UE : Compilation
Compilateur Ristretto

Arnaud Peralta, Yohann Goffart

https://github.com/arnaudperalta/ristretto-compiler

Sommaire

1 Introduction

Ristretto est un compilateur de fichier d'extension .ris qui traduit ce fichier en un fichier .class Java compréhensible par la Java Virtual Machine.

Le fichier .ris est composé d'un code C simplifié qui sera expliqué dans la section Manuel d'utilisation. Tout le code Ristretto est traduit dans une seule classe Java nommée comme le nom du fichier source et commençant par une majuscule.
Les instructions en dehors des fonctions seront donc traduites dans le constructeur de classe. Pour le reste, chaque fonction sera une méthode de cette classe et pourra utiliser les attributs de classe déclarés dans le scope global du fichier source.

2 Informations pratiques

Pour bien utiliser ce compilateur et comprendre son fonctionnement, voici plusieurs commandes à saisir :

Pour compiler un fichier d'extension .ris:

```
$ ./ristretto factorielle.ris
```

Pour exécuter le fichier .class produit par le compilateur:

```
$ java -noverify Factorielle
```

Pour observer les instructions bytecode du fichier .class:

```
$ javap -c -s -v -p -l -constants Factorielle.class
```

3 Manuel d'utilisation

Le compilateur Ristretto utilise une version simplifiée du langage C. Chacun des concepts sont expliqués dans les sous-sections suivantes :

3.1 Fonction main

Comme en C, la fonction main est la première fonction exécutée, et doit obligatoirement posséder le prototype suivant :

```
void main(void) {
     [...]
}
```

3.2 Instructions d'affichage

Le langage interprété par Ristretto possède deux instructions pour réaliser une sortie de données par le programme. Ces deux instructions sont print et println et possède la même utilité que leurs équivalents en Java.

```
void main(void) {
    print "Ce message s'affiche ";
    println "sur la même ligne";
    print "Je viens de sauter une ligne";
}
```

Ces instructions prennent en paramètre une valeur de type bool, int, float ou une chaine de caractères, elles sont seulement utilisables à l'intérieur d'une fonction.

3.3 Variables globales

Les variables globales sont déclarées en dehors des fonctions, leurs déclarations au sein du fichier .class, sont effectuées dans le constructeur de la classe.
Elles peuvent être de type int, float, bool et void.

```
int var1 = 0;
float var2 = 0.0;
bool var3 = true;

void main(void) {
    println var1;
}
```

Dans cet exemple la variable var1 est affichée sur la sortie standard.

3.4 Fonctions

Les fonctions possèdent la même syntaxe que le langage C, elles ont un type de retour, un identificateur et des paramètres:

```
int facto(int n) {
    if (n == 0) {
    return 1;
    }
    return facto(n - 1) * n;
}
```

La fonction facto retourne une valeur de type int et prend en paramètre un int d'identificateur n.

3.5 Conditions

A l'intérieur des fonctions, il est possible d'utiliser des instructions de condition telles que if et else :

```
void main(void) {
    if (true) {
        println "Je suis affiché";
    } else {
        println "Je ne suis pas affiché";
    }
}
```

3.6 Boucles

En ce qui concerne les boucles, seule l'instruction while est disponible dans le langage compris par Ristretto, le while fonctionne de la même façon qu'en C :

```
int i = 0;
while (i < 5) {
    println i;
    i = i + 1;
}
```

Cette boucle affiche un entier par ligne de 0 à 4.

4 Programmes d'exemples

Tous les exemples donnés ci-dessous sont disponibles dans les sources du projet, dans plusieurs fichiers .ris.

4.1 Factorielle

Le programme ci-dessous calcule la factorielle selon le paramètre donné à la fonction facto :

```
int facto(int n) {
    if (n == 0) {
    return 1;
    }
    return facto(n - 1) * n;
}
```

4.2 Série de Gregory-Leibniz

Le programme ci-dessous affiche les résultats de la suite de Grégory Leibniz de 0 à n. Cette suite calcule une approximation du nombre Pi.
Plus n est grand, plus la dernière approximation de Pi sera précise.

```
void gregory_leibniz(int n) {
    float pi = 0.0;
    float denominateur = 1.0;
    boolean sign = true;
    while (n > 0) {
        if (sign) {
            pi = pi + 1.0 / denominateur;
        } else {
            pi = pi - 1.0 / denominateur;
        }
        sign = !sign;
        println 4.0 * pi;
        denominateur = denominateur + 2.0;
        n = n - 1;
    }
}
```

Après plusieurs milliers d'itérations, la limite de taille d'un float est visible et la précision atteint son maximum.

5 Technicités

L'écriture du fichier .class se réalise d'un trait lorsque le fichier .ris est totalement analysé. Toutes les informations sont donc stockées dans des structures afin de les modifier si nécessaire pendant l'analyse syntaxique. Les structures utilisées sont détaillées dans la documentation officielle Java.

5.1 Constant-pool

La constant pool est l'endroit où les données écrites dans le code sont stockées, ainsi que les données implicites telles que le nom de la classe et la libraire stockant les fonctions d'affichage. Elle est gérée dans un module nommé constant_pool.
Chaque constante est donc stockée dans un tableau appartenant à une structure principale, chaque membre de cette structure possède un numéro d'index équivalent à l'index de cette constante dans le fichier class au sein de la vrai constant-pool.
On pourra donc charger des constantes au sommet de pile dans le bytecode grâce à ces numéros d'index. Chaque types de constantes possède une structure différente dans le code C

5.2 Method-pool

Les méthodes traduites en bytecode sont stockées dans cet espace. Un module est consacré à la gestion de cet espace car nous avons besoin d'organiser les variables locales de la fonction ainsi qu'enregistrer les instructions bytecode de celle-ci.

5.3 Gestion des if et while

Lorsqu'un if ou un while évalue une condition, on évalue si la valeur en sommet de pile est inférieur ou égale à 0. On traduit donc les expressions booléennes avec des entiers, 1 pour true et 0 pour false.
Quand l'analyseur syntaxique rencontrera un if ou un while, il agira de la même manière, peu importe la condition que comporte cette structure, et exécutera les procédures suivantes :

While :

1. enregistrement de la ligne où l'évaluation de la condition commence
2. évaluation de la condition pour se retrouver avec un entier en sommet de pile
3. calcul du nombre d'instructions bytecode à exécuter dans la boucle
4. insérer un saut

If ... Else :

1. insertion de la valeur 1 en sommet de pile
2. évaluation de l'expression de la condition du if
3. si l'expression est inférieure à 1, elle est jugée comme fausse; on saute jusqu'au crochet fermant. Sinon on dépile le 1 précédent puis on empile un 0 en exécutant les instructions dans le if
4. si un else est présent, on regarde la valeur en sommet de pile. Si on a 1, on exécute l'intérieur du else, sinon on ne l'exécute pas.

Cette procédure nous garantit qu'on rentrera soit dans le if, soit dans le else.

5.4 Opérations

Les opérations écrites en Ristretto sont directement traduites en instructions bytecode. La particularité de cette traduction est que le langage bytecode utilise une pile pour procéder à toutes ses opérations, et que par conséquent, chaque opération a du être repensé pour fonctionner avec une pile. La priorité des opérations joue un rôle lors de cette traduction puisque les opérations les plus prioritaires sont exécutées en premier dans le bytecode.

Par exemple pour $3 + 4 * 5$, on obtient ce schéma :

1. on push 4
2. on push 5
3. on push la multiplication
4. on push 3
5. on push l'addition

6 Problèmes rencontrés

Ce projet a demandé une très grande quantité de recherche et de travail. Beaucoup de concepts ont du être compris pour parfaitement manipuler les fichier .class. Le débugage du programme était complexe car il a fallu lire octet par octet le fichier .class sorti par le programme pour comprendre ce qu'il se passait réellement.
Le premier obstacle a été de comprendre et de corriger le boutisme lors de l'écriture du fichier. En effet les entiers de 2 et 4 octets étaient écrit à l'envers (en little-endian) lors du début du développement du programme. Pour corriger ce problème, nous nous sommes aidés des fonctions htons et htonl de la librairie ¡netinet/in.h¿. Ces fonctions renversent l'ordre des couples d'hexadécimales pour obtenir un petit-boutisme pour les entiers de 2 et 4 octets.

Le second problème a été de modéliser les instructions telles que le if et le while avec des instructions bytecode très basiques. Nous n'avons sans doute pas réalisé de la meilleure des façons notre bytecode, mais nous nous sommes assurés d'avoir un compilateur fonctionnel.