Projet de compilation – L3 Informatique
RISTRETTO *: un compilateur pour JVM*

Nicolas Bedon Arnaud Lefebvre

20 janvier 2020

1 Présentation générale

Le but du projet est d'écrire un compilateur pour un petit langage structuré, que nous appellerons RISTRETTO, et qui génère en sortie du *byte-code* Java.

RISTRETTO contient les types de base `int`, `float` et `bool`, qui sont exactement ceux de Java. On ne peut pas définir de nouveaux types. C'est un langage impératif (comme C) structuré en blocs, comme C ou Java. Chaque variable est globale ou locale au bloc dans lequel elle est déclarée, suivant les règles habituelles (celles de C et Java). Même si une JVM est utilisée pour exécuter les programmes RISTRETTO, ça n'est pas un langage objets.

Comme en C, toute variable doit être définie avant d'être utilisée. Lors de sa définition, une variable peut être initialisée par une constante. Si l'initialisation est absente, la variable prend une valeur par défaut, suivant les conventions de Java. Le compilateur contrôle les types : toute opération sur une valeur de type inattendu génère un message d'erreur. Contrairement au C, mais conformément à Java, les fonctions n'ont pas besoin d'être définies avant d'être utilisées.

Les structures de contrôles sont celles habituelles : boucles `for` et `while`, `if`, `return` avec la syntaxe du C. Comme en C et en Java, il est possible de déclarer une variable de boucle dans la structure `for`.

Le langage dispose d'une instruction `print` affichant la valeur en argument, qui peut être de type `int`, `float`, `bool` ou bien une chaîne de caractères constante. Le `print` est le seul cas où on trouve des chaînes de caractères. Elles ont le format habituel (entre guillemets), et ne contiennent que des caractères dont le code ASCII se trouve dans l'intervalle $[32 \dots 128[$. Cette contrainte facilite l'écriture du compilateur, car pour ces caractères le code UTF-8 [1] et le code ASCII coïncident. Une version `println` permet le passage à la ligne après affichage de la valeur. Le langage ne contient pas d'opération sur les chaînes de caractères.

RISTRETTO permet la récursivité dans les fonctions.

Voici un exemple de code en RISTRETTO, contenu dans un fichier de nom `exemple.ris` :

```
int a=5; // Une variable globale avec initialisation

int fact(int n) {
  if (n==0)
    return 1;
  else
    return n*fact(n-1);
}

/* Il est obligatoire de définir une fonction void main(void)
```

1. Le format appelé UTF-8 de Java diffère légèrement du format UTF-8 de la norme internationale. Dans l'intervalle de codes $[32 \dots 128[$, ces deux formats UTF-8 et le format ASCII coïncident.

```
   qui est la fonction principale du programme
*/
void main(void) {
  println fact(a); // On remarque que println est une instruction, pas une fonction
}
```

Il se compile et s'exécute de cette manière :

```
% ristretto exemple.ris
% java -noverify Exemple
```

L'exécution de la première ligne produit le fichier `Exemple.class`, dont le nom est obtenu par transformation de `exemple.ris`. L'option `-noverify` de la JVM inhibe [2] la vérification du *byte-code* par la JVM. En effet, pour simplifier, il n'est pas demandé de générer dans le fichier `class` les informations nécessaires à la JVM pour vérifier le *byte-code*.

Le *byte-code* produit est équivalent à celui qui serait produit par `javac Exemple.java` :

```
class Exemple {
  public static int a = 5;
  public static int fact(int n) {
    if (n==0)
      return 1;
    else
      return n*fact(n-1);
  }
  public static void main(String args[]) {
    System.out.println(fact(a));
  }
}
```

Les fonctions ont été traduites par RISTRETTO en méthodes statiques, et les variables globales en champs statiques. Les membres de classe sont tous publics et statiques.

Il ne vous est pas demandé de générer le code source Java ci-dessus, mais directement son *byte-code*.

Votre compilateur devra faire les contrôles de bon sens, et prévenir en cas de problèmes : erreurs syntaxiques, de typage, etc.

2 Fichier `class`

La sortie du compilateur RISTRETTO est un fichier `class` exécutable par une machine virtuelle Java standard. Les auteurs de Java maintiennent deux documents de référence décrivant les deux parties de Java : son langage [1] et sa machine virtuelle [2]. Il existe une version de ces deux documents par version de Java. Comme nous n'utiliserons pas les spécificités des versions les plus récentes de Java, et que les différentes versions de Java ont une compatibilité ascendante, vous pouvez vous référer à n'importe quelle version, à condition qu'elle ne soit pas trop ancienne, et compatible avec la version de Java que vous utilisez quotidiennement. Le premier document [1] ne devrait pas vous être utile. Le second document [2] vous sera utile. Il contient en particulier la description complète du format d'un fichier `class`, et du *byte-code* de la JVM. Comme il est assez long, nous en résumons les grands principes ici. Cependant, il vous sera peut être nécessaire de vous y référer pour les détails. En particulier, vous n'aurez pas besoin d'utiliser toutes les instructions du *byte-code* (vous n'avez donc pas besoin de toutes les connaître), mais nous ne donnons pas ici toutes celles dont vous avez besoin : il faudra éventuellement les chercher dans [2].

2. En dehors du projet, il est fortement déconseillé d'utiliser cette option. En effet, inhiber la vérification de *byte-code* au chargement peut amener à des problèmes lors de son exécution, ce qui est une faille de sécurité.

2.1 Format d'un fichier `class`

Un fichier `class` contient des données qui peuvent être de plusieurs types :
`u1` : un entier non signé sur un octet (8 bits) ;
`u2` : un entier non signé sur deux octets (*big-endian*) ;
`u4` : un entier non signé sur quatre octets (*big-endian*).
Il est formé, dans l'ordre :

- d'un nombre magique [3] composé d'un `u4`, dont les quatre octets sont dans l'ordre et en hexadécimale : `0xCAFEBABE` ;
- de deux numéros de version, dits *mineur* et *majeur*, dans cet ordre, et tous les deux de type `u2`. Ils indiquent le numéro de version de format du fichier `class`, et permettent par exemple d'éviter à une JVM obsolète de charger une version de format d'un fichier `class` qu'elle ne peut pas comprendre parce qu'il est trop récent pour elle. Pour le projet, nous pourrons prendre par exemple `0x0000` pour mineur et `0x0034` pour majeur ;
- du nombre d'entrées de la section suivante (*constant-pool*), augmenté de 1, sous la forme d'un `u2` ;
- d'une zone appelée *constant-pool* (voir Section 2.2). Elle contient en particulier, sous la forme d'*entrées*, tous les noms (sous la forme de chaînes de caractères) et tous les types (sous la forme de chaînes de caractères) utilisés à l'intérieur du fichier `class` ;
- d'un `u2` contenant les droits d'accès de la classe. Pour le projet, nous utiliserons `ACC_PUBLIC` (`0x0001`) et `ACC_STATIC` (`0x0008`), qui peuvent être composés bit à bit ;
- de l'indice (`u2`) d'une entrée du *constant-pool* au format `Class_info` (voir Section 2.2.2) représentant cette classe (la classe de `this`) ;
- de l'indice (`u2`) d'une entrée du *constant-pool* au format `Class_info` représentant la super-classe de cette classe. Pour le projet, la super-classe de cette classe sera toujours `java.lang.Object` ;
- du nombre (`u2`) d'interfaces implantées par cette classe. Pour le projet, toujours `0x0000` ;
- pour chaque interface implantée par cette classe (aucune, donc, pour RISTRETTO), un indice (`u2`) d'une entrée du *constant-pool* au format `Class_info` représentant l'interface ;
- du nombre (`u2`) de champs contenus dans cette classe ;
- pour chaque champ contenu dans cette classe, une entrée de type `field_info` (voir Section 2.3.1) ;
- du nombre (`u2`) de méthodes (les constructeurs et initialisateurs sont comptés comme des méthodes) contenues dans cette classe ;
- pour chaque méthode contenue dans cette classe, une entrée de type `method_info` (voir Section 2.3.2) ;
- du nombre (`u2`) d'attributs (on en voit un exemple dans la Section 2.5.2 ; pour le projet on pourra ne pas en mettre, dans ce cas leur nombre est 0) contenus dans cette classe ;
- une description de chacun des attributs, s'il y en a.

2.2 Le *constant-pool*

2.2.1 Les descripteurs de types

Ce sont des chaînes de caractères décrivant les types. Dans ces chaînes :

- `I`, `Z`, `F` et `V` représentent respectivement `int`, `boolean`, `float` et `void` ;
- les tableaux commencent par `[`, suivi du descripteur de type de chaque élément du tableau. Par exemple, `[I` représente le type « tableau d'`int` » ;
- les références commencent par `L` suivi du descripteur de type de l'objet référencé, suivi de `;`. Par exemple, un tableau de références sur des `java.lang.Object` est décrit par `[Ljava/lang/Object;`. On note que dans les noms de types, les points `.` des noms complètement qualifiés de classes sont remplacés par des slashs `/` ;
- les descripteurs des types pour les méthodes contiennent entre des parenthèses les types des arguments, dans l'ordre. Après la parenthèse fermante se trouve le descripteur du type de

3. Lisez-le à haute voix !

retour. Par exemple, `(II[[ZLjava/lang/String;)Ljava/lang/Integer;` décrit une fonction retournant une instance de `java.lang.Integer` et prenant en argument, dans l'ordre, deux `int`, un tableau bi-dimensionnel de booléens et une instance de `java.lang.String`.

2.2.2 Les différents types d'entrées

Toutes les entrées du *constant-pool* commencent par un `u1` qui identifie le type de l'entrée. Derrière cet identifiant viennent d'autres données dépendant du type de l'entrée. Voici les différents types d'entrées (et leurs identifiants) utiles pour RISTRETTO et les données qui les composent :

- `Utf8_info`
  - un identifiant `u1` de valeur `CONSTANT_Utf8` (`1`) ;
  - un `u2` contenant la longueur l de la chaîne ;
  - une suite de l `u1`, dont chacun code un caractère de la chaîne. On rappelle que par soucis de simplification, ces caractères ont un code se trouvant dans l'intervalle $[32 \dots 128[$. Notons que contrairement au C, il n'y a pas de caractère `\0` de fin ;
- `Class_info`
  - un identifiant `u1` de valeur `CONSTANT_Class` (`7`) ;
  - un `u2`, indice dans le *constant-pool* d'une entrée de type `Utf8_info` étant le nom complètement qualifié de la classe ;
- `NameAndType_info`
  - un identifiant `u1` de valeur `CONSTANT_NameAndType` (`12`) ;
  - un `u2`, indice dans le *constant-pool* d'une entrée de type `Utf8_info` contenant un nom ;
  - un `u2`, indice dans le *constant-pool* d'une entrée de type `Utf8_info` décrivant un type ;
- `Fieldref_info`
  - un identifiant `u1` de valeur `CONSTANT_Fieldref` (`9`) ;
  - un `u2`, indice dans le *constant-pool* d'une entrée de type `Class_info` représentant la classe contenant la déclaration du champ ;
  - un `u2`, indice dans le *constant-pool* d'une entrée de type `NameAndType_info` représentant le nom et le type du champ ;
- `Methodref_info`
  - un identifiant `u1` de valeur `CONSTANT_Methodref` (`10`) ;
  - un `u2`, indice dans le *constant-pool* d'une entrée de type `Class` représentant la classe contenant la déclaration de la méthode ;
  - un `u2`, indice dans le *constant-pool* d'une entrée de type `NameAndType_info` représentant le nom et le type de la méthode ;
- `String_info`
  - un identifiant `u1` de valeur `CONSTANT_String` (`8`) ;
  - un `u2`, indice dans le *constant-pool* d'une entrée de type `Utf8_info` étant le contenu de la chaîne ;
- `Integer_info`
  - un identifiant `u1` de valeur `CONSTANT_Integer` (`3`) ;
  - un `u4` étant la valeur de l'entier, codée par complément à deux, en *big-endian* ;
- `Float_info`
  - un identifiant `u1` de valeur `CONSTANT_Float` (`4`) ;
  - un `u4` étant la valeur du réel, codée en IEEE 754.

On note qu'il n'y a aucun type d'entrée pour les booléens. En effet, ils sont codés en *byte-code* Java par des entiers `int` (avec des conventions que vous choisirez).

2.3 Les membres de classe

2.3.1 Les champs

Chaque champ est codé par une structure nommée `field_info` formée des données suivantes, dans l'ordre :

— un `u2` contenant les droits d'accès du champ. Pour le projet, nous utiliserons `ACC_PUBLIC` (`0x0001`) et `ACC_STATIC` (`0x0008`), qui peuvent être composés bit à bit ;
— un `u2`, indice dans le *constant-pool* d'une entrée de type `Utf8_info` étant le nom du champ ;
— un `u2`, indice dans le *constant-pool* d'une entrée de type `Utf8_info` décrivant le type du champ ;
— un `u2`, nombre n d'attributs du champ (pour le projet, toujours 0 par soucis de simplification) ;
— une suite de n attributs (donc, aucun pour le projet).

2.3.2 Les méthodes

En Java, chaque méthode s'exécute dans son *cadre d'exécution* (*stackframe*), composé d'une zone de *variables locales* et d'une zone de *pile*. Un nouveau cadre d'exécution est créé à chaque appel de méthode ; il est détruit quand l'appel de méthode se termine. Le compilateur calcule le nombre de variables locales et la taille maximale de la pile pour que la méthode s'exécute. Les variables locales sont composées, dans l'ordre (numérotées à partir de 0) de la cible de l'appel de méthode (si elle n'est pas statique), de ses paramètres, et des variables locales déclarées dans la méthode.

Chaque méthode est codée par une structure nommée `method_info` similaire à `field_info`. Cette structure a toutefois un attribut qu'il faudra utiliser (appelé `Code_attribute`), décrivant le code de la méthode. Le format d'un attribut `Code_attribute` est le suivant :
— un `u2`, indice dans le *constant-pool* d'une entrée de type `Utf8-info` contenant le nom de l'attribut (toujours "Code") ;
— un `u4` contenant la taille totale, en octet, de ce `Code_attribute`, à laquelle on retire la taille de ses deux premières données (donc on retire 2+4=6) ;
— un `u2` contenant la taille maximale de la pile d'exécution de cette méthode ;
— un `u2` contenant le nombre de variables locales de cette méthode. Attention : les paramètres de la méthode sont comptés comme des variables locales, et, pour les méthodes d'instance (non statiques) la référence `this` aussi ;
— un `u4` contenant le nombre n d'octets du code de la méthode ;
— une suite de n octets représentant le *byte-code* du code de la méthode (nous renvoyons à [2] pour une description du *byte-code*) ;
— un `u2` contenant la taille t du tableau des exceptions gérées par la méthode. Dans Ristretto nous ne gérons pas les exceptions. Cette taille est donc toujours 0 ;
— une suite de t informations relatives aux exceptions. Pour Ristretto, t vaut 0, donc cette suite est vide ;
— un `u2` contenant le nombre m d'attributs du code de la méthode. Pour Ristretto, toujours 0 ;
— une suite de m attributs (donc, aucun pour Ristretto).

2.3.3 Les attributs du *constant-pool*

Ils ne nous intéressent pas dans le cadre de Ristretto.

2.4 Exemple

Nous donnons maintenant, dans l'ordre, toutes les informations composants un fichier `class` résultant de la compilation du fichier `Exemple.java`.

D'abord, les quatre octets composants le nombre magique (`0xCAFEBABE`), suivis du numéro de version mineur (`0x0000`), suivis du numéro de version majeur (`0x0034`). Vient ensuite le nombre d'entrées du *constant-pool*, augmenté de 1 (=36+1 en base 10, sur deux octets = `0x0025`).

Vient ensuite le *constant-pool* composé des 36 entrées suivantes, dans l'ordre (bien sûr, l'ordre peut changer, mais il faut alors bien faire attention à tout renuméroter) :

1. `Methodref_info` : (5,18) ;
2. `Methodref_info` : (4,19) ;
3. `Fieldref_info` : (4,20) ;
4. `Class_info` : 21 ;
5. `Class_info` : 22 ;
6. `Utf8_info` : "a" ;
7. `Utf8_info` : "I" ;
8. `Utf8_info` : "<init>" ;
9. `Utf8_info` : "()V" ;
10. `Utf8_info` : "Code" ;
11. `Utf8_info` : "LineNumberTable" ;
12. `Utf8_info` : "fact" ;
13. `Utf8_info` : "(I)I" ;
14. `Utf8_info` : "StackMapTable" ;
15. `Utf8_info` : "<clinit>" ;
16. `Utf8_info` : "SourceFile" ;
17. `Utf8_info` : Exemple.java" ;
18. `NameAndType_info` : (8,9) ;
19. `NameAndType_info` : (12,13) ;
20. `NameAndType_info` : (6,7) ;
21. `Utf8_info` : "Exemple" ;
22. `Utf8_info` : "java/lang/Object" ;
23. `Fieldref_info` : (27,28) ;
24. `Methodref_info` : (29,30) ;
25. `Utf8_info` : "main" ;
26. `Utf8_info` : "([Ljava/lang/String ;)V" ;
27. `Class_info` : 31 ;
28. `NameAndType_info` : (32,33) ;
29. `Class_info` : 34 ;
30. `NameAndType_info` : (35,36) ;
31. `Utf8_info` : "java/lang/System" ;
32. `Utf8_info` : "out" ;
33. `Utf8_info` : "Ljava/io/PrintStream ;" ;
34. `Utf8_info` : "java/io/PrintStream" ;
35. `Utf8_info` : "println" ;
36. `Utf8_info` : "(I)V".

Décrivons par exemple la première entrée. Elle donne des informations sur une méthode. La première information, qui se trouve à l'entrée 5, donne des informations sur la classe contenant la déclaration de la méthode. Le nom de la classe se trouve à l'entrée 22 : c'est `java.lang.Object`. Le reste des informations sur la méthode est localisé à l'entrée 18 : le nom est à l'entrée 8 (c'est "<init>". Ce nom est en fait celui du constructeur d'instanciation de la classe) et son type est décrit à l'entrée 9 : le constructeur ne prend rien en paramètre et ne retourne rien. Pour résumer, la première entrée du *constant-pool* référence le constructeur d'instance par défaut de `java.lang.Object`. Les entrées 11,14,16,17 ont été ajoutées par le compilateur pour aider à la vérification du *byte-code* et au débogage : elles pourraient être supprimées. Le nom "<clinit>" est celui du constructeur de classe, dont le rôle est par exemple l'initialisation des champs statiques de classe.

Après le *constant-pool* viennent les données des champs. D'abord, leur nombre, il n'y en a qu'un. Ensuite, les informations (`field_info`) de cet unique champ : ses droits (`ACC_PUBLIC | ACC_STATIC`), son nom (entrée 6), son type (entrée 7), son nombre d'attributs (0).

Enfin, les méthodes. D'abord, leur nombre. Les constructeurs étant considérés comme des méthodes dans les fichiers `class`, il y en a 4 (l'ordre n'a pas d'importance) :

- le constructeur d'instance par défaut `public init()`, dont le rôle est d'appeler le constructeur sans argument de la super-classe (`super()`) ;
- le constructeur de classe `static clinit()`, dont le rôle est l'initialisation des champs de classe (statiques) ;
- la méthode `public static int fact(int)` ;
- la méthode `public static void main(String args[])`.

Première méthode : le constructeur d'instance `init` par défaut La première donnée du `method_info` concerne les droits d'accès (`ACC_PUBLIC`). Ensuite, l'indice de son nom dans le *constant-pool* (8) et l'indice de son descripteur de type dans le *constant-pool* (9). Puis vient le nombre d'attributs de la méthode (1), et la description des attributs. La première information de l'attribut est un indice dans le *constant-pool* décrivant sa catégorie (10 : "Code"). Ensuite, sa longueur totale à laquelle on a retiré 6 (ici, 17). Puis, la taille maximale de la pile d'exécution du constructeur et son nombre de variables locales. Ce constructeur a été ajouté automatiquement

par le compilateur ; son rôle est simplement d'appeler le constructeur sans argument de la super-classe. Quand ce constructeur est appelé, et comme pour toute méthode d'instance (non statique) la référence `this` se trouve dans sa variable locale 0. Il n'y a pas de paramètres au constructeur, et aucune autre variable locale. Le nombre de variables locales est donc 1. Nous expliquerons la taille maximale de la pile (ici 1) après avoir vu et expliqué son *byte-code*. Après le nombre de variables locales vient la taille du *byte-code* : ici, 5 octets, puis les octets composant le *byte-code* du constructeur, qui est :

```
0: aload_0                      // 0x2a
1: invokespecial 0x0001         // 0xb7 0x00 0x01
4: return                       // 0xb1
```

La première instruction charge la variable locale numéro 0 (qui contient donc la référence `this`), sur le somme de pile. Il existe plusieurs instructions de chargement d'une variable locale vers la pile, de la forme p`load_`i, i variant entre 0 et 5. Le préfixe a signifie que la variable locale contient une référence, i pour un entier, f pour un flottant, etc. Pour chaque p`load_`i il existe une instruction réciproque p`store_`i. La taille de ces instructions est 1 octet. Le code de `aload_0` est `0x2a`. Quand il est nécessaire de manipuler des variables locales du numéro plus grand que 5, on peut utiliser les instructions de la forme p`load`, dont la taille est deux octets : le premier octet est le code de l'instruction, le second le numéro de la variable locale concernée, qui peut donc varier de 0 à 255.

La seconde instruction est de la forme `invoke`x. La taille de ces instructions est 3 ou 5 octets : le code de l'instruction (1 octet), et en fonction de l'instruction, un ou plusieurs arguments, le premier faisant deux octets. Le premier argument est un indice `u2` dans le *constant-pool* indiquant quelle est la méthode à appeler. Ici c'est `0x0001`, qui comme nous l'avons vu fait référence au constructeur d'instance par défaut de `java.lang.Object`, qui est la super-classe de `Exemple`. Les arguments de la méthode doivent au préalable avoir été empilés dans l'ordre. Si la méthode à appeler est une méthode d'instance, la référence à l'objet sur laquelle on l'appelle (la cible de l'appel) doit avoir été empilée avant le premier argument. En fait, la cible de l'appel de méthode est assimilable à un paramètre supplémentaire (le premier). Il existe 5 instructions de la forme `invoke`x :

- `invokespecial`, de code `0xb7`, est utilisée pour appeler les constructeurs ;
- `invokevirtual`, de code `0xb6`, est utilisée pour appeler les méthodes d'instances ;
- `invokestatic`, de code `0xb8`, est utilisée pour appeler les méthodes de classe (statiques) ;
- `invokeinterface`, de code `0xb9`, est utilisée pour appeler une méthode déclarée dans une interface ;
- `invokedynamic`, de code `0xba`, est utilisée pour appeler les méthodes d'instances.

Elles diffèrent entre autres par l'algorithme de résolution de méthode employé.

La troisième instruction de la méthode, `return`, de code `0xb1` et de taille 1, indique qu'il faut rendre la main à la fonction appelante, sans rien retourner. Il existe plusieurs instructions x`return` différentes, où x indique le type de ce qu'il faut retourner. Le préfixe x est analogue à celui des instructions de la famille x`load`. Par exemple, `ireturn` est utilisée quand on retourne un `int`, `freturn` pour un `float`, etc. Au retour de la fonction, les paramètres qui se trouvent sur la pile de l'appelant sont automatiquement supprimés, et si la fonction retourne une valeur, cette valeur (qui se trouve en sommet de la pile de la fonction qui exécute le x`return`) est mise automatiquement en sommet de pile de la fonction appelante avant qu'elle ne reprenne son exécution.

Pour résumer, la taille du code de la fonction est 5 et les octets la composant sont dans l'ordre `0x2a`, `0xb7`, `0x00`, `0x01`, `0xb1`. La seule instruction à manipuler la pile est la première, qui empile une référence, dont la taille est un `u4`. La taille maximale de la pile étant comptée en `u4`, c'est donc 1. Certaines instructions empilent/dépilent des données dont la taille est deux `u4`, par exemple celles qui empilent/dépilent des `long` ou des `double`. Pour celles-ci, il faut donc compter deux unités dans la taille de la pile. Une règle similaire s'applique pour le comptage des variables locales (par exemple, une variable locale de type `long` ou `double` compte pour deux).

Seconde méthode : la méthode `fact` La première donnée du `method_info` concerne les droits d'accès (`ACC_PUBLIC | ACC_STATIC`). Ensuite, l'indice de son nom dans le *constant-pool* (12) et l'indice de son descripteur de type dans le *constant-pool* (13). Puis vient le nombre d'attributs de la méthode (1), et la description des attributs. La première information de l'attribut est un indice dans le *constant-pool* décrivant sa catégorie (10 : "Code"). Ensuite, sa longueur totale à laquelle on a retiré 6 (ici, 27, la taille du *byte-code* étant 15 octets). Puis viennent la taille maximale de la pile (3) et le nombre de variables locales (1 : le paramètre de la méthode). Ensuite la taille du *byte-code* (15 octets) et la suite des octets le composant :

```
 0: iload_0               // 0x1a
 1: ifne 0x0005           // 0x9a 0x00 0x05
 4: iconst_1              // 0x04
 5: ireturn               // 0xac
 6: iload_0               // 0x1a
 7: iload_0               // 0x1a
 8: iconst_1              // 0x04
 9: isub                  // 0x64
10: invokestatic 0x0002   // 0xb8 0x00 0x02
13: imul                  // 0x68
14: ireturn               // 0xac
```

La première instruction copie la valeur de la première variable locale de la méthode sur le sommet de pile. Comme c'est une méthode statique, il s'agit de son premier (et unique) argument. L'instruction `ifne` retire la valeur en sommet de pile et la compare à 0. Si la comparaison n'est pas positive (ça n'est pas 0), on réalise un saut au déplacement relatif donné par les deux octets suivant l'instruction (ici `0x0005`). Comme le `ifne` est à l'adresse 1, le branchement est à l'adresse 1+5=6. Si la comparaison est positive (c'est 0), alors on continue sur l'instruction suivante (ici 4). Le déplacement relatif peut être négatif. Il existe beaucoup d'autres instructions de saut. L'instruction `iconst_1` met l'entier 1 en sommet de pile. Comme pour x`load`, il existe toute une famille d'instructions x`const`. L'instruction `ireturn` retourne l'entier en sommet de pile à la méthode appelante. Les instructions arithmétiques sur les entiers `isub` et `imul` dépilent deux entiers, réalisent l'opération et empilent l'entier résultat. Des instructions similaires existent pour les réels (par exemple, `fsub` réalise la différence des deux `float` en sommet de pile). Bien entendu, il existe des instructions pour les autres opérations arithmétiques.

Troisième méthode : le constructeur de classe `clinit` par défaut Comme les données composant les méthodes ont déjà été détaillées plus haut, nous passons directement au *byte-code*.

```
0: iconst_5          // 0x08
1: putstatic 0x0003  // 0xb3 0x00 0x03
4: return            // 0xb1
```

L'instruction `putstatic` déplace la valeur en sommet de pile vers le champ statique correspondant à l'entrée du *constant-pool* dont l'indice est donné est argument, et qui doit être un `Fieldref_info`. Ici il s'agit du champ `Exemple.a`. Ce constructeur initialise donc le champ `Exemple.a` avec la valeur 5.

Quatrième méthode : la méthode `main`

```
 0: getstatic 0x0023        // 0xb2 0x00 0x23
 3: getstatic 0x0003        // 0xb2 0x00 0x03
 6: invokestatic 0x0002     // 0xb8 0x00 0x02
 9: invokevirtual 0x0024    // 0xb6 0x00 0x24
12: return                  // 0xb1
```

La première instruction recopie le champ statique spécifié par l'entrée 23 du *constant-pool* en sommet de pile. Il s'agit du champ `java.lang.System.out`. La seconde instruction recopie le champ statique spécifié par l'entrée 3 du *constant-pool* (`Exemple.a`) en sommet de pile. La troisième appelle la méthode `fact` avec pour argument l'entier en sommet de pile, c'est-à-dire la valeur de `Exemple.a`. Au retour de l'appel, cet entier est dépilé, et remplacé par le résultat. La quatrième instruction appelle `java.io.PrintStream.println(int)`. Il s'agit d'une méthode d'instance. L'unique argument du `println` se trouve en sommet de pile : il s'agit du retour de `fact`. La cible de l'appel se trouve juste en dessous : il s'agit de `java.lang.System.out`.

2.5 Des outils pour aider

Nous donnons ici deux outils très utiles, par exemple, pour apprendre le format d'un fichier `class`, ou pour vérifier le contenu d'un fichier `class` généré par RISTRETTO.

2.5.1 javap

C'est l'outil d'examen des fichiers `class` fourni en standard avec Java. Exécuté avec suffisament d'options

```
% javap -c -s -v -p -l -constants Exemple.class
```

il produit sur sa sortie standard, au format texte, l'ensemble des données contenues dans le fichier.

2.5.2 JClasslib

JCLASSLIB (`https://github.com/ingokegel/jclasslib`) est un outil permettant d'explorer un fichier `class` avec une interface graphique. Une capture d'écran de JCLASSLIB exécuté sur le fichier `class` issu de `java Exemple.java` est donnée par la Figure 1. La figure montre en

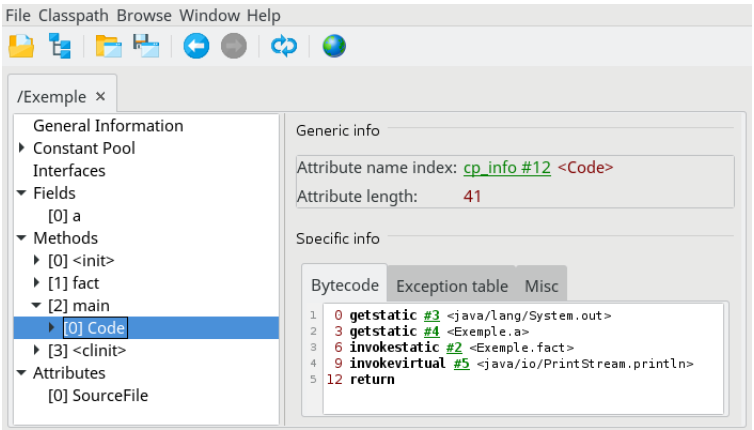

FIGURE 1 – L'utilitaire JCLASSLIB.

particulier que le compilateur a généré, dans le fichier `class`, un attribut `SourceFile`. Cet attribut est optionnel : il permet de spécifier le nom du fichier source ayant servi à produire le fichier `class` (ici, `Exemple.java`). Il est en particulier utilisé par le débogueur.

3 Ce qu'il vous est demandé

Vous devez écrire en C11, et en utilisant `flex` et `bison` pour les parties concernant les analyses lexicales et syntaxiques, un compilateur ayant les capacités présentées dans le sujet.

Bien entendu, votre projet devra être écrit le plus proprement possible : algorithmique adaptée, code clair et commenté.

Vous pouvez étendre le projet si vous le souhaitez, en rajoutant des fonctionnalités par exemple. Cependant, ne le faites que si la base qui vous est demandée est implantée et fonctionne correctement : il est préférable d'avoir un projet qui fait correctement le minimum plutôt que d'avoir un projet étendu dont le minimum demandé ne fonctionne pas.

Votre projet devra être rendu avec un jeu d'exemples illustrant le mieux possible ses fonctionnalités, ainsi qu'un rapport contenant un rapport de développement et un manuel d'utilisation.

Il devra être développé individuellement ou par binôme, et rendu au plus tard le *A FIXER* au soir, dans une archive au format `tar` gzippé de nom `FrancoisDupontJacquesDurant.tar.gz` pour un binôme (si Francois Dupont et Jacques Durant sont vos noms), envoyée en pièce jointe à un courriel de sujet « Projet de compilation L3 Info » à `Nicolas.Bedon@univ-rouen.fr` et `Arnaud.Lefebvre@univ-rouen.fr`). Vous vous mettrez en copie du courriel pour vérifier que vous n'oubliez pas la pièce jointe. L'extraction du fichier d'archive devra produire un répertoire de nom `FrancoisDupontJacquesDurant` contenant le code source de votre projet, un makefile, des jeux d'exemples et un rapport de projet.

Votre projet fera l'objet d'une soutenance sur machine. Il devra en particulier compiler et s'exécuter sans erreur et sans avertissement dans les salles de travaux pratiques.

Références

[1] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. The Java Language Specification. `https://docs.oracle.com/javase/specs/index.html`, 2018.

[2] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. The Java Virtual Machine Specification. `https://docs.oracle.com/javase/specs/index.html`, 2018.