Compilateur Java
troisième partie : production de code

à rendre avant le mardi 6 janvier 10 heures

Le but de cette dernière partie est de produire un code exécutable, étant donné un fichier Java et une fonction principale (voir la section 2 pour des détails sur ce dernier point).

Pour la sémantique de Java, on vous renvoie à vos propres connaissances de Java, aux tests que vous pourrez effectuer avec le compilateur Java et de façon plus formelle à l'ouvrage *The Java Language Specification* disponible en ligne à l'adresse `http://java.sun.com/docs/books/jls/`.

1 Production de code

1.1 Langage cible

La machine abstraite à pile servant de langage cible est une légère extension de la machine décrite et utilisée dans le cours. Elle est détaillée, ainsi que les interprètes qui en sont fournis, dans le document annexe *Machine virtuelle pour le projet de compilation.*

1.2 Organisation des données

1.2.1 Représentation des valeurs

Types primitifs. Les entiers et les flottants sont représentés de manière immédiate par les entiers et les flottants de la machine virtuelle.

Les booléens sont représentés par des entiers, avec la même convention que la machine virtuelle : 0 représente `false` et 1 représente `true`.

Objets. Un objet est représenté par une adresse, pointant sur un bloc alloué sur le tas. Ce bloc contient

- un *descripteur de classe* (voir ci-dessous) ;
- les valeurs des attributs *dynamiques* de l'objet (variables d'instance).

Les attributs dynamiques sont organisés de telle manière qu'un même attribut d'une classe C se trouve dans le bloc à la même position pour tout objet de la classe C ou de toute sous-classe de C (organisation dite en *préfixe*). Ceci est possible car l'héritage est *simple.* Ainsi, les deux classes `A` et `B` suivantes :

```
class A { int x; boolean b; }
class B extends A { double d; }
```

donneront lieu à des objets ayant la forme suivante :

descr. A
x
b

descr. B
x
b
d

Le compilateur maintiendra donc une table donnant, pour chaque classe C et chaque attribut dynamique de C, la position où trouver cet attribut dans un objet de la classe C.

La valeur `null`. Elle est représentée par une adresse sur le tas, différente de toute autre adresse d'objet. Le plus simple est de l'allouer dès le démarrage du programme.

Les classes prédéfinies `Objet` et `String`. Les objets de ces classes seront représentés exactement comme les autres, avec des descripteurs particuliers pour ces deux classes. Un objet de la classe `String` possédera en outre un champ contenant un pointeur vers la chaîne qu'il représente (pointeur vers le tas des chaînes de caractères).

1.2.2 Descripteurs de classes

Chaque classe est représentée par un descripteur de classe. Il sera par exemple alloué dans la zone des variables globales (c'est-à-dire au fond de la pile et référencé par GP). Il contient :
- le descripteur de sa super-classe ;
- la liste des codes des *méthodes dynamiques*.

Exactement comme pour la représentation des objets, la liste de ces codes obéit à une organisation préfixe : le code pour la méthode `f` doit se trouver au même endroit dans le descripteur de la classe `A` qui la définit et dans celui de la classe `B` qui la redéfinit. Ainsi le code suivant :

```
class A {
    int f() { return 0; }
    double g() { return 1.2; }
}
class B extends A {
    int f() { return 1; }
    boolean h() { return true; }
}
```

donnera lieu à des descripteurs de la forme suivante :

descr. de `A` :

descr. de `Object`
A_f
A_g

descr. de `B` :

descr. de `A`
B_f
A_g
B_h

De même que pour les attributs dynamiques, le compilateur maintiendra une table donnant pour chaque classe C et chaque méthode dynamique de C, la position où trouver le code de cette méthode dans le descripteur de la classe C.

1.2.3 Attributs et méthodes statiques

Les attributs statiques sont des variables globales, donc alloués au fond de la pile (au démarrage du programme) et référencés par GP. Le compilateur maintiendra une table indiquant, pour chaque classe C et chaque attribut statique de C, la position de cet attribut par rapport à GP.

Les méthodes statiques correspondent à des étiquettes de code déterminées au typage : la méthode statique `f` de la classe `A` correspondra directement à un code étiqueté `A_f`.

1.3 Schéma de compilation

1.3.1 Méthodes et constructeurs

Les méthodes et les constructeurs sont représentés par des fonctions (i.e. du code appelé par un `call`). L'appelant empile successivement

1. une place pour la valeur de retour (le cas échéant) ;
2. la valeur de l'objet (`this`) si la méthode est non statique ;
3. les arguments e_1, …, e_n.

Au retour, l'appelant se charge de dépiler ces valeurs (sauf la valeur de retour le cas échéant).

L'appelé se retrouve donc dans la situation (générale) suivante :

FP $\rightarrow$	
	e_n
	$\vdots$
	e_1
	`this`
	résultat

et il alloue ensuite au dessus de FP la place nécessaire aux variables locales de son code.

Constructeurs. L'appel à un constructeur nécessite au préalable une initialisation des attributs dynamiques. Le plus simple est de créer une fonction pour cela, que l'on appelle juste après avoir empilé l'objet et juste avant d'empiler les arguments et d'appeler le constructeur proprement dit.

1.3.2 Transtypage et `instanceof`

Les opérations de transtypage (*cast*) et `instanceof`, lorsqu'elles ne peuvent être résolues statiquement (au typage), doivent être effectuées dynamiquement. Pour cela on exploite le fait que les objets contiennent des descripteurs de classes, contenant eux-mêmes les descripteurs de leur super-classe. Ainsi on peut faire la vérification adéquate en suivant ce chaînage depuis la classe de l'objet jusqu'à la classe désirée. En cas de succès, le transtypage se contente de retourner (laissant l'objet inchangé) et `instanceof` retourne `true`. Si l'on parvient à la classe `Object` sans succès, alors le transtypage échoue (par exemple avec `err "cast failure"`) et `instanceof` retourne `false`. Le plus simple est d'écrire deux fonctions pour ces deux opérations.

Notez qu'il convient de tester si l'objet est `null` avant de démarrer cette vérification.

1.3.3 Ordre des opérations de compilation

On effectuera les différentes opérations nécessaires à la production de code dans l'ordre suivant :

1. Allocation de `null`.
2. Allocation des descripteurs de classe, des attributs statiques, et construction de la table décrivant les objets (indiquant où se trouvent les attributs dynamiques au sein des objets).

   Cette étape nécessite que les classes soient traitées dans un ordre compatible avec l'héritage (les classes les plus vieilles étant traitées les premières). Pour cela il suffit de suivre l'algorithme suivant :

   traiter(C) $\equiv$ si C a déjà été traitée, ne rien faire ;
   sinon,
   si C hérite de C', appeler traiter(C')
   construire le descripteur de C
   allouer les attributs statiques de C
   remplir la table des attributs dynamiques de C

   et d'appeler traiter(C) sur toutes les classes C du fichier. L'absence de cycle dans la relation d'héritage, vérifiée au typage, garantie la terminaison de cet algorithme.
3. Initialisation des attributs statiques.
4. Appel de la fonction principale.
5. Compilation de toutes les fonctions (code des constructeurs et des méthodes, des fonctions d'initialisation des objets, et des fonctions réalisant les opérations de transtypage et `instanceof`).

Au final, le code doit donc avoir la structure suivante :

allocation des globaux (`null`, descripteurs et attributs statiques)
`start`
initialisation des attributs statiques
appel de la fonction principale
`stop`
code des fonctions (constructeurs, méthodes, et fonctions primitives)

2 Travail demandé

Adapter votre compilateur pour qu'il effectue la production de code. Votre compilateur devra reconnaître les deux options existantes `-parse-only` et `-type-only` (cette dernière indiquant désormais de s'arrêter juste après la phase d'analyse sémantique), ainsi qu'une nouvelle option `-main classe.méthode` qui indique la méthode à exécuter. Cette méthode doit être une méthode statique sans argument (mais de valeur de retour quelconque). Par défaut, il s'agira de la méthode `main` de la classe `Main`.

En l'absence d'erreur lexicale, syntaxique ou sémantique, la production de code ne doit pas échouer, et le code doit être produit dans le fichier `fichier.vm` si le fichier source s'appelle `fichier.java`.

Tests avec un (vrai) compilateur Java. Il est possible de compiler et d'exécuter les fichiers sources du fragment de Java considéré ici avec un vrai compilateur Java. Pour cela, il suffit de rajouter une vraie méthode `main` pour Java, qui se contente d'appeler votre fonction principale. Le script `java-compil` installé au bâtiment 308 fait exactement ceci. On l'utilise de la manière suivante :

```
java-compil fichier.java classe.méthode
```

ce qui a pour effet de compiler le fichier avec `javac` et de l'exécuter avec `java` en cas de succès de la compilation.

Modalités de remise de votre projet. Votre projet doit se présenter sous forme d'une archive tar compressée (option "z" de tar), appelée *vos_noms*`.tgz` qui doit contenir un répertoire appelé *vos_noms* (exemple : `dupont-durand.tgz`). Dans ce répertoire doivent se trouver les *sources* de votre programme (ne donnez pas les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé `jc`. La commande `make clean` doit effacer tous les fichiers que `make` a engendré et ne laisser dans le répertoire que les fichiers sources.

Votre projet est à rendre à votre chargé de TP par e-mail[1] au plus tard le **mardi 6 janvier avant 10h00**.

[1] `sylvain.conchon@lri.fr`, `jean-christophe.filliatre@lri.fr`, `julien.forest@lri.fr`, `clement.renard@inria.fr`, `julien.signoles@lri.fr`, `julien.narboux@inria.fr`