

Compilateur du langage *java--*

Daniel Côté et Richard St-Denis¹

Département d'informatique

Université de Sherbrooke

Étude de cas

IFT 580 — Compilation et interprétation des langages

Octobre 2006

¹Ce projet a été réalisé à partir d'une aide financière du *Fonds d'appui à la pédagogie universitaire*. Nous remercions l'Université de Sherbrooke pour son soutien dans ce projet.

Table des matières

1	Présentation globale du compilateur <i>java--</i>	1
1.1	Architecture du compilateur <i>java--</i>	1
1.2	Construction du compilateur <i>java--</i>	3
1.3	Compilation et exécution d'un programme écrit en <i>java--</i>	7
1.4	Norme d'écriture des spécifications	7
1.5	Exercices	8
2	L'analyseur syntaxique	9
2.1	Contenu du fichier <i>java--.prs</i>	9
2.2	Spécification des règles syntaxiques	9
2.3	Définition des attributs associés aux symboles	10
2.4	Construction de l'arbre syntaxique abstrait	11
2.4.1	Construction d'un noeud	12
2.4.2	Construction d'une liste chaînée de noeuds	13
2.5	Exercices	18
3	L'analyseur lexical	19
3.1	Spécification des règles lexicales	19
3.2	Calcul des attributs intrinsèques	19
3.3	Exercices	22
4	La machine virtuelle <i>Java</i>	23
4.1	Contenu d'un bloc d'activation	24
4.2	Taille d'un bloc d'activation	25
4.3	Bloc des variables locales	25
5	L'assembleur <i>Jasmin</i>	27
5.1	Organisation d'un programme <i>JasminXT</i>	27
5.1.1	En-tête d'un programme	29
5.1.2	Déclaration d'un champ	29
5.1.3	Définition d'une méthode	29
5.1.4	Énoncés	30
5.2	Assemblage et exécution d'un programme <i>JasminXT</i>	30
5.3	Instructions utiles	30
5.3.1	Instruction de branchement inconditionnel	31

5.3.2	Instructions de branchement conditionnel	31
5.3.3	Instructions de comparaison	33
5.3.4	Instructions arithmétiques	33
5.3.5	Instructions de chargement d'une constante	34
5.3.6	Instructions de chargement et de stockage d'une donnée locale	35
5.3.7	Instructions d'accès à un champ d'une classe	37
5.3.8	Instructions de transfert de contrôle	39
5.3.9	Instruction de création d'un tableau	40
5.3.10	Instruction de conversion de type	40
5.3.11	Instructions de manipulation de la pile de travail	41
5.4	Exercices	42
6	Le système de vérification et d'inférence de types	43
6.1	Fonctions et prédicats pour l'analyseur sémantique	43
6.2	Fonctions et prédicats pour le générateur de code	44
6.3	Exercice	44
7	Le générateur de code	45
7.1	Structure des schémas de génération de code	46
7.2	Organisation du générateur de code	47
7.3	Génération de code pour une unité de compilation	47
7.4	Génération de code pour les types	47
7.5	Génération de code pour les champs	48
7.6	Génération de code pour la méthode d'initialisation des champs statiques . .	48
7.7	Génération de code pour le constructeur par défaut	49
7.8	Génération de code pour l'appel de la méthode statique <code>main</code>	50
7.9	Génération de code pour les méthodes	50
7.10	Génération de code pour les énoncés	51
7.10.1	Énoncé de déclaration de variables locales	51
7.10.2	Énoncé de bloc	52
7.10.3	Énoncé <code>if-then</code>	52
7.10.4	Énoncé <code>if-then-else</code>	53
7.10.5	Énoncé <code>while</code>	53
7.10.6	Énoncé <code>return</code>	54
7.10.7	Énoncé d'affectation	55
7.10.8	Appel d'une méthode	55
7.10.9	Énoncés d'entrée/sortie	56
7.11	Génération de code pour le stockage d'une donnée	57
7.12	Génération de code pour les expressions	58
7.12.1	Règles pour les constantes	58
7.12.2	Règles pour un nom	59
7.12.3	Règle pour un appel de méthode	60
7.12.4	Règle pour la création d'un tableau	60
7.12.5	Règle pour un opérateur binaire	60
7.12.6	Règle pour un opérateur unaire	61

7.12.7 Règles pour les dimensions d'un tableau	61
7.12.8 Règle pour un paramètre actuel	61
7.13 Génération de code pour les expressions booléennes	62
7.13.1 Règle pour un terme booléen	62
7.13.2 Règles pour un opérateur binaire booléen	62
7.13.3 Règle pour un opérateur unaire booléen	63
7.14 Génération de code pour les branchements positifs	64
7.15 Génération de code pour les branchements négatifs	64
7.16 Génération de code pour les variables et les paramètres	64
7.17 Exercice	65
8 Le gestionnaire de la table des symboles	67
8.1 Définition des structures de données	67
8.1.1 Définition d'un environnement	67
8.1.2 Définition d'un <i>package</i>	68
8.1.3 Définition d'un type	68
8.1.4 Définition d'un membre	69
8.1.5 Définition d'une variable	69
8.2 Fonctions d'interrogation	70
8.3 Exercice	71
9 L'analyseur sémantique	73
9.1 Contenu du fichier <code>java--.cg</code>	73
9.2 Modules administratifs	73
9.3 Modules spécifiques à l'analyse sémantique	74
9.4 Modules spécifiques à la machine virtuelle <i>Java</i>	74
9.5 Déclaration des attributs	75
9.6 Calcul des attributs	75
9.7 Exercice	76
A Le programme principal	77
B Les fonctions de calcul des attributs intrinsèques	83
C La spécification de l'analyseur lexical	99
D La spécification de l'analyseur syntaxique	105
E La spécification de l'analyseur sémantique	117
E.1 Modules administratifs	117
E.2 Modules pour l'évaluation des attributs	118
F Les procédures de génération de code	147
G La spécification de la table des symboles	169

H	Les procédures relatives aux types	177
I	Le fichier Makefile	185
J	L'environnement d'exécution <i>java--</i>	191
K	Un sous-ensemble du langage JasminXT	195
K.1	Grammaire hors contexte	195
K.2	Quelques instructions	196
K.2.1	Instruction de branchement inconditionnel	196
K.2.2	Instructions de branchement conditionnel à un opérande de type référence	196
K.2.3	Instructions de branchement conditionnel à deux opérandes de type référence	196
K.2.4	Instructions de branchement conditionnel à un opérande entier	196
K.2.5	Instructions de branchement conditionnel à deux opérandes entiers	196
K.2.6	Instructions de comparaison	196
K.2.7	Instructions arithmétiques entières	197
K.2.8	Instructions arithmétiques en point flottant	197
K.2.9	Instructions de chargement d'une constante	197
K.2.10	Instructions de chargement de la valeur d'une donnée du bloc des variables locales	197
K.2.11	Instructions de stockage de la valeur d'une donnée dans le bloc des variables locales	197
K.2.12	Instructions de chargement de la valeur d'une composante d'un tableau	198
K.2.13	Instructions de stockage de la valeur d'une donnée dans une composante d'un tableau	198
K.2.14	Instructions d'accès à un champ de classe	198
K.2.15	Instructions d'appel d'une méthode	198
K.2.16	Instructions de retour d'une méthode	198
K.2.17	Instruction de création d'un tableau	198
K.2.18	Instruction de conversion de type	198
K.2.19	Instructions de manipulation de la pile de travail	199
	Bibliographie	200

Table des figures

1.1	Architecture générale du compilateur	2
1.2	Procédure pour la construction du compilateur	5
1.3	Compilation et exécution d'un programme <i>java--</i>	7
2.1	Création d'un noeud de l'arbre syntaxique abstrait	13
2.2	Structure arborescente versus liste chaînée	13
2.3	Partie de la machine caractéristique	14
2.4	Évolution de la machine caractéristique	15
2.5	Résultat de l'analyse syntaxique d'une suite de noms de champs	16
2.6	Inversion d'une liste de noeuds dans l'arbre syntaxique abstrait	18
4.1	Une machine virtuelle <i>Java</i>	23
4.2	Les piles <i>Java</i>	24
4.3	Exemple d'un bloc de variables locales	26
5.1	Programme <i>JasminXT</i> qui affiche <i>Merci 1000 fois !</i>	28
8.1	Liste des variables locales	70

Liste des tableaux

1.1	Modules de la librairie <i>cocktail</i>	4
1.2	Norme pour les noms des attributs et des symboles	7
2.1	Attributs des symboles terminaux	11

Chapitre 1

Présentation globale du compilateur *java--*

java-- Le langage *java--* est un sous-ensemble du langage *Java* [2]. Le compilateur du langage *java--* est construit à partir de la boîte à outils *cocktail* (pour *C*ompiler *C*ompiler *T*oolkit *K*Arlsruhe) [11, 10]. Son architecture est complexe même s'il est relativement petit. Il comporte plusieurs modules ayant de nombreux liens entre eux. C'est cela qui rend sa compréhension un peu laborieuse pour une personne non initiée à telle approche de construction de compilateurs.

1.1 Architecture du compilateur *java--*

La figure 1.1 présente l'architecture du compilateur *java--*. Dans cette figure, les boîtes de couleur jaune sont des modules de la librairie *cocktail*. La boîte de couleur magenta représente quatre programmes *C* pour le calcul des attributs associés aux constantes en point flottant, aux constantes entières, aux caractères et aux chaînes de caractères. Les boîtes de couleur verte sont des modules générés par les outils de *cocktail* à partir de spécifications de haut niveau qui contiennent, entre autres, les règles lexicales, les règles syntaxiques et les règles sémantiques du langage *java--*, ainsi que les procédures de génération de code pour la machine virtuelle java. La boîte de couleur bleue est un programme *C* qui représente le programme principal.

Il y a sept modules générés par les outils de *cocktail*. Dans la figure 1.1, chaque boîte verte possède trois cellules. La première indique le nom des fichiers générés (le « *.h* » et le « *.c* »). La deuxième contient le ou les noms des fichiers d'entrée nécessaires pour générer le module. La troisième donne le nom des outils de *cocktail* qui construisent le module à partir du ou des fichiers d'entrée. Voici une brève description de chacun de ces modules.

- L'analyseur lexical (**Scanner**) — Ce module effectue l'analyse lexicale d'un programme source par extraction une à une de ses unités lexicales commandée par l'analyseur syntaxique. Le fichier *java--.scn* contient les règles lexicales les plus complexes et le fichier *java--.prs* les symboles terminaux de la grammaire hors contexte à partir desquels des règles lexicales simples seront automatiquement produites. Les outils *lpp* [6], *rpp* [6] et *rex* [9] sont utiles dans la construction de l'analyseur lexical.

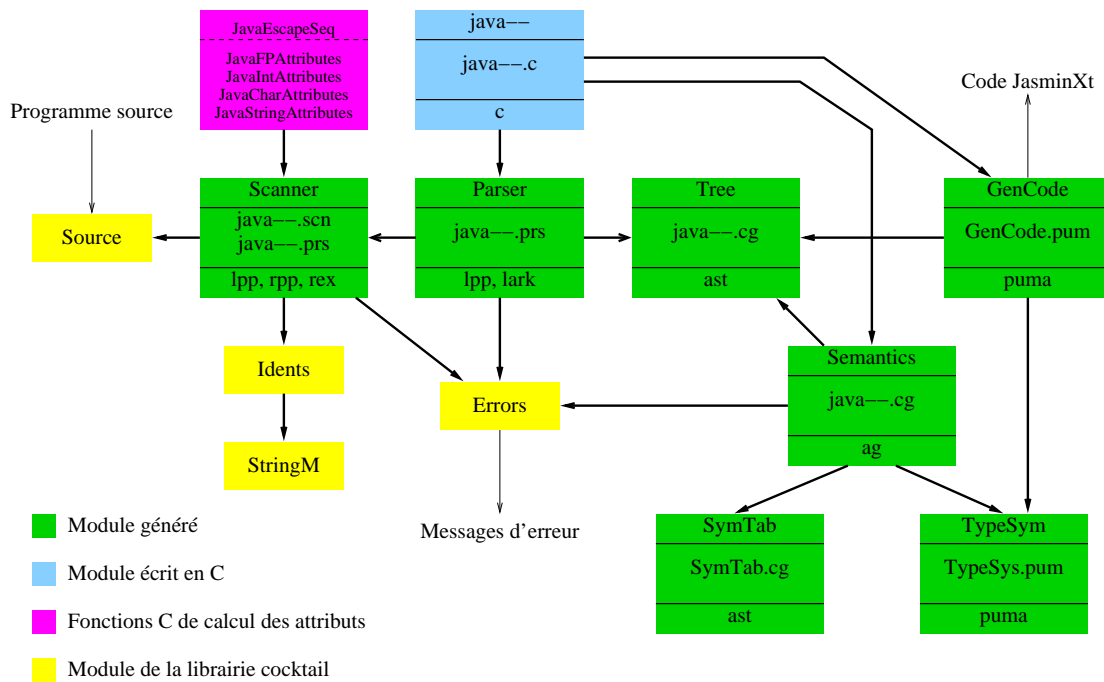


FIG. 1.1 – Architecture générale du compilateur

- L’analyseur syntaxique (**Parser**) — Ce module effectue l’analyse syntaxique d’un programme source à partir de règles syntaxiques. Il traduit aussi le programme source sous la forme d’un arbre syntaxique abstrait en appelant des fonctions du module **Tree**. Le fichier `java--.prs` contient les règles syntaxiques, c’est-à-dire la grammaire hors contexte (appelée aussi grammaire concrète) du langage *java--*. Les outils `lpp` [6] et `lark` [5] sont utiles dans la construction de l’analyseur syntaxique.
- Le constructeur de l’arbre syntaxique abstrait (**Tree**) — Ce module se présente sous la forme d’un ensemble de fonctions qui construisent les noeuds de l’arbre syntaxique abstrait. Ces fonctions, générées à partir d’une grammaire abstraite du langage *java--* contenue dans le fichier `java--.cg`, sont appelées par l’analyseur syntaxique qui relie les noeuds entre eux à l’aide de pointeurs pour former un arbre syntaxique abstrait. L’outil `ast` [4] est utile dans la construction de ce module.
- L’analyseur sémantique (**Semantics**) — Ce module effectue l’analyse sémantique à partir d’un certain nombre de parcours de l’arbre syntaxique abstrait. Le fichier `java--.cg` contient les règles sémantiques pour effectuer, par exemple, la vérification et l’inférence de types, l’organisation de l’espace des données ainsi que le traitement relatif aux énoncés de contrôle. L’outil `ag` [3] est utile dans la construction de l’analyseur sémantique.
- Le vérificateur de types (**TypeSys**) — Ce module, appelé par l’analyseur sémantique, effectue la vérification et l’inférence de types à l’aide de règles contenue dans le fichier `TypeSys.pum`. L’outil `puma` [7] est utile dans la construction de ce module.
- Le gestionnaire de la table des symboles (**SymTab**) — Ce module, appelé par l’analyseur sémantique, gère une table des symboles qui permet de tenir compte de la portée des symboles dans un programme *java--*. Le gestionnaire de table des symboles est donné

sous la forme d'une grammaire abstraite contenue dans le fichier `SymTab.cg`. L'outil `ast` [4] est utile dans la construction de ce module.

- Le générateur de code (**GenCode**) — Le générateur de code permet, comme son nom l'indique, de générer du code pour la machine virtuelle *Java* [14] à partir de procédures de génération de code contenues dans le fichier `GenCode.pum`. Ce module ne génère pas du code machine, mais un programme en langage d'assemblage **JasminXT** qui est traduit sous la forme d'un fichier « `.class` » par l'assembleur **Jasmin** [13]. L'outil `puma` [7] est utile dans la construction du générateur de code.

Le programme principal (`java--`), représenté par la boîte bleue dans la figure 1.1, spécifie l'ordre d'exécution des opérations à l'aide d'appels séquentiels aux trois principaux modules du compilateur, c'est-à-dire :

- l'appel à l'analyseur syntaxique — `Parser()` ;
- l'appel à l'analyseur sémantique — `Semantics(TreeRoot)`, où `TreeRoot` est un pointeur sur la racine de l'arbre syntaxique abstrait ;
- l'appel au générateur de code — `GenCode(TreeRoot)`.

Le programme principal est dans le fichier `java--.c`. L'annexe A contient une copie de ce fichier avec une description complète.

Il y a plusieurs modules dans la librairie *cocktail*. Seuls ceux qui sont pertinents à la compréhension du compilateur *java--* sont inclus dans la figure 1.1 (les boîtes jaunes).

- Le module **Source** effectue la lecture du fichier d'entrée qui contient le programme source.
- Le module **Errors** traite les messages d'erreur produit par l'analyseur lexical, l'analyseur syntaxique et l'analyseur sémantique.
- Le module **Idents** manipule une table de chaînes de caractères (souvent les identificateurs du programme source).
- Le module **StringM** gère des chaînes de caractères en mémoire.

Le tableau 1.1 contient la liste des principaux modules de la librairie *cocktail* [8] avec une brève description de leur traitement.

1.2 Construction du compilateur *java--*

La figure 1.2 montre plus en détail comment le compilateur *java--* est construit à l'aide de la boîte à outils *cocktail* à partir des différents fichiers de spécification. Cette approche de construction de compilateurs permet d'obtenir plus rapidement un prototype d'un compilateur, car même si le compilateur fait plus de 21 000 lignes de code *C*, les fichiers de spécification ne contiennent qu'environ 6 000 lignes écrites dans des langages de plus haut niveau que le langage de programmation *C*. Les programmes *C* originaux font environ 1 200 lignes de code.

Pour chacune des étapes de la construction du compilateur, numérotées dans la figure 1.2, voici la commande utilisée avec une explication des options retenues, s'il y a lieu, ainsi qu'une brève description.

1. `lpp -c -j -x -z java--.prs`
 - option `c` — générer du code *C* ;
 - option `j` — considérer les noeuds non définis comme des symboles terminaux ;

TAB. 1.1 – Modules de la librairie *cocktail*

Nom	Description
DynArray	Tableaux dynamiques et flexibles
Errors	Gestion des messages d'erreur
Idents	Table de chaînes de caractères
Position	Gestion des positions des lexèmes en entrée
Relation	Relations binaires entre des valeurs scalaires
ratc	Type booléen
rFsearch	Recherche de fichiers
rGetopt	Analyse des options d'une commande
rMemory	Gestion dynamique de la mémoire
rSrcMem	Mémorisation de code source
rString	Gestion portable de chaînes de caractères
rTime	Accès au temps de l'UCT
Sets	Ensembles de valeurs scalaires
Source	Lecture de fichiers sources
StringM	Mémoire de chaînes de caractères

- option **x** — créer le fichier **Scanner.rpp** ;
- option **z** — créer le fichier **Parser.lrk**.

Extraction des symboles terminaux à partir de la grammaire concrète (**java--.prs**) et écriture de ces symboles dans un fichier intermédiaire (**Scanner.rpp**). Extraction des règles syntaxiques sous la forme de règles de production dans la notation non étendue ainsi que des actions pour la construction de l'arbre syntaxique abstrait et écriture de ces règles et de ces actions dans le fichier **Parser.lrk**.

2. **rpp < java--.scn > java--.rex**

Fusion des règles lexicales simples (**Scanner.rpp**) et complexes (**java--.scn**), et écriture de ces règles dans le fichier **java--.rex**.

3. **rex -c -d java--.rex**

- option **c** — générer du code *C* ;
- option **d** — créer le fichier **Scanner.h**.

Création des fichiers sources de l'analyseur lexical (**Scanner.h** et **Scanner.c**) à partir de la spécification de toutes les règles lexicales (**java--.rex**).

4. **lark -c -d -i -v Parser.lrk**

- option **c** — générer du code *C*
- option **d** — créer le fichier **Parser.h** ;
- option **i** — créer le fichier **Parser.c** ;
- option **v** — expliquer les conflits *reduce/reduce* et *shift/reduce*.

Création des fichiers sources de l'analyseur syntaxique (**Parser.h** et **Parser.c**) à partir de la spécification des règles syntaxiques (**Parser.lrk**).

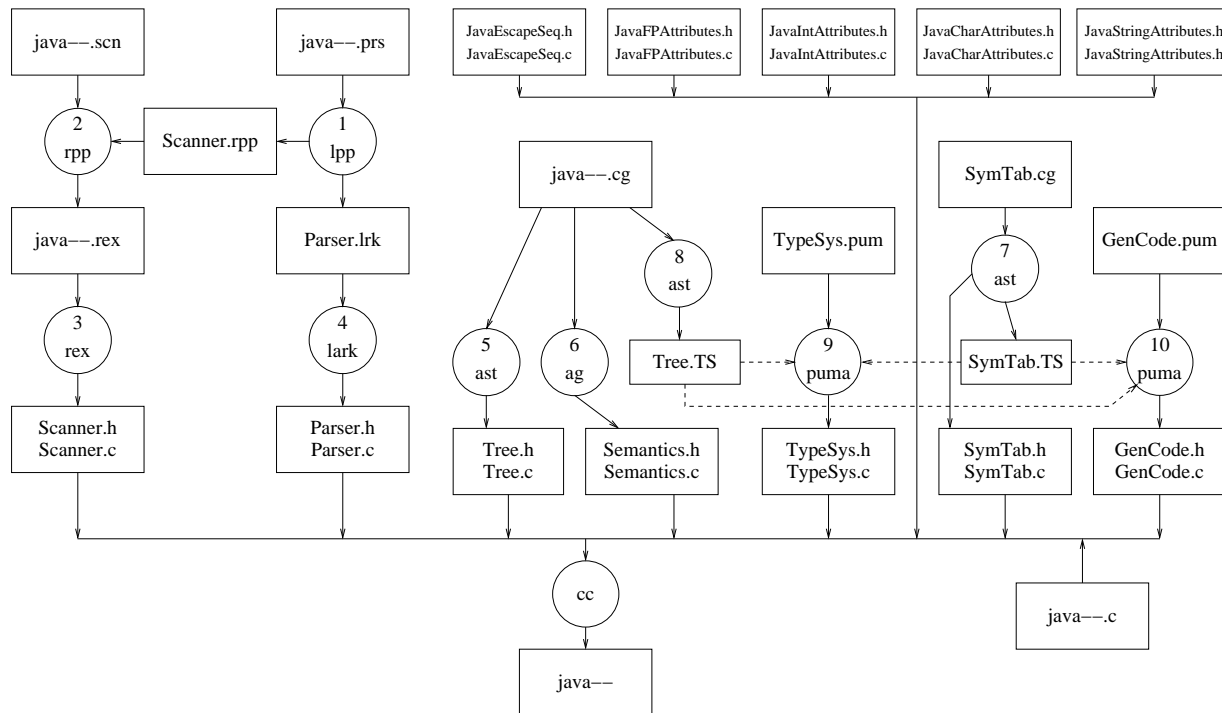


FIG. 1.2 – Procédure pour la construction du compilateur

5. `ast -c -d -i -m -w -R -= java--.cg`

- option `c` — générer du code `C` ;
- option `d` — créer le fichier `Tree.h` ;
- option `i` — créer le fichier `Tree.c` ;
- option `m` — générer les fonctions de construction des noeuds avec le préfixe `m` (pour *make*) ;
- option `w` — générer la fonction `WriteTree` ;
- option `R` — générer la fonction `ReverseTree` ;
- option `=` — générer la fonction `IsEqualTree`.

Création des fichiers sources du constructeur de l'arbre syntaxique abstrait (`Tree.h` et `Tree.c`), qui contiennent la définition des types de données pour les noeuds de l'arbre syntaxique abstrait ainsi que les fonctions de construction de ces noeuds, à partir de la grammaire abstraite (`java--.cg`).

6. `ag -c -D -I -0 -1 -2 java--.cg`

- option `c` — générer du code `C` ;
- option `D` — créer le fichier `Semantics.h` ;
- option `I` — créer le fichier `Semantics.c` ;
- option `0` — optimiser le stockage des attributs ;
- option `1` — afficher les règles de copie automatiquement insérées dans la grammaire attribuée (*Inserted Copy Rules*) ;
- option `2` — afficher les règles de calcul des attributs hérités (*Inherited Attribute Computation Rules*).

Création des fichiers sources de l'analyseur sémantique (`Semantics.h` et `Semantics.c`)

à partir de la grammaire attribuée (`java--.cg`). Le nombre de règles ajoutées automatiquement par l'outil `ag` doit être zéro.

7. `ast -c -d -i -m -w -= -4 SymTab.cg`

- option `c` — générer du code `C` ;
- option `d` — créer le fichier `SymTab.h` ;
- option `i` — créer le fichier `SymTab.c` ;
- option `m` — générer les fonctions de construction des noeuds avec le préfixe `m` (pour *make*) ;
- option `w` — générer la fonction `WriteTree` ;
- option `=` — générer la fonction `IsEqualTree`.
- option `4` — générer une description externe de l'arbre dans le fichier `SymTab.TS`.

Création des fichiers sources du gestionnaire de la table des symboles (`SymTab.h` et `SymTab.c`), qui contiennent la définition des types de données pour les noeuds de la table des symboles ainsi que les fonctions de construction de ces noeuds, à partir de la grammaire abstraite (`SymTab.cg`). Une description externe de la table des symboles sous la forme d'un graphe est produite dans le fichier `SymTab.TS`.

8. `echo SELECT AbstractGrammar Output | cat - java--.cg | ast -c -4`

- option `c` — générer du code `C` ;
- option `4` — générer une description externe de la table des symboles dans le fichier `Tree.TS`.

Génération de la description de l'arbre syntaxique abstrait dans le fichier `Tree.TS`.

9. `puma -c -d -i -k -p TypeSys.pum`

- option `c` — générer du code `C` ;
- option `d` — créer le fichier `TypeSys.h` ;
- option `i` — créer le fichier `TypeSys.c` ;
- option `k` — permettre des patrons non linéaires ;
- option `p` — permettre des constructeurs de noeuds sans parenthèses.

Création des fichiers sources du système de typage (`TypeSys.h` et `TypeSys.c`) à partir de patrons d'appariement (`TypeSys.pum`).

10. `puma -c -d -i GenCode.pum`

- option `c` — générer du code `C` ;
- option `d` — créer le fichier `GenCode.h` ;
- option `i` — créer le fichier `GenCode.c`.

Création des fichiers sources du générateur de code (`GenCode.h` et `GenCode.c`) à partir des procédures de génération de code données sous la forme de patrons d'appariement (`GenCode.pum`).

À la fin tous les modules « `.h` » et « `.c` » sont compilés à l'aide d'un compilateur `C`.

Toutes ces commandes sont intégrées dans le fichier `Makefile` de l'annexe I.

1.3 Compilation et exécution d'un programme écrit en *java--*

La figure 1.3 indique comment compiler et exécuter un programme écrit en langage *java--*. Cette procédure comporte trois étapes :

1. `java-- fichier.mjv` — compilation du programme écrit en *java--*;
2. `java -jar /opt/jasmin/jasmin-2.1/jasmin.jar fichier.j` — assemblage du programme;
3. `java nomClasse.class` — exécution du programme par la machine virtuelle *Java*.

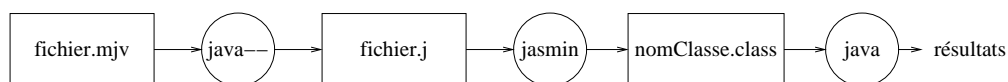


FIG. 1.3 – Compilation et exécution d'un programme *java--*

L'exécution d'un programme *java--*, préalablement compilé, nécessite un environnement d'exécution pour la lecture de données en entrée (l'entrée standard). Cet environnement est décrit dans l'annexe J.

1.4 Norme d'écriture des spécifications

Afin de faciliter la lecture et la modification des différentes spécifications du langage *java--*, certaines normes de nomenclature ont été adoptées. Le tableau 1.2 résume celles retenues pour les attributs et les symboles.

Les noms des attributs particuliers à l'analyse lexicale sont formés de lettres, mais ils commencent par la lettre « l ». Les noms des attributs qui apparaissent dans la grammaire concrète commencent généralement par la lettre « g ». Enfin, les noms des attributs qui apparaissent dans la grammaire abstraite ou dans la grammaire attribuée commencent par une lettre majuscule.

Les symboles terminaux de la grammaire concrète commencent par une lettre minuscule et les variables (symboles non terminaux) commencent par une lettre majuscule. Enfin, les symboles utilisés dans la grammaire abstraite et dans la grammaire attribuée sont souvent copiés de la grammaire concrète, mais préfixés de la lettre « a ».

TAB. 1.2 – Norme pour les noms des attributs et des symboles

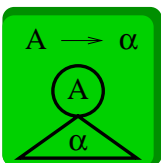
Spécification	Attribut	Symbole terminal	Variable
règles lexicales	l	N/A	N/A
grammaire concrète	g	minuscule	majuscule
grammaire abstraite	majuscule	N/A	a
grammaire attribuée	majuscule	N/A	a

1.5 Exercices

1. Téléchargez tous les fichiers pour la construction du compilateur *java--* dans un de vos répertoires.
2. Exécutez la commande **make**, puis observez tous les fichiers qui ont été créés.
3. Exécutez la commande **make clean**, puis observez tous les fichiers qui ont été détruits.
4. Écrivez un petit programme en langage *java--* , puis compilez-le à partir du compilateur généré par la commande **make**. Vous pouvez introduire intentionnellement des erreurs dans votre programme pour observer les différents messages affichés par le compilateur.

Chapitre 2

L'analyseur syntaxique



L'analyseur syntaxique vérifie qu'un programme écrit en langage *java--* respecte les règles syntaxiques de ce langage. Il construit aussi un arbre syntaxique abstrait qui représente le programme reconnu en entrée. Les outils `ast` et `lark` permettent de générer automatiquement un analyseur syntaxique ascendant à partir d'une grammaire hors contexte contenue dans le

fichier `java--.prs`. L'annexe D contient une copie de ce fichier avec une description succincte.

2.1 Contenu du fichier `java--.prs`

Le fichier `java--.prs` comporte deux modules : un premier module pour la description des règles syntaxiques et un deuxième module pour la construction de l'arbre syntaxique abstrait. Chaque module est divisé en sections identifiées par des clauses. Par exemple, dans le module *ConcreteSyntax*, la clause `PREC` introduit une section qui permet de spécifier la priorité des opérateurs du langage *java--*. Le module *ASTBuilder* contient plusieurs sections de code *C*. Celui sous la clause `GLOBAL` est inséré au niveau global dans le fichier `Parser.c`. Généralement, celui sous la clause `BEGIN` permet la création et l'initialisation de données déclarées dans la section `GLOBAL` et celui sous la section `CLOSE` permet la destruction de données à la fin de l'exécution de l'analyseur syntaxique.

2.2 Spécification des règles syntaxiques

La grammaire hors contexte apparaît dans le premier module du fichier `java--.prs` sous la clause `RULE`. Dans cette grammaire, les symboles entre apostrophes sont des symboles terminaux. Les autres symboles sont des variables à l'exception des symboles terminaux *identifier*, *stringLiteral*, *integerLiteral* et *flPointLiteral* introduits à la suite des règles de production. Notez la différence entre l'utilisation du caractère deux-points (« : ») et du caractère d'égalité (« = ») juste après l'introduction d'un nouveau symbole. C'est ce qui distingue un symbole terminal d'une variable. Enfin, la variable *CompilationUnit* est l'axiome de la grammaire, car il est le premier symbole défini dans la grammaire.

Contrairement à la notation non étendue usuelle, une règle de production se présente dans un format particulier. Par exemple, les deux règles de production

$$\textit{Modifier} \rightarrow \texttt{public} \text{ et } \textit{Modifier} \rightarrow \texttt{static}$$

sont codées sous la forme suivante :

```
Modifier = <
  PublicModifier = 'public' .
  StaticModifier = 'static' .
> .
```

Cette notation a l'avantage de regrouper les règles de production ayant le même membre de gauche (*Modifier*) et de les spécialiser à l'aide de nouveaux symboles (*PublicModifier* et *StaticModifier*), ce qui est fort utile pour la construction de l'arbre syntaxique abstrait.

Notons que la grammaire hors contexte contient quelques règles de production de la forme $A \rightarrow B$, appelées règles de production unitaires. Celles-ci ont été volontairement introduites afin de faciliter l'ajout éventuel de nouvelles règles de production issues de la grammaire originale du langage *Java*. Il y a aussi des règles de production d'effacement ($A \rightarrow \lambda$), celles pour lesquelles le membre de droite est absent.

La grammaire hors contexte du langage *java--* est ambiguë à cause de la forme particulière des règles de production pour les expressions booléennes (*BooleanExpression*) et les expressions infixées (*InfixExpression*). Pour lever les ambiguïtés, il est nécessaire de spécifier le type d'associativité (à gauche ou à droite) et la priorité de chacun des opérateurs apparaissant dans ces règles de production à l'aide de la clause **PREC**. Ceci permet d'obtenir une grammaire LALR(1) pour laquelle tous les conflits ont été explicitement résolus [1].

2.3 Définition des attributs associés aux symboles

La construction de l'arbre syntaxique abstrait requiert des attributs qui sont associés aux variables et aux symboles terminaux. Les attributs définis dans le fichier `java--.prs` sont généralement préfixés de la lettre « g ».

Le tableau 2.1 contient la liste des symboles terminaux, chacun avec leur attribut et le type de l'attribut. Ces attributs sont définis au même endroit que les symboles terminaux dans le fichier `java--.prs`. L'attribut *gIdent* est de type *tIdent*. Il désigne l'adresse (un entier positif) d'une suite de caractères reconnue par l'analyseur lexical (lexème) comme un identificateur. Les attributs *gStr*, *gInt* et *gFP* sont de types plus complexes, types qui sont introduits dans le chapitre 3. Tous les attributs des symboles terminaux sont calculés par l'analyseur lexical.

Presque toutes les variables de la grammaire possèdent un seul attribut, l'attribut *gPtr* de type *tTree* qui est essentiellement un pointeur vers un noeud de l'arbre syntaxique abstrait. La définition de cet attribut apparaît dans le module *ASTBuilder* sous la clause **DECLARE**. Même si cet attribut n'a pas été explicitement défini pour toutes les variables de la grammaire, une variable peut hériter de cet attribut si elle est une spécialisation d'un autre symbole. Par exemple, la variable *FunctionCall* hérite de l'attribut *gPtr* associé au symbole *Primary* à cause de la forme particulière des règles de production qui correspondent à ce dernier symbole.

TAB. 2.1 – Attributs des symboles terminaux

Symbole	Attribut	Type
identifieur	<i>gIdent</i>	<i>tIdent</i>
stringLiteral	<i>gStr</i>	<i>tStringLiteral</i>
integerLiteral	<i>gInt</i>	<i>tIntegerLiteral</i>
flPointLiteral	<i>gFP</i>	<i>tFPLiteral</i>

Les variables *Modifiers*, *Modifier*, *MethodHeader*, *MethodInvocation* et *ArrayCreatorSpec* ainsi que leurs spécialisations n'ont pas l'attribut *gPtr*, car elles possèdent des attributs particuliers. Ces attributs permettent :

- d'effectuer des vérifications qui auraient été difficiles à mettre en oeuvre à l'aide de règles syntaxiques (comme l'usage répétitif d'un modificateur) ;
- de regrouper de l'information propre à un élément syntaxique (les attributs de l'en-tête d'une méthode sont placés dans le noeud de la déclaration de méthode) ;
- de limiter le nombre de noeuds dans l'arbre syntaxique abstrait à cause de la présence de règles de production unitaires (*ProcedureCall* → *MethodInvocation*) ;
- de calculer des attributs (comme le nombre de dimensions pour lesquelles la taille est connue lors de la création d'un tableau).

2.4 Construction de l'arbre syntaxique abstrait

La construction de l'arbre syntaxique abstrait requiert une grammaire concrète (la grammaire hors contexte) et une grammaire abstraite. Les fichiers `java--.prs` de l'annexe D et `java--.cg` de l'annexe E contiennent respectivement ces deux grammaires. Une grammaire abstraite est obtenue à partir d'une grammaire hors contexte dépourvue des éléments syntaxiques qui ne sont pas pertinents pour une analyse sémantique. Par exemple, la règle suivante de la grammaire abstraite

```

aExpression = [Pos: tPosition] <
  (...)
  aBinary = [Op: short] [OpFamily: short]
    Left: aExpression Right: aExpression .

```

correspond aux règles suivantes de la grammaire concrète :

```

InfixExpression = <
  (...)
  Mul    = gLop:InfixExpression '*' gRop:InfixExpression .
  Div    = gLop:InfixExpression '/' gRop:InfixExpression .
  Mod    = gLop:InfixExpression '%' gRop:InfixExpression .
  Plus   = gLop:InfixExpression '+' gRop:InfixExpression .
  Minus  = gLop:InfixExpression '-' gRop:InfixExpression .
  > .

```

La grammaire abstraite a été construite de façon à ce que chaque symbole de cette grammaire corresponde à un ou plusieurs symboles de la grammaire hors contexte. Les symboles de la grammaire abstraite sont souvent obtenus à partir de ceux de la grammaire concrète auxquels est ajoutée la lettre « a » comme préfixé. Un nom générique est utilisé s'il correspond à plus d'un symbole de la grammaire concrète.

Dans l'exemple précédent, les symboles terminaux qui représentent les opérations de multiplication, de division, de modulo, d'addition et de soustraction sont absents dans la règle correspondante de la grammaire abstraite. Ils sont remplacés par trois attributs *Pos*, *Op* et *OpFamily* associés au symbole *aBinary*. Ce symbole correspond à un type de noeud de l'arbre avec six champs : le type de noeud (*aBinary*), l'attribut *Pos* pour la position de l'opérateur dans le fichier d'entrée, l'attribut *Op* pour un entier qui représente l'opérateur, l'attribut *OpFamily* pour un entier qui représente la famille de l'opérateur, un pointeur vers le noeud de l'opérande de gauche (*Left*) et un pointeur vers le noeud de l'opérande de droite (*Right*). Les attributs *Left* et *Right* permettent de distinguer l'opérande de gauche de l'opérande de droite, tout comme les attributs *gLop* et *gRop* dans la grammaire hors contexte. Un noeud de type *aBinary* est alors qualifié à l'aide d'un nom d'arbre.

2.4.1 Construction d'un noeud

Avec ces éléments en place, il est aisé de construire l'arbre syntaxique abstrait, car l'outil `ast` génère automatiquement toutes les déclarations des types de noeuds et toutes les fonctions pour créer ces noeuds. Par exemple, la fonction *maBinary*, qui accepte cinq paramètres, permet de construire un noeud de type *aBinary*. L'ordre des paramètres correspond à celui de la définition des attributs associés au symbole *aBinary*. Le code pour la construction de l'arbre syntaxique abstrait est placé dans le module *ASTBuilder* sous la clause `RULE` du fichier `java--.prs`. Par exemple, le code suivant construit un noeud pour une expression additive :

```
Plus = {
    gPtr := maBinary('':Position, OpPLUS, OPArithmetic, gLop:gPtr, gRop:gPtr);
} .
```

Ce code est implicitement associé à la règle de production

```
Plus = gLop:InfixExpression '+' gRop:InfixExpression .
```

Lorsque l'analyseur syntaxique effectue une réduction (*Reduce* en anglais) du membre de droite vers le membre de gauche de cette règle de production, il exécute aussi l'appel vers la fonction *maBinary*. À cette étape, l'analyseur dispose de deux pointeurs : un vers le noeud de l'opérande de gauche (*gLop*) et un vers le noeud de l'opérande de droite (*gRop*) qui sont passés en paramètres avec la position du symbole terminal « + » en entrée ainsi que du code *OpPLUS* (dont la valeur est 12) et du code *OPArithmetic* (dont la valeur est 4) pour indiquer qu'il s'agit d'une addition arithmétique. Les codes associés aux opérateurs sont définis dans le module *GlobalDeclarations* du fichier `java--.cg`. La figure 2.1 schématise cette opération.

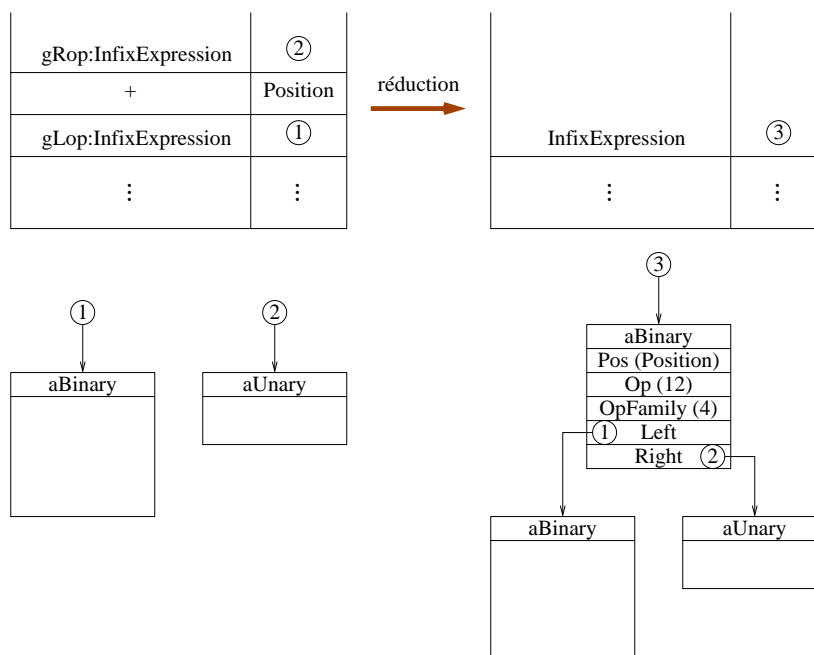


FIG. 2.1 – Création d'un noeud de l'arbre syntaxique abstrait

2.4.2 Construction d'une liste chaînée de noeuds

Les règles de production de la forme $Liste \rightarrow Liste \text{ Élément}$ et $Liste \rightarrow \text{Élément}$ engendrent des sous-arbres qui ont l'allure d'une cascade. Ils peuvent être remplacés par des listes chaînées de noeuds pour des raisons d'efficacité, car ils sont généralement nombreux et de grande taille dans un arbre syntaxique abstrait. La figure 2.2 illustre cette situation.

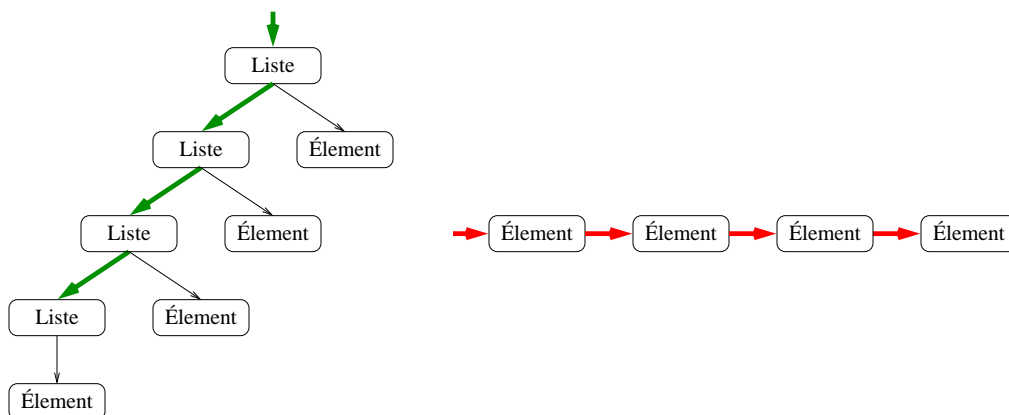


FIG. 2.2 – Structure arborescente versus liste chaînée

Cette façon de faire exige un traitement particulier, car il est très important de tenir compte de l'ordre dans lequel les éléments sont présentés en entrée. Par exemple, les règles suivantes de la grammaire abstraite

La figure 2.4 montre l'évolution de la machine caractéristique lors de l'analyse syntaxique de la chaîne « ...**int x, y ;** », avec le pointeur d'entrée sur **x**.

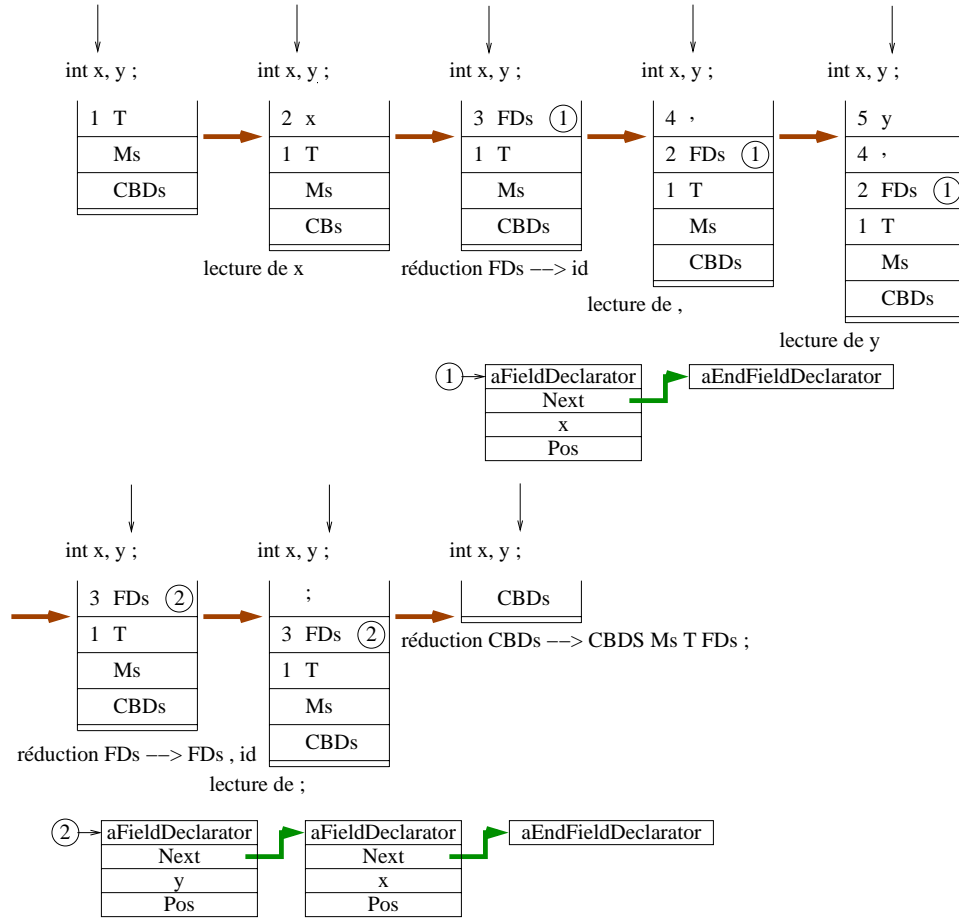


FIG. 2.4 – Évolution de la machine caractéristique

Supposons que la machine caractéristique est dans l'état 1. Dans cet état, il y a tout d'abord une lecture (*Shift* en anglais) du symbole **x** en entrée, donc une transition étiquetée « **x** » de l'état 1 vers l'état 2. La seule possibilité dans l'état 2 est une réduction à partir de la règle

$$\text{FieldDeclarators} \rightarrow \text{identifieur}$$

ce qui a pour effet de créer une liste chaînée de deux noeuds puisque l'appel de la fonction *maFieldDeclarator* est associé au symbole *FirstFieldDeclarator*, une spécialisation du membre de droite de la règle de production : tout d'abord un premier noeud de type *aFieldDeclarator* et ensuite un deuxième noeud de type *aEndFieldDeclarators* chaîné au premier. La machine caractéristique passe alors à l'état 3.

À partir de l'état 3, il y a deux lectures successives. Tout d'abord, la lecture de la virgule, donc une transition étiquetée « **,** » de l'état 3 vers l'état 4. Puis la lecture du symbole **y**, donc une transition étiquetée « **y** » de l'état 4 vers l'état 5. La seule possibilité dans l'état 5 est une réduction à partir de la règle

$$\text{FieldDeclarators} \rightarrow \text{FieldDeclarators} , \text{identifieur}$$

puisque le membre de droite de cette règle de production est au sommet de la pile (y , « , » et FDs). Notons que le code associé à cette réduction est celui du symbole *LastFieldDeclarators* (voir la correspondance entre les règles de la grammaire concrète et celles de la grammaire abstraite). Le terme « **FieldDeclarators:gPtr** » désigne le pointeur numéroté 1, car dans la pile ce pointeur est vis-à-vis le symbole FDs (l'abréviation de *FieldDeclarators*). Globalement, tout le code associé à cette réduction permet de créer un noeud de type *aFieldDeclaration*, de chaîner les noeuds tel qu'illustré dans la figure 2.4 et de conserver le pointeur numéroté 2 au sommet de la pile, c'est-à-dire de l'associer au membre de droite de la règle de production utilisée lors de la réduction. La machine caractéristique passe alors à l'état 3.

Le résultat est une liste chaînée d'identificateurs. Mais elle ne se présente pas dans le bon ordre! Le premier noeud est celui de l'identificateur y . La liste chaînée doit être inversée. Éventuellement, lors de la réduction à partir de la règle

$$ClassBodyDeclarations \rightarrow ClassBodyDeclarations \text{ Modifiers } Type \text{ FieldDeclarators } ;$$

le code

```
FieldDeclaration = {
  gPtr := maFieldDeclaration( ClassBodyDeclarations:gPtr,
                             Modifiers:modifiers,
                             Type:gPtr,
                             ReverseTree( FieldDeclarators:gPtr ) );
}
```

associé au symbole *FieldDeclaration* (une spécialisation du symbole *ClassBodyDeclarations*, le membre de droite de la règle de production), est exécuté, provoquant l'appel de la fonction *ReverseTree*, avec le pointeur numéroté 2 comme paramètre actuel, qui inverse la liste chaînée. La figure 2.5 donne le résultat de cette opération. Notez que le noeud de type *aEndFieldDeclarators* reste toujours le dernier noeud de la liste chaînée.

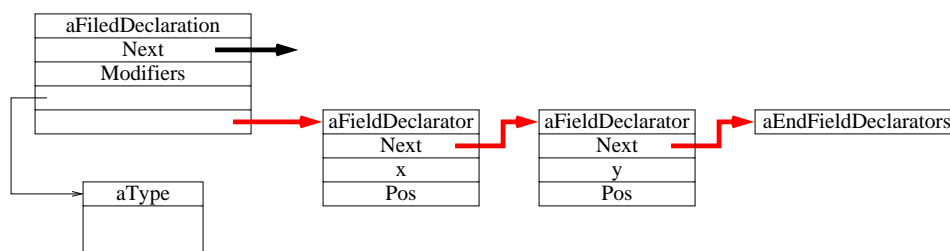


FIG. 2.5 – Résultat de l'analyse syntaxique d'une suite de noms de champs

La manipulation de l'attribut *Next* requiert parfois une codification particulière. Par exemple, le code suivant dans le fichier `java--.prs`

```
SomeBlockStatements = {
  gPtr := { BlockStatement:gPtr->\aStatement.Next = BlockStatements:gPtr;
           gPtr = BlockStatement:gPtr; };
}
```

permet de chaîner une liste d'énoncés. L'expression

```
BlockStatement:gPtr->\aStatement.Next
```

réfère au champ *Next* du noeud représentant un énoncé. Dans cette expression, la barre oblique inverse (*Backslash* en anglais) indique à l'outil *ast* de ne pas interpréter le symbole *aStatement*, mais plutôt de le considérer comme un nom de champ d'une structure *C*.

Enfin, certaines variantes pour la construction d'une liste chaînée sont possibles. Par exemple, le code *C*

```
NoArguments = {
    gPtr := maNoArgument( '(' :Position );
} .
SomeArguments = {
    gPtr := ReverseTree( ListOfArguments:gPtr );
} .

OneArgument = {
    gPtr := maArgument( maNoArgument( NoPosition ), Expression:gPtr );
} .
MoreArguments = {
    gPtr := maArgument( ListOfArguments:gPtr, Expression:gPtr );
} .
```

est associé aux règles de production suivantes de la grammaire concrète :

```
Arguments = <
    NoArguments    = '(' ')' .
    SomeArguments = '(' ListOfArguments ')' .
> .

ListOfArguments = <
    OneArgument    = Expression .
    MoreArguments = ListOfArguments ',' Expression .
> .
```

Dans ce code, la fonction *ReverseTree* inverse une liste d'expressions qui représente la liste des paramètres actuels dans l'appel d'une méthode, car cette liste n'est pas dans le bon ordre. Cette opération est possible, car un noeud de type *aArgument* possède un champ *Next* avec la propriété REV :

```
aArguments = <
    aNoArgument = [Pos: tPosition OUT] .
    aArgument    = Next:aArguments REV aExpression .
> .
```

C'est ce type de noeud qui est utilisé lors de l'analyse d'une liste d'expressions (voir l'appel de la fonction *maArgument* associé aux symboles *OneArgument* et *MoreArguments*). Dans cette variante, le pointeur (*gPtr*) associé au symbole *ListOfArguments* s'apparente au pointeur *Next* dans l'approche précédente. Le symbole *ListOfArguments* correspond au symbole

aArguments dans la grammaire abstraite. Comme l'illustre la figure 2.6, l'inversion est faite à même la structure de l'arbre. Cette façon de faire évite d'avoir des noeuds dont leur type est une spécialisation du type *aExpression* avec un pointeur *Next*, car le symbole *Expression* apparaît dans plusieurs règles de production, sans pour autant être un élément d'une liste d'expressions.

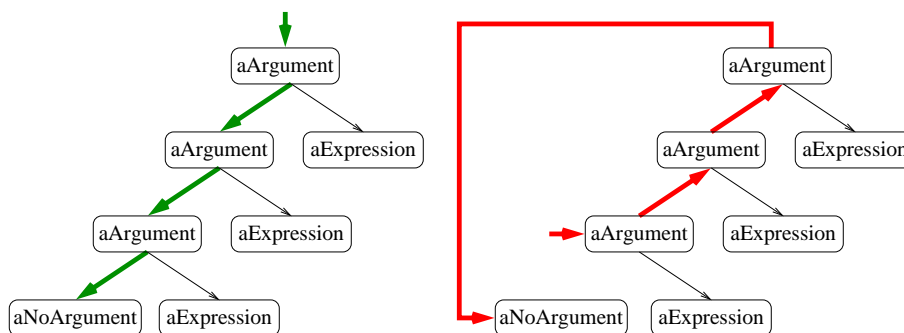


FIG. 2.6 – Inversion d'une liste de noeuds dans l'arbre syntaxique abstrait

Il est possible d'écrire le code d'une façon plus opaque pour l'outil **ast**, c'est-à-dire presque complètement dans le langage *C*. Ainsi, le code

```
MoreArguments = {
    gPtr := maArgument( ListOfArguments:gPtr, Expression:gPtr );
} .
```

peut être remplacé par le code suivant :

```
MoreArguments = {
    gPtr := { gPtr = maArgument( ListOfArguments:gPtr, Expression:gPtr ) };
} .
```

Le code entre les accolades imbriquées est presque du code *C*. Seuls les termes

`ListOfArguments:gPtr` et `Expression:gPtr`

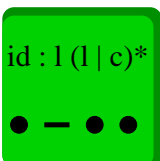
sont interprétés par l'outil **ast** (à cause de l'absence de la barre oblique inverse). Le symbole « = » correspond à l'opérateur d'affectation du langage *C*. Ce code est opaque pour l'outil **ast**, mais il permet l'insertion de tout énoncé *C*, en particulier des énoncés qui sont utiles dans la mise au point de l'analyseur syntaxique.

2.5 Exercices

1. Examinez le contenu du fichier `Parser.lrk` généré par l'outil `lpp`.
2. Examinez le contenu des fichiers `Parser.h` et `Parser.c` générés par l'outil `lark`.
3. Examinez le contenu des fichiers `Tree.h` et `Tree.c` générés par l'outil `ast`.

Chapitre 3

L'analyseur lexical



L'analyseur lexical identifie les unités lexicales d'un programme *java--*, une à une, c'est-à-dire lors de l'appel de la fonction *GetToken* par l'analyseur syntaxique.

3.1 Spécification des règles lexicales

Les unités lexicales correspondent aux symboles terminaux de la grammaire hors contexte du langage de programmation *java--*, symboles à partir desquels un programmeur écrit un programme dans ce langage. Généralement, une règle lexicale doit être donnée pour chaque unité lexicale sous la forme d'une expression régulière. Cependant, l'utilisation d'outils puissants pour la construction automatique du compilateur *java--* simplifie grandement le travail. Une expression régulière est nécessaire :

- lorsque le langage associé à une unité lexicale contient plus d'un lexème (par exemple, les identificateurs ou les constantes entières) ;
- lorsque des cas particuliers se présentent, comme l'interdiction d'utiliser des mots réservés (par exemple, **const** et **goto**) ou le traitement de commentaires dans un programme.

Dans tous les autres cas, les expressions régulières sont automatiquement générées à partir de la grammaire hors contexte contenue dans le fichier *java--.prs* de l'annexe D.

La section **RULE** du fichier *java--.scn* de l'annexe C contient les expressions régulières pour les commentaires, les mots réservés **const** et **goto**, les identificateurs, les caractères, les chaînes de caractères, les constantes entières et les constantes en point flottant.

3.2 Calcul des attributs intrinsèques

À la suite de chaque expression régulière, il y a du code *C* qui permet :

- de retourner une valeur entière qui représente l'unité lexicale associée à l'expression régulière (l'énoncé **return** met fin à l'exécution de la fonction *GetToken* et provoque un retour vers l'analyseur syntaxique) ;

- de calculer des attributs sémantiques, appelés attributs intrinsèques, propres à l'unité lexicale associée à l'expression régulière.

Le calcul des attributs d'une unité lexicale peut être relativement complexe. Par exemple, les structures de données suivantes définissent les attributs d'une constante entière (voir le fichier `JavaIntAttributes.h` de l'annexe B) :

```
typedef int tIntType;
typedef long long int tLongType;
typedef enum { Int, Long } tCategory;

typedef union { tIntType B32; tLongType B64; } tRepresentation;

typedef struct { tStringRef lLexeme;
                tRepresentation lValue;
                tCategory lGroup;
                short lMinValue;
                short lOverflow; } tIntegerLiteral;
```

L'attribut *lLexeme* contient l'adresse du lexème reconnu en entrée par l'analyseur lexical comme une constante entière. L'attribut *lValue* est la valeur binaire de la constante entière sur 32 ou 64 bits. L'attribut *lGroup* permet de différencier une constante entière de 32 bits d'une constante entière de 64 bits (à partir du suffixe *l* ou *L*). L'attribut *lMinValue* indique si la constante entière dans la notation décimale correspond à la plus petite valeur possible en valeur absolue (par exemple 2 147 483 648 ou 9 223 372 036 854 775 808). Enfin, l'attribut *lOverflow* indique si la constante entière est trop grande pour tenir sur 32 ou 64 bits.

Généralement, le code associé au calcul des attributs intrinsèques ne peut être généré automatiquement, car il dépend beaucoup trop du langage de programmation source et du langage de programmation cible. C'est pour cette raison que le calcul des attributs intrinsèques a été programmé. Pour les constantes en point flottant, les constantes entières, les caractères et les chaînes de caractères, la déclaration de leurs attributs ainsi que leur calcul sont contenus dans les fichiers suivants :

Nom du fichier	Contenu
<code>JavaFPAttributes.h</code>	déclaration des attributs d'une constante en point flottant
<code>JavaIntAttributes.h</code>	déclaration des attributs d'une constante entière
<code>JavaCharAttributes.h</code>	déclaration des attributs d'un caractère
<code>JavaStringAttributes.h</code>	déclaration des attributs d'une chaîne de caractères
<code>JavaFPAttributes.c</code>	calcul des attributs d'une constante en point flottant
<code>JavaIntAttributes.c</code>	calcul des attributs d'une constante entière
<code>JavaCharAttributes.c</code>	calcul des attributs d'un caractère
<code>JavaStringAttributes.c</code>	calcul des attributs d'une chaîne de caractères

Notons que le calcul des attributs d'un caractère et d'une chaîne de caractères nécessite le traitement de séquences d'échappement comme « `\b` » ou « `\177` ». Les fichiers suivants contiennent le code relatif à ce traitement :

Nom du fichier	Contenu
JavaEscapeSeq.h	déclaration des données d'une séquence d'échappement
JavaEscapeSeq.c	calcul des données d'une séquence d'échappement

Tous les fichiers contenant le code *C* pour la déclaration et le calcul des attributs se trouvent en annexe B.

Comme le calcul de l'attribut associé à un identificateur est relativement simple, il est placé à la suite de l'expression régulière. Le code pour le calcul de l'attribut *gIdent* d'un identificateur est le suivant :

```
Attribute.identifier.gIdent = MakeIdent(TokenPtr, TokenLength);
return identifier;
```

La fonction *MakeIdent* de l'outil *cocktail* emmagasine le lexème courant dans un tableau et retourne son adresse sous la forme d'un entier positif. Le lexème pourra donc être récupéré par l'analyseur syntaxique à partir de la valeur de l'attribut *gIdent*.

Les symboles *identifier*, *stringLiteral*, *integerLiteral* et *flPointLiteral* sont introduits à la fin de la grammaire hors contexte contenue dans le fichier `java--.prs` de l'annexe D. Ce sont des symboles terminaux de la grammaire hors contexte. Les symboles *gIdent*, *gStr*, *gInt* et *gFP*, introduits au même endroit dans le fichier `java--.prs`, représentent les attributs associés à ces symboles terminaux. À partir de ces définitions, l'outil *lpp* génère automatiquement les structures de données, en particulier le type *tScanAttribute*, qui permettent la mémorisation des différents attributs.

```
typedef struct { tPosition zzPos; tIdent gIdent; } zz_identifier;
typedef struct { tPosition zzPos; tStringLiteral gStr; } zz_stringLiteral;
typedef struct { tPosition zzPos; tIntegerLiteral gInt; } zz_integerLiteral;
typedef struct { tPosition zzPos; tFPLiteral gFP; } zz_flPointLiteral;

typedef union {
tPosition Position;
zz_identifier identifier;
zz_stringLiteral stringLiteral;
zz_integerLiteral integerLiteral;
zz_flPointLiteral flPointLiteral;
} tScanAttribute;
```

Notez la présence de l'attribut *Position* commun à toutes les unités lexicales qui représente la position (numéro de ligne et numéro de colonne) du lexème associé à l'unité lexicale reconnue par l'analyseur lexical. Cet attribut occupe le même emplacement que les attributs *zzPos* associés respectivement à un identificateur, à une chaîne de caractères, à une constante entière et à une constante en point flottant.

Le type *tScanAttribute* agit comme une interface entre l'analyseur lexical et l'analyseur syntaxique.

Premièrement, l'outil *rex* génère la déclaration de la variable globale *Attribute* de type *tScanAttribute*. Ainsi, l'analyseur lexical affecte une valeur à un attribut en utilisant cette variable. Par exemple, l'affectation suivante dans la fonction *SetAttributesOctal* (voir le fichier `JavaIntAttributes.c` de l'annexe B)

```
Attribute.integerLiteral.gInt.lValue.B32 = strtol(lexeme, %d, 8);
```

positionne la valeur binaire d'une constante entière octale de 32 bits.

Deuxièmement, l'outil `lark` génère la déclaration de la variable globale *Scan* de type *tScanAttribute*. Ainsi, l'analyseur syntaxique récupère généralement la valeur d'un attribut lors de la construction d'un noeud de l'arbre syntaxique abstrait en utilisant cette variable. Par exemple, l'expression pour le dernier paramètre actuel de l'appel de la fonction `maIntConst`

```
IntConst = {
  gPtr := {
    (...)
    gPtr = maIntConst( integerLiteral:Position,
                      integerLiteral:gInt.lValue.B32 );
```

récupère la valeur binaire d'une constante entière. La variable *Scan* n'apparaît pas explicitement dans cette expression puisque ce n'est pas du code *C*. Toutefois, elle apparaît dans le fichier `Parser.lrk` généré par l'outil `lark` :

```
Primary : integerLiteral {
  (...)
  $$ .Primary.gPtr = maIntConst( $1.Scan.Position,
                                $1.Scan.integerLiteral.gInt.lValue.B32 );
```

Dans ce code, `$1` désigne le premier symbole du membre de droite de la règle de production et le double signe de dollar désigne le membre de gauche.

Comme le fichier `java--.scn` ainsi que les fichiers de déclaration des attributs et ceux de leur calcul sont relativement complets, ils exigeront peu ou pas de modifications. Il est donc important de ne pas trop s'attarder sur les détails, mais bien de comprendre les principaux éléments introduits dans ce chapitre.

3.3 Exercices

1. Examinez le contenu du fichier `Scanner.rpp` généré par l'outil `lpp`.
2. Examinez le contenu du fichier `java--.rex` généré par l'outil `rpp`.
3. Examinez le contenu des fichiers `Scanner.h` et `Scanner.c` générés par l'outil `rex`.

Chapitre 4

La machine virtuelle *Java*

J2VM

Une machine virtuelle *Java* permet l'exécution de tous les programmes *Java*. Elle peut être considérée comme un ordinateur abstrait indépendant de tout support matériel sur lequel il est mise en oeuvre. La figure 4.1 présente quelques éléments d'une machine virtuelle *Java* dont les ressources physiques sont gérées par un système d'exploitation hôte [14]. Le chargeur de classes et le moteur d'exécution constituent les deux principaux éléments. Le premier permet le chargement de types, qu'ils s'agissent de classes ou d'interfaces, à partir de noms qualifiés. Le second permet l'exécution des instructions contenues dans les méthodes des classes préalablement chargées.

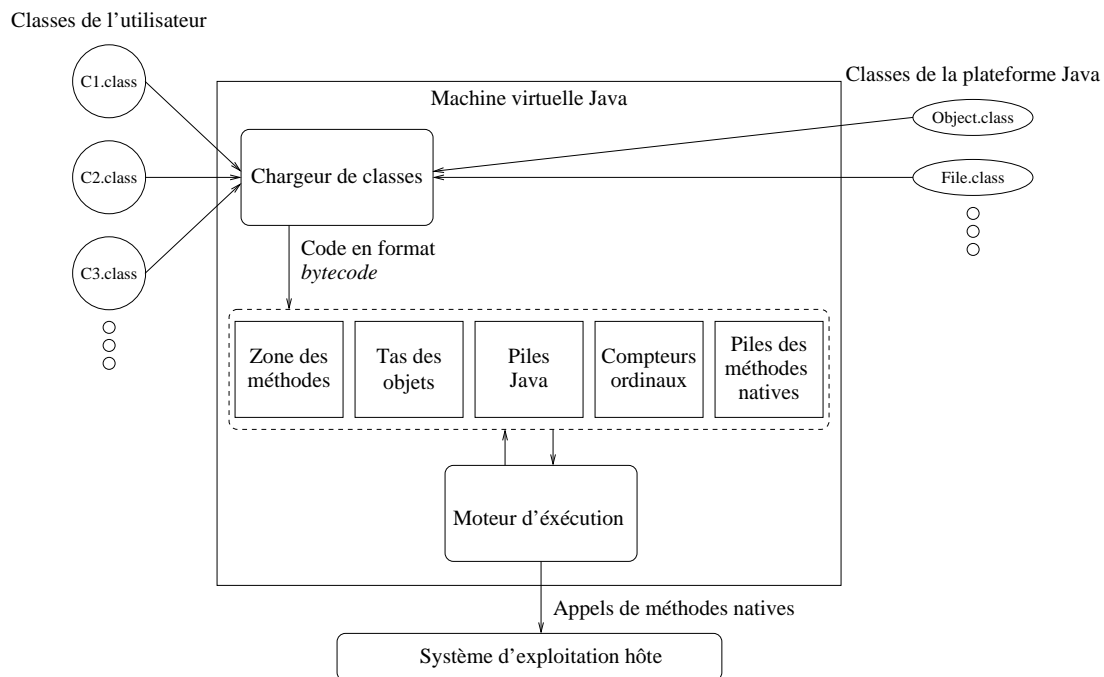


FIG. 4.1 – Une machine virtuelle *Java*

La figure 4.1 inclut aussi les zones de données requises pour l'exécution d'un programme *Java*. Elles contiennent des structures de données qui sont créées, modifiées et détruites à différents moments durant la vie d'un programme.

- Lors du chargement d'une classe, la zone des méthodes est modifiée à partir de l'information contenue dans le fichier `.class`.
- Lors d'un lancement d'un *thread*, une pile *Java* et un compteur ordinal sont respectivement alloués dans la zone des piles *Java* et dans la zone des compteurs ordinaux.
- Lors de l'instanciation d'un objet, l'objet est placé dans le tas (*Heap*) des objets instanciés par le programme.
- Lors de l'exécution d'une méthode native, l'état de l'appel d'une méthode native est ajouté sur une pile particulière de la zone de piles des méthodes natives, car la gestion d'une telle pile dépend du langage qui a été utilisé pour programmer la méthode.

Chaque appel d'une méthode *Java*, lors de l'exécution d'un *thread*, provoque l'ajout d'un bloc d'activation (*Stack Frame*) sur la pile correspondante. Ce bloc d'activation est détruit lors du retour vers l'appelant. La figure 4.2 illustre l'état de l'exécution d'un programme *Java* dans lequel trois *threads* sont actifs. La pile associée au premier *thread* contient trois blocs d'activation qui représentent trois niveaux imbriqués d'appel d'une méthode. Seule la description du contenu d'un bloc d'activation est nécessaire à la compréhension du module de génération de code.

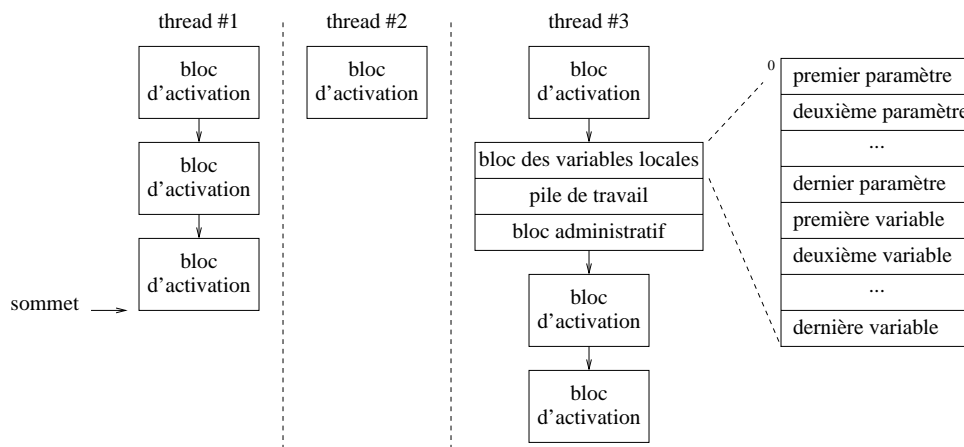


FIG. 4.2 – Les piles *Java*

4.1 Contenu d'un bloc d'activation

Comme le montre la figure 4.2, le bloc d'activation associé à l'appel d'une méthode est divisé en trois parties distinctes :

- la partie des variables locales (*Local Variables*), appelée bloc des variables locales, qui contient les paramètres actuels passés lors de l'appel de la méthode ainsi que ses variables locales ;
- la partie de travail, appelée pile de travail (*Operand Stack*), qui contient les opérandes des opérations effectuées par les instructions ;

- la partie administrative, appelée bloc administratif (*Frame Data*), qui contient, entre autres, l'adresse de retour, c'est-à-dire l'adresse de l'instruction qui suit l'appel de la méthode.

Chaque partie d'un bloc d'activation est accessible à l'aide d'instructions particulières qui lui sont propres. Cela ne veut pas dire qu'un programme a nécessairement accès à toute la structure d'un bloc d'activation, comme la partie administrative qui n'est pas directement accessible. Par contre, des instructions d'appel d'une méthode comme `invokestatic` et `invokevirtual` effectuent toutes les opérations élémentaires sur cette partie, opérations qui sont normalement générées sous la forme d'instructions par un compilateur, mais qui ne sont pas nécessaires à cause des capacités de la machine virtuelle *Java*. La partie administrative est donc à ignorer par le générateur de code puisque sa gestion est faite en totalité par la machine virtuelle *Java*.

4.2 Taille d'un bloc d'activation

La taille du bloc des variables locales et la taille de la pile de travail doivent être déterminées lors de la compilation de chaque méthode. Celle-ci dépend de la taille des données, donc de leur type. La machine virtuelle *Java* supporte deux types :

- les types longs, c'est-à-dire `long` et `double`, qui occupent deux mots ;
- les types courts, c'est-à-dire `int`, `float` et référence, qui occupent un mot.

Une donnée de type `short`, `char`, `byte` ou `boolean` est considérée comme une donnée de type `int` lorsqu'elle est placée sur la pile. Comme les entiers sont signés, il y a alors propagation du bit de signe.

Les types ne permettent pas seulement de déterminer la taille des données, mais aussi d'utiliser le code mnémorique approprié pour une opération. En effet, chaque code mnémorique a un préfixe, c'est-à-dire, `l` (pour `long`), `d` (pour `double`), `i` (pour `int`), `f` (pour `float`) ou `a` (pour référence), qui indique le type de l'opération. Par exemple, l'addition de deux données de type `int` est représentée par le code mnémorique `iadd`, alors que l'addition de deux données de type `float` est représentée par le code mnémorique `fadd`. Bien entendu, une opération ne s'applique pas toujours à tous les types de données. Par exemple, l'addition de deux références n'existe pas, d'où l'absence du code mnémorique `aadd`.

Enfin, le jeu d'instructions de la machine virtuelle *Java* n'est pas très homogène, car par exemple, il y a des familles d'instructions dont le comportement pour des données de type `int` est différent de celui pour des données des autres types.

4.3 Bloc des variables locales

Le bloc des variables locales est organisé comme un tableau de mots d'origine zéro en fonction du type des données et de leur catégorie (paramètre ou variable locale).

Dans le cas d'une méthode de classe, c'est-à-dire une méthode avec le modificateur `static`, les paramètres sont placés à partir du mot d'origine zéro selon l'ordre de leur apparition dans la liste des paramètres. Ils sont suivis des variables locales placées selon l'ordre de leur apparition dans les déclarations.

Dans le cas d'une méthode d'instance, le mot d'origine zéro est réservé pour la référence à l'instance courante (**this**). Ce mot est initialisé par la machine virtuelle *Java*. Ainsi, par rapport à une méthode de classe, les paramètres et les variables locales sont décalés d'un mot.

L'accès à un paramètre ou à une variable locale est effectué à l'aide d'un déplacement par rapport à l'origine du bloc. Mais comme les données n'ont pas toutes la même taille, un tel déplacement est déterminé en considérant leur type. Par exemple, la figure 4.3 contient le bloc des variables locales correspondant à la méthode suivante :

```
static void p(int x, long y)
{
    int i;
    double d;
    boolean b;
    ...
}
```

Les données de types longs peuvent être alignées sur des frontières impaires, car il n'y a pas de contraintes d'alignement comme c'est souvent le cas dans des architectures d'ordinateurs.

Les déplacements apparaissent comme paramètres dans les instructions. Ainsi, l'instruction

`iload 3`

place la valeur de la variable `i` sur la pile de travail. Cette valeur pourra être ultérieurement utilisée comme opérande dans une autre instruction. L'instruction

`istore 3`

stocke la valeur entière au sommet de la pile de travail dans l'emplacement réservé à la variable `i` dans le bloc des variables locales. Les instructions `dload` et `dstore`, avec un déplacement égal à quatre, permettent de réaliser les mêmes traitements, mais pour la variable `d`.

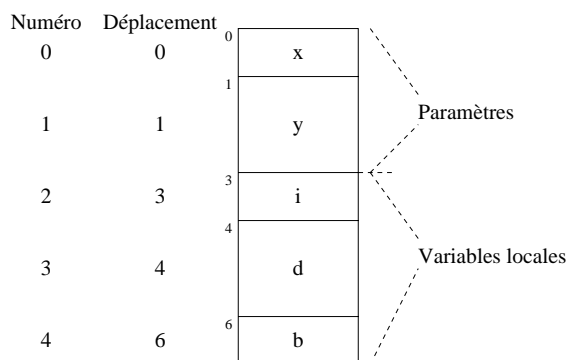


FIG. 4.3 – Exemple d'un bloc de variables locales

Chapitre 5

L'assembleur Jasmin



Le logiciel **Jasmin** (pour *Java Assembler Interface*) est un assembleur qui accepte une description d'une classe *Java* écrite en langage d'assemblage **JasminXT** et qui la traduit en *bytecode*, le langage de la machine virtuelle *Java*. Le jeu d'instructions de cet assembleur correspond à celui de la machine virtuelle *Java*. L'information relative à cet assembleur est disponible sur le site Web à partir de l'adresse URL suivante : <http://jasmin.sourceforge.net/> [13]. Seuls les éléments utiles à la génération de code d'un programme *java--* sont décrits dans ce chapitre.

5.1 Organisation d'un programme JasminXT

Un programme **JasminXT** comporte trois parties : un en-tête, des déclarations de champs et des définitions de méthodes avec leurs instructions. Cette structure est décrite à l'aide d'une règle de production d'une grammaire hors contexte :

$$jas_file \rightarrow jasmin_header\{field\}\{method\}$$

L'en-tête ou le prologue comporte des directives qui positionnent des attributs d'une classe. Celles pertinentes à la génération de code d'un programme *java--* sont introduites dans la section 5.1.1. Les autres parties d'un programme **JasminXT** contiennent aussi des directives.

Une directive commence par un point suivi de son nom et de zéro ou plusieurs paramètres séparés par des espaces. Par exemple, l'énoncé `.class public SimpleProgramme` dans le programme de la figure 5.1 est une directive avec deux paramètres. Elle introduit une classe publique nommée **SimpleProgramme**.

L'en-tête d'un programme **JasminXT** est suivi de zéro ou plusieurs déclarations de champs de la classe. Dans le programme de la figure 5.1, il y a deux déclarations, celles des champs statiques *M* et *N*. La directive utile à la déclaration et à l'initialisation d'un champ de classe est introduite dans la section 5.1.2.

La dernière partie d'un programme **JasminXT** contient les définitions de méthodes de la classe. En plus des directives qui permettent la déclaration d'une méthode et le positionnement de ses attributs, cette partie contient des instructions qui sont placées entre les directives `.method` et `.end method`. Une instruction comporte un code mnémotechnique suivi de

```

; Prologue
.source merci.j                ; Nom du fichier source
.class public SimpleProgramme   ; Nom de la classe
.super java/lang/Object         ; Nom de la super classe

; Champs de la classe
.field public static 'M' I = 5   ; Champ statique M
.field public static 'N' I = 200 ; Champ statique N

; Constructeur
.method public <init>()V
    .limit locals 1              ; Taille du bloc des variables locales
    .limit stack 2              ; Taille de la pile de travail
    aload_0
    invokevirtual java/lang/Object/<init>()V
    return
.end method

; Fonction main de la classe
.method public static main([Ljava/lang/String;)V
    .limit stack 3              ; Taille de la pile de travail

    getstatic java/lang/System/out Ljava/io/PrintStream;      ; Empiler System.out
    ldc "Merci "                                                ; Empiler "Merci "
    invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V ; Appel de print

    getstatic java/lang/System/out Ljava/io/PrintStream;      ; Empiler System.out
    getstatic SimpleProgramme/M I                             ; Empiler la valeur de M
    getstatic SimpleProgramme/N I                             ; Empiler la valeur de N
    imul                                                         ; Calculer M * N
    invokevirtual java/io/PrintStream/print(I)V                ; Appel de print pour un entier

    getstatic java/lang/System/out Ljava/io/PrintStream;      ; Empiler System.out
    ldc " fois!"                                                ; Empiler " fois!"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V ; Appel de println

    return
.end method

```

FIG. 5.1 – Programme JasminXT qui affiche *Merci 1000 fois!*

zéro ou plusieurs paramètres séparés par des espaces. Par exemple, l'énoncé `ldc "Merci "` dans la méthode `main` du programme de la figure 5.1 est une instruction avec un paramètre. Elle place au sommet de la pile de travail une référence vers la chaîne de caractères « `Merci` ». Une instruction peut être précédée d'une étiquette qui consiste en un nom suivi du caractère deux-points.

Des commentaires peuvent être placés à tout endroit dans un programme. Ils commencent par un point-virgule, généralement précédé d'un délimiteur, comme un espace, une fin de ligne ou un tabulateur, et se terminent à la fin de la ligne.

5.1.1 En-tête d'un programme

La règle de production suivante décrit l'en-tête d'un programme JasminXT :

$$\begin{aligned} \textit{jasmin_header} \rightarrow & \text{ [.source sourcefile]} \\ & \text{ .class \{access_spec\} class_name} \\ & \text{ .super class_name} \end{aligned}$$

La directive `.source` indique le nom du fichier source qui ne doit pas inclure le chemin d'accès. La directive `.class` donne le nom complet de la classe avec ses modificateurs (seulement `public` par `java--`). Enfin, la directive `.super` spécifie le nom de la super classe qui est toujours `java/lang/Object` pour tout programme généré par `java--`. Ces directives apparaissent dans le programme de la figure 5.1.

5.1.2 Déclaration d'un champ

Une déclaration d'un champ de la classe est faite à partir de la directive `.field` selon la règle de production suivante :

$$\textit{field} \rightarrow \text{ .field \{access_spec\} field_name descriptor [= value]}$$

Les modificateurs `public` et `static` sont toujours ceux donnés comme spécification d'accès (*access_spec*) par `java--`. Un descripteur est formé à partir des symboles suivants, mais agencés pour former un type de données : `I` pour `int`, `F` pour `float`, `L` pour une classe et `[` pour un tableau. La seule classe permise par `java--` est `java/lang/String` pour une chaîne de caractères. Le nom d'une classe inclut son chemin d'accès et doit être précédé du symbole `L` et suivi d'un point-virgule. Un champ de classe doit toujours être initialisé, ce qui n'est pas le cas d'un champ d'instance.

5.1.3 Définition d'une méthode

Une définition d'une méthode de la classe est faite à partir de la directive `.method` selon la règle de production suivante :

$$\textit{method} \rightarrow \text{ .method \{access_spec\} method_name descriptor \{statement\} .end method}$$

Les modificateurs `public` et `static` sont toujours ceux donnés comme spécification d'accès (*access_spec*) par `java--`. Un descripteur est une suite de types entre parenthèses, un type

par paramètre formel, tout cela suivi par le type de la valeur retournée par la méthode. Un type de données est formé à partir des symboles suivants : **I** pour `int`, **F** pour `float`, **L** pour une classe, **V** pour aucun type (`void`) et **[** pour un tableau.

5.1.4 Énoncés

La règle de production suivante décrit les énoncés possibles d'une méthode d'un programme `JasminXT` généré à partir d'un programme `java--` :

```
statement → .limit stack size
           | .limit locals size
           | .line number
           | .var var_number is var_name descriptor from label1 to label2
           | instruction [instruction_args]
           | label :
```

Un énoncé est soit une directive soit une instruction. Les instructions sont décrites dans la section 5.3. Quatre directives sont possibles :

- la directive `.limit stack` qui indique la taille de la pile de travail contenue dans les blocs d'activation de la méthode ;
- la directive `.limit locals` qui indique la taille du bloc des variables locales contenu dans les blocs d'activation de la méthode ;
- la directive `.line` qui indique le numéro d'une ligne dans un programme `java--`, information utile au débogage ;
- la directive `.var` qui associe une zone dans le bloc des variables locales pour un paramètre formel ou une variable locale de la méthode et qui indique son nom, son type et sa portée.

5.2 Assemblage et exécution d'un programme `JasminXT`

La commande suivante permet d'assembler le programme `JasminXT` contenu dans le fichier `merci.j`, celui de la figure 5.1 :

```
java -jar /opt/jasmin/jasmin-2.1/jasmin.jar merci.j
```

La commande suivante permet d'exécuter le programme traduit en *bytecode* :

```
java SimpleProgramme
```

5.3 Instructions utiles

La présentation des instructions se veut uniforme. Pour chaque famille d'instructions, les opérandes nécessaires pour leur exécution, c'est-à-dire les paramètres des instructions et les opérandes sur la pile de travail, sont fournis ainsi que les codes mnémoniques et

l'évolution de la pile de travail lors de leur exécution. Une description succincte est fournie si nécessaire. Généralement, les opérandes au sommet de la pile de travail sont consommés lors de l'exécution des instructions. La signification de chaque instruction est donnée dans l'annexe K.

5.3.1 Instruction de branchement inconditionnel

Il n'y a qu'une seule instruction de branchement inconditionnel dont le code mnémonique est **goto**. L'étiquette qui apparaît comme paramètre dans cette instruction est l'adresse de branchement (voir K.2.1).

Opérandes	
Paramètres dans l'instruction : étiquette	Opérandes sur la pile :

Mnémonique Paramètres Évolution de la pile



5.3.2 Instructions de branchement conditionnel

Les instructions de branchement conditionnel comportent une opération de comparaison exprimée comme suit dans la notation infixée :

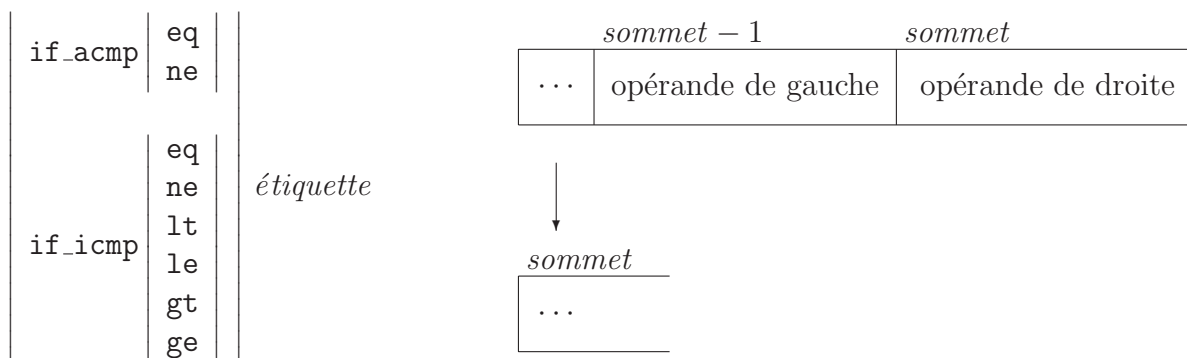
opérande de gauche *op* opérande de droite,

où *op* est un opérateur de comparaison parmi les suivants : **eq** pour égal, **ne** pour différent, **lt** pour plus petit, **le** pour plus petit ou égal, **gt** pour plus grand et **ge** pour plus grand ou égal. L'étiquette qui apparaît comme paramètre dans cette instruction représente l'adresse de branchement si la condition est satisfaite.

Opérandes	
Paramètres dans l'instruction : étiquette	Opérandes sur la pile : opérande de gauche opérande de droite

Le code mnémorique d'une instruction de branchement conditionnel dépend du type des données au sommet de la pile (**a** pour référence et **i** pour **int**) et de l'opération de comparaison (voir K.2.3 et K.2.5).

Mnémorique Paramètres Évolution de la pile



Il existe une variante des instructions de branchement conditionnel avec un seul opérande sur la pile de travail, l'autre opérande étant implicitement la constante zéro. L'opération de comparaison s'exprime alors comme suit dans la notation infixée :

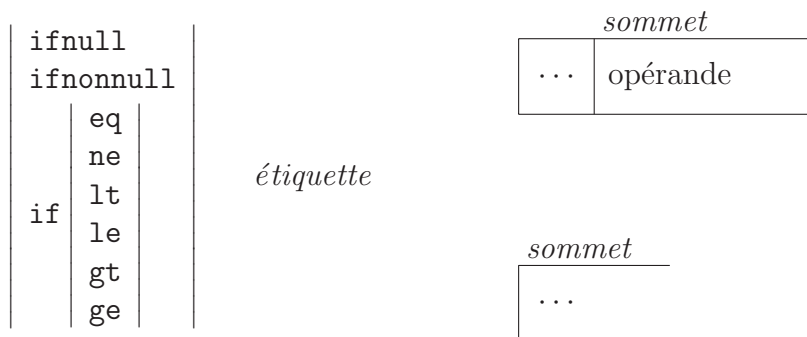
opérande *op* 0

où *op* est un opérateur de comparaison ou un code qui représente un test pour la valeur **null** (ou 0) ou une valeur différente de **null**.

Opérandes	
Paramètres dans l'instruction : étiquette	Opérandes sur la pile : opérande

Le code mnémorique d'une instruction de branchement conditionnel avec un seul opérande sur la pile de travail dépend du type de la donnée au sommet de la pile (**null** et **nonnull** pour référence et le type **int**) et de l'opération de comparaison (voir K.2.2 et K.2.4).

Mnémorique Paramètres Évolution de la pile



Pour le sous-type **boolean**, les instructions **ifeq** et **ifne** testent respectivement pour la valeur **false** et **true**, puisque la constante zéro représente la constante booléenne **false**.

5.3.3 Instructions de comparaison

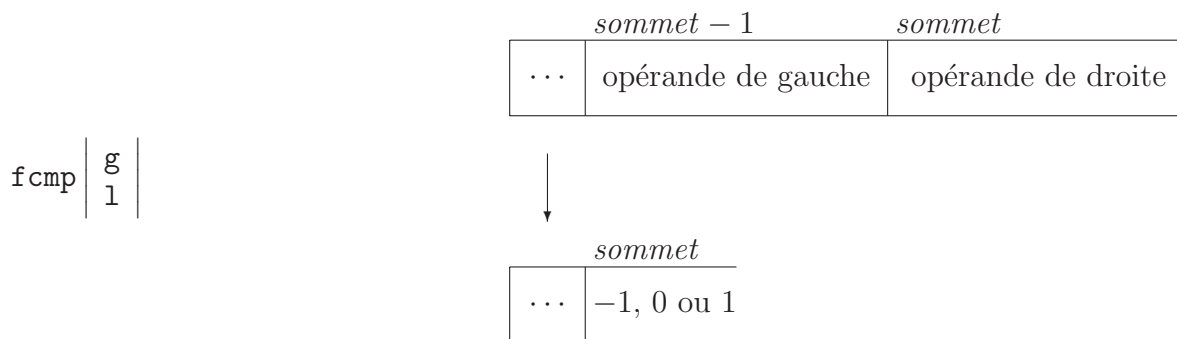
Les instructions de branchement conditionnel ne permettent pas la comparaison entre deux données de type `float`. Cela nécessite l'utilisation explicite d'instructions de comparaison qui empilent un résultat selon les conditions suivantes exprimées dans la notation infixée :

1 si opérande de gauche $>$ opérande de droite
 0 si opérande de gauche $=$ opérande de droite
 -1 si opérande de gauche $<$ opérande de droite

Opérandes	
Paramètres dans l'instruction :	Opérandes sur la pile : opérande de gauche opérande de droite

Le code mnémotechnique d'une instruction de comparaison dépend du type des données (`f` pour `float`) et des suffixes `g` et `l` qui indiquent deux façons de traiter la valeur NaN, c'est-à-dire *Not a Number* (voir K.2.6).

Mnémonique Paramètres Évolution de la pile



Pour simuler un branchement conditionnel, une instruction de comparaison doit être suivie d'une instruction de branchement avec un seul opérande de type entier sur la pile de travail.

5.3.4 Instructions arithmétiques

Les instructions arithmétiques comportent une opération arithmétique exprimée comme suit dans la notation infixée :

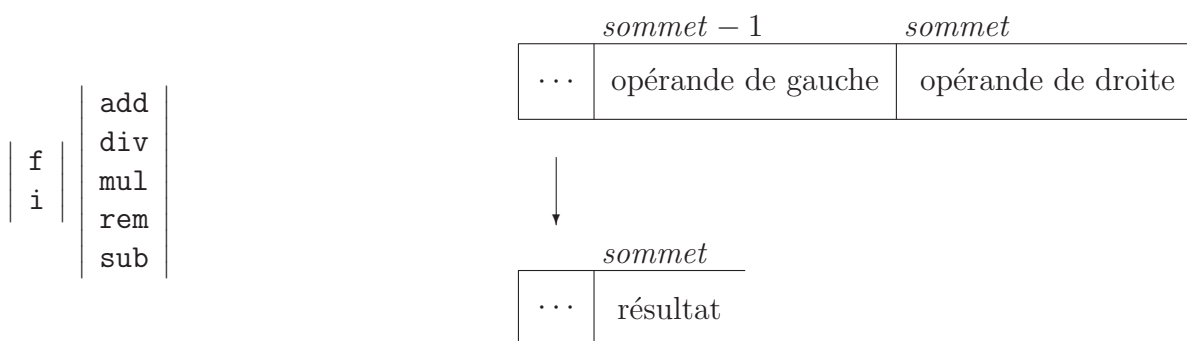
opérande de gauche *op* opérande de droite,

où *op* est un opérateur arithmétique parmi les suivants : `add` pour l'addition, `sub` pour la soustraction, `mul` pour la multiplication, `div` pour la division et `rem` pour le modulo.

Opérandes	
Paramètres dans l'instruction :	Opérandes sur la pile : opérande de gauche opérande de droite

Le code mnémonique d'une instruction arithmétique dépend du type des données (**f** pour float et **i** pour int) et de l'opération arithmétique (voir K.2.7 et K.2.8).

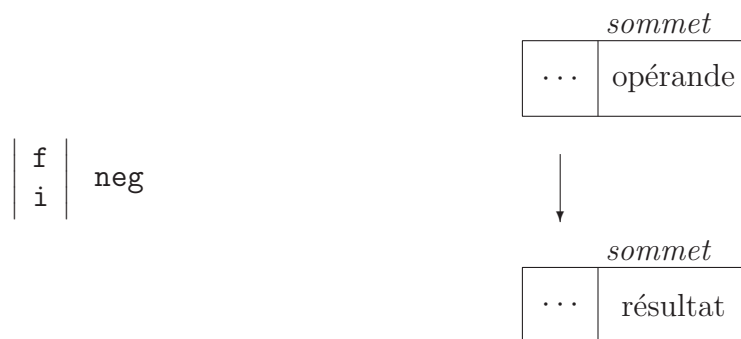
Mnémonique Paramètres Évolution de la pile



Contrairement aux opérations précédentes, l'opération de complément ne requiert qu'un opérande sur la pile de travail.

Opérandes	
Paramètres dans l'instruction :	Opérandes sur la pile : opérande

Mnémonique Paramètres Évolution de la pile



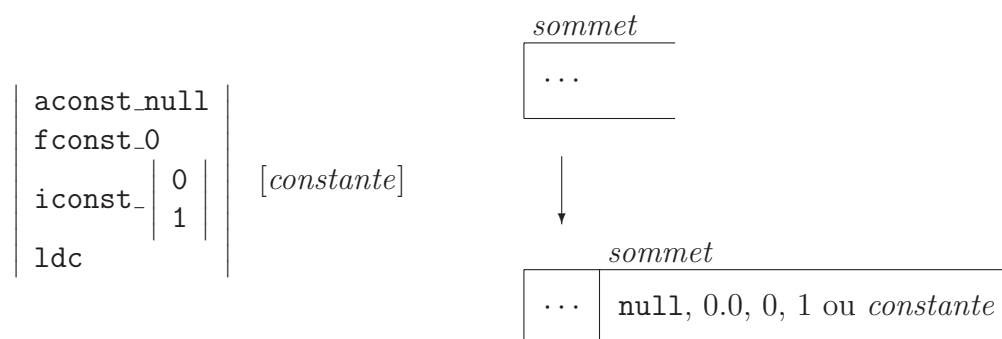
5.3.5 Instructions de chargement d'une constante

Il existe plusieurs instructions de chargement d'une constante au sommet de la pile de travail. Seule l'instruction `ldc` requiert un paramètre.

Opérandes	
Paramètres dans l'instruction : [constante]	Opérandes sur la pile :

Le code mnémotique d'une instruction de chargement d'une constante dépend du type de la constante (**a** pour référence, **f** pour **float** et **i** pour **int**) et de l'absence (constance implicite) ou de la présence (constante explicite) d'une constante dans l'instruction (voir K.2.9).

Mnémotique Paramètres Évolution de la pile



Une constante de type **float** comme paramètre de l'instruction **ldc** doit avoir le suffixe « **f** » ou « **F** » (par exemple 3.14159E0F).

Si une chaîne de caractères apparaît dans l'instruction **ldc**, une référence à l'objet constant de type `java.lang.String` contenant la chaîne de caractères est placée au sommet de la pile.

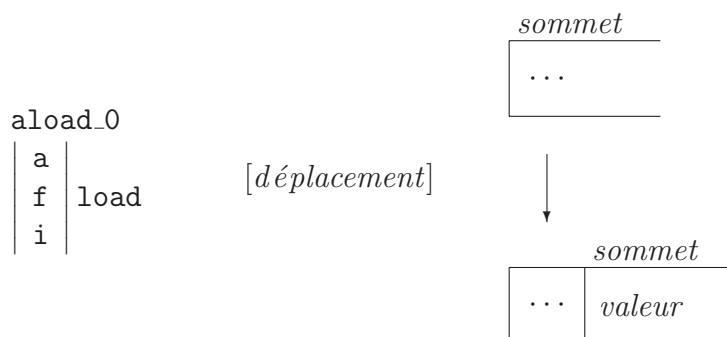
5.3.6 Instructions de chargement et de stockage d'une donnée locale

Les instructions de chargement de la valeur d'une donnée du bloc des variables locales au sommet de la pile de travail requiert le déplacement de la donnée par rapport au début du bloc des variables locales, sauf s'il apparaît dans le code mnémotique de l'instruction.

Opérandes	
Paramètres dans l'instruction : [déplacement]	Opérandes sur la pile :

Le code mnémotique d'une instruction de chargement dépend du type de la donnée (**a** pour **reference**, **f** pour **float** et **i** pour **int**) et de l'absence (déplacement implicite) ou de la présence (déplacement explicite) d'un déplacement dans l'instruction (voir K.2.10).

Mnémonique Paramètres Évolution de la pile

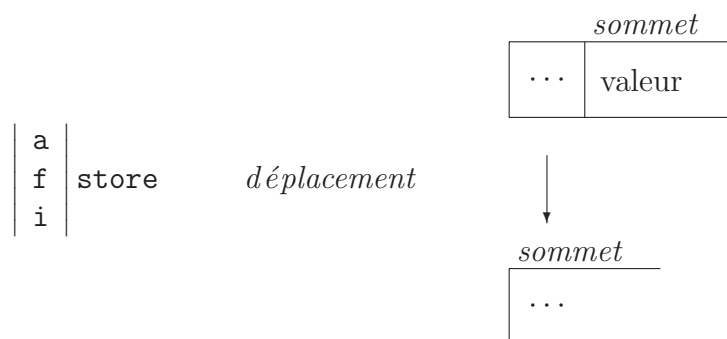


Les instructions de stockage de la valeur au sommet de la pile de travail dans le bloc des variables locales requiert un déplacement par rapport au début du bloc des variables locales.

Opérandes	
Paramètres dans l'instruction : déplacement	Opérandes sur la pile : valeur

Tout comme pour les instructions de chargement, il y a une instruction de stockage pour chaque type de données (voir K.2.11).

Mnémonique Paramètres Évolution de la pile



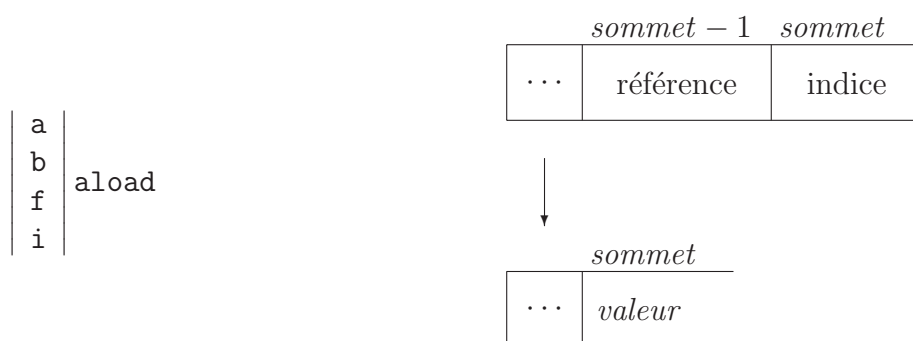
Des instructions spécifiques permettent d'accéder aux composantes d'un tableau.

Le chargement de la valeur d'une composante d'un tableau au sommet de la pile de travail s'effectue à partir d'une référence au tableau et d'un indice de la composante.

Opérandes	
Paramètres dans l'instruction :	Opérandes sur la pile : référence indice

Le code mnémonique d'une instruction de chargement dépend du type des composantes du tableau (`a` pour `reference`, `b` pour `byte`, `f` pour `float` et `i` pour `int`) (voir K.2.12).

Mnémonique Paramètres Évolution de la pile

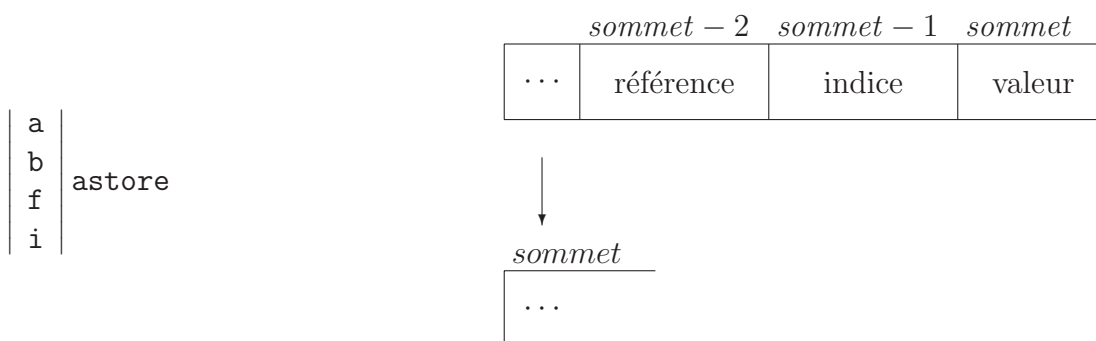


Le stockage de la valeur au sommet de la pile de travail dans une composante d'un tableau requiert aussi une référence au tableau et un indice de la composante.

Opérandes	
Paramètres dans l'instruction :	Opérandes sur la pile : référence indice valeur

Le code mnémonique d'une instruction de stockage dépend aussi du type des composantes du tableau (voir K.2.13).

Mnémonique Paramètres Évolution de la pile



Les tableaux de booléens sont considérés comme des tableaux d'octets (*bytes*), d'où la nécessité des instructions **baload** et **bastore**. Les tableaux à plusieurs dimensions sont des tableaux de tableaux, c'est-à-dire des tableaux de références, d'où la nécessité des instructions **aaload** et **aastore**.

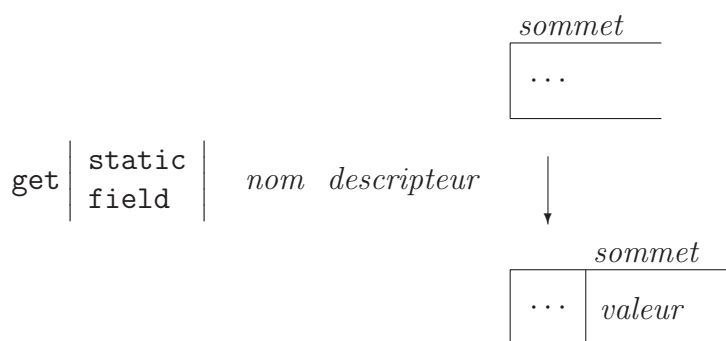
5.3.7 Instructions d'accès à un champ d'une classe

Le chargement de la valeur d'un champ de classe au sommet de la pile de travail est fait à l'aide de l'instruction **getstatic**. Celui de la valeur d'un champ d'instance est fait à l'aide de l'instruction **getfield**. Dans les deux cas, cette opération est faite à partir d'un nom qualifié (*Fully Qualified Name*) et d'un descripteur de type (*Type Encode JNI*).

Opérandes	
Paramètres dans l'instruction : nom qualifié descripteur de type	Opérandes sur la pile :

Le code mnémotique dépend des modificateurs associés à un champ (voir K.2.14). Le descripteur de type JNI est construit à l'aide des caractères **I** (pour **int**), **F** (pour **float**), **Z** (pour **boolean**), **S** (pour **short**), **B** (pour **byte**), **C** (pour **char**) et **[** (pour un tableau). En particulier, le descripteur de type `[[[I` représente le type `int[][][]`.

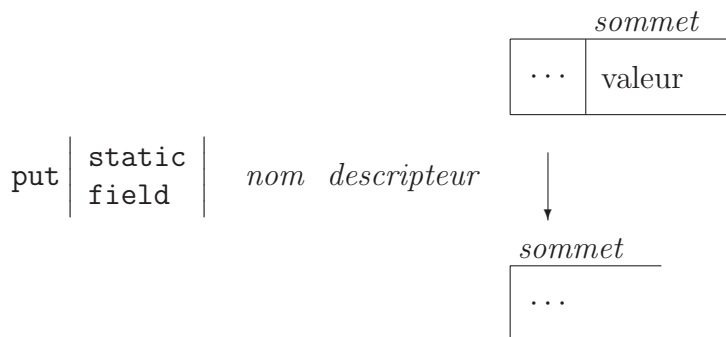
Mnémotique Paramètres Évolution de la pile



Le stockage de la valeur de la donnée au sommet de la pile de travail dans un champ de classe est fait à l'aide de l'instruction `putstatic`. Celui de la valeur de la donnée au sommet de la pile de travail dans un champ d'instance est fait à l'aide de l'instruction `putfield`.

Opérandes	
Paramètres dans l'instruction : nom qualifié descripteur de type	Opérandes sur la pile : valeur

Mnémotique Paramètres Évolution de la pile



5.3.8 Instructions de transfert de contrôle

L'appel d'une méthode de classe est fait à l'aide de l'instruction `invokestatic`. Celui d'une méthode d'instance est fait à l'aide de l'instruction `invokevirtual`. L'appel d'un constructeur est fait à l'aide de l'instruction `invokenonvirtual`. Dans tous les cas, cette opération est faite à partir d'un nom qualifié de la méthode immédiatement suivi de sa signature JNI (*FQNameAndSignature*). Les paramètres actuels doivent être au sommet de la pile précédés de la référence à un objet dans le cas d'une méthode d'instance.

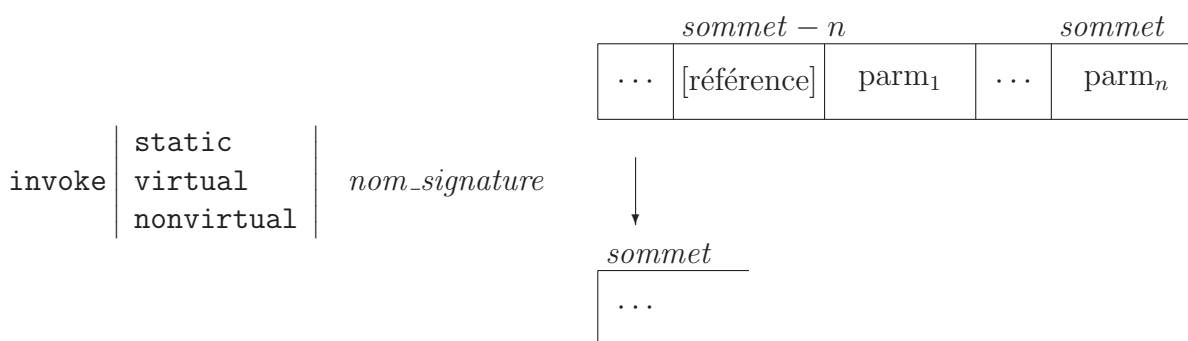
Opérandes	
Paramètres dans l'instruction : nom et signature	Opérandes sur la pile : [référence] premier paramètre : dernier paramètre

Le code mnémonique d'une instruction d'appel d'une méthode dépend des modificateurs associés à la méthode (voir K.2.15).

Mnémonique

Paramètres

Évolution de la pile

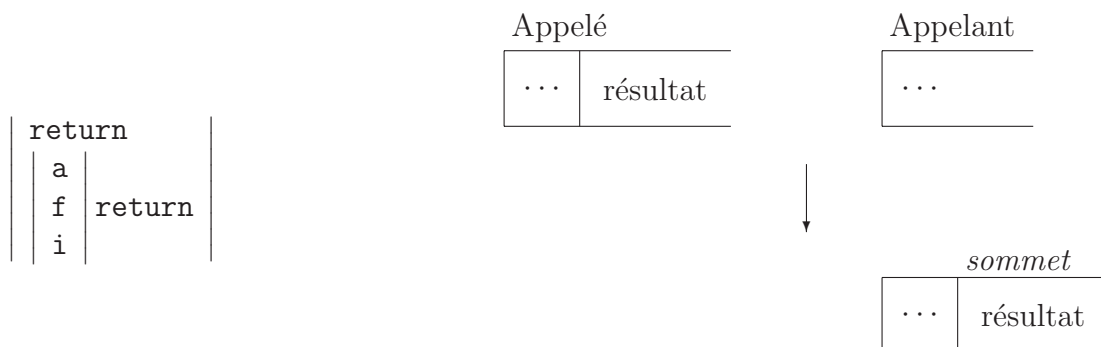


Le retour à l'appelant est fait à l'aide d'une instruction de retour. Si une valeur est retournée, elle doit être au sommet de la pile de travail du bloc d'activation de l'appelé. L'exécution d'une instruction de retour provoque le transfert de cette valeur vers le sommet de la pile de travail du bloc d'activation de l'appelant. Le bloc d'activation de l'appelé est ensuite détruit.

Opérandes	
Paramètres dans l'instruction :	Opérandes sur la pile : [valeur]

Le code mnémonique d'une instruction de retour dépend du type de la donnée retournée (voir K.2.16).

Mnémonique Paramètres Évolution des piles

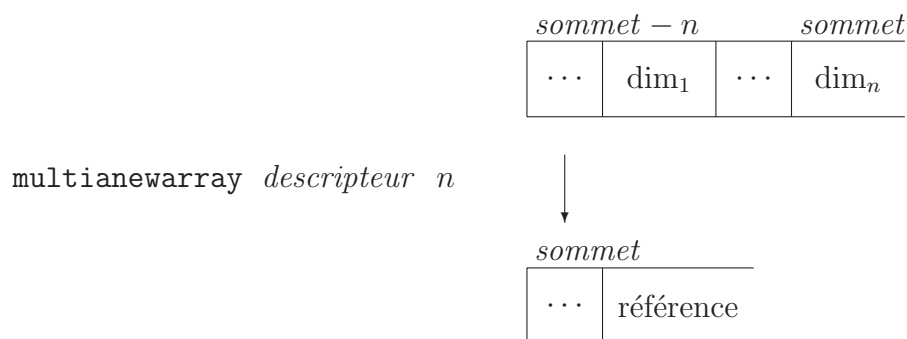


5.3.9 Instruction de création d'un tableau

La création d'un tableau à une ou plusieurs dimensions est faite à partir de la taille de chaque dimension. Le nombre de dimensions du tableau n doit être plus petit ou égal au nombre de dimensions spécifié dans la définition d'un tableau, celui éventuellement lié à la référence créée lors de l'exécution de cette instruction (voir K.2.17).

Opérandes	
Paramètres dans l'instruction : descripteur n	Opérandes sur la pile : dimension ₁ ... dimension _{n}

Mnémonique Paramètres Évolution de la pile



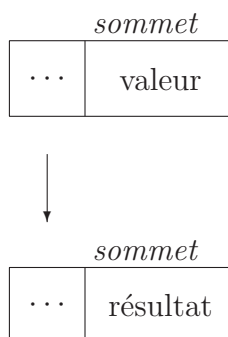
5.3.10 Instruction de conversion de type

Il existe plusieurs instructions de conversion de type. Mais une seule est utilisée dans la génération de code, celle qui permet de convertir une donnée de type `int` en une donnée de type `float` (voir K.2.18).

Opérandes	
Paramètres dans l'instruction :	Opérandes sur la pile : valeur

Mnémonique	Paramètres	Évolution de la pile
------------	------------	----------------------

i2f



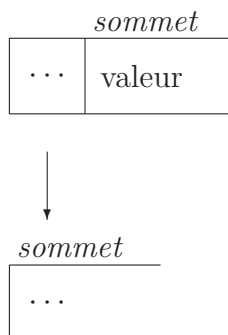
5.3.11 Instructions de manipulation de la pile de travail

Les instructions de manipulation de la pile de travail permettent de retirer des éléments au sommet de la pile (voir K.2.19).

Opérandes	
Paramètres dans l'instruction :	Opérandes sur la pile : valeur

Mnémonique	Paramètres	Évolution de la pile
------------	------------	----------------------

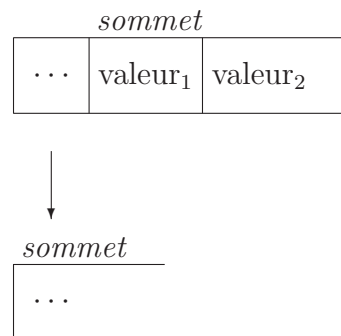
pop



Opérandes	
Paramètres dans l'instruction :	Opérandes sur la pile : valeur ₁ valeur ₂

Mnémonique Paramètres Évolution de la pile

pop2

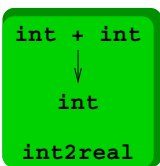


5.4 Exercices

1. Écrivez un petit programme en langage d'assemblage `JasminXT`.
2. Assemblez un petit programme en langage d'assemblage `JasminXT` à l'aide de l'assembleur `Jasmin`.
3. Exécutez le programme obtenu à la question précédente.
4. Traduisez un petit programme *Java* en langage d'assemblage `JasminXT`.

Chapitre 6

Le système de vérification et d'inférence de types



Le système de vérification et d'inférence de type est un module utilitaire utilisé par l'analyseur sémantique et le générateur de code. Il comporte un ensemble de fonctions, de procédures et de prédicats composés d'un ensemble de règles qui sont mises en correspondance avec leurs paramètres. Ces derniers sont des noeuds de l'arbre syntaxique abstrait, des attributs ou des valeurs. Le format des règles et la technique d'appariement sont décrits dans le chapitre 7. L'outil **puma** permet d'obtenir automatiquement le système de vérification et d'inférence de types à partir d'une spécification contenue dans le fichier `TypeSys.pum`. L'annexe H contient une copie de ce fichier avec une description succincte.

Les symboles *aNaT* (*Not a Type*) et *aErrorType* apparaissent souvent dans des fonctions. Ils permettent de traiter les cas d'erreurs.

6.1 Fonctions et prédicats pour l'analyseur sémantique

Les prédicats suivants sont appelés par l'analyseur sémantique :

- le prédicat *IsVarDesc* vérifie si un noeud correspond à un descripteur de variable ;
- le prédicat *IsStaticMbr* vérifie si un membre (champ ou méthode) d'une classe a le modificateur **static**.
- le prédicat *IsScalarType* vérifie si un type est un type primitif ;
- le prédicat *AssignmentCompatible* vérifie si le type du membre de gauche et le type du membre de droite d'une affectation sont compatibles, en particulier pour les tableaux à partir d'appels récursifs au prédicat auxiliaire *TypeEqv* qui vérifie si deux types sont structurellement équivalents ;
- le prédicat *CheckParamNbr* vérifie si le nombre de paramètres actuels est égal au nombre de paramètres formels ;
- le prédicat *CheckPrmTypes* vérifie si le type de chaque paramètre actuel est compatible avec le type du paramètre formel correspondant.

Les fonctions suivantes sont appelées par l'analyseur sémantique :

- la fonction *GetElementType* retourne le type des composantes d'un tableau ;

- la fonction *TypeSize* retourne la taille d'une donnée en fonction de son type ;
- la fonction *CoercedTypeSize* retourne la taille d'une donnée en fonction du transtypage (*Coercion* en anglais) appliqué à son type ;
- la fonction *BinaryResultType* retourne le type du résultat d'une opération binaire en fonction du type de ses opérandes et de son opérateur ;
- la fonction *BinaryOpType* retourne le type d'une opération binaire en fonction du type de ses opérandes et de son opérateur ;
- la fonction *UnaryResultType* retourne le type du résultat d'une opération unaire en fonction du type de son opérande et de son opérateur ;
- la fonction *GetNextFormal* retourne un pointeur vers le prochain paramètre d'une liste de paramètres formels ;
- la fonction *GetFormalType* retourne le type d'un paramètre formel ;
- la fonction *Coerce* retourne le type de transtypage à partir de deux types.

6.2 Fonctions et prédicats pour le générateur de code

Les prédicats suivants sont appelés par le générateur de code :

- le prédicat *IsPrimary* vérifie si une expression est une expression primaire ;
- le prédicat *IsBoolean* vérifie si un type est le type booléen.

Les fonctions suivantes sont appelées par le générateur de code :

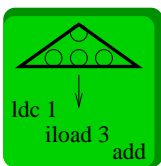
- la fonction *PushDefaultValue* retourne le code mnémorique pour le chargement de la valeur par défaut d'une donnée entière ou booléenne, d'une donnée en point flottant et d'une référence ;
- la fonction *TypedOps* retourne le code mnémorique d'une opération en lui ajoutant un préfixe (**a**, **f** ou **i**), représentant le type de l'opération, à partir d'un appel à la fonction auxiliaire *TypeCode* ;
- la fonction *ArrayTypedOps* retourne le code mnémorique d'une opération en lui ajoutant un préfixe (**aa**, **ba**, **fa** ou **ia**), représentant le type de l'opération, à partir d'un appel à la fonction auxiliaire *ArrayTypeCode* ;
- la fonction *EncodeType* retourne le format JNI d'un type par concaténation de caractères (**I**, **F**, **V**, **Z** et **,** **[]**) à partir d'appels récursifs à la procédure auxiliaire *BuildJNI* ;
- la fonction *EncodeSignature* retourne la signature d'une méthode en format JNI, par rapport à ses paramètres formels et au type de son résultat, à partir d'appels récursifs aux procédures auxiliaires *PrmsTypes* et *BuildJNI* ;
- la fonction *Ops* retourne le code mnémorique d'une instruction arithmétique ou logique à partir du code numérique d'un opérateur ;
- la fonction *TypedRead* retourne la concaténation d'un nom et de la signature d'une méthode de lecture de l'environnement *java--*.

6.3 Exercice

1. Examinez le contenu des fichiers `TypeSys.h` et `TypeSys.c` générés par l'outil `puma`.

Chapitre 7

Le générateur de code



Le générateur de code produit un programme écrit en langage d'assemblage **JasminXT** à partir de l'arbre syntaxique abstrait décoré par l'analyseur sémantique. L'outil **puma** permet d'obtenir automatiquement un tel générateur de code à partir de procédures de parcours de l'arbre syntaxique abstrait contenues dans le fichier **GenCode.pum**. L'annexe F contient une

copie de ce fichier avec une description succincte.

Les techniques de génération de code mises en oeuvre dans le compilateur *java--* sont très proches de celles introduites dans le livre de Grune et al [12]. En particulier les trois problèmes classiques rencontrés en génération de code sont résolus de la façon suivante :

- la sélection du code (*Code Selection*) est réalisée selon une transcription fixe déterminée a priori pour chaque forme rencontrée dans la syntaxe abstraite et utilisée pour traduire toute occurrence de cette forme présente dans l'arbre syntaxique abstrait ;
- l'allocation et l'affectation des registres (*Register Allocation and Assignment*) ne constituent pas un problème, car il n'y a pas de registre d'usage général dans la machine virtuelle *Java* ;
- le code est généré d'une manière séquentielle (*Linearization*) en respectant le flot des données dans les expressions et le flux de contrôle autrement.

La génération de code se fait par écriture directe dans un fichier de code intermédiaire **JasminXT** avec l'extension ".j". Par exemple, l'instruction

```
fprintf( fp, "\t.super\tjava/lang/Object\n" );
```

génère la directive **.super** qui identifie la super classe de la classe. La ligne de code

```
.super          java/lang/Object
```

apparaît donc dans le fichier de code intermédiaire.

Une première lecture de ce chapitre avec un examen rapide des procédures contenues dans le fichier **GenCode.pum** est nécessaire afin de comprendre tous les schémas de génération de code : ceux de nature générale et ceux associés à chaque partie d'un programme *Java*. Ceci permettra au lecteur d'identifier les attributs requis dans chaque schéma de génération de code et d'apprécier davantage les traitements faits par l'analyseur sémantique. Une fois ces traitements bien assimilés, une deuxième lecture avec un examen approfondi des procédures

de génération de code, permettra au lecteur d'avoir une vision globale du fonctionnement du compilateur *java--*.

7.1 Structure des schémas de génération de code

Toutes les procédures du générateur de code suivent le même schéma et comportent trois sections :

- une section optionnelle de préconditions,
- une section obligatoire de génération de code,
- une section optionnelle de détection de cas non traités.

Les sections optionnelles provoquent une erreur fatale qui termine brutalement la génération de code. Ces erreurs sont irrécupérables, car il y a violation des préconditions supposées satisfaites à l'appel d'une procédure (pour la première section) ou détection d'un cas non traité par la logique de génération de code (pour la troisième section).

La section obligatoire contient le traitement de toutes les occurrences de formes valides pouvant être rencontrées dans l'arbre syntaxique abstrait. Ainsi, la génération de code termine normalement si aucune des formes appartenant à une des sections optionnelles n'a été identifiée. Dans le cas contraire, il y a nécessairement une erreur, soit dans la logique de la construction de l'arbre syntaxique abstrait (par exemple, un noeud non initialisé représenté par la valeur *NoTree*), soit dans le générateur de code (par exemple, un appel de procédure avec des paramètres actuels incorrects).

Chaque procédure est composée d'un ensemble de règles placées les unes à la suite des autres. Chaque règle comporte une forme (*Pattern*) et une clause conséquente, constituée d'une suite d'instructions. La forme et les instructions sont séparées par le symbole « :- » et le tout se termine par un point :

```
<forme>
:-
<clause conséquente>
.
```

Une forme est généralement associée à un type de noeud d'un sous-arbre de l'arbre syntaxique abstrait et elle peut être décomposée par rapport aux descendants de ce type de noeud. Chaque décomposition commence par un type de noeud suivi d'une liste de formes entre parenthèses. Un type de noeud peut être qualifié par un nom d'arbre et être précédé d'une étiquette suivie du symbole « := ». Par exemple, dans la règle

```
aReturnNoValueStatement(
  Pos := Pos,
  Next := Next:aStatements( ... )
)
:-
fprintf( fp, "\t.line\t%i\n", Pos.Line );
fprintf( fp, "\treturn\n", );
CodeStatements( Next );
.
```


la forme *aReturnNoValueStatement*, qui est un type de noeud, est décomposée en deux formes. La première (*Pos*) est une forme élémentaire qui correspond à un attribut. Elle est précédée de l'étiquette *Pos*. La deuxième (*aStatements*) est une forme non élémentaire qui correspond à un type de noeud qualifié par le nom *Next* suivi du deux-points. L'étiquette *Next* précède le tout. La suite de points (« .. ») à l'intérieur de la deuxième forme indique un nombre quelconque d'attributs ou de types de noeuds sans les mentionner explicitement.

Une règle peut échouer ou réussir. Le résultat d'une règle dépend du succès de l'appariement de la forme et du succès de l'exécution des instructions. Dans le cas positif, l'appel de procédure prend fin normalement. Si une des actions de la règle échoue, l'appariement de la forme ou l'exécution des instructions prend fin immédiatement et la prochaine règle est alors essayée.

L'appariement entre une forme et les éléments d'un noeud de l'arbre syntaxique abstrait est défini récursivement selon le type de la forme. Une forme avec un type de noeud *t* correspond à un noeud *n* de type *s* si $n \neq \text{NoTree}$, $s = t$ ou *s* est un sous-type de *t*, et les sous-formes de *t* correspondent aux attributs et aux sous-arbres de *n*. Si une étiquette apparaît dans une forme, alors il y a une liaison entre l'étiquette et l'élément correspondant. La forme NIL correspond seulement à un noeud de valeur NoTree. Si l'appariement réussit, la clause conséquente est déclenchée.

Le manuel de référence de l'outil **puma** contient une description détaillée de la procédure d'appariement et de l'usage de la clause conséquente [7].

7.2 Organisation du générateur de code

Le générateur de code comporte 13 procédures dont une seule est visible de l'extérieur, la procédure *GenCode* (section 7.3). Toutes les autres procédures sont des procédures locales.

Le générateur de code inclut aussi deux fonctions statiques *C*, *ResetLbl* et *GetNextLbl*, qui permettent la génération d'étiquettes uniques. Ces étiquettes sont insérées dans des instructions du code intermédiaire.

7.3 Génération de code pour une unité de compilation

La procédure de génération de code pour une unité de compilation, nommée *GenCode*, sauvegarde le nom du fichier source et le nom du *package* par défaut dans des variables globales statiques à partir de valeurs d'attributs du noeud de type *aCompilationUnit*. Elle déclenche la génération le code pour les déclarations de types d'une unité de compilation à l'aide d'un appel de la procédure *GenTypes* (section 7.4) avec comme paramètre la liste des déclarations de types (*typeList*).

7.4 Génération de code pour les types

La procédure de génération de code pour les types, nommée *GenTypes*, parcourt une liste de déclarations de types à l'aide d'appels récursifs en utilisant l'attribut *Next* comme paramètre.

Pour chaque déclaration de type, elle construit tout d'abord le nom complet du type à partir du nom du *package* et de son nom. Le nom complet avec comme suffixe « .j » est alors utilisé comme nom de fichier de code intermédiaire (incluant le chemin d'accès). Elle ouvre ensuite le fichier en mode écriture et elle écrit dans ce fichier les directives suivantes :

```
.source      nom du fichier source
.class      modificateurs  nom complet du type
.super      java/lang/Object
```

Une fois l'en-tête du fichier de code intermédiaire généré, la procédure *GenTypes* appelle :

- la procédure *CodeFields* (section 7.5) pour générer les déclarations de champs ;
- la procédure *Code_clinit* (section 7.6) pour générer la méthode d'initialisation des champs statiques ;
- la procédure *Code_init* (section 7.7) pour générer le constructeur par défaut ;
- la procédure *Code_main* (section 7.8) pour générer la méthode qui appelle la méthode **main** de la classe, s'il y a lieu ;
- la procédure *CodeMethods* (section 7.9) pour générer le code des méthodes.

Enfin, la procédure *GenTypes* ferme le fichier de code intermédiaire et traite la prochaine déclaration de type.

7.5 Génération de code pour les champs

La procédure de génération de code pour les champs, nommée *CodeFields*, parcourt une liste de déclarations de champs et de méthodes à l'aide d'appels récursifs en utilisant l'attribut *Next* comme paramètre. Pour chaque déclaration de champ, elle parcourt une liste d'identificateurs (*nameList: aFieldDeclarators*), toujours à l'aide d'appels récursifs, en utilisant l'attribut *Next* comme paramètre. L'assembleur *Jasmin* exige que les déclarations de champs précèdent les définitions des méthodes. Pour chaque champ, cette procédure écrit dans le fichier de code intermédiaire la directive **.field** avec les paramètres appropriés. Le code résultat a la forme suivante :

```
{ .field      modificateurs  'nom du champ'  descripteur de type JNI }
```

7.6 Génération de code pour la méthode d'initialisation des champs statiques

La procédure de génération de code pour la méthode d'initialisation des champs statiques, nommée *Code_clinit*, est divisée en trois parties. Elle écrit tout d'abord le code de début de la méthode d'initialisation des champs statiques. Cette méthode publique et statique a nécessairement comme nom **<clinit>**. Elle est appelée par la machine virtuelle *Java* lors du chargement de la classe afin que chaque champ de classe possède une valeur par défaut. La méthode **<clinit>** n'a pas de paramètre (présence de **()**) et elle ne retourne pas de résultat (présence de **V**). Son bloc de variables locales est vide et la taille de sa pile de travail est de deux mots. Le code de début est constitué de trois directives :

```
.method      static public <clinit>()V
.limit      locals 0
.limit      stack 2
```

La procédure *Code_clinit* parcourt ensuite la liste de déclarations de champs et de méthodes de la même manière que la procédure *CodeFields*. Pour l'ensemble des champs de classe (champs statiques), elle produit le code qui a la forme suivante selon le type de chaque champ de classe :

$$\left\{ \begin{array}{l} \left| \begin{array}{l} \text{aconst_null} \\ \left| \begin{array}{l} \text{f} \\ \text{i} \end{array} \right| \text{const_0} \end{array} \right| \\ \text{putstatic} \quad \text{nom complet du type/nom du champ} \quad \text{descripteur de type JNI} \end{array} \right\}$$

Enfin, la procédure *Code_clinit* écrit le code de fin de la méthode d'initialisation :

```
return
.end          method
```

7.7 Génération de code pour le constructeur par défaut

La procédure de génération de code pour le constructeur par défaut, nommée *Code_init*, écrit dans le fichier de code intermédiaire la suite de directives et d'instructions suivante :

```
.method      public <init>()V
.limit      locals 1
.limit      stack 2
aload_0
invokenonvirtual  Java/lang/Object/<init>()V
return
.end          method
```

Elle peut aussi insérer du code juste avant l'instruction **return** pour initialiser les champs d'instance de tout objet (ceci est présentement impossible, car la version actuelle du langage *java--* ne permet pas de tels champs). Tout comme la procédure *Code_clinit*, elle parcourt la liste de déclarations de champs et de méthodes. Pour l'ensemble des champs d'instance (ceux qui n'ont pas le modificateur **static**), elle produit le code qui a la forme suivante selon le type de chaque champ d'instance :

$$\left\{ \begin{array}{l} \left| \begin{array}{l} \text{aconst_null} \\ \left| \begin{array}{l} \text{f} \\ \text{i} \end{array} \right| \text{const_0} \end{array} \right| \\ \text{putfield} \quad \text{nom complet du type/nom du champ} \quad \text{descripteur de type JNI} \end{array} \right\}$$

Le constructeur par défaut d'une classe qui a nécessairement comme nom `<init>` appelle le constructeur par défaut de sa super classe (la classe *Object*). L'instruction `aload_0` juste avant l'appel charge la référence vers l'objet `this` contenu dans le premier emplacement du bloc des variables locales. Il positionne aussi la valeur par défaut de chaque champ d'instance de la classe. Ce constructeur est appelé par la machine virtuelle *Java* lors de la création d'une instance de la classe.

7.8 Génération de code pour l'appel de la méthode statique `main`

La procédure de génération de code pour l'appel de la méthode statique `main`, nommée *Code_main*, écrit dans le fichier de code intermédiaire la suite de directives et d'instructions suivantes :

```
.method      public static main([Ljava/lang/String;)V
.limit      locals 1
.limit      stack 0
invokestatic nom complet du type/main()V
return
.end        method
```

Dans le langage *java--*, toute classe doit contenir une méthode statique nommée `main`. La machine virtuelle *Java* amorce son exécution à l'aide du code généré par la procédure *Code_main*.

7.9 Génération de code pour les méthodes

La procédure de génération de code pour les méthodes, nommée *CodeMethods*, parcourt une liste de déclarations de champs et de méthodes à l'aide d'appels récursifs en utilisant l'attribut *Next* comme paramètre. Pour chaque déclaration de méthode, elle écrit tout d'abord le code de début de la méthode constitué de trois directives (`.method`, `.limit locals` et `.limit stack`).

Elle génère ensuite deux étiquettes qui sont associées au début (L_i) et à la fin (L_{i+1}) du bloc des énoncés de la méthode. Ces étiquettes apparaissent comme des paramètres dans les directives `.var`. Elles fournissent l'information de débogage, c'est-à-dire la portée des paramètres formels et des variables locales de niveau 0. Ainsi, la procédure *CodeMethods* appelle la procédure *CodeVarDirectives* (section 7.16) deux fois, une première fois pour les paramètres formels et une deuxième fois pour les variables locales afin de générer les directives `.var` appropriées.

Puis, la procédure *CodeMethods* effectue les traitements suivants :

- écriture de l'étiquette L_i dans le fichier de code intermédiaire ;
- appel de la procédure *CodeStatements* (section 7.10) qui génère le code pour tous les énoncés de la méthode ;
- écriture de l'étiquette L_{i+1} dans le fichier de code intermédiaire.

Enfin, la procédure *CodeMethods* écrit le code de fin de la méthode. L'instruction **return** n'est écrite dans le fichier de code intermédiaire que si la méthode ne retourne aucun résultat. Le code qui résulte de tous ces traitements a la forme suivante :

```
.method    modificateurs    nom de la méthode•signature JNI
.limit     locals taille du bloc des variables locales
.limit     stack taille de la pile de travail
{ .var      déplacement is 'nom' descripteur de type JNI from  $L_i$  to  $L_{i+1}$  }
```

L_i :
code des énoncés de la méthode

L_{i+1} :
 [**return**]
 .end

7.10 Génération de code pour les énoncés

La procédure de génération de code pour les énoncés, nommée *CodeStatements*, parcourt une liste d'énoncés d'une méthode à l'aide d'appels récursifs en utilisant l'attribut *Next* comme paramètre. En général, il y a une règle pour chaque type d'énoncé.

7.10.1 Énoncé de déclaration de variables locales

Il y a deux règles pour la génération de code associé à une déclaration de variables locales. La première règle permet l'appariement avec un noeud de type *aLocalVariableDeclaration*. La clause conséquente de cette règle écrit tout d'abord la directive **.line** dans le fichier de code intermédiaire. Cette directive fournit l'information de débogage, en particulier le numéro de la ligne qui contient la déclaration de variables locales dans le fichier source du programme *java--*. Ensuite, un appel récursif à la procédure *CodeStatements*, avec comme paramètre la liste des variables déclarées dans un tel énoncé de déclaration, est exécuté. La deuxième règle, qui permet l'appariement avec un noeud de type *aVariableDeclarator*, est alors déclenchée de façon à produire le code d'initialisation de chaque variable, une à la fois, à l'aide d'un appel récursif à la procédure *CodeStatements* en utilisant l'attribut *Next* comme paramètre. Le code résultant a la forme suivante qui dépend du type de chaque variable :

```
.line          numéro de ligne de l'énoncé
{
  ( aconst_null
    astore      déplacement )
  ( fconst_0
    fstore      déplacement )
  ( iconst_0
    istore      déplacement )
}
```

7.10.2 Énoncé de bloc

La règle pour la génération de code associé à un bloc permet l'appariement avec un noeud de type *aInnerBlock*. La clause conséquente de cette règle effectue un traitement semblable à celui fait dans la procédure *CodeMethods*.

Premièrement, elle génère deux étiquettes qui sont associées au début (L_i) et à la fin (L_{i+1}) du bloc d'énoncés. Ces étiquettes apparaissent comme des opérandes dans les directives `.var`, lesquelles fournissent l'information de débogage, c'est-à-dire la portée des variables locales du bloc.

Deuxièmement, un appel à la procédure *CodeVarDirectives* (section 7.16) permet de générer les directives `.var` appropriées pour chaque variable.

Troisièmement, la clause conséquente effectue les traitements suivants :

- écriture de l'étiquette L_i dans le fichier de code intermédiaire ;
- appel de la procédure *CodeStatements* (section 7.10) qui génère le code pour tous les énoncés du bloc ;
- écriture de l'étiquette L_{i+1} dans le fichier de code intermédiaire.

Tous ces traitements produisent du code qui a la forme suivante :

```
{ .var      déplacement is 'nom' descripteur de type JNI from  $L_i$  to  $L_{i+1}$  }
```

L_i :
code des énoncés du bloc

L_{i+1} :

7.10.3 Énoncé if-then

La règle pour la génération de code associé à un énoncé `if-then` permet l'appariement avec un noeud de type *aIfThen*. La clause conséquente de cette règle effectue les traitements suivants :

- génération d'une étiquette (L_i) associée à la fin de la partie `then` ;
- écriture de la directive `.line` avec comme numéro de ligne celui de l'énoncé dans le fichier source du programme `java--` ;
- appel de la procédure *CodeLazyExpr* (section 7.13) qui génère le code pour la condition selon la technique d'évaluation paresseuse à partir, entre autres, de la constante *NoLbl* indiquant qu'il n'y a aucun point de branchement (condition vraie) et de l'étiquette L_i (condition fausse) ;
- appel de la procédure *CodeStatements* (section 7.10) qui génère le code pour les énoncés de la partie `then` ;
- écriture de l'étiquette L_i .

Le code résultant a la forme suivante :

```
.line      numéro de ligne de l'énoncé  
code pour l'évaluation paresseuse de la condition  
code pour la partie then
```

L_i :

7.10.4 Énoncé if-then-else

La règle pour la génération de code associé à un énoncé **if-then-else** permet l'appariement avec un noeud de type *aIfThenElse*. La clause conséquente de cette règle effectue les traitements suivants :

- génération de deux étiquettes associées au début (L_i) et à la fin (L_{i+1}) de la partie **else** ;
- écriture de la directive `.line` avec comme numéro de ligne celui de l'énoncé dans le fichier source du programme *java--* ;
- appel de la procédure *CodeLazyExpr* (section 7.13) qui génère le code pour la condition selon la technique d'évaluation paresseuse à partir, entre autres, de la constante *NoLbl* indiquant qu'il n'y a aucun point de branchement (condition vraie) et de l'étiquette L_i (condition fausse) ;
- appel de la procédure *CodeStatements* (section 7.10) qui génère le code pour les énoncés de la partie **then** ;
- écriture de l'instruction `goto` avec comme opérande l'étiquette L_{i+1} ;
- écriture de l'étiquette L_i ;
- appel de la procédure *CodeStatements* (section 7.10) qui génère le code pour les énoncés de la partie **else** ;
- écriture de l'étiquette L_{i+1} .

Le code résultant a la forme suivante :

```

        .line      numéro de ligne de l'énoncé
        code pour l'évaluation paresseuse de la condition
        code pour la partie then
        goto      Li+1
Li  :
        code pour la partie else
Li+1 :
```

7.10.5 Énoncé while

La règle pour la génération de code associé à un énoncé **while** permet l'appariement avec un noeud de type *aWhileStatement*. La clause conséquente de cette règle effectue les traitements suivants :

- génération de deux étiquettes associées au début des énoncés (L_i) et au début de l'évaluation de la condition (L_{i+1}) ;
- écriture de la directive `.line` avec comme numéro de ligne celui de l'énoncé dans le fichier source du programme *java--* ;
- écriture de l'instruction `goto` avec comme opérande l'étiquette L_{i+1} ;
- écriture de l'étiquette L_i ;
- appel de la procédure *CodeStatements* (section 7.10) qui génère le code pour les énoncés ;
- écriture de l'étiquette L_{i+1} ;
- appel de la procédure *CodeLazyExpr* (section 7.13) qui génère le code pour la condition selon la technique d'évaluation paresseuse à partir, entre autres, de l'étiquette

L_i (condition vraie) et de la constante *NoLbl* indiquant qu'il n'y a aucun point de branchement (condition fausse).

Le code résultant a la forme suivante :

```

        .line      numéro de ligne de l'énoncé
goto       $L_{i+1}$ 
 $L_i$  :
        code pour les énoncés
 $L_{i+1}$  :
        code pour l'évaluation paresseuse de la condition

```

7.10.6 Énoncé return

Il y a deux règles pour la génération de code associé à un énoncé **return** selon qu'il y ait retour ou non d'une valeur par une méthode.

La première règle est spécifique à un noeud de type *aReturnNoValueStatement*. La clause conséquente de cette règle produit le code suivant :

```

        .line      numéro de ligne de l'énoncé
return

```

La deuxième règle permet l'appariement avec un noeud de type *aReturnValueStatement*. Si l'expression de l'énoncé de retour n'est pas une expression booléenne complexe, alors la clause conséquente appelle la procédure *CodeExpression* (section 7.12). Le code résultant a la forme suivante qui dépend de la signature de la méthode :

```

        .line      numéro de ligne de l'énoncé
        code pour l'évaluation de l'expression et le transtypage
        a |
        f | return
        i |

```

Si l'expression de l'énoncé de retour est une expression booléenne complexe, alors la clause conséquente appelle la procédure *CodeLazyExpr* (section 7.13) à partir, entre autres, de la constante *NoLbl* indiquant qu'il n'y a aucun point de branchement (expression évaluée à vrai) et de l'étiquette L_i préalablement générée (expression évaluée à faux). Le code résultant a la forme suivante :

```

        .line      numéro de ligne de l'énoncé
iconst_0
        code pour l'évaluation paresseuse de l'expression
pop
iconst_1
 $L_i$  :
        ireturn

```


7.10.7 Énoncé d'affectation

La règle pour la génération de code associé à un énoncé d'affectation permet l'appariement avec un noeud de type *aAssignment*. Le code résultant est constitué de trois fragments.

Le premier fragment de code contient la directive qui indique le numéro de la ligne de l'énoncé d'affectation dans le fichier source du programme *java--*. Un appel à la procédure *CodeExpression* (section 7.12), avec comme paramètre un pointeur sur la racine du sous-arbre du membre de gauche de l'affectation, permet dans le cas d'une affectation d'une valeur à une composante d'un tableau, de générer les instructions qui placent au sommet de la pile de travail la référence au tableau et l'indice de la composante. Ces deux opérandes et la valeur du membre de droite de l'affectation sont utilisés par l'instruction qui apparaît dans le troisième fragment de code. Le code résultant pour ce premier fragment de code a la forme suivante :

```
.line          numéro de ligne de l'énoncé
[ code pour empiler la référence à un tableau et l'indice d'une composante ]
```

Le deuxième fragment de code dépend du type de l'expression qui apparaît dans le membre de droite de l'affectation. Si le membre de droite n'est pas une expression booléenne complexe, alors la clause conséquente appelle la procédure *CodeExpression* (section 7.12) qui génère le code pour le membre de droite, c'est-à-dire pour une expression. Dans ce cas, le code résultant pour le deuxième fragment de code a la forme suivante :

code pour l'évaluation de l'expression et le transtypage

Si le membre de droite est une expression booléenne complexe, une évaluation paresseuse de cette expression ne produit pas de résultat sur la pile. Le résultat, c'est-à-dire 0 ou 1, est alors généré à l'aide d'instructions comme dans le cas d'un énoncé de retour. Dans ce cas, le code résultant pour le deuxième fragment de code a la forme suivante :

```
iconst_0
code pour l'évaluation paresseuse de l'expression
pop
iconst_1
```

L_i :

Quel que soit le type de l'expression du membre de droite, la clause conséquente appelle la procédure *CodeLeftValueStore* (section 7.11) qui génère le code pour stocker le résultat dans le membre de gauche de l'affectation. Le code résultant pour le troisième fragment de code a la forme suivante :

code pour stocker le résultat dans le membre de gauche

7.10.8 Appel d'une méthode

La règle pour la génération de code associé à un appel de méthode considéré comme un énoncé, c'est-à-dire qui n'apparaît pas dans une expression, permet l'appariement avec un

noeud de type *aProcedureCall*. La clause conséquente considère deux cas : celui pour l'appel d'une méthode de classe et celui pour l'appel non qualifié d'une méthode d'instance.

Bien que ce dernier cas soit impossible dans la version actuelle du langage *java--*, il indique, entre autres, que la référence à l'objet courant (représenté par **this**) doit être empilée avant les valeurs des paramètres actuels. La référence à l'objet courant se trouve alors dans le premier emplacement du bloc des variables locales. Ce cas n'est possible que si l'appel non qualifié de la méthode est fait dans un contexte d'instance.

Dans tous les cas, la clause conséquente appelle la procédure *CodeExpression* (section 7.12) qui génère le code pour l'évaluation des paramètres actuels. Comme un énoncé d'appel d'une méthode est équivalent à un énoncé d'appel de procédure, tout résultat retourné par une méthode de type autre que **void** est retiré de la pile de travail. Le code résultant a la forme suivante qui dépend des modificateurs de la méthode et de sa signature :

```
.line          numéro de ligne de l'énoncé
[ aload_0 ]    ; empiler la référence à l'objet courant pour
               l'appel non qualifié à une méthode d'instance
code pour l'évaluation des paramètres actuels
| invokestatic |   nom de la méthode•signature JNI
| invokevirtual|
|
| pop          |
| pop2         | ] ; retirer le résultat dans la pile de travail
```

7.10.9 Énoncés d'entrée/sortie

Il y a quatre règles pour la génération de code associé aux énoncés d'entrée-sortie. Ces règles traduisent des appels de méthodes simples dans des appels de méthodes de la plateforme *Java* pour les opérations de sortie et dans des appels de l'environnement d'exécution *java--* pour les opérations d'entrée.

La première règle permet l'appariement avec un noeud de type *aRead* et génère tout d'abord le code suivant pour l'écriture d'un message-guide (« > ») et la lecture d'une donnée de type simple :

```
.line          numéro de ligne de l'énoncé
               ; empiler System.out
getstatic      java/lang/System/out   Ljava/io/PrintStream;
ldc            "> "
invokevirtual   java/io/PrintStream/print(Ljava/lang/String;)V
[ code pour empiler la référence à un tableau et l'indice d'une composante ]
| mjitio/scanf/readF()F |
| mjitio/scanf/readI()I |
| mjitio/scanf/readZ()Z |
```

Ensuite, la clause conséquente appelle la procédure *CodeLeftValueStore* (section 7.11) qui génère le code pour stocker la donnée lue dans le paramètre de l'énoncé de lecture. Le code résultant a la forme suivante :

code pour stocker la donnée lue

La deuxième règle permet l'appariement avec un noeud de type *aPrintln* et génère le code suivant pour l'écriture d'un changement de ligne :

```
.line          numéro de ligne de l'énoncé
                ; empiler System.out
getstatic      java/lang/System/out  Ljava/io/PrintStream;
invokevirtual   java/io/PrintStream/println()V
```

La troisième règle permet l'appariement avec un noeud de type *aPrintStringLiteral* et génère le code suivant pour l'écriture d'une chaîne de caractères constante :

```
.line          numéro de ligne de l'énoncé
                ; empiler System.out
getstatic      java/lang/System/out  Ljava/io/PrintStream;
ldc            chaîne de caractères
invokevirtual   java/io/PrintStream/print(Ljava/lang/String;)V
```

La quatrième règle permet l'appariement avec un noeud de type *aPrintValue* et génère le code suivant pour l'écriture de la valeur d'une expression qui n'est pas une expression booléenne complexe :

```
.line          numéro de ligne de l'énoncé
                ; empiler System.out
getstatic      java/lang/System/out  Ljava/io/PrintStream;
code pour l'évaluation de l'expression et le transtypage

invokevirtual   java/io/PrintStream/print( 

|   |
|---|
| F |
| I |
| Z |

 )V
```

Si l'expression est une expression booléenne complexe, alors la clause conséquente de cette règle produit le code suivant :

```
.line          numéro de ligne de l'énoncé
                ; empiler System.out
getstatic      java/lang/System/out  Ljava/io/PrintStream;
iconst_0
code pour l'évaluation paresseuse de l'expression
pop
iconst_1
Li :
invokevirtual   java/io/PrintStream/print(Z)V
```

7.11 Génération de code pour le stockage d'une donnée

La procédure de génération de code pour stocker une donnée dans le membre de gauche d'une affectation ou dans le paramètre d'un énoncé *read*, nommée *CodeLeftValueStore*, traite trois cas.

- Pour le stockage d’une donnée dans une variable locale de type simple, le code résultant a la forme suivante qui dépend du type de la variable locale :

$$\left| \begin{array}{c} a \\ f \\ i \end{array} \right| \text{store} \quad \text{déplacement}$$

- Pour le stockage d’une donnée dans un champ de classe ou à un champ d’instance, le code résultant a la forme suivante qui dépend des modificateurs du champ :

$$\left| \begin{array}{c} \text{putstatic} \\ \text{putfield} \end{array} \right| \quad \text{nom complet du type/nom du champ} \quad \text{descripteur de type JNI}$$

- Pour le stockage d’une donnée dans une composante d’un tableau, le code résultant a la forme suivante qui dépend du type du tableau :

$$\left| \begin{array}{c} a \\ b \\ f \\ i \end{array} \right| \text{astore}$$

7.12 Génération de code pour les expressions

La procédure de génération de code pour les expressions, nommée *CodeExpression*, comporte plusieurs règles. Elles génèrent le code selon une évaluation complète par opposition à une évaluation paresseuse d’une expression booléenne. Elles effectuent un parcours récursif en post-ordre à partir de la racine d’un sous-arbre de l’arbre syntaxique abstrait correspondant à une expression qui n’est pas une expression booléenne complexe. Les expressions booléennes complexes ne sont pas considérées par cette procédure. Elles sont traitées dans la procédure *CodeLazyExpr* (section 7.13).

7.12.1 Règles pour les constantes

Il y a quatre règles pour la génération de code associé aux constantes. Chaque règle génère une instruction pour le chargement d’une constante dans la pile de travail.

Les premières règles permettent l’appariement avec un noeud de type *aBooleanConst*, *aIntConst* et *aFloatConst*. La partie conséquente de chacune de ces règles génère le code suivant :

`ldc` *valeur de la constante*

La dernière règle permet l’appariement avec un noeud de type *aNullConst* et génère le code suivant :

`aconst_null`

7.12.2 Règles pour un nom

Il y a plusieurs règles pour la génération de code associé à un nom.

Deux règles permettent l'appariement avec un noeud de type *arValue* selon qu'il s'agisse d'un nom d'une variable locale ou d'un nom de champ.

- Pour une variable locale de type simple ou de type tableau, le code résultant a la forme suivante qui dépend du type de la variable locale :

$$\left| \begin{array}{c} a \\ f \\ i \end{array} \right| \text{load} \quad \text{déplacement}$$

- Pour un champ de classe ou un champ d'instance, le code résultant a la forme suivante qui dépend des modificateurs du champ :

$$\left| \begin{array}{c} \text{getstatic} \\ \text{getfield} \end{array} \right| \text{nom complet du type/nom du champ} \quad \text{descripteur de type JNI}$$

Dans les deux cas, le nom désigne une *rvalue* (*Right Value*), c'est-à-dire que le nom ne peut apparaître que dans le membre de droite d'une affectation ou dans une expression libre. Il existe un autre cas dans lequel le nom désigne une *lvalue* (*Left Value*), c'est-à-dire que le nom apparaît dans le membre de gauche d'une affectation. La règle qui permet l'appariement avec un noeud de type *alValue* représente ce cas qui n'est possible que pour une affectation. Aucun traitement n'est effectué dans ce cas particulier.

Un nom suivi d'indices désigne une composante d'un tableau. Il y a deux règles pour la génération de code associé aux indices selon que le nom représente une *lvalue* ou une *rvalue*.

La partie conséquente de la règle qui permet l'appariement avec un noeud de type *aIndexrValue* fait tout d'abord un appel récursif à la procédure *CodeExpression* (section 7.12) avec comme paramètre le sous-arbre de la forme constituée du nom du tableau et possiblement d'autres indices (l'indice le plus à droite est exclu). Ceci permet un parcours en profondeur dans l'arbre syntaxique abstrait jusqu'au noeud qui représente le nom du tableau, c'est-à-dire un noeud de type *arValue*. L'instruction *aload*, *getstatic* ou *getfield* est alors écrite dans le fichier de code intermédiaire. À chaque étape du retour vers le haut dans le sous-arbre, la partie conséquente fait un appel à la procédure *CodeExpression* (section 7.12) qui génère le code pour évaluer une expression, c'est-à-dire la valeur d'un indice. Ce code est suivi d'une instruction *aaload*. Le même traitement est effectué pour le dernier indice, c'est-à-dire celui le plus à droite, mais l'instruction pour le chargement de la donnée dépend du type du tableau. Le code résultant a la forme suivante :

$$\left\{ \begin{array}{l} \text{aload} \quad \text{déplacement} \\ \left| \begin{array}{c} \text{getstatic} \\ \text{getfield} \end{array} \right| \text{nom complet du type/nom du champ} \quad \text{descripteur de type JNI} \\ \text{code pour l'évaluation de la valeur d'un indice et transtypage} \\ \text{aaload} \end{array} \right\}$$

code pour l'évaluation de la valeur de l'indice le plus à droite et transtypage

b	aload
f	
i	

La partie conséquente de la règle qui permet l'appariement avec un noeud de type *aIndexValue* fait sensiblement le même traitement, mais sans générer la dernière instruction de chargement, puisque celle-ci est remplacée par une instruction de stockage, ce cas n'étant possible que pour le membre de gauche d'une affectation. Le code résultant a la forme suivante :

aload	déplacement
getstatic	nom complet du type/nom du champ descripteur de type
getfield	
{ code pour l'évaluation de la valeur d'un indice et transtypage }	
aaload	
code pour l'évaluation de la valeur de l'indice le plus à droite et transtypage	

7.12.3 Règle pour un appel de méthode

La règle pour la génération de code associé à un appel de méthode dans une expression permet l'appariement avec un noeud de type *aFunctionCall*. Elle est semblable à la règle de la procédure *CodeStatements* pour un appel de méthode considéré comme un énoncé (section 7.10.8), sauf qu'aucune instruction n'est générée pour retirer le résultat de la pile de travail. Le code résultant a la forme suivante qui dépend des modificateurs de la méthode et de sa signature :

[aload_0]	; empiler la référence à l'objet courant pour l'appel non qualifié à une méthode d'instance
code pour l'évaluation des paramètres actuels	
invokestatic	nom de la méthode•signature JNI
invokevirtual	

7.12.4 Règle pour la création d'un tableau

La règle pour la génération de code associé à la création d'un tableau permet l'appariement avec un noeud de type *aArrayCreator*. La partie conséquente de cette règle fait un appel récursif à la procédure *CodeExpression* (section 7.12) à partir de l'unique opérande. Le code résultant a la forme suivante qui dépend du type de l'opérateur :

code pour l'évaluation de la taille de chaque dimension
multianewarray descripteur de type JNI nombre de dimensions

7.12.5 Règle pour un opérateur binaire

La règle pour la génération de code associé à un opérateur binaire permet l'appariement avec un noeud de type *aBinary*. La partie conséquente de cette règle fait des appels récursifs à la procédure *CodeExpression* (section 7.12) à partir de l'opérande de gauche et de l'opérande de droite. Le code résultant a la forme suivante qui dépend de l'opérateur et de son type :

code pour l'évaluation de l'opérande de gauche et le transtypage
code pour l'évaluation de l'opérande de droite et le transtypage

	add
f	div
i	mul
	rem
	sub

7.12.6 Règle pour un opérateur unaire

La règle pour la génération de code associé à un opérateur unaire permet l'appariement avec un noeud de type *aUnary*. La partie conséquente de cette règle fait un appel récursif à la procédure *CodeExpression* (section 7.12) à partir de l'unique opérande. Le code résultant a la forme suivante qui dépend du type de l'opérateur :

code pour l'évaluation de l'opérande et le transtypage

f	neg
i	

7.12.7 Règles pour les dimensions d'un tableau

Il y a trois règles pour la génération de code associé aux dimensions d'un tableau lors de sa création. Les règles qui permettent l'appariement avec un noeud de type *aUndefDimSpec* et de type *aDefDimSpec* effectuent un parcours en profondeur dans l'arbre syntaxique abstrait jusqu'au noeud qui représente la dimension la plus à gauche, c'est-à-dire un noeud de type *aBasicDimSpe* ; les dimensions les plus à droite pouvant être sans taille. À chaque étape du retour vers le haut dans le sous-arbre associé à cette forme, la partie conséquente fait un appel à la procédure *CodeExpression* (section 7.12) qui génère le code pour évaluer une expression, c'est-à-dire la taille d'une dimension d'un tableau à la condition que le noeud ne soit pas de type *aUndefDimSpec*. Le code résultant a la forme suivante :

code pour l'évaluation de la taille de la dimension la plus à gauche et transtypage
 { *code pour l'évaluation de la taille d'une dimension et transtypage* }

7.12.8 Règle pour un paramètre actuel

La règle pour la génération de code associé à un paramètre actuel permet l'appariement avec un noeud de type *aArgument*. La partie conséquente parcourt une liste de paramètres actuels à l'aide d'appels récursifs en utilisant l'attribut *Next* comme paramètre. Pour chaque paramètre actuel, le code résultant a la forme suivante selon le type de l'expression :

$\left(\begin{array}{l} L_i : \end{array} \right.$	<i>code pour l'évaluation de l'expression et le transtypage</i>
	iconst_0
	<i>code pour l'évaluation paresseuse de l'expression</i>
	pop
	iconst_1

7.13 Génération de code pour les expressions booléennes

La procédure de génération de code pour les expressions booléennes complexes, nommée *CodeLazyExpr*, permet la génération de code selon la technique d'évaluation paresseuse, car dans le cas d'une expression booléenne, l'évaluation de tous les termes n'est pas nécessaire pour déterminer sa valeur finale. La technique d'évaluation paresseuse des expressions booléennes est basée sur celle exposée dans le livre de Grune et al [12]. Cependant, il y a une petite erreur dans le traitement exposé dans cet ouvrage. Si les deux étiquettes fournies comme paramètres à la procédure sont invalides, c'est-à-dire que si *TrueLabel* = *NoLabel* et *FalseLabel* = *NoLabel*, alors la procédure ne génère aucun code de branchement ; ceci équivaut à négliger un terme. En plus de corriger cette erreur, la procédure a été adaptée à une machine à pile puisque la machine virtuelle *Java* n'a pas de registre d'état.

L'appel à la procédure *CodeLazyExpr* a comme paramètre, en plus d'un noeud de type *aExpression*, l'étiquette du cas vrai (L_t) et l'étiquette du cas faux (L_f). Pour être valide, au moins une de ces étiquettes doit être différente de la constante *NoLbl*.

7.13.1 Règle pour un terme booléen

La règle pour la génération de code associé à un terme booléen permet l'appariement avec un noeud de type *aPrimary*. Seuls les cas suivants sont traités : une constante booléenne, un nom qui désigne une *rvalue* booléenne et l'appel à une fonction booléenne. Le code résultant a la forme suivante qui dépend de la valeur des étiquettes (notez que le test a 0 comme deuxième opérande) :

$$\left. \begin{array}{ll} \text{code pour l'évaluation du terme booléen} \\ \left(\begin{array}{ll} \text{ifne } L_t \\ \text{goto } L_f \end{array} \right) & ; \text{ si } L_t \text{ et } L_f \text{ sont valides} \\ \text{ifne } L_t & ; \text{ si seule } L_t \text{ est valide} \\ \text{ifeq } L_f & ; \text{ si seule } L_f \text{ est valide} \end{array} \right|$$

7.13.2 Règles pour un opérateur binaire booléen

Il y a cinq règles pour la génération de code associé un opérateur binaire avec un résultat booléen. Toutes ces règles permettent l'appariement avec un noeud de type *aBinary*.

- La partie conséquente de celle pour les opérateurs $<$, $>$, \leq et \geq (avec nécessairement des opérandes non booléens et où « \leq » et « \geq » ayant été préalablement remplacés par « **not** $>$ » et « **not** $<$ ») et de celle pour les opérateurs d'égalité et d'inégalité (« $!=$ » ayant été préalablement remplacé par « **not** $==$ ») avec des opérandes non booléens génèrent le code qui a la forme suivante :

code pour l'évaluation de l'opérande de gauche et le transtypage

code pour l'évaluation de l'opérande de droite et le transtypage

$$\left| \begin{array}{ll} \left(\begin{array}{l} \text{code de branchement positif à } L_t \\ \text{goto } L_f \end{array} \right) & ; \text{ si } L_t \text{ et } L_f \text{ sont valides} \\ \text{code de branchement positif à } L_t & ; \text{ si seule } L_t \text{ est valide} \\ \text{code de branchement négatif à } L_f & ; \text{ si seule } L_f \text{ est valide} \end{array} \right|$$

- La partie conséquente de celle pour les opérateurs d'égalité et d'inégalité (« != » ayant été préalablement remplacé par « not == ») avec des opérandes booléens génère le code qui a la forme suivante :

code pour l'évaluation paresseuse de l'opérande de gauche($-, L_i$)
code pour l'évaluation paresseuse de l'opérande de droite(L_t, L_f)
 goto L_j
 $[L_i :]$
code pour l'évaluation paresseuse de l'opérande de droite(L_f, L_t)
 $[L_j :]$

- La partie conséquente de celle pour le ET logique génère le code qui a la forme suivante :

$[L_i :]$ $\left| \begin{array}{ll} \text{code pour l'évaluation paresseuse de l'opérande de gauche}(-, L_i) & \\ & ; \text{ si } L_f \text{ n'est pas valide} \\ \text{code pour l'évaluation paresseuse de l'opérande de gauche}(-, L_f) & \\ & ; \text{ si } L_f \text{ est valide} \end{array} \right|$
code pour l'évaluation paresseuse de l'opérande de droite(L_t, L_f)

- La partie conséquente de celle pour le OU logique génère le code qui a la forme suivante :

$[L_i :]$ $\left| \begin{array}{ll} \text{code pour l'évaluation paresseuse de l'opérande de gauche}(L_i, -) & \\ & ; \text{ si } L_t \text{ n'est pas valide} \\ \text{code pour l'évaluation paresseuse de l'opérande de gauche}(L_t, -) & \\ & ; \text{ si } L_t \text{ est valide} \end{array} \right|$
code pour l'évaluation paresseuse de l'opérande de droite(L_t, L_f)

7.13.3 Règle pour un opérateur unaire booléen

La règle pour la génération de code associé à un opérateur unaire booléen permet l'appariement avec un noeud de type *aUnary*. La clause conséquente de cette règle génère le code suivant :

code pour l'évaluation paresseuse de l'opérande de gauche(L_f, L_t)

7.14 Génération de code pour les branchements positifs

La procédure de génération de code pour les branchements positifs spécifiques à l'évaluation paresseuse des expressions booléennes complexes, nommée *CodeBinBranchOnCond*, n'est appelée que pour les cas de comparaison de deux opérandes avec les opérateurs $<$, $>$, et $==$. Le code résultant a la forme suivante qui dépend du type des opérandes et de l'opérateur :

$$\left| \begin{array}{l|l} \text{if_icmp} & \begin{array}{l} \text{lt} \\ \text{gt} \\ \text{eq} \end{array} \\ \left(\begin{array}{l|l} \text{if_cmpg} & \begin{array}{l} \text{lt} \\ \text{gt} \\ \text{eq} \end{array} \\ \text{if} & \begin{array}{l} \text{lt} \\ \text{gt} \\ \text{eq} \end{array} \end{array} \right) \\ \text{if_acmpeq} \end{array} \right| \begin{array}{l} ; \text{ si les opérandes sont de type entier} \\ ; \text{ si les opérandes sont de type point flottant} \\ ; \text{ si les opérandes sont de type référence} \end{array}$$

7.15 Génération de code pour les branchements négatifs

La procédure de génération de code pour les branchements négatifs spécifiques à l'évaluation paresseuse des expressions booléennes, nommée *CodeBinBranchOnNotCond*, n'est appelée que pour les cas de comparaison de deux opérandes avec les opérateurs $<$, $>$ et $==$. Toutefois, les règles de cette procédure considèrent les opérateurs opposés. Le code résultant a la forme suivante qui dépend du type des opérandes et de l'opérateur :

$$\left| \begin{array}{l|l} \text{if_icmp} & \begin{array}{l} \text{ge} \\ \text{le} \\ \text{ne} \end{array} \\ \left(\begin{array}{l|l} \text{if_cmpg} & \begin{array}{l} \text{ge} \\ \text{le} \\ \text{ne} \end{array} \\ \text{if} & \begin{array}{l} \text{ge} \\ \text{le} \\ \text{ne} \end{array} \end{array} \right) \\ \text{if_acmpne} \end{array} \right| \begin{array}{l} ; \text{ si les opérandes sont de type entier} \\ ; \text{ si les opérandes sont de type point flottant} \\ ; \text{ si les opérandes sont de type référence} \end{array}$$

7.16 Génération de code pour les variables et les paramètres

La procédure de génération de code pour les variables et les paramètres formels, nommée *CodeVarDirectives*, parcourt une liste d'énoncés ou une liste de paramètres formels. La règle pour la génération de la directive `.var` associée à une déclaration d'une variable locale permet l'appariement avec un noeud de type *aVariableDeclarator*. La règle pour la génération de la

directive `.var` associée à un paramètre formel permet l'appariement avec un noeud de type *aFormalParameter*. Dans les deux cas, la partie conséquente génère le code suivant :

`.var déplacement is 'nom' descripteur de type JNI from L_i to L_{i+1}`

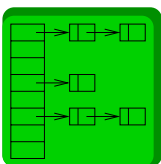
dans laquelle l'opérande *nom* désigne le nom d'une variable locale ou le nom d'un paramètre formel.

7.17 Exercice

1. Examinez le contenu des fichiers `GenCode.h` et `GenCode.c` générés par l'outil `puma`.

Chapitre 8

Le gestionnaire de la table des symboles



Le gestionnaire de la table des symboles permet le stockage de l'information symbolique d'une unité de compilation dans un ensemble de structures de données, appelé la table de symboles, et l'interrogation de cette information lors de l'analyse sémantique et de la génération de code. Le gestionnaire de la table des symboles est mis en oeuvre à partir d'une grammaire abstraite pour la définition des différentes structures de données et de fonctions *C* pour la recherche et la consultation de l'information contenue dans la table des symboles. L'outil **ast** permet de générer automatiquement le gestionnaire de la table des symboles à partir d'une spécification contenue dans le fichier `SymTab.cg`. L'annexe G contient une copie de ce fichier avec une brève description.

8.1 Définition des structures de données

Les structures de données utiles au stockage de l'information symbolique sont définies sous la forme d'une grammaire abstraite de façon à ce qu'elles soient utilisables par une spécification **puma** dans la génération de code. Il y a une structure de données pour les objets d'information suivants :

- un environnement,
- un *package*,
- un type (classe ou interface),
- un membre (champ ou méthode),
- une variable.

8.1.1 Définition d'un environnement

Un environnement est constitué d'un ensemble de *packages* et d'un ensemble de types. Toutefois, le langage *java--* ne permet qu'un seul *package* et qu'une seule classe comme type.

```
sEnv = CurrentPkg: sPackages IN  
      CurrentClass: sTypes IN .
```

8.1.2 Définition d'un *package*

Les descripteurs de *packages* sont chaînés pour former une liste en ordre inverse de leur déclaration. Dans la table de symboles, un *package* possède :

- un nom,
- une liste de types,
- une liste de sous-*packages*.

```
sPackages = <
  sEndOfPackages = .
  sPkgDesc       = Previous: sPackages IN
                  [Name: tIdent IN]
                  TypList: sTypes IN
                  PkgList: sPackages IN .
> .
```

Le nom d'un *package* doit être unique dans le contexte du *package* dans lequel il est défini. Le nom du seul package admis dans le langage *java--* (**NoIdent**) désigne le *package* courant (au sens *Java*).

8.1.3 Définition d'un type

Les descripteurs de types sont chaînés pour former une liste en ordre inverse de leur déclaration. Un type est soit une classe (représentée par un noeud de type *sClassDesc*), soit une interface (représentée par un noeud de type *sIntfDesc*).

Le nom d'un type doit être unique dans le contexte du *package* dans lequel il est défini. Seules les classes sont présentement supportées par le langage *java--*. Dans la table de symboles, un type possède :

- un nom,
- un ensemble d'indicateurs d'accès (**public** ou **aucun**),
- une liste de membres.

```
sTypes = <
  sEndOfTypes = .
  sTypDesc    = Previous: sTypes IN
                [Name: tIdent IN]
                [Modifiers: short IN]
                MemberList: sMembers IN <
    sClassDesc = .
    sIntfDesc  = .
  > .
> .
```

L'information relative à la spécialisation d'une classe (**refines**) et à l'implémentation d'une interface (**implements**) est absente dans cette structure de données. Elle devrait faire partie d'une extension du langage *java--*.

8.1.4 Définition d'un membre

Les descripteurs de membres sont habituellement chaînés pour former une liste en ordre inverse de leur déclaration. Un membre est soit un champ (représenté par un noeud de type *sFieldDesc*), soit une méthode (représentée par un noeud de type *sMethodDesc*). Dans la table de symboles, un membre possède :

- un nom,
- un ensemble d'indicateurs d'accès (**public** ou aucun) et d'un indicateur de membre de classe ou de membre d'instance (**static** ou aucun),
- un type.

Une méthode comporte aussi une liste de paramètres formels.

```
sMembers = <
  sEndOfMembers = .
  sMbrDesc      = Previous: sMembers IN
                  [Name: tIdent IN]
                  [Modifiers: short IN]
                  [Type: tTree IN] <
    sFieldDesc  = .
    sMethodDesc = [Formals: tTree IN] .
  > .
> .
```

Le nom d'un membre doit être unique dans le contexte du type dans lequel il est défini. Ceci interdit la surcharge (*Overloading*) d'une méthode. Notez que l'infrastructure pour supporter la surcharge d'une méthode est introduite dans la grammaire abstraite, c'est-à-dire l'information de signature d'une méthode), mais la surcharge d'une méthode n'est pas supportée par *java--*.

8.1.5 Définition d'une variable

Les descripteurs de variables, incluant les descripteurs des paramètres formels d'une méthode, sont chaînés pour former une liste en ordre inverse de leur déclaration. Dans la table de symboles, une variable possède :

- un nom,
- un type,
- un déplacement dans le bloc des variables locales.

```
sVariables = <
  sEndOfVariables = .
  sVarDesc        = Previous: sVariables IN
                    [Name: tIdent IN]
                    [Type: tTree IN]
                    [Offset: short IN] .
  > .
```

Cette structure est particulière au langage *Java* puisque ce langage ne fait pas usage de la notion de portée lexicale avec environnement. Ainsi, une liste de variables locales constitue une table de symboles de niveau d'une méthode seulement. Par exemple, le programme *java--*

```
{
  int i, j;
  (...)
  {
    int k;          // (1)
    {
      int l;        // (1)
    }
  }
  (...)
  int k, l;          // (2)
  (...)
}
```

produit la structure de la figure 8.1 suite à sa compilation. Dans une telle structure, le nom d'une variable doit être unique par rapport à l'ensemble des noms visibles au niveau de sa déclaration. Le nom d'une variable peut cependant camoufler (*shadow*) un nom d'attribut. Cependant, puisque les noms qualifiés ne sont pas supportés par *java--*, une variable qui couvre un nom d'attribut constitue une occlusion.

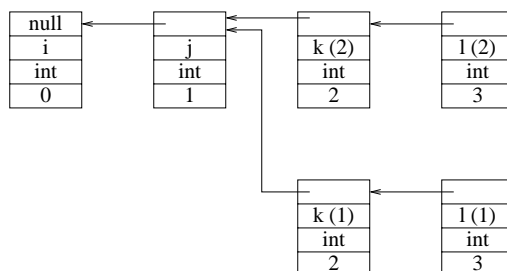


FIG. 8.1 – Liste des variables locales

8.2 Fonctions d'interrogation

Les fonctions d'interrogation accessibles à l'analyseur sémantique sont les suivantes :

- la fonction *IdentifySymbolRef* retourne un pointeur vers le descripteur d'une variable ou le descripteur d'un champ à partir d'un identificateur ;
- la fonction *IdentifyMethodRef* retourne un pointeur vers le descripteur d'une méthode à partir d'un identificateur et d'une liste de paramètres actuels ;
- la fonction *GetSymbolType* retourne un pointeur vers le sous-arbre de l'arbre syntaxique abstrait qui représente le type d'une variable, d'un champ ou d'une méthode à partir d'un descripteur de la table des symboles ;
- la fonction *GetMethodPrms* retourne un pointeur vers le sous-arbre de l'arbre syntaxique abstrait qui représente les paramètres formels d'une méthode à partir d'un descripteur de la table des symboles ;
- la fonction *IsDeclaredType* vérifie si un identificateur est un identificateur de type à l'aide d'un appel à la fonction *IdentifyType* qui parcourt une liste de types ;

- la fonction *IsDeclaredVariable* vérifie si un identificateur représente un identificateur de variable à l'aide d'un appel à la fonction *IdentifyVariable* qui parcourt une liste de variables ;
- la fonction *IsDeclaredField* vérifie si un identificateur est un identificateur de champ à l'aide d'un appel à la fonction *IdentifyField* qui parcourt une liste de champs ;
- la fonction *IsDeclaredMethod* vérifie si un identificateur est un identificateur de méthode à l'aide d'un appel à la fonction *IdentifyMethod* qui parcourt une liste de méthodes ;
- la fonction *IsObjectKind* vérifie si un descripteur est un descripteur de *package*, de type, de membre ou de variable.

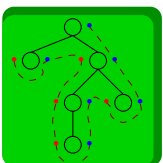
Il n'a pas de fonctions pour créer les descripteurs d'objets d'une liste chaînée, car elles sont générées automatiquement par l'outil **ast** à partir de la grammaire abstraite, tout comme celles pour créer les noeuds de l'arbre syntaxique abstrait.

8.3 Exercice

1. Examinez le contenu des fichiers `SymTab.h` et `SymTab.c` générés par l'outil **ast**.

Chapitre 9

L'analyseur sémantique



L'analyseur sémantique vérifie qu'un programme écrit en langage *java--* respecte les règles sémantiques de ce langage. Le principal traitement de l'analyseur sémantique consiste à évaluer les valeurs d'attributs qui sont stockées dans les noeuds de l'arbre syntaxique abstrait du programme. Ce traitement nécessite généralement plusieurs parcours de l'arbre. Les valeurs de certains attributs sont permanentes de façon à être utilisées par le générateur de code. L'outil **ag** permet de générer automatiquement un analyseur sémantique à partir de la grammaire attribuée de type OAG (*Ordered Attribute Grammar*) contenue dans le fichier `java--.cg`. L'annexe E contient une copie de ce fichier avec une description succincte.

9.1 Contenu du fichier `java--.cg`

Le fichier `java--.cg` comporte 14 modules. Deux modules sont relatifs à la construction de l'arbre syntaxique abstrait d'un programme par l'analyseur syntaxique. Les déclarations des types de noeuds de l'arbre et les définitions de leurs constructeurs sont automatiquement produites à partir d'une grammaire abstraite, grammaire calquée sur la grammaire hors contexte décrite dans le chapitre 2, mais dépouillée d'éléments qui ne sont pas essentiels pour l'analyse sémantique ou encore simplifiée par regroupement de structures similaires. Un module permet la déclaration des attributs dont les valeurs seront utilisées par le générateur de code. Un autre module contient du code *C* spécifique à l'analyseur sémantique, code qui ne peut être généré automatiquement par l'outil **ag**. Ces quatre modules sont de nature administrative.

Les autres modules constituent essentiellement une grammaire attribuée. Ils sont conceptuellement divisés en deux catégories : ceux spécifiques à l'analyse sémantique et ceux spécifiques à la machine virtuelle *Java*.

9.2 Modules administratifs

- Le module *GlobalDeclarations* contient les définitions de constantes utiles à la construction de l'arbre syntaxique abstrait, à l'évaluation des attributs et à la génération de

code. En particulier, les types primaires du langage *java--* sont représentés par des pointeurs constants sur des noeuds élémentaires.

- Le module *AbstractGrammar* contient la définition de la grammaire abstraite du langage *java--* sous la forme de types de noeuds avec leurs attributs.
- Le module *TargetCode*, contient le code *C* pour la partie initiale et la partie finale de l'analyseur sémantique ainsi qu'une fonction pour le traitement d'une erreur fatale.
- Le module *Output* contient les déclarations d'attributs de sortie. Ces attributs et tous ceux déclarés dans le module *AbstractGrammar* comme attributs d'entrée (PROPERTY INPUT) sont ceux dont les valeurs sont essentielles pour la génération de code.

9.3 Modules spécifiques à l'analyse sémantique

- Le module *TypeLevelDecls* recueille l'information symbolique au niveau de l'unité de compilation, c'est-à-dire ses types (classes et interfaces). Il effectue aussi la vérification de déclarations multiples d'un même symbole dans l'unité de compilation.
- Le module *MethodLevelDecls* recueille l'information symbolique relative aux méthodes d'un type, c'est-à-dire les paramètres et variables locales. Il assigne à chaque symbole son emplacement dans le bloc des variables locales d'une méthode. Il effectue aussi la vérification de déclarations multiples d'un même symbole dans chaque méthode.
- Le module *Environment* distribue l'information symbolique recueillie par les modules *TypeLevelDecls* et *MethodLevelDecls* dans les sous-arbres des méthodes de façon à effectuer des vérifications de cohérence par rapport à l'utilisation des symboles.
- Le module *Identification* identifie toutes les références à un symbole dans les sous-arbres des méthodes. Ceci correspond à l'interrogation de la table de symboles distribuée par le module *Environment*. Dans ce module, l'attribut *Type* associé au noeud de type *aArguments* est un attribut virtuel (VIRTUAL). Cette propriété indique à l'analyseur sémantique de ne pas calculer cet attribut et de considérer les dépendances introduites dans les règles sémantiques à l'aide du mot DEP. Par exemple, la valeur de l'attribut *Descriptor*, associé aux types de noeuds *aProcedureCall* et *aFunctionCall*, est obtenue par un appel à la fonction *IdentifyMethodRef*. Mais l'attribut *Descriptor* dépend du type des paramètres actuels. Cette façon de faire indique que le calcul du type des paramètres actuels doit être fait avant les vérifications relatives à un appel de méthode.
- Le module *ExprType* calcule le type de chaque expression présente dans les sous-arbres des méthodes. Il effectue aussi toutes les vérifications de types et il affiche les messages d'erreur le cas échéant.

9.4 Modules spécifiques à la machine virtuelle *Java*

- Le module *Coercions* effectue si nécessaire le transtypage de *int* vers *float* pour les d'affectations.
- Le module *OpType* calcule le type des instructions de la machine virtuelle *Java* à partir du type des opérandes de chaque opérateur binaire et de l'opérande de chaque opérateur unaire.

- Le module *Rtn* détermine pour chaque méthode s'il y a un énoncé de retour (**return**) dans l'une des branches de son flux de contrôle. Dans le cas contraire, il affiche un message d'erreur.
- Le module *FrameSize* calcule la taille du bloc des variables locales (incluant les paramètres formels) pour chaque méthode. Ce module calcule aussi le déplacement relatif de chaque variable dans un bloc.
- Le module *WSSize* calcule la taille maximum (profondeur) de la pile de travail pour chaque méthode.

9.5 Déclaration des attributs

Les attributs d'un module sont déclarés dans la clause **DECLARE**. Un attribut peut être :

- un attribut d'entrée (**IN**) ;
- un attribut de sortie (**OUT**) ;
- un attribut virtuel (**VIRTUAL**) ;
- un attribut hérité (**INH**) ;
- un attribut synthétisé (**SYN**) ;
- un attribut correspondant à deux attributs (**THREAD**).

Dans le dernier cas, la déclaration d'un attribut de type **THREAD** introduit deux attributs. Par exemple, dans le module *MethodLevelDecls*, la déclaration de l'attribut *Offset*, associé aux noeuds de type *aFormalParameters*, introduit l'attribut hérité *OffsetIn* et l'attribut synthétisé *OffsetOut*.

Il est possible de forcer l'ordre d'évaluation des attributs en utilisant la clause **BEFORE**, comme c'est le cas pour l'attribut *RtnOut* par rapport à l'attribut *Env* dans le module *Rtn*, cet attribut étant associé aux noeuds de type *aStatements*. Ceci permet d'obtenir une grammaire attribuée de type OAG.

9.6 Calcul des attributs

Le calcul des attributs est exprimé à l'aide de règles sémantiques qui sont insérées dans des fonctions de parcours de l'arbre syntaxique abstrait (voir la section 5.7 du livre de Aho et al [1]). Ces règles ne sont pas du code *C*, sauf si le calcul d'un attribut est délimité par des accolades. Par exemple, les règles suivantes dans le module *MethodLevelDecls*, associées aux noeuds de type *aFormalParameters*, contiennent du code *C*, l'appel de la fonction *AbstractCError* définie dans le module *TargetCode* :

```
aFormalParameters = {
  OffsetOut  := { AbstractCError( "FrameLayout", "aFormalParameters" ); };

  VarListOut := { AbstractCError( "SymbolTable", "aFormalParameters" ); };
} .
```

Par contre, le code

```

aFormalParameter = {
  Next:OffsetIn := OffsetIn + TypeSize( aType );
  Next:Offset    := OffsetIn + TypeSize( aType );
  OffsetOut := Next:OffsetOut;

  Next:VarListIn := msVarDesc( VarListIn, Name, aType, OffsetIn );
  VarListOut := Next:VarListOut;

  CHECK ( ! IsDeclaredVariable( Name, VarListIn ) )
  => Message( "Parameter: duplicate symbol declaration.", xxError, Pos);
} .

```

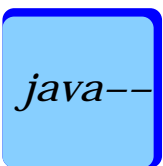
contient cinq règles de calcul d'un attribut (distinguable par le symbole « := ») exprimées dans la notation de l'outil `ag`. Il contient aussi une règle de vérification (distinguable par le verbe `CHECK`) avec une condition qui ne doit pas être satisfaite pour déclencher la partie conséquente.

9.7 Exercice

1. Examinez le contenu des fichiers `Semantics.h` et `Semantics.c` générés par l'outil `ag`.

Annexe A

Le programme principal



Le fichier `java--.c` contient le programme principal du compilateur *java--*. Ce programme appelle des fonctions qui correspondent aux trois phases du compilateur, soit l'analyse syntaxique et la construction de l'arbre syntaxique abstrait, l'analyse sémantique et la génération de code. Ces trois fonctions ont sensiblement la même structure :

- un prologue qui initialise un ou des modules à l'aide d'appels à des fonctions *BeginX()* ;
- un traitement principal qui est essentiellement un appel de fonction au module correspondant ;
- un épilogue qui termine le traitement d'un ou plusieurs modules à l'aide d'appels à des fonctions *CloseX()* et qui affiche quelques statistiques.

La fonction *GenerateAST* appelle l'analyseur syntaxique (*Parser()*) qui construit l'arbre syntaxique abstrait à la volée pendant la vérification des règles syntaxiques. L'analyseur syntaxique emmagasine le pointeur vers la racine de l'arbre dans la variable globale *TreeRoot*.

La fonction *EvaluateAST* appelle l'analyseur sémantique (*Semantics(TreeRoot)*) qui évalue les attributs par rapport à la grammaire attribuée en parcourant plusieurs fois l'arbre syntaxique abstrait à partir du pointeur vers sa racine (*TreeRoot*).

La fonction *GenerateCode* appelle le générateur de code (*GenCode(TreeRoot)*) qui produit un fichier « .j » prêt à être traduit par l'assembleur *Jasmin* dans du code compatible à la machine virtuelle *Java*. Le générateur de code applique des schémas de génération de code suite à des appariements sur l'arbre syntaxique abstrait (*TreeRoot*).

Si des erreurs sont détectées suite à l'exécution d'un module, le programme principal termine abruptement sans effectuer les appels aux modules correspondant aux phases en aval. Toutefois, il affiche les messages d'erreurs (syntaxiques ou sémantiques) à l'aide d'un appel à la fonction *WriteMessages(stderr)*.

En plus de la logique qui lie les différentes phases du compilateur, le programme principal effectue l'analyse de la commande d'appel du compilateur à l'aide des variables *argc* et *argv*. Cette commande doit avoir la forme suivante :

```
java-- {-a | -p | -e | -g} chemin/nom_de_fichier.mjv
```

La fonction *ProcessOptions* contient le code d'analyse de cette commande. En plus de positionner différents indicateurs selon les options contenues dans la commande, elle retourne le

numéro de l'argument qui indique le chemin d'accès ainsi que le nom du fichier qui contient le programme à compiler. Ce numéro est passé en paramètre à la fonction *PrepareSourceFile* qui effectue des vérifications et des manipulations sur le nom du fichier source.

Le traitement particulier de la fonction *PrepareSourceFile* est l'appel de la fonction *BeginFile* de l'analyseur lexical avec le paramètre *argv[argno]*, qui est un pointeur vers un argument de la commande, celui du chemin d'accès ainsi que du nom du fichier qui contient le programme *java--* à compiler. L'appel à cette fonction est fait après

- l'extraction du nom du fichier ;
- la vérification que le nom du fichier a le suffixe « *.mjv* » ;
- la vérification que le fichier existe.

La fonction *PrepareSourceFile* retourne le nom du fichier au programme principal afin de remplacer le suffixe « *.mjv* » par « *.j* » et ainsi former le nom du fichier de sortie, c'est-à-dire celui qui contient le programme en langage d'assemblage *JasminXT*. Ce fichier est créé dans le répertoire courant et non pas dans le répertoire indiqué dans la commande *java--*.

Notez que l'appel à la fonction *BeginFile* doit toujours être fait avant l'initialisation de l'analyseur syntaxique (*BeginParser*) qui appelle la fonction *BeginScanner* de l'analyseur lexical.


```

/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.

                                */
                                /* java--.c                                */

/* Description: This program is mostly a shell that holds all the modules
together; aside from arguments analysis, it initializes and
invokes the modules one after another.
Author: Daniel Cote.
Date: February 2006.

                                */

#include <stdio.h>           /* ISO C Standard: 4.9 input/output      */
#include <errno.h>           /* ISO C Standard: 4.1.3 errors         */
#include <string.h>          /* ISO C Standard: 4.11 string handling  */

#include "ratc.h"            /* cocktail: Boolean type               */
#include "rGetopt.h"         /* cocktail: Command line options analyzer */
#include "rMemory.h"         /* cocktail: Dynamic storage            */
#include "Position.h"        /* cocktail: Source positions handler    */
#include "Errors.h"          /* cocktail: Error handler              */

#include "Parser.h"          /* Generated by cocktail: Parser        */
#include "Tree.h"            /* Generated by cocktail: AST node constructors */
#include "Semantics.h"       /* Generated by cocktail: Attribute evaluator */

#include "TypeSys.h"         /* Generated by cocktail: Type system    */
#include "SymTab.h"          /* Generated by cocktail: Symbol table handler */
#include "GenCode.h"         /* Generated by cocktail: Code generator */

#define DO_NOT_DUMP_AST 0    /* Do not dump ASTs                    */
#define DUMP_AST 1          /* Dump both, the pre and post evaluation ASTs */

#define PHASE_PARSING 0      /* Compile up to parsing only          */
#define PHASE_EVALUATION 1   /* Compile up to evaluation only        */
#define PHASE_CODEGEN 2     /* Compile up to code generation        */

static short optAST = DO_NOT_DUMP_AST; /* Default option: do not dump */
static short optPhase = PHASE_CODEGEN; /* Default option: all phases */

```

```

void DumpPreEvalAST( void )
{
/* Description: Write the abstract syntax tree created by the parser.
Author: Daniel Cote.
Date: February 2006.
Input: None.
Output: None.
                                */

FILE *fp = NULL;

fp = fopen( "AST.psr", "w" );

if ( fp == NULL )
    fprintf( stderr, "*** File creation error,"
             " cannot dump the pre eval AST...\n" );
else
{
    fprintf( stderr, " Writing the AST created by the parser"
             " in file \"AST.psr\"...\n" );
    WriteTree( fp, TreeRoot );
    fclose( fp );
    fp = NULL;
}
}

/* ----- */

void DumpPostEvalAST( void )
{
/* Description: Write the abstract syntax tree decorated by the evaluator.
Author: Daniel Cote.
Date: February 2006.
Input: None.
Output: None.
                                */

FILE *fp = NULL;

fp = fopen( "AST.evl", "w" );

if ( fp == NULL )
    fprintf( stderr, "*** File creation error,"
             " cannot dump the post eval AST...\n" );
else
{
    fprintf( stderr, " Writing the AST decorated by the evaluator"
             " in file \"AST.evl\"...\n" );
    WriteTree( fp, TreeRoot );
    fclose( fp );
    fp = NULL;
}
}

```

```

int GenerateAST( void )
{
    /* Description: Invoke the parser (including construction of the
       abstract syntax tree).
       Author:      Daniel Cote.
       Date:        February 2006.
       Input:       None.
       Output:      0 if success; 1 if errors. */

    fprintf( stderr, " Parsing and constructing the AST...\n" );

    BeginParser();                /* Initialize the parser
                                   (section BEGIN of java--.prs) */
    Parser();

    CloseParser();                /* Finalize the parser
                                   (section CLOSE of java--.prs) */

    fprintf( stderr, " Memory%8d", MemoryUsed );
    fprintf( stderr, " Tree%8d", Tree_HeapUsed );
    fprintf( stderr, " Symb%8d", SymTab_HeapUsed );
    fprintf( stderr, "\n" );    fflush( stderr );

    return 0;
}

/* ----- */

int EvaluateAST( void )
{
    /* Description: Invoke the attribute evaluator.
       Author:      Daniel Cote.
       Date:        February 2006.
       Input:       None.
       Output:      0 if success; 1 if errors. */

    fprintf( stderr, " Evaluating the attributes...\n" );

    BeginSemantics();             /* Initialize the attribute evaluator
                                   (section BEGIN of java--.cg) */
    BeginSymTab();               /* Initialize the symbol table handler
                                   (section BEGIN of SymTab.cg) */
    BeginTypeSys();             /* Initialize the type system
                                   (section BEGIN of TypeSys.pum) */
    Semantics( TreeRoot );

    CloseTypeSys();              /* Finalize the type system
                                   (section CLOSE of TypeSys.pum) */
    CloseSymTab();              /* Finalize the symbol table handler
                                   (section CLOSE of SymTab.cg) */
    CloseSemantics();           /* Finalize the attribute evaluator
                                   (section CLOSE of java--.cg) */

    fprintf( stderr, " Memory%8d", MemoryUsed );
    fprintf( stderr, " Tree%8d", Tree_HeapUsed );
    fprintf( stderr, " Symb%8d", SymTab_HeapUsed );
    fprintf( stderr, "\n" );    fflush( stderr );

    return 0;
}

```

```

int GenerateCode( char *srcFileName )
{
    /* Description: Invoke the code generator.
       Author:      Daniel Cote.
       Date:        February 2006.
       Input:       srcFileName: pointer to the input file name.
       Output:      0 if success; 1 if errors. */

    fprintf( stderr, " Generating code for file %s...\n", srcFileName );

    BeginGenCode();              /* Initialize the code generator
                                   (section BEGIN of GenCode.pum) */
    GenCode( TreeRoot );

    CloseGenCode();             /* Finalize the code generator
                                   (section CLOSE of GenCode.pum) */
    fflush( stderr );

    return 0;
}

```

```

int ProcessOptions( int argc, char *const *argv )
{
    /* Description: Analyze the command line options.
    Author:      Daniel Cote.
    Date:       February 2006.
    Input:      argc: number of arguments.
               argv: the arguments.
    Output:     The index in argv of the first argument that is not an
               option.
    */

    int result = 1;
    char option;

    rOptind = 0;
    option = rGetopt( argc, argv, "apeg" );
    while ( ( option != EOF ) && ( rOptmsg == NULL ) )
    {
        switch ( option )
        {
            case 'a' :
                optAST = DUMP_AST;
                break;

            case 'p' :
                if ( optPhase > PHASE_PARSING )
                    optPhase = PHASE_PARSING;
                break;

            case 'e' :
                if ( optPhase > PHASE_EVALUATION )
                    optPhase = PHASE_EVALUATION;
                break;

            case 'g' :
                /* Default option
                */
                break;
        }

        option = rGetopt( argc, argv, "apeg" );
    }

    if ( rOptmsg == NULL )
        result = rOptind;
    else
    {
        fprintf( stderr, "%s \"%c\"\\n", rOptmsg, rOptopt );

        fprintf( stderr, "  usage : java-- [ options... ] <file_name>.mjb\\n" );
        fprintf( stderr, "  options :\\n" );
        fprintf( stderr, "    -a : Dump both, the pre and"
        " post evaluation ASTs.\\n" );
        fprintf( stderr, "    -p : Compile up to parsing only"
        " (overrides -e and -g).\\n" );
        fprintf( stderr, "    -e : Compile up to evaluation only"
        " (overrides -g).\\n" );
        fprintf( stderr, "    -g : (default) Complete compile up"
        " to code generation.\\n" );
        fprintf( stderr, "\\n" );

        exit( 1 );
    }
}

```

```

if ( result >= argc )
{
    fprintf( stderr, "(%s) Missing source file name\\n", argv[0] );
    exit( 1 );
}

return result;
}

/* ----- */

char *PrepareSourceFile( int argc, char *argv[], int argno )
{
    /* Description: Check the suffix of the file name and if the file exists.
    Author:      Daniel Cote.
    Date:       February 2006.
    Input:      argc: number of arguments.
               argv: the arguments.
               argno: the index in argv of the first argument that is not an
               option.
    Output:     A pointer to the input file name (without the path).
    */

    char *ptr;
    char *ptr_file_name = argv[argno];

    FILE *inFile;

    for ( ptr = ptr_file_name; *ptr != '\\0'; ptr++ )
        if ( *ptr == '/' )
            ptr_file_name = ptr + 1;

    if ( strlen(ptr_file_name) < 4 ||
        strcmp( ".mjb", ptr_file_name + strlen( ptr_file_name ) - 4, 4 ) != 0 )
    {
        fprintf( stderr, "(%s) A \".mjb\" file is required\\n", argv[0] );
        exit( 1 );
    };

    inFile = fopen( argv[argno], "r" );

    if ( inFile == NULL )
    {
        fprintf( stderr, "(%s) Cannot open file named \"%s\"\\n",
            argv[0], argv[argno] );
        exit( 1 );
    }
    else
        fclose( inFile );

    BeginFile( argv[argno] ); /* Call the scanner to redirect input from file */

    return ptr_file_name;
}

```

```

int main(int argc, char *argv[])
{
    /* Description: Main program.
       Author:      Daniel Cote.
       Date:        February 2006.
       Input:       argc: number of arguments.
                   argv: the arguments.
       Output:      0 if success; 1 if errors.          */

    int r;

    r = ProcessOptions( argc, argv );
    sourceFileName = PrepareSourceFile( argc, argv, r );

    switch ( optPhase )
    {
        case PHASE_PARSING :
            fprintf( stderr, "Compiling: \"%s\" up to parsing only...\n",
                    sourceFileName );
            break;

        case PHASE_EVALUATION :
            fprintf( stderr, "Compiling: \"%s\" up to evaluation only...\n",
                    sourceFileName );
            break;

        default:
            fprintf( stderr, "Compiling: \"%s\"\n", sourceFileName );
    }
}

```

```

StoreMessages( rtrue );          /* Store error messages          */

r = GenerateAST();               /* Phase 1: parsing and construction
                                of the abstract syntax tree */

if ( r == 0 && GetCount( xxError ) == 0 )
{
    if ( optAST )
        DumpPreEvalAST();

    if ( optPhase > PHASE_PARSING )
    {
        r = EvaluateAST();        /* Phase 2: attribute evaluation    */

        if ( r == 0 && GetCount( xxError ) == 0 )
        {
            if ( optAST )
                DumpPostEvalAST();

            if ( optPhase > PHASE_EVALUATION )
                r = GenerateCode( sourceFileName ); /* Phase 3: code generation */
        }
        else
        {
            fprintf( stderr, "*** Semantic errors:\n" );
            /* Write error messages in right order */
            WriteMessages( stderr );
            fprintf( stderr, "\n" );    fflush( stderr );    r = 1;
        }
    }
}
else
{
    fprintf( stderr, "*** Syntactic errors:\n" );
    WriteMessages( stderr );          /* Write error messages in right order */
    fprintf( stderr, "\n" );    fflush( stderr );    r = 1;
}

if ( r == 0 )
    fprintf( stderr, "Compilation completed: \"%s\"\n", sourceFileName );

fflush( stderr );

return r;
}

```

Annexe B

Les fonctions de calcul des attributs intrinsèques

Attributs

173

'\177'

"ABC\n"

Le fichier `JavaFPAttributes.h` contient la déclaration de tous les attributs intrinsèques associés à une constante en point flottant non signée. Le fichier `JavaFPAttributes.c` contient leur calcul. Voici la liste des attributs associés à une constante en point flottant :

- *lLexeme* — une référence vers le lexème en entrée correspondant à une constante en point flottant ;
- *lValue* — la valeur binaire de la constante en point flottant sur 32 ou 64 bits ;
- *lGroup* — le type de la constante en point flottant (32 ou 64 bits) ;
- *lOverflow* — un indicateur de dépassement de capacité positive ;
- *lUnderflow* — un indicateur de dépassement de capacité négative.

Le calcul des attributs d'une constante en point flottant n'est pas complet. Seul un traitement minimal est effectué à l'aide de la fonction `sscanf`.

Le fichier `JavaIntAttributes.h` contient la déclaration de tous les attributs intrinsèques associés à une constante entière non signée. Le fichier `JavaIntAttributes.c` contient leur calcul. Voici la liste des attributs associés à une constante entière :

- *lLexeme* — une référence vers le lexème en entrée correspondant à une constante entière ;
- *lValue* — la valeur binaire de la constante entière sur 32 ou 64 bits ;
- *lGroup* — le type de la constante entière (32 ou 64 bits) ;
- *lMinValue* — un indicateur pour une constante entière ayant la plus petite valeur possible en valeur absolue ;
- *lOverflow* — un indicateur de débordement pour une constante entière ne pouvant tenir sur 32 ou 64 bits.

Le fichier `JavaCharAttributes.h` contient la déclaration de tous les attributs intrinsèques associés à un caractère. Le fichier `JavaCharAttributes.c` contient leur calcul. Voici la liste des attributs associés à un caractère :

- *lLexeme* — une référence vers le lexème en entrée correspondant à un caractère ;
- *lValue* — le code UTF-16 sur 16 bits du caractère.

Seul le sous-ensemble de UTF-16 qui correspond aux caractères ASCII est présentement considéré.

Le fichier `JavaStringAttributes.h` contient la déclaration de tous les attributs intrinsèques associés à une chaîne de caractères. Le fichier `JavaStringAttributes.c` contient leur calcul. Voici la liste des attributs associés à une chaîne de caractères :

- *lLexeme* — une référence vers le lexème en entrée correspondant à une chaîne de caractères ;
- *lLength* — la longueur de la chaîne de caractères ;
- *lString* — un pointeur sur la chaîne de caractères.

Seul le sous-ensemble de UTF-16 qui correspond aux caractères ASCII est présentement considéré.

À partir de la référence vers un lexème (*lLexeme*), il est possible d'obtenir le lexème à l'aide d'un appel à la fonction *StGetCStr* définie dans le fichier `StringM.h` de la boîte à outils *cocktail*.

Le fichier `JavaEscapeSeq.h` contient la déclaration des données associées à une séquence d'échappement. Le fichier `JavaEscapeSeq.c` contient leur calcul. Ces fichiers sont utiles pour le calcul des attributs d'un caractère ou d'une chaîne de caractères. Voici la liste des données associées à une séquence d'échappement :

- *value* — le code UTF-16 sur 16 bits du caractère correspondant à la séquence d'échappement ;
- *next* — le déplacement par rapport au début de la séquence d'échappement pour atteindre le dernier caractère de la séquence d'échappement.

```
/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.
*/

#ifdef JavaFPAttributes
#define JavaFPAttributes

/* JavaFPAttributes.h */

/* Description: Definition of attributes for floating point literals.
Author:      Daniel Côté.
Date:       June 2006.
Remark:     THIS IS A TEMPORARY SOLUTION
*/

#include "StringM.h" /* cocktail: String memory */

typedef float tFloatType;
typedef double tDoubleType;
typedef enum { Float, Double } tFPCategory;

typedef union { tFloatType B32; tDoubleType B64; } tFPRRepresentation;

typedef struct { tStringRef lLexeme;
                tFPRRepresentation lValue;
                tFPCategory lGroup;
                short lOverflow;
                short lUnderflow; } tFPLiteral;

extern const tFPLiteral NoFPLiteral;

extern void SetFPAttributesHex(char [], unsigned int);
extern void SetFPAttributesDecimal(char [], unsigned int);

#endif
```

```

/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.
*/

/* JavaFPAttributes.c */

#include <stdio.h> /* ISO C Standard: 4.9 input/output */
#include <stdlib.h> /* ISO C Standard: strtoll */
#include <string.h> /* ISO C Standard: 4.11 string handling */
#include <values.h> /* Sun for the MAXFLOAT constant */

#include "Scanner.h" /* Generated by cocktail: variable Attribute */

/* Description: Implementation of functions for floating point literals.
Author: Daniel Côté.
Date: June 2006.
Remark: This is rather a bit of a temporary solution to process
floating point literals in an homogeneous way akin to what is
done for integer literals.

1) Lexical validations are not done really, rather "sscanf"
and its result is used in order to obtain a value from the
input lexeme.

2) This is motivated by the following facts:
a) The assembler jasmin requires a specific format for
floating point literals. Therefore a "value" of the
appropriate type must be supplied to enable the code
generator to produce a literal in the correct symbolic
assembly code format.

b) In order not to wreck havoc in the language's typing
system (which does not allow certain types of variable but
does allow the corresponding literals) the basic types
MUST be implemented and overflow checking of literals
MUST be implemented as part of lexical or syntax analysis.

Attempt to assign a "double" literal to a "float" variable
(x = 0.0D) is allowed on input. So, this situation must
be trapped in semantic analysis, but for that to occur
without patching too much, the literal must be typed
correctly (i.e., be of aDoubleType), ergo the type MUST
be implemented even though only literal constants of this
type can go through syntax analysis.
*/

```

```

const tFPLiteral NoFPLiteral = { 0, 0.0, Float, 0, 0 };

```



```

void SetFPAttributesHex(char lexeme[], unsigned int l)
{
    /* Description: Set the attributes of a hexadecimal floating point literal.
    Author:      Daniel Côté.
    Date:       June 2006.
    Input:      lexeme: a C string of characters representing the lexeme.
                l: length of the lexeme.
    Output:     None.
    Remark:     As long as exhaustive floating point constant literal
                validations are not implemented, hexadecimal floating point
                format cannot be implemented. */

    /* Leave it as a double, not supported in the language */
    Attribute.flPointLiteral.gFP.lValue.B64 = 0.0L;
    Attribute.flPointLiteral.gFP.lGroup = Double;
    Attribute.flPointLiteral.gFP.lOverflow = 0;
    Attribute.flPointLiteral.gFP.lUnderflow = 0;
}

```

```

void SetFPAttributesDecimal(char lexeme[], unsigned int l)
{
    /* Description: Compute the attributes of a decimal floating point literal.
    Author:      Daniel Côté.
    Date:       June 2006.
    Input:      lexeme: a C string of characters representing the lexeme.
                l: length of the lexeme.
    Output:     None. */

    char number[256];

    if (lexeme[l-1] == 'f' || lexeme[l-1] == 'F')
    {
        Attribute.flPointLiteral.gFP.lValue.B32 = 0.0;
        Attribute.flPointLiteral.gFP.lGroup = Float;
    }
    else
    {
        Attribute.flPointLiteral.gFP.lValue.B64 = 0.0L;
        Attribute.flPointLiteral.gFP.lGroup = Double;
    }

    Attribute.flPointLiteral.gFP.lOverflow = 0;
    Attribute.flPointLiteral.gFP.lUnderflow = 0;

    if (lexeme[l-1] == 'f' || lexeme[l-1] == 'F')
    {
        strncpy(number, lexeme, l-1);
        number[l-1] = 0;

        if (sscanf(number, "%f", &Attribute.flPointLiteral.gFP.lValue.B32) == 1)
        {
            if (Attribute.flPointLiteral.gFP.lValue.B32 > MAXFLOAT)
                Attribute.flPointLiteral.gFP.lOverflow = 1;
        }
        else
            Attribute.flPointLiteral.gFP.lOverflow = 1; /* unsound */
    }
    else
    {
        strncpy(number, lexeme, l);
        number[l] = '0';

        if (sscanf(number, "%lf", &Attribute.flPointLiteral.gFP.lValue.B64) == 1)
        {
            if (Attribute.flPointLiteral.gFP.lValue.B64 > MAXDOUBLE)
                Attribute.flPointLiteral.gFP.lOverflow = 1;
        }
        else
            Attribute.flPointLiteral.gFP.lOverflow = 1; /* unsound */
    }
}

```

```
/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.
*/

#ifdef JavaIntAttributes
#define JavaIntAttributes

/* JavaIntAttributes.h */

/* Description: Definition of attributes for integer literals.
Author:      Richard St-Denis.
Date:       February 2006.
*/

#include "StringM.h" /* cocktail: String memory */

typedef int tIntType;
typedef long long int tLongType;
typedef enum { Int, Long } tCategory;

typedef union { tIntType B32; tLongType B64; } tRepresentation;

typedef struct { tStringRef lLexeme;
                tRepresentation lValue;
                tCategory lGroup;
                short lMinValue;
                short lOverflow; } tIntegerLiteral;

extern const tIntegerLiteral NoIntegerLiteral;

extern void SetAttributesHex(char [], unsigned int);
extern void SetAttributesOctal(char [], unsigned int);
extern void SetAttributesDecimal(char [], unsigned int);

#endif
```

```

/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.

                                */
                                /* JavaIntAttributes.c                                */
#include <stdlib.h>                /* ISO C Standard: strtoll                */
#include "Scanner.h"              /* Generated by cocktail: variable Attribute */

/* Description: Implementation of functions for integer literals.
Author:      Richard St-Denis.
Date:       February 2006.
                                */

```

```

const tIntegerLiteral NoIntegerLiteral = { 0, 0, Int, 0, 0};

/* ----- */
char Complement1Hex(char hexDigit)
{
/* Description: Compute the complement of an hexadecimal digit.
Author:      Richard St-Denis.
Date:       February 2006.
Input:      hexDigit: a character representing a hexadecimal digit.
Output:     A character representing the complement of the digit.
*/

char C[] = { 'F', 'E', 'D', 'C', 'B', 'A', '9', '8', '7', '6', '5', '4', '3', '2', '1', '0' };

if (hexDigit >= '0' && hexDigit <= '9')
    return C[hexDigit - 0x30];
else
    if (hexDigit >= 'A' && hexDigit <= 'F')
        return C[hexDigit - 0x41 + 10];
    else
        return C[hexDigit - 0x61 + 10];
}

/* ----- */
char Complement1Octal(char octalDigit)
{
/* Description: Compute the complement of an octal digit.
Author:      Richard St-Denis.
Date:       February 2006.
Input:      octalDigit: a character representing an octal digit.
Output:     A character representing the complement of the digit.
*/

char C[] = { '7', '6', '5', '4', '3', '2', '1', '0' };

return C[octalDigit - 0x30];
}

```

```

void SetAttributesHex(char lexeme[], unsigned int l)
{
    /* Description: Compute the attributes of an hexadecimal integer literal.
    Author:      Richard St-Denis.
    Date:       February 2006.
    Input:      lexeme: a C string of characters representing the lexeme.
                l: length of the lexeme.
    Output:     None.
    */

    unsigned int i;
    char number[256];
    char *d;

    Attribute.integerLiteral.gInt.lValue.B64 = 0;
    Attribute.integerLiteral.gInt.lGroup = Int;
    Attribute.integerLiteral.gInt.lMinValue = 0;
    Attribute.integerLiteral.gInt.lOverflow = 0;

    if (lexeme[l-1] == 'l' || lexeme[l-1] == 'L')
    {
        Attribute.integerLiteral.gInt.lGroup = Long;
        if (l > 19)
            Attribute.integerLiteral.gInt.lOverflow = 1;
        else
        {
            strncpy(number, lexeme+2, l-3);    number[l-3] = 0;
            if (l == 19 && strcmp(number, "8000000000000000") == 0)
            {
                Attribute.integerLiteral.gInt.lMinValue = 1;
                Attribute.integerLiteral.gInt.lValue.B64 = -9223372036854775808L;
            }
            else
            if (l == 19 && lexeme[2] >= '8')
            {
                for (i = 2; i < 18; i++)
                    lexeme[i] = Complement1Hex(lexeme[i]);
                Attribute.integerLiteral.gInt.lValue.B64 =
                    -(strtoll(lexeme, &d, 16) + 1);
            }
            else
                Attribute.integerLiteral.gInt.lValue.B64 =
                    strtoll(lexeme, &d, 16);
        }
    }
}

```

```

else
{
    if (l > 10)
        Attribute.integerLiteral.gInt.lOverflow = 1;
    else
    {
        strncpy(number, lexeme+2, l-2);    number[l-2] = 0;
        if (l == 10 && strcmp(number, "80000000") == 0)
        {
            Attribute.integerLiteral.gInt.lMinValue = 1;
            Attribute.integerLiteral.gInt.lValue.B32 = -2147483648;
        }
        else
        if (l == 10 && lexeme[2] >= '8')
        {
            for (i = 2; i < 10; i++)
                lexeme[i] = Complement1Hex(lexeme[i]);
            Attribute.integerLiteral.gInt.lValue.B32 =
                -(strtol(lexeme, &d, 16) + 1);
        }
        else
            Attribute.integerLiteral.gInt.lValue.B32 =
                strtol(lexeme, &d, 16);
    }
}
}
}

```

```

void SetAttributesOctal(char lexeme[], unsigned int l)
{
    /* Description: Compute the attributes of an octal integer literal.
       Author:      Richard St-Denis.
       Date:        February 2006.
       Input:       lexeme: a C string of characters representing the lexeme.
                   l: length of the lexeme.
       Output:      None.
    */

    unsigned int i;
    char number[256];
    char *d;

    Attribute.integerLiteral.gInt.lValue.B64 = 0;
    Attribute.integerLiteral.gInt.lGroup = Int;
    Attribute.integerLiteral.gInt.lMinValue = 0;
    Attribute.integerLiteral.gInt.lOverflow = 0;

    if (lexeme[l-1] == '1' || lexeme[l-1] == 'L')
    {
        Attribute.integerLiteral.gInt.lGroup = Long;
        if (l > 24 || (l == 24 && lexeme[l] != '1'))
            Attribute.integerLiteral.gInt.lOverflow = 1;
        else
        {
            strncpy(number, lexeme+1, l-2); number[l-2] = 0;
            if (l == 24 && strcmp(number, "1000000000000000000000") == 0)
            {
                Attribute.integerLiteral.gInt.lMinValue = 1;
                Attribute.integerLiteral.gInt.lValue.B64 = -9223372036854775808L;
            }
            else
            if (l == 24 && lexeme[l] == '1')
            {
                for (i = 2; i < 23; i++)
                    number[i-1] = Complement1Octal(lexeme[i]);
                number[0] = '0'; number[l-2] = 'L'; number[l-1] = 0;
                Attribute.integerLiteral.gInt.lValue.B64 =
                    -(strtoll(number, &d, 8) + 1);
            }
            else
                Attribute.integerLiteral.gInt.lValue.B64 =
                    strtoll(lexeme, &d, 8);
        }
    }
}

```

```

else
{
    if (l > 12 || (l == 12 && strcmp(lexeme, "037777777777") > 0))
        Attribute.integerLiteral.gInt.lOverflow = 1;
    else
    {
        if (l == 12 && strcmp(lexeme, "020000000000") == 0)
        {
            Attribute.integerLiteral.gInt.lMinValue = 1;
            Attribute.integerLiteral.gInt.lValue.B32 = -2147483648;
        }
        else
            if (l == 12 && lexeme[l] >= '2')
            {
                for (i = 2; i < 12; i++)
                    number[i] = Complement1Octal(lexeme[i]);
                number[0] = '0'; number[1] = 0;
                if (lexeme[l] == '2') number[l] = '1'; else number[l] = '0';
                Attribute.integerLiteral.gInt.lValue.B32 =
                    -(strtol(number, &d, 8) + 1);
            }
            else
                Attribute.integerLiteral.gInt.lValue.B32 =
                    strtol(lexeme, &d, 8);
        }
    }
}

```

```
void SetAttributesDecimal(char lexeme[], unsigned int l)
{
    /* Description: Compute the attributes of a decimal integer literal.
    Author:      Richard St-Denis.
    Date:       February 2006.
    Input:      lexeme: a C string of characters representing the lexeme.
                l: length of the lexeme.
    Output:     None.
    */

    char number[256];
    char *d;

    Attribute.integerLiteral.gInt.lValue.B64 = 0;
    Attribute.integerLiteral.gInt.lGroup = Int;
    Attribute.integerLiteral.gInt.lMinValue = 0;
    Attribute.integerLiteral.gInt.lOverflow = 0;

    if (lexeme[l-1] == 'l' || lexeme[l-1] == 'L')
    {
        Attribute.integerLiteral.gInt.lGroup = Long;
        strncpy(number, lexeme, l-1); number[l-1] = 0;
        if (l > 20 || (l == 20 && strcmp(number, "9223372036854775808") > 0))
            Attribute.integerLiteral.gInt.lOverflow = 1;
        else
            if (l == 20 && strcmp(number, "9223372036854775808") == 0)
            {
                Attribute.integerLiteral.gInt.lMinValue = 1;
                Attribute.integerLiteral.gInt.lValue.B64 = -9223372036854775808L;
            }
            else
                Attribute.integerLiteral.gInt.lValue.B64 = strtoll(lexeme, &d, 10);
    }
    else
    {
        if (l > 10 || (l == 10 && strcmp(lexeme, "2147483648") > 0))
            Attribute.integerLiteral.gInt.lOverflow = 1;
        else
            if (l == 10 && strcmp(lexeme, "2147483648") == 0)
            {
                Attribute.integerLiteral.gInt.lMinValue = 1;
                Attribute.integerLiteral.gInt.lValue.B32 = -2147483648;
            }
            else
                Attribute.integerLiteral.gInt.lValue.B32 = strtol(lexeme, &d, 10);
    }
}
```

```
/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.
*/

#ifdef JavaCharAttributes
#define JavaCharAttributes

/* JavaCharAttributes.h */

/* Description: Definition of attributes for character literals.
Author:      Richard St-Denis.
Date:       February 2006.
*/

#include "StringM.h" /* cocktail: String memory */

typedef struct { tStringRef lLexeme;
                unsigned short lValue; } tCharLiteral;

extern const tCharLiteral NoCharLiteral;

extern void SetAttributesChar(char [], unsigned int);

#endif
```

```
/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.
*/

/* JavaCharAttributes.c */

#include "JavaEscapeSeq.h"

#include "Scanner.h" /* Generated by cocktail: variable Attribute */

/* Description: Implementation of functions for character literals.
Author: Richard St-Denis.
Date: February 2006.
*/
```

```
const tCharLiteral NoCharLiteral = { 0, 0 };

/* ----- */

void SetAttributesChar(char lexeme[], unsigned int l)
{
/* Description: Compute the attributes of a character literal.
Author: Richard St-Denis.
Date: February 2006.
Input: lexeme: a C string of characters representing the lexeme.
l: length of the lexeme.
Output: None.
*/

if (lexeme[l] != '\\')
Attribute.charLiteral.gChr.lValue = lexeme[l];
else
Attribute.charLiteral.gChr.lValue = GetEscapeValue(&lexeme[2]).value;
}
```



```
/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.
*/

#ifdef JavaStringAttributes
#define JavaStringAttributes

/* JavaStringAttributes.h */

/* Description: Definition of attributes for string literals.
Author:      Richard St-Denis.
Date:       February 2006.
*/

#include "StringM.h" /* cocktail: String memory */

typedef struct { tStringRef lLexeme;
                unsigned int lLength;
                unsigned short *lString; } tStringLiteral;

extern const tStringLiteral NoStringLiteral;

extern void SetAttributesString(char [], unsigned int);

#endif
```

```

/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.
*/

/* JavaStringAttributes.c */

#include "JavaEscapeSeq.h"

#include "Scanner.h" /* Generated by cocktail: variable Attribute */

/* Description: Implementation of functions for string literals.
Author:      Richard St-Denis.
Date:       February 2006.
*/

```

```

const tStringLiteral NoStringLiteral = { 0, 0, 0 };

/* ----- */

void SetAttributesString(char lexeme[], unsigned int l)
{
    /* Description: Compute the attributes of a string literal.
    Author:      Richard St-Denis.
    Date:       February 2006.
    Input:      lexeme: a C string of characters representing the lexeme.
                l: length of the lexeme.
    Output:     None.
    */

    unsigned int i = 1;
    unsigned int k = 0;

    PairValueNext x;

    /* Compute the length of the string */
    while (i < l-1)
    {
        if (lexeme[i] == '\\')
            i = i + GetEscapeValue(&lexeme[i+1]).next;
        i++; k++;
    }

    Attribute.stringLiteral.gStr.lLength = k;
    Attribute.stringLiteral.gStr.lString = (unsigned short *)
        malloc(k * sizeof(short));

    /* Store the string */
    i = 1; k = 0;
    while (i < l-1)
    {
        if (lexeme[i] != '\\')
        {
            Attribute.stringLiteral.gStr.lString[k] = lexeme[i];
        }
        else
        {
            x = GetEscapeValue(&lexeme[i+1]); i = i + x.next;
            Attribute.stringLiteral.gStr.lString[k] = x.value;
        }
        i++; k++;
    }
}

```

```
/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.
*/

#ifdef JavaEscapeSeq
#define JavaEscapeSeq

/* JavaEscapeSeq.h */

/* Description: Definition of an escape character.
Author:      Richard St-Denis.
Date:       February 2006.
*/

typedef struct { unsigned short value;
                unsigned short next;
                } PairValueNext;

extern PairValueNext GetEscapeValue(char []);

#endif
```

```

/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.

                                */
                                /* JavaEscapeSeq.c
#include "JavaEscapeSeq.h"

/* Description: Implementation of functions for escape characters.
Author:      Richard St-Denis.
Date:       February 2006.
                                */

```

```

PairValueNext GetEscapeValue(char esc[])
{
/* Description: Compute the attributes of an escape character.
Author:      Richard St-Denis.
Date:       February 2006.
Input:      esc: a C string of characters representing an
              escape character.
Output:     Internal value of the character with the relative next
              position.
                                */

PairValueNext x;

x.next = 1;
switch (esc[0])
{
case 'b' : x.value = 0x8; break;
case 't' : x.value = 0x9; break;
case 'n' : x.value = 0xa; break;
case 'f' : x.value = 0xc; break;
case 'r' : x.value = 0xd; break;
case '\"' : x.value = 0x22; break;
case '\'' : x.value = 0x27; break;
case '\\' : x.value = 0x5c; break;
case '0' : case '1' : case '2' : case '3' : case '4' :
case '5' : case '6' : case '7' :
{
unsigned short v = esc[0] - 0x30;
unsigned short n = 1;

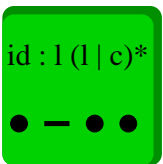
if (esc[0] <= '3')
{
if (esc[1] >= '0' && esc[1] <= '7')
{
n = 2; v <= 3; v += esc[1] - 0x30;
if (esc[2] >= '0' && esc[2] <= '7')
{
n = 3; v <= 3; v += esc[2] - 0x30;
}
}
}
else
if (esc[1] >= '0' && esc[1] <= '7')
{
n = 2; v <= 3; v += esc[1] - 0x30;
}

x.value = v; x.next = n;
}
}
return x;
}

```

Annexe C

La spécification de l'analyseur lexical



id : 1 (1 | c)*

Le fichier `java--.scn` contient la spécification de l'analyseur lexical. Il est divisé en dix sections identifiées par les clauses `EXPORT`, `GLOBAL`, `BEGIN`, `CLOSE`, `LOCAL`, `DEFAULT`, `EOF`, `DEFINE`, `START` et `RULE`.

- Le code *C* sous la clause `EXPORT` est inséré intégralement dans le fichier `Scanner.h` généré par l'outil `rex`, à l'exception de la commande

`INSERT tScanAttribute`

qui est interprétée par l'outil `rpp`. Cette commande récupère les déclarations des structures de données propres aux attributs intrinsèques, en particulier la déclaration du type *tScanAttribute*. Ces déclarations sont contenues dans le fichier `Scanner.rpp` généré par l'outil `lpp` à partir des données du fichier `java--.prs`.

- Le code *C* sous la clause `GLOBAL` est inséré intégralement au niveau global dans le fichier `Scanner.c` généré par l'outil `rex`, à l'exception de la commande

`INSERT ErrorAttribute`

qui est interprétée par l'outil `rpp`. Cette commande récupère la définition de la fonction *ErrorAttribute* contenue dans le fichier `Scanner.rpp` généré par l'outil `lpp`. Cette fonction est appelée par l'analyseur syntaxique, lors de l'insertion automatique d'une unité lexicale à la suite de la détection d'une erreur syntaxique, dans le but d'obtenir les valeurs par défaut de ses attributs.

- Le code *C* sous la clause `BEGIN` est inséré intégralement dans la fonction *BeginScanner* de l'analyseur lexical (`Scanner.c`).
- Le code *C* sous la clause `CLOSE` est inséré intégralement dans la fonction *CloseScanner* de l'analyseur lexical (`Scanner.c`).
- Le code *C* sous la clause `LOCAL` est inséré intégralement au début de la fonction *GetToken* de l'analyseur lexical (`Scanner.c`).
- Le code *C* sous la clause `DEFAULT` correspond au traitement par défaut associé à la lecture d'un caractère qui n'est pas un préfixe d'un lexème appartenant au langage défini par une expression régulière contenue dans la section identifiée par la clause `RULE`.


```

/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.
*/

/* java--.scn */

/* Description: Specification of lexical rules for java--.
Author:      Richard St-Denis.
Date:       February 2006.

EXPORT      /* Code to be inserted in Scanner.h */
{
#include "Position.h"          /* cocktail: Source positions handler */
#include "JavaFPAttributes.h"  /* Attributes for FP literals */
#include "JavaIntAttributes.h" /* Attributes for integer literals */
#include "JavaStringAttributes.h" /* Attributes for string literals */

INSERT tScanAttribute /* Data structures for all attributes */
}

GLOBAL      /* Code to be inserted in Scanner.c (level 0) */
{
#include "rabc.h" /* cocktail: Boolean type */
#include "StringM.h" /* cocktail: String memory */
#include "Idents.h" /* cocktail: Identifier (string) table */
#include "Errors.h" /* cocktail: Error handler */

INSERT ErrorAttribute /* Function to be called when a token is added */
/* because a syntactic error */

static rbool bModuleInitialized = rfalse; /* Module initialization flag */
static rbool bModuleFinalized = rfalse; /* Module finalization flag */
}

```

```

BEGIN      /* Code for initializing the scanner */
{
if (! bModuleInitialized)
{
/* Once code */
fprintf(stderr, "      Initializing the scanner...\n");

bModuleInitialized = rtrue;
}
}

CLOSE      /* Code for finalizing the scanner */
{
if (bModuleInitialized && (! bModuleFinalized))
{
/* Once code */
fprintf(stderr, "      Finalizing the scanner...\n");

bModuleFinalized = rtrue;
}
}

LOCAL      /* Code to be inserted in GetToken */
{
static int WordSize = 256;
char Word[WordSize]; /* Temporary array for the current lexeme */
unsigned int l; /* Length of the current lexeme */
}

DEFAULT    /* Code for an illegal input character */
{
MessageI("illegal character", xxError, Attribute.Position,
xxCharacter, TokenPtr);
}

EOF      /* Code for the end of file */
{
/* Fatal error */
if (yyStartState == Comment) Message("unclosed comment",
xxError, Attribute.Position);
}

DEFINE      /* Regular expressions to avoid duplication */
javaDigit = {0-9}.
javaLetter = {a-zA-Z$_}.
digit = {0-9}.
zeroToThree = {0-3}.
nonZeroDigit = {1-9}.
octalDigit = {0-7}.
hexDigit = {0-9A-Fa-f}.
singleCharacter = -{\\|'|\"\\n\\r}.
stringCharacter = -{\\|'|\"\\n\\r}.
escapeSequence = \\ (b | t | n | f | r | \\\" | \\' | \\ |
octalDigit[1-2] | zeroToThree octalDigit[2]).

START      /* Internal states of the scanner */
Comment

```

```

RULE                                /* Lexical rules                */
                                /* Comments                        */
"//" (-{\n\r})* > : { }

"/*"                                : { yyStart(Comment); }
#Comment# "*/"                      : { yyStart(STD); }
#Comment# "/*" | -{*\\n\\r}+ : { }

INSERT RULES #STD#                  /* Tokens generated by rpp    */

"const" : {
    printf(" *** The keyword const is a reserved keyword.\\n");
    return 999;
}

"goto" : {
    printf(" *** The keyword goto is a reserved keyword.\\n");
    return 999;
}

javaLetter (javaLetter | javaDigit)* :          /* Identifier                */
{
    /* The token is mapped to a unique integer */
    /* (long) which is returned by MakeIdent */
    Attribute.identifier.gIdent = MakeIdent(TokenPtr, TokenLength);
    return identifier;
}

\\' (singleCharacter | escapeSequence) \\' :      /* Character literal          */
{
    if (TokenLength < WordSize)
    {
        l = GetWord(Word);
        printf(" *** Character literals are not supported.\\n");
        return 998;
    }
    else
        Message("lexeme too long", xxFatal, Attribute.Position);
}

\\" (stringCharacter | escapeSequence)* \\" :      /* String literal             */
{
    if (TokenLength < WordSize)
    {
        l = GetWord(Word);
        SetAttributesString(Word, l);
        Attribute.stringLiteral.gStr.lLexeme = PutString(TokenPtr,
                                                            TokenLength);
        return stringLiteral;
    }
    else
        Message("lexeme too long", xxFatal, Attribute.Position);
}

```

```

("0" | nonZeroDigit digit*) ("l" | "L")? :      /* Integer literal           */
{
    if (TokenLength < WordSize)
    {
        l = GetWord(Word);
        SetAttributesDecimal(Word, l);
        Attribute.integerLiteral.gInt.lLexeme = PutString(TokenPtr,
                                                            TokenLength);
        return integerLiteral;
    }
    else
        Message("lexeme too long", xxFatal, Attribute.Position);
}

"0" octalDigit+ ("l" | "L")? :
{
    if (TokenLength < WordSize)
    {
        l = GetWord(Word);
        SetAttributesOctal(Word, l);
        Attribute.integerLiteral.gInt.lLexeme = PutString(TokenPtr,
                                                            TokenLength);
        return integerLiteral;
    }
    else
        Message("lexeme too long", xxFatal, Attribute.Position);
}

("0x" | "0X") hexDigit+ ("l" | "L")? :
{
    if (TokenLength < WordSize)
    {
        l = GetWord(Word);
        SetAttributesHex(Word, l);
        Attribute.integerLiteral.gInt.lLexeme = PutString(TokenPtr,
                                                            TokenLength);
        return integerLiteral;
    }
    else
        Message("lexeme too long", xxFatal, Attribute.Position);
}

```



```

/* Floating point literal */
(digit+ "." digit* | "." digit+
 ("e" | "E") ("+" | "-")? digit+)?
 ("f" | "F" | "d" | "D")? :
{
    if (TokenLength < WordSize)
    {
        l = GetWord(Word);
        SetFPAttributesDecimal(Word, l);
        Attribute.flPointLiteral.gFP.lLexeme = PutString(TokenPtr,
                                                         TokenLength);
        return flPointLiteral;
    }
    else
        Message("lexeme too long", xxFatal, Attribute.Position);
}

digit+ ("e" | "E") ("+" | "-")? digit+
 ("f" | "F" | "d" | "D")? :
{
    if (TokenLength < WordSize)
    {
        l = GetWord( Word );
        SetFPAttributesDecimal(Word, l);
        Attribute.flPointLiteral.gFP.lLexeme = PutString(TokenPtr,
                                                         TokenLength );
        return flPointLiteral;
    }
    else
        Message("lexeme too long", xxFatal, Attribute.Position);
}

digit+ (("e" | "E") ("+" | "-")? digit+)?
 ("f" | "F" | "d" | "D") :
{
    if (TokenLength < WordSize)
    {
        l = GetWord(Word);
        SetFPAttributesDecimal(Word, l);
        Attribute.flPointLiteral.gFP.lLexeme = PutString(TokenPtr,
                                                         TokenLength);
        return flPointLiteral;
    }
    else
        Message("lexeme too long", xxFatal, Attribute.Position);
}

```

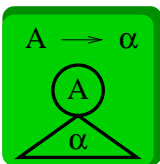
```

("0x" | "0X") ((hexDigit+ ".")? | (hexDigit* "." hexDigit+))
 ("p" | "P") ("+" | "-")? digit+
 ("f" | "F" | "d" | "D")? :
{
    if (TokenLength < WordSize)
    {
        l = GetWord(Word);
        SetFPAttributesHex(Word, l);
        Attribute.flPointLiteral.gFP.lLexeme = PutString(TokenPtr,
                                                         TokenLength);
        printf(" *** Hex floating point literals are not supported.\n");
        return 997;
    }
    else
        Message("lexeme too long", xxFatal, Attribute.Position);
}

```


Annexe D

La spécification de l'analyseur syntaxique



Le fichier `java--.prs` contient la spécification de l'analyseur syntaxique. Il comporte deux modules.

Le premier module, nommé *ConcreteSyntax*, est divisé en quatre sections identifiées par les clauses `PARSER`, `PREC`, `PROPERTY` et `RULE`.

- La clause `PARSER` spécifie le nom des fichiers qui contiennent le code *C* de l'analyseur syntaxique (`Parser.h` et `Parser.c`).
- Le code sous la clause `PREC` n'est pas du code *C*. Il indique la priorité et l'associativité des opérateurs. L'opérateur « `||` » est le moins prioritaire (car il est le premier de la liste) et associatif à gauche (`LEFT`). Les opérateurs « `*` », « `/` » et « `%` » sont les plus prioritaires (car ils sont les derniers de la liste) et associatifs à gauche (`LEFT`). Cependant, ils ont tous la même priorité.
- La clause `PROPERTY` ou la commande

PROPERTY INPUT

permet d'utiliser une forme plus simple par rapport à celle d'une version antérieure de l'outil `ast`.

- Le code sous la clause `RULE` contient toutes les règles de production de la grammaire hors contexte du langage *java--*. À la fin de cette section, il y a la définition des attributs intrinsèques ou d'entrée (`IN`) associés aux symboles terminaux *identifier*, *stringLiteral*, *integerLiteral* et *flPointLiteral*.

Le deuxième module, nommé *ASTBuilder*, est divisé en sept sections identifiées par les clauses `PARSER`, `EXPORT`, `GLOBAL`, `BEGIN`, `CLOSE`, `DECLARE` et `RULE`.

- La clause `PARSER` a la même signification que celle du premier module.
- Le code *C* sous la clause `EXPORT` est inséré intégralement dans le fichier `Parser.h` généré par l'outil `lark`.
- Le code *C* sous la clause `GLOBAL` est inséré intégralement au niveau global dans le fichier `Parser.c` généré par l'outil `lark`.
- Le code *C* sous la clause `BEGIN` est inséré intégralement dans la fonction *BeginParser* de l'analyseur syntaxique (`Parser.c`). Cette section contient essentiellement un appel de

la fonction *BeginScanner* et un appel de la fonction *BeginTree*. La première fonction initialise l'analyseur lexical. La deuxième fonction crée des noeuds constants, ceux associés aux types primitifs.

- Le code *C* sous la clause **CLOSE** est inséré intégralement dans la fonction *CloseParser* de l'analyseur syntaxique (**Parser.c**).
- Le code sous la clause **DECLARE** n'est pas du code *C*. Il associe à certains symboles (et leurs descendants) l'attribut synthétisé (**SYN**) *gPtr*, un pointeur sur un noeud de l'arbre syntaxique abstrait (le type *tTree*). Il définit aussi des attributs associés aux symboles *Modifiers*, *Modifier*, *MethodHeader*, *MethodInvocation*, *ArrayCreatorSpec*, *Block*, *Type*, *PrimitiveType* et *ArrayType*.
- Le code sous la clause **RULE** est un mélange de code *C* et de code particulier à l'outil **ast** pour construire l'arbre syntaxique abstrait.

```

/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.

                                */
                                /* java--.prs                                */
/* Description: Specification of syntactic rules for java-- and
construction of the abstract syntax tree.
Author:      Daniel Cote and Richard St-Denis.
Date:        February 2006.
                                */

MODULE ConcreteSyntax          /* Context-free grammar (concrete syntax) */
PARSER Parser                  /* Generate code into Parser.h and Parser.c */
PREC                          /* Precedence and associativity of operators */

    LEFT '|' '|'               /* left-associative and lower priority */
    LEFT '&&'
    LEFT '==' '!='
    LEFT '<' '>' '<=' '>='
    LEFT '+' '-'
    LEFT '*' '/' '%'           /* left-associative and higher priority */

PROPERTY INPUT                 /* Use of a simpler form compared with the one
                                of a former version of ast.
                                */

```

```

RULE                          /* Context-free grammar                                */
                                /* Compilation unit                                */
CompilationUnit = Opt_semis TypeDeclarations .                                */
                                /* Type declarations                                */
TypeDeclarations = <
    OneTypeDeclaration = TypeDeclaration .
> .                                /* Type declaration                                */

TypeDeclaration = <
    ClassTypeDeclaration =
        Modifiers 'class' identifier '{' ClassBodyDeclarations '}' Opt_semis .
> .                                /* Semicolon                                */

Opt_semis = <
    = .
    = Opt_semis ';' .
> .                                /* Modifiers                                */

Modifiers = <
    NoModifiers = .
    SomeModifiers = Modifiers Modifier .
> .                                /* Modifier                                */

Modifier = <
    PublicModifier = 'public' .
    StaticModifier = 'static' .
> .                                /* Class body declarations                                */

ClassBodyDeclarations = <
    NoClassBodyDeclarations = .
    SomeClassBodyDeclarations = ClassBodyDeclarations <
        FieldDeclaration = Modifiers Type FieldDeclarators ';' .
        MethodDeclaration = Modifiers MethodHeader ConstructorOrMethodBody .
    > .
> .                                /* Type                                */

Type = <
    BasicType = PrimitiveType .
    ReferenceType = ArrayType .
> .                                /* Primitive type                                */

PrimitiveType = <
    IntType = 'int' .
    FloatType = 'float' .
    BooleanType = 'boolean' .
> .                                /* Array type                                */

ArrayType = Type '[' ']' .                                /* Field declarators                                */

FieldDeclarators = <
    FirstFieldDeclarator = identifier .
    LastFieldDeclarators = FieldDeclarators ',' identifier .
> .                                /* Constructor or method body                                */

ConstructorOrMethodBody = Block .

```

```

/* Method header */
MethodHeader = <
  Procedure = 'void' identifier FormalParameters .
  Function = Type identifier FormalParameters .
> .

/* Formal parameters */
FormalParameters = <
  NoParameter = '(' ' ' )' .
  SomeParameters = '(' FormalParameterDeclarations ')' .
> .

/* Formal parameter declarations */
FormalParameterDeclarations = <
  OneFormalParameter = Type identifier .
  MoreFormalParameters = FormalParameterDeclarations ','
  Type identifier .
> .

/* Block */
Block = '{' BlockStatements '}' .

/* Block statements */
BlockStatements = <
  NoBlockStatements = Opt_semis .
  SomeBlockStatements = BlockStatements BlockStatement Opt_semis .
> .

/* Block statement */
BlockStatement = <
  LocalVariableDeclarationStatement = LocalVariableDeclaration ';' .
  InnerBlock = Block .
  UsualStatement = <
    UnbalancedStatements = UnbalancedStatement .
    BalancedStatements = Statement .
  > .
> .

/* Local variable declaration */
LocalVariableDeclaration = Type VariableDeclarators .

/* Variable declarators */
VariableDeclarators = <
  FirstVariableDeclarator = identifier .
  LastVariableDeclarators = VariableDeclarators ',' identifier .
> .

/* Unbalanced statement */
UnbalancedStatement = <
  UnbalancedIf = 'if' '(' Expression ')' <
    IfThen = UsualStmtOrBlk .
    IfThenElseUnbalanced = gThen:StmtOrBlk 'else' gElse:UnbalancedStatement .
  > .
  UnbalancedWhile = 'while' '(' Expression ')' UnbalancedStatement .
> .

```

```

/* Usual statement or block */
UsualStmtOrBlk = <
  UsualEmpty = ';' .
  UsualStmt = UsualStatement .
  UsualBlk = Block .
> .

/* Statement or block */
StmtOrBlk = <
  Empty = ';' .
  Stmt = Statement .
  Blk = Block .
> .

/* Statement */
Statement = <
  IfThenElseBalanced = 'if' '(' Expression ')'
    gThen:StmtOrBlk 'else' gElse:StmtOrBlk .
  WhileStatement = 'while' '(' Expression ')' StmtOrBlk .
  ReturnNoValueStatement = 'return' ';' .
  ReturnValueStatement = 'return' Expression ';' .
  ExpressionStatement = StatementExpression ';' .
  Read = 'read' '(' LeftHandSide ')' ';' .
  Println = 'println' '(' ' ' ')' ';' .
  PrintStringLiteral = 'printstr' '(' stringLiteral ')' ';' .
  PrintValue = 'print' '(' Expression ')' ';' .
> .

/* Statement expression */
StatementExpression = <
  Assignment = LeftHandSide '=' Expression .
  ProcedureCall = MethodInvocation .
> .

/* Left-hand side */
LeftHandSide = <
  SimpleVariableLHS = identifier .
  ArrayAccessLHS = VarRef '[' Expression ']' .
> .

/* Method invocation */
MethodInvocation = identifier Arguments .

/* Arguments */
Arguments = <
  NoArguments = '(' ' ' )' .
  SomeArguments = '(' ListOfArguments ')' .
> .

/* List of arguments */
ListOfArguments = <
  OneArgument = Expression .
  MoreArguments = ListOfArguments ',' Expression .
> .

/* Expression */
Expression = ConditionalExpression .

/* Conditional expression */
ConditionalExpression = BooleanExpression .

```

```

/* Boolean expression */
BooleanExpression = <
  Infix = InfixExpression .
  Lt = gLop:BooleanExpression '<' gRop:BooleanExpression .
  Gt = gLop:BooleanExpression '>' gRop:BooleanExpression .
  Leq = gLop:BooleanExpression '<=' gRop:BooleanExpression .
  Geq = gLop:BooleanExpression '>=' gRop:BooleanExpression .
  Eq = gLop:BooleanExpression '=' gRop:BooleanExpression .
  Neq = gLop:BooleanExpression '!=' gRop:BooleanExpression .
  And = gLop:BooleanExpression '&&' gRop:BooleanExpression .
  Or = gLop:BooleanExpression '||' gRop:BooleanExpression .
> .

/* Infix expression */
InfixExpression = <
  Unary = UnaryExpression .
  Mul = gLop:InfixExpression '*' gRop:InfixExpression .
  Div = gLop:InfixExpression '/' gRop:InfixExpression .
  Mod = gLop:InfixExpression '%' gRop:InfixExpression .
  Plus = gLop:InfixExpression '+' gRop:InfixExpression .
  Minus = gLop:InfixExpression '-' gRop:InfixExpression .
> .

/* Unary expression */
UnaryExpression = <
  Postfix = PostfixExpression .
  Uminus = '-' UnaryExpression .
  Not = '!' UnaryExpression .
> .

/* Postfix expression */
PostfixExpression = Primary .

/* Primary */
Primary = <
  ParExpression = '(' Expression ')' .
  TrueConst = 'true' .
  FalseConst = 'false' .
  IntConst = integerLiteral .
  FloatConst = flPointLiteral .
  NullRefConst = 'null' .
  VarRef = <
    SimpleVariable = identifier .
    ArrayAccess = VarRef '[' Expression ']' .
  > .
  FunctionCall = MethodInvocation .
  ArrayCreator = 'new' ArrayCreatorSpec .
> .

/* Array creator expression */
ArrayCreatorSpec = <
  FullInitACSpec = <
    BasicACSpec = PrimitiveType '[' Expression ']' .
    SuperACSpec = FullInitACSpec '[' Expression ']' .
  > .
  PartInitACSpec = <
    BasicNDACSpec = FullInitACSpec '[' ']' .
    SuperNDACSpec = PartInitACSpec '[' ']' .
  > .
> .

```

```

/* Terminals with their attributes */
identifier : [gIdent : tIdent IN] { gIdent := NoIdent; } .
stringLiteral : [gStr : tStringLiteral IN] { gStr := NoStringLiteral; } .
integerLiteral : [gInt : tIntegerLiteral IN] { gInt := NoIntegerLiteral; } .
flPointLiteral : [gFP : tFPLiteral IN] { gFP := NoFPLiteral; } .

END ConcreteSyntax

```

```

MODULE ASTBuilder          /* S-Attribution rules for building the AST */
PARSER Parser              /* Generate code into Parser.h and Parser.c */
EXPORT                     /* Code to be inserted in Parser.h */
{
  extern char *sourceFileName;          /* Supplied by the driver java--.c */
}

GLOBAL                     /* Code to be inserted in Parser.c (level 0) */
{
  #include "StringM.h"          /* cocktail: String memory */
  #include "Errors.h"          /* cocktail: Error handler */
  #include "Idents.h"          /* cocktail: Identifier (string) table */
  #include "Position.h"        /* cocktail: Source positions handler */

  #include "Tree.h"            /* Generated by cocktail: AST node constructors */

  char *sourceFileName = NULL;

  static rbool bModuleInitialized = rfalse; /* Module initialization flag */
  static rbool bModuleFinalized = rfalse; /* Module finalization flag */
}

BEGIN                      /* Code for initializing the parser */
{
  if ( ! bModuleInitialized )
  {
    fprintf( stderr, "    Initializing the parser...\n" );

    BeginScanner();          /* Initialize the scanner */
    BeginTree();             /* Initialize the tree module */

    bModuleInitialized = rtrue;
  }
}

CLOSE                      /* Code for finalizing the parser */
{
  if ( bModuleInitialized && ( ! bModuleFinalized ) )
  {
    fprintf( stderr, "    Finalizing the parser...\n" );
    CloseTree();             /* Finalize the tree module */
    CloseScanner();          /* Finalize the scanner */

    bModuleFinalized = rtrue;
  }
}

```

```

DECLARE                     /* Declaration of attributes */

/* Most concrete syntax constructs need to either build a sub-AST node, or
   copy their child node sub-AST to relay it up to their parent, so they
   need an AST node pointer */

TypeDeclarations TypeDeclaration
ClassBodyDeclarations
Type PrimitiveType ArrayType
FieldDeclarators ConstructorOrMethodBody
FormalParameters FormalParameterDeclarations
Block BlockStatements BlockStatement
LocalVariableDeclaration VariableDeclarators
UnbalancedStatement UsualStmtOrBlk StmtOrBlk Statement
StatementExpression LeftHandSide Arguments ListOfArguments
Expression ConditionalExpression BooleanExpression InfixExpression
UnaryExpression PostfixExpression Primary = [gPtr: tTree SYN] .

/* Some concrete syntax constructs do not build sub-AST nodes, but they
   relay proper information to their concrete syntax parent to enable it
   to build the corresponding sub-AST */

Modifiers Modifier = [modifiers: short SYN] .
Modifier          = [Position: tPosition SYN] .

MethodHeader = [gType: tTree SYN]
               [gMidPos: tPosition SYN]
               [gMid: tIdent SYN]
               [gPrms: tTree SYN] .

MethodInvocation = [gMidPos: tPosition SYN]
                  [gMid: tIdent SYN]
                  [gArgs: tTree SYN] .

ArrayCreatorSpec = [gArrayType: tTree SYN]
                  [gDimSpec: tTree SYN]
                  [gNbInitDim: short SYN]
                  [gPos: tPosition SYN] .

Block Type PrimitiveType ArrayType = [gPos: tPosition SYN] .

```



```

RULE                                /* Construction of the abstract syntax tree */
                                /* Compilation unit */
CompilationUnit = {
=> {
    tStringRef src = ( sourceFileName == NULL )
        ? NoString
        \: PutString1( sourceFileName );

    TreeRoot = maCompilationUnit( src,
                                ReverseTree( TypeDeclarations:gPtr ) );
};
} .                                /* Type declarations */
OneTypeDeclaration = {
gPtr := {
    TypeDeclaration:gPtr->aTypeDeclaration.Next = maNoTypeDeclarations();
gPtr = TypeDeclaration:gPtr;
};
} .
ClassTypeDeclaration = {
gPtr := maClassDeclaration( NoTree,
    Modifiers:modifiers,
    identifier:gIdent,
    identifier:Position,
    ReverseTree( ClassBodyDeclarations:gPtr ) );
} .                                /* Modifiers */
NoModifiers = {
modifiers := ModNONE;
} .
SomeModifiers = {
modifiers := {
    if ( Modifiers:modifiers & Modifier:modifiers )
        Message( "Redundant modifier.", xxError, Modifier:Position );
    else
        modifiers = Modifiers:modifiers | Modifier:modifiers;
};
} .                                /* Modifier */
PublicModifier = {
modifiers := ModPUBLIC;
Position := 'public':Position;
} .
StaticModifier = {
modifiers := ModSTATIC;
Position := 'static':Position;
} .                                /* Class body declarations */
NoClassBodyDeclarations = {
gPtr := maNoClassBodyDeclarations();
} .

```

```

FieldDeclaration = {
gPtr := maFieldDeclaration( ClassBodyDeclarations:gPtr,
    Modifiers:modifiers,
    Type:gPtr,
    ReverseTree( FieldDeclarators:gPtr ) );
} .
MethodDeclaration = {
gPtr := maMethodDeclaration( ClassBodyDeclarations:gPtr,
    Modifiers:modifiers,
    MethodHeader:gType,
    MethodHeader:gMid,
    MethodHeader:gMidPos,
    MethodHeader:gPrms,
    ConstructorOrMethodBody:gPtr );
} .                                /* Type */
BasicType =
{
gPtr := PrimitiveType:gPtr;
gPos := PrimitiveType:gPos;
} .
ReferenceType =
{
gPtr := ArrayType:gPtr;
gPos := ArrayType:gPos;
} .                                /* Primitive type */
IntType =
{
gPtr := nIntType;
gPos := 'int':Position;
} .
FloatType =
{
gPtr := nFloatType;
gPos := 'float':Position;
} .
BooleanType =
{
gPtr := nBooleanType;
gPos := 'boolean':Position;
} .                                /* Array type */
ArrayType =
{
gPtr := maArrayType( Type:gPtr );
gPos := Type:gPos;
} .

```

```

/* Field declarators */
FirstFieldDeclarator = {
    gPtr := maFieldDeclarator( maEndFieldDeclarators(),
                              identifier:gIdent,
                              identifier:Position );
} .

LastFieldDeclarators = {
    gPtr := maFieldDeclarator( FieldDeclarators:gPtr,
                              identifier:gIdent,
                              identifier:Position );
} .

/* Constructor or method body */
ConstructorOrMethodBody = {
    gPtr := Block:gPtr;
} .

/* Method header */
Procedure = {
    gType := nVoidType;
    gMidPos := identifier:Position; gMid := identifier:gIdent;
    gPrms := FormalParameters:gPtr;
} .

Function = {
    gType := Type:gPtr;
    gMidPos := identifier:Position; gMid := identifier:gIdent;
    gPrms := FormalParameters:gPtr;
} .

/* Formal parameters */
NoParameter = {
    gPtr := maNoParameter();
} .

SomeParameters = {
    gPtr := ReverseTree( FormalParameterDeclarations:gPtr );
} .

/* Formal parameter declarations */
OneFormalParameter = {
    gPtr := maFormalParameter( maNoParameter(),
                              Type:gPtr,
                              identifier:gIdent,
                              identifier:Position );
} .

MoreFormalParameters = {
    gPtr := maFormalParameter( FormalParameterDeclarations:gPtr,
                              Type:gPtr,
                              identifier:gIdent,
                              identifier:Position );
} .

```

```

/* Block */
Block = {
    gPtr := ReverseTree( BlockStatements:gPtr );
    gPos := '':Position;
} .

/* Block statements */
NoBlockStatements = {
    gPtr := maNoStatement();
} .

SomeBlockStatements = {
    gPtr := { BlockStatement:gPtr->aStatement.Next = BlockStatements:gPtr;
              gPtr = BlockStatement:gPtr; };
} .

/* Block statement */
LocalVariableDeclarationStatement = {
    gPtr := LocalVariableDeclaration:gPtr;
} .

InnerBlock = {
    gPtr := maInnerBlock( NoTree, Block:gPos, Block:gPtr );
} .

UnbalancedStatements = {
    gPtr := UnbalancedStatement:gPtr;
} .

BalancedStatements = {
    gPtr := Statement:gPtr;
} .

/* Local variable declaration */
LocalVariableDeclaration = {
    gPtr :=
        maLocalVariableDeclaration( NoTree,
                                    Type:gPos,
                                    Type:gPtr,
                                    ReverseTree( VariableDeclarators:gPtr ) );
} .

/* Variable declarators */
FirstVariableDeclarator = {
    gPtr := maVariableDeclarator( maNoVariableDeclarators(),
                                  identifier:gIdent,
                                  identifier:Position );
} .

LastVariableDeclarators = {
    gPtr := maVariableDeclarator( VariableDeclarators:gPtr,
                                  identifier:gIdent,
                                  identifier:Position );
} .

```

```

/* Unbalanced statement */
IfThen = {
    gPtr := maIfThen( NoTree,
        'if':Position,
        Expression:gPtr,
        UsualStmtOrBlk:gPtr );
} .

IfThenElseUnbalanced = {
    gPtr := { gElse:gPtr->\aStatement.Next = maNoStatement();
        gPtr = maIfThenElse( NoTree,
            'if':Position,
            Expression:gPtr,
            gThen:gPtr, gElse:gPtr );
    };
} .

UnbalancedWhile = {
    gPtr := { UnbalancedStatement:gPtr->\aStatement.Next = maNoStatement();
        gPtr = maWhileStatement( NoTree,
            'while':Position,
            Expression:gPtr,
            UnbalancedStatement:gPtr );
    };
} .

/* Usual statement or block */
UsualEmpty = {
    gPtr := maNoStatement();
} .

UsualStmt = {
    gPtr := { UsualStatement:gPtr->\aStatement.Next = maNoStatement();
        gPtr = UsualStatement:gPtr;
    };
} .

UsualBlk = {
    gPtr := maInnerBlock( maNoStatement(), Block:gPos, Block:gPtr );
} .

/* Statement or block */
Empty = {
    gPtr := maNoStatement();
} .

Stmt = {
    gPtr := { Statement:gPtr->\aStatement.Next = maNoStatement();
        gPtr = Statement:gPtr;
    };
} .

Blk = {
    gPtr := maInnerBlock( maNoStatement(), Block:gPos, Block:gPtr );
} .

```

```

/* Statement */
IfThenElseBalanced = {
    gPtr := maIfThenElse( NoTree,
        'if':Position,
        Expression:gPtr,
        gThen:gPtr, gElse:gPtr );
} .

WhileStatement = {
    gPtr := maWhileStatement( NoTree,
        'while':Position,
        Expression:gPtr,
        StmtOrBlk:gPtr );
} .

ReturnNoValueStatement = {
    gPtr := maReturnNoValueStatement( NoTree, 'return':Position );
} .

ReturnValueStatement = {
    gPtr := maReturnValueStatement( NoTree,
        'return':Position,
        Expression:gPtr );
} .

ExpressionStatement = {
    gPtr := StatementExpression:gPtr;
} .

Read = {
    gPtr := maRead( NoTree, 'read':Position, LeftHandSide:gPtr );
} .

Println = {
    gPtr := maPrintln( NoTree, 'println':Position );
} .

PrintStringLiteral = {
    gPtr := maPrintStringLiteral( NoTree,
        'printstr':Position,
        stringLiteral:Position,
        stringLiteral:gStr.lLexeme );
} .

PrintValue = {
    gPtr := maPrintValue( NoTree, 'print':Position, Expression:gPtr );
} .

```

```

/* Statement expression */
Assignment = {
  gPtr := maAssignment( NoTree,
    '=':Position,
    LeftHandSide:gPtr,
    Expression:gPtr );
} .

ProcedureCall = {
  gPtr := maProcedureCall( NoTree,
    MethodInvocation:gMidPos,
    MethodInvocation:gMid, MethodInvocation:gMidPos,
    MethodInvocation:gArgs );
} .

/* Left-hand side */
SimpleVariableLHS = {
  gPtr := malValue( identifier:Position, identifier:gIdent );
} .

ArrayAccessLHS = {
  gPtr := maIndexlValue( '[':Position,
    VarRef:gPtr,
    Expression:gPtr );
} .

/* Method invocation */
MethodInvocation = {
  gMid := identifier:gIdent;
  gMidPos := identifier:Position;
  gArgs := Arguments:gPtr;
} .

/* Arguments */
NoArguments = {
  gPtr := maNoArgument( '(':Position );
} .

SomeArguments = {
  gPtr := ReverseTree( ListOfArguments:gPtr );
} .

/* List of arguments */
OneArgument = {
  gPtr := maArgument( maNoArgument( NoPosition ), Expression:gPtr );
} .

MoreArguments = {
  gPtr := maArgument( ListOfArguments:gPtr, Expression:gPtr );
} .

```

```

/* Expression */
Expression = {
  gPtr := ConditionalExpression:gPtr;
} .

/* Conditional expression */
ConditionalExpression = {
  gPtr := BooleanExpression:gPtr;
} .

/* Boolean expression */
Infix = {
  gPtr := InfixExpression:gPtr;
} .

Lt = {
  gPtr := maBinary( '<':Position, OpLT, OpCmp, gLop:gPtr, gRop:gPtr );
} .

Gt = {
  gPtr := maBinary( '>':Position, OpGT, OpCmp, gLop:gPtr, gRop:gPtr );
} .

Leq = {
  gPtr :=
    maUnary( '<=':Position, OpNot, OpLogic,
      maBinary( '<=':Position, OpGT, OpCmp, gLop:gPtr, gRop:gPtr ) );
} .

Geq = {
  gPtr :=
    maUnary( '>=':Position, OpNot, OpLogic,
      maBinary( '>=':Position, OpLT, OpCmp, gLop:gPtr, gRop:gPtr ) );
} .

Eq = {
  gPtr := maBinary( '==':Position, OpEQ, OpEqv, gLop:gPtr, gRop:gPtr );
} .

Neq = {
  gPtr :=
    maUnary( '!=':Position, OpNot, OpLogic,
      maBinary( '!=':Position, OpEQ, OpEqv, gLop:gPtr, gRop:gPtr ) );
} .

And = {
  gPtr := maBinary( '&&':Position, OpAND, OpLogic, gLop:gPtr, gRop:gPtr );
} .

Or = {
  gPtr := maBinary( '||':Position, OpOR, OpLogic, gLop:gPtr, gRop:gPtr );
} .

```

```

/* Infix expression */
Unary = {
  gPtr := UnaryExpression:gPtr;
} .

Mul = {
  gPtr := maBinary( '*' :Position, OpMUL, OpArithmetic, gLop:gPtr, gRop:gPtr );
} .

Div = {
  gPtr := maBinary( '/' :Position, OpDIV, OpArithmetic, gLop:gPtr, gRop:gPtr );
} .

Mod = {
  gPtr := maBinary( '%' :Position, OpMOD, OpArithmetic, gLop:gPtr, gRop:gPtr );
} .

Plus = {
  gPtr := maBinary( '+' :Position, OpPLUS, OpArithmetic, gLop:gPtr, gRop:gPtr );
} .

Minus = {
  gPtr := maBinary( '-' :Position, OpMINUS, OpArithmetic, gLop:gPtr, gRop:gPtr );
} .

```

```

/* Unary expression */
Postfix = {
  gPtr := PostfixExpression:gPtr;
} .

Uminus = {
  gPtr := {
    if ( UnaryExpression:gPtr->Kind == kaIntConst )
    {
      if (
        UnaryExpression:gPtr->aIntConst.Val
        == (tIntType) 0x80000000 )
        Message( "Number too big.", xxError, '-' :Position );
      else
        UnaryExpression:gPtr->aIntConst.Val =
          - UnaryExpression:gPtr->aIntConst.Val;
      gPtr = UnaryExpression:gPtr;
    }
    else if ( UnaryExpression:gPtr->Kind == kaLongConst )
    {
      if (
        UnaryExpression:gPtr->aLongConst.Val
        == (tLongType) 0x8000000000000000LL )
        Message( "Number too big.", xxError, '-' :Position );
      else
        UnaryExpression:gPtr->aLongConst.Val =
          - UnaryExpression:gPtr->aLongConst.Val;
      gPtr = UnaryExpression:gPtr;
    }
    else if ( UnaryExpression:gPtr->Kind == kaFloatConst )
    {
      UnaryExpression:gPtr->aFloatConst.Val =
        - UnaryExpression:gPtr->aFloatConst.Val;
      gPtr = UnaryExpression:gPtr;
    }
    else if ( UnaryExpression:gPtr->Kind == kaDoubleConst )
    {
      UnaryExpression:gPtr->aDoubleConst.Val =
        - UnaryExpression:gPtr->aDoubleConst.Val;
      gPtr = UnaryExpression:gPtr;
    }
    else
      gPtr = maUnary( '-' :Position, OpUMINUS,
                    OpArithmetic, UnaryExpression:gPtr );
  };
} .

Not = {
  gPtr := {
    if ( UnaryExpression:gPtr->Kind == kaBooleanConst )
    {
      UnaryExpression:gPtr->aBooleanConst.Val =
        ( UnaryExpression:gPtr->aBooleanConst.Val ^ ( ( short ) 1 ) );
      gPtr = UnaryExpression:gPtr;
    }
    else
      gPtr = maUnary( '!' :Position, OpNot, OpLogic, UnaryExpression:gPtr );
  };
} .

```

```

/* Postfix expression */
PostfixExpression = {
  gPtr := Primary:gPtr;
} .

/* Primary */
ParExpression = {
  gPtr := Expression:gPtr;
} .

TrueConst = {
  gPtr := maBooleanConst( 'true':Position, LitTRUE );
} .

FalseConst = {
  gPtr := maBooleanConst( 'false':Position, LitFALSE );
} .

IntConst = {
  gPtr := { if ( integerLiteral:gInt.lGroup == Int )
            gPtr = maIntConst( integerLiteral:Position,
                               integerLiteral:gInt.lValue.B32 );
            else
              /* integerLiteral:gInt.lGroup == Long */
              gPtr = maLongConst( integerLiteral:Position,
                                   integerLiteral:gInt.lValue.B64 );
          };
  => {
    if ( integerLiteral:gInt.lOverflow )
      Message( "Number too big.", xxError, integerLiteral:Position );
  };
} .

FloatConst =
{
  gPtr := { if ( flPointLiteral:gFP.lGroup == Float )
            gPtr = maFloatConst( flPointLiteral:Position,
                                 flPointLiteral:gFP.lValue.B32 );
            else
              /* flPointLiteral:gInt.lGroup == Double */
              gPtr = maDoubleConst( flPointLiteral:Position,
                                    flPointLiteral:gFP.lValue.B64 );
          };
  => {
    if ( flPointLiteral:gFP.lOverflow )
      Message( "Number too big.", xxError, flPointLiteral:Position );
  };
} .

NullRefConst = {
  gPtr := maNullConst( 'null':Position );
} .

SimpleVariable = {
  gPtr := marValue( identifier:Position, identifier:gIdent );
} .

ArrayAccess = {
  gPtr := maIndexrValue( '[':Position, VarRef:gPtr, Expression:gPtr );
} .

```

```

FunctionCall = {
  gPtr := maFunctionCall( MethodInvocation:gMidPos, MethodInvocation:gMid,
                          MethodInvocation:gArgs );
} .

ArrayCreator = {
  gPtr := maArrayCreator( ArrayCreatorSpec:gPos,
                          ArrayCreatorSpec:gArrayType,
                          ArrayCreatorSpec:gDimSpec,
                          ArrayCreatorSpec:gNbInitDim );
} .

/* Array creator expression */
BasicACSpec = {
  gArrayType := maArrayType( PrimitiveType:gPtr );
  gDimSpec := maBasicDimSpec( Expression:gPtr );
  gNbInitDim := 1;
  gPos := PrimitiveType:gPos;
} .

SuperACSpec = {
  gArrayType := maArrayType( FullInitACSpec:gArrayType );
  gDimSpec := maDefDimSpec( FullInitACSpec:gDimSpec, Expression:gPtr );
  gNbInitDim := FullInitACSpec:gNbInitDim + 1;
  gPos := FullInitACSpec:gPos;
} .

BasicNDACSpec = {
  gArrayType := maArrayType( FullInitACSpec:gArrayType );
  gDimSpec := maUndefDimSpec( FullInitACSpec:gDimSpec );
  gNbInitDim := FullInitACSpec:gNbInitDim;
  gPos := FullInitACSpec:gPos;
} .

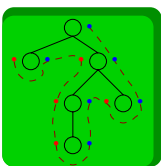
SuperNDACSpec = {
  gArrayType := maArrayType( PartInitACSpec:gArrayType );
  gDimSpec := maUndefDimSpec( PartInitACSpec:gDimSpec );
  gNbInitDim := PartInitACSpec:gNbInitDim;
  gPos := PartInitACSpec:gPos;
} .

END ASTBuilder

```

Annexe E

La spécification de l'analyseur sémantique



Le fichier `java--.cg` contient la spécification de l'analyseur sémantique. Il comporte 14 modules. Quatre modules sont de nature administrative : *GlobalDeclarations*, *AbstractGrammar*, *TargetCode* et *Output*. Les autres modules comportent tous des déclarations d'attributs et des règles pour leur évaluation. Ils sont conceptuellement divisés en deux catégories. Premièrement, cinq modules constituent le coeur de l'analyseur sémantique : *TypeLevelDecls*, *MethodLevelDecls*, *Environment*, *Identification* et *ExprType*. Deuxièmement, cinq modules sont spécifiques à la machine virtuelle *Java* : *Coercions*, *OpType*, *Rtn*, *FrameSize* et *WSSize*.

E.1 Modules administratifs

Le module nommé *GlobalDeclarations* contient les définitions de données utilisées par l'analyseur syntaxique lors de la construction de certains noeuds de l'arbre syntaxique abstrait. Ce module, propre à la construction de l'arbre syntaxique abstrait, comporte deux sections identifiées par les clauses `TREE` et `EXPORT`.

- La section identifiée par la clause `TREE` spécifie le nom des fichiers (`Tree.h` et `Tree.c`) qui contiennent le code *C* pour la construction de l'arbre syntaxique abstrait.
- Le code *C* sous la clause `EXPORT` est inséré intégralement dans le fichier `Tree.h` généré par l'outil `ast`.

Le module nommé *AbstractGrammar* contient la grammaire abstraite. Ce module, propre à la construction de l'arbre syntaxique abstrait, est divisé en quatre sections identifiées par les clauses `TREE`, `EVAL`, `PROPERTY` et `RULE`.

- La section identifiée par la clause `TREE` spécifie le nom des fichiers (`Tree.h` et `Tree.c`) qui contiennent le code *C* pour la construction de l'arbre syntaxique abstrait.
- La section identifiée par la clause `EVAL` spécifie le nom des fichiers (`Semantics.h` et `Semantics.c`) qui contiennent le code *C* pour l'analyseur sémantique.
- La clause `PROPERTY` ou la commande

indique que tous les attributs sont des attributs d'entrée lors de la création des noeuds de l'arbre syntaxique abstrait.

- Le code sous la clause **RULE** n'est pas du code *C*, mais la définition des symboles de la grammaire abstraite avec leurs attributs. Un symbole peut être vu comme un type de noeud de l'arbre syntaxique abstrait avec tous ses champs.

À partir de la définition de la grammaire abstraite, l'outil **ast** génère du code *C* qui contient la définition de plusieurs identificateurs. Ils sont déduits à partir des symboles qui apparaissent dans la section **RULE**. Par exemple, à partir du symbole *aRead*, les identificateurs suivants sont automatiquement définis :

- *kaRead* pour la sorte d'un noeud *aRead* ;
- *taRead* pour le type (au sens *C*) d'un noeud *aRead* ;
- *maRead* pour la fonction de création d'un noeud *aRead*.

Enfin, deux identificateurs sont définis par l'outil **ast** à partir du nom sous la clause **TREE** :

- *NoTree* pour le pointeur nul ;
- *tTree* pour le type générique des noeuds.

Le module nommé *TargetCode* contient du code *C* pour certaines parties de l'analyseur sémantique. Il est divisé en quatre sections identifiées par les clauses **EVAL**, **GLOBAL**, **BEGIN** et **CLOSE**.

- La section identifiée par la clause **EVAL** spécifie le nom des fichiers (**Semantics.h** et **Semantics.c**) qui contiennent le code *C* pour l'analyseur sémantique.
- Le code *C* sous la clause **GLOBAL** est inséré intégralement au niveau global dans le fichier **Semantics.c** généré par l'outil **ag**.
- Le code *C* sous la clause **BEGIN** est copié intégralement dans la fonction *BeginSemantics* de l'analyseur sémantique (**Semantics.c**).
- Le code *C* sous la clause **CLOSE** est copié intégralement dans la fonction *CloseSemantics* de l'analyseur sémantique (**Semantics.c**).

Le dernier module nommé *Output* contient les déclarations des attributs visibles au générateur de code. Il est divisé en trois sections identifiées par les clauses **EVAL**, **PROPERTY** et **DECLARE**.

- La section identifiée par la clause **EVAL** spécifie le nom des fichiers (**Semantics.h** et **Semantics.c**) qui contiennent le code *C* pour l'analyseur sémantique.
- La clause **PROPERTY** ou la commande

PROPERTY OUTPUT

indique que tous les attributs sont des attributs de sortie.

- La section identifiée par la clause **DECLARE** contient la déclaration de tous les attributs de sortie.

E.2 Modules pour l'évaluation des attributs

Les modules *TypeLevelDecls*, *MethodLevelDecls*, *Environment*, *Identification* et *ExprType* qui constituent le coeur de l'analyseur sémantique, ainsi que les modules *Coercions*, *OpType*,

Rtn, *FrameSize* et *WSSize* qui sont spécifiques à la machine virtuelle *Java*, ont la même structure.

Premièrement, tous ces modules comportent les sections suivantes :

- la section identifiée par la clause **EVAL** qui spécifie le nom des fichiers *C* (**Semantics.h** et **Semantics.c**) pour l'analyseur sémantique ;
- la section identifiée par la clause **DECLARE** qui contient la déclaration d'attributs locaux ;
- la section identifiée par la clause **RULE** qui contient des règles d'évaluation d'attributs.

Deuxièmement, certains modules, comme *TypeLevelDecls* et *MethodLevelDecls*, possèdent une section identifiée par la clause **TREE** qui spécifie le nom des fichiers *C* (**Tree.h** et **Tree.c**) pour la construction de l'arbre syntaxique abstrait. Le code *C* contenu dans les sections identifiées par les clauses **EXPORT**, **GLOBAL**, **BEGIN** et **CLOSE** est inséré intégralement dans ces fichiers à la condition qu'elles soient placées avant la section identifiée par la clause **EVAL**.


```
/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.
*/

/* java--.cg */

/* Description: Specification of abstract grammar and semantic rules for java--.
Author:      Daniel Cote.
Date:       February 2006.
*/
```

```

MODULE GlobalDeclarations      /* Declaration and creation of external data */
TREE Tree                      /* Generate code into Tree.h and Tree.c */
EXPORT                         /* Code to be inserted in Tree.h */
{
#include "Idents.h"             /* cocktail: Identifier (string) table */
#include "Position.h"           /* cocktail: Source positions handler */

#include "JavaFPAttributes.h"   /* java--: attributes for fp literals */
#include "JavaIntAttributes.h"  /* java--: attributes for integer literals */

#define NoFormals NoTree
#define NoFormal NoTree

#define ModNONE 0
#define ModPUBLIC 1
#define ModSTATIC 2           /* Modifiers */

#define OpLogic 1
#define OpCmp 2
#define OpEquiv 3
#define OpArithmetic 4
#define OpBitwiseLogic 5

#define OpLT 1
#define OpGT 2
#define OpLEQ 3
#define OpGEQ 4
#define OpEQ 5
#define OpNEQ 6
#define OpAND 7
#define OpOR 8
#define OpMUL 9
#define OpDIV 10
#define OpMOD 11
#define OpPLUS 12
#define OpMINUS 13
#define OpUMINUS 14
#define OpNot 15

#define LitFALSE 0
#define LitTRUE 1

extern tTree nVoidType,        /* Constant nodes declarations */
nIntType,
nLongType,
nFloatType,
nDoubleType,
nBooleanType,
nNullType;

extern tTree nErrorType,      /* Type system error nodes declarations */
nNaT,
nNoParameter;
}

```

```

GLOBAL                         /* Code to be inserted in Tree.c (level 0) */
{
tTree nVoidType,              /* Constant nodes declarations */
nIntType,
nLongType,
nFloatType,
nDoubleType,
nBooleanType,
nNullType;

tTree nErrorType,            /* Type system error nodes declaration */
nNaT,
nNoParameter;

static rbool bModuleInitialized = rfalse; /* Module initialization flag */
static rbool bModuleFinalized = rfalse; /* Module finalization flag */

BEGIN                         /* Code for initializing the tree handler */
{
if ( ! bModuleInitialized )
{
fprintf( stderr, "      Initializing the tree handler...\n" );

nVoidType = maVoidType();      /* Constant node creation */
nIntType = maIntType();
nLongType = maLongType();
nFloatType = maFloatType();
nDoubleType = maDoubleType();
nBooleanType = maBooleanType();
nNullType = maNullType();

nErrorType = maErrorType();    /* Type system error nodes creation */
nNaT = maNaT();
nNoParameter = maNoParameter();

bModuleInitialized = rtrue;
}
}

CLOSE                         /* Code for finalizing the tree handler */
{
if ( bModuleInitialized && ( ! bModuleFinalized ) )
{
fprintf( stderr, "      Finalizing the tree handler...\n" );

bModuleFinalized = rtrue;
}
}

END GlobalDeclarations

```

```

MODULE AbstractGrammar      /* Abstract grammar          */
TREE Tree
EVAL Semantics              /* Generate code into Semantics.[hc]          */
PROPERTY INPUT              /* All attributes are INPUT attributes          */
RULE                        /* Abstract syntax                              */
aCompilationUnit = [SourceFile: tStringRef OUT] /* Compilation unit                              */
                  aTypeDeclarations .
                  /* Type declarations          */
aTypeDeclarations = <
  aNoTypeDeclarations = .
  aTypeDeclaration = Next: aTypeDeclarations REV
                  [Modifiers: short]
                  [Name: tIdent]
                  [Pos: tPosition] <
    aClassDeclaration = aClassBodyDeclarations .
  > .
> .
                  /* Class body declarations          */
aClassBodyDeclarations = <
  aNoClassBodyDeclarations = .
  aClassBodyDeclaration = Next: aClassBodyDeclarations REV
                  [Modifiers: short] <
    aFieldDeclaration = aType aFieldDeclarators .
    aMethodDeclaration = aType [Name: tIdent] [Pos: tPosition]
                        aFormalParameters
                        aStatements .
  > .
> .
                  /* Type          */
aType = <
  aPrimitiveType = <
    aIntType = .
    aLongType = .
    aFloatType = .
    aDoubleType = .
    aBooleanType = .
  > .
  aReferenceType = <
    aArrayType = ElementType: aType OUT .
    aNullType = .
  > .
  aVoidType = .

  aTypeSysErrors = <
    aErrorType = .
    aNaT = .
  > .
> .

```

```

aFieldDeclarators = <
  aEndFieldDeclarators = .
  aFieldDeclarator = Next: aFieldDeclarators REV
                  [Name: tIdent] [Pos: tPosition] .
> .
                  /* Formal parameters          */
aFormalParameters = <
  aNoParameter = .
  aFormalParameter = Next: aFormalParameters REV
                  aType
                  [Name: tIdent]
                  [Pos: tPosition] .
> .
                  /* Statements          */
aStatements = <
  aNoStatement = .
  aStatement = Next: aStatements REV [Pos: tPosition] <
    aLocalVariableDeclaration = aType aVariableDeclarators .
    aInnerBlock = aStatements .
    aIfStatement = aExpression Then: aStatements <
      aIfThen = .
      aIfThenElse = Else: aStatements .
    > .
    aWhileStatement = aExpression aStatements .
    aReturnNoValueStatement = .
    aReturnValueStatement = aExpression .
    aAssignment = aLeftValue aExpression .
    aProcedureCall = [Name: tIdent] [NamePos: tPosition] aArguments .
    aRead = aLeftValue .
    aPrintln = .
    aPrintStringLiteral = [StrPos: tPosition OUT] [Str: tStringRef OUT] .
    aPrintValue = aExpression .
  > .
> .
                  /* Variable declarators          */
aVariableDeclarators = <
  aNoVariableDeclarators = .
  aVariableDeclarator = Next: aVariableDeclarators REV
                  [Name: tIdent] [Pos: tPosition] .
> .
                  /* Arguments          */
aArguments = <
  aNoArgument = [Pos: tPosition OUT] .
  aArgument = Next: aArguments REV
                  aExpression .
> .

```

```

aExpression = [Pos: tPosition] <                                /* Expression */
  aPrimary = <
    aBooleanConst = [Val: short OUT] .
    aNumericConst = <
      aIntConst = [Val: tIntType OUT] .
      aLongConst = [Val: tLongType OUT] .
      aFloatConst = [Val: tFloatType OUT] .
      aDoubleConst = [Val: tDoubleType OUT] .
    > .
    aNullConst = .
    aLeftValue = <
      alValue = [Name: tIdent] .
      aIndexlValue = aRightValue aExpression .
    > .
    aRightValue = <
      arValue = [Name: tIdent] .
      aIndexrValue = aRightValue aExpression .
    > .
    aFunctionCall = [Name: tIdent] aArguments .
    aArrayCreator = aArrayType aDimSpec [NbInitDim: short OUT] .
  > .
aBinary = [Op: short] [OpFamily: short]
  Left: aExpression Right: aExpression .

aUnary = [Op: short] [OpFamily: short] aExpression .
> .
/* Array creator expression */
aDimSpec = <
  aBasicDimSpec = aExpression .
  aMultiDimSpec = <
    aUndefDimSpec = aDimSpec .
    aDefDimSpec = aDimSpec aExpression .
  > .
> .
/* Coercion */
aCoercions = <
  aNoCoercion = .
  aIntToFloat = .
> .

END AbstractGrammar

```

```

MODULE TargetCode          /* Code for initializing and finalizing the
                             evaluator */
EVAL Semantics              /* Generate code into Semantics.[hc] */
GLOBAL                      /* Code to be inserted in Semantics.c (level 0) */
{
#include <stdio.h>           /* ISO C Standard: 4.9 input/output */

#include "StringM.h"         /* cocktail: String memory */
#include "Errors.h"          /* cocktail: Error handler */
#include "Position.h"        /* cocktail: Source positions handler */

static rbool bModuleInitialized = rfalse; /* Module initialization flag */
static rbool bModuleFinalized = rfalse; /* Module finalization flag */

static void AbstractCRError( char *module, char *node_type )
{
/* Description: No abstract node should ever be encountered in the AST, and if
one is actually encountered, it should give rise to a fatal
error and abort the compilation.
Author:      Daniel Cote.
Date:        March 2006.
Input:       module: a pointer to a module name.
             node_type: a pointer to a node type.
Output:      None.
*/

char msg[256];

sprintf( msg, "Abstract computation rule called :\n"
"  MODULE = %s, Abstract node type = \"%s\".\n\n",
module, node_type );
Message( msg, xxFatal, NoPosition );
}

static short max( short a, short b )
{
/* Description: Determine the maximum of two numbers.
Author:      Daniel Cote.
Date:        March 2006.
Input:       a: a number.
             b: a number
Output:      The maximum of the two numbers.
*/

return ( ( a > b ) ? a : b );
}
}

```

```

BEGIN                      /* Code for initializing the attribute evaluator*/
{
if ( ! bModuleInitialized )
{
/* Once code
fprintf( stderr, "    Initializing the attribute evaluator...\n" );
*/

bModuleInitialized = rtrue;
}
}

CLOSE                      /* Code for finalizing the attribute evaluator */
{
if ( bModuleInitialized && ( ! bModuleFinalized ) )
{
/* Once code
fprintf( stderr, "    Finalizing the attribute evaluator...\n" );
*/

bModuleFinalized = rtrue;
}
}

END TargetCode

```

```
MODULE Output          /* Declaration of evaluator attributes visible
                        to the code generator (all abstract grammar
                        attributes are also visible to the code
                        generator since they are INPUT attributes) */
EVAL Semantics          /* Generate code into Semantics.[hc]          */
PROPERTY OUTPUT         /* All attributes are OUTPUT attributes      */
DECLARE                 /* Declaration of attributes          */

  aCompilationUnit = [CurrentPkg: tsPackages SYN] .
  aFieldDeclarators = [Modifiers: short INH] aType INH .
  aMethodDeclaration = [FrameSize: short SYN] .
  aMethodDeclaration = [WSSize: short SYN] .
  aMethodDeclaration = [Rtn: short SYN] .
  aExpression = Type: aType SYN Co: aCoercions INH .
  aVariableDeclarators = [Offset: short INH] aType INH .
  aFormalParameters    = [Offset: short INH] .
  aProcedureCall aFunctionCall aValue arValue = [Descriptor: tSymTab SYN] .
  aStatements = [RtnType: tTree INH] .
  aBinary aUnary = OpType: aType SYN .
END Output
```



```

MODULE TypeLevelDecls          /* Gather information about types (classes and
                                interfaces) */

/* Classes are registered into the current package and verified for uniqueness.
   For each type, member is registered into the type and verified for
   uniqueness (depending on scope). */

TREE Tree

EXPORT                          /* Code to be inserted in Tree.h */
{
  #include "SymTab.h"          /* Generated by cocktail: symbol table handler */
}

EVAL Semantics                  /* Generate code into Semantics.[hc] */

DECLARE                         /* Declaration of attributes */
/* see MODULE Output: aCompilationUnit = [CurrentPkg: tsPackages SYN] */

  aTypeDeclarations = [TypeList: tsTypes THREAD OUT] .
  aClassDeclaration = [CurrentClass: tsTypes SYN OUT] .

  aClassBodyDeclarations = [MbrList: tsMembers THREAD OUT] .
  aFieldDeclarators = [MbrList: tsMembers THREAD OUT] .
/* see MODULE Output: aFieldDeclarators = [Modifiers: short INH] aType INH */

RULE                            /* Attributed grammar */

aCompilationUnit = {
  aTypeDeclarations:TypeListIn := nEndOfTypes;
  CurrentPkg := msPkgDesc( nEndOfPackages,
                          NoIdent,
                          aTypeDeclarations:TypeListOut,
                          nEndOfPackages );
} .

aTypeDeclarations = {
  TypeListOut := { AbstractCError( "SymbolTable", "aTypeDeclarations" ); };
} .

aTypeDeclaration = {
  TypeListOut, Next:TypeListIn := { AbstractCError( "SymbolTable",
                                                    "aTypeDeclaration" ); };
} .

aNoTypeDeclarations = {
  TypeListOut := TypeListIn;
} .

```

```

aClassDeclaration = {
  aClassBodyDeclarations:MbrListIn := nEndOfMembers;
  Next:TypeListIn := msClassDesc( TypeListIn,
                                  Name,
                                  Modifiers,
                                  aClassBodyDeclarations:MbrListOut );

  TypeListOut := Next:TypeListOut;

  CurrentClass := msClassDesc( nEndOfTypes,
                              Name,
                              Modifiers,
                              aClassBodyDeclarations:MbrListOut );

  CHECK ( ! ( ModSTATIC & Modifiers ) )
  => Message( "Modifier static not allowed for class.", xxError, Pos );

  CHECK ( ! IsDeclaredType( Name, TypeListIn ) )
  => Message( "Class name already defined.", xxError, Pos );
} .

aClassBodyDeclarations = {
  MbrListOut := { AbstractCError( "SymbolTable", "aClassBodyDeclarations" ); };
} .

aClassBodyDeclaration = {
  MbrListOut, Next:MbrListIn := { AbstractCError( "SymbolTable",
                                                  "aClassBodyDeclaration" ); };
} .

aNoClassBodyDeclarations = {
  MbrListOut := MbrListIn;
} .

aFieldDeclaration = {
  aFieldDeclarators:MbrListIn := MbrListIn;
  aFieldDeclarators:Modifiers := Modifiers;
  aFieldDeclarators:aType := aType;
  Next:MbrListIn := aFieldDeclarators:MbrListOut;

  MbrListOut := Next:MbrListOut;
} .

aFieldDeclarators = {
  MbrListOut := { AbstractCError( "SymbolTable", "aFieldDeclarators" ); };
} .

aEndFieldDeclarators = {
  MbrListOut := MbrListIn;
} .

```

```
aFieldDeclarator = {
  Next:Modifiers := Modifiers;
  Next:aType     := aType;
  Next:MbrListIn := msFieldDesc( MbrListIn, Name, Modifiers, aType );

  MbrListOut := Next:MbrListOut;

  CHECK ( ! IsDeclaredField( Name, MbrListIn ) )
  => Message( "Field: duplicate symbol declaration.", xxError, Pos );
}.

aMethodDeclaration = {
  Next:MbrListIn := msMethodDesc( MbrListIn,
                                Name,
                                Modifiers, aType, aFormalParameters );
  MbrListOut     := Next:MbrListOut;

  /* This check should be removed or modified if overloading is allowed */
  CHECK ( ! IsDeclaredMethod( Name, MbrListIn ) )
  => Message( "Method: duplicate symbol declaration.", xxError, Pos );
} .

END TypeLevelDecls
```

```

MODULE MethodLevelDecls      /* Gather information about methods */
/* Local variables are registered in the symbol table and their offset in
   the JVM method's frame is calculated. */

TREE Tree

EXPORT                       /* Code to be inserted in Tree.h */
{
#include "SymTab.h"          /* Generated by cocktail: symbol table handler */
}

EVAL Semantics               /* Generate code into Semantics.[hc] */

DECLARE                      /* Declaration of attributes */
aFormalParameters = [Offset: short THREAD OUT] .
/* see MODULE Output: aFormalParameters = [Offset: short INH] */
aVariableDeclarators = [Offset: short THREAD OUT] .
/* see MODULE Output: aVariableDeclarators = [Offset: short INH] */
aStatements = [Offset: short INH OUT] .

aFormalParameters = [VarList: tsVariables THREAD OUT] .
aVariableDeclarators = [VarList: tsVariables THREAD OUT] .
/* see MODULE Output: aVariableDeclarators = aType INH. */
aStatements = [VarList: tsVariables INH OUT] .

RULE                         /* Attributed grammar */

aMethodDeclaration = {
aFormalParameters:OffsetIn := ( ModSTATIC & Modifiers ) ? 0 \: 1;
aFormalParameters:Offset := ( ModSTATIC & Modifiers ) ? 0 \: 1;
aStatements:Offset := aFormalParameters:OffsetOut;

aFormalParameters:VarListIn := nEndOfVariables;
aStatements:VarList := aFormalParameters:VarListOut;
} .

aFormalParameters = {
/* Formal parameter list */
OffsetOut := { AbstractCError( "FrameLayout", "aFormalParameters" ); };

VarListOut := { AbstractCError( "SymbolTable", "aFormalParameters" ); };
} .

aNoParameter = {
OffsetOut := OffsetIn;

VarListOut := VarListIn;
} .

```

```

aFormalParameter = {
Next:OffsetIn := OffsetIn + TypeSize( aType );
Next:Offset := OffsetIn + TypeSize( aType );
OffsetOut := Next:OffsetOut;

Next:VarListIn := msVarDesc( VarListIn, Name, aType, OffsetIn );
VarListOut := Next:VarListOut;

CHECK ( ! IsDeclaredVariable( Name, VarListIn ) )
=> Message( "Parameter: duplicate symbol declaration.", xxError, Pos);
} .

/* Statements */
/* Normally, impossible case */
aStatement = {
Next:Offset, Next:VarList :=
{ AbstractCError( "SymbolTable", "aStatement" ); };
} .

aLocalVariableDeclaration = {
aVariableDeclarators:OffsetIn := Offset;
aVariableDeclarators:Offset := Offset;
Next:Offset := aVariableDeclarators:OffsetOut;

aVariableDeclarators:aType := aType;
aVariableDeclarators:VarListIn := VarList;
Next:VarList := aVariableDeclarators:VarListOut;
} .

aInnerBlock = {
aStatements:Offset := Offset;
Next:Offset := Offset;

aStatements:VarList := VarList;
Next:VarList := VarList;
} .

aIfStatement = {
/* Normally, impossible case */
Next:Offset, Then:Offset, Next:VarList, Then:VarList :=
{ AbstractCError( "SymbolTable", "aIfStatement" ); };
} .

aIfThen = {
Then:Offset := Offset;
Next:Offset := Offset;

Then:VarList := VarList;
Next:VarList := VarList;
} .

aIfThenElse = {
Then:Offset := Offset;
Else:Offset := Offset;
Next:Offset := Offset;

Then:VarList := VarList;
Else:VarList := VarList;
Next:VarList := VarList;
} .

```

```

aWhileStatement = {
  aStatements:Offset := Offset;
  Next:Offset       := Offset;

  aStatements:VarList := VarList;
  Next:VarList       := VarList;
} .

aReturnNoValueStatement = {
  Next:Offset := Offset;

  Next:VarList := VarList;
} .

aReturnValueStatement = {
  Next:Offset := Offset;

  Next:VarList := VarList;
} .

aAssignment = {
  Next:Offset := Offset;

  Next:VarList := VarList;
} .

aProcedureCall = {
  Next:Offset := Offset;

  Next:VarList := VarList;
} .

aRead = {
  Next:Offset := Offset;

  Next:VarList := VarList;
} .

aPrintln = {
  Next:Offset := Offset;

  Next:VarList := VarList;
} .

aPrintStringLiteral = {
  Next:Offset := Offset;

  Next:VarList := VarList;
} .

aPrintValue = {
  Next:Offset := Offset;

  Next:VarList := VarList;
} .

```

```

/* Variable declarators list */
/* Normally, impossible case */
aVariableDeclarators = {
  VarListOut, OffsetOut :=
    { AbstractCError( "MethodLevelDecls", "aVariableDeclarators" ); };
} .

aNoVariableDeclarators = {
  VarListOut := VarListIn;
  OffsetOut := OffsetIn;
} .

aVariableDeclarator = {
  Next:aType := aType;
  Next:VarListIn := msVarDesc( VarListIn, Name, aType, OffsetIn );
  VarListOut := Next:VarListOut;

  Next:OffsetIn := OffsetIn + TypeSize( aType );
  Next:Offset := Offset + TypeSize( aType );
  OffsetOut := Next:OffsetOut;

  CHECK ( ! IsDeclaredVariable( Name, VarListIn ) )
  => Message( "Local variable: duplicate symbol declaration.", xxError, Pos);
} .

END MethodLevelDecls

```

```

MODULE Environment          /* Distribute information gathered in the two
                             previous modules across the class's methods
                             and in turn to their statements and
                             expressions */
EVAL Semantics              /* Generate code into Semantics.[hc] */
DECLARE                    /* Declaration of attributes */
  aTypeDeclarations = [CurrentPkg: tsPackages INH OUT] .
  aClassBodyDeclarations = [Env: tsEnv INH OUT] .

  aStatements aExpression aArguments aDimSpec = [Env: tsEnv INH OUT] .
  aExpression aArguments aDimSpec = [VarList: tsVariables INH OUT] .

  aStatements aExpression aArguments aDimSpec = [Context: short INH OUT] .
/* see MODULE Output: aStatements = [RtnType: tTree INH] */
RULE                      /* Attributed grammar */

aCompilationUnit = {
  aTypeDeclarations:CurrentPkg := CurrentPkg;
} .

aTypeDeclaration = {
  Next:CurrentPkg := { AbstractCError( "Environment", "aTypeDeclaration" ) };
} .

aClassDeclaration = {
  Next:CurrentPkg := CurrentPkg;
  aClassBodyDeclarations:Env := msEnv( CurrentPkg, CurrentClass );
} .

/* Distribute across a type */
aClassBodyDeclaration = {
  Next:Env := { AbstractCError( "Environment", "aClassBodyDeclaration" ) };
} .

aFieldDeclaration = {
  Next:Env := Env;
} .

aMethodDeclaration = {
  Next:Env := Env;
  aStatements:Env := Env;

  aStatements:RtnType := aType;

  aStatements:Context := Modifiers;
} .

/* Distribute across a statement */
aStatement = {
  Next:Env, Next:RtnType, Next:Context :=
    { AbstractCError( "Environment", "aStatement" ) };
} .

```

```

aLocalVariableDeclaration = {
  Next:Env := Env;

  Next:RtnType := RtnType;

  Next:Context := Context;
} .

aInnerBlock = {
  Next:Env := Env;
  aStatements:Env := Env;

  Next:RtnType := RtnType;
  aStatements:RtnType := RtnType;

  aStatements:Context := Context;
  Next:Context := Context;
} .

aIfStatement = {
  Next:Env, Then:Env, aExpression:Env, aExpression:VarList, Next:RtnType,
  Then:RtnType, Next:Context, aExpression:Context, Then:Context :=
    { AbstractCError( "Environment", "aIfStatement" ) };
} .

aIfThen = {
  Next:Env := Env;
  Then:Env := Env;
  aExpression:Env := Env;

  aExpression:VarList := VarList;

  Next:RtnType := RtnType;
  Then:RtnType := RtnType;

  Next:Context := Context;
  aExpression:Context := Context;
  Then:Context := Context;
} .

aIfThenElse = {
  Next:Env := Env;
  Then:Env := Env;
  Else:Env := Env;
  aExpression:Env := Env;

  aExpression:VarList := VarList;

  Next:RtnType := RtnType;
  Then:RtnType := RtnType;
  Else:RtnType := RtnType;

  Next:Context := Context;
  aExpression:Context := Context;
  Then:Context := Context;
  Else:Context := Context;
} .

```

```
aWhileStatement = {
  Next:Env      := Env;
  aStatements:Env := Env;
  aExpression:Env := Env;

  aExpression:VarList := VarList;

  Next:RtnType      := RtnType;
  aStatements:RtnType := RtnType;

  Next:Context      := Context;
  aExpression:Context := Context;
  aStatements:Context := Context;
} .

aReturnNoValueStatement = {
  Next:Env := Env;

  Next:RtnType := RtnType;

  Next:Context := Context;
} .

aReturnValueStatement = {
  Next:Env      := Env;
  aExpression:Env := Env;

  aExpression:VarList := VarList;

  Next:RtnType := RtnType;

  Next:Context      := Context;
  aExpression:Context := Context;
} .

aAssignment = {
  Next:Env      := Env;
  aExpression:Env := Env;
  aLeftValue:Env := Env;

  aExpression:VarList := VarList;
  aLeftValue:VarList := VarList;

  Next:RtnType := RtnType;

  Next:Context      := Context;
  aLeftValue:Context := Context;
  aExpression:Context := Context;
} .
```

```
aProcedureCall = {
  Next:Env      := Env;
  aArguments:Env := Env;

  aArguments:VarList := VarList;

  Next:RtnType := RtnType;

  Next:Context      := Context;
  aArguments:Context := Context;
} .

aRead = {
  Next:Env      := Env;
  aLeftValue:Env := Env;

  aLeftValue:VarList := VarList;

  Next:RtnType := RtnType;

  Next:Context      := Context;
  aLeftValue:Context := Context;
} .

aPrintln = {
  Next:Env := Env;

  Next:RtnType := RtnType;

  Next:Context := Context;
} .

aPrintStringLiteral = {
  Next:Env := Env;

  Next:RtnType := RtnType;

  Next:Context := Context;
} .

aPrintValue = {
  Next:Env      := Env;
  aExpression:Env := Env;

  aExpression:VarList := VarList;

  Next:RtnType := RtnType;

  Next:Context      := Context;
  aExpression:Context := Context;
} .
```

```

/* Distribute across expressions */
aArgument = {
  Next:Env      := Env;
  aExpression:Env := Env;

  Next:VarList   := VarList;
  aExpression:VarList := VarList;

  Next:Context   := Context;
  aExpression:Context := Context;
} .

aIndexlValue = {
  aRightValue:Env := Env;
  aExpression:Env := Env;

  aRightValue:VarList := VarList;
  aExpression:VarList := VarList;

  aRightValue:Context := Context;
  aExpression:Context := Context;
} .

aIndexrValue = {
  aRightValue:Env := Env;
  aExpression:Env := Env;

  aRightValue:VarList := VarList;
  aExpression:VarList := VarList;

  aRightValue:Context := Context;
  aExpression:Context := Context;
} .

aFunctionCall = {
  aArguments:Env := Env;

  aArguments:VarList := VarList;

  aArguments:Context := Context;
} .

aArrayCreator = {
  aDimSpec:Env := Env;

  aDimSpec:VarList := VarList;

  aDimSpec:Context := Context;
} .

```

```

aBinary = {
  Left:Env := Env;
  Right:Env := Env;

  Left:VarList := VarList;
  Right:VarList := VarList;

  Left:Context := Context;
  Right:Context := Context;
} .

aUnary = {
  aExpression:Env := Env;

  aExpression:VarList := VarList;

  aExpression:Context := Context;
} .

aBasicDimSpec = {
  aExpression:Env := Env;

  aExpression:VarList := VarList;

  aExpression:Context := Context;
} .

aUndefDimSpec = {
  aDimSpec:Env := Env;

  aDimSpec:VarList := VarList;

  aDimSpec:Context := Context;
} .

aDefDimSpec = {
  aDimSpec:Env := Env;
  aExpression:Env := Env;

  aDimSpec:VarList := VarList;
  aExpression:VarList := VarList;

  aDimSpec:Context := Context;
  aExpression:Context := Context;
} .

END Environment

```

```

MODULE Identification      /* Identify references to symbolic names and
                           link them to their definition in the
                           environment */
EVAL Semantics             /* Generate code into Semantics.[hc] */
DECLARE                   /* Declaration of attributes */
/* see MODULE Output: aProcedureCall aFunctionCall aValue arValue
   = [Descriptor: tSymTab SYN] */
aArguments = [Formal: tTree INH OUT] [FormalType: tTree SYN OUT] .

/* The following attribute is only used to express dependencies between
   attribute "Descriptor" and "aExpression:Type" of each actual parameters of
   a method call. Method look-up requires the type of the actual parameters to
   be set before the search. VIRTUAL attributes are never actually allocated,
   nor are their computation rules actually generated or executed. Their
   dependencies, however, are taken on account. */
aArguments = [ Type: tTree VIRTUAL SYN] .

RULE                      /* Attributed grammar */
                           /* Code never generated because "Type" is a
                           VIRTUAL attribute. */
aArguments = { Type := nNaT; } .
aNoArgument = { Type := nNaT; } .
aArgument = { Type := DEP( aExpression:Type, Next:Type ); } .

aArguments = {
  FormalType := { AbstractCError( "Identification", "aArguments" ); } ;
} .

aNoArgument = { FormalType := nNaT; } .

aArgument = {
  Next:Formal := ( Formal != NoFormal ) ? GetNextFormal( Formal ) \: NoFormal;
  FormalType := ( Formal != NoFormal ) ? GetFormalType( Formal ) \: nNaT;
} .

aProcedureCall = {
  /* Method look-up requires the type of the actuals to be set */
  Descriptor :=
    DEP( IdentifyMethodRef( Name, aArguments, Env ), aArguments:Type );
  aArguments:Formal := ( Descriptor != NoDescriptor )
    ? GetMethodPrms( Descriptor ) \: NoFormal;

  CHECK ( Descriptor != NoDescriptor )
  => Message( "Method not declared.", xxError, NamePos )
  AND THEN
  CHECK ( ( ! ( Context & ModSTATIC ) ) || ( IsStaticMbr( Descriptor ) ) )
  => Message( "Object method call not allowed from static context.",
    xxError, NamePos )
  CHECK ( CheckParamNbr( aArguments, GetMethodPrms( Descriptor ) ) )
  => { /* Diagnostics already given, nothing to do. */ }
  AND THEN
  CHECK ( ChkPrmTypes( aArguments, GetMethodPrms( Descriptor ) ) )
  => { /* Diagnostics already given, nothing to do. */ }
} .

```

```

aFunctionCall = {
  /* Method look-up requires the type of the actuals to be set */
  Descriptor :=
    DEP( IdentifyMethodRef( Name, aArguments, Env ), aArguments:Type );
  aArguments:Formal := ( Descriptor != NoDescriptor )
    ? GetMethodPrms( Descriptor ) \: NoFormal;

  CHECK ( Descriptor != NoDescriptor )
  => Message( "Method not declared.", xxError, Pos )
  AND THEN
  CHECK ( ( ! ( Context & ModSTATIC ) ) || ( IsStaticMbr( Descriptor ) ) )
  => Message( "Object method call not allowed from static context.",
    xxError, Pos )
  AND THEN
  CHECK ( CheckParamNbr( aArguments, GetMethodPrms( Descriptor ) ) )
  => { /* Diagnostics already given, nothing to do. */ }
  AND THEN
  CHECK ( ChkPrmTypes( aArguments, GetMethodPrms( Descriptor ) ) )
  => { /* Diagnostics already given, nothing to do. */ }
} .

aValue = {
  Descriptor := IdentifySymbolRef( Name, Env, VarList );

  CHECK ( Descriptor != NoDescriptor )
  => Message( "Symbol not declared.", xxError, Pos )
  AND THEN
  CHECK ( Context & ModSTATIC )
  => { /* If false, no need to check for static membership. */ }
  CHECK ( IsVarDesc( Descriptor ) || IsStaticMbr( Descriptor ) )
  => Message( "Object member reference not allowed from static context.",
    xxError, Pos );
} .

arValue = {
  Descriptor := IdentifySymbolRef( Name, Env, VarList );

  CHECK ( Descriptor != NoDescriptor )
  => Message( "Symbol not declared.", xxError, Pos )
  AND THEN
  CHECK ( Context & ModSTATIC )
  => { /* If false, no need to check for static membership. */ }
  CHECK ( IsVarDesc( Descriptor ) || IsStaticMbr( Descriptor ) )
  => Message( "Object member reference not allowed from static context.",
    xxError, Pos );
} .

END Identification

```



```

MODULE ExprType          /* Determine the type of each expression and
                           perform type checking */

EVAL Semantics            /* Generate code into Semantics.[hc] */

DECLARE                  /* Declaration of attributes */
/* see MODULE Output: aExpression = Type: aType SYN OUT */

RULE                    /* Attributed grammar */

aExpression = {
    Type := { AbstractCError( "ExprType", "aExpression" ); };
} .

aPrimary = {
    Type := { AbstractCError( "ExprType", "aPrimary" ); };
} .

aBooleanConst = {
    Type := nBooleanType;
} .

aNumericConst = {
    Type := { AbstractCError( "ExprType", "aNumericConst" ); };
} .

aIntConst = {
    Type := nIntType;
} .

aLongConst = {
    Type := nLongType;
} .

aFloatConst = {
    Type := nFloatType;
} .

aDoubleConst = {
    Type := nDoubleType;
} .

aNullConst = {
    Type := nNullType;
} .

aLeftValue = {
    Type := { AbstractCError( "ExprType", "aLeftValue" ); };
} .

aIValue = {
    Type := ( Descriptor != NoDescriptor ) ? GetSymbolType( Descriptor ) \: nNaT;
} .

```

```

aIndexIValue = {
    Type := GetElementType( aRightValue:Type );

    CHECK ( Type->Kind != kaErrorType )
    => Message( "Invalid type for indexed reference.", xxError, Pos )
    AND THEN
    CHECK ( Type->Kind != kaNaT )
    => { /* Error has already been reported, nothing to do. */ }
    AND THEN
    CHECK ( aExpression:Type->Kind == kaIntType )
    => Message( "Indexing expression must be integer.",
               xxError, aExpression:Pos );
} .

aRightValue = {
    Type := { AbstractCError( "ExprType", "aRightValue" ); };
} .

aRValue = {
    Type := ( Descriptor != NoDescriptor ) ? GetSymbolType( Descriptor ) \: nNaT;
} .

aIndexRValue = {
    Type := GetElementType( aRightValue:Type );

    CHECK ( Type->Kind != kaErrorType )
    => Message( "Invalid type for indexed variable.", xxError, Pos )
    AND THEN
    CHECK ( Type->Kind != kaNaT )
    => { /* Error has already been reported, nothing to do. */ }
    AND THEN
    CHECK ( aExpression:Type->Kind == kaIntType )
    => Message( "Indexing expression must be integer.",
               xxError, aExpression:Pos );
} .

aFunctionCall = {
    Type := {
        Type = ( Descriptor != NoDescriptor )
            ? GetSymbolType( Descriptor ) \: nNaT;
        if ( Type->Kind == kaVoidType )
            Type = nErrorType;
    };

    CHECK ( Type->Kind != kaErrorType )
    => Message( "Invalid return type for method call.", xxError, Pos );
} .

aArrayCreator = {
    Type := aArrayType;
} .

```

```

aBinary = {
  Type := BinaryResultType( Left:Type, Right:Type, Op, OpFamily );

  CHECK ( Type->Kind != kaErrorType )
  => Message( "Incompatible operand types in binary expression.",
             xxError, Pos );
} .

aUnary = {
  Type := UnaryResultType( aExpression:Type, Op, OpFamily );

  CHECK ( Type->Kind != kaErrorType )
  => Message( "Invalid operand type in unary expression.", xxError, Pos );
} .

/* Type consistency checks */
aIfThen = {
  CHECK ( ( aExpression:Type->Kind != kaErrorType )
        && ( aExpression:Type->Kind != kaNaT ) )
  => { /* Error has already been reported, nothing to do. */ }
  AND THEN
  CHECK ( aExpression:Type->Kind == kaBooleanType )
  => Message( "Boolean expression required.",
             xxError, aExpression:Pos );
} .

aIfThenElse = {
  CHECK ( ( aExpression:Type->Kind != kaErrorType )
        && ( aExpression:Type->Kind != kaNaT ) )
  => { /* Error has already been reported, nothing to do. */ }
  AND THEN
  CHECK ( aExpression:Type->Kind == kaBooleanType )
  => Message( "Boolean expression required.",
             xxError, aExpression:Pos );
} .

aWhileStatement = {
  CHECK ( ( aExpression:Type->Kind != kaErrorType )
        && ( aExpression:Type->Kind != kaNaT ) )
  => { /* Error has already been reported, nothing to do. */ }
  AND THEN
  CHECK ( aExpression:Type->Kind == kaBooleanType )
  => Message( "Boolean expression required",
             xxError, aExpression:Pos );
} .

aReturnNoValueStatement = {
  CHECK ( RtnType->Kind == kaVoidType )
  => Message( "Missing expression in return statement.",
             xxError, Pos );
} .

```

```

aReturnValueStatement = {
  CHECK ( aExpression:Type->Kind != kaNaT )
  => { /* Error has already been reported, nothing to do. */ }
  AND THEN
  CHECK ( AssignmentCompatible( RtnType, aExpression:Type ) )
  => Message( "Expression type incompatible with method's return type.",
             xxError, aExpression:Pos );
} .

aAssignment = {
  CHECK ( ( aLeftValue:Type->Kind != kaNaT )
        && ( aLeftValue:Type->Kind != kaErrorType ) )
  => { /* Error has already been reported, nothing to do. */ }
  AND THEN
  CHECK ( ( aExpression:Type->Kind != kaNaT )
        && ( aExpression:Type->Kind != kaErrorType ) )
  => { /* Error has already been reported, nothing to do. */ }
  AND THEN
  CHECK ( AssignmentCompatible( aLeftValue:Type, aExpression:Type ) )
  => Message( "Expression type not compatible with target of assignment.",
             xxError, aExpression:Pos );
} .

aRead = {
  CHECK ( aLeftValue:Type->Kind != kaNaT )
  => { /* Error has already been reported, nothing to do. */ }
  AND THEN
  CHECK ( IsScalarType( aLeftValue:Type ) )
  => Message( "Invalid operand type in Read operation.",
             xxError, aLeftValue:Pos );
} .

aPrintValue = {
  CHECK ( aExpression:Type->Kind != kaNaT )
  => { /* Error has already been reported, nothing to do. */ }
  AND THEN
  CHECK ( IsScalarType( aExpression:Type ) )
  => Message( "Invalid operand type in Print operation.",
             xxError, aExpression:Pos );
} .

aArgument = {
  /* Argument <-> Formal CHECKs are done at identification */
} .

aBasicDimSpec = {
  CHECK ( aExpression:Type->Kind != kaErrorType )
  => Message( "Invalid indexing expression.", xxError, aExpression:Pos )
  AND THEN
  CHECK ( aExpression:Type->Kind != kaNaT )
  => { /* Error already reported. */ }
  AND THEN
  CHECK ( aExpression:Type->Kind == kaIntType )
  => Message( "Indexing expression must be integer.",
             xxError, aExpression:Pos );
} .

```

```
aDefDimSpec = {  
  CHECK ( aExpression:Type->Kind != kaErrorType )  
  => Message( "Invalid indexing expression.", xxError, aExpression:Pos )  
  AND THEN  
  CHECK ( aExpression:Type->Kind != kaNaT )  
  => { /* Error already reported. */ }  
  AND THEN  
  CHECK ( aExpression:Type->Kind == kaIntType )  
  => Message( "Indexing expression must be integer.",  
             xxError, aExpression:Pos );  
} .  
  
END ExprType
```

```

MODULE Coercions          /* Apply coercions in expressions */
EVAL Semantics            /* Generate code into Semantics.[hc] */
DECLARE                  /* Declaration of attributes */
/* see MODULE Output: aExpression = Co: aCoercions INH */
RULE                     /* Attributed grammar */

aIndexlValue = {
  aRightValue:Co := maNoCoercion();
  aExpression:Co := maNoCoercion();
} .

aIndexrValue = {
  aRightValue:Co := maNoCoercion();
  aExpression:Co := maNoCoercion();
} .

aBinary = {
  Left:Co := maNoCoercion();
  Right:Co := maNoCoercion();
} .

aUnary = {
  aExpression:Co := maNoCoercion();
} .

aIfStatement = {
  aExpression:Co := { AbstractCRError( "Coercions", "aIfStatement" ); };
} .

aIfThen = {
  aExpression:Co := maNoCoercion();
} .

aIfThenElse = {
  aExpression:Co := maNoCoercion();
} .

aWhileStatement = {
  aExpression:Co := maNoCoercion();
} .

aReturnValueStatement = {
  aExpression:Co := Coerce( RtnType, aExpression:Type );
} .

aAssignment = {
  aLeftValue:Co := maNoCoercion();
  aExpression:Co := Coerce( aLeftValue:Type, aExpression:Type );
} .

aRead = {
  aLeftValue:Co := maNoCoercion();
} .

```

```

aPrintValue = {
  aExpression:Co := maNoCoercion();
} .

aArgument = {
  aExpression:Co := Coerce( FormalType, aExpression:Type );
} .

aBasicDimSpec = {
  aExpression:Co := maNoCoercion();
} .

aDefDimSpec = {
  aExpression:Co := maNoCoercion();
} .

END Coercions

```

```
MODULE OpType          /* Calculate the type of the operator to apply
                        for binary and unary operators          */

/* Operators are polymorphic, but the JVM has typed mnemonics for all
   operations, the correct one must be used.                    */

EVAL Semantics          /* Generate code into Semantics.[hc]    */

DECLARE                /* Declaration of attributes             */
/* see MODULE Output: aBinary aUnary = [OpType: tTree SYN]      */

RULE                  /* Attributed grammar                   */

aBinary = {
  OpType := BinaryOpType( Left:Type, Right:Type );
} .

aUnary = {
  OpType := aExpression:Type;
} .

END OpType
```

```

MODULE Rtn
    /* Detect the return statements in a method
       (Built on data flow equations [Grune et al,
       pp. 253-260, 2002]) */

/* Basically, a bit map representing a set of two indicators "MayReturn",
   the method is capable of returning, and "MayNotReturn", the method may
   fail to return, is circulated around the statements once, according to the
   logic on p. 255. At the end, the set should indicate that a method cannot
   fail to return (MayNotReturn == false). */

EVAL Semantics
    /* Generate code into Semantics.[hc] */

EXPORT
    /* Code to be inserted in Semantics.h */
    {
        static short MayReturn = 0x0001;
        static short MayNotReturn = 0x0002;
    }

DECLARE
    /* Declaration of attributes */
    aStatements = [Rtn: short THREAD OUT] .
/* see MODULE Output: aMethodDeclaration = [Rtn: short SYN OUT] */

RULE
    /* Attributed grammar */

aMethodDeclaration = {
    aStatements:RtnIn := MayNotReturn;
    Rtn := aStatements:RtnOut;

    CHECK ( ( Rtn & MayNotReturn ) || ( Rtn & MayReturn ) )
    => Message( "Data flow equation bug.", xxFatal, Pos )
    AND THEN
    CHECK ( aType->Kind != kaVoidType )
    => { /* return generated automatically as needed */ }
    AND THEN
    CHECK ( ! ( Rtn & MayNotReturn ) )
    => Message( "Method missing return statement.", xxError, Pos );
} .

aStatements = {
    RtnOut := { AbstractCError( "Rtn", "aStatements" ); };

    RtnOut BEFORE Env;
} .

aStatement = {
    Next:RtnIn, RtnOut :=
    { AbstractCError( "Rtn", "aStatement" ); };
} .

aNoStatement = {
    RtnOut := RtnIn;
} .

aLocalVariableDeclaration = {
    Next:RtnIn := RtnIn;
    RtnOut := Next:RtnOut;
} .

```

```

aInnerBlock = {
    aStatements:RtnIn := RtnIn;
    Next:RtnIn := aStatements:RtnOut;
    RtnOut := Next:RtnOut;
} .

aIfStatement = {
    Next:RtnIn, RtnOut, Then:RtnIn :=
    { AbstractCError( "Rtn", "aIfStatement" ); };
} .

aIfThen = {
    Then:RtnIn := RtnIn;
    Next:RtnIn := RtnIn | Then:RtnOut;
    RtnOut := Next:RtnOut;
} .

aIfThenElse = {
    Then:RtnIn := RtnIn;
    Else:RtnIn := RtnIn;
    Next:RtnIn := Then:RtnOut | Else:RtnOut;
    RtnOut := Next:RtnOut;
} .

aWhileStatement = {
    aStatements:RtnIn := RtnIn;
    Next:RtnIn := RtnIn | aStatements:RtnOut;
    RtnOut := Next:RtnOut;
} .

aReturnNoValueStatement = {
    Next:RtnIn := MayReturn;
    RtnOut := Next:RtnOut;
} .

aReturnValueStatement = {
    Next:RtnIn := MayReturn;
    RtnOut := Next:RtnOut;
} .

aAssignment = {
    Next:RtnIn := RtnIn;
    RtnOut := Next:RtnOut;
} .

aProcedureCall = {
    Next:RtnIn := RtnIn;
    RtnOut := Next:RtnOut;
} .

aRead = {
    Next:RtnIn := RtnIn;
    RtnOut := Next:RtnOut;
} .

```

```
aPrintln = {  
  Next:RtnIn := RtnIn;  
  RtnOut := Next:RtnOut;  
} .
```

```
aPrintStringLiteral = {  
  Next:RtnIn := RtnIn;  
  RtnOut := Next:RtnOut;  
} .
```

```
aPrintValue = {  
  Next:RtnIn := RtnIn;  
  RtnOut := Next:RtnOut;  
} .
```

```
END Rtn
```

```

MODULE FrameSize          /* Calculate the frame size (local variables) */
EVAL Semantics             /* Generate code into Semantics.[hc]          */
DECLARE                   /* Declaration of attributes                  */
/* see MODULE Output: aMethodDeclaration = [FrameSize: short SYN OUT] */

    aStatements           = [MaxOffset: short THREAD OUT] .

RULE                      /* Attributed grammar                        */

aMethodDeclaration = {
    aStatements:MaxOffsetIn := aFormalParameters.OffsetOut;
    FrameSize := aStatements:MaxOffsetOut;
} .

aStatements = {
    /* Normally, impossible case */
    MaxOffsetOut := { AbstractCError( "FrameSize", "aStatements" ); };

    MaxOffsetOut BEFORE Env;
}.

aNoStatement = {
    MaxOffsetOut := MaxOffsetIn;
} .

aStatement = {
    /* Normally, impossible case */
    Next:MaxOffsetIn, MaxOffsetOut :=
    { AbstractCError( "FrameSize", "aStatement" ); };
} .

aLocalVariableDeclaration = {
    Next:MaxOffsetIn := max( aVariableDeclarators.OffsetOut, MaxOffsetIn );
    MaxOffsetOut := Next:MaxOffsetOut;
} .

aInnerBlock = {
    aStatements:MaxOffsetIn := MaxOffsetIn;
    Next:MaxOffsetIn := max( aStatements:MaxOffsetOut, MaxOffsetIn );
    MaxOffsetOut := Next:MaxOffsetOut;
} .

aIfStatement = {
    /* Normally, impossible case */
    Next:MaxOffsetIn, MaxOffsetOut, Then:MaxOffsetIn :=
    { AbstractCError( "FrameSize", "aIfStatement" ); };
} .

aIfThen = {
    Then:MaxOffsetIn := MaxOffsetIn;
    Next:MaxOffsetIn := max( Then:MaxOffsetOut, MaxOffsetIn );
    MaxOffsetOut := Next:MaxOffsetOut;
} .

```

```

aIfThenElse = {
    Then:MaxOffsetIn := MaxOffsetIn;
    Else:MaxOffsetIn := Then:MaxOffsetOut;
    Next:MaxOffsetIn := max( Else:MaxOffsetOut, MaxOffsetIn );
    MaxOffsetOut := Next:MaxOffsetOut;
} .

aWhileStatement = {
    aStatements:MaxOffsetIn := MaxOffsetIn;
    Next:MaxOffsetIn := max( aStatements:MaxOffsetOut, MaxOffsetIn );
    MaxOffsetOut := Next:MaxOffsetOut;
} .

aReturnNoValueStatement = {
    Next:MaxOffsetIn := MaxOffsetIn;
    MaxOffsetOut := Next:MaxOffsetOut;
} .

aReturnValueStatement = {
    Next:MaxOffsetIn := MaxOffsetIn;
    MaxOffsetOut := Next:MaxOffsetOut;
} .

aAssignment = {
    Next:MaxOffsetIn := MaxOffsetIn;
    MaxOffsetOut := Next:MaxOffsetOut;
} .

aProcedureCall = {
    Next:MaxOffsetIn := MaxOffsetIn;
    MaxOffsetOut := Next:MaxOffsetOut;
} .

aRead = {
    Next:MaxOffsetIn := MaxOffsetIn;
    MaxOffsetOut := Next:MaxOffsetOut;
} .

aPrintln = {
    Next:MaxOffsetIn := MaxOffsetIn;
    MaxOffsetOut := Next:MaxOffsetOut;
} .

aPrintStringLiteral = {
    Next:MaxOffsetIn := MaxOffsetIn;
    MaxOffsetOut := Next:MaxOffsetOut;
} .

aPrintValue = {
    Next:MaxOffsetIn := MaxOffsetIn;
    MaxOffsetOut := Next:MaxOffsetOut;
} .

END FrameSize

```



```

MODULE WSSize          /* Calculate the maximum size of the working
                        stack required by a method */

/* The attribute "StackDept" corresponds to the maximum stack depth required
by a statement or an expression. For an expression, the maximum stack depth
always accounts for the expression's result including any required coercion
and the stack requirements for the result's calculation. Therefore, a
method's WSSize, is the maximum of its statements' StackDepth.

Stack depth calculation is sensitive to lazy evaluation of Boolean
expressions, since lazy evaluation does not require stack space except
for primary Boolean expressions (like a function call) and non-Boolean
terms (as with comparison operators).

Stack depth requirements of a node depends on the code generated for that
node. Therefore, this module follows very closely the what is done in the
code generator (GenCode.pum).

EVAL Semantics          /* Generate code into Semantics.[hc]

DECLARE                /* Declaration of attributes
/* see MODULE Output: aMethodDeclaration = [WSSize: short SYN]

aStatements aVariableDeclarators
= [StackDepth: short SYN OUT] [MaxWSSize: short THREAD OUT] .
aExpression aDimSpec = [ReltSize: short SYN OUT]
                      [StackDepth: short SYN OUT] .
aArguments = [Base: short INH OUT] [StackDepth: short SYN OUT] .
aProcedureCall aFunctionCall = [CallOverhead: short SYN OUT] .

RULE                  /* Attributed grammar

aMethodDeclaration = {
  aStatements:MaxWSSizeIn := 0;
  WSSize                := aStatements:MaxWSSizeOut;
} .

aStatements = {
                      /* Normally, impossible case */
  StackDepth, MaxWSSizeOut := { AbstractCError( "WSSize", "aStatements" ); };
} .

aNoStatement = {
  StackDepth := 0;
  MaxWSSizeOut := MaxWSSizeIn;
} .

aStatement = {
                      /* Normally, impossible case */
  StackDepth, Next:MaxWSSizeIn, MaxWSSizeOut :=
    { AbstractCError( "WSSize", "aStatement" ); };
} .

```

```

aLocalVariableDeclaration = {
  StackDepth := 0;
  aVariableDeclarators:MaxWSSizeIn := MaxWSSizeIn;
  Next:MaxWSSizeIn                := aVariableDeclarators:MaxWSSizeOut;

  MaxWSSizeOut := Next:MaxWSSizeOut;
} .

aVariableDeclarators = {
                      /* Normally, impossible case */
  StackDepth, MaxWSSizeOut :=
    { AbstractCError( "WSSize", "aVariableDeclarators" ); };

  MaxWSSizeIn BEFORE VarListOut;
} .

aNoVariableDeclarators = {
  StackDepth := 0;
  MaxWSSizeOut := MaxWSSizeIn;
} .

aVariableDeclarator = {
  StackDepth := TypeSize( aType );
  Next:MaxWSSizeIn := max( MaxWSSizeIn, StackDepth );

  MaxWSSizeOut := Next:MaxWSSizeOut;
} .

aInnerBlock = {
  StackDepth := 0;
  aStatements:MaxWSSizeIn := MaxWSSizeIn;
  Next:MaxWSSizeIn        := aStatements:MaxWSSizeOut;

  MaxWSSizeOut := Next:MaxWSSizeOut;
} .

aIfStatement = {
                      /* Normally, impossible case */
  StackDepth, Next:MaxWSSizeIn, Then:MaxWSSizeIn, MaxWSSizeOut :=
    { AbstractCError( "WSSize", "aIfStatement" ); };
} .

aIfThen = {
  StackDepth := aExpression:StackDepth;
  Then:MaxWSSizeIn := max( MaxWSSizeIn, StackDepth );
  Next:MaxWSSizeIn := Then:MaxWSSizeOut;

  MaxWSSizeOut := Next:MaxWSSizeOut;
} .

aIfThenElse = {
  StackDepth := aExpression:StackDepth;
  Then:MaxWSSizeIn := max( MaxWSSizeIn, StackDepth );
  Else:MaxWSSizeIn := max( MaxWSSizeIn, StackDepth );
  Next:MaxWSSizeIn := max( Then:MaxWSSizeOut, Else:MaxWSSizeOut );

  MaxWSSizeOut := Next:MaxWSSizeOut;
} .

```

```

aWhileStatement = {
  StackDepth := aExpression.StackDepth;
  aStatements.MaxWSSizeIn := max( MaxWSSizeIn, StackDepth );
  Next.MaxWSSizeIn      := aStatements.MaxWSSizeOut;

  MaxWSSizeOut := Next.MaxWSSizeOut;
} .

aReturnNoValueStatement = {
  StackDepth := 0;
  Next.MaxWSSizeIn := MaxWSSizeIn;

  MaxWSSizeOut := Next.MaxWSSizeOut;
} .

aReturnValueStatement = {
  StackDepth := {
    {
      short wrapperSize = ( IsBoolean( aExpression.Type )
                            && ( ! IsPrimary( aExpression ) ) ) ? 1 \: 0;
      StackDepth = wrapperSize + aExpression.StackDepth;
    };
  };
  Next.MaxWSSizeIn := max( MaxWSSizeIn, StackDepth );

  MaxWSSizeOut := Next.MaxWSSizeOut;
} .

aAssignment = {
  StackDepth := {
    {
      short wrapperSize = ( IsBoolean( aExpression.Type )
                            && ( ! IsPrimary( aExpression ) ) ) ? 1 \: 0;

      StackDepth = max( aLeftValue.StackDepth,
                       aLeftValue.RsltSize + aExpression.StackDepth
                       + wrapperSize
                       );
    };
  };
  Next.MaxWSSizeIn := max( MaxWSSizeIn, StackDepth );

  MaxWSSizeOut := Next.MaxWSSizeOut;
} .

aProcedureCall = {
  CallOverhead := ( IsStaticMbr( Descriptor ) ) ? 0 \: 1;
  StackDepth := max( TypeSize( GetSymbolType( Descriptor ) ),
                    CallOverhead + aArguments.StackDepth );
  Next.MaxWSSizeIn := max( MaxWSSizeIn, StackDepth );

  MaxWSSizeOut := Next.MaxWSSizeOut;
} .

```

```

aRead = {
  StackDepth :=
    max( aLeftValue.StackDepth,
         aLeftValue.RsltSize + max( 2, TypeSize( aLeftValue.Type ) )
    );
  Next.MaxWSSizeIn := max( MaxWSSizeIn, StackDepth );

  MaxWSSizeOut := Next.MaxWSSizeOut;
} .

aPrintln = {
  StackDepth := 1;
  Next.MaxWSSizeIn := max( MaxWSSizeIn, StackDepth );

  MaxWSSizeOut := Next.MaxWSSizeOut;
} .

aPrintStringLiteral = {
  StackDepth := 2;
  Next.MaxWSSizeIn := max( MaxWSSizeIn, StackDepth );

  MaxWSSizeOut := Next.MaxWSSizeOut;
} .

aPrintValue = {
  StackDepth := {
    {
      short wrapperSize = ( IsBoolean( aExpression.Type )
                            && ( ! IsPrimary( aExpression ) ) ) ? 1 \: 0;
      StackDepth = 1 + wrapperSize + aExpression.StackDepth;
    };
  };
  Next.MaxWSSizeIn := max( MaxWSSizeIn, StackDepth );

  MaxWSSizeOut := Next.MaxWSSizeOut;
} .

aExpression = {
  RsltSize, StackDepth := { AbstractCError( "WSSize", "aExpression" ); };
} .

aPrimary = {
  RsltSize, StackDepth := { AbstractCError( "WSSize", "aPrimary" ); };
} .

aBooleanConst = {
  RsltSize := CoercedTypeSize( Type, Co );
  StackDepth := RsltSize;
} .

aNumericConst = {
  RsltSize, StackDepth := { AbstractCError( "WSSize", "aNumericConst" ); };
} .

```

```

aIntConst = {
  RsltSize := CoercedTypeSize( Type, Co );
  StackDepth := RsltSize;
} .

aLongConst = {
  RsltSize := CoercedTypeSize( Type, Co );
  StackDepth := RsltSize;
} .

aFloatConst = {
  RsltSize := CoercedTypeSize( Type, Co );
  StackDepth := RsltSize;
} .

aDoubleConst = {
  RsltSize := CoercedTypeSize( Type, Co );
  StackDepth := RsltSize;
} .

aNullConst = {
  RsltSize := TypeSize( Type );
  StackDepth := RsltSize;
} .

aLeftValue = {
  /* Normally, impossible case */
  RsltSize, StackDepth := { AbstractCError( "WSSize", "aLeftValue" ); };
} .

aIValue = {
  RsltSize := 0;
  StackDepth := RsltSize;
} .

aIndexIValue = {
  RsltSize := aRightValue:RsltSize + aExpression:RsltSize;
  StackDepth :=
    max( RsltSize,
          max( aRightValue:StackDepth,
                aRightValue:RsltSize + aExpression:StackDepth ) );
} .

aRightValue = {
  /* Normally, impossible case */
  RsltSize, StackDepth := { AbstractCError( "WSSize", "aRightValue" ); };
} .

arValue = {
  RsltSize := CoercedTypeSize( Type, Co );
  StackDepth := RsltSize;
} .

```

```

aIndexrValue = {
  RsltSize := CoercedTypeSize( Type, Co );
  StackDepth :=
    max( RsltSize,
          max( aRightValue:StackDepth,
                aRightValue:RsltSize + aExpression:StackDepth ) );
} .

aFunctionCall = {
  CallOverhead := ( IsStaticMbr( Descriptor ) ) ? 0 \: 1;
  RsltSize := CoercedTypeSize( Type, Co );
  StackDepth := max( RsltSize, CallOverhead + aArguments:StackDepth );
} .

aArrayCreator = {
  RsltSize := CoercedTypeSize( Type, Co );
  StackDepth := max( RsltSize, aDimSpec:StackDepth );
} .

aBinary = {
  RsltSize := IsBoolean( Type ) ? 0 \: CoercedTypeSize( Type, Co );
  StackDepth :=
    IsBoolean( OpType )
      ? max( Left:StackDepth, Right:StackDepth )
      \: max( RsltSize,
              max( Left:StackDepth,
                    Left:RsltSize + Right:StackDepth ) );
} .

aUnary = {
  RsltSize := IsBoolean( OpType ) ? 0 \: CoercedTypeSize( Type, Co );
  StackDepth := max( RsltSize, aExpression:StackDepth );
} .

aFunctionCall = {
  aArguments:Base := 0;
} .

aProcedureCall = {
  aArguments:Base := 0;
} .

aArguments = {
  /* Normally, impossible case */
  StackDepth := { AbstractCError( "WSSize", "aArguments" ); };
} .

aNoArgument = {
  StackDepth := Base;
} .

```

```
aArgument = {
  Next:Base := Base + aExpression:RsLtSize;
  StackDepth := {
    {
      short wrapperSize = (      IsBoolean( aExpression:Type )
                             && ( ! IsPrimary( aExpression ) ) ) ? 1 \: 0;
      StackDepth = max( Next:StackDepth,
                       ( Base + wrapperSize + aExpression:StackDepth ) );
    };
  };
} .

aDimSpec = {
  RsLtSize, StackDepth := { AbstractCError( "WSSize", "aDimSpec" ); };
} .

/* aExpression:StackDepth already includes aExpression:RsLtSize */
aBasicDimSpec = {
  RsLtSize := aExpression:RsLtSize;
  StackDepth := max( RsLtSize, aExpression:StackDepth );
} .

aMultiDimSpec = {
  RsLtSize, StackDepth := { AbstractCError( "WSSize", "aMultiDimSpec" ); };
} .

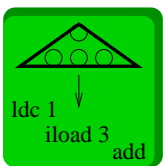
aUndefDimSpec = {
  RsLtSize := aDimSpec:RsLtSize;
  StackDepth := aDimSpec:StackDepth;
} .

/* aExpression:StackDepth already includes aExpression:RsLtSize */
aDefDimSpec = {
  RsLtSize := aDimSpec:RsLtSize + aExpression:RsLtSize;
  StackDepth :=
    max( RsLtSize, aDimSpec:RsLtSize + aExpression:StackDepth );
} .

END WSSize
```

Annexe F

Les procédures de génération de code



Le fichier `GenCode.pum` contient les procédures de génération de code de programmes écrits en *Java* –. Le code produit est un programme en langage d’assemblage *JasminXT* pouvant être traduit en *bytecode* pour une exécution éventuelle sur une machine virtuelle *Java*. Il est divisé en six sections identifiées par les clauses `TRAFO`, `TREE`, `GLOBAL`, `BEGIN` et `CLOSE` ainsi que par un

ensemble de procédures de génération de code.

- La section identifiée par la clause `TRAFO` spécifie le nom des fichiers qui contiennent le code *C* du générateur de code (`GenCode.h` et `GenCode.c`).
- La section identifiée par la clause `TREE` spécifie le nom des fichiers (`Tree.h`, `Tree.TS`, `SymTab.h` et `SymTab.TS`) qui contiennent les définitions des structures de données manipulées par le générateur de code. Les fichiers `Tree.TS` et `SymTab.TS` doivent donc être préalablement générés par l’outil `ast` (l’option `-4`).
- Le code *C* sous la clause `GLOBAL` est inséré intégralement au niveau global dans le fichier `GenCode.c` généré par l’outil `puma`. Les fonctions statiques *C*, *ResetLbl* et *GetNextLbl*, sont utilisées lors de la génération d’étiquettes.
- Le code *C* sous la clause `BEGIN` est inséré intégralement dans la fonction *BeginGenCode* du générateur de code (`GenCode.c`).
- Le code *C* sous la clause `CLOSE` est inséré intégralement dans la fonction *CloseGenCode* du générateur de code (`GenCode.c`).

La procédure *GenCode* est celle appelée par le programme principal. Elle démarre le processus de génération de code. Toutes les autres procédures sont des procédures auxiliaires. Voici la liste des procédures :

- *GenCode* – procédure de démarrage ;
- *GenTypes* – génération de code pour un type (classe ou interface) ;
- *CodeFields* – génération de code pour les champs ;
- *Code_clinit* – génération de code pour la méthode d’initialisation des champs statiques ;
- *Code_init* – génération de code pour le constructeur par défaut ;
- *Code_main* – génération de code pour l’appel de la méthode statique `main` ;
- *CodeMethods* – génération de code pour les méthodes ;
- *CodeStatements* – génération de code pour les énoncés ;
- *CodeLeftValueStore* – génération de code pour le stockage d’une donnée ;

- *CodeExpression* – génération de code pour les expressions non booléennes et booléennes primaires ;
- *CodeLazyExpr* – génération de code pour les expressions booléennes ;
- *CodeBinBranchOnCond* – génération de code pour les branchements positifs ;
- *CodeBinBranchOnNotCond* – génération de code pour les branchements négatifs ;
- *CodeVarDirectives* – génération de code pour les variables et les paramètres formels.

```

/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.

/* GenCode.pum
/* Description: Procedures for code generation
Author: Daniel Cote.
Date: June 2006.

TRAFO GenCode /* Generate code into GenCode.h and GenCode.c */
TREE Tree SymTab /* Use the definitions from Tree.h, Tree.TS,
Symtab.h and Symtab.TS */

GLOBAL
{
#include <stdio.h> /* Code to be inserted in GenCode.c (level 0) */
#include <string.h> /* ISO C Standard: 4.9 input/output */
/* ISO C Standard: 4.11 string handling */

#include "StringM.h" /* cocktail: String memory */
#include "Idents.h" /* cocktail: Identifier (string) table */
#include "Errors.h" /* cocktail: Error handler */
#include "Position.h" /* cocktail: Source positions handler */

#include "Semantics.h" /* Generated by cocktail: attribute evaluator */
#include "TypeSys.h" /* Generated by cocktail: type system */

static rbool bModuleInitialized = rfalse; /* Module initialization flag */
static rbool bModuleFinalized = rfalse; /* Module finalization flag */

static FILE *fp = NULL; /* Current output file for intermediate code */
static char *srcFile = NULL; /* java-- source code file name (.mjv) */
static char *strPkgName = NULL; /* Current package name */
static char strTypeName[256]; /* Current class name */
static char fileName[256]; /* Current class jasmin file name (.j) */

static errMsg[256]; /* Error message */

```

```

/* Label management */
static short CurrentLbl = 1; /* current label number */
static short NoLbl = 0; /* no label */

static void ResetLbl( void )
{
/* Description: Reset the current label number.
Author: Daniel Cote.
Date: June 2006.
Input: None.
Output: None. */
CurrentLbl = 1;
}

static short GetNextLbl( void )
{
/* Description: Get the next label number.
Author: Daniel Cote.
Date: June 2006.
Input: None.
Output: The next label number. */

return CurrentLbl++;
}

BEGIN
{ /* Code for initializing the code generator */
if ( ! bModuleInitialized )
{ /* Once code
fprintf( stderr, " Initializing the code generator...\n" );

bModuleInitialized = rtrue;
}
}
CLOSE
{ /* Code for finalizing the code generator */
if ( bModuleInitialized && ( ! bModuleFinalized ) )
{ /* Once code
fprintf( stderr, " Finalizing the code generator...\n" );

bModuleFinalized = rtrue;
}
}

```

```

/* All procedures follow the same structure and are made-up of three sections:
  1) A pre-conditions section (optional);
  2) A main section (mandatory);
  3) A unprocessed node traps section (optional).

```

Sections (1) and (3) are optional and are mainly used to express conditions that SHOULD NEVER OCCUR during code generation.

- When patterns in these sections are triggered, they issue a fatal error message, and the code generation process is ABORTED.
- Such fatal error will occur whenever there is a problem in the AST building process (java--psr) or an internal error in the code generator itself (improper use of a function, or missing code generation pattern).

The section (2) is the only one responsible for real code generation.

- For a code generation run to be successful, ONLY patterns in the main section of procedures should be triggered.
- Main section patterns should cover all possible cases of legitimate code generation. If one such pattern is forgotten, it will be trapped by a section 3 pattern, provided it is present in the AST. */

```

/* Main procedure -- GenCode:
 * - Code generation for the compilation unit from the root of the abstract
 *   syntax tree (TreeRoot of type aCompilationUnit). */
PUBLIC PROCEDURE GenCode( aCompilationUnit )

/* ----- Pre-conditions section ----- */
NIL := Message( "GenCode( NoTree ), invalid call.", xxFatal, NoPosition ); .

/* ----- Main section ----- */
aCompilationUnit(
  SourceFile := SrcFile,
  CurrentPkg := pkg:sPkgDesc( Name := Name ),
  aTypeDeclarations := typeList:aTypeDeclarations( .. )
) :-

  IF ( SrcFile != NoString ) THEN srcFile = StGetCStr( SrcFile ); END
  IF ( Name != NoIdent ) THEN strPkgName = GetCStr( Name ); END

  GenTypes( typeList );
  .

/* ----- Unprocessed node traps section ----- */
:-
  Message( "GenCode( Unknown_node_type ), invalid call.", xxFatal, NoPosition );
  .

```



```

/* Auxiliary procedure -- GenTypes:
 * - Code generation for types (classes and interfaces). */

PROCEDURE GenTypes( aTypeDeclarations )

/* ----- Pre-conditions section ----- */

NIL :- Message( "GenTypes( NoTree ), invalid call.", xxFatal, NoPosition ); .

/* ----- Main section ----- */

clsDecl:aClassDeclaration(
  Modifiers := Modifiers,
  Name := Name,
  aClassBodyDeclarations := classBody:aClassBodyDeclarations( .. ),
  Next := Next:aTypeDeclarations( .. )
) :-

IF ( strPkgName != NULL ) THEN
  sprintf( strTypeName, "%s/%s", strPkgName, GetCStr( Name ) );
ELSE
  strcpy( strTypeName, GetCStr( Name ) );
END
strcpy( fileName, strTypeName );
strcat( fileName, ".j" );
fp = fopen( fileName, "w" );
IF ( fp == NULL ) THEN
  sprintf( errMsg, "GenTypes(), unable to open file %s.\n", fileName );
  Message( errMsg, xxFatal, NoPosition );
END

fprintf( fp, "*****\n" );
fprintf( fp, "; Class %s.\n", strTypeName );
fprintf( fp, "*****\n" );
fprintf( fp, "\n" );
IF ( srcFile != NULL ) THEN
  fprintf( fp, "\t.source\t%s\n", srcFile );
END
fprintf( fp, "\t.class\t%s %s\n", EncodeModifiers( Modifiers ), strTypeName );
fprintf( fp, "\t.super\tjava/lang/Object\n" );
fprintf( fp, "\n" );

fprintf( fp, "\n" );
fprintf( fp, "; Fields.\n" );
fprintf( fp, "\n" );
CodeFields( classBody );

fprintf( fp, "\n" );
fprintf( fp, "; Class infrastructure.\n" );
fprintf( fp, "\n" );
Code_clinit( clsDecl );
Code_init( clsDecl );
Code_main( clsDecl );

fprintf( fp, "\n" );
fprintf( fp, "; Methods.\n" );
fprintf( fp, "\n" );
CodeMethods( classBody );

```

```

fp = fclose( fp );

GenTypes( Next );
.

aNoTypeDeclarations( .. ) :- /* end of type declarations */ .

/* ----- Unprocessed node traps section ----- */

n : aTypeDeclarations( .. )
:-
  sprintf( errMsg, "GenTypes() : Unprocessed node of type \"%s\".\n",
    Tree_NodeName[n->Kind] );
  Message( errMsg, xxFatal, NoPosition );
.

```

```

/* Auxiliary procedure -- CodeFields:
 * - Code generation for fields.
 */

PROCEDURE CodeFields( [aClassBodyDeclarations aFieldDeclarators] )

/* ----- Pre-conditions section ----- */

NIL :- Message( "CodeFields( NoTree ), invalid call.", xxFatal, NoPosition ); .

/* ----- Main section ----- */

aFieldDeclaration(
  Modifiers := Modifiers,
  aType := aType( .. ),
  aFieldDeclarators := nameList:aFieldDeclarators( .. ),
  Next := Next:aClassBodyDeclarations( .. )
)
:-
CodeFields( nameList );

CodeFields( Next );
.

aMethodDeclaration(
  Next := Next:aClassBodyDeclarations( .. )
)
:-
CodeFields( Next );
.

aNoClassBodyDeclarations( .. )
:- /* End of class body declarations. */
.

aFieldDeclarator(
  Modifiers := Modifiers,
  Name := Name,
  aType := fieldType:aType( .. ),
  Next := Next:aFieldDeclarators( .. )
)
:-
IF ( Modifiers != ModNONE ) THEN
  fprintf( fp, "\t.field\t%s '%s' %s\n", EncodeModifiers( Modifiers ),
          GetCStr( Name ),
          EncodeType( fieldType ) );

ELSE
  fprintf( fp, "\t.field\t'%s' %s\n", GetCStr( Name ),
          EncodeType( fieldType ) );

END

CodeFields( Next );
.

aEndFieldDeclarators( .. ) :- /* end of field declarators */ .

```

```

/* ----- Unprocessed node traps section ----- */

n : aClassBodyDeclarations( .. )
:-
sprintf( errMsg, "CodeFields() : Unprocessed node of type \"%s\"\n",
          Tree_NodeName[n->Kind] );
Message( errMsg, xxFatal, NoPosition );
.

n : aFieldDeclarators( .. )
:-
sprintf( errMsg, "CodeFields() : Unprocessed node of type \"%s\"\n",
          Tree_NodeName[n->Kind] );
Message( errMsg, xxFatal, NoPosition );
.

:-
Message( "CodeFields( Unknown_node_type ), invalid call.",
          xxFatal, NoPosition );
.

```

```

/* Auxiliary procedure -- Code_clinit:
 * - Code generation for initialization of static fields.          */
PROCEDURE Code_clinit( [aClassDeclaration aClassBodyDeclarations
                      aFieldDeclarators] )

/* ----- Pre-conditions section ----- */
NIL :- Message( "Code_clinit( NoTree ), invalid call.", xxFatal, NoPosition ); .

/* ----- Main section ----- */
aClassDeclaration(
  aClassBodyDeclarations := classBody:aClassBodyDeclarations( .. )
)
:-
  fprintf( fp, "\n" );
  fprintf( fp, "\t.method\tstatic public <clinit>()V\n" );
  fprintf( fp, "\t.limit\tlocals 0\n" );
  fprintf( fp, "\t.limit\tstack 2\n" );
  fprintf( fp, "\n" );

  fprintf( fp, "\t;\n" );
  fprintf( fp, "\t; Static fields initialisers.\n" );
  fprintf( fp, "\t;\n" );
  Code_clinit( classBody );

  fprintf( fp, "\treturn\n" );
  fprintf( fp, "\n" );
  fprintf( fp, "\t.end\tmethod ; <clinit>()V\n" );
  .

aFieldDeclaration(
  Modifiers := Modifiers,
  aFieldDeclarators := nameList:aFieldDeclarators( .. ),
  Next := Next:aClassBodyDeclarations( .. )
)
:-
  IF ( Modifiers & ModSTATIC ) THEN
    Code_clinit( nameList );
  END

  Code_clinit( Next );
  .

aNoClassBodyDeclarations( .. ) :- /* end of class body declarations */ .

aClassBodyDeclaration(
  Next := Next:aClassBodyDeclarations( .. )
)
:-
  Code_clinit( Next );
  .

```

```

aFieldDeclarator(
  Name := Name,
  aType := fieldType:aType( .. ),
  Next := Next:aFieldDeclarators( .. )
)
:-
  fprintf( fp, "\t%s\n", PushDefaultValue( fieldType ) );
  fprintf( fp, "\tputstatic\t%s/%s %s\n", strTypeName, GetCStr( Name ),
           EncodeType( fieldType ) );

  Code_clinit( Next );
  .

aEndFieldDeclarators( .. ) :- /* end of field declarators */ .

/* ----- Unprocessed node traps section ----- */
n : aClassBodyDeclarations( .. )
:-
  sprintf( errMsg, "Code_clinit() : Unprocessed node of type \"%s\"\n",
           Tree_NodeName[n->Kind] );
  Message( errMsg, xxFatal, NoPosition );
  .

n : aFieldDeclarators( .. )
:-
  sprintf( errMsg, "Code_clinit() : Unprocessed node of type \"%s\"\n",
           Tree_NodeName[n->Kind] );
  Message( errMsg, xxFatal, NoPosition );
  .

:-
  Message( "Code_clinit( Unknown_node_type ), invalid call.",
          xxFatal, NoPosition );
  .

```

```

/* Auxiliary procedure -- Code init:
 * - Code generation for the default constructor. */

PROCEDURE Code_init( [aClassDeclaration aClassBodyDeclarations
                    aFieldDeclarators] )

/* ----- Pre-conditions section ----- */
NIL :- Message( "Code_init( NoTree ), invalid call.", xxFatal, NoPosition ); .

/* ----- Main section ----- */
aClassDeclaration(
  aClassBodyDeclarations := classBody:aClassBodyDeclarations( .. )
)
:-
  fprintf( fp, "\n" );
  fprintf( fp, "\t.method\tpublic <init>()V\n" );
  fprintf( fp, "\t.limit\tlocals 1\n" );
  fprintf( fp, "\t.limit\tstack 2\n" );
  fprintf( fp, "\n" );

  fprintf( fp, "\taload 0\n" );
  fprintf( fp, "\t.invokevirtual\tjava/lang/Object/<init>()V\n" );

  Code_init( classBody );

  fprintf( fp, "\treturn\n" );
  fprintf( fp, "\n" );
  fprintf( fp, "\t.end\tmethod ; <init>()V\n" );
  .

aFieldDeclaration(
  Modifiers := Modifiers,
  aFieldDeclarators := nameList:aFieldDeclarators( .. ),
  Next := Next:aClassBodyDeclarations( .. )
)
:-
  IF ( ! ( Modifiers & ModSTATIC ) ) THEN
    Code_init( nameList );
  END

  Code_init( Next );
  .

aNoClassBodyDeclarations( .. ) :- /* end of class body declarations */ .

aClassBodyDeclaration(
  Next := Next:aClassBodyDeclarations( .. )
)
:-
  Code_init( Next );
  .

```

```

aFieldDeclarator(
  Name := Name,
  aType := fieldType:aType( .. ),
  Next := Next:aFieldDeclarators( .. )
)
:-
  fprintf( fp, "\taload 0\n", "" );
  fprintf( fp, "\t%s\n", PushDefaultValue( fieldType ) );
  fprintf( fp, "\tputfield\t%s/%s %s\n", strTypeName, GetCStr( Name ),
            EncodeType( fieldType ) );

  Code_init( Next );
  .

aEndFieldDeclarators( .. ) :- /* end of field declarators */ .

/* ----- Unprocessed node traps section ----- */

n : aClassBodyDeclarations( .. )
:-
  sprintf( errMsg, "Code_init() : Unprocessed node of type \"%s\"\n",
            Tree_NodeName[n->Kind] );
  Message( errMsg, xxFatal, NoPosition );
  .

n : aFieldDeclarators( .. )
:-
  sprintf( errMsg, "Code_init() : Unprocessed node of type \"%s\"\n",
            Tree_NodeName[n->Kind] );
  Message( errMsg, xxFatal, NoPosition );
  .

:-
  Message( "Code_init( Unknown_node_type ), invalid call.",
            xxFatal, NoPosition );
  .

```

```
/* Auxiliary procedure -- Code_main:
 * - Code generation for invocation the static method main.          */
PROCEDURE Code_main( aClassDeclaration )

/* ----- Pre-conditions section ----- */

NIL :- Message( "Code_main( NoTree ), invalid call.", xxFatal, NoPosition ); .

/* ----- Main section ----- */

aClassDeclaration( .. )
:-
fprintf( fp, "\n" );
fprintf( fp, "\t.method\tpublic static main([Ljava/lang/String;)V\n" );
fprintf( fp, "\t.limit\tlocals 1\n" );
fprintf( fp, "\t.limit\tstack 0\n" );
fprintf( fp, "\n" );

fprintf( fp, "\t.invokestatic\t%s/main()V\n", strTypeName );

fprintf( fp, "\treturn\n" );
fprintf( fp, "\n" );
fprintf( fp, "\t.end\tmethod ; main([Ljava/lang/String;)V\n" );
.
```

```

/* Auxiliary procedure -- CodeMethods:
 * - Code generation for methods.
 */

PROCEDURE CodeMethods( aClassBodyDeclarations )
LOCAL
{
  short BlkBegin = NoLbl;
  short BlkEnd   = NoLbl;
}

/* ----- Pre-conditions section ----- */

NIL :- Message( "CodeMethods( NoTree ), invalid call.", xxFatal, NoPosition ); .

/* ----- Main section ----- */

aMethodDeclaration(
  Modifiers := Modifiers,
  aType := rtnType:aType( .. ),
  Name := Name,
  FrameSize := FrameSize,
  WSSize := WSSize,
  aFormalParameters := prms:aFormalParameters( .. ),
  aStatements := stmts:aStatements( .. ),
  Rtn := Rtn,
  Next := Next:aClassBodyDeclarations( .. )
)
:-
  fprintf( fp, ";\n" );
  fprintf( fp, "*****\n" );
  fprintf( fp, ";\n" );
  fprintf( fp, "\t.method\t%s %s\n", EncodeModifiers( Modifiers ),
                                     GetCStr( Name ),
                                     EncodeSignature( prms, rtnType ) );
  fprintf( fp, "\t.limit\tlocals %i\n", FrameSize );
  fprintf( fp, "\t.limit\tstack %i\n", WSSize );
  fprintf( fp, "\n" );

  ResetLbl();
  BlkBegin = GetNextLbl();
  BlkEnd   = GetNextLbl();
  CodeVarDirectives( prms, BlkBegin, BlkEnd );
  CodeVarDirectives( stmts, BlkBegin, BlkEnd );

  fprintf( fp, "L%i:\n", BlkBegin );
  CodeStatements( stmts );
  fprintf( fp, "L%i:\n", BlkEnd );
  fprintf( fp, "\n" );

  IF ( ( rtnType->Kind == kaVoidType ) && ( Rtn & MayNotReturn ) ) THEN
    fprintf( fp, "\treturn\n" );
  END

  fprintf( fp, "\t.end\tmethod ; %s\n", GetCStr( Name ),
                                     EncodeSignature( prms, rtnType ) );
  CodeMethods( Next );
.

```

```

aFieldDeclaration(
  Next := Next:aClassBodyDeclarations( .. )
)
:-
  CodeMethods( Next );
.

aNoClassBodyDeclarations( .. ) :- /* end of class body declarations */ .

/* ----- Unprocessed node traps section ----- */

n : aClassBodyDeclarations( .. )
:-
  sprintf( errMsg, "CodeMethods() : Unprocessed node of type \"%s\"\n",
               Tree_NodeName[n->Kind] );
  Message( errMsg, xxFatal, NoPosition );
.

```

```

/* Auxiliary procedure -- CodeStatements:
* - Code generation for statements.
*
* - For control flow statements, the type of the expression is necessarily
* Boolean, so lazy evaluation is used. Since all control flow expressions
* are lazy-evaluated, control flow statements do not use the aBinary()
* and aUnary() patterns of the CodeExpression() procedure. */

PROCEDURE CodeStatements( [aStatements aVariableDeclarators] )
LOCAL
{
  short DoLbl    = NoLbl;
  short EndLbl   = NoLbl;
  short ElseLbl  = NoLbl;
  short TestLbl  = NoLbl;
  short BlkBegin = NoLbl;
  short BlkEnd   = NoLbl;
}

/* ----- Pre-conditions section ----- */

NIL :-
Message( "CodeStatements( NoTree ), invalid call.", xxFatal, NoPosition );
.

/* ----- Main section ----- */

aNoStatement( .. ) :- /* end of statements */ .

aLocalVariableDeclaration(
  Pos := Pos,
  aVariableDeclarators := vdecl:aVariableDeclarators( .. ),
  Next := Next:aStatements( .. )
)
:-
fprintf( fp, "\t.line\t%i ; Local var declarations.\n", Pos.Line );
CodeStatements( vdecl );
CodeStatements( Next );
.

aInnerBlock(
  aStatements := innerBody:aStatements( .. ),
  Next := Next:aStatements( .. )
)
:-
BlkBegin = GetNextLbl();
BlkEnd   = GetNextLbl();
CodeVarDirectives( innerBody, BlkBegin, BlkEnd );

fprintf( fp, "L%i:\n", BlkBegin );
CodeStatements( innerBody );
fprintf( fp, "L%i:\n", BlkEnd );

CodeStatements( Next );
.

```

```

aIfThen(
  Pos := Pos,
  aExpression := expr : aExpression( .. ),
  Then := thenPart:aStatements( .. ),
  Next := Next:aStatements( .. )
)
:-
EndLbl = GetNextLbl();
fprintf( fp, "\t.line\t%i ; IfThen up to L%i.\n",
  Pos.Line, EndLbl );
CodeLazyExpr( expr, NoLbl, EndLbl ); /* TrueLbl, FalseLbl */
CodeStatements( thenPart );
fprintf( fp, "L%i:\n", EndLbl );

CodeStatements( Next );
.

aIfThenElse(
  Pos := Pos,
  aExpression := expr:aExpression( .. ),
  Then := thenPart:aStatements( .. ),
  Else := elsePart:aStatements( .. ),
  Next := Next:aStatements( .. )
)
:-
ElseLbl = GetNextLbl();
EndLbl = GetNextLbl();
fprintf( fp, "\t.line\t%i ; IfThenElse up to L%i (Else L%i).\n",
  Pos.Line, EndLbl, ElseLbl );
CodeLazyExpr( expr, NoLbl, ElseLbl ); /* TrueLbl, FalseLbl */
CodeStatements( thenPart );
fprintf( fp, "\tgoto\tL%i\n", EndLbl );
fprintf( fp, "L%i:\n", ElseLbl );
CodeStatements( elsePart );
fprintf( fp, "L%i:\n", EndLbl );

CodeStatements( Next );
.

aWhileStatement(
  Pos := Pos,
  aExpression := expr:aExpression( .. ),
  aStatements := body:aStatements( .. ),
  Next := Next:aStatements( .. )
)
:-
DoLbl = GetNextLbl();
TestLbl = GetNextLbl();
fprintf( fp, "\t.line\t%i ; While from L%i to L%i.\n",
  Pos.Line, DoLbl, TestLbl );
fprintf( fp, "\tgoto\tL%i\n", TestLbl );
fprintf( fp, "L%i:\n", DoLbl );
CodeStatements( body );
fprintf( fp, "L%i:\n", TestLbl );
CodeLazyExpr( expr, DoLbl, NoLbl ); /* TrueLbl, FalseLbl */

CodeStatements( Next );
.

```

```

aReturnNoValueStatement(
  Pos := Pos,
  Next := Next:aStatements( .. )
)
:-
fprintf( fp, "\t.line\t%i\n", Pos.Line );
fprintf( fp, "\treturn\n", );
CodeStatements( Next );
.

aReturnValueStatement(
  Pos := Pos,
  RtnType := RtnType,
  aExpression := expr:aExpression( .. ),
  Next := Next:aStatements( .. )
)
:-
fprintf( fp, "\t.line\t%i ; return Value.\n", Pos.Line );
IF ( IsPrimary( expr ) || ( ! IsBoolean( expr::Type ) ) ) THEN
  CodeExpression( expr );
  CodeExpression( expr::Co );
ELSE
  EndLbl := GetNextLbl();
  fprintf( fp, "\ticonst_0\n", );
  CodeLazyExpr( expr, NoLbl, EndLbl ); /* TrueLbl, FalseLbl */
  fprintf( fp, "\tpop\n" );
  fprintf( fp, "\ticonst_1\n" );
  fprintf( fp, "L%i:\n", EndLbl );
END
fprintf( fp, "\t%s\n", TypedOps( RtnType, "return" ) );
CodeStatements( Next );
.

aAssignment(
  Pos := Pos,
  aLeftValue := lval:aLeftValue( .. ),
  aExpression := expr:aExpression( .. ),
  Next := Next:aStatements( .. )
)
:-
fprintf( fp, "\t.line\t%i ; Assignment.\n", Pos.Line );
CodeExpression( lval );

IF ( IsPrimary( expr ) || ( ! IsBoolean( expr::Type ) ) ) THEN
  CodeExpression( expr );
  CodeExpression( expr::Co );
ELSE
  EndLbl := GetNextLbl();
  fprintf( fp, "\ticonst_0\n" );
  CodeLazyExpr( expr, NoLbl, EndLbl ); /* TrueLbl, FalseLbl */
  fprintf( fp, "\tpop\n" );
  fprintf( fp, "\ticonst_1\n" );
  fprintf( fp, "L%i:\n", EndLbl );
END

CodeLeftValueStore( lval );
CodeStatements( Next );
.

```

```

aProcedureCall(
  Pos := Pos,
  Name := Name,
  Descriptor := method:sMethodDesc( .. ),
  aArguments := actuals:aArguments( .. ),
  Next := Next:aStatements( .. )
)
:-
fprintf( fp, "\t.line\t%i ; Procedure Call.\n", Pos.Line );
IF ( ! ( method::Modifiers & ModSTATIC ) ) THEN
  fprintf( fp, "\taload_0\n" );
END
CodeExpression( actuals );

IF ( method::Modifiers & ModSTATIC ) THEN
  fprintf( fp, "\tinvokestatic\t%s/%s\n", strTypeName, GetCStr( Name ),
    EncodeSignature( method::Formals, method::Type ) );
ELSE
  fprintf( fp, "\tinvokevirtual\t%s/%s\n", strTypeName,
    GetCStr( Name ),
    EncodeSignature( method::Formals, method::Type ) );
END

/* If non void result, clean the stack */
IF ( TypeSize( method::Type ) == 1 ) THEN
  fprintf( fp, "\tpop\n" );
ELSIF ( TypeSize( method::Type ) == 2 ) THEN
  fprintf( fp, "\tpop2\n" );
END

CodeStatements( Next );
.

aRead(
  Pos := Pos,
  aLeftValue := lval:aLeftValue( .. ),
  Next := Next:aStatements( .. )
)
:-
fprintf( fp, "\t.line\t%i ; read(..).\n", Pos.Line );
CodeExpression( lval );

/* Prompt */
fprintf( fp, "\tgetstatic\tjava/lang/System/out Ljava/io/PrintStream;\n" );
fprintf( fp, "\tldc\t\"> \n" );
fprintf( fp, "\tinvokevirtual\tjava/io/PrintStream/"
  "print(Ljava/lang/String;)V\n" );
/* Read */
fprintf( fp, "\tinvokestatic\t%s\n", TypedRead( lval::Type ) );

CodeLeftValueStore( lval );

CodeStatements( Next );
.

```



```

aPrintln(
  Pos := Pos,
  Next := Next:aStatements( .. )
)
:-
fprintf( fp, "\t.line\t%i ; println().\n", Pos.Line );
fprintf( fp, "\tgetstatic\tjava/lang/System/out Ljava/io/PrintStream;\n" );
fprintf( fp, "\tinvokevirtual\tjava/io/PrintStream/println()\n" );
CodeStatements( Next );
.

aPrintStringLiteral(
  Pos := Pos,
  Str := Str,
  Next := Next:aStatements( .. )
)
:-
fprintf( fp, "\t.line\t%i ; printstr().\n", Pos.Line );
fprintf( fp, "\tgetstatic\tjava/lang/System/out Ljava/io/PrintStream;\n" );
fprintf( fp, "\tlldc\t%s\n", StGetCStr( Str ) );
fprintf( fp, "\tinvokevirtual\tjava/io/PrintStream/"
  "print(Ljava/lang/String;)V\n" );
CodeStatements( Next );
.

aPrintValue(
  Pos := Pos,
  aExpression := expr:aExpression( .. ),
  Next := Next:aStatements( .. )
)
:-
fprintf( fp, "\t.line\t%i ; print(...).\n", Pos.Line );
fprintf( fp, "\tgetstatic\tjava/lang/System/out Ljava/io/PrintStream;\n" );

IF ( IsPrimary( expr ) || ( ! IsBoolean( expr::Type ) ) ) THEN
  CodeExpression( expr );
  CodeExpression( expr::Co );
ELSE
  EndLbl := GetNextLbl();
  fprintf( fp, "\ticonst_0\n" );
  CodeLazyExpr( expr, NoLbl, EndLbl ); /* TrueLbl, FalseLbl */
  fprintf( fp, "\tpop\n" );
  fprintf( fp, "\ticonst_1\n" );
  fprintf( fp, "L%i:\n", _EndLbl );
END

fprintf( fp, "\tinvokevirtual\tjava/io/PrintStream/print(%s)V\n",
  EncodeType( expr::Type ) );

CodeStatements( Next );
.

aNoVariableDeclarators( .. ) :- /* end of variable declarators */ .

```

```

aVariableDeclarator(
  Offset := Offset,
  aType := vtype:aType( .. ),
  Next := Next:aVariableDeclarators( .. )
)
:-
fprintf( fp, "\t%s\n", PushDefaultValue( vtype ) );
fprintf( fp, "\t%s\t%i\n", TypedOps( vtype, "store" ), Offset );
CodeStatements( Next );
.

/* ----- Unprocessed node traps section ----- */

n : aStatements( .. )
:-
sprintf( errMsg, "CodeStatements() : Unprocessed node of type \"%s\"\n",
  Tree_NodeName[n->Kind] );
Message( errMsg, xxFatal, NoPosition );
.

n : aVariableDeclarators( .. )
:-
sprintf( errMsg, "CodeStatements() : Unprocessed node of type \"%s\"\n",
  Tree_NodeName[n->Kind] );
Message( errMsg, xxFatal, NoPosition );
.

- :-
Message( "CodeStatements( Unknown_node_type ), invalid call.",
  xxFatal, NoPosition );
.

```

```

/* Auxiliary procedure -- CodeLeftValueStore:
 * - Code generation for storing a value.
 */

PROCEDURE CodeLeftValueStore( aLeftValue )

/* ----- Pre-conditions section ----- */

NIL :- Message( "CodeLeftValueStore( NoTree ), invalid call.",
                xxFatal, NoPosition ); .

/* ----- Main section ----- */

alValue(
    Descriptor := v:sVarDesc( .. )
)
:-
fprintf( fp, "\t%s\t%i\n", TypedOps( v::Type, "store" ), v::Offset );
.
/* Field target
 */
alValue(
    Descriptor := fld:sFieldDesc( .. )
)
:-
IF ( fld::Modifiers & ModSTATIC ) THEN
    fprintf( fp, "\tputstatic\t%s/%s %s\n", strTypeName,
            GetCStr( fld::Name ),
            EncodeType( fld::Type ) );
ELSE
    fprintf( fp, "\tputfield\t%s/%s %s\n", strTypeName,
            GetCStr( fld::Name ),
            EncodeType( fld::Type ) );
END
.
/* Array object target
 */
lval:aIndexlValue( .. )
:-
fprintf( fp, "\t%s\n", ArrayTypedOps( lval::Type, "store" ) );
.

/* ----- Unprocessed node traps section ----- */

n : aLeftValue( .. )
:-
sprintf( errMsg, "CodeLeftValueStore() : Unprocessed node of type \"%s\"\n",
        Tree_NodeName[n->Kind] );
Message( errMsg, xxFatal, NoPosition );
.

```

```

/* Auxiliary procedure -- CodeExpression:
 * - Code generation for non-Boolean expressions. */

PROCEDURE CodeExpression( [aExpression aDimSpec aArguments aCoercions] )
  LOCAL
  {
    short EndLbl;
  }

/* ----- Pre-conditions section ----- */

NIL := Message( "CodeExpression( NoTree ), invalid call.",
                xxFatal, NoPosition ); .

/* ----- Main section ----- */

aBooleanConst(
  Val := Val,
)
:-
  fprintf( fp, "\tldc\t%i\n", Val );
.

aIntConst(
  Val := Val,
)
:-
  fprintf( fp, "\tldc\t%i\n", Val );
.

aFloatConst(
  Val := Val,
)
:-
  fprintf( fp, "\tldc\t%ef\n", Val );
.

aNullConst( .. )
:-
  fprintf( fp, "\taconst_null\n", );
.

aValue( .. ) :- /* never generates any code */ .

aIndexlValue(
  aRightValue := rval:aRightValue( .. ),
  aExpression := expr:aExpression( .. )
)
:-
  CodeExpression( rval );
  /* No coercion on array references... */
  CodeExpression( expr );
  CodeExpression( expr::Co );
.

```

```

arValue(
  Descriptor := sFieldDesc(
    Name := Name,
    Modifiers := Modifiers,
    Type := fieldType:aType( .. )
  )
)
:-
  IF ( Modifiers & ModSTATIC ) THEN
    fprintf( fp, "\tgetstatic\t%s/%s %s\n", strTypeName,
              GetCStr( Name ),
              EncodeType( fieldType ) );
  ELSE
    fprintf( fp, "\tgetfield\t%s/%s %s\n", strTypeName,
              GetCStr( Name ),
              EncodeType( fieldType ) );
  END
.

arValue(
  Descriptor := sVarDesc(
    Type := valType:aType( .. ),
    Offset := Offset
  )
)
:-
  fprintf( fp, "\t%s\t%i\n", TypedOps( valType, "load" ), Offset );
.

aIndexrValue(
  Type := elemType:aType( .. ),
  aRightValue := rval:aRightValue( .. ),
  aExpression := expr:aExpression( .. )
)
:-
  CodeExpression( rval );
  /* No coercion on array references... */
  CodeExpression( expr );
  CodeExpression( expr::Co );
  fprintf( fp, "\t%s\n", ArrayTypedOps( elemType, "load" ) );
.

```

```

aFunctionCall(
  Name := Name,
  Descriptor := method:sMethodDesc( .. ),
  aArguments := actuals:aArguments( .. )
)
:-
IF ( ! ( method::Modifiers & ModSTATIC ) ) THEN
  fprintf( fp, "\taload_0\n" );
END
CodeExpression( actuals );
IF ( method::Modifiers & ModSTATIC ) THEN
  fprintf( fp, "\tinvokestatic\t%s/%s\n",
    strTypeName, GetCStr( Name ),
    EncodeSignature( method::Formals, method::Type ) );
ELSE
  fprintf( fp, "\tinvokevirtual\t%s/%s\n",
    strTypeName, GetCStr( Name ),
    EncodeSignature( method::Formals, method::Type ) );
END
.

aArrayCreator(
  aArrayType := arrayType:aArrayType( .. ),
  aDimSpec := spec:aDimSpec( .. ),
  NbInitDim := NbInitDim
)
:-
CodeExpression( spec );
fprintf( fp, "\tmultianewarray\t%s %i\n", EncodeType( arrayType ),
  NbInitDim );
.

aBinary(
  Op := Operator,
  OpFamily := (OpArithmetic),
  OpType := OpType:aType( .. ),
  Left := Left:aExpression( .. ),
  Right := Right:aExpression( .. )
)
:-
CodeExpression( Left );
CodeExpression( Left::Co );
CodeExpression( Right );
CodeExpression( Right::Co );
fprintf( fp, "\t%s\n", TypedOps( OpType, Ops( Operator ) ) );
.

aUnary(
  Op := Operator,
  OpFamily := (OpArithmetic),
  OpType := OpType:aType( .. ),
  aExpression := expr:aExpression( .. )
)
:-
CodeExpression( expr );
CodeExpression( expr::Co );
fprintf( fp, "\t%s\n", TypedOps( OpType, Ops( Operator ) ) );
.

```

```

aBasicDimSpec(
  aExpression := expr:aExpression( .. )
)
:-
CodeExpression( expr );
CodeExpression( expr::Co );
.

aUndefDimSpec(
  aDimSpec := subSpec:aDimSpec( .. )
)
:-
CodeExpression( subSpec );
.

aDefDimSpec(
  aDimSpec := subSpec:aDimSpec( .. ),
  aExpression := expr:aExpression( .. )
)
:-
CodeExpression( subSpec );
CodeExpression( expr );
CodeExpression( expr::Co );
.

aNoArgument( .. ) :- /* end of argument list. */ .

aArgument(
  aExpression := expr:aExpression( .. ),
  Next := Next:aArguments( .. )
)
:-
IF ( IsPrimary( expr ) || ( ! IsBoolean( expr::Type ) ) ) THEN
  CodeExpression( expr );
  CodeExpression( expr::Co );
ELSE
  EndLbl := GetNextLbl();
  fprintf( fp, "\ticonst_0\n" );
  CodeLazyExpr( expr, NoLbl, EndLbl ); /* TrueLbl, FalseLbl */
  fprintf( fp, "\tpop\n" );
  fprintf( fp, "\ticonst_1\n" );
  fprintf( fp, "L%i:\n", EndLbl );
END

CodeExpression( Next );
.

aNoCoercion( .. ) :- .

aIntToFloat( .. )
:-
fprintf( fp, "\ti2f\n" );
.

```

```
/* ----- Unprocessed node traps section ----- */
n : aExpression( .. )
:-
  sprintf( errMsg, "CodeExpression() : Unprocessed node of type \"%s\\n\",
    Tree_NodeName[n->Kind] );
  Message( errMsg, xxFatal, NoPosition );
.

n : aDimSpec( .. )
:-
  sprintf( errMsg, "CodeExpression() : Unprocessed node of type \"%s\\n\",
    Tree_NodeName[n->Kind] );
  Message( errMsg, xxFatal, NoPosition );
.

n : aArguments( .. )
:-
  sprintf( errMsg, "CodeExpression() : Unprocessed node of type \"%s\\n\",
    Tree_NodeName[n->Kind] );
  Message( errMsg, xxFatal, NoPosition );
.

n : aCoercions( .. )
:-
  sprintf( errMsg, "CodeExpression() : Unprocessed node of type \"%s\\n\",
    Tree_NodeName[n->Kind] );
  Message( errMsg, xxFatal, NoPosition );
.
:-
  Message( "CodeExpression( Unknown_node_type ), invalid call.",
    xxFatal, NoPosition );
.
```

```

/* Auxiliary procedure -- CodeLazyExpr:
* - Code generation for lazy expressions.
*
* - Primary expressions being standard evaluated, they use the lazy
* evaluation jump-on-result scheme. Only Boolean result expressions will be
* found ("aBooleanConst", Boolean "aRightValue" and Boolean "aFunctionCall")
* since lazy evaluation is only used for Boolean expressions. Also, Boolean
* "aLeftValue" will never be encountered since assignments use lazy
* evaluation only on their source expression.
*
* - Non-primary expressions with operators >, >=, < and <=:
* These cases are terminal cases, since Booleans cannot be compared in
* magnitude, that is the arguments are necessarily primary numeric or
* arithmetic expressions.
*
* - Non-primary expressions with operators == and !=:
* Boolean operands must be lazy evaluated. Non-Boolean operands must be
* standard evaluated. The result must always be given according to
* lazy evaluation jump-on-result scheme. */
PROCEDURE CodeLazyExpr( aExpression, TrueLbl : short, FalseLbl : short )
LOCAL {
    short EndLbl, ElseLbl;
}
/* ----- Pre-conditions section ----- */
NIL, _, _ :=
    Message( "CodeLazyExpr( NoTree, _, _ ), invalid call.",
            xxFatal, NoPosition );
.
n : aExpression( .. ), (NoLbl), (NoLbl)
:=
    sprintf( errMsg, "CodeLazyExpr( %s( ... ), NoLbl, NoLbl ) : Illegal labels "
            "Configuration...\n", Tree_NodeName[n->Kind] );
    Message( errMsg, xxFatal, NoPosition );
.
/* ----- Main section ----- */
expr : aPrimary(
    Type := aBooleanType( .. ),
    ), _, _
:=
    CodeExpression( expr );

IF ( TrueLbl != NoLbl ) THEN
    fprintf( fp, "\t\tifne\tL%i\n", TrueLbl );
    IF ( FalseLbl != NoLbl ) THEN
        fprintf( fp, "\t\tgoto\tL%i\n", FalseLbl );
    END
ELSIF ( FalseLbl != NoLbl ) THEN
    fprintf( fp, "\t\tifeq\tL%i\n", FalseLbl );
END
.

```

```

aBinary(
    OpFamily := (OpCmp),
    Op := Operator,
    OpType := t : aType( .. ),
    Left := Left:aExpression( .. ),
    Right := Right:aExpression( .. )
    ), _, _
:=
    CodeExpression( Left );
    CodeExpression( Left::Co );
    CodeExpression( Right );
    CodeExpression( Right::Co );

IF ( TrueLbl != NoLbl ) THEN
    CodeBinBranchOnCond( Operator, t, TrueLbl );
    IF ( FalseLbl != NoLbl ) THEN
        fprintf( fp, "\t\tgoto\tL%i\n", FalseLbl );
    END
ELSIF ( FalseLbl != NoLbl ) THEN
    CodeBinBranchOnNotCond( Operator, t, FalseLbl );
END
.

aBinary(
    OpType := aBooleanType( .. ),
    OpFamily := (OpEqv),
    Op := Operator,
    Left := Left:aExpression( .. ),
    Right := Right:aExpression( .. )
    ), _, _
:=
    ElseLbl = GetNextLbl();
    EndLbl = GetNextLbl();

    CodeLazyExpr( Left, NoLbl, ElseLbl );

                                /* Reach iff Left == true */
    CodeLazyExpr( Right, TrueLbl, FalseLbl );
    fprintf( fp, "\t\tgoto\tL%i\n", EndLbl );

                                /* Reach iff Left == false */
    fprintf( fp, "L%i:\n", ElseLbl );
    CodeLazyExpr( Right, FalseLbl, TrueLbl );

    fprintf( fp, "L%i:\n", EndLbl );
.

```

```

/* terminal case with non-Boolean operands */
aBinary(
  OpFamily := (OpEqv),
  Op := Operator,
  OpType := t : aType( .. ),
  Left := Left:aExpression( .. ),
  Right := Right:aExpression( .. )
), _' _
:-
CodeExpression( Left );
CodeExpression( Left::Co );
CodeExpression( Right );
CodeExpression( Right::Co );

IF ( TrueLbl != NoLbl ) THEN
  CodeBinBranchOnCond( Operator, t, TrueLbl );
  IF ( FalseLbl != NoLbl ) THEN
    fprintf( fp, "\tgoto\tL%i\n", FalseLbl );
  END
ELSIF ( FalseLbl != NoLbl ) THEN
  CodeBinBranchOnNotCond( Operator, t, FalseLbl );
END
.
/* And operator */
aBinary(
  Op := (OpAND),
  Left := Left:aExpression( .. ),
  Right := Right:aExpression( .. )
), _' _
:-
EndLbl = NoLbl;
IF ( FalseLbl == NoLbl ) THEN
  EndLbl = GetNextLbl();
  CodeLazyExpr( Left, NoLbl, EndLbl );
ELSE
  CodeLazyExpr( Left, NoLbl, FalseLbl );
END
/* Reach if Left == true */
CodeLazyExpr( Right, TrueLbl, FalseLbl );

IF ( EndLbl != NoLbl ) THEN
  fprintf( fp, "L%i:\n", EndLbl );
END
.

```

```

/* Or operator */
aBinary(
  Op := (OpOR),
  Left := Left:aExpression( .. ),
  Right := Right:aExpression( .. )
), _' _
:-
EndLbl = NoLbl;
IF ( TrueLbl == NoLbl ) THEN
  EndLbl = GetNextLbl();
  CodeLazyExpr( Left, EndLbl, NoLbl );
ELSE
  CodeLazyExpr( Left, TrueLbl, NoLbl );
END
/* Reach if Left == false */
CodeLazyExpr( Right, TrueLbl, FalseLbl );

IF ( EndLbl != NoLbl ) THEN
  fprintf( fp, "L%i:\n", EndLbl );
END
.
/* Not operator */
aUnary(
  Op := (OpNot),
  aExpression := expr:aExpression( .. )
), _' _
:-
CodeLazyExpr( expr, FalseLbl, TrueLbl );
.
/* ----- Unprocessed node traps section ----- */
n : aExpression( .. ), _' _
:-
sprintf( errMsg, "CodeLazyExpr() : Unprocessed node of type \"%s\"\n",
  Tree_NodeName[n->Kind] );
Message( errMsg, xxFatal, NoPosition );
.

```

```

/* Auxiliary procedure -- CodeBinBranchOnCond:
 * - Code generation for positive branches.
 */

PROCEDURE CodeBinBranchOnCond( op : short, type : aType, Lbl : short )

/* ----- Pre-conditions section ----- */

_, NIL, _ :=
  Message( "CodeBinBranchOnCond( _, NoTree, _ ), invalid call.",
           xxFatal, NoPosition );
.

_, _ := (NoLbl)
  :=
  sprintf( errMsg, "CodeBinBranchOnCond( %i, %s, NoLabel ) : Illegal call.\n",
           op, Tree_NodeName[type->Kind] );
  Message( errMsg, xxFatal, NoPosition );
.

/* ----- Main section ----- */

(OpLT), aIntType( .. ), _
  :=
  fprintf( fp, "\tif_icmplt\tL%i\n", Lbl );
.

(OpLT), aFloatType( .. ), _
  :=
  fprintf( fp, "\tfcmpg\n" );
  fprintf( fp, "\tiflt\tL%i\n", Lbl );
.

(OpGT), aIntType( .. ), _
  :=
  fprintf( fp, "\tif_icmpgt\tL%i\n", Lbl );
.

(OpGT), aFloatType( .. ), _
  :=
  fprintf( fp, "\tfcmpg\n" );
  fprintf( fp, "\tifgt\tL%i\n", Lbl );
.

(OpEQ), aIntType( .. ), _
  :=
  fprintf( fp, "\tif_icmpeq\tL%i\n", Lbl );
.

(OpEQ), aFloatType( .. ), _
  :=
  fprintf( fp, "\tfcmpg\n" );
  fprintf( fp, "\tifeq\tL%i\n", Lbl );
.

(OpEQ), aReferenceType( .. ), _
  :=
  fprintf( fp, "\tif_acmpeq\tL%i\n", Lbl );
.

```

```

/* ----- Unprocessed node traps section ----- */

_ := _
  :=
  sprintf( errMsg, "CodeBinBranchOnCond( %i, %s, _ ) "
               ": should not happen (unprocessed).\n",
           op, Tree_NodeName[type->Kind] );
  Message( errMsg, xxFatal, NoPosition );
.

```



```

/* Auxiliary procedure -- CodeBinBranchOnNotCond:
 * - Code generation for negative branches. */

PROCEDURE CodeBinBranchOnNotCond( op : short, type : aType, Lbl : short )

/* ----- Pre-conditions section ----- */

_, NIL, _ :-
  Message( "CodeBinBranchOnNotCond( _, NoTree, _ ), invalid call.",
    xxFatal, NoPosition );
.

_, _', (NoLbl)
  :-
    sprintf( errMsg, "CodeBinBranchOnNotCond( %i, %s, NoLabel) "
      ": Illegal call.\n",
      op, Tree_NodeName[type->Kind] );
    Message( errMsg, xxFatal, NoPosition );
.

/* ----- Main section ----- */

(OpLT), aIntType( .. ), _
  :-
    fprintf( fp, "\tif_icmpge\tL%i\n", Lbl );
.

(OpLT), aFloatType( .. ), _
  :-
    fprintf( fp, "\tfcmpg\n" );
    fprintf( fp, "\tifge\tL%i\n", Lbl );
.

(OpGT), aIntType( .. ), _
  :-
    fprintf( fp, "\tif_icmple\tL%i\n", Lbl );
.

(OpGT), aFloatType( .. ), _
  :-
    fprintf( fp, "\tfcmpg\n" );
    fprintf( fp, "\tifl\tL%i\n", Lbl );
.

(OpEQ), aIntType( .. ), _
  :-
    fprintf( fp, "\tif_icmpne\tL%i\n", Lbl );
.

(OpEQ), aFloatType( .. ), _
  :-
    fprintf( fp, "\tfcmpg\n" );
    fprintf( fp, "\tifne\tL%i\n", Lbl );
.

```

```

(OpEQ), aReferenceType( .. ), _
  :-
    fprintf( fp, "\tif_acmpne\tL%i\n", Lbl );
.

/* ----- Unprocessed node traps section ----- */

_, _', _
  :-
    sprintf( errMsg, "CodeBinBranchOnNotCond( %i, %s, _ ) "
      ": should not happen (unprocessed).\n",
      op, Tree_NodeName[type->Kind] );
    Message( errMsg, xxFatal, NoPosition );
.

```

```

/* Auxiliary procedure -- CodeVarDirectives:
 * - Code generation for var directives.
 */

PROCEDURE CodeVarDirectives( [aStatements aVariableDeclarators
                             aFormalParameters],
                             BlkBegin : short, BlkEnd : short )

/* ----- Pre-conditions section ----- */

NIL, _, _ :-
  Message( "CodeVarDirectives( NoTree, _, _ ), invalid call.",
           xxFatal, NoPosition );
.

_, (NoLbl), _
:-
  sprintf( errMsg, "CodeVarDirectives( _, NoLabel, _ ) "
             ": Illegal call.\n" );
  Message( errMsg, xxFatal, NoPosition );
.

_, (NoLbl)
:-
  sprintf( errMsg, "CodeVarDirectives( _, _, NoLabel ) "
             ": Illegal call.\n" );
  Message( errMsg, xxFatal, NoPosition );
.

/* ----- Main section ----- */

aNoStatement( .. ), Lbl1, Lbl2 :- .

stmt : aLocalVariableDeclaration( .. ), Lbl1, Lbl2
:-
  CodeVarDirectives( stmt::aVariableDeclarators, BlkBegin, BlkEnd );
  CodeVarDirectives( stmt::Next, BlkBegin, BlkEnd );
.

stmt : aStatement( .. ), Lbl1, Lbl2
:-
  CodeVarDirectives( stmt::Next, BlkBegin, BlkEnd );
.

aNoVariableDeclarators( .. ), Lbl1, Lbl2 :- .

decl : aVariableDeclarator( .. ), Lbl1, Lbl2
:-
  fprintf( fp, "\t.var\t%i is '%s' %s from L%i to L%i\n",
           decl::Offset, GetCStr( decl::Name ),
           EncodeType( decl::aType ), Lbl1, Lbl2 );
  CodeVarDirectives( decl::Next, BlkBegin, BlkEnd );
.

aNoParameter( .. ), Lbl1, Lbl2 :- .

```

```

prm : aFormalParameter( .. ), Lbl1, Lbl2
:-
  fprintf( fp, "\t.var\t%i is '%s' %s from L%i to L%i\n",
           prm::Offset, GetCStr( prm::Name ),
           EncodeType( prm::aType ), Lbl1, Lbl2 );
  CodeVarDirectives( prm::Next, BlkBegin, BlkEnd );
.

/* ----- Unprocessed node traps section ----- */

n : aStatements( .. ), _, _
:-
  sprintf( errMsg, "CodeVarDirectives() : Unprocessed node of type \"%s\"\n",
           Tree_NodeName[n->Kind] );
  Message( errMsg, xxFatal, NoPosition );
.

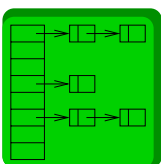
n : aVariableDeclarators( .. ), _, _
:-
  sprintf( errMsg, "CodeVarDirectives() : Unprocessed node of type \"%s\"\n",
           Tree_NodeName[n->Kind] );
  Message( errMsg, xxFatal, NoPosition );
.

n : aFormalParameters( .. ), _, _
:-
  sprintf( errMsg, "CodeVarDirectives() : Unprocessed node of type \"%s\"\n",
           Tree_NodeName[n->Kind] );
  Message( errMsg, xxFatal, NoPosition );
.

```

Annexe G

La spécification de la table des symboles



Le fichier `SymTab.cg` contient la spécification du gestionnaire de la table des symboles. Il est divisé en six sections identifiées par les clauses `TREE`, `EXPORT`, `GLOBAL`, `BEGIN`, `CLOSE` et `RULE`.

- La section identifiée par la clause `TREE` spécifie le nom des fichiers (`SymTab.h` et `SymTab.c`) qui contiennent le code *C* du gestionnaire de la table des symboles.
- Le code *C* sous la clause `EXPORT` est inséré intégralement dans le fichier `SymTab.h` généré par l'outil `ast`. Cette section contient principalement la déclaration des fonctions d'interrogation visibles de l'extérieur ainsi que les variables externes pour les pointeurs vers les noeuds constants de fin de listes (*nEndOfPackages*, *nEndOfTypes*, *nEndOfMembers* et *nEndOfVariables*) créés dans la section `BEGIN`. Elle contient aussi la définition de symboles (*NoEnv*, *NoVariable* et *NoDescriptor*) qui sont vus comme des spécialisations du symbole *NoSymTab* généré par l'outil `ast`.
- Le code *C* sous la clause `GLOBAL` est inséré intégralement au niveau global dans le fichier `SymTab.c` généré par l'outil `ast`. Cette section contient principalement la définition des fonctions d'interrogation de la table des symboles.
- Le code *C* sous la clause `BEGIN` est inséré intégralement dans la fonction *BeginSymTab* du gestionnaire de la table des symboles (`SymTab.c`).
- Le code *C* sous la clause `CLOSE` est inséré intégralement dans la fonction *CloseSymTab* du gestionnaire de la table des symboles (`SymTab.c`).
- Le code sous la clause `RULE` n'est pas du code *C* mais des règles d'une grammaire abstraite qui représente les structures de données de la table des symboles.

L'outil `ast` génère du code *C* qui contient la définition de plusieurs identificateurs. Ils sont déduits à partir des symboles qui apparaissent dans la section `RULE`. Par exemple, à partir du symbole *sTypes*, les identificateurs suivants sont automatiquement définis :

- *ksTypes* pour la sorte d'un noeud *sTypes* ;
- *tsTypes* pour le type (au sens *C*) d'un noeud *sTypes* ;
- *msTypes* pour la fonction de création d'un noeud *sTypes*.

Enfin, deux identificateurs sont définis par l'outil **ast** à partir du nom sous la clause **TREE** :

- *NoSymTab* pour le pointeur nul ;
- *tSymTab* pour le type générique des noeuds.

```

/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.
*/

/* SymTab.cg */

/* Description: Declaration of data structures for the symbol table and
definition of the query functions.
Author: Daniel Cote.
Date: March 2006.
*/

TREE SymTab /* Generate code into SymTab.h and SymTab.c */

EXPORT /* Code to be inserted in SymTab.h */
{
#define NoEnv NoSymTab
#define NoVariable NoSymTab
#define NoDescriptor NoSymTab

#include "Idents.h" /* cocktail: Identifier (string) table */
#include "Tree.h" /* Generated by cocktail: AST node constructors */

/* Module interface */
extern tSymTab IdentifySymbolRef( tIdent, tsEnv, tsVariables );
extern tSymTab IdentifyMethodRef( tIdent, tTree, tsEnv );
extern tTree GetSymbolType( tSymTab );
extern tTree GetMethodPrms( tSymTab );

extern rbool IsDeclaredType( tIdent, tsTypes );
extern rbool IsDeclaredVariable( tIdent, tsVariables );
extern rbool IsDeclaredField( tIdent, tsMembers );
extern rbool IsDeclaredMethod( tIdent, tsMembers );

extern rbool IsObjectKind( tSymTab, short );

/* Constant nodes */
extern tsPackages nEndOfPackages;
extern tsTypes nEndOfTypes;
extern tsMembers nEndOfMembers;
extern tsVariables nEndOfVariables;
}

```

```

GLOBAL /* Code to be inserted in SymTab.c (level 0) */
{
#include "ratc.h" /* cocktail: Boolean type */

static rbool bModuleInitialized = rfalse; /* Module initialization flag */
static rbool bModuleFinalized = rfalse; /* Module finalization flag */

tsPackages nEndOfPackages;
tsTypes nEndOfTypes;
tsMembers nEndOfMembers;
tsVariables nEndOfVariables;

/* Intermediate functions */
static tsTypes IdentifyType( tIdent, tsTypes );
static tsVariables IdentifyVariable( tIdent, tsVariables );
static tsMembers IdentifyMethod( tIdent, tsMembers );
static tsMembers IdentifyField( tIdent, tsMembers );
}

```

```

tSymTab IdentifySymbolRef( tIdent id, tsEnv env, tsVariables vlist )
{
    /* Description: Identify a symbol from a list of variables or an environment.
    Author:      Daniel Cote.
    Date:       March 2006.
    Input:      id: the address of an identifier.
                env: an environment.
                vlist: a list of variables.
    Output:     A pointer to the descriptor of a variable or a field.    */

    tSymTab result = NoDescriptor;

    if ( vlist != NoVariable )
    {
        vlist = IdentifyVariable( id, vlist );
        if ( vlist->Kind != ksEndOfVariables )
            result = vlist;
    }

    if ( ( result == NoDescriptor )
        && ( env != NoEnv )
        && ( env->sEnv.CurrentClass->Kind == ksClassDesc ) )
    {
        tsMembers mlist = env->sEnv.CurrentClass->sClassDesc.MemberList;

        mlist = IdentifyField( id, mlist );
        if ( mlist->Kind != ksEndOfMembers )
            result = mlist;
    }
    return result;
}

/* ----- */

tSymTab IdentifyMethodRef( tIdent id, tTree actuals, tsEnv env )
{
    /* Description: Identify a method from an environment.
    Author:      Daniel Cote.
    Date:       March 2006.
    Input:      id: the address of an identifier.
                actuals: a list of actual parameters.
                env: an environment.
    Output:     A pointer to the descriptor of a method.    */

    tsMembers result = NoDescriptor;

    if ( ( actuals != NoTree )
        && ( env != NoEnv )
        && ( env->sEnv.CurrentClass->Kind == ksClassDesc ) )
    {
        tsMembers mlist = env->sEnv.CurrentClass->sClassDesc.MemberList;

        mlist = IdentifyMethod( id, mlist );
        if ( mlist->Kind != ksEndOfMembers )
            result = mlist;
    }
    return result;
}

```

```

tTree GetSymbolType( tSymTab descr )
{
    /* Description: Get the type of a symbol.
    Author:      Daniel Cote.
    Date:       March 2006.
    Input:      descr: a symbol descriptor.
    Output:     A pointer (taType) to the root of a subtree representing
                the type of a symbol.    */

    tTree result = nErrorType;

    if ( descr != NoDescriptor )
    {
        if ( descr->Kind == ksVarDesc )
            result = descr->sVarDesc.Type;
        else if ( descr->Kind == ksFieldDesc )
            result = descr->sFieldDesc.Type;
        else if ( descr->Kind == ksMethodDesc )
            result = descr->sMethodDesc.Type;
    }
    return result;
}

/* ----- */

tTree GetMethodPrms( tSymTab descr )
{
    /* Description: Get the formal parameters of a method.
    Author:      Daniel Cote.
    Date:       March 2006.
    Input:      descr: a symbol descriptor.
    Output:     A pointer (taFormalParameters) to the root of a subtree
                representing the formals parameters.    */

    tTree result = NoFormal;

    if ( descr != NoDescriptor )
    {
        if ( descr->Kind == ksMethodDesc )
            result = descr->sMethodDesc.Formals;
    }
    return result;
}

```

```

static tsTypes IdentifyType( tIdent id, tsTypes tlist )
{
    /* Description: Find a class or an interface in a list of types.
    Author:      Daniel Cote.
    Date:       March 2006.
    Input:      id: the address of an identifier.
                tlist: a list of types.
    Output:     A pointer to the descriptor of the type.          */

    if ( tlist != NULL )
    {
        while ( tlist->Kind != ksEndOfTypes )
            if ( tlist->sTypDesc.Name == id )
                return tlist;
            else
                tlist = tlist->sTypDesc.Previous;
    }
    return nEndOfTypes;
}

/* ----- */

rbool IsDeclaredType( tIdent id, tsTypes tlist )
{
    /* Description: Check if an identifier is a type.
    Author:      Daniel Cote.
    Date:       March 2006.
    Input:      id: the address of an identifier.
                tlist: a list of types.
    Output:     true if success; false otherwise.                */

    return ( IdentifyType( id, tlist )->Kind != ksEndOfTypes );
}

```

```

static tsVariables IdentifyVariable( tIdent id, tsVariables vlist )
{
    /* Description: Find an identifier in a list of variables.
    Author:      Daniel Cote.
    Date:       March 2006.
    Input:      id: the address of an identifier.
                vlist: a list of variables.
    Output:     A pointer to the descriptor of the variable.      */

    if ( vlist != NULL )
    {
        while ( vlist->Kind != ksEndOfVariables )
            if ( vlist->sVarDesc.Name == id )
                return vlist;
            else
                vlist = vlist->sVarDesc.Previous;
    }
    return nEndOfVariables;
}

/* ----- */

rbool IsDeclaredVariable( tIdent id, tsVariables vlist )
{
    /* Description: Check if an identifier is a variable.
    Author:      Daniel Cote.
    Date:       March 2006.
    Input:      id: the address of an identifier.
                vlist: a list of variables.
    Output:     true if success; false otherwise.                */

    return ( IdentifyVariable( id, vlist )->Kind != ksEndOfVariables );
}

```

```

static tsMembers IdentifyField( tIdent id, tsMembers mlist )
{
    /* Description: Find a field in a list of members.
    Author:      Daniel Cote.
    Date:        March 2006.
    Input:       id: the address of an identifier.
                mlist: a list of members.
    Output:      A pointer to the descriptor of the field.          */

    if ( mlist != NULL )
    {
        while ( mlist->Kind != ksEndOfMembers )
            if ( ( mlist->Kind == ksFieldDesc )
                && ( mlist->sFieldDesc.Name == id ) )
                return mlist;
            else
                mlist = mlist->sMbrDesc.Previous;
    }
    return nEndOfMembers;
}

/* ----- */

rbool IsDeclaredField( tIdent id, tsMembers mlist )
{
    /* Description: Check if an identifier is a field.
    Author:      Daniel Cote.
    Date:        March 2006.
    Input:       id: the address of an identifier.
                mlist: a list of members.
    Output:      true if success; false otherwise.          */

    return ( IdentifyField( id, mlist )->Kind != ksEndOfMembers );
}

```

```

static tsMembers IdentifyMethod( tIdent id, tsMembers mlist )
{
    /* Description: Find a method in a list of members.
    Author:      Daniel Cote.
    Date:        March 2006.
    Input:       id: the address of an identifier.
                mlist: a list of members.
    Output:      A pointer to the descriptor of the method.      */

    if ( mlist != NULL )
    {
        while ( mlist->Kind != ksEndOfMembers )
            if ( ( mlist->Kind == ksMethodDesc )
                && ( mlist->sMethodDesc.Name == id ) )
                return mlist;
            else
                mlist = mlist->sMbrDesc.Previous;
    }
    return nEndOfMembers;
}

/* ----- */

rbool IsDeclaredMethod( tIdent id, tsMembers mlist )
{
    /* Description: Check if an identifier is a method.
    Author:      Daniel Cote.
    Date:        March 2006.
    Input:       id: the address of an identifier.
                mlist: a list of members.
    Output:      true if success; false otherwise.          */

    return ( IdentifyMethod( id, mlist )->Kind != ksEndOfMembers );
}

/* ----- */

rbool IsObjectKind( tSymTab object, short kind )
{
    /* Description: Check if an object is of a specific kind.
    Author:      Daniel Cote.
    Date:        March 2006.
    Input:       object: an objet of the symbol table.
                kind: a kind of object.
    Output:      true if success; false otherwise.          */

    return object->Kind == kind;
}

```



```

BEGIN                                /* Code for initializing the symbol table handler */
{
  if ( ! bModuleInitialized )
  {
    /* Once code */
    fprintf( stderr, "    Initializing the symbol table...\n" );

    nEndOfPackages = msEndOfPackages();      /* Constant node creation */
    nEndOfTypes    = msEndOfTypes();
    nEndOfMembers  = msEndOfMembers();
    nEndOfVariables = msEndOfVariables();

    bModuleInitialized = rtrue;
  }
}

CLOSE                                /* Code for finalizing the symbol table handler */
{
  if ( bModuleInitialized && ( ! bModuleFinalized ) )
  {
    /* Once code */
    fprintf( stderr, "    Finalizing the symbol table...\n" );

    bModuleFinalized = rtrue;
  }
}

```

```

RULE                                /* Definition of data structures */

sEnv = CurrentPkg: sPackages IN
      CurrentClass: sTypes IN .

sPackages = <
  sEndOfPackages = .
  sPkgDesc       = Previous: sPackages IN
                  [Name: tIdent IN]
                  TypList: sTypes IN
                  PkgList: sPackages IN .
> .

sTypes = <
  sEndOfTypes = .
  sTypDesc    = Previous: sTypes IN
                [Name: tIdent IN]
                [Modifiers: short IN]
                MemberList: sMembers IN <
                  sClassDesc = .
                  sIntfDesc = .
                > .
> .

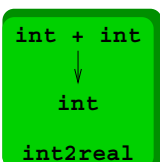
sMembers = <
  sEndOfMembers = .
  sMbrDesc      = Previous: sMembers IN
                  [Name: tIdent IN]
                  [Modifiers: short IN]
                  [Type: tTree IN] <
                    sFieldDesc = .
                    sMethodDesc = [Formals: tTree IN] .
                  > .
> .

sVariables = <
  sEndOfVariables = .
  sVarDesc        = Previous: sVariables IN
                  [Name: tIdent IN]
                  [Type: tTree IN]
                  [Offset: short IN] .
> .

```


Annexe H

Les procédures relatives aux types



Le fichier `TypeSys.pum` contient les procédures de vérification et d'inférence de types utilisées par l'analyseur sémantique et le générateur de code. Il est divisé en sept sections identifiées par les clauses `TRAFO`, `TREE`, `EXPORT`, `GLOBAL`, `BEGIN` et `CLOSE` ainsi que par un ensemble de définitions de fonctions, de procédures et de prédicats.

- La section identifiée par la clause `TRAFO` spécifie le nom des fichiers qui contiennent le code *C* du module de vérification et d'inférence de types (`TypeSys.h` et `TypeSys.c`).
- La section identifiée par la clause `TREE` spécifie le nom des fichiers (`Tree.h`, `Tree.TS`, `SymTab.TS` et `SymTab.h`) qui contiennent les définitions des structures de données manipulées par le module de vérification et d'inférence de types.
- Le code *C* sous la clause `EXPORT` est inséré intégralement dans le fichier `TypeSys.h` généré par l'outil `puma`.
- Le code *C* sous la clause `GLOBAL` est inséré intégralement au niveau global dans le fichier `TypeSys.c` généré par l'outil `puma`. La fonction *C* `EncodeModifiers` construit une liste des modificateurs à partir d'une suite de bits.
- Le code *C* sous la clause `BEGIN` est inséré intégralement dans la fonction `BeginTypeSys` du module de vérification et d'inférence de types (`TypeSys.c`).
- Le code *C* sous la clause `CLOSE` est inséré intégralement dans la fonction `CloseTypeSys` du module de vérification et d'inférence de types (`TypeSys.c`).

Les fonctions, les procédures et les prédicats sont groupés en deux sous-sections (chacune identifiée par un commentaire), l'une pour celles et ceux utilisés par l'analyseur sémantique et l'autre pour celles et ceux utilisés par le générateur de code.

Les fonctions, les procédures et les prédicats se présentent sous la forme d'un groupe de règles qui se terminent par un point. Chaque règle comporte un antécédent et un conséquent séparés par les symboles « `:-` ». L'antécédent d'une règle possède des formes (*Pattern*) groupées en fonction du nombre de paramètres dans la définition d'une fonction, d'une procédure ou d'un prédicat. Les groupes de formes sont séparés par un point-virgule qui agit comme un *ou* logique. Le symbole « `_` » est une forme particulière qui correspond à tout paramètre. Une suite de points (« `..` ») à l'intérieur d'une forme indique un nombre quelconque d'attributs ou de types de noeuds sans les mentionner explicitement. Une procédure d'appariement détermine à quel groupe de formes correspondent les paramètres.

La valeur par défaut retournée par un prédicat est la valeur *vrai*. La valeur **FAIL** est explicitement retournée. Dans les règles d'une fonction, le verbe **RETURN** est toujours placé dans l'antécédent. Il est suivi d'expression à partir de laquelle l'outil **puma** peut effectuer des appariements. La valeur de cette expression est la valeur retournée par la fonction et elle peut dépendre des instructions dans la clause conséquente.

```

/*
Copyright (c) 2006 Richard St-Denis and Universite de Sherbrooke
All rights reserved.

This file is part of the java-- compiler software. Redistribution and use
in source and binary forms are permitted provided that the above copyright
notice and an acknowledgment that the software was developed by Daniel Cote
and Richard St-Denis, Universite de Sherbrooke are inserted in any
documentation, advertising materials, and other materials related to such
use.

The name of the university may not be used to endorse or promote products
derived from this software without specific prior written permission.

This software is provided "AS IS" and without any express or implied
warranties, including, without limitation, the implied warranties of
merchantability and fitness for a particular purpose.

                                */
                                /* TypeSys.pum                                */

/* Description: Specification of the type system.
Author:      Daniel Cote.
Date:       March 2006.

                                */

TRAFO TypeSys                /* Generate code into TypeSys.h and TypeSys.c */
TREE Tree SymTab            /* Use the definitions from Tree.h, Tree.TS,
                                Symtab.h and Symtab.TS                                */

EXPORT
{
    /* Code to be inserted in TypeSys.h */
    extern char *EncodeModifiers( short );
}

```

```

GLOBAL
{
    #include <stdio.h>                /* Code to be inserted in TypeSys.c (level 0) */
    #include <string.h>              /* ISO C Standard: 4.9 input/output */
    #include "ratc.h"                /* ISO C Standard: 4.11 string handling */

    #include "ratc.h"                /* cocktail: Boolean type */
    #include "Errors.h"              /* cocktail: Error handler */

    #include "Semantics.h"           /* Generated by cocktail: Attribute evaluator */

    static rbool bModuleInitialized = rfalse; /* Module initialization flag */
    static rbool bModuleFinalized   = rfalse; /* Module finalization flag */

    char *EncodeModifiers( short mods )
    {
        static char str[256];

        strcpy( str, "" );

        if ( mods & ModSTATIC )
            strcat( str, "static" );

        if ( mods & ModPUBLIC )
        {
            if ( strlen( str ) > 0 )
                strcat( str, " " );
            strcat( str, "public" );
        }

        return str;
    }
}

BEGIN
{
    /* Code for initializing the type system */
    if ( ! bModuleInitialized )
    {
        /* Once code */
        fprintf( stderr, "    Initializing the type system...\n" );

        bModuleInitialized = rtrue;
    }
}

CLOSE
{
    /* Code for finalizing the type system */
    if ( bModuleInitialized && ( ! bModuleFinalized ) )
    {
        /* Once code */
        fprintf( stderr, "    Finalizing the type system...\n" );

        bModuleFinalized = rtrue;
    }
}

```

```

/* Predicates, functions and procedures used by the attribute evaluator */

/* Check for a variable descriptor */
PUBLIC PREDICATE IsVarDesc( [sMbrDesc sVarDesc] )
sVarDesc( .. ) :- .
_ :- FAIL .

/* Check for a static member descriptor */
PUBLIC PREDICATE IsStaticMbr( sMbrDesc )
sMbrDesc( Modifiers := Modifiers ) :-
IF ( ! ( Modifiers & ModSTATIC ) ) THEN FAIL; END .

/* Check for a primitive non-reference type */
PUBLIC PREDICATE IsScalarType( aType )
aIntType ;
aFloatType ;
aBooleanType :- .
_ :- FAIL .

/* Extract the type of the elements of an array */
PUBLIC FUNCTION GetElementType( aType ) aType
aErrorType ;
aNaT RETURN nNaT :- .
aArrayType( t, .. ) RETURN t :- .
.. RETURN nErrorType :- .

/* Return the JVM stack size requirement of a type */
PUBLIC FUNCTION TypeSize( aType ) short
aIntType ;
aFloatType ;
aBooleanType ;
aReferenceType RETURN 1 :- .
aLongType ;
aDoubleType RETURN 2 :- .
_ RETURN 0 :- .

/* Return the JVM stack size requirement of a type
once coercion (if any) has been applied */
PUBLIC FUNCTION CoercedTypeSize( t : aType, co : aCoercions ) short
_ , aNoCoercion( .. ) RETURN TypeSize( t ) :- .
aIntType, aIntToFloat RETURN TypeSize( nFloatType ) :- .

```

```

/* Calculate the result type of a binary expression */
PUBLIC FUNCTION BinaryResultType( aType, aType, Op : short,
OpFamily : short ) aType
aErrorType , _aErrorType , _ , _ ;
t:aNaT , t:aNaT , _ , _ ; RETURN nNaT :- .
_ , t:aNaT , _ , _ ; RETURN t :- .

t:aBooleanType , aBooleanType , _ , (OpLogic) ;
t:aIntType , aIntType , _ , (OpArithmetic) ;
t:aFloatType , aFloatType , _ , (OpArithmetic) RETURN t :- .

aIntType , aIntType , _ , (OpCmp) ;
aIntType , aIntType , _ , (OpEqv) ;
aFloatType , aFloatType , _ , (OpCmp) ;
aFloatType , aFloatType , _ , (OpEqv) ;
aBooleanType , aBooleanType , _ , (OpEqv) ;
aReferenceType , aReferenceType , _ , (OpEqv) RETURN nBooleanType :- .
.. RETURN nErrorType :- .

/* Determine the type of a binary operator */
PUBLIC FUNCTION BinaryOpType( aType, aType, Op : short,
OpFamily : short ) aType
aErrorType , _aErrorType , _ , _ ;
t:aNaT , t:aNaT , _ , _ ; RETURN nNaT :- .
_ , t:aNaT , _ , _ ; RETURN t :- .

t:aBooleanType , aBooleanType , _ , (OpLogic) ;
t:aIntType , aIntType , _ , (OpArithmetic) ;
t:aFloatType , aFloatType , _ , (OpArithmetic) RETURN t :- .

t:aIntType , aIntType , _ , (OpCmp) ;
t:aIntType , aIntType , _ , (OpEqv) RETURN t :- .
t:aFloatType , aFloatType , _ , (OpCmp) ;
t:aFloatType , aFloatType , _ , (OpEqv) RETURN t :- .
t:aBooleanType , aBooleanType , _ , (OpEqv) ;
t:aReferenceType , aReferenceType , _ , (OpEqv) RETURN t :- .
.. RETURN nErrorType :- .

/* Calculate the result type of a unary expression */
PUBLIC FUNCTION UnaryResultType( aType, Op : short, OpFamily : short ) aType
aErrorType , _ , _ ; RETURN nNaT :- .
t:aNaT , _ , _ ; RETURN t :- .

t:aBooleanType , _ , (OpLogic) ;
t:aIntType , _ , (OpArithmetic) ;
t:aFloatType , _ , (OpArithmetic) RETURN t :- .
.. RETURN nErrorType :- .

```

```

/* Test two types for structural equivalence
   Two types are structurally equivalent iff:
   - they are the same (reference-wise)
   - or they are the same basic types
   - or they have the exact same structure */
PREDICATE TypeEqv( t1 : aType, t2 : aType )
  t
  aIntType      , aIntType      ;
  aLongType     , aLongType     ;
  aFloatType    , aFloatType    ;
  aDoubleType   , aDoubleType   ;
  aBooleanType  , aBooleanType  ;
  aArrayType( ta, .. ) , aArrayType( tb, .. ) :- TypeEqv( ta, tb ); .
  ..
  :- FAIL; .

/* Check for assignment compatibility of two types, but
   it is essential to check two structured types for
   structural equivalence */
PUBLIC PREDICATE AssignmentCompatible( target : aType, src : aType )
  t
  aIntType      , aIntType      ;
  aLongType     , aLongType     ;
  aFloatType    , aFloatType    ;
  aDoubleType   , aDoubleType   ;
  aFloatType    , aIntType      ;
  aBooleanType  , aBooleanType :- .
  aArrayType    , aArrayType    :- TypeEqv( target, src ); .
  aArrayType    , aNullType     :- .
  ..
  :- FAIL; .

/* Verify that a method call has the same number of
   parameters as the method's declaration */
PUBLIC PREDICATE CheckParamNbr( Actuels : aArguments,
                               Formels : aFormalParameters )
  aNoArgument( .. ) , aNoParameter( .. ) :- .
  aNoArgument( Pos := Pos ) , _ :-
    Message( "Too few arguments to method call.", xxError, Pos );
    FAIL; .
  aArgument( aExpression := expr ) , aNoParameter( .. ) :-
    Message( "Too many arguments to method call.",
             xxError, expr::Pos );
    FAIL .
  aArgument( .. ) , p:aFormalParameter( .. ) :-
    CheckParamNbr( a::Next, p::Next ); .

```

```

/* Check that every argument of a method call is
   assignment compatible with its corresponding formal
   parameter in the method's declaration */
PUBLIC PREDICATE ChkPrmTypes( Actuels : aArguments,
                              Formels : aFormalParameters )

  aNoArgument( .. ) , aNoParameter( .. ) :- .
  aNoArgument( .. ) , _ :- FAIL; .
  aArgument( .. ) , aNoParameter( .. ) :- FAIL; .
  aArgument( aExpression := expr ) , p:aFormalParameter( .. ) :-
    IF ( AssignmentCompatible( p::aType, expr::Type ) ) THEN
      ChkPrmTypes( a::Next, p::Next );
    ELSE
      Message( "Argument type does not match formal type.",
               xxError, expr::Pos );
      FAIL;
    END;
  .

/* Extract the next formal in a formal parameters list */
PUBLIC FUNCTION GetNextFormal( f : aFormalParameters ) aFormalParameters
  aNoParameter( .. ) RETURN f :- .
  p:aFormalParameter( .. ) RETURN p::Next :- .

/* Extract the type of a formal parameter */
PUBLIC FUNCTION GetFormalType( f : aFormalParameters ) aType
  p:aFormalParameter( .. ) RETURN p::aType :- .
  .. RETURN nNaT :- .

/* Calculate the coercion applicable between source
   and target types in an assignment */
PUBLIC FUNCTION Coerce( target : aType, src : aType ) aCoercions
  aFloatType , aIntType RETURN aIntToFloat() :- .
  .. RETURN aNoCoercion() :- .

```

```

/* Predicates, functions and procedures used by the code generator */

/* Check whether an expression is a primary */
PUBLIC PREDICATE IsPrimary( aExpression )
aPrimary( .. ) :- .
_ :- FAIL; .

/* Check for a Boolean type */
PUBLIC PREDICATE IsBoolean( aType )
aBooleanType :- .
_ :- FAIL; .

/* Generate the "load" JVM instruction corresponding
to the default initial value of a given type */
PUBLIC FUNCTION PushDefaultValue( aType ) { char* }
aIntType ;
aBooleanType RETURN "iconst_0" :- .
aFloatType RETURN "fconst_0" :- .
aReferenceType RETURN "aconst_null" :- .

/* Prefix a type to a JVM mnemonic operator */
PUBLIC FUNCTION TypedOps( aType, op : { char* } ) { char* }
LOCAL
{
static char str[40];
}

t:aType( .. ), _ RETURN str :-
strcat( strcpy( str, TypeCode( t ) ), op ); .

/* Return the JVM code for a given type */
PUBLIC FUNCTION TypeCode( aType ) { char* }
aIntType ;
aBooleanType RETURN "i" :- .
aFloatType RETURN "f" :- .
aReferenceType RETURN "a" :- .

NIL RETURN "" :- "TypeCode() Error *** NIL received..."; NL; .
_ RETURN "" :- "TypeCode() Error *** ??? received..."; NL; .

/* Prefix a type to the JVM array "load" or "store" */
PUBLIC FUNCTION ArrayTypedOps( aType, op : { char* } ) { char* }
LOCAL
{
static char str[40];
}

t:aType( .. ), _ RETURN str :-
strcat( strcpy( str, ArrayTypeCode( t ) ), op ); .

/* Return the JVM code for a given array type */
PUBLIC FUNCTION ArrayTypeCode( aType ) { char* }
aIntType RETURN "ia" :- .
aBooleanType RETURN "ba" :- .
aFloatType RETURN "fa" :- .
aReferenceType RETURN "aa" :- .

```

```

/* Encode a type (primitive and array) according to
JNI types :
- void V
- int I
- short int S
- long int J
- byte B
- char C
- boolean Z
- float F
- double D
- array [[(<primitive type code>
where there are as many "[" as there are
dimensions
- object L<Fully Qualified type name>;
where prefix qualifier id are separated by
a "/" (not yet implemented) */
PUBLIC FUNCTION EncodeType( aType ) { char* }
LOCAL
{
static char str[256];
}

t:aType( .. ) RETURN str :- strcpy( str, "" ); BuildJNI( t, str );
.

/* Add the JNI type for a given type */
PROCEDURE BuildJNI( aType, str : { char* } )
aIntType , _ :- strcat( str, "I" ); .
aBooleanType , _ :- strcat( str, "Z" ); .
aFloatType , _ :- strcat( str, "F" ); .
aVoidType , _ :- strcat( str, "V" ); .
aArrayType( ElementType := t:aType( .. ) ), _ :- strcat( str, "[" );
BuildJNI( t, str ); .

```



```

/* Encode the JNI signature of a method */
PUBLIC FUNCTION EncodeSignature( aFormalParameters, aType ) { char* }
LOCAL
{
    static char str[256];
}

f:aFormalParameters( .. ), t:aType( .. ) RETURN str :-
    strcpy( str, "(" ); PrmTypes( f, str ); strcat( str, ")" );
    BuildJNI( t, str );
.

/* Add the JNI type from the current parameter */
PROCEDURE PrmTypes( aFormalParameters, str : { char* } )

aFormalParameter(
    aType := prmType:aType( .. ),
    Next := Next:aFormalParameters( .. )
),
:- BuildJNI( prmType, str );
    PrmTypes( Next, str );
.

aNoParameter( .. ), _ :- .

/* Get the JVM mnemonic (without its type) of a
   binary or unary operator */
PUBLIC FUNCTION Ops( op : short ){ char* }
(OpMUL)    RETURN "mul" :- .
(OpDIV)    RETURN "div" :- .
(OpMOD)    RETURN "rem" :- .
(OpPLUS)   RETURN "add" :- .
(OpMINUS)  RETURN "sub" :- .

(OpAND)    RETURN "and" :- .
(OpOR)     RETURN "or"  :- .

(OpUMINUS) RETURN "neg" :- .

/* Get the method's name with the method's signature
   for a "read()" statement of java-- */
PUBLIC FUNCTION TypedRead( aType ) { char* }
aIntType    RETURN "mjoy/scanf/readI()I" :- .
aBooleanType RETURN "mjoy/scanf/readZ()Z" :- .
aFloatType  RETURN "mjoy/scanf/readF()F" :- .

```


Annexe I

Le fichier Makefile



Le fichier `Makefile` contient les commandes pour la construction du compilateur `java--` (les cibles `java--` et `all`). Il contient aussi la commande de destruction des fichiers créés (la cible `clean`) et les commandes de l'exécution du compilateur avec un jeu d'essais (la cible `tests`).

1. Extraction des symboles terminaux et des règles syntaxiques

```
Scanner.rpp Parser.lrk: java--.prs
    lpp -c -j -x -z java--.prs;
```

2. Fusion des règles lexicales

```
java--.rex: java--.scn Scanner.rpp
    rpp < java--.scn > java--.rex;
```

3. Construction de l'analyseur lexical

```
Scanner.h Scanner.c: java--.rex
    rex -c -d java--.rex;
```

4. Construction de l'analyseur syntaxique

```
Parser.h Parser.c: Parser.lrk
    lark -c -d -i -v Parser.lrk;
```

5. Construction des types de noeuds de l'arbre syntaxique abstrait et des fonctions de leur création

```
Tree.h Tree.c: java--.cg
    ast -c -d -i -m -w -R -= java--.cg
```

6. Construction de l'analyseur sémantique

```
Semantics.h Semantics.c: java--.cg
    ag -c -D -I -O -1 -2 java--.cg;
```

7. Construction du gestionnaire de la table des symboles

```
SymTab.h SymTab.c SymTab.TS: SymTab.cg
    ast -c -d -i -m -w -= -4 SymTab.cg;
```

8. Construction de la description externe de l'arbre syntaxique abstrait

```
Tree.TS: java--.cg
        echo SELECT AbstractGrammar Output | cat - java--.cg | ast -c -4
```

9. Construction du système de vérification et d'inférence de types

```
TypeSys.h TypeSys.c: TypeSys.pum Tree.TS SymTab.TS
        puma -c -d -i -k -p TypeSys.pum;
```

10. Construction du générateur de code

```
GenCode.h GenCode.c: GenCode.pum Tree.TS SymTab.TS
        puma -c -d -i GenCode.pum;
```

11. Compilation des programmes *C*

```
.c.o:
        $(CC) $(CFLAGS) -c -w $*.c;
```

12. Exécution du compilateur *java--*

```
all:      java--
          java-- test.mjv
```

```
#####
#
#   Sun's CFLAGS... "strict ansi C" == "-Xc",
#                   "heavy optimizations" == "-xO4"
#
#CFLAGS = -Xc -I${LIB}/include
#CFLAGS = -g -Xc -xO4 -I${LIB}/include
#
#   gcc CFLAGS... "ansi C" == "-ansi -pedantic",
#                   "heavy optimizations" == "-O4"
#
#CFLAGS = -g -O4 -ansi -pedantic -I${LIB}/include
#CFLAGS = -ansi -pedantic -I${LIB}/include
#
#   sun...
#JASMIN = java -jar /opt/jasmin/jasmin-2.1/jasmin.jar
#   pc ...
#JASMIN = java jasmin.Main
#
JASMIN = java -jar /opt/jasmin/jasmin-2.2/jasmin.jar
#
LIB    = ${COCKTAILHOME}/lib
CFLAGS = -I${LIB}/include
CC     = cc
#
SOURCES =      java--.c Scanner.h Scanner.c Parser.h Parser.c Tree.h Tree.c \
               Semantics.h Semantics.c SymTab.h SymTab.c TypeSys.h TypeSys.c \
               GenCode.h GenCode.c \
               JavaEscapeSeq.c JavaIntAttributes.c JavaFPAttributes.c \
               JavaStringAttributes.c
#
BINS = java--.o Scanner.o Parser.o Tree.o Semantics.o SymTab.o TypeSys.o \
        GenCode.o \
        JavaEscapeSeq.o JavaIntAttributes.o JavaFPAttributes.o \
        JavaStringAttributes.o
#
# Make all with a small test run.
#
all:      java--
          java-- test.mjv
#
# Make all with test runs.
#
tests:    java--
sh -c "java-- ./tests/ClassFrame.mjv >/dev/null 2>/dev/null"
sh -c "${JASMIN} ClassFrame.j >/dev/null 2>/dev/null"
sh -c "java ClassFrame >/dev/null 2>/dev/null"
sh -c "java-- ./tests/QuotedNames.mjv >/dev/null 2>/dev/null"
sh -c "${JASMIN} QuotedNames.j >/dev/null 2>/dev/null"
sh -c "java QuotedNames >/dev/null 2>/dev/null"
sh -c "java-- ./tests/RValLVal.mjv >/dev/null 2>/dev/null"
sh -c "${JASMIN} RValLVal.j >/dev/null 2>/dev/null"
sh -c "java RValLVal > RValLVal.rslt 2>/dev/null"
diff -bw ./tests/RValLVal.rslt RValLVal.rslt
sh -c "java-- ./tests/NPEExpr.mjv >/dev/null 2>/dev/null"
sh -c "${JASMIN} NPEExpr.j >/dev/null 2>/dev/null"
```

```
sh -c "java NPEExpr > NPEExpr.rslt 2>/dev/null"
diff -bw ./tests/NPEExpr.rslt NPEExpr.rslt
sh -c "java-- ./tests/NPArgs.mjv >/dev/null 2>/dev/null"
sh -c "${JASMIN} NPArgs.j >/dev/null 2>/dev/null"
sh -c "java NPArgs > NPArgs.rslt 2>/dev/null"
diff -bw ./tests/NPArgs.rslt NPArgs.rslt
sh -c "java-- ./tests/NPAssign.mjv >/dev/null 2>/dev/null"
sh -c "${JASMIN} NPAssign.j >/dev/null 2>/dev/null"
sh -c "java NPAssign > NPAssign.rslt 2>/dev/null"
diff -bw ./tests/NPAssign.rslt NPAssign.rslt
sh -c "java-- ./tests/NPRtnValue.mjv >/dev/null 2>/dev/null"
sh -c "${JASMIN} NPRtnValue.j >/dev/null 2>/dev/null"
sh -c "java NPRtnValue > NPRtnValue.rslt 2>/dev/null"
diff -bw ./tests/NPRtnValue.rslt NPRtnValue.rslt
sh -c "java-- ./tests/LazyEval.mjv >/dev/null 2>/dev/null"
sh -c "${JASMIN} LazyEval.j >/dev/null 2>/dev/null"
sh -c "java LazyEval > LazyEval.rslt 2>/dev/null"
diff -bw ./tests/LazyEval.rslt LazyEval.rslt
sh -c "java-- ./tests/Conditionals.mjv >/dev/null 2>/dev/null"
sh -c "${JASMIN} Conditionals.j >/dev/null 2>/dev/null"
sh -c "java Conditionals > Conditionals.rslt 2>/dev/null"
diff -bw ./tests/Conditionals.rslt Conditionals.rslt
sh -c "java-- ./tests/Read.mjv >/dev/null 2>/dev/null"
sh -c "${JASMIN} Read.j >/dev/null 2>/dev/null"
sh -c "java Read < ./tests/Read.data > Read.rslt 2>/dev/null"
diff -bw ./tests/Read.rslt Read.rslt
sh -c "java-- ./tests/SmoothSort.mjv >/dev/null 2>/dev/null"
sh -c "${JASMIN} SmoothSort.j >/dev/null 2>/dev/null"
sh -c "java SmoothSort < ./tests/SmoothSort.data > SmoothSort.rslt \
2>/dev/null"
diff -bw ./tests/SmoothSort.rslt SmoothSort.rslt
#
# Compiler assembly; link all objects into an executable.
#
java--: $(BINS)
        $(CC) $(BINS) -L${LIB} -o java-- -lreuse -lm
#
# Preprocessing (phase 1)
#
# Extract the terminal symbols from the concrete grammar (java--.prs)
# for eventual inclusion into the scanner specification (java--.rex).
# The terminal symbols are written into Scanner.rpp.
#
# Extracts production rules and reduction actions from the concrete
# grammar (java--.prs) to produce a parser specification (Parser.lrk).
#
Scanner.rpp Parser.lrk: java--.prs
        lpp -c -j -x -z java--.prs;
#
# Preprocessing (phase 2)
#
# Produce a scanner specification (java--.rex) from a description
# of complex tokens (java--.scn) and a description of simple
# tokens (Scanner.rpp).
#
java--.rex:      java--.scn Scanner.rpp
        rpp < java--.scn > java--.rex;
```

```
#
# Generate the scanner (Scanner.h and Scanner.c)
# from regular expressions (java--.rex).
#
Scanner.h Scanner.c:    java--.rex
                      rex -c -d java--.rex;

#
# Generate the parser (Parser.h and Parser.c)
# from the context-free grammar (Parser.lrk).
#
Parser.h Parser.c:      Parser.lrk
                    lark -c -d -i -v Parser.lrk;

#
# Generate the source code (Tree.h and Tree.c) that implements
# the data structures necessary to represent the AST, along with
# the routines required for their manipulation (e.g., node
# constructors, traversal functions), starting from the
# abstract grammar (java--.cg).
#
Tree.h Tree.c:  java--.cg
              ast -c -d -i -m -w -R == java--.cg

#
# Generate the attribute evaluator (Semantics.h and Semantics.c)
# from the attributed grammar (java--.cg).
#
Semantics.h Semantics.c:    java--.cg
                        ag -c -D -I -O -1 -2 java--.cg;

#
# Generate source code (SymTab.h and SymTab.c) for the
# implementation of the symbol table and environment
# data structures, along with the routines required to
# build and search these structures, from an attributed
# grammar specification (SymTab.cg). The attributed
# grammar is not required per se, but it is very useful
# if one contemplates using a "puma" specification to
# describe the code generator logic.
#
# N.B.: To use a "puma" generated code generator, an
# external description of the data structures must
# also be generated (SymTab.TS) as part of this phase.
#
SymTab.h SymTab.c SymTab.TS:    SymTab.cg
                             ast -c -d -i -m -w == -4 SymTab.cg;

#
# To allow the use a "puma" generated code generator, an
# external description of the data structures of the AST
# must also be generated (Tree.TS).
#
# N.B.: This description (used by a puma generated software
# module) need only contain the description of the AST
# data structures (i.e. attributes) along with the
# attributes necessary to support code generation (module
# Output of the attribute grammar specification).
#
Tree.TS:    java--.cg
           echo SELECT AbstractGrammar Output | cat - java--.cg | ast -c -4
```

```
#
# Generation of the source code (TypeSys.h and TypeSys.c) for
# the routines necessary to test and enforce the typing system
# rules of the compiler, starting from their "puma"
# specification (TypeSys.pum).
#
# N.B.: Use option "t" to get (on the console) a description
# of the structure of the AST that is visible to that
# software module (this is useful to specify patterns
# in the puma specification).
#
puma -c -d -i -k -p -t TypeSys.pum;
#
TypeSys.h TypeSys.c: TypeSys.pum Tree.TS SymTab.TS
                  puma -c -d -i -k -p TypeSys.pum;

#
# Generation of the source code (GenCode.h and GenCode.c) of
# the code generator of the compiler, starting from a "puma"
# specification (GenCode.pum).
#
# N.B.: Use option "t" to get (on the console) a description
# of the structure of the AST that is visible to that
# software module (this is useful to specify patterns
# in the puma specification).
#
puma -c -d -i -t GenCode.pum;
#
GenCode.h GenCode.c:    GenCode.pum Tree.TS SymTab.TS
                    puma -c -d -i GenCode.pum;

#
# C code dependencies to get the object (*.o) modules...
#
Scanner.o:    Scanner.h
Parser.o:    Scanner.h Parser.h Tree.h
Tree.o:      Tree.h
Semantics.o:  Semantics.h Tree.h
SymTab.o:    SymTab.h Tree.h
TypeSys.o:   TypeSys.h SymTab.h Tree.h
GenCode.o:   GenCode.h SymTab.h Tree.h Semantics.h
java--.o:    Scanner.h Parser.h Tree.h Semantics.h SymTab.h TypeSys.h \
              GenCode.h

JavaEscapeSeq.o:    JavaEscapeSeq.h JavaEscapeSeq.c
JavaIntAttributes.o: Scanner.h JavaIntAttributes.h JavaIntAttributes.c
JavaFPAttributes.o:  Scanner.h JavaFPAttributes.h JavaFPAttributes.c
JavaStringAttributes.o: Scanner.h JavaStringAttributes.h JavaStringAttributes.c

#
# C code compilation rule...
#
.c.o:
    $(CC) $(CFLAGS) -c -w $*.c;
```

clean:

```
rm -f Scanner.[cho] Scanner.rpp ScannerDrv.[cho] java--.rex \  
Parser.[cho] Parser.lrk Parser.dbg ParserDrv.[cho] \  
Errors.[cho] Tree.[cho] yy*.w *.TS SymTab.[cho] Semantics.[cho] \  
TypeSys.[cho] GenCode.[cho] java--.o TRACE.* *.lst *.rslt \  
yy*.h java--.exe *.stackdump core AST.* java-- *.j \  
*.class JavaEscapeSeq.o JavaIntAttributes.o JavaFPAttributes.o \  
JavaStringAttributes.o
```


Annexe J

L'environnement d'exécution *java--*

java--

L'exécution d'un programme *java--*, préalablement compilé, nécessite un environnement d'exécution. Dans son état actuel, cet environnement se présente sous la forme d'une librairie contenant une seule classe, nommée `scanf` avec huit méthodes statiques sans paramètre :

- `readS`, pour la lecture d'une donnée de type `short` ;
- `readI`, pour la lecture d'une donnée de type `int` ;
- `readJ`, pour la lecture d'une donnée de type `long` ;
- `readB`, pour la lecture d'une donnée de type `byte` ;
- `readC`, pour la lecture d'une donnée de type `char` ;
- `readZ`, pour la lecture d'une donnée de type `boolean` ;
- `readF`, pour la lecture d'une donnée de type `float` ;
- `readD`, pour la lecture d'une donnée de type `double`.

La classe `scanf` est définie dans le *package* `mjio`.

```
// Simple input from the keyboard for all primitive types. ver 1.0
// Copyright (c) Peter van der Linden, May 5 1997.
// corrected error message 11/21/97
//
// The creator of this software hereby gives you permission to:
// 1. copy the work without changing it.
// 2. modify the work providing you send me a copy which I can
// use in any way I want, including incorporating into this work.
// 3. distribute copies of the work to the public by sale, lease,
// rental, or lending.
// 4. perform the work.
// 5. display the work.
// 6. fold the work into a funny hat and wear it on your head.
//
// This is not thread safe, not high performance, and doesn't tell EOF.
// It's intended for low-volume easy keyboard input.
// An example of use is:
// EasyIn easy = new EasyIn();
// int i = easy.readInt(); // reads an int from System.in
// float f = easy.readFloat(); // reads a float from System.in
//
/*
Modified heavily by Frederic Briere <fbriere@abacom.com> for use in the
MiniJav compiler. Better error-handling was a priority.

Daniel Côté
2006-06-07: Make all static and avoid having to create a field in
the client class.
New usage in client is now:
scanf.readI();
scanf.readF(); ...
Added other methods for all Java primitive types:
scanf.readS() for short integers,
scanf.readB() for bytes,
scanf.readC() for characters,
scanf.readJ() for long integers,
scanf.readZ() for booleans,
scanf.readD() for double precision floating point numbers.
*/

package mjo;

import java.io.*;
import java.util.*;
```

```
public class scanf
{
    /* Reading from stdin */
    private static BufferedReader br
        = new BufferedReader( new InputStreamReader( System.in ) );
    private static StringTokenizer st = null; /* This holds a StringTokenizer for the current line */
    private static boolean isEOF = false; /* Flag to report end of file */

    /** Fetch a line if necessary */
    private static void fetchLine() throws IOException
    {
        String s = br.readLine();
        if ( s == null )
        {
            isEOF = true;
            st = null;
            System.err.println( "Reading beyond EOF" );
            throw new EOFException();
        }
        st = new StringTokenizer( s );
    }

    /** Get the next token, fetching lines if necessary */
    private static String getToken() throws IOException
    {
        String token = null;

        /* Although unnecessary, this avoids system calls after EOF */
        if ( isEOF ) throw new EOFException();

        if ( st == null ) fetchLine();
        while ( token == null )
        {
            try {
                token = st.nextToken();
            }
            catch ( NoSuchElementException e )
            {
                fetchLine();
            }
        }
        return( token );
    }
}
```

```

/*****
** Read a short integer
*   return 0 on any error.
*/
public static short readS()
{
    short value = 0;

    try {
        value = Short.parseShort( getToken() );
    }
    catch ( NumberFormatException e )
    {
        System.err.println( "Badly formatted short integer on input." );
    }
    catch ( EOFException e ) { }
    catch ( IOException e )
    {
        System.err.println( "IO Exception in scanf/readS()S." );
    }
    return value;
}

/*****
** Read an integer
*   return 0 on any error.
*/
public static int readI()
{
    int value = 0;

    try {
        value = Integer.parseInt( getToken() );
    }
    catch ( NumberFormatException e )
    {
        System.err.println( "Badly formatted integer on input." );
    }
    catch ( EOFException e ) { }
    catch ( IOException e )
    {
        System.err.println( "IO Exception in scanf/readI()I." );
    }
    return value;
}

```

```

/*****
** Read a long integer
*   return 0 on any error.
*/
public static long readJ()
{
    long value = 0L;

    try {
        value = Long.parseLong( getToken() );
    }
    catch ( NumberFormatException e )
    {
        System.err.println( "Badly formatted long integer on input." );
    }
    catch ( EOFException e ) { }
    catch ( IOException e )
    {
        System.err.println( "IO Exception in scanf/readJ()J." );
    }
    return value;
}

/*****
** Read a byte
*   return 0 on any error.
*/
public static byte readB()
{
    byte value = 0;

    try {
        value = Byte.parseByte( getToken() );
    }
    catch ( NumberFormatException e )
    {
        System.err.println( "Badly formatted byte on input." );
    }
    catch ( EOFException e ) { }
    catch ( IOException e )
    {
        System.err.println( "IO Exception in scanf/readB()B." );
    }
    return value;
}

```

```

/*****
** Read a char
*   return 0 on any error.
*/
public static char readC()
{
    char value = 0;

    try {
        String tk = getToken();
        value = ( tk == null || tk.length() == 0 ) ? 0 : tk.charAt( 0 );
    }
    catch ( EOFException e ) { }
    catch ( IOException e )
    {
        System.err.println( "IO Exception in scanf/readC()C." );
    }
    return value;
}

/*****
** Read a boolean
*   return false on any error.
*/
public static boolean readZ()
{
    boolean value = false;

    try {
        value = ( new Boolean( getToken() ) ).booleanValue();
    }
    catch ( EOFException e ) { }
    catch ( IOException e )
    {
        System.err.println( "IO Exception in scanf/readZ()Z." );
    }
    return value;
}

```

```

/*****
** Read a floating point number
*   return 0.0F on any error.
*/
public static float readF()
{
    float value = 0.0F;

    try {
        value = Float.parseFloat( getToken() );
    }
    catch ( NumberFormatException e )
    {
        System.err.println( "Badly formatted floating-point on input." );
    }
    catch ( EOFException e ) { }
    catch ( IOException e )
    {
        System.err.println( "IO Exception in scanf/readF()F." );
    }
    return value;
}

/*****
** Read a double precision floating point number
**   return 0.0D on any error.
*/
public static double readD()
{
    double value = 0.0D;

    try {
        value = Double.parseDouble( getToken() );
    }
    catch ( NumberFormatException e )
    {
        System.err.println( "Badly formatted double precision" );
        System.err.println( "floating-point on input." );
    }
    catch ( EOFException e ) { }
    catch ( IOException e )
    {
        System.err.println( "IO Exception in scanf/readD()D." );
    }
    return value;
}
}

```

Annexe K

Un sous-ensemble du langage JasminXT



Cette annexe contient la grammaire hors contexte d'un sous-ensemble du langage JasminXT ainsi qu'une brève description des instructions utilisées par *java--*.

K.1 Grammaire hors contexte

```
jas_file   → jasmin_header{field}{method}
jasmin_header → [.source sourcefile]
               .class {access_spec} class_name
               .super class_name
access_spec → public | static
class_name  → name {/name}
field      → .field {access_spec} field_name descriptor [= value]
method     → .method
               {access_spec} method_name descriptor {statement}
               .end method
statement  → .limit stack size
               | .limit locals size
               | .line number
               | .var var_number is var_name descriptor from label1 to label2
               | instruction [instruction_args]
               | label :
```

K.2 Quelques instructions

K.2.1 Instruction de branchement inconditionnel

<code>goto</code>	Branchement
-------------------	-------------

K.2.2 Instructions de branchement conditionnel à un opérande de type référence

<code>ifnull</code>	Branchement si égal à <code>null</code>
<code>ifnonnull</code>	Branchement si différent de <code>null</code>

K.2.3 Instructions de branchement conditionnel à deux opérandes de type référence

<code>if_acmpeq</code>	Branchement si égal
<code>if_acmpne</code>	Branchement si différent

K.2.4 Instructions de branchement conditionnel à un opérande entier

<code>ifeq</code>	Branchement si égal à 0
<code>ifne</code>	Branchement si différent de 0
<code>iflt</code>	Branchement si plus petit que 0
<code>ifle</code>	Branchement si plus petit ou égal à 0
<code>ifgt</code>	Branchement si plus grand que 0
<code>ifge</code>	Branchement si plus grand ou égal à 0

K.2.5 Instructions de branchement conditionnel à deux opérandes entiers

<code>if_icmpeq</code>	Branchement si égal
<code>if_icmpne</code>	Branchement si différent
<code>if_icmplt</code>	Branchement si plus petit
<code>if_icmple</code>	Branchement si plus petit ou égal
<code>if_icmpgt</code>	Branchement si plus grand
<code>if_icmpge</code>	Branchement si plus grand ou égal

K.2.6 Instructions de comparaison

<code>fcmpl</code>	Comparaison de deux données de type <code>float</code> (−1 pour NaN)
<code>fcmpg</code>	Comparaison de deux données de type <code>float</code> (1 pour NaN)

K.2.7 Instructions arithmétiques entières

iadd	Addition de deux données de type <code>int</code>
isub	Différence entre deux données de type <code>int</code>
imul	Multiplication de deux données de type <code>int</code>
idiv	Quotient de deux données de type <code>int</code>
irem	Reste d'une division entière
ineg	Complément d'une donnée de type <code>int</code>

K.2.8 Instructions arithmétiques en point flottant

fadd	Addition de deux données de type <code>float</code>
fsub	Différence entre deux données de type <code>float</code>
fmul	Multiplication de deux données de type <code>float</code>
fdiv	Quotient de deux données de type <code>float</code>
frem	Reste d'une division en point flottant
fneg	Complément d'une donnée de type <code>float</code>

K.2.9 Instructions de chargement d'une constante

aconst_null	Chargement de la constante <code>null</code>
fconst_0	Chargement de la constante <code>0.0E0F</code>
iconst_0	Chargement de la constante <code>0</code> (ou <code>false</code>)
iconst_1	Chargement de la constante <code>1</code> (ou <code>true</code>)
ldc	Chargement d'une constante

K.2.10 Instructions de chargement de la valeur d'une donnée du bloc des variables locales

aload_0	Chargement de la valeur de la variable <code>0</code> de type <code>reference</code>
aload	Chargement de la valeur d'une donnée de type <code>reference</code>
fload	Chargement de la valeur d'une donnée de type <code>float</code>
iload	Chargement de la valeur d'une donnée de type <code>int</code>

K.2.11 Instructions de stockage de la valeur d'une donnée dans le bloc des variables locales

astore	Stockage de la valeur d'une donnée de type <code>reference</code>
fstore	Stockage de la valeur d'une donnée de type <code>float</code>
istore	Stockage de la valeur d'une donnée de type <code>int</code>

K.2.12 Instructions de chargement de la valeur d'une composante d'un tableau

<code>aaload</code>	Chargement de la valeur d'une composante de type <code>reference</code>
<code>baload</code>	Chargement de la valeur d'une composante de type <code>byte</code>
<code>faload</code>	Chargement de la valeur d'une composante de type <code>float</code>
<code>iaload</code>	Chargement de la valeur d'une composante de type <code>int</code>

K.2.13 Instructions de stockage de la valeur d'une donnée dans une composante d'un tableau

<code>aastore</code>	Stockage de la valeur d'une donnée de type <code>reference</code> dans un tableau
<code>bastore</code>	Stockage de la valeur d'une donnée de type <code>byte</code> dans un tableau
<code>fastore</code>	Stockage de la valeur d'une donnée de type <code>float</code> dans un tableau
<code>iastore</code>	Stockage de la valeur d'une donnée de type <code>int</code> dans un tableau

K.2.14 Instructions d'accès à un champ de classe

<code>getfield</code>	Extraction de la valeur d'un champ d'instance
<code>getstatic</code>	Extraction de la valeur d'un champ de classe
<code>putfield</code>	Positionnement de la valeur d'un champ de classe
<code>putstatic</code>	Positionnement de la valeur d'un champ d'instance

K.2.15 Instructions d'appel d'une méthode

<code>invokestatic</code>	Appel d'une méthode de classe
<code>invokevirtual</code>	Appel d'une méthode d'instance
<code>invokenonvirtual</code>	Appel d'un constructeur

K.2.16 Instructions de retour d'une méthode

<code>return</code>	Retour vers l'appelant, mais sans résultat
<code>areturn</code>	Retour vers l'appelant, avec résultat de type <code>reference</code>
<code>freturn</code>	Retour vers l'appelant, avec résultat de type <code>float</code>
<code>ireturn</code>	Retour vers l'appelant, avec résultat de type <code>int</code>

K.2.17 Instruction de création d'un tableau

<code>multianewarray</code>	Création d'un tableau
-----------------------------	-----------------------

K.2.18 Instruction de conversion de type

<code>i2f</code>	Conversion de <code>int</code> à <code>float</code>
------------------	---

K.2.19 Instructions de manipulation de la pile de travail

pop	Retrait de la valeur au sommet de la pile de travail
pop2	Retrait des deux valeurs au sommet de la pile de travail

Bibliographie

- [1] A. V. Aho, R. Sethi et J. D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.
- [2] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Third Edition*. The Java Series, Sun microsystems, 2005. <http://java.sun.com/docs/books/jls/>.
- [3] J. Grosch. *Ag — An Attribute Evaluator Generator*. Document No. 16, CoCoLab — Datenverarbeitung, 2002, 38 pages.
- [4] J. Grosch. *Ast — A Generator for Abstract Syntax Trees*. Document No. 15, CoCoLab — Datenverarbeitung, 2002, 74 pages.
- [5] J. Grosch. *Lark — An LALR(2) Parser Generator with Backtracking*. Document No. 32, CoCoLab — Datenverarbeitung, 2002, 90 pages.
- [6] J. Grosch. *Preprocessors*. Document No. 24, CoCoLab — Datenverarbeitung, 2000, 17 pages.
- [7] J. Grosch. *Puma — A Generator for the Transformation of Attributed Trees*. Document No. 26, CoCoLab — Datenverarbeitung, 2002, 39 pages.
- [8] J. Grosch. *Reusable Software — A Collection of C-Modules*. Document No. 30, CoCoLab — Datenverarbeitung, 2000, 31 pages.
- [9] J. Grosch. *Rex — A Scanner Generator*. Document No. 5, CoCoLab — Datenverarbeitung, 2000, 57 pages.
- [10] J. Grosch. *Toolbox Introduction*. Document No. 25, CoCoLab — Datenverarbeitung, 2002, 14 pages.
- [11] J. Grosch et H. Emmelmann. *A Tool Box for Compiler Construction*. Document No. 20, CoCoLab — Datenverarbeitung, 1990, 11 pages.
- [12] D. Grune, H. E. Bal, C. J. H. Jacobs et K. G. Langendoen. *Modern Compiler Design*. Wiley, Chichester, 2000
- [13] J. Meyer and D. Reynaud. *Jasmin home page*. <http://jasmin.sourceforge.net/>.
- [14] B. Venners. *Inside The Java 2 Virtual Machine*. McGraw-Hill, 1999. <http://www.artima.com/insidejvm/ed2/>.