production du code cible

Stéphane Malandain - Techniques de compilation - hepia

22 mai 2019

Résumé

Ce document guide pour la production de code cible.

2.2.3.1 Design visiteur

L'arbre syntaxique abstrait est accompagné d'un design pattern *Visiteur*. Cette architecture permet de créer différentes classes qui parcourent l'arbre sans devoir modifier les classes de l'arbre en question.

Voici son diagramme de classes:

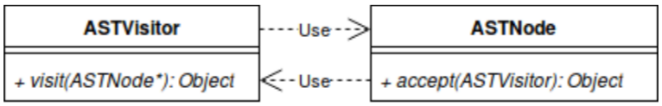

Figure 3: Diagramme de classes du visiteur de l'arbre syntaxique abstrait

Chaque nœud de l'arbre implémente une méthode nommée `accept(ASTVisitor)` avec comme paramètre le visiteur qui souhaite parcourir l'arbre. Le code de cette méthode est très simple: elle se contente d'appeler `visit(ASTNode*)` avec en paramètre le nœud de l'arbre en question, permettant au visiteur d'exécuter du code pour le type du nœud en question.

En effet, la classe *ASTVisitor* implémente une méthode `visit` par type de nœud que l'on souhaite parcourir. C'est pour cette raison qu'un astérisque est présent à côté de `ASTNode` dans la définition de `visit`, il indique qu'il existe une méthode par type. En théorie, tous les types de nœud doivent être implémentés pour que cela fonctionne.

Pour qu'un parcours de l'arbre soit possible, il faut que la méthode `visit` appelle `accept` des nœuds sous-jacents visités.

2.5 Générateur de bytecode

Le générateur de bytecode suit le design *visiteur* (voir: 2.2.3.1). Il est implémenté dans une seule classe `ByteCodeGenerator` qui implémente `visit(ASTNode*)` pour tous les types instanciables de l'arbre.

Cette classe va convertir chaque noeud de l'AST dans du code assembleur Jasmin convertissable directement en byte-code exécutable par la JVM.

La JVM utilise un context isolé pour chaque fonction appelée, nommé *stack frame*. Dans chaque *stack frame*, il existe:

- des variables locales à la fonction;
- une pile (*stack*) sur laquelle sont executées les diférentes instructions.

Toutes les opérations sont appliquées sur des valeurs en haut de la pile. La JVM a quelques particularités intéressantes: elle comprend le concept d'object ainsi que celui des fonctions, elle offre l'accès à toute la librairie standard JAVA et est capable de gérer un nombre de variables arbitraire.

2.5.1 Constantes

Les valeurs constantes utilisées dans un programmes sont copiées dans la pile d'exécution afin d'être consommées par les opérations qui les utilisent. L'instruction utilisée pour mettre la valeur constante dans la pile est `ldc`, suivie de la valeur à mettre sur la pile. Les constantes HEPIAL `vrai` et `faux` sont symbolisées dans la pile par `1` et `0` respectivement.

2.5.2 Simples instructions

Bien d'instruction de notre langange HEPIAL sont de simples instructions qui s'appliquent sur 1 ou 2 valeurs. Des exemples sont les opérateurs binaires et unaires. Ces instructions sont converties en assembleur Jasmin en deux simples étapes:

1. on met sur la pile le ou les opérandes sur lesquels s'appliquent l'instruction.
   - si l'opérateur est binaire, alors on ajoute d'abord l'opérande de gauche et ensuite celui de droite.
2. on met sur la pile l'instruction de l'opération qui va s'appliquer sur les valeurs du haut de la pile.

Après l'exécution des étapes ci-dessus, la pile ne contient qu'un seul élément: le résultat de l'opération.

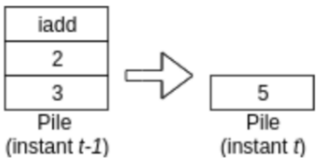

Figure 4: Exemple d'opération binaire d'addition sur la pile

Voici le code Jasmin correspondant:

```
; Code jasmin pour l'expression HEPIAL 2 + 3
ldc 3 ; met l'opérande de gauche dans la pile
ldc 2 ; met l'opérande de droite dans la pile
iadd ; applique l'addition sur les deux valeurs en haut de la pile
```

Le tableau suivant donne la correspondance entre l'instruction HEPIAL et l'instruction Jasmin:

Instruction HEPIAL	Instruction Jasmin
Addition	`iadd`
Division	`idiv`
Produit	`imul`
Soustraction	`isub`
Moins	`ineg`
Et	`iand`
Ou	`ior`
Tilda*	`ixor`

*Le *tilda* est un cas particulier. Bien qu'il s'agisse d'un opérateur unaire dans le language HEPIAL, on utilise l'opérateur binaire Jasmin. Dans ce cas, on mettra dans la pile la valeur constante `-1` avant d'ajouter l'instruction `ixor`.

L'instruction binaire `non` fut implémentée avec une méthode différente, expliquée plus en détail dans le chapitre suivant.

Les parenthèses ne font l'object d'aucun traitement dans la génération de l'assembleur. Leurs présences dans un code HEPIAL conditionnent la construction de l'AST, de manière à ce que l'évaluation des expressions respecte l'ordre qu'elles imposent.

2.5.3 Comparaisons binaires

Dans ce chapitre, l'implémentation des comparateurs est décrite. Le résultat de l'exécution d'une de ces instructions laisse dans la pile la valeur booléenne `0` ou `1`.

Toutes les comparaisons se font selon le modèle de code ci-dessous. On suppose que les valeurs à comparer se trouvent déjà dans la pile.

```
; Exemple de code testant l'égalité de deux valeurs en haut de la pile
if_icmpeq label_0 ; si vrai, saute vers label_0 (ligne 5)
iconst_0 ; condition fausse, met 'faux' dans la pile
goto label_1 ; saute vers la fin de la condition (ligne 7)
label_0:
iconst_1 ; condition vraie, met 'vrai' dans la pile
label_1: ; fin de la condition
```

L'instruction de comparaison, ci-dessus `if_icmpeq`, va s'appliquer sur les valeurs de la pile et, si vrai, on va sauter (`goto`) vers le label spécifié, sinon on continue dans la prochaine ligne. Si le saut a lieu, alors on charge sur la pile la valeur `1` (*vrai*), sinon on charge sur la pile la valeur `0` (*faux*).

Dans d'autres languages, cela correspond à l'opérateur ternaire:

```python
# example en python
1 if comparaison else 0
```

Les labels utilisés (dans l'exemple `label_0` et `label_1`) sont des valeurs arbitraires données par une séquence qui assure leur unicité. Le modèle ci-dessus pour la comparaison binaire est réutilisé pour tous les opérateurs.

Le tableau suivant donne la correspondance entre l'opérateur HEPIAL et l'instruction Jasmin:

Comparaison HEPIAL	Instruction Jasmin
Egal `==`	`if_icmpeq`
Diff `!=`	`if_icmpne`
Superieur `>`	`if_icmpgt`
SupEgal `>=`	`if_icmpge`
Inferieur `<`	`if_icmplt`
InfEgal `<=`	`if_icmple`
Non* `non`	`ifeq`

*Le `non` n'est pas une une instruction de comparaison mais son implémentation l'est.

2.5.4 Conditions

L'instruction `si [cond] alors [instructions] sinon [instructions] finsi` a été implémentée de manière similaire aux comparaisons du chapitre précédent.

On suppose que la condition (`[cond]`) a déjà été évaluée et que la pile contient au sommet la valeur `1` ou `0` selon le résultat de la condition. L'implémentation de l'instruction `si` est faite selon le modèle suivant:

```
1  ; Exemple d'exécution de code conditionel
2  ifeq label_0 ; si la condition (valeur sur la pile) est fausse, va vers label_0
3  ; ici liste d'instruction du le bloc 'alors'
4  goto label_1 ; saute vers la fin de la condition (ligne 7)
5  label_0:
6  ; ici liste d'instruction du bloc 'sinon'
7  label_1: ; fin de la condition
```

2.5.5 Boucles

Les boucles implémentées utilisent des instructions `goto` pour boucler, tout en réévaluant la condition de sortie à chaque tour.

2.5.5.1 Tantque

Exemple de boucle `tantque` en langage HEPIAL:

```
1  tantque [cond] faire
2      // [instructions]
```

```
fintantque
```

Ce qui donne les instructions Jasmin suivantes:

```
label_loop:
; instructions de condition ([cond])
ifeq goto label_end ; si condition fausse (== 0) alors sortie
; corps de la boucle ([instructions])
goto label_loop ; on recommance
label_end: ; fin de la boucle
```

2.5.5.2 Pour

La boucle `pour` à la particuliarité de maintenir une variable compteur.

Exemple de boucle `pour` en langage HEPIAL:

```
pour x allantde 0 a 10 faire
    // [instructions]
finpour
```

Ce qui donne les instructions Jasmin suivantes:

```
ldc 0 ; on met sur la pile la valeur initial du compteur
istore 0 ; qu'on assigne à la variable du compteur
label_loop: ; début de la boucle
ldc 10 ; on met sur la pile la valeur final du compteur
iload 0 ; ; ainsi que la valeur du compteur
if_icmplt label_end ; si égalité, on finit la boucle
; corps de la boucle ([instructions])
iload 0 ; on met le compteur en mémoire
ldc 1 ; ainsi que la constante 1
iadd ; et on additione le deux
istore 0 ; et on met à jour la valeur du compteur
goto label_loop ; et on recomence
label_end: ; fin de la boucle
```

2.5.6 Variables

2.5.6.1 Déclaration

Toutes les variables dans le language HEPIAL sont déclarées comme des variables privées.

Les variables déclarées au niveau du programme sont considérées comme des variables privées de la méthode principale du programme (cf 2.5.8).

Voici un exemple de code HEPIAL:

```
entier nb;
booleen b;
```

Et voici le code Jasmin correspondant:

```
.var 0 is nb I ; déclare la variable 'nb' à l'index 0
.var 1 is b Z  ; déclare la variable 'b' à l'index 1
```

Chaque déclaration d'une variable commence par l'instruction `.var`, suivie de l'index de la variable, l'instruction `is`, le nom de la variable et le type (`I` pour des entiers et `Z` pour des booléens).

Les instructions Jasmin pour lire et écrire des variables référencent la variable par son index. L'index des variables commence à 0 et les paramètres de la fonction prennent les premiers index.

Toutes les variables scalaires locales sont initialisées car la JVM nous empêche de lire une variable non initialisée. Toutes les variables, entières et booléennes, sont initialisées à *0* (*false*) sauf pour les constantes qui sont initialisées à leur valeur.

2.5.6.2 Lecture

Pour charger sur la pile la valeur d'une variable locale de type entier ou booléen, nous utilisons l'instruction `iload` suivie de l'index de la variable concernée.

Pour charger des variables de référence, nous utilisons `astore`.

Exemple:

```
iload 0 ; met la valeur de la variable 'nb' sur la pile
```

2.5.6.3 Affectation

Pour affecter la valeur du sommet de pile sur une variable locale, nous utilisons l'instruction `istore` pour des entiers ou des booléens ou `astore` pour les références, suivie de l'index de la variable concernée.

Exemple:

```
istore 0 ; affectation de la valeur sur la pile à la variable 'nb'
```

2.5.7 La déclaration de fonctions

Comme dans Java, le code exécutable du programme Jasmin doit se trouver dans des méthodes de classe. Pour des raisons pragmatiques, le choix a été fait de n'utiliser que des fonctions statiques.

Regardons un exemple en détail:

```
// fonction HEPIAL
entier carre(entier i, entier y, entier z)
    entier test;
    debutfonc
        // [instructions]
        retourne i;
    finfonc
```

Donne le code Jasmin suivant:

```
.method public static carre(III)I
.limit stack 20000
```

```
3  .limit locals 5
4  .var 3 is test I
5  ldc 0
6  istore 3
7  ; [instructions]
8  iload 0
9  exit_label:
10 ireturn
11 .end method
```

Regardons les lignes en détail:

1. Contient la signature de la fonction. La partie initiale (`public static carre`) donne le nom et qualifiants de la fonction. Chaque caractère entre parenthèses correspond à un paramètre de la méthode et le dernier caractère de la ligne correpond au type de retour de la fonction.
2. Définit la taille de la pile de la fonction. Pour des raisons pragmatiques, nous avons fixé la valeur à 20000, mais il est probablement possible de faire une analyse des instructions assembleur de la fonction afin d'estimer une taille maximale de pile plus précisément.
3. Spécifie le nombre de variables. La valeur est égale à *nombre de paramètres* + *variables locales* + 1. Dans notre exemple, nous avons 3 paramètres et 1 variable locale, ce qui donne la valeur 5. On rajoute *+1* car on se réserve une variable temporaire interne pour certaines opérations telles que l'affectation à des tableaux.
4. Suit une liste des variables locales avec leur index. Notre variable locale a l'index 3 car les paramètres prennent les premiers index. Chaque déclaration de variable à un index (*3*), un nom (*test*) et leur type (*I*).
5. Première étape de l'initialisation de la variable: on met sur la pile la valeur initiale de *0*
6. La valeur de *0* dans la pile est stockée dans la variable *test* (index *3*)
7. Suit le corps de la fonction
8. La valeur de retour est chargée sur la pile
9. Ce label, rajouté manuellement, est utilisé pour sortir prématurément de la fonction car il ne peut y avoir qu'une seule intruction "return" dans une fonction et elle doit se retrouver à la fin de la fonction
10. Instructon de *return*, de type entier (***i***return)
11. Marqueur de fin de fonction

2.5.8 Déclaration de programme

La déclaration d'un programme prend la forme d'une classe, où les instructions de la fonction principale HEPIAL sont mises dans une méthode statique de la classe appelée `main`.

Regardons un exemple:

```
1 programme Program
2
3 debutprg
4 // [instructions]
5 finprg
```

Donne le code Jasmin suivant:

```
1 .class public Program
2 .super java/lang/Object
3 .method public static main([Ljava/lang/String;)V
4 .limit stack 20000
5 .limit locals 1
6 return
7 .end method
```

La ligne 1 déclare la classe, nommée selon le nom du programme HEPIAL. La deuxième ligne déclare la classe mère de notre classe. Les lignes suivantes déclarent la fonction `main`, contruite comme toutes les autres fonctions décrites dans le chapitre précédent.

2.5.9 Appel de fonction

Nos fonctions HEPIAL sont toujours implémentées comme des méthodes statiques de classe.

L'appel particulier de ce type de méthode se fait à l'aide de l'instruction `invokestatic`, suivi du nom de la classe et de la signature de la méthode appelée.

Voici l'exemple pour l'appel de la fonction exemplifié dans le chapitre 2.5.7.

```
1 invokestatic Program/carre(III)I
```

2.5.10 Tableaux

2.5.10.1 Déclaration

La déclaration de tableaux se fait à l'aide de l'instruction `multianewarray`.

Cette instruction attend en paramètre autant de crochets ouvrants qu'il y a de dimensions, suivi du type (`I` pour entier, `Z` pour booléen), et du nombre de dimensions.

Des valeurs sont ensuite dépilées de la pile pour définir la taille de chaque dimension du tableau.

Voici un exemple à 1 dimension:

```
1 entier[1..5] arr; // tableau d'entiers à une dimention de taille 5
```

Le code Jasmin correspondant est le suivant:

```
1 ldc 5 ; on met sur la pile la taille du tableau
2 multianewarray [I 1 ; crée un tableau d'entier à 1 dimension
3 astore 0 ; met la référence du tableau sur la variable d'index 0
```

Et voici un exemple à 2 dimensions:

```
1 entier[3..5, 2..5] arr2; // tableau d'entiers à 2 dimentions de tailles 3 et 4
```

Le code Jasmin correspondant est le suivant:

```
1 ldc 3 ; on met sur la pile la taille de la première dimension
2 ldc 4 ; ainsi que la taille de la deuxième dimension
3 multianewarray [[I 2 ; crée un tableau d'entiers à 2 dimensions
4 astore 1 ; met la référence du tableau sur la variable d'index 1
```

A noter que le language HEPIAL supporte la définition de tableaux dont l'index ne commence pas à 0 contrairement à la JVM. Ce décalage doit être géré à chaque accès au tableau.

2.5.10.2 Lecture

Pour lire une valeur dans un tableau, on doit avoir sur la pile, de bas en haut:

1. l'adresse du tableau
2. l'indice de la dimension du tableau
3. si d'autres dimensions existent:
   (a) l'adresse de la dimension suivante avec `aaload`
   (b) on recommence à l'étape 2

Ensuite nous utilisons l'intruction `iaload` pour faire l'affectation.

Voici un exemple à 1 dimension:

```
a = arr[2] // on lit la valeur tableau pour assigner à une variable
```

Le code Jasmin correspondant est le suivant:

```
aload 0 ; on met sur la pile l'adresse du tableau
ldc 2 ;  on met sur la pile l'index spécifié dans le code
ldc 1 ;  ainsi que l'index de début de la dimension
isub ; on soustrait pour calculer l'index à 0
iaload ; et on met sur la pile la valeur du tableau à la position spécifiée
```

Et voici un exemple à 2 dimensions:

```
a = arr2[4][5] //
```

Le code Jasmin correspondant est le suivant:

```
aload 1 ; on met sur la pile l'adresse du tableau
ldc 4 ;  on met sur la pile l'index spécifié dans le code
ldc 3 ;  ainsi que l'index de début de la dimension
isub ; on soustrait pour calculer l'index à 0
aaload ; on charge sur la pile l'adresse de la prochain dimension
ldc 5 ; on met sur la pile l'index spécifié dans le code
ldc 2 ; ainsi que l'index de début de la dimension
isub ; on soustrait pour calculer l'index à 0
iaload ; et on met sur la pile la valeur du tableau à la position spécifiée
```

2.5.10.3 Affectation

Pour affecter une valeur dans un tableau, la pile doit contenir les mêmes éléments que durant la lecture (voir section 2.5.10.2), avec un élément supplémentaire qui est la valeur à affecter.

Ensuite nous utilisons l'intruction `iastore` pour faire l'affectation.

Le méchanisme générique de parcours de l'AST met la valeur à assigner d'abord sur la pile, ce qui est contraire à l'ordre des valeurs de la pile attendus enumérées ci-dessus. Pour corriger ceci, nous utilisons une variable temporaire locale afin de remmetre la valeur d'assignation dans la pile au bon

moment.

Voici un exemple à 1 dimension:

```
arr[1] = 2;
```

Le code Jasmin correspondant est le suivant:

```
ldc 2 ; valeur d'assignation est mise dans la pile
istore 2 ; on assigne cette valeur dans une variable temporaire
aload 0 ; on met sur la pile l'adresse du tableau
ldc 1 ; on met sur la pile l'index spécifié dans le code
ldc 1 ; ainsi que l'index de début de la dimension
isub ; on soustrait pour  calculer l'index à 0
iload 2; on remet sur la pile la valeur de la variable temporaire
iastore ; et on l'affecte au tableau
```

Et voici un exemple à 2 dimensions:

```
arr2[4][5] = 4;
```

Le code Jasmin correspondant est le suivant:

```
ldc 4 ; valeur d'assignation est mise dans la pile
istore 2 ; on assigne cette valeur dans une variable temporaire
aload 1 ; on met sur la pile l'adresse du tableau
ldc 4 ; on met sur la pile l'index spécifié dans le code
ldc 3 ; ainsi que l'index de début de la dimension
isub ; on soustrait pour calculer l'index à 0
aaload ; on charge sur la pile l'adresse de la prochain dimension
ldc 5 ; on met sur la pile l'index spécifié dans le code
ldc 2 ; ainsi que l'index de début de la dimension
isub ; on soustrait pour calculer l'index à 0
iload 2; on remet sur la pile la valeur de la variable temporaire
iastore ; et on l'affecte au tableau
```

2.5.11 Appels à la libraire standard

Une des avantages de la JVM est de pouvoir accéder à toutes les fonctionnalités de la librairie standard. Nous utilisons ces fonctionalités dans l'implémentation des instructions `ecrire` et `lire`, décrites ci-dessous.

2.5.11.1 Ecrire

Pour `ecrire`, nous utilisons les méthodes java suivantes:

```
System.out.println(Integer):void // si l'on veut écrire un numéro
System.out.println(String):void  // si l'on veut écrire une chaine
```

Pour les booléens, on écrit les chaines ***`vrai`*** *ou* ***`faux`****.*

Voici le modèle du code Jasmin pour l'instruction `ecrire` dans le cas on l'on désire écrire une chaine

de caractères:

```
1 ; Ecriture d'une chaine sur la console
2 getstatic java/lang/System/out Ljava/io/PrintStream
3 ldc "test" ; met sur la pile la chaine d'exemple "test"
4 invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
```

L'instruction `getstatic`, à la ligne 2, met sur la pile la propriété statique `System.out` qui est de type `java.io.PrintStream`.

L'instruction `invokevirtual`, à la ligne 4, va exécuter la fonction `println`, de la classe `java.io.PrintStream`, qui accepte en paramètre une chaine de caractères. En haut de la pile se trouve le paramètre et en dessus l'instance de `java.io.PrintStream` sur laquelle appeler la fonction.

2.5.11.2 Lire

Pour la lecture, nous utilisons la classe `java.util.Scanner`, selon l'exemple en java:

```
1 java.util.Scanner scanner = new java.util.Scanner(System.in);
2 var nb = scanner.nextInt()
```

En assembleur Jasmin, nous avons:

```
1 new java/util/Scanner ; instanciation de java.util.Scanner
2 dup ; duplication du pointeur de l'instance sur la pile
3 getstatic java/lang/System/in Ljava/io/InputStream; on met System.in dans la pile
4 invokespecial java/util/Scanner/<init>(Ljava/io/InputStream;)V ;appel constructor
5 invokevirtual java/util/Scanner/nextInt()I ; appel de nextInt()
6 istore 0
```

Pour chaque ligne:

1. Nous créons une nouvelle instance de java.util.Scanner, et la référence est mise dans la pile.
2. On duplique la référence dans la pile avec `dup`. Elle sera utilisée une fois pour le contructeur et la deuxième pour l'appel de la fonction `nextInt`.
3. On recupère la propriété statique `System.in`, de type `java.io.InputStream`, que nous mettons dans la pile.
4. On appelle le contructeur (`<init>`) de l'instance du Scanner avec l'input stream en paramètre.
5. On appelle la fonction `nextInt()` du Scanner qui retournera un entier, assigné à une variable local en ligne 6.