Le but de ce TP est de générer du bytecode Java pour Ava, en parcourant par un visiteur l'arbre abstrait du TP4. Il est impératif d'avoir terminé le TP4 avant d'attaquer ce TP, et préférable d'avoir terminé le contrôle de type du TP5 (sinon la JVM se chargera à l'exécution de rejeter les programmes mal typés).

Merci à Yves Roos pour la version précédente de ce TP.

1 Description du matériel fourni

1.1 Matériel fourni

Copiez chez vous l'archive `tp6.tgz` qui est sur le portail. Elle contient des scripts, des sources, et la structure de répertoire à laquelle vous êtes maintenant habitués.

Le répertoire `src` contient tous les sources fournis jusqu'à présent, avec en plus un paquetage `ava.genCode` qui contient les classes :

- `GenerateurCode` : squelette de votre générateur à compléter ;
- `GenByteCode` : point d'accès à tout le matériel fourni pour la génération de bytecode Java (table des variables, des constantes, bytecode, etc). Les mnémoniques pour les instructions de bytecode sont décrits en annexe A.

Pour plus de détails, consulter la doc (`ant genDoc`).

Le résultat de la génération de code est double :

- affichage sur la sortie standard du contenu de la table des variables, de la table des constantes, des mnémoniques pour les instructions de bytecode générées (aide au débogage) ;
- pour la compilation d'un programme Ava d'entête `program "atester" ;` : écriture d'un fichier `atester.class` qui peut s'exécuter avec des JVM depuis la version 1.3 jusque la version 1.6 incluse.

Pour tester votre générateur de code, deux solutions complémentaires :

- comprendre le bytecode affiché (rapidement pénible) ;
- exécuter le `.class` (pas très informatif tant que les impressions n'ont pas été traitées). Si on veut éviter la multiplication des `.class`, il est préférable de choisir le même nom de programme pour tous les tests.

Les scripts habituels sont fournis, avec en plus deux scripts dédiés à la génération de code :

- `execEnLigneGenerateurCode.sh` lance l'analyse syntaxique, puis le contrôle de type puis le générateur de code en ligne ;
- `execSurFichierGenerateurCode.sh <fichier>` fait de même pour le fichier passé en ligne de commande.

1.2 Principes de fonctionnement / génération d'un `.class`

Voir aussi les transparents du cours.

Un `.class` est une suite de bytecodes (d'octets) découpés en sections.

La première section est une séquence d'octets particulières permettant à la JVM de vérifier la validité du `.class`. La méthode `GenByteCode.closeClass()`, qui écrit sur le disque le fichier `.class`, génère automatiquement cette section.

La seconde section contient la *table des constantes* utilisées dans le `.class` (chaînes de caractères, constantes entières, noms et profils des méthodes, etc). Cette table est gérée par index : une constante est référencée par son *index* (un entier) dans cette table. Pour ajouter une constante dans la table, deux méthodes dans `GenByteCode` retournent l'index de la constante ajoutée :

- `int newConstant(int entier)` ;
- `int newConstant(String chaine)`.

On notera que le type booléen n'existe pas au niveau du bytecode : il est codé par les entiers 0 (faux) et 1 (vrai).

La section suivante contient le code du corps des méthodes. Dans ce TP, la méthode `closeClass` génère juste un `main` correspondant aux instructions Ava contenues dans l'arbre abstrait.

Les méthodes possèdent généralement des variables locales. Dans le byte-code de la méthode, on référence une variable par son *index* (de type entier) dans une *table des variables*. Cette table n'est pas une table des symboles, mais juste une collection de couples (index, variable). `GenByteCode` offre trois méthodes pour cette table :

- `int newVarInt(String varName)` qui ajoute dans la table la variable `varName` de type entier et retourne son index ;
- `int newVarBool(String varName)` qui ajoute dans la table la variable `varName` de type booléen et retourne son index ;
- `int getIndex(String varName)` qui retourne l'index de la variable `varName` déjà dans la table.

Le code d'une méthode est une suite d'instructions rangées séquentiellement par adresse croissante. Une adresse est un *index* de type entier. Le code s'exécute grâce à une *pile d'exécution*. Les instructions de bytecode, référencées par un mnémonique (`iadd`, `ldc`, etc), sont de plusieurs type :

- chargement d'une valeur dans la pile, avec ou sans paramètre d'index de constante ou de variable ;
- stockage d'une valeur dans une variable, avec paramètre d'index de variable ;
- opérations arithmétiques, booléennes, sur pile ;
- manipulation de pile ;
- instructions de saut ;
- d'autres créées pour les besoins du TP ($\sim$ macros).

Leur liste est donnée en annexe A. À chaque instruction `X` correspond dans la classe `GenByteCode` une méthode `addX()`. La création d'une instance de `GenByteCode` crée automatiquement une séquence de code vide, auquel on ajoute des instructions de bytecode par ces méthodes `addX` de `GenByteCode`. Les exemples qui suivent illustrent les principales instructions dont vous aurez besoin par des exemples.

1.2.1 Chargement dans la pile, stockage dans une variable

Chargement dans la pile :

- `iconst_0` empile la constante 0 ;
- `iconst_1` empile la constante 1 ;
- `ldc <index>` empile la valeur de la constante de numéro `index` ;
- `iload <index>` empile la valeur de la variable de numéro `index`.

Stockage dans une variable :

- `istore <index>` stocke le sommet de pile dans la variable d'index `index`, et le dépile.

Exemple : le bytecode généré pour `x := 4` si `x` a l'index 0 et `4` a l'index 1 sera : `ldc 1 ; istore 0 ;`

Pour générer ce bytecode, on utilise les méthodes de `GenByteCode` (`gen` est une instance de `GenByteCode`) : `gen.addLDC(1); gen.addISTORE(0).`

1.2.2 Opérations arithmétiques et booléennes

Les opérations binaires fonctionnent toutes de la même manière : elles dépilent le sommet de pile et le sous-sommet de pile et les remplacent par le résultat de l'opération. Par exemple :

- `iadd` additionne le sommet de pile et le sous-sommet de pile, remplace le sommet et le sous-sommet par le résultat de l'addition ;
- `iand` effectue le $\wedge$ bit à bit entre le sommet de pile et le sous-sommet de pile, remplace le sommet et le sous-sommet par le résultat du $\wedge$.

Exemple : le code généré pour `x := x + 4` si `x` a l'index 0 et `4` a l'index 1 sera `iload 0 ; ldc 1 ; iadd ; istore 0 ;`, grâce aux instructions Java

```
gen.addILOAD(0) ;  gen.addLDC(1) ;
gen.addIADD() ;  gen.addISTORE(0) ;
```

Les opérations unaires fonctionnent toutes de la même manière : elles dépilent le sommet de pile et le remplacent par le résultat de l'opération. Par exemple :

– `ineg` inverse le signe de l'entier en sommet de pile ;
– `inot`[1] effectue un non bit à bit sur l'entier en sommet de pile[2].

1.2.3 Entrées/sorties

Ces instructions ne sont pas des instructions machine Java :
– `iread` empile la valeur d'un entier lu sur l'entrée standard ;
– `iprint` (resp. `bprint`) affiche l'entier en sommet de pile sur la sortie standard, et le dépile ;
– `sprint` affiche la chaîne en sommet de pile[3] sur la sortie standard, et la dépile ;
– `println` effectue un saut de ligne sur la sortie standard.

Par exemple, pour lire et afficher le contenu de la variable `x` d'index 0, on générera `iread ; istore 0 ; iload 0 ; iprint ;`.

1.2.4 Instructions à saut

Par défaut, le compteur de programme (PC) suit linéairement la séquence d'instructions de byte-code (le compteur s'incrémente de la taille en octet du byte-code courant). Pour encoder des structures de contrôle, on utilise des instructions à saut, par exemple :
– `goto <index>` positionne le PC à l'instruction d'index `index` ;
– `ifeq <index>` positionne le PC à l'instruction d'index `index` si le sommet de pile vaut 0 ;

Exemple : pour `if b then writeln ; end if ; read x ;` on génère (les instructions de bytecode sont numérotées par leur index, `b` a l'index 1) :

```
18 - iload 1       on empile la valeur de b
20 - ifeq --> 30   si b vaut faux (0) on saute au read (30)
23 - println       sinon on fait le writeln
30 - iread         et on passe en séquence (pas de else)
33 - istore 0
```

Pour ajouter une instruction de saut au code courant, on procède en 2 temps : d'abord on ajoute l'instruction par `addX()` (par ex : `addIFEQ()`). Ensuite seulement on fixe la valeur de l'index de saut par la méthode de `GenByteCode` : `void setTarget(int from , int to)`. Elle fixe à `to` la valeur de saut associée à l'instruction de saut d'index `from`. Il est nécessaire de procéder en 2 temps car au moment où on génère le test (par ex `if_cmpge`), on n'a pas encore généré la suite du code : on ne sait pas où sauter.

Exemple : pour `if b then writeln ; end if ; read x ;`

```
18 - iload 1     on empile la valeur de b
20 - ifeq -->?   si b vaut faux (0) on saute où ?
```

À ce stade on mémorise l'index du test (ici 20), puis on génère le bloc correspondant à la partie `then`. À noter que `GenByteCode` ne permet pas de récupérer l'index de la dernière instruction ajoutée, mais seulement de la prochaine isntruction ajoutée (`getNextInstructionIndex()`). Donc, après avoir généré le `iload`, on mémorise *d'abord* le futur index du test `ifeq` par un appel à `getNextInstructionIndex()`, *puis* on ajoute le test.

Les instructions à saut permettent aussi d'encoder les comparaisons, par exemple :
– `if_cmplt <index>` (lt = lower than) positionne le PC à l'instruction de numéro `index` si le sous-sommet de pile est strictement inférieur au sommet de pile (*saut*), incrémente le PC sinon. Dépile le sommet et le sous-sommet de pile ;
– `if_icmpge <index>` (ge = greater or equal) : idem avec supérieur ou égal.

Si le résultat de la comparaison est vrai, le sommet de pile doit être 1 (vrai), 0 (faux) sinon. Pour empiler l'une ou l'autre valeur, on utilise une instruction à saut. Par exemple, pour `write(%b,3<4)`, on aura, en supposant que le 3 est à l'index 75 et le 4 à l'index 76 :

[1] `inot` n'est pas une instruction machine Java, c'est une macro créée pour les besoins du TP.
[2] Un booléen est codé par un entier.
[3] Pour une raison qui m'échappe, les `\n` et autres `\"` ne semblent pas interprétés par la JVM.

```
12 - ldc 75             on empile 3
14 - ldc 76             on empile 4
16 - if_icmpge --> 23   si 3 ≥ 4, on saute au iconst_0 (empiler faux)
19 - iconst_1           si on n'a pas sauté, cas où 3 < 4 : empiler vrai (1)
20 - goto --> 24        on saute à la fin
23 - iconst_0           si on a sauté, cas où 3 ≥ 4 : empiler faux (0)
24 - iprint             on imprime le sommet de pile
```

NB Noter l'utilisation du `if_cmpge` (greater or equal) alors que le code Ava contient un inférieur strict.

Pour générer ce code, on procède en deux temps, comme expliqué précédemment. Pour le code ci dessus :

```
int indexIf = this.gen.getNextInstructionIndex();   mémorisation index if
this.gen.addIF_ICMPNE();                            ajout du if, sans savoir où sauter
this.gen.addICONST_1();                             si vrai alors 1 dans la pile
                                                    et on saute à la fin (qu'on ne connait pas encore)
int indexGoto =
        this.gen.getNextInstructionIndex();         mémorisation index GOTO
this.gen.addGOTO();
this.gen.setTarget(indexIf,
             this.gen.getNextInstructionIndex());   si faux alors 0 dans la pile
                                                    et le if saute à cette instruction
this.gen.addICONST_0();
this.gen.setTarget(indexGoto,
           this.gen.getNextInstructionIndex());     le goto saute à la prochaine instruction
```

2 Travail à réaliser

Décompresser l'archive. Copier les spécifications du TP4 dans le répertoire `spec`, puis générer les analyseurs par `ant genAnLex` et `ant genAnSynt`. Copier aussi le contrôleur de type `ava.typeChecking.TypeChecker` dans le répertoire `src`, si vous l'avez réalisé.

Votre travail est maintenant de compléter la classe `genCode.GenerateurCode`. Le squelette fourni contient juste de quoi assurer la compilation de la classe principale : l'exécution n'affiche encore rien.

Commencer par ajouter ce qu'il faut pour que `GenerateurCode` soit un `Visiteur`. Ensuite compléter le corps des méthodes de manière itérative et incrémentale : coder un petit peu, compiler, tester en remplissant `test/OK` au fur et à mesure, en s'aidant de `execEnLigneGenerateurCode.sh` et `execSurFichierGenerateurCode.sh` pour la mise au point. Visiter un nœud de l'arbre revient à générer le bytecode pour la portion de programme qu'il représente et l'ajouter au code déjà généré. Les explications qui suivent suggèrent un ordre de mise en œuvre et devraient vous aider. `this.gen` fait référence à l'attribut de type `GenByteCode` de `GenerateurCode`, créé par la classe principage `ava.executeurs.LanceurGenerateurCode`.

Visite d'un `Programme` Le nom de la classe Java à générer a déjà été fixé par la classe principale (classe `LanceurGenerateurCode`), donc le nom du programme n'est pas utilisé ici. Il suffit de générer le code pour les listes de déclarations et d'instructions du programme (simple visite). Puis on appelle `gen.closeClass();` qui ajoute une instruction de retour `RET`, finalise le code, et écrit le `.class` sur le disque.

À tester : pour un programme ne contenant que `program "atester";` :

- il y a bien écriture sur le disque d'un fichier `atester.class`;
- `java atester` ne produit rien mais s'exécute sans erreur;
- le code généré ne contient que `ret`;
- les tables sont vides.

Visite d'une déclaration Lors de la visite on ajoute la variable dans la table des variables par `this.gen.newVar(...)`. Les variables d'Ava possèdent une valeur par défaut : 0 pour les entiers et faux (donc 0) pour les booléens. Il faut générer le code de cette initialisation pour chaque variable (`iconst_0` et `istore`). **À tester** : pour un programme `atester` ne contenant que des déclarations de variables :

- `java atester` ne produit rien mais s'exécute sans erreur ;
- le code généré ne contient que `ret` ;
- la table des constantes est vide ;
- la table des variables contient les variables déclarées ;
- le code généré effectue correctement les initialisations (vérifier les index).

Visite d'une impression de chaîne Lors de la visite on ajoute la chaîne dans la table des constantes par `this.gen.newConstant(...);`. Ensuite on génère le code pour charger la valeur de la chaîne dans la pile (`ldc`) et l'imprimer (`sprint`). **À tester** : pour un programme `atester` ne contenant que des impressions de chaîne :

- `java atester` effectue les affichages ;
- la table des constantes contient les chaînes ;
- la table des variables est vide.

Dans la foulée, traiter l'impression avec retour à la ligne et le `ImpressionSautDeLigne`.

Visite d'une expression réduite à un `Entier` ou un `IdentExpr`

Entier Comme pour les chaînes, on entre systématiquement les entiers dans la table des constantes. Puis on rend disponible la valeur de cet entier en l'empilant (`ldc`).

IdentExpr la variable a déjà été entrée dans la table des variables. On récupère son index par `this.gen.getIndex(...);` et on rend disponible sa valeur en l'empilant (`iload`).

Pour tester, le plus rapide est de traiter l'impression d'une expression entière.

Visite d'une impression d'entier On a besoin de connaître la valeur de l'expression pour la calculer. Visiter cette expression déclenchera la génération du code qui effectue ce calcul et rend la valeur disponible en la stockant en sommet de pile. C'est ce qu'on a fait au paragraphe précédent. Après avoir visité l'expression, il suffit d'ajouter au code un `iprint`. **À tester** : pour un programme `atester` de la forme :

```
program "atester";
int x;
write(%s,"Doit afficher 0 :"); write(%i,x);
```

`java atester` effectue le bon affichage.

Traiter de la même manière les constantes booléennes et les autres impressions.

Visite d'une expression binaire Le principe est toujours le même. Pour connaître la valeur des opérandes de l'expression, on visite l'opérande gauche puis l'opérande droite. L'exécution du code généré placera la valeur de gauche en sous-sommet de pile et la valeur de droite en sommet de pile. Il ne reste plus qu'à générer le code de l'opération (`imul`, `iand`, etc). Tester avec des impressions. Les expressions unaires suivent le même principe.

Visite d'une `Lecture` et d'une `Affectation`

Lecture On commence par générer le code qui lit une valeur au clavier et la place en sommet de pile.

Affectation On commence par visiter l'expression, ce qui a pour effet de générer le code qui la calcule et place sa valeur en sommet de pile.

Ensuite il ne reste plus qu'à stocker la valeur de sommet de pile dans la variable (`istore`).

Terminer et tester toutes les expressions simples (sans saut) avant de passer aux instructions à saut et aux comparaisons.

Visite d'une comparaison Comme pour n'importe quelle autre expression binaire, on commence par visiter les opérandes. Ensuite on procède comme indiqué en section 1.2.4.

Visite d'une `ConditionnelleAvecAlternative` On commence par visiter la condition : le code généré place un 0 dans la pile si la condition est fausse, un 1 sinon. Ensuite le test permet de passer à la partie alors en séquence (il faudra à la fin sauter à la fin de l'instruction) ou de sauter à la partie sinon (voir figure 1). Pour générer le code, il suffit d'alterner judicieusement instructions de saut, mémorisation d'index, visites, et positionnement d'index de saut.

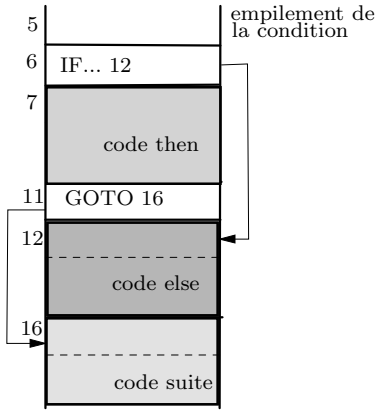

FIG. 1 – Conditionnelle avec alternative

La conditionnelle sans partie sinon est une simplification du cas précédent. La boucle while est ressemblante. Linéairement dans le code on trouve la condition, le test, le corps, et un saut à l'évaluation de la condition (dont il faut donc mémoriser l'index).

A Instructions de bytecode

Voici la liste des instructions utilisables. Dans la mesure du possible, ces noms correspondent à la mnémonique officielle d'assemblage du byte-code `java`.

Pour chaque instruction, il est précisé si un argument est attendu et ce que fait l'instruction. En particulier les modifications du contenu de la pile d'opérandes et/ou des variables sont données sous la forme *précondition* $\Longrightarrow$ *postcondition* où *précondition* décrit le contenu de la pile et/ou de variables nécessaires à la bonne exécution de l'instruction et où *postcondition* décrit le contenu de la pile et/ou de variables après l'exécution de l'instruction. Le contenu de le pile est donné sous la forme `pile :?`, v_n , v_{n-1} , ..., v_1 où v_1 représente la valeur en sommet de pile, v_2 la valeur du sous-sommet de pile, ...et où `?` représente une suite éventuellement vide quelconque de valeurs contenues sous la valeur v_n et qui reste inchangée après exécution de l'instruction. Les autres notations utilisées sont les suivantes :

- `Ival` : une valeur entière
- `Sval` : une valeur chaîne de caractères
- `val` : une valeur entière ou chaîne
- `<index>` : une valeur d'index argument d'une instruction
- `C<index>` : la constante de numéro index
- `V<index>` : la variable de numéro index

nop : instruction nulle, ne fait rien.

iconst_0 : empile la constante 0.
`(pile :?)` $\Longrightarrow$ `(pile :?,0)`

iconst_1 : empile la constante 1.
`(pile :?)` $\Longrightarrow$ `(pile :?,1)`

pop : dépile le sommet de pile.
`(pile : ? , val)` $\Longrightarrow$ `(pile : ?)`

dup : duplique le sommet de pile.
`(pile : ? , val)` $\Longrightarrow$ `(pile : ? , val , val)`

swap : échange sommet de pile et sous-sommet de pile.
`(pile : ? , val1 , val2)` $\Longrightarrow$ `(pile : ? , val2 , val1)`

iadd : addition entre les deux entiers en sommet de pile.
`(pile : ? , Ival1 , Ival2)` $\Longrightarrow$ `(pile : ? , Ival1 + Ival2)`

isub : soustraction entre les deux entiers en sommet de pile.
`(pile : ? , Ival1 , Ival2)` $\Longrightarrow$ `(pile : ? , Ival1 - Ival2)`

imul : multiplication entre les deux entiers en sommet de pile.
`(pile : ? , Ival1 , Ival2)` $\Longrightarrow$ `(pile : ? , Ival1 * Ival2)`

idiv : division entre les deux entiers en sommet de pile.
`(pile : ? , Ival1 , Ival2)` $\Longrightarrow$ `(pile : ? , Ival1 / Ival2)`

irem : modulo entre les deux entiers en sommet de pile.
`(pile : ? , Ival1 , Ival2)` $\Longrightarrow$ `(pile : ? , Ival1 rem Ival2)`

ineg : inversion du signe de l'entier en sommet de pile.
`(pile : ? , Ival)` $\Longrightarrow$ `(pile : ? , - Ival)`

iand : `et` bit à bit entre les deux entiers en sommet de pile.
`(pile : ? , Ival1 , Ival2)` $\Longrightarrow$ `(pile : ? , Ival1 et Ival2)`

ior : `ou` bit à bit entre les deux entiers en sommet de pile.
`(pile : ? , Ival1 , Ival2)` $\Longrightarrow$ `(pile : ? , Ival1 ou Ival2)`

ret : fin de méthode, la mnémonique réelle java est `return`.

Les instructions suivantes ne font pas partie des instructions machine en `Java`. Elles ont été créées pour votre TP et génèrent en fait des séquences d'appels de méthodes appartenant à certaines classes *standard* `Java`.

inot : `non` bit à bit sur l'entier en sommet de pile.
`(pile : ? , Ival)` $\Longrightarrow$ `(pile : ? , not Ival1)`

iread : empile un entier lu sur l'entrée standard.
`(pile : ?)` $\Longrightarrow$ `(pile : ? , Ival)`

iprint : affiche l'entier en sommet de pile sur la sortie standard.
`(pile : ?, Ival)` $\Longrightarrow$ `(pile : ?)`

bprint : affiche le booléen en sommet de pile sur la sortie standard.
`(pile : ?, Ival)` $\Longrightarrow$ `(pile : ?)`

sprint : affiche la chaîne en sommet de pile sur la sortie standard.
`(pile : ?, Sval)` $\Longrightarrow$ `(pile : ?)`

println : effectue un saut de ligne sur la sortie standard.

Les instructions suivantes sont des instructions qui attendent une valeur d'index en argument.

ldc : `ldc <index>` empile la valeur de la constante de numéro `<index>`.
`(pile : ? ; C<index> : val)` $\Longrightarrow$ `(pile : ?, val ; C<index> : val)`

iload : `iload <index>` empile le contenu de la variable de numéro `<index>`.
`(pile : ? ; V<index> : Ival)` $\Longrightarrow$ `(pile : ?, Ival ; V<index> : Ival)`

istore : `istore <index>` range le sommet de pile dans la variable de numéro `<index>`.
`(pile : ?, Ival ; V<index> : ?)` $\Longrightarrow$ `(pile : ? ; V<index> : Ival)`

Les instructions suivantes sont des instructions de branchement qui attendent une valeur de saut en argument. Elles effectuent le saut si et seulement si la condition indiquée est vraie.

ifeq :
`(pile :? , Ival)` $\Longrightarrow$ `(pile :?)`
condition : `Ival = 0`

ifne :
`(pile :? , Ival)` $\Longrightarrow$ `(pile :?)`
condition : `Ival /= 0`

if_icmpeq :
`(pile :? , Ival1 , Ival2)` $\Longrightarrow$ `(pile :?)`
condition : `Ival1 = Ival2`

if_icmpne :
`(pile :? , Ival1 , Ival2)` $\Longrightarrow$ `(pile :?)`
condition : `Ival1 /= Ival2`

if_icmplt :
`(pile :? , Ival1 , Ival2)` $\Longrightarrow$ `(pile :?)`
condition : `Ival1 < Ival2`

if_icmpge :
`(pile :? , Ival1 , Ival2)` $\Longrightarrow$ `(pile :?)`
condition : `Ival1 >= Ival2`

if_icmpgt :
`(pile :? , Ival1 , Ival2)` $\Longrightarrow$ `(pile :?)`
condition : `Ival1 > Ival2`

if_icmple :
`(pile :? , Ival1 , Ival2)` $\Longrightarrow$ `(pile :?)`
condition : `Ival1 <= Ival2`

goto :
condition : `true`