons opté de les modéliser par des objets, avec les classes MemPointer et ExecPointer.

2.1.3 Pourquoi cette solution ?

Le raisonnement derrière ce choix est double : d'une part les pointeurs ont maintenant plusieurs tâches : pointer la case mémoire/instruction actuelle et modifier des indicateurs quand ils se déplacent. Les pointeurs sont plus complexes, il est donc logique d'en faire des objets à part entière. De plus, on ne doit pas saturer la classe Memory et d'autre part, limiter la gestion des indicateurs dans le code afin d'avoir une implémentation la plus orthogonale possible.

2.1.4 Représentation

La représentation des macros se fait de la manière suivante :
Commande : -p files/Level3/hello.bf

Exemple de résultat:

> **Metrics:**
>
> **PROG_SIZE:** *Nombre d'instructions*
> **EXEC_TIME:** *Temps d'exécution*
> **EXEC_MOVE:** *Nombre de fois que le pointeur d'exécution a été bougé*
> **DATA_MOVE:** *Nombre de fois que le pointeur de mémoire a été bougé*
> **DATA_WRITE:** *Nombre de fois que l'on écrit dans la mémoire*
> **DATA_READ:** *Nombre de fois qu'on a lu dans la mémoire*

Commande : --moff -p files/Level3/hello.bf //N'affiche pas les indicateurs

2.2 Slice 13 : L'option trace

2.2.1 Besoin utilisateur

Lorsqu'un utilisateur développe en utilisant un IDE, il a accès à un mode de débogage. Pour simuler ce mode, nous avons développé au cours de ce slice une nouvelle option : la trace. Elle permet à un utilisateur de voir un rapport de ses différentes commandes, ainsi si jamais un problème a lieu lors de l'exécution ce sera la dernière suite d'information présente.

2.2.2 Solution apportée

Dans le but de faciliter le débogage, nous avons implémenté l'option --trace. Elle permet à l'utilisateur la génération d'un fichier de logs, contenant les différentes instructions lancées par l'utilisateur au cours de son programme ainsi que des informations relatives à l'état courant de la mémoire (numéro d'exécution de la commande, contenu de celle-ci, valeur de la case courante et une copie de l'état courant de la mémoire).

Ce fichier de logs permettra à l'utilisateur de connaître l'instruction sur le point d'être exécutée lorsque le programme bloque, et ainsi de récupérer les informations énoncées précédemment.

L'implémentation de cette fonctionnalité est assez simple : à chaque commande, sont écrits dans le fichier de logs les différentes informations que l'on doit afficher. À chaque nouvelle commande, le fichier est repris à partir de sa dernière ligne, et les nouvelles informations sont écrites à la suite. Ainsi, lors du déclenchement d'une exception causée par un code Brainf*ck non fonctionnel, le fichier contient bien tous les logs jusqu'à ce que celle-ci soit levée.

Pour la création du fichier, nous avons choisi d'utiliser un FileWriter.

Le fichier généré aura pour nom le nom du fichier sans son extension, suivi de l'extension ".log".

2.2.3 Pourquoi cette solution ?

Nous avons choisi d'implémenter l'option trace de la manière suivante, pour toujours obtenir un fichier de logs. En effet, écrire les commandes au fur et à mesure sans jamais fermer et rouvrir le fichier ne permettent pas de gérer les cas d'exception, et renverraient un fichier vide dans le cas échéant.

2.2.4 Représentation

La représentation de la trace dans un fichier (.logs) est la suivante :

> **Step n°** *Numéro d'exécution de la méthode*
> **Next command :** *Commande sur le point d'être exécutée*
> **Data pointer value :** *Position de l'index dans la mémoire*
> **Memory SNAPSHOT :**
>
> *Cellules non égales à 0 de la mémoire*
> ---

2.3 Slice 14 : Commentaires et indentation

2.3.1 Besoin utilisateur

Le besoin de l'utilisateur concernant ce slice se situe au niveau de la mise en forme du code. En effet, l'écriture de commentaires est une fonctionnalité souvent présente dans les langages de programmation et il est donc fondamental qu'il soit supporté par notre interpréteur Brainf*ck. Cela permet à l'utilisateur d'aérer son code pour améliorer la lisibilité, ainsi que de le documenter pour s'aider lors du développement.

2.3.2 Mise en œuvre

Le principe utilisé pour permettre son utilisation est le suivant :

- **Si un "#" ou "\n" est lu, on passe à la ligne suivante**
- **Si un "\t" ou " " est lu, on passe au caractère suivant**

Ces vérifications sont faites dans une classe Formatting qui comporte trois méthodes : delCom (renvoyant la ligne sans commentaire), delSyntax (renvoyant la ligne sans \t, \n, ' ') et delNullColor (supprimant les lignes contenant la couleur noire : celle-ci est utilisée pour remplir les espaces manquant sur une image d'instructions Brainf*ck).

2.3.3 Pourquoi cette mise en œuvre ?

Le slice a été mise en œuvre de cette manière pour, tout d'abord nous permettre de réutiliser les commentaires et l'indentation dans d'autres classes, comme par exemple les macros. L'autre implémentation "possible" aurait été de rajouter les conditions (commentaire ou indentation) directement dans le Reader, qui était nettement moins intéressante : non réutilisable, et les conditions auraient été illisibles si on souhaite ajouter de nouveaux symboles non lus lors de la compilation.

Pour éviter tout problème relatif à des caractères non existants, seuls les caractères reconnus en tant que commentaire ou indentation sont ignorés. En effet, ignorer tout ce qui ne correspondait pas à une instruction aurait été une erreur : des caractères non existants pourraient être glissés dans le programme sans être détectés.

Nous avons choisi de placer les différents appels vers Formatting dans le Reader de manière à ne pas lancer l'instruction si elle correspond à un des caractères ou string énoncé précédemment.

2.3.4 Problèmes mis en cause

L'implémentation de ce slice n'a pas posé de problèmes au niveau du code. En effet le code n'a pas subi de grandes modifications : il suffit de vérifier si le caractère (ou string) suivant correspond ou non à de l'indentation ou à un commentaire. Il nous a donc seulement fallu créer une nouvelle classe contenant des méthodes permettant supprimer les commentaires ou l'indentation, et de l'appeler avant les différentes exécutions.

2.4 Slice 15 & 16 : Gestion des macros

2.4.1 Besoin utilisateur

Le client peut être amené à définir ses propres macros permettant à une suite de commandes d'être répétée un certain nombre de fois sans avoir besoin de réécrire cette suite. Et ensuite il sera possible d'appeler les macros à différents endroits avec des paramètres différents.

2.4.2 Mise en œuvre

Le principe utilisé pour créer une macro dans notre interpréteur est le suivant :

- **"/" pour préciser au programme que l'on va créer une macro**
- **Nom de la macro**
- **0, 1, ou 2 paramètres ou une macro déjà existante (il n'y aura pas dans ce cas de liste de commandes à écrire derrière)**
- **Commandes à effectuer**

Le premier paramètre correspond au nombre de fois que l'on souhaite répéter la macro, alors que le second paramètre correspond au nombre de commandes que la macro va passer : si une macro est appelée avec un second paramètre égal à 3, on remplace alors l'appel de celle-ci par les instructions

contenues dans la macro, à partir de la 4e instruction. L'on peut donc rappeler une même macro, et l'adapter en fonction des instructions désirées.

Il est également possible d'utiliser des paramètres lors de la création de la macro et donc lors des appels de celle-ci. Les arguments utilisés lors de l'appel d'une macro vont écraser ceux définis lors de la création de celle-ci. Si l'on appelle une macro avec deux paramètres, les paramètres définis lors de la création ne seront alors pas pris en compte. Si l'on appelle une macro avec seulement le premier paramètre, le second paramètre sera celui de la déclaration. Enfin, si l'on appelle une macro sans paramètres ce seront ceux défini à la création qui seront pris en compte.

Une macro sans paramètre s'écrira : /NomMacro listeCommandes
Une macro avec paramètres s'écrira : /NomMacro param1 param2 listeCommandes
Une macro de macro s'écrira : /NomMacro nomMacroExistante

Le principe utilisé pour appeler une macro est le suivant :

- **Nom de la macro**
- **0, 1, 2 paramètres**

Ainsi pour appeler une macro sans paramètre et donc prendre ceux d'origine on écrira : NomMacro. Pour appeler une macro avec paramètres on écrira : NomMacro param1 param2

Pour cela, nous avons créé une classe Macro permettant la création et le remplacement de macros par leur équivalent en instructions. Lors de l'appel de la macro avec '\' on crée un nouvel objet macro qui possède un nom, une liste de commandes et deux paramètres. Ensuite, à chaque appel de la macro, une liste de Commandes (correspondant à une liste d'instructions Brainf*ck) est renvoyée. Par la suite, on exécute les commandes comme l'on a fait depuis le début du projet, la seule modification est que les macros sont remplacées par leur équivalent en instructions.

2.4.3 Pourquoi cette mise en œuvre ?

Le slice a été mise en œuvre afin de pouvoir donner à l'utilisateur une utilisation des macros ni trop restreinte (pour que l'utilisateur puisse avoir le sentiment que les macros sont utiles), ni trop complexes pour ne pas le perdre avec des fonctionnalités diverses et surement inutiles.

Nous avons décidé de créer les macros à partir de cette syntaxe car elle reste compréhensible pour l'utilisateur.

Puis avec une syntaxe comme celle-ci, nous pouvons facilement récupérer les différents paramètres des macros créées ou appelées : en effet, les différents paramètres ne sont séparés que d'un espace. Il nous suffit donc de couper la ligne entre chaque espace et de récupérer le nom de la macro ainsi que ses différents paramètres.

Concernant l'implémentation de ce slice, nous avons eu pour objectif de ne pas changer le modèle utilisé jusqu'à présent. En effet, les macros sont des commandes préprocesseur permettant de transformer un appel (donc du texte) en instructions, ce qui n'a aucune raison de modifier le code déjà écrit. Cela explique donc la mise en place d'une classe macro, et les appels de celle-ci depuis le Reader.

2.4.4 Problèmes mis en cause

Le premier problème a été de savoir quelle sera la portée de nos macros, c'est-à-dire dire combien de paramètres nous allons lui donner et à quoi correspond chacun pour ne pas faire des macros trop basiques ni trop compliquées pour l'implémentation.

Un autre problème a été de savoir comment nous allons interpréter les macros dans notre code. Nous avons donc décidé qu'une macro créée fonctionnerait comme une commande qui pourra donc être appelée n'importe quoi, mais qui mettra une erreur si la macro n'existe pas.

2.5 Slice 17 : Optimisation de l'interpréteur

2.5.1 Besoin utilisateur

Si l'interpréteur n'est pas performant, l'utilisateur n'y peut rien. On doit donc par conséquent faire en sorte qu'il soit le plus optimisé que possible.

2.5.2 Mise en œuvre

Jusqu'à présent, quand on rencontrait une instruction Jump ou Back satisfaisant la condition correspondante, on parcourait la bande d'exécution de manière linéaire. Il est possible de faire mieux en sautant directement à la bonne instruction. Pour cela Nous avons décidé de créer une nouvelle classe : JumpTable. Comme son nom l'indique, elle va stocker les indices des instructions Jump et Back, ainsi que son instruction associée. Étant donné que l'on vérifie l'intégrité du programme dans tous les cas, nous n'avons pas besoin de le faire en même temps.

La création de cet objet repose sur le principe d'une pile, sans pour autant utiliser cette structure de données de manière rigoureuse. On parcourt la bande d'exécution, à chaque fois que l'on rencontre un Jump, on empile l'indice correspondant ; dans notre cas on l'ajoute dans une liste. Quand on rencontre un Back, on récupère l'indice correspondant ainsi que l'indice du Jump associé en le dépilant ; dans notre cas, on le récupère en supprimant le dernier élément de la liste. On met les deux indices dans un vecteur de taille 2 que l'on ajoute dans une autre liste, qui sera la liste finale.

2.5.3 Pourquoi cette mise en œuvre ?

L'exercice d'association des parenthèses (ou dans notre cas des crochets) est un cas classique d'utilisation des piles. Pour notre utilisation, nous n'avons pas jugé intéressant à implémenter proprement la structure de pile, nous en avons simulé le comportement dans notre utilisation, en utilisant une liste de vecteurs.

3. Keep calm and take a step back

What were the pros and cons of using the object-oriented paradigm for this project?

L'avantage de l'utilisation du paradigme orienté objet pour ce projet est de simplifier énormément le code. En effet, grâce aux objets l'on peut décomposer les différentes fonctionnalités attendues pour ce projet. Une fois ses différents objets combinés, le problème peut être résolu assez simplement.

Les inconvénients sont que la mise en place de l'architecture est assez compliqué, surtout sur les premiers niveaux. En effet, lors de la mise en place d'un projet même en le lisant dans son ensemble, il est très difficile de prévoir une architecture qui ne serait modifiée au cours de l'avancée. Nous avons notamment perdu beaucoup de temps sur cet aspect, mais au final sans ces diverses modifications le projet n'aurait pu avancer jusqu'à ce point.

What is the role of the macros in the language specifications?

Lors de l'appel d'une macro, celle-ci sera remplacée par les instructions désirées avant l'exécution du code. Elles permettent ainsi d'éviter des duplications de code, pouvant être gênante pour un utilisateur : il préférera déclarer une fois la macro, et l'appeler que de réécrire à chaque fois la suite d'instructions. Si l'utilisateur veut faire une suite de commandes répétitives, il aura seulement à appeler une même macro le nombre voulu de fois pour que l'exécution se fasse de la même manière que si l'on avait directement écrit dans le code les instructions contenues dans la macro.

How can you empirically measure the optimization benefits?

Les macros permettent à l'utilisateur de ne pas avoir à réécrire son code. L'optimisation se fait alors au niveau de l'utilisation du logiciel par l'utilisateur : son code sera rendu plus lisible, et moins de répétitions seront faites. Au niveau de l'interpréteur, une macro ne va pas optimiser quoi que ce soit. En effet, son seul et unique rôle étant d'être remplacée par les instructions correspondantes à celle-ci, l'exécution se fera de la même manière avec ou sans.

What will happen in case of recursive macros?

Une macro, tel que nous l'avons défini ne peut que remplacer dans le code son appel par une suite d'instructions. Ainsi, un appel récursif d'une macro ne comporte pas de conditions d'arrêt (ce qui sera différent dans le cadre des fonctions du niveau 4). Donc, créer une telle macro bouclera indéfiniment : aucune condition d'arrêt n'est utilisée.

What was the impact of introducing the metrics in your code?

Les indicateurs permettent de donner bon nombre d'indications concernant l'exécution du code de l'utilisateur. En effet, lorsque celui-ci va coder et exécuter depuis l'interpréteur, il va chercher à optimiser celui-ci. Ainsi, il pourra se rendre compte par exemple d'une durée d'exécution trop importante, et par la suite essayer de diminuer cette durée d'exécution en apportant des modifications à son code. Outre l'optimisation du code, il peut aussi se rendre compte de problèmes dans son code, en observant des métriques ayant des statistiques vraiment trop élevées.

What does the jump table cost to your interpreter?

La "jump table" ne nécessite pas beaucoup de modifications : il s'agit de parcourir la liste des instructions pour associer chaque "Jump" à un "Back" (cf. slice 17). C'est donc linéaire en fonction du nombre d'instructions, et en contrepartie, les opérations "Jumps" et "Back" deviennent des opérations à coût linéaire en le nombre de Jump/Back.

Does it worth it?

À **partir du moment où le programme rentre dans les conditions de plusieurs Jump ou Back, ce sera rentable. En effet, si l'on a un programme qui ne rentre jamais dans les conditions des Jump et Back, on aura parcouru une fois les instructions entièrement et sauvegardé les correspondances entre Jump et Back pour rien. Cependant, si l'on rentre dans les conditions de Jump ou Back, l'on n'aura pas à parcourir les instructions jusqu'à trouver le Jump/Back correspondant, ce qui est un gain de temps d'exécution et d'appels mémoire.**

Some metrics can be statically computed in given cases (e.g., no conditional instructions). Does it make sense to separate the two implementations?

Certaines métriques peuvent être calculées de manière statique : en l'absence de Jump/Back, on sait par avance toutes les métriques, à l'exception du temps d'exécution.

En revanche, il n'est pas pertinent de faire la distinction des deux cas, car d'une part on n'y gagnerait rien (ou très peu), donc pas de valeur ajoutée et d'autre part il est beaucoup plus simple de faire directement le cas général.

Do you see other things one can optimize in the virtual machine?

Une optimisation possible de notre interpréteur serait de transformer nos listes en vecteurs. Nous avons beaucoup de données stockées dans des listes : la liste des instructions et la "JumpTable" par exemple. En les transformant en vecteur, on optimise le temps d'accès aux données.

4. Les perspectives pour la suite

Pour le prochain et dernier niveau, nous avons prévu d'implémenter les slices de la manière suivante :

- **Slices 18, 19 (procédures, procédures avec paramètres) :**
  Etant donné qu'elles sont liées (de la même manière que les 15 et 16) nous allons les implémenter de manière « rapprochée » (pratiquement en même temps). En effet, nous préférons travailler en prenant directement en compte les paramètres, comme nous l'avons fait pour les macros.
- **Slice 20 (fonctions) :**
  Comme ce slice ressemble aux procédures (la différence est que les fonctions retournent quelque chose, pas les procédures), nous allons attendre la fin de l'implémentation des procédures avant de s'y intéresser.
- **Slice 21 :**
  Cette partie étant indépendante des trois précédentes, elle sera implémentée en même temps. Dans un premier temps, seules les fonctionnalités des trois premiers niveaux seront intégrées. Une fois que l'on sera sûr de l'implémentation du reste du niveau, l'on ajoutera la génération de code à partir de fonctions et procédures pour finir cette partie. Nous pensons générer du code Java ou C, qui sont les deux langages que nous avons le plus l'habitude d'utiliser.

5. Annexes

5.1 Annexe 1 : Diagramme de classe