Projet Génie Logiciel 2^e ann\'ee

version avec hyperliens en couleur

Grenoble INP – Ensimag

Décembre 2024

Table des matières

Ι	Organisation et évaluation du projet	9
[Int	roduction] Introduction au projet Génie Logiciel	11
1	Buts	11
2	2 Comment lire ce document?	11
3	3 Vue globale du travail à réaliser	12
4	Organisation temporelle	12
[M o	otivation] Intérêt de ce projet dans le cursus et place dans votre formation	13
[Exe	empleSansObjet] Exemple introductif « sans objet »	17
1	Étape d'analyse syntaxique	17
2	Étape de vérifications contextuelles et décorations	18
3	B Étape de génération de code	19
4	Définition de la partie « sans objet »	19
Re	nduIntermediaire Rendu Intermédiaire	23
· .	•	23
2	•	23
3	Barème indicatif	23
[A -]	Rendre Description des produits du rendu final	25
		25
2	Liste des produits à rendre (contenu attendu du dépôt Git)	27
3	•	27
[Sot	itenance] Soutenance du projet Génie Logiciel	29
[Co	mmunication] Communication à l'écrit et à l'oral	31
[Sui	vi-SHEME-I] Charte de travail en équipe (livrable suivi SHEME 1)	33
[Co	mmunicationEnseignants] Protocole de communication enseignants-étudiants	37
1	La page du projet « Actualité du Projet GL »	37
2	Site web du projet	37
3		37
[Introduction] Introduction au projet Génie Logiciel 1 Buts 2 Comment lire ce document? 3 Vue globale du travail à réaliser 4 Organisation temporelle [Motivation] Intérêt de ce projet dans le cursus et place dans votre formation [ExempleSansObjet] Exemple introductif « sans objet » 1 Étape d'analyse syntaxique 2 Étape de yénifications contextuelles et décorations 3 Étape de génération de code. 4 Définition de la partie « sans objet » [RenduIntermediaire] Rendu Intermédiaire 1 Consignes à respecter 2 Conseils 3 Barème indicatif [A-Rendre] Description des produits du rendu final 1 Documents à rendre 2 Liste des produits à rendre (contenu attendu du dépôt Git) 3 Note sur la qualité du compilateur rendu. [Soutenance] Soutenance du projet Génie Logiciel [Communication] Communication à l'écrit et à l'oral [Suivi-SHEME-I] Charte de travail en équipe (livrable suivi SHEME 1) [CommunicationEnseignants] Protocole de communication enseignants-étudiants 1 La page du projet « Actualité du Projet GL » 2 Site web du projet 3 Le courrier électronique [Suivis] Contenu des séances de suivi du projet Génie Logiciel [Consignes] Consignes et conseils pour l'implémentation du compilateur Deca et de sa base de tests 1 Le compilateur decac 2 Étape A: Analyse lexicale et syntaxique 3 Étape B: Vérifications contextuelles	39	
[Co	nsignes] Consignes et conseils pour l'implémentation du compilateur Deca et de sa base	е
C	de tests	43
1	Le compilateur decac	43
2	Étape A : Analyse lexicale et syntaxique	45
3	Etape B : Vérifications contextuelles	46
4	Étape C : Génération de code	47

5	Consignes particulières pour l'extension TRIGO	49
II S _I	pécifications du compilateur decac	51
[Lexico	ographie] Lexicographie de Deca	53
[Synta	xe] Syntaxe concrète de Deca	57
[Synta	xeAbstraite] Syntaxe abstraite de Deca	61
1	Notations de la grammaire des arbres de syntaxe abstraite	61
2	Grammaire des arbres de syntaxe abstraite	62
Decor	npilation] Décompilation des arbres de syntaxe abstraite	65
1	Notations de la grammaire attribuée de décompilation	65
$\overset{-}{2}$	Correspondance entre la syntaxe abstraite et la syntaxe concrète	66
3	Spécification de l'option -p	67
4	Grammaire attribuée de décompilation	68
ra .		
• •	xeContextuelle] Syntaxe contextuelle du langage Deca	73
1	Introduction	73
2	Domaines d'attributs	73
3	Conventions d'écriture	77
4	Règles communes aux trois passes de vérifications contextuelles	79
5	Grammaire attribuée spécifiant la passe 1	80
6	Grammaire attribuée spécifiant la passe 2	80
7	Grammaire attribuée spécifiant la passe 3	82
8	Profils d'attributs des symboles non-terminaux et terminaux	89
9 10	Note sur les champs protégés	91 92
10	Implementation de l'environnement	92
Biblio	thequeStandard] Bibliothèque standard du langage Deca	97
1	Recherche de fichiers dans la bibliothèque standard	97
2	Extension TRIGO: le fichier Math.decah	97
3	Autres extensions utilisant la bibliothèque standard	98
[C		00
•	ntique] Sémantique de Deca	99
$\frac{1}{2}$	Initialisation des variables et champs	99 99
$\frac{2}{3}$	Instruction « new »	99
3 4	Instruction « return »	100
5	Débordements lors de l'évaluation des expressions	100
6	Procédures d'affichage	100
7	Appels de méthodes	100
8	Méthode « equals » de la classe Object	100
9	Opérateur de conversion de type : (type) (valeur)	101
10	Méthodes écrites en assembleur	101
11	Liste des catégories d'erreurs à l'exécution	101
		-01
[Decad	e] Description du compilateur decac	103
1	Ligne de commande	103
2	Formatage des messages d'erreur	104

[Mac	hine Abstraite Définition de la Machine Abstraite et de son langage d'assemblage	105
1	Données et mémoires	105
2	Modes d'adressages	106
3	Instructions	106
4	Syntaxe du langage d'assemblage	111
5	Exemple de programme assembleur : la factorielle récursive	111
[Con	ventionsLiaison] Conventions de liaison pour la Machine Abstraite	113
1	État de la pile lors de l'exécution du programme principal	113
2	Bloc d'activation d'une méthode	113
3	Nommage des étiquettes	113
III	Compléments sur les outils et les méthodes de Génie Logiciel	115
[Envi	ironnement] Environnement de développement pour le projet Génie Logiciel	117
1	Organisation en répertoires	117
2	Travail en parallèle et gestion de versions	117
3	Utilisation de Maven, fichier pom.xml	119
4	Résumé des commandes utiles pour le projet	122
[Sean	nceMachine] Séance de prise en main de l'environnement du projet	125
1	Lire l'actualité du projet sur le site du Projet GL	125
2	Création des répertoires du projet	125
3	Modification du path et des variables d'environnement	126
4	Compilation (avec gestion des dépendances) et exécution	127
5	Choix et prise en main d'un IDE	129
6	Utilisation de Git	132
7	Tout nettoyer	133
8	Regarder l'exemple ANTLR (Calculette)	133
9	Tester l'outil de planification planner	133
10	Travailler sur l'analyseur lexical	134
[Git]	Utilisation basique de Git	135
[Test	s] Validation du compilateur Deca	143
1	Conseils généraux sur les tests	143
2	Gestion des risques et gestion des rendus	150
3	Consignes à respecter impérativement	151
[Jaco	co] Utilisation de Jacoco pour la mesure de couverture	155
1	Découverte et utilisation manuelle de Jacoco	155
2	Couverture pour les tests automatisés	156
3	Méthode	156
4	Remarque	156
•	grammationDefensive] Programmation défensive	159
1	Introduction	159
2	Test de précondition	159
3	Vérification des post-conditions et des invariants	160
4	Remarque sur la récupération des exceptions	161
5	Coût de la programmation défensive	162

	ventionsCodage Conventions et Styles de Codage
1	Introduction
2	Types abstraits de données
3	Règles de mise en forme du code
4	Fonctionnalités du langage Java
5	Affichages, traces, et debug
6	Représentation de l'arbre abstrait avec le patron « interprète »
Guid	lePlanner] Outil « Planner » pour gnome
1	Définir le calendrier
2	Création de ressource
3	Création des tâches
4	Création des dépendances
5	Exportation
Fran	de] Fraude, propriété intellectuelle et logiciels
Frau 1	Introduction
$\frac{1}{2}$	
	Protection du logiciel
3	Notion d'auteur du logiciel
4	Le titulaire des droits
5	Les droits d'auteur
6	L'exploitation des droits patrimoniaux
7	Cas d'exploitation spécifique : les logiciels libres
8	La défense des droits
. V	Compléments sur la compilation du langage Deca
Exer	nple Exemple avec objet illustrant les étapes de compilation
Exer	nple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique
Exer	nple Exemple avec objet illustrant les étapes de compilation
Exer 1 2 3	nple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique
Exer 1 2 3 (AN)	nple Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code
Exer 1 2 3 (ANT) 1	nple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique
Exer 1 2 3 [ANT 1 2	nple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code CLR] ANTLR: ANother Tool for Language Recognition ANTLR: vue d'ensemble Structure d'un fichier source ANTLR
[Exer	nple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code CLR] ANTLR : ANother Tool for Language Recognition ANTLR : vue d'ensemble Structure d'un fichier source ANTLR Analyse lexicale (Lexer) avec ANTLR
Exer 1 2 3 ANT 1 2 3 4	nple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code CLR] ANTLR: ANother Tool for Language Recognition ANTLR: vue d'ensemble Structure d'un fichier source ANTLR Analyse lexicale (Lexer) avec ANTLR Analyse syntaxique (Parser) avec ANTLR
Exer 1 2 3 ANT 1 2 3	nple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code CLR] ANTLR : ANother Tool for Language Recognition ANTLR : vue d'ensemble Structure d'un fichier source ANTLR Analyse lexicale (Lexer) avec ANTLR
Exer 1 2 3 (ANT) 1 2 3 4 5	mple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code [LR] ANTLR : ANother Tool for Language Recognition ANTLR : vue d'ensemble Structure d'un fichier source ANTLR Analyse lexicale (Lexer) avec ANTLR Analyse syntaxique (Parser) avec ANTLR Exemple de programme utilisant ANTLR : calculette
Exer 1 2 3 (ANT) 1 2 3 4 5	mple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code CLR] ANTLR : ANother Tool for Language Recognition ANTLR : vue d'ensemble Structure d'un fichier source ANTLR Analyse lexicale (Lexer) avec ANTLR Analyse syntaxique (Parser) avec ANTLR Exemple de programme utilisant ANTLR : calculette
Exer 1 2 3 (ANT) 1 2 3 4 5	mple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code LTR] ANTLR: ANother Tool for Language Recognition ANTLR: vue d'ensemble Structure d'un fichier source ANTLR Analyse lexicale (Lexer) avec ANTLR Analyse syntaxique (Parser) avec ANTLR Exemple de programme utilisant ANTLR: calculette reEnrichi] Arbres enrichis et décorés Enrichissement avec le nœud ConvFloat Les décors
Exer 1 2 3 ANT 1 2 3 4 5 Arbi	nple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code CLR] ANTLR: ANother Tool for Language Recognition ANTLR: vue d'ensemble Structure d'un fichier source ANTLR Analyse lexicale (Lexer) avec ANTLR Analyse syntaxique (Parser) avec ANTLR Exemple de programme utilisant ANTLR: calculette reEnrichi] Arbres enrichis et décorés Enrichissement avec le nœud ConvFloat
Exer 1 2 3 ANT 1 2 3 4 5 Arbi 1 2 3	nple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code LTR] ANTLR: ANother Tool for Language Recognition ANTLR: vue d'ensemble Structure d'un fichier source ANTLR Analyse lexicale (Lexer) avec ANTLR Analyse syntaxique (Parser) avec ANTLR Exemple de programme utilisant ANTLR: calculette reEnrichi] Arbres enrichis et décorés Enrichissement avec le nœud ConvFloat Les décors Un exemple complet
Exer 1 2 3 ANT 1 2 3 4 5 Arbi 1 2 3	nple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code CLR] ANTLR: ANother Tool for Language Recognition ANTLR: vue d'ensemble Structure d'un fichier source ANTLR Analyse lexicale (Lexer) avec ANTLR Analyse syntaxique (Parser) avec ANTLR Exemple de programme utilisant ANTLR: calculette reEnrichi] Arbres enrichis et décorés Enrichissement avec le nœud ConvFloat Les décors Un exemple complet code] Génération de code pour le langage Deca
Exer 1 2 3 (ANT) 1 2 3 4 5 (Arbital) 1 2 3 (Gene) 1	nple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code . CLR] ANTLR : ANother Tool for Language Recognition ANTLR : vue d'ensemble Structure d'un fichier source ANTLR Analyse lexicale (Lexer) avec ANTLR Analyse syntaxique (Parser) avec ANTLR Exemple de programme utilisant ANTLR : calculette reEnrichi] Arbres enrichis et décorés Enrichissement avec le nœud ConvFloat Les décors Un exemple complet code] Génération de code pour le langage Deca
Exer 1 2 3 (ANT) 1 2 3 4 5 (Arbital) 1 2 3 (Genoted)	mple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code CLR] ANTLR: ANother Tool for Language Recognition ANTLR: vue d'ensemble Structure d'un fichier source ANTLR Analyse lexicale (Lexer) avec ANTLR Analyse syntaxique (Parser) avec ANTLR Exemple de programme utilisant ANTLR: calculette reEnrichi] Arbres enrichis et décorés Enrichissement avec le nœud ConvFloat Les décors Un exemple complet code] Génération de code pour le langage Deca Génération de code pour le langage Deca « sans objet » Génération de code pour le langage Deca complet
[Exer 1 2 3 4 5 5 [Arbit 2 3 3 4 5 5 [General 2 3 3 4 5]	mple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code CLR] ANTLR : ANother Tool for Language Recognition ANTLR : vue d'ensemble Structure d'un fichier source ANTLR Analyse lexicale (Lexer) avec ANTLR Analyse syntaxique (Parser) avec ANTLR Exemple de programme utilisant ANTLR : calculette reEnrichi] Arbres enrichis et décorés Enrichissement avec le nœud ConvFloat Les décors Un exemple complet code] Génération de code pour le langage Deca Génération de code pour le langage Deca « sans objet » Génération de code pour le langage Deca complet Construction de la table des méthodes
[Exer 1 2 3 3 4 5 5 [Arbit 1 2 3 3 4 5 3 4 4 5 3 4 4 5]	inple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code. LR] ANTLR: ANother Tool for Language Recognition ANTLR: vue d'ensemble Structure d'un fichier source ANTLR Analyse lexicale (Lexer) avec ANTLR Analyse syntaxique (Parser) avec ANTLR Exemple de programme utilisant ANTLR: calculette reEnrichi] Arbres enrichis et décorés Enrichissement avec le nœud ConvFloat Les décors Un exemple complet code] Génération de code pour le langage Deca Génération de code pour le langage Deca Génération de code pour le langage Deca complet Construction de la table des méthodes Codage des champs
[Exer 1 2 3 4 5 5 [Arbit 1 2 3 4 4 5 5 4 5 5]	inple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code ELR] ANTLR: ANother Tool for Language Recognition ANTLR: vue d'ensemble Structure d'un fichier source ANTLR Analyse lexicale (Lexer) avec ANTLR Analyse syntaxique (Parser) avec ANTLR Exemple de programme utilisant ANTLR: calculette reEnrichi] Arbres enrichis et décorés Enrichissement avec le nœud ConvFloat Les décors Un exemple complet code] Génération de code pour le langage Deca Génération de code pour le langage Deca « sans objet » Génération de code pour le langage Deca complet Construction de la table des méthodes Codage des champs Codage des méthodes
[Exer 1 2 3 4 5 5 [Arbi 1 2 3 4 5 5 6 6]	Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code CLR] ANTLR : ANother Tool for Language Recognition ANTLR : vue d'ensemble Structure d'un fichier source ANTLR Analyse lexicale (Lexer) avec ANTLR Analyse syntaxique (Parser) avec ANTLR Exemple de programme utilisant ANTLR : calculette reEnrichi] Arbres enrichis et décorés Enrichissement avec le nœud ConvFloat Les décors Un exemple complet code] Génération de code pour le langage Deca Génération de code pour le langage Deca « sans objet » Génération de code pour le langage Deca complet Construction de la table des méthodes Codage des champs Codage des méthodes Codage des déclarations
Exer 1 2 3 ANT 1 2 3 4 5 Arbi 1 2 3 Geno 1 2 3 4 5	inple] Exemple avec objet illustrant les étapes de compilation Étape d'analyse syntaxique Étape de vérifications contextuelles et décorations Étape de génération de code ELR] ANTLR: ANother Tool for Language Recognition ANTLR: vue d'ensemble Structure d'un fichier source ANTLR Analyse lexicale (Lexer) avec ANTLR Analyse syntaxique (Parser) avec ANTLR Exemple de programme utilisant ANTLR: calculette reEnrichi] Arbres enrichis et décorés Enrichissement avec le nœud ConvFloat Les décors Un exemple complet code] Génération de code pour le langage Deca Génération de code pour le langage Deca « sans objet » Génération de code pour le langage Deca complet Construction de la table des méthodes Codage des champs Codage des méthodes

[Ima]	Descriptif d'utilisation de l'interpréteur ima	227
1	Appel de ima	227
2	Utilisation de ima en mode metteur au point	227

Première partie

Organisation et évaluation du projet

[Introduction]

Introduction au projet Génie Logiciel

1 Buts

- Écrire en Java un compilateur pour le langage Deca (petit langage ressemblant à Java) : adopter le point de vue d'un compilateur permet de mieux comprendre les langages de programmation (et de progresser en programmation).
- Adopter une démarche qualité, avec un objectif de produire du code « zéro défaut », en développant une base de tests capable d'évaluer la conformité de ce compilateur à la spécification fournie dans la doc.
- Mettre en place une organisation projet et une démarche de développement, en sachant dès le départ que le projet est irréalisable parfaitement dans le temps imparti, et qu'il faudra donc adapter au fur et à mesure l'organisation et les objectifs en fonction des problèmes rencontrés.
- Utiliser des outils d'aide au développement : Maven, ANTLR, Git, Jacoco, un IDE (NetBeans, Eclipse, . . .).
- Comprendre la façon dont les calculs sont traduits par les machines, par exemple sur les flottants.
- Expérimenter des techniques agiles de développement : développement dirigé par les tests, intégration continue, programmation par paires, etc.
- Comprendre et respecter des spécifications, formelles (grammaires attribuées) ou non.
- Mettre en place une organisation projet et la faire évoluer en fonction des problèmes rencontrés.

Plus de détails sur les buts du projet sont donnés dans le document [Motivation].

2 Comment lire ce document?

Ce document est constitué d'une collection de sous-documents de référence détaillant un aspect du projet GL. Pour gagner du temps en première lecture, il est conseillé de ne pas chercher à comprendre chaque document en détail mais plutôt de chercher d'abord à avoir une vue globale du projet. Chaque sous-document pourra être étudié en détail le moment venu (plusieurs relectures des sous-documents seront sans doute nécessaires).

Pour naviguer plus rapidement dans la version électronique du document, utilisez

- l'affichage du « panneau latéral » (ou « sommaire ») qui donne la table des matières dans une colonne à gauche.
- les hyperliens en couleurs, par exemple pour aller à la définition du langage de lexèmes STRING ou du non-terminal expr de la grammaire concrète ou encore pour aller à la figure 1 du document [ExempleSansObjet]. Il est souvent commode d'ouvrir l'hyperlien dans une nouvelle fenêtre à partir du menu contextuel (obtenu en cliquant sur le bouton droit).
- la fonction de recherche du visualisateur pdf (raccourci Ctrl+F). Par exemple : recherchez « [Consignes] » pour aller rapidement au document [Consignes].
- les touches de navigation du visualisateur (e.g. Ctrl+Début pour retourner au début du document vers la table des matières).

Pensez aussi qu'il est aussi possible de copier-coller du texte depuis les documents pdf.

[Introduction] 12

3 Vue globale du travail à réaliser

Pour lire ce document, il est essentiel d'avoir compris la structure du compilateur en 3 étapes :

« étape A » Analyseur syntaxique

entrée : programme source sortie : arbre abstrait primitif

« étape B » Vérificateur de la syntaxe contextuelle

entrée : arbre abstrait primitif sortie : arbre abstrait décoré

« étape C » Générateur de code pour une machine abstraite

entrée : arbre abstrait décoré

sortie : programme en langage d'assemblage

Vous pouvez donc commencer par survoler le document [ExempleSansObjet] qui donne une vue détaillée du fonctionnement de ces 3 étapes sur un exemple très simple. Ensuite il est conseillé de lire les documents : [A-Rendre] et [Consignes].

4 Organisation temporelle

- 1. 6h40 de vidéos de présentation du projet
- 2. 6h00 heures de Cours/TD et une séance de 1h30 de prise en main du projet sur machine
- 3. 3 séances de suivi d'une demi-heure par équipe
- 4. Analyse, conception, développement, validation (y compris extension) jusqu'au :
 - => lundi 20 janvier 2025 à 16h00
- 5. Rendu intermédiaire (langage sans-objet) :
 - => lundi 13 janvier 2025 à 12h00
- 6. Récupération des projets (programmes et tests du compilateur et de l'extension) :
 - => lundi 20 janvier 2025 à 16h00
- 7. Rendu de la documentation utilisateur :
 - => lundi 20 janvier 2025 à 20h00
- 8. Rétrospectives collectives :
 - => mardi 21 janvier 2025
- 9. Rendu du bilan de gestion d'équipe et de projet :
 - => mercredi 22 janvier 2025 à 9h00
- 10. Rendu des documentations de conception, de validation, d'analyse énergétique et de l'extension:
 - => le jour de la soutenance
- 11. Soutenances:
 - => jeudi 25 janvier et vendredi 26 janvier 2024.

Intérêt de ce projet dans le cursus et place dans votre formation

Le projet de Génie Logiciel occupe une place centrale dans la formation de l'Ensimag : à la fois par sa position au milieu du cursus de 3 ans, mais aussi par l'intégration des compétences qu'il développe qui sont emblématiques de l'école, ainsi que des individus, et des différentes filières. Tous ceux qui revendiquent le diplôme passent par ce projet, même lorsqu'ils rejoignent l'Ensimag après 3 ans de formation à l'Ecole Polytechnique ou une admission sur titres en 2A, ou lors d'une convention de double diplôme avec un institut étranger.

Cette place lui est largement reconnue par les anciens diplômés : c'est l'élément principal qu'ils citent tous dans les retours positifs qu'ils font sur la formation qu'ils ont reçue. Projet initiatique donc, parfois objet d'appréhension a priori de la part d'étudiants qui craignent d'y voir un test de passage imposé conçu pour des geeks.

Voici donc quelques éléments de justification pour expliquer la place centrale et les principales dimensions de ce projet ainsi que les motivations des choix pédagogiques faits dans sa conception.

Dimension 1 : le travail en mode projet

Si la scolarité antérieure permet souvent de réaliser quelques travaux en équipe (binômes etc.), le projet GL est un apprentissage d'un travail réalisé dans un cadre plus réaliste par rapport à la vie professionnelle, incluant un cahier des charges complexe, un certain stress avec une quantité de travail significative et des échéances rapprochées, et la gestion des relations inter-individuelles dans ce contexte.

Autrement dit, le projet GL nécessite de travailler en mode projet, de façon similaire au milieu professionnel : définition d'une organisation d'équipe (qui fait quoi ?), d'une organisation dans le temps (planification des tâches et suivi de leur avancement), reportings du projet, gestion des priorités (qualité vs. quantité des livrables), utilisation d'une méthode (Scrum ou classique) et d'outils de gestion de projet (Trello, Gitlab ou logiciels Gantt) .

Dimension 2 : compilateur et complexité du calcul

Des étudiants se demandent souvent pourquoi ils doivent produire un compilateur, exercice que peu auront à faire ultérieurement. En fait, la complexité des niveaux de représentation (exécution, code machine, niveaux lexicaux, syntaxiques et sémantiques), des structures de données et les abstractions correspondantes (niveau concret, niveaux méta) est un excellent moyen de réfléchir en profondeur à ce qu'est une automatisation et ce qu'est le calcul que peut faire une machine. En sortant de là, les étudiants Ensimag comprennent beaucoup mieux ce qui est automatisable et ce qui ne peut l'être, et valorisent leurs capacités d'abstraction mathématique (manipulation de concepts abstraits mais très bien définis) dans un contexte de manipulation de structures d'information.

En outre, la problématique d'analyse et d'interprétation de données structurées, dont la compilation est un archétype, se retrouve dans beaucoup de domaines où vous pourriez travailler ensuite, que ce soit pour analyser des flux vidéos en MPEG4, des bases de clientèles issues d'un CRM (système de gestion des relations clients d'une entreprise), des flux de données issues de transactions etc.

Dimension 3 : Génie Logiciel

Si dans ce projet il y a certes une part de programmation, c'est surtout l'occasion d'aborder réellement du génie logiciel, par l'apprentissage de méthodes de développement et par l'importance accordée au test et à la qualité du logiciel.

Dimension 4 : évaluation et négociation de la complexité d'un développement

Une partie significative du projet (le développement du compilateur Deca pour la machine ima) est très bien définie, avec des attentes très précises pour évaluer vos compétences de base. Cependant, près d'un quart du projet, appelé "extension" est une partie beaucoup plus ouverte, que vous serez amenés à définir et négocier dans le cadre d'un dialogue avec vos enseignants. Cela vous fera aborder des activités représentatives de ce que peut faire une entreprise pour rencontrer son marché, ses clients, avec la prise de risque associé, et une forme de liberté sous la contrainte de réussir et de convenir aux attentes.

Dimension 5 : calcul mathématique et calcul flottant

Des représentants d'entreprise apprécient que les diplômés Ensimag, par leur double culture, contrairement à la plupart des mathématiciens ou informaticiens, comprennent pourquoi 1.1+1.1 n'est pas égal à 2.2 (allez expliquer à vos parents et amis qu'on vous apprend ici que $1+1\neq 2$:-). On a fait le choix de proposer dans le projet GL, à travers des extensions de calcul mathématique que vous pouvez choisir, une compréhension des mécanismes par lesquels une machine va compiler des calculs mathématiques, avec les choix et les contraintes que cela va entraîner. C'est exactement dans la lignée de ce qui a fait la force de l'école grenobloise d'informatique et de mathématiques appliquées, à la suite de Jean Kuntzmann dans les années 1950.

Dimension 6 : responsabilité sociale et développement durable

La Responsabilité Sociétale des Entreprises (RSE) implique la prise en compte des préoccupations sociales et environnementales de la société. Le projet GL est l'occasion de porter une attention particulière à un impact environnemental du numérique : celui de la consommation énergétique des logiciels. En tant qu'ingénieurs, vous devez être non seulement conscients de l'impact (social, environnemental et en particulier) énergétique des systèmes que vous développez, mais également capable d'analyser cet impact pour les choix d'architecture, de programmation ou d'organisation que vous effectuez.

Méthode pédagogique: initiation, prolongements possibles

Sur tous ces points, le principe du projet est celui d'un Apprentissage Par Problème, ou du moins par Projet. Concrètement, cela veut dire que vous n'avez eu qu'une introduction sommaire des concepts dans les cours antérieurs (un peu de compilation en TL, un peu de calcul flottant en architecture des machines et en méthodes numériques, un peu de gestion de projet en P3, quasiment pas de GL). Le projet GL vous permet de découvrir beaucoup plus de choses par vous-même, dans un contexte encadré, mais où vous apprenez en faisant.

En projet GL vous apprendrez les bases minimales. Selon les filières, vous pourrez approfondir certains aspects, par exemple :

- analyse de code pour la sûreté et la sécurité (ISI)
- méthodes de test (cours en 3A ISI, SEOC)
- calcul scientifique et sur GPU (en MMIS et IF)

— ...

Nous vous souhaitons un bon projet GL, riche de découvertes, d'apprentissages, et de temps forts entre co-équipiers.

[ExempleSansObjet]

Exemple introductif « sans objet »

Ce document présente les étapes du compilateur decac sur un exemple très simple (mais pas trop). Il définit par la même occasion la partie « sans objet » du langage qui est à implémenter en premier (voir section 4). Un autre exemple, contenant des classes, est donné en section [Exemple].

On considère le programme Deca suivant dans un fichier "moitie.deca" contenant :

```
// affichage en flottant de la moitié du carré d'un entier
{
   int x;
   x=readInt();
   println(0.5*(x*x));
}
```

Pour obtenir le fichier assembleur (donné en section 3), on utilise le script decac du répertoire "src/main/bin". Pour obtenir un fichier de sortie aux différentes étapes intermédiaires, on utilise les scripts du répertoire "src/test/script/launchers". Ces répertoires sont à ajouter dans votre variable du shell PATH pour pouvoir les exécuter comme dans ce document. Par ailleurs, il est *impératif* d'utiliser ces scripts pour mettre en place des tests de non-régression pour chacune des étapes du compilateur (voir sections 1.3 à 1.6 du document [Tests]).

1 Étape d'analyse syntaxique

1.1 Analyse lexicale

L'analyse lexicale transforme la suite de caractères du fichier d'entrée en suite de lexèmes. Elle est spécifiée dans le document [Lexicographie] (avec les restrictions de la section 4 pour la partie « sans objet »). Ci-dessous, on écrit chaque lexème sous la forme d'une ligne de la forme suivante :

```
TYPE_LEXEME: [...='SUITE_CARACTÈRES',<...>, NUMÉRO_DE_LIGNE: NUMÉRO_DE_COLONNE]
```

Ainsi, le résultat de l'analyse lexicale peut être visualisé via la commande "test_lex moitie.deca":

```
OBRACE: [@0,59:59='{',<42>,2:0]}
IDENT: [@1,64:66='int',<25>,3:3]
IDENT: [@2,68:68='x',<25>,3:7]
SEMI: [@3,70:70=';',<44>,3:9]
IDENT: [@4,75:75='x',<25>,4:3]
EQUALS: [@5,76:76='=',<47>,4:4]
READINT: [@6,77:83='readInt',<10>,4:5]
OPARENT: [@7,84:84='(',<45>,4:12]
CPARENT: [@8,85:85=')',<46>,4:13]
SEMI: [@9,87:87=';',<44>,4:15]
PRINTLN: [@10,93:99='println',<13>,5:3]
OPARENT: [@11,100:100='(',<45>,5:10]
```

```
FLOAT: [@12,101:103='0.5',<24>,5:11]
TIMES: [@13,104:104='*',<28>,5:14]
OPARENT: [@14,105:105='(',<45>,5:15]
IDENT: [@15,106:106='x',<25>,5:16]
TIMES: [@16,107:107='*',<28>,5:17]
IDENT: [@17,108:108='x',<25>,5:18]
CPARENT: [@18,109:109=')',<46>,5:19]
CPARENT: [@19,110:110=')',<46>,5:20]
SEMI: [@20,112:112=';',<44>,5:22]
CBRACE: [@21,115:115='}',<43>,6:0]
```

1.2 Arbre de syntaxe abstraite

Sur un programme syntaxiquement correct, l'analyseur syntaxique transforme la suite de lexèmes en un arbre de syntaxe abstraite (sinon, il affiche un message d'erreur). Les programmes syntaxiquement corrects sont décrits dans le document [Syntaxe] (avec les restrictions de la section 4 pour la partie « sans objet »). La structure des arbres de syntaxe abstraite est décrite dans le document [SyntaxeAbstraite]. Pour l'exemple d'entrée, on obtient l'arbre de syntaxe via la commande "test_synt moitie.deca", ce qui affiche le texte de la figure 1.

Sur cette figure, les arbres de syntaxe abstraite sont représentés de manière à refléter la structure d'arbre décrite par la grammaire d'arbre (cf. [SyntaxeAbstraite]). Chaque nœud est affiché sur une ligne telle que :

- la taille de l'indentation est proportionnelle à la profondeur du nœud;
- les fils (ou éléments) d'une liste commençent par "[]>" et sont reliés entre eux par une double ligne verticale formée de "||";;
- les fils d'un non-terminal (i.e. qui ne sont pas une liste) sont situés en dessous de lui et sont reliés entre eux par une simple ligne verticale formée de "|";
- un nœud commençant par "+>" est le fils d'un non-terminal, mais n'est pas son dernier fils;
- un nœud commençant par "'>" est soit la racine, soit le dernier fils d'un non-terminal;
- les nœuds listes d'un non-terminal "LIST_A" de la grammaire d'arbres, sont de la forme

```
ListA [List with n elements]
```

— les autres nœuds sont de la forme

```
[NUMÉRO DE LIGNE: NUMÉRO DE COLONNE] NOM DU NOEUD (ATTRIBUTS EVENTUELS)
```

2 Étape de vérifications contextuelles et décorations

Cette étape a deux objectifs :

- 1. Rejeter les programmes « mal typés » (c'est-à-dire qui sont non dérivables avec les grammaires attribuées de [SyntaxeContextuelle]) ou « non supportés » au sens de la section 4 (pour la partie « sans objet »).
- 2. Ajouter des informations de typage dans l'arbre de syntaxe pour préparer la génération de code (celles-ci sont décrites dans le document [ArbreEnrichi]).

Lorsque le programme est bien contextuellement correct, l'arbre de syntaxe décoré est obtenu via la commande "test_context moitie.deca" et est indiqué en figure 2. Les décorations apparaissent comme des informations supplémentaires dans l'arbre de syntaxe situées sous un nœud : typiquement, des informations de type attachées aux identificateurs ou sur chaque nœud d'une sous-expression. Un nœud de conversion en flottant est ici aussi ajouté pour préparer la génération de code.

3 Étape de génération de code

La dernière étape du compilateur produit le fichier assembleur à partir de l'arbre décoré. La machine cible du compilateur est décrite dans [MachineAbstraite]. La sémantique de Deca (c'est-à-dire le comportement attendu des programmes Deca à l'exécution) est décrite dans [Semantique]. Voir aussi les explications de la section 4.

Concrètement, la commande "decac moitie.deca" produit un fichier moitie.ass dont le contenu est par exemple :

```
; start main program
       TSTO #1
                       ; 0 (temporary & params on stack) + 1 (variables) + 0 (registers)
       BOV stack_overflow_error
       ADDSP #1
; Main program
; Variables declarations:
; Beginning of main instructions:
; allocate R2 (> 1 register available in assignment with non-trivial RHS)
       RINT
       BOV io_error
       LOAD R1, R3
; release R2
                              ; assignment
       STORE R3, 1(GB)
                     B) ; assignment; return value of assignment
       LOAD R3, R2
; Expression value computed in R2 ignored
       LOAD #0x1.0p-1, R2 ; Float constant
; allocate R2 (> 1 reg available in binary expression with non-trivial RHS)
       LOAD 1(GB), R3
; simple binary expression
       MUL 1(GB), R3
; No need for overflow check for type int
       FLOAT R3, R3
; release R2
; non-trivial expression, registers available
       MUL R3, R2
       BOV overflow_error ; Overflow check for previous operation
       LOAD R2, R1 ; Load in R1 to be able to display
       WFLOAT
       WNL
       HALT
; end main program
overflow_error:
       WSTR "Error: Overflow during arithmetic operation"
       WNL
       ERROR
stack_overflow_error:
       WSTR "Error: Stack Overflow"
       WNT.
       ERROR
io error:
       WSTR "Error: Input/Output error"
       WNL
       ERROR
```

4 Définition de la partie « sans objet »

Le langage « Deca sans objet » est un sous-ensemble du langage Deca décrit dans les documents de la partie II, hormis les restrictions suivantes :

Syntaxe concrète spécifiée dans [Lexicographie] et [Syntaxe], sauf que les constructions suivantes ne sont pas supportées :

```
return instanceof . new this null class
```

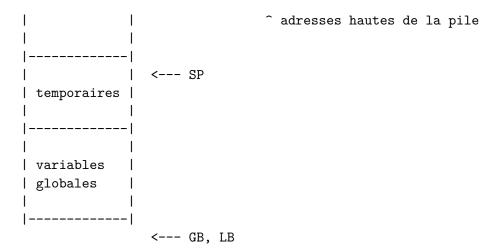
Les constructions « '(' type ')' '(' expr ')' » et « ident '(' list_expr ')' » ne sont pas non plus supportées.

Syntaxe abstraite spécifiée en [SyntaxeAbstraite] et [Decompilation] est inchangée (elle comporte des noeuds inutilisés);

Syntaxe contextuelle réduite uniquement aux règles communes (section 4 de [SyntaxeContextuelle]) et à celles de la passe 3 (section 7) sauf que :

- l'environnement env_types_predef décrit en section 2.3 ne contient pas le type Object;
- et la règle (3.1) sur l'axiome de la grammaire est remplacée par

Conventions de liaison Par rapport au document [ConventionsLiaison], l'état de la pile à l'exécution est plus simplement décrit par :



Le document [Gencode] donne des précisions sur la généreration du code exécutable sans objet.

`> [2, 0] Program

+> ListDeclClass [List with 0 elements]

```
`> [2, 0] Main
      +> ListDeclVar [List with 1 elements]
      | []> [3, 7] DeclVar
             +> [3, 3] Identifier (int)
             +> [3, 7] Identifier (x)
             `> NoInitialization
      `> ListInst [List with 2 elements]
         []> [4, 4] Assign
         || +> [4, 3] Identifier (x)
         || `> [4, 5] ReadInt
         [] > [5, 3] Println
             `> ListExpr [List with 1 elements]
                [] > [5, 14] Multiply
                    +> [5, 11] Float (0.5)
                    `> [5, 17] Multiply
                       +> [5, 16] Identifier (x)
                       `> [5, 18] Identifier (x)
                       Figure 1 – Arbre de syntaxe abstraite non décoré
`> [2, 0] Program
  +> ListDeclClass [List with 0 elements]
   `> [2, 0] Main
      +> ListDeclVar [List with 1 elements]
      | []> [3, 7] DeclVar
             +> [3, 3] Identifier (int)
             | definition: type (builtin), type=int
             +> [3, 7] Identifier (x)
             | definition: variable defined at [3, 7], type=int
             `> NoInitialization
      `> ListInst [List with 2 elements]
         [] > [4, 4] Assign
         || type: int
         || +> [4, 3] Identifier (x)
         || | definition: variable defined at [3, 7], type=int
         \prod
            `> [4, 5] ReadInt
         \Pi
                type: int
         []> [5, 3] Println
             `> ListExpr [List with 1 elements]
                [] > [5, 14] Multiply
                    type: float
                    +> [5, 11] Float (0.5)
                    | type: float
                    `> ConvFloat
                       type: float
                       `> [5, 17] Multiply
                          type: int
                          +> [5, 16] Identifier (x)
                          | definition: variable defined at [3, 7], type=int
                          `> [5, 18] Identifier (x)
                             definition: variable defined at [3, 7], type=int
```

FIGURE 2 – Arbre de syntaxe décoré

$[{f RenduIntermediaire}]$

Rendu Intermédiaire

1 Consignes à respecter

Une dizaine de jours avant la fin du projet, nous vous demanderons un premier rendu, ou « rendu intermédiaire » de votre compilateur. Le rendu sera simplement ramassé dans le dépôt Git de votre équipe. Il ne concerne que le compilateur Deca vers ima, pas les extensions ou les bibliothèques que vous développez à votre rythme.

Nous n'attendons pas que vous ayez terminé l'implémentation du compilateur, mais le compilateur doit marcher de la même manière que la version finale sur la partie sans-objet, définie dans la section 4 du document [ExempleSansObjet]. Votre base de tests devra également être la plus complète possible sur cette partie.

Le fonctionnement externe du compilateur doit être le même que pour la version finale : mêmes messages d'erreur, même manière de générer le code, et la base de tests doit également être organisée de la manière imposée par les spécifications.

Vos enseignants feront passer la partie de leur base de tests correspondant à ce sous-ensemble, et évalueront la couverture de votre base de tests, toujours sur ce même sous-ensemble. La documentation n'est pas prise en compte pour ce rendu. Vous aurez un retour dans les 48h suivant le rendu, et la qualité de votre rendu intermédiaire sera prise en compte dans la note finale.

À la fin du projet, les enseignants réévalueront la totalité du compilateur et de la base de tests (i.e. ce qui a déjà été testé sur le rendu intermédiaire sera re-testé à la fin).

2 Conseils

Au moment du rendu intermédiaire, il faut faire le maximum pour que la partie non-orientée objet soit la plus complète et la mieux testée possible. Si vous avez voulu trop avancer la partie orientée-objet ou l'extension au détriment de la qualité de la partie sans-objet, ce sera en un sens du temps perdu.

Il est donc indispensable, dès le début du projet, d'organiser le projet de manière à terminer (syntaxe, vérifications contextuelles, et génération de code) ce sous-ensemble du langage avant de revenir au début de la chaîne de compilation pour la partie orientée objet. Paradoxalement, il serait donc contre-productif d'écrire le code dans l'ordre où il sera exécuté par le compilateur.

Concrètement, cela signifie que les incréments doivent être choisis en suivant les fonctionnalités du langage, mais porter sur l'ensemble de la chaîne de compilation.

3 Barème indicatif

Pour l'évaluation du rendu intermédiaire, la partie objet ne sera pas prise en compte. L'évaluation (essentiellement automatique) donnera lieu à deux notes (chacune de A à G) pour la qualité du compilateur et de la base de tests.

— Pour le compilateur, l'évaluation se concentre sur les erreurs dans le code qui ont un impact important sur la réussite (ou non) de la compilation de la majorité des programmes Deca testés. La note est donnée en fonction du niveau atteint, dans cet ordre (le premier échec empêche l'évaluation des niveaux suivants) :

- G "decac -p" n'est pas conforme à la spécification
- F "decac -v" n'est pas conforme à la spécification
- E "decac" ne se comporte pas conformément à la spécification sur des programmes Deca corrects.
- **D** le compilateur marche sur quelques exemples, mais il est plus facile de trouver un programme Deca qui exhibe un bug que de trouver un programme correctement compilable.
- **B à C** le compilateur marche globalement, mais il reste des bugs (la quantité de bugs detectés donne la différence entre B et C)
- A Le compilateur fonctionne conformément à la spécification.
- Un compilateur qui passe les tests fournis dans common-tests.sh devrait arriver au niveau D.
- Pour la base de tests, la note est fonction de la couverture et de la bonne classification (répertoires Valide/, Invalide/, ...).

Description des produits du rendu final

Ce document récapitule les différents produits à rendre à la fin du projet GL. Les dates et heures de rendu sont données dans le document [Introduction]. Un rendu intermédiaire, ne portant que sur une partie du langage, est fait en cours de projet : voir [RenduIntermediaire] pour les détails.

1 Documents à rendre

Les documents à rendre sont les suivants :

- 1.1 Une documentation pour l'utilisateur décrivant ce qu'implémente le compilateur : le manuel utilisateur.
- 1.2 Une documentation sur l'implémentation : la documentation de conception.
- **1.3** Une documentation sur la validation.
- 1.4 Une documentation sur les extensions.
- 1.5 Un document présentant votre analyse de l'impact énergétique de votre projet.
- **1.6** Un bilan sur la gestion d'équipe et de projet.

1.1 Manuel utilisateur

La documentation utilisateur doit être **concise**. Elle s'adresse à un (futur) utilisateur du compilateur, qui a déjà à sa disposition les spécifications détaillées du langage Deca. Elle doit contenir en particulier :

- Les limitations ou les points propres à l'implémentation de votre compilateur (par exemple, les portions du langages non ou mal implémentées). On précisera les effets de ces limitations pour l'utilisateur en diagnostiquant éventuellement la cause de ces limitations.
- Les messages d'erreur qui peuvent être retournés à l'utilisateur (erreurs de lexicographie, de syntaxe hors-contexte, de syntaxe contextuelle, d'exécution du code assembleur). On précisera la liste des messages d'erreurs et les configurations qui les provoquent.
- Les extensions éventuelles de la [BibliothequeStandard].
- Le mode opératoire pour utiliser vos extensions (options de la commande decac, configurations à utiliser etc.).
- Les limitations de vos extensions (par exemple, pour l'extension TRIGO, on attend les marges d'erreur des calculs; pour l'extension BYTE, on attend le sous-ensemble du langage supporté; etc.).

Ce manuel doit être au format PDF et être ajouté dans le dépôt Git sous le nom docs/Manuel-Utilisateur.pdf (avant l'heure limite).

1.2 Documentation de conception

La documentation de conception s'adresse à un développeur qui souhaiterait maintenir et/ou faire évoluer le compilateur. Elle décrit l'organisation générale de l'implémentation, c'est-à-dire :

— les architectures (liste des classes et leurs dépendances);

- les spécifications sur le code du compilateur autres que celles fournies et leurs justifications;
- la description des algorithmes et structures de données employés autres que ceux fournis et leurs justifications.

Il faut éviter de donner des listings de code ou de rentrer dans les détails de la liste des méthodes par classes (le javadoc est là pour ça). Répétons-le : il faut absolument vous limiter aux informations supplémentaires par rapport aux documents fournis par les enseignants.

1.3 Documentation de validation

Voir les consignes sur le contenu du document en section 3.3 de [Tests].

1.4 Documentation de l'extension

Vous rendrez une documentation détaillée de votre extension, (d'environ 20 à 30 pages). Cette documentation doit aborder en particulier :

- Une spécification de l'extension.
- Une analyse bibliographique.
- Vos choix de conception, d'architecture, et d'algorithmes.
- Votre méthode de validation.
- Les résultats de la validation de l'extension.

A titre d'exemple, on détaille ce que pourrait contenir une documentation pour l'extension TRIGO.

Documentation technique de l'extension TRIGO

Cette documentation doit aborder en particulier :

- La conception de la classe Math, les algorithmes utilisés et les choix mathématiques et informatiques que vous avez faits. Cette partie du projet étant beaucoup moins guidée que le reste du compilateur, vous aurez beaucoup plus de choix à expliquer et à justifier que dans la documentation du compilateur.
- Une analyse théorique de la précision que vous pouvez attendre de ces algorithmes.
- La validation des algorithmes et de leur implémentation en Deca. Contrairement à la validation du compilateur, la validation de l'extension TRIGO fait intervenir la notion de précision des calculs (cf. section 5 du document [Consignes]).
- Références bibliographiques pour les algorithmes et méthodes que vous avez envisagés.

La documentation doit être lisible par un informaticien sensibilisé au calcul sur nombres flottants. Elle doit être écrite comme un rapport scientifique, en citant précisément les sources des travaux utilisés et en identifiant les contributions propres de l'approche proposée.

1.5 Analyse des impacts énergétiques de votre projet et propositions

Vous devez réfléchir au long du projet à l'efficience de votre projet et de son produit, le compilateur decac. Ce document devra aborder en particulier deux dimensions importantes dans lesquels vos choix ont un impact sur l'énergie consommée par les logiciels.

- Efficience du code produit : un compilateur sera utilisé pour produire des logiciels, qui consommeront de l'énergie lors de leur exécution. Il faut donc évaluer le coût énergétique des assemblages d'instructions que réalise le compilateur, et privilégier des assemblages moins gourmands. C'est encore plus important pour votre extension, où vous êtes peu contraints.
- Efficience du procédé de fabrication. Dans le cas du projet GL, c'est principalement la compilation et l'exécution des tests qui seront consommateurs. Vous devrez donc réfléchir à l'optimisation de vos processus et de vos scripts de validation, alors même que le respect de la qualité de votre compilateur doit rester votre objectif principal.

Vous pouvez utiliser des informations générales sur la consommation énergétique des ordinateurs, mais vous devez principalement rendre compte de la consommation de votre propre projet. Des commandes comme /usr/bin/times permettent d'avoir des informations indirectes sur la consommation énergétique à travers celle des ressources (temps CPU, accès mémoire, échanges réseau etc). Dans le cas de ima qui est un processeur virtuel, on considèrera que le nombre de cycles est une approximation raisonnable de la consommation d'un programme.

Votre document devrait faire 4 à 10 pages, et pourrait comporter les éléments suivants :

- moyens mis en œuvre pour évaluer la consommation énergétique de votre projet (pensez-y très tôt pour ne pas manquer de données en fin de projet);
- discussion de l'impact de vos choix de compilation de Deca vers ima;
- discussion de votre processus de validation, et stratégie mise en œuvre pour en diminuer l'impact énergétique sans nuire à l'effort de validation et à la qualité du compilateur;
- prise en compte de l'impact énergétique dans votre extension

Mais au-delà de ces suggestions, toute analyse qui vous paraîtrait pertinente sera lue avec attention.

1.6 Bilan de gestion d'équipe et de projet

Sur la méthode de développement, en s'appuyant sur Planning.pdf et Realisation.pdf :

- Description critique de l'organisation adoptée dans l'équipe.
- Présentation de l'historique du projet : ordre choisi pour la conception et le développement des étapes B et C, temps passé sur les différentes activités (analyse, conception, codage, validation, documentation).

L'objectif de l'enseignant n'est pas d'évaluer la qualité intrinsèque de la démarche adoptée, mais l'analyse qui en est faite a posteriori.

Le bilan doit être au format PDF et être commité dans le dépôt sous le nom docs/Bilan_Projet.pdf.

2 Liste des produits à rendre (contenu attendu du dépôt Git)

Vous serez évalués sur le dernier commit fait et envoyé dans *la branche « master »* du dépôt partagé (via git push) avant l'heure limite (cf. [Introduction]). Votre dépôt devra contenir au moins (respectez les conventions de nommage des fichiers) :

- Les sources Java du compilateur (voir détails dans [Consignes] et section 3 ci-dessous)
- Votre base de tests et vos scripts de tests (voir détails dans **Tests**] et notamment sa section 3);
- planning/Planning.pdf: planning prévisionnel
- planning/Realisation.pdf: temps passé effectivement
- docs/Bilan_Equipe.pdf: bilan de gestion d'équipe et de projet (cf. section 1.6)
- docs/Manuel-Utilisateur.pdf: manuel utilisateur (cf. section 1.1).

Documentations de conception, de validation, d'analyse énergétique et documentation de l'extension : à rendre dans le dépôt git (répertoire docs) pour la soutenance.

3 Note sur la qualité du compilateur rendu

Un objectif est bien entendu de rendre un compilateur avec le moins de défauts possibles. Une attention particulière sera portée sur les « erreurs bêtes » faciles à corriger mais avec un impact majeur sur le fonctionnement du compilateur (exemples : code Java non-compilable suite à un git add oublié, compilateur ne respectant pas l'interface en ligne de commande, cf. section 2 du document [Tests]). Il est *impératif* que votre compilateur passe tous les tests du script common-tests.sh fourni. Si la version rendue de votre compilateur ne passe pas ces tests, votre enseignant vous enverra un mail

dans les 24h qui suivent votre rendu, et appliquera une pénalité non rattrapable de 3 points sur 20 sur votre note totale de projet. Vous aurez alors 24h pour fournir une version corrigeant les bugs, et l'enseignant relancera la base de tests sur cette version.

[Soutenance]

Soutenance du projet Génie Logiciel

La soutenance dure une heure, devant un ou plusieurs terminaux. Elle se déroule en deux parties :

- 1. vous disposez de 30 minutes pour présenter votre projet;
- 2. l'enseignant dispose de 30 minutes pour vous poser diverses questions; il vous demandera en particulier de corriger quelques erreurs dans votre projet.

La première partie comprendra deux aspects:

- Une démo « technico-commerciale » de votre compilateur et de l'extension (environ 20 minutes)
- Un bilan sur l'organisation du projet (environ 10 minutes)

Gardez en tête que votre soutenance s'adresse à la fois à votre enseignant en informatique (donc, quelqu'un qui peut comprendre les discours techniques, et qui a déjà une idée assez précise de la qualité technique de votre compilateur), mais surtout à un éventuel client, qui ne connaît pas la façon dont on peut écrire un compilateur.

Chaque étudiant devra intervenir successivement, soit pendant la démo (2 étudiants minimum), soit pendant le bilan. Les temps d'intervention doivent être équilibrés.

Le temps alloué passe vite : pensez à planifier et répéter votre intervention. Prévoyez plutôt un exposé légèrement plus court que le temps alloué : les enseignants peuvent vous poser quelques questions à chaud, et le timing est déjà très serré.

Elle se fera au choix sur vos machines personnelles ou sur un PC de l'école. Pensez à bien utiliser des tailles de police assez grosses, visibles par vos enseignants qui ne seront pas assis au clavier.

L'évaluation de vos compilateurs et de vos bases de tests aura déjà été faite. Les éventuelles corrections de bugs faites entre la date du rendu et la soutenance ne changeront pas la note technique.

Vous pouvez également apporter du code écrit après le rendu final. Si vous avez continué à coder sur votre compilateur, ce que vous aurez ajouté entre temps ne modifiera pas votre évaluation technique. Il peut être intéressant d'écrire des programmes Deca pour rendre la démonstration plus intéressante (un bon programme de test n'est pas forcément un bon programme pour une démonstration technico-commerciale).

La démo doit illustrer le bon comportement et la qualité de votre compilateur. Exemples (non limitatifs) :

- illustrer les messages d'erreurs;
- montrer la conformité de votre compilateur (conformité aux spécifications, au manuel utilisateur et à la sémantique);
- montrer la qualité du code produit;
- montrer les capacités de votre extension.

La démo doit être structurée : vous devez la préparer.

Le bilan sur l'organisation du projet peut être vu comme une présentation orale du document décrit dans [A-Rendre] (« Bilan de gestion d'équipe et de projet »). Il doit porter, en particulier, sur les points suivants (liste non limitative) :

[Soutenance] 30

- organisation des différents rôles;
- temps passé dans les différentes activités (analyse, conception, codage, validation...)
- méthode de test et de mise au point; nombre de retours sur des étapes validées (en étape A pour correction d'erreurs détectées en étapes B ou C, en étape B pour correction d'erreurs détectées en étape C)...
- votre point de vue sur le cadre fixé (les spécifications, les répertoires...) et le projet.

Pour la partie « technico-commerciale », mettez-vous dans la tête que vous n'êtes pas des étudiants en train de présenter un mois de travail scolaire à des enseignants, mais les représentants d'une entreprise qui doivent convaincre leur client qu'ils ont un bon produit, et sont une équipe techniquement et humainement solide et fiable. Sachez faire preuve du professionnalisme de jeunes ingénieurs.

En revanche, dans la partie « Bilan », vous redevenez des étudiants qui présentent à leurs enseignants les leçons qu'ils ont eux-mêmes tirés de cette expérience.

A RENDRE LORS DE LA SOUTENANCE

Pour la soutenance, vous devez imprimer et apporter :

- le plan de la démo,
- la documentation d'analyse énergétique de votre projet,
- le bilan de gestion d'équipe et de projet.

[Communication] 31

Restitution du travail : la communication à l'écrit et à l'oral¹

1. Communication écrite

La méthode retenue est celle du PPQQOQC, méthode largement utilisée en conduite de projet. Cette méthode s'applique à l'ensemble du document que vous rendez mais également à la page de garde.

Pour qui?

Quelle est la cible ?

Pourquoi?

Quels sont les objectifs du travail ? Quel est le but à atteindre?

Qui?

Identification des auteurs du document.

Quoi?

De quoi s'agit-il ? Il est indispensable qu'un document porte un nom, un titre, ou fasse apparaître un objet. Le titre ou l'objet doit permettre de situer le contenu réel du travail en quelques mots.

Où?

Situation dans l'espace. Pour vous il s'agit de l'Ensimag, école du Groupe Grenoble INP.

Quand?

Date d'élaboration du document. Les diverses versions doivent être identifiées.

Comment?

Doit respecter les règles spécifiques au travail demandé ainsi que les outils présentés ci-après. La méthode MOTS développée et dispensée par Bernard Biais « Au-delà des mots, l'image »

M	Matériaux	Rassembler tous les éléments à présenter								
0	Organisation	Construire le plan								
T	Texte	Développer	Développer et argumenter							
S	Style : règle des 3	Court	\rightarrow	Une idée par phrase						
	«C»		\rightarrow	Des phrases de 25 mots						
				maximum						
		→ Des mots adaptés→ Des phrases simples								
				(sujet+verbe+complément)						
		Concret et	\rightarrow	Préférez des mots concrets et						
		précis		précis à des mots abstraits						
			→ N'utiliser que les mots							
		techniques et les abréviation connues par le destinataire								
				inclure un glossaire						
		Clair	\rightarrow	Préférer les tournures						
				personnelles et actives						
			→ Un style le plus direct							
				possible (des verbes à						
				l'indicatif, des verbes au						
				présent)						

¹ Issu d'un document établi par A. Jumbou

[Communication] 32

N'oubliez pas!

→ Lorsque vous mettez de graphiques, captures d'écran,....il est indispensable de les numéroter, leur donner un titre et une légende, et de les référencer et les expliquer dans le texte.

→ Pensez à agrafer vos documents.

2. La communication orale

Une prestation orale s'apprécie au même titre qu'un écrit :

→ L'auditoire est-il convaincu?
 → Les objectifs ont-ils été atteints?
 → L'auditoire s'est-il ennuvé?

L'important est ce que l'on expose et l'aisance avec laquelle on l'expose. La méthode MOTS se transpose à l'oral. Le support projeté doit venir à l'appui de l'exposé et respecter les règles suivantes.

- → Le support projeté à l'oral : l'utilisation d'un support projeté n'est pas obligatoire. Cependant, en cas d'utilisation de powerpoint, l'utilisation d'un masque est requise afin d'unifier la présentation. Ceci permet que chaque diapositive soit positionnée dans le document (n° courant/nombre total de diapos) et fasse apparaître le Qui, Quoi et Date du document.
- → En cas d'utilisation de powerpoint, il est souhaitable que la diapositive soit peu « chargée » et respecte la règle des 3 couleurs. On conseille par diapositive : au max 30 mots, 3 points maximum par diapo, et uniquement les points clés. A ne pas faire : écrire tout ce qui est exposé, lire le contenu de la diapositive.
- → Le support papier : un support papier doit être remis à l'auditoire. Pour vous, il s'agit du document bilan. Nous vous conseillons également de remettre le plan de votre exposé oral, les diagrammes que vous présentez à l'oral. Si vous utilisez powerpoint, nous vous conseillons de donner vos diapositives sous format papier à votre auditoire.

La charte de travail en équipe

livrable pour le suivi SHEME n°1

Pourquoi une charte d'équipe

Dans le cadre du projet GL, vous allez travailler en équipe de 4 ou 5 étudiants. Le travail en équipe est une aventure intéressante, parfois même passionnante, mais vous rencontrerez très probablement des difficultés, voire des conflits. Ces difficultés sont inévitables du fait de la pression subie par l'équipe, de la diversité des profils et des personnalités ... Elles ne sont pas nécessairement des indicateurs de « mauvaise qualité » du groupe. Une équipe peut se fourvoyer sur une mauvaise piste dans un parfait consensus. Les conflits peuvent être source de dynamisme si vous savez les accepter, les analyser et choisir un mode de résolution.

La charte d'équipe définit les règles de fonctionnement que vous mettez en place collectivement pour rendre le plus efficace possible le travail de l'équipe projet.

Ces règles précisent notamment :

- l'organisation du travail au sein de l'équipe,
- la communication dans l'équipe,
- les valeurs communes des membres de l'équipe

Une charte permet de développer une meilleure cohésion des membres de l'équipe, de susciter l'implication de chacun et servir de référence en cas de désaccord ou de conflit.

Elle vous sera d'autant plus utile que vous avez pris le temps de vous connaître, de vous poser les bonnes questions sur votre fonctionnement en équipe, de vous écouter et de réfléchir à des solutions simples et partagées.

Contenu attendu dans la charte

Connaître les compétences de l'équipe, ses points forts et ses points faibles :

- Quelles sont les compétences personnelles et techniques de chaque membre de l'équipe ? (utiliser la grille d'auto-évaluation en annexe 1)
- Quels sont vos points forts et vos points faibles, en tant qu'équipe ? Vous pouvez utiliser la matrice SWOT présentée en annexe 2 pour en faire la synthèse.

Les valeurs communes :

Elles répondent à la question : quels sont les comportements attendus, tolérés, proscrits dans votre équipe? Au niveau global d'une organisation, elles relèvent de la culture d'entreprise.

Elles sont définies collectivement, et on peut s'appuyer sur ces règles pour éventuellement « recadrer » un membre de l'équipe qui serait « hors jeu ». Elles définissent vos règles de

fonctionnement : il est important de les expliciter pour éviter des malentendus qui pourraient se transformer en tensions inutiles au cours du projet.

Quelques exemples:

- Pouvoir parler de tout, discuter de toutes les positions à condition de respecter l'autre
- Faire confiance aux autres membres de l'équipe
- Ne pas dissimuler les problèmes
- Faire preuve d'initiatives
- Droit à l'erreur
- Partager toutes les informations pouvant être utiles à l'équipe
- Etre présent aux réunions d'équipe

_													
	•	٠	•	•	٠	•	٠	٠	٠	٠	٠	٠	•

Rôles et responsabilités dans l'équipe

- Quelle est la répartition des rôles dans l'équipe ?
- Quelles sont les responsabilités liées à chacun de ces rôles? Par exemple : définition de l'architecture technique, développement, tests, définition des besoins client, ...
- En particulier, quelles sont les responsabilités du chef de projet dans votre équipe ?
- Sur quels critères avez-vous réparti les rôles parmi les membres de l'équipe : compétences techniques, expérience, motivation, traits de personnalité, autres ...?
- Rôles fixes tout au long du projet ? ou rôles tournants ?

La communication au sein de l'équipe

- Comment organisez-vous la communication au sein de l'équipe ?
- De façon formelle (réunions d'équipe, workshops sur des sujets spécifiques, autres …) et/ou informelle (travail dans un même lieu, …) ?
- Utilisez-vous des outils pour partager votre documentation ? Quelles règles avez-vous mis en place ?
- Qui est chargé d'animer les réunions ? de rédiger les comptes-rendus ?
- Comment pensez-vous régler les tensions, voire les conflits au sein de l'équipe ?

Annexe 1 : Fiche d'auto-évaluation

Votre prénom NOM :	
Formation d'origine, filière :	
Equipe :	
Tuteurs :	

Cette grille vous aide à formaliser vos caractéristiques personnelles et compétences techniques afin de favoriser votre connaissance de vos propres compétences ainsi que de celles des autres membres de l'équipe. Le but est de mieux situer les qualités de chacun et favoriser le travail en équipe. L'évaluation individuelle sur chaque critère sera suivie d'un échange en équipe sur les notes attribuées individuellement.

Evaluez honnêtement vos caractéristiques personnelles et compétences techniques en **entourant le chiffre** (1=faible ; 2=passable ; 3=dans la moyenne ; 4=très bien ; 5=excellent) qui correspond à votre opinion personnelle.

Caractéristiques personnelles		Ev	aluati	ion	
Leadership	1	2	3	4	5
Planification	1	2	3	4	5
Esprit d'équipe	1	2	3	4	5
Organisation	1	2	3	4	5
Persévérance	1	2	3	4	5
Ponctualité	1	2	3	4	5
Créativité	1	2	3	4	5
Débrouillardise	1	2	3	4	5
Orateur	1	2	3	4	5
Communication écrite.	1	2	3	4	5
Compétences techniques	_1	2	3	4	5
Compétences techniques utiles pour le projet GL, à préciser :					
	1	2	3	4	5
	1	2	3	4	5
	1	2	3	4	5
	1	2	3	4	5
	1	2	3	4	5

Annexe 2 : La matrice SWOT d'équipe

Matrice Forces, Faiblesses, Opportunités et Menaces

FORCES

Ex : cohésion de l'équipe, niveau de compétence technique élevé,

FAIBLESSES

Ex : équipe hétérogène qui ne se connaît pas , une compétence clef n'est pas maîtrisée, ..

OPPORTUNITES

Ex : équipe hétérogène avec des compétences variées mais complémentaires, ..

MENACES

Ex : une compétence technique clef repose sur un seul membre de l'équipe, ...

$[{f Communication Enseignants}]$

Protocole de communication enseignants-étudiants

Chaque équipe du projet de génie logiciel est affectée à un enseignant. C'est à celui-ci, et à celui-ci seulement, que les étudiants de l'équipe doivent s'adresser lorsqu'ils sont confrontés à un problème dans le déroulement du projet. Ceci inclut : problèmes algorithmiques, problèmes de compréhension de spécifications, erreurs difficiles à trouver...

Pour les problèmes liés à la machine ou au système, s'adresser au service informatique ou aux BugBusters.

En dehors des périodes prévues pour les discussions enseignants-étudiants (stage ou suivis), les informations sont transmises par trois moyens de communication (outre le tableau d'affichage pour les emplois du temps) :

1 La page du projet « Actualité du Projet GL »

```
https://projet-gl.pages.ensimag.fr/actualites/
```

Elle est mise à jour régulièrement par les enseignants et consultée par les étudiants. On y trouve les informations destinées à l'ensemble des étudiants du projet.

2 Site web du projet

```
https://projet-gl.pages.ensimag.fr/
```

Ce site web est utilisé par les enseignants pour y placer des documents non fournis dans la documentation papier (soit trop gros, soit terminés trop tard pour le tirage). Les étudiants peuvent aussi contribuer (mais bien sûr, morceaux de code Java ou Deca ne doivent pas être diffusés, ni ici ni ailleurs). Tout placement d'un nouveau document important par un enseignant sur ce site sera signalé sur la page « Actualité du Projet GL ».

3 Le courrier électronique

- Dans le sens enseignant \rightarrow étudiants, il est utilisé lorsqu'un message est destiné à un groupe particulier d'élèves.
- Dans le sens étudiants → enseignant, il sert essentiellement à correspondre au sujet des problèmes dont il est question ci-dessus.

Les questions posées doivent l'être de façon claire et précise (il arrive souvent que le seul fait d'arriver à clarifier une question suffise à y apporter une réponse). Les messages doivent contenir suffisamment d'éléments pour que l'enseignant puisse effectivement répondre aux questions. Utiliser le cas échéant des « scripts » de session pour produire des traces (voir la commande script Unix). Ces traces doivent être « dégraissées » avant d'être envoyées. Il faut trouver le bon compromis entre un message trop laconique (« il y a une erreur dans DeclVar. java. ») et un message de 500 lignes.

Le courrier électronique est un moyen de communication asynchrone. Cela signifie qu'une réponse n'est pas censée arriver dans les cinq minutes, ni même dans les deux heures, qui suivent la demande. L'enseignant garantit cependant une réponse au plus tard dans les 24 heures en semaine. Exemple : un mail posté le vendredi à 20 heures recevra une réponse au plus tard le mardi suivant, dans la matinée.

Un enseignant qui se trouve dans la situation de ne pouvoir répondre à son courrier électronique dans les délais requis doit en informer les étudiants au préalable. Il doit également leur indiquer à quel autre enseignant ils doivent s'adresser en cas de problème urgent (c'est-à-dire dont le traitement ne peut absolument pas attendre le retour de l'enseignant).

Contenu des séances de suivi du projet Génie Logiciel

Chaque suivi doit être **préparé**, en particulier chaque équipe doit apporter les documents relatifs à chaque suivi.

Un suivi dure 30mn et comporte deux parties :

- Partie 1 (20 minutes) Avancement : L'équipe présente l'avancement du projet ainsi que les parties spécifiques à chaque suivi qui sont détaillées ci-dessous. L'enseignant joue le rôle du client, la qualité de la présentation des étudiants est prise en compte dans la note finale.
- Partie 2 (10 minutes) Soutien technique : L'enseignant répond à des questions techniques pour aider les étudiants à avancer. Pour les questions délicates, il est FORTEMENT conseillé que ces questions soient posées par mail au moins 1/2 journée avant la séance; cela permet :
 - à l'enseignant d'y réfléchir avant la séance;
 - aux étudiants d'y répondre parfois tous seuls, rien qu'en la rédigeant.

Un plan du travail d'une séance de suivi sur l'autre doit être élaboré.

A chaque séance, il faut rendre votre document d'avancement du projet (fait avec un outil approprié comme planner, cf. Projet_GL/planning/Realisation.pdf), à jour en particulier pour la période écoulée depuis le suivi précédent.

Contenu de chaque suivi

Cette liste est bien entendu adaptable en fonction de la vitesse d'avancement et de l'ordre choisi pour la réalisation des différentes tâches.

Suivi 1

Contenu indicatif:

- Présentation de l'équipe, l'organisation, le planning
- Avancement du projet [note 1]
- Etat du « hello world » [note 3]
- Choix de l'extension négociée avec les enseignants

Documents à apporter :

- Charte « Fraude interdite et sanctionnée pour les TP et projets », signée (plus d'informations [Fraude])
- Charte de travail (cf. le document [Suivi-SHEME-I])
- Liste des erreurs pour l'étape B (au moins sans-objet) [note 5]
- Planning [note 2]
- Proposition(s) pour l'extension

Suivi 2

Contenu indicatif:

— État des réflexions, lectures de documentations, pour les étapes B et C (hors programmation objet)

- Avancement du projet [note 1]
- Négociation du contenu attendu pour l'extension
- Préparation du rendu intermédiaire

Documents à apporter :

- Planning prévisionnel mis à jour suite au suivi 1 si nécessaire
- Spécification détaillée de l'extension, architecture et choix des algorithmes
- Partie « Gestion des risques » et « gestion des rendus » de la documentation de validation (cf. section 2 et 3.3 du document [Tests])

Suivi 3 avec l'enseignant en informatique

Contenu indicatif:

- Retour sur le rendu intermédiaire
- Avancement du projet [note 1]
- Avancement sur l'extension
- Point sur les dernières tâches avant le rendu final

Documents à apporter :

- État des documentations et de l'architecture du projet (à discuter avec l'enseignant avant la version finale) [note 4]
- Si besoin, mise à jour des parties « Gestion des risques » et « gestion des rendus » de la documentation de validation suite au rendu intermédiaire.

Notes

- 1. Avancement du projet : Organisation et planning du sprint effectué. Présentation des tâches effectuées, avec une démonstration du bon fonctionnement du compilateur. Présentation des tâches prévues pour le sprint suivant.
- 2. Planning: rendre le planning prévisionnel, c'est-à-dire .../Projet_GL/planning/Planning.pdf. Le planning peut prendre plusieurs formes suivant la méthode de gestion de projet que vous utilisez, Scrum ou classique. Dans le premier cas, il s'agira d'un Gantt simplifié pour montrer l'enchainement des sprints et leur contenu en user stories ET du tableau des tâches (to do/doing/done) du sprint en cours. Dans le second cas, il s'agira d'un Gantt détaillé au niveau de chaque incrément fonctionnel. Vous utiliserez de préférence l'outil de planification planner pour le diagramme Gantt (faire la commande 'planner', puis ouvrir le fichier Planning.planner). Pour le tableau des tâches vous pouvez utiliser Trello ou GitLab, en les structurant de façon à faire apparaître les user stories et les tâches. Tout au long du projet, mettre à jour (le plus souvent possible) le fichier Realisation.planner, qui contiendra le planning effectif. Toujours faire 'git pull' avant de modifier ce fichier; toujours faire 'git commit -a' et 'git push' après avoir modifié et exporté ce fichier pour éviter les conflits.
- 3. « Hello, world »: les étudiants doivent s'organiser pour avoir un compilateur capable de compiler entièrement (étapes A, B et C, jusqu'à la génération du fichier assembleur) un programme simple (typiquement, constitué uniquement de l'instruction println("Hello, world");). Au premier suivi, les étudiants en font une démonstration, ou présentent leur plan pour y arriver dans les plus brefs délais.

4. **Présentation architecture :** liste des composants, dépendances entre les composants, spécification des composants (spécifications de paquetages), définition des incréments (ordre de codage choisi) et méthode de test des composants.

5. Erreurs étape B : rendre un document décrivant la liste des messages d'erreurs de syntaxe contextuelle (au moins pour le langage sans-objet), avec pour chacun le ou les numéros correspondants des règles de la grammaire attribuée.

Consignes et conseils pour l'implémentation du compilateur Deca et de sa base de tests

Ce document donne une vue d'ensemble du travail à réaliser, étape par étape. Pour plus de détails, on peut se référer au squelette de code : certaines parties sont guidées avec des commentaires A FAIRE, et l'exception UnsupportedOperationException est levée sur la plupart des portions non-implémentées. Une manière de voir ce qui reste à faire est donc :

git grep -e 'A FAIRE' -e 'UnsupportedOperationException'

Il est conseillé de suivre les indications de [**ProgrammationDefensive**] pour écrire votre code : vous gagnerez beaucoup de temps au débogage.

Attention, ce document ne donne pas l'ordre du développement du compilateur et de sa base de tests : il décrit juste les tâches à faire pour implémenter chaque étape du compilateur (et des tests associés). Vous devez trouver l'ordre du développement par vous-mêmes. On conseille typiquement de suivre une démarche incrémentale par « user stories », c'est-à-dire où chaque incrément correspond à une « fonctionnalité » du point de vue de l'utilisateur. Dans le cas d'un compilateur, un tel incrément correspond typiquement à un « sous-ensemble cohérent » du langage. Autrement dit, si on représente le développement comme une marche dans la grille en 2 dimensions ci-dessous, qui part de la case « DÉPART » pour aller en direction de la case « OBJECTIF FINAL », il s'agit de construire le compilateur et ses tests ligne à ligne plutôt que colonne par colonne. Et, bien sûr, en vérifiant que les tests passent comme attendu au fur et à mesure...

décomposition interne de decac

niveaux
de complexité
de decac

	r		
	étape A	étape B	étape C
incrément 1	Départ		
incrément 2			
incrément 3			
Déca sans-objet			Objectif intermédiaire
Déca complet			Objectif final
	incrément 2 incrément 3 Déca sans-objet	incrément 1 DÉPART incrément 2 incrément 3 Déca sans-objet	incrément 1 DÉPART incrément 2 incrément 3 Déca sans-objet

En cas de doute sur l'ordre de développement que vous envisagez, demandez conseil à votre enseignant.

1 Le compilateur decac

Le compilateur à proprement parler est le programme principal qui appelle les étapes A, B et C sur les programmes sources. Ce document le présente en premier, pour donner une vue d'ensemble du compilateur, et annoncer certaines contraintes sur le style de codage, qu'il sera nécessaire d'appliquer par la suite. La compilation d'un fichier se fait essentiellement en enchaînant ces trois instructions Java :

```
AbstractProgram prog = doLexingAndParsing(sourceName, err); // étape A prog.verifyProgram(this); // étape B prog.codeGenProgram(this); // étape C
```

1.1 Code fourni

- Une classe DecacCompiler, qui est la classe permettant la compilation d'un fichier Deca. La classe contient des méthodes qui permettent d'enchaîner les étapes A, B et C (cf. ci-dessus), et d'autres objets utilisés pendant la compilation (par exemple, une instance de IMAprogram dans laquelle le code sera généré, et d'autres classes à ajouter, notamment SymbolTable).
- Une classe CompilerOption, qui permet de gérer les options du compilateur (récupérer les options passées sur la ligne de commande, et les mettre à disposition du reste du programme).
- Une classe DecacMain, la classe principale du compilateur, qui s'occupe d'instancier CompilerOption pour gérer la ligne de commande, puis d'instancier DecacCompiler (Dans le cas où decac est appelé sur plusieurs fichiers Deca, cette classe sera instanciée une fois par fichier à compiler) et de lancer la compilation de chaque fichier.
- Des exceptions DecacFatalError et CLIException levées en cas de problème avec la ligne de commande decac.

1.2 Travail demandé

Les classes DecacCompiler, CompilerOption et DecacMain sont incomplètes. Attention en particulier à implémenter les options (-b, -p, -v, -n, -r ...) et à respecter le format de sortie, sans quoi votre compilateur ne pourra pas être évalué automatiquement.

Un point qui demande une attention particulière est l'implémentation de l'option -P (parallel). En utilisant le paquetage java.util.concurrent de la bibliothèque standard (depuis Java 1.5), l'implémentation à proprement parler se fait en quelques dizaines de lignes. Une solution possible :

- Utiliser java.util.concurrent.Executors pour créer un ensemble de fils d'exécution travailleurs (worker threads). Par exemple, utiliser Executors.newFixedThreadPool et java.lang.Runtime.getRuntime().availableProcessors() pour créer autant de fils d'exécution que de processeurs sur la machine.
- Soumettre aux travailleurs une tâche par fichier à exécuter (ExecutorService.submit()). On obtient un ensemble de Future<Boolean>, c'est à dire une classe représentant une valeur qui n'est peut-être pas encore calculée. Ici, la valeur est la valeur de retour de DecacCompiler.compile()
- Pour chaque Future, appeler la méthode get qui va attendre que la compilation soit terminée si ce n'est pas déjà le cas.

1.3 Gestion du parallélisme

Le fait que decac puisse compiler plusieurs fichiers en parallèle a une conséquence très importante sur l'ensemble du compilateur : il n'est pas possible d'utiliser des variables globales (i.e. des variables static).

Supposons par exemple que les définitions de classes Deca soient stockées dans une structure de données EnvironmentType, partagée entre les instances de DecacCompiler. Le compilateur pourrait compiler un premier fichier Deca et trouver une définition de classe (disons, MaClasse), qui serait ajoutée à l'environnement. Pendant ce temps, un autre fil d'exécution qui compilerait un programme incorrect utilisant la classe MaClasse sans la définir pourrait accéder à l'environnement, trouver la définition de MaClasse ajoutée par l'autre fil d'exécution, et considérer à tort le programme comme correct.

On peut remarquer que c'est le fait de partager l'environnement qui pose problème, et pas seulement le fait d'avoir des accès concurrents. Ajouter des synchronisations (mutex, mot clé synchronized en

Java) ne résoudrait pas le problème. En fait, même compiler plusieurs fichiers en séquence poserait déjà certains problèmes (mais ceux-ci pourraient être résolus en faisant une réinitialisation brutale entre deux compilations).

Il est donc indispensable d'écrire l'ensemble du projet sans utiliser de variable globale. Les structures de données partagées entre différents paquetages du compilateur doivent être définies comme membres de la classe DecacCompiler et non comme variable globale. Par exemple, la structure EnvironmentType mentionnée ci-dessus (utilisée en étape B) est définie comme un champ de DecacCompiler. C'est le cas aussi pour l'instance de IMAProgram dans laquelle le code assembleur est généré, et pour d'autres paquetages que vous aurez besoin d'ajouter en particulier en étape C.

Faire des mesures et optimisations de performances est plus difficile qu'il n'y parait, puisque la JVM utilise elle-même plusieurs threads (par exemple pour le garbage collector). On cherchera donc avant tout à avoir du code propre et parallélisable, mais les performances ne sont pas très importantes pour notre projet.

2 Étape A : Analyse lexicale et syntaxique

Il s'agit de mettre en œuvre un analyseur syntaxique, pour Deca, qui construit l'arbre abstrait primitif du programme. Une partie importante du travail consiste donc à fournir une preuve constructive du théorème 1 de [Decompilation] sous la forme d'un programme source ANTLR. ¹ La lexicographie est décrite dans [Lexicographie], la syntaxe est décrite dans [Syntaxe], les arbres de syntaxe abstraite sont décrits dans [SyntaxeAbstraite] et [Decompilation].

Un programme syntaxiquement correct peut être rejeté par l'analyseur si ses caractéristiques provoquent un dépassement des limites de l'implémentation : domaine des valeurs entières et flottantes, mémoire disponible, etc.

2.1 Principaux répertoires concernés

src/main/antlr4/fr/ensimag/deca/syntax/: Fichiers sources pour le générateur d'analyseur ANTLR,

src/main/java/fr/ensimag/deca/syntax/ : Fichiers sources Java,

src/main/java/fr/ensimag/deca/tree/ : Implémentation de l'arbre abstrait (utilisé également dans les étapes B et C),

src/test/java/fr/ensimag/deca/syntax/ : Fichiers de tests Java (en particulier, tests JUnit),

src/test/script/ et src/test/script/launchers/ : Scripts de tests (utilisé également dans les étapes B et C),

src/test/deca/syntax/: Cas de tests en Deca (cf. [Tests]). Pour cette étape comme pour les étapes
B et C, quelques tests sont fournis dans les répertoires valid/provided et invalid/provided,
à vous d'en ajouter!

2.2 Code fourni

Le squelette de code contient en particulier :

- Une implémentation incomplète de SymbolTable, la table de symboles (qui associe chaque identificateur Déca à une objet String unique), ainsi qu'un test unitaire (JUnit) SymbolTest pour vous aider à tester la table de symboles, et pour vous servir d'exemple pour écrire de nouveaux tests unitaires (pour SymbolTable et pour d'autres classes).
- Des classes de base pour l'analyseur lexical et l'analyseur syntaxique : AbstractDecaLexer et AbstractDecaParser
- Des exceptions pour certaines catégories d'erreur : CircularInclude, IncludeFileNotFound et InvalidLValue.

^{1.} Vérifier cette preuve n'est pas forcément évident ;-)

— Un ensemble de classes pour représenter un arbre abstrait : le contenu du paquetage fr.ensimag.deca.tree et en particulier la classe de base Tree commune à tous les nœuds. Les classes correspondant aux nœuds du langage sans-objet sont toutes fournies (mais incomplètes); il faudra ajouter de nouvelles classes pour gérer le langage complet. Voir le document [ConventionsCodage] pour les détails sur l'implémentation de l'arbre abstrait en utilisant le patron « interprète ».

— Des squelettes d'analyseurs lexicaux et syntaxiques : DecaLexer.g4 et DecaParser.g4, à compléter.

Par ailleurs, des programmes de tests sont fournis : test_lex et test_synt (qui sont des scripts lanceurs pour les classes ManualTestLex et ManualTestSynt). Ces programmes lisent depuis un fichier ou depuis leur entrée standard, et affichent le résultat de l'analyse.

Des exemples de scripts de tests automatiques sont également fournis (basic-lex.sh et basic-synt.sh). Comme leurs noms l'indiquent, ces scripts sont très basiques, et ne sont là que pour servir d'exemple pour écrire de meilleurs scripts.

2.3 Travail demandé

On demande:

- De la documentation (cf. [A-Rendre]),
- Une implémentation de l'analyse,
- Des jeux de tests, dans le répertoire src/test/ (cf. [Tests]).

2.4 Conseils

L'implémentation de l'analyse lexicale et syntaxique se fait en utilisant l'outil ANTLR, dont une documentation rapide est fournie dans le document [ANTLR].

Un point potentiellement délicat est l'implémentation du #include dans l'analyse lexicale, mais la quasi-totalité du code nécessaire vous est fournie par la classe AbstractDecaLexer, avec la méthode doInclude(String includeDirective). Il vous suffira donc d'écrire une règle dans DecaLexer.g4 et d'appeler la fonction doInclude dans l'action associée à cette règle.

L'implémentation fournie gère pour vous l'inclusion de fichier (en utilisant l'instruction setCharStream pour changer temporairement l'entrée du lexer), la détection d'inclusion circulaire, et la recherche dans la bibliothèque standard, qui se fait avec l'instruction Java suivante (name est le nom du fichier à inclure) :

```
ClassLoader.getSystemResource("include/" + name);
```

En pratique, cela signifie donc que les fichiers de la bibliothèque standard seront placés dans le répertoire src/main/resources/include/. La commande mvn compile va copier ces fichiers dans target/classes/include/, et l'appel à getSystemResource ira chercher les fichiers dans ce répertoire. Attention pendant le développement : il faudra modifier les fichiers de src/main/resources/include/ et relancer mvn compile. Si vous éditez les fichiers de target/classes/include/ directement, les modifications seront écrasées à la prochaine compilation.

Remarque : la commande mvn package génère un fichier .jar qui contient tous les fichiers .class de Decac et de ses dépendances, et les fichiers de la bibliothèque standard. L'implémentation de doInclude est capable de lire les fichiers depuis le .jar généré. Ce n'est pas très important dans le cadre du projet, mais serait important dans la vraie vie pour faciliter la distribution du compilateur.

3 Étape B : Vérifications contextuelles

Il s'agit d'une part de vérifier la syntaxe contextuelle de Deca décrite dans le document [Syntaxe-Contextuelle] et d'autre part d'enrichir l'arbre abstrait et d'y ajouter des informations contextuelles. Ces « décorations » sont décrites dans [ArbreEnrichi].

3.1 Principaux répertoires concernés

src/main/java/fr/ensimag/deca/tree/: Implémentation des parcours de l'arbre abstrait,
src/main/java/fr/ensimag/deca/context/: Fichiers sources Java,
src/test/java/fr/ensimag/deca/context/: Fichiers de tests Java
src/test/deca/context/: Cas de tests en Deca.

3.2 Code fourni

On fournit:

- Un squelette de classe EnvironmentType pour représenter l'environnement des types (i.e. EnvironmentType dans [SyntaxeContextuelle]). Au départ cette classe est juste initialisée sur l'environnement des types prédéfinis (i.e. env_types_predef dans [SyntaxeContextuelle]) du Déca « sans-objet ». Il faudra l'étendre au moment du passage au Déca « objet ».
- Une classe abstraite Type, dont dérivent les classes StringType, VoidType, BooleanType, IntType, FloatType, NullType et ClassType;
- Des classes *Definition et une classe Signature permettant de représenter les définitions d'identificateurs Deca (définition de types, de variables, de classes ...);
- Un squelette de classe EnvironmentExp, qui devra implémenter un dictionnaire qui associe à chaque identificateur sa définition, lorsque celle-ci est instance de ExpDefinition;
- Une exception ContextualError, à lever pour arrêter l'exécution du compilateur en cas d'erreur contextuelle.

Comme pour l'étape A, on fournit également un programme test_context (script lanceur pour la classe ManualTestContext) permettant de tester l'analyse contextuelle sans avoir terminé le compilateur.

3.3 Conseils

Cette partie est conséquente et nécessite de procéder de manière très méthodique. Pour être exhaustif il faut procéder par incréments en considérant successivement les différentes « passes » de la vérification et les différents sous-ensembles du langage.

Il sera demandé une architecture modulaire de l'étape de vérifications contextuelles, vous devez donc introduire de nouveaux paquetages. Par contre, il est très important de garder un code le mieux factorisé possible (i.e. d'éviter la duplication de code inutile). La hiérarchie de classes qui vous est proposée pour l'arbre abstrait (dans le squelette de code et la grammaire d'arbres) permet cela en utilisant judicieusement l'héritage. Par exemple, la classe Plus devra contenir très peu de code, vu que sa spécification est très proche d'autres classes comme Minus ou même And. Le code gérant l'opérateur + sera autant que possible écrit dans une des classes de base de Plus. Par exemple, la décompilation est la même pour tous les opérateurs binaires et est donc implémentée dans AbstractBinaryExpr.decompile(), qui utilise une méthode getOperatorName() qui devra être implémenté dans chaque classe fille (pour la classe Plus, elle renvoie la chaîne "+"). Les vérifications contextuelles sont trop différentes pour être factorisées entre opérateurs arithmétiques et booléens, mais sont exactement les mêmes pour tous les opérateurs arithmétiques, donc elles sont implémentées un niveau plus bas dans la hiérarchie de classe, dans AbstractOpArith.verifyExpr().

Dans l'implémentation des vérifications contextuelles, on cherchera à commenter l'implémentation en référence au document de spécification [SyntaxeContextuelle].

4 Étape C : Génération de code

Il s'agit de produire une suite d'instructions de la machine abstraite qui corresponde à l'arbre d'un programme Deca contextuellement correct, en s'inspirant des algorithmes décrits dans [Gencode]. La

machine abstraite est décrite dans [MachineAbstraite] et est exécutable avec le programme fourni ima décrit dans [Ima]. La sémantique de Deca est décrite dans [Semantique]. L'interface en ligne de commande est décrite dans [Decac] (cf. également la section 1 ci-dessus).

Cette étape étant la dernière de la chaîne de compilation, on ne cherchera pas à la tester unitairement (c'est possible de tester la génération de code indépendemment des étapes A et B, mais très fastidieux : ça prend beaucoup de temps d'écrire les tests). La génération de code va donc de paire avec l'implémentation de l'interface en ligne de commande du compilateur, décrite section 1. En d'autres termes, il n'y a pas besoin d'un équivalent de test_lex, test_synt et test_verif pour l'étape C, on utilisera directement decac.

4.1 Principaux répertoires concernés

```
src/main/java/fr/ensimag/deca/tree/: Implémentation des parcours de l'arbre abstrait,
src/main/java/fr/ensimag/deca/codegen/: Fichiers sources Java,
src/test/java/fr/ensimag/deca/codegen/: Fichiers de tests Java,
src/test/deca/codegen/: Cas de tests en Deca.
```

4.2 Code fourni

On fournit:

- Un paquetage vide fr.ensimag.deca.codegen, dans lequel pourront être ajoutées des classes utiles à la génération de code.
- Une structure de données représentant la syntaxe abstraite d'un programme en langage d'assemblage IMA, dans le paquetage fr.ensimag.deca.ima.pseudocode. La classe représentant un programme complet est IMAProgram.

A titre d'exemple, on fournit aussi un embryon d'implémentation de la génération de code des instructions à partir de l'arbre de syntaxe d'un programme Deca. Celui-ci ne sait traiter que les programmes Deca qui se réduisent à une suite de print ou println sur des chaînes de caractères (voir les méthodes codeGenInst des classes AbstractPrint et Println et la méthode StringLiteral.codeGenPrint). La classe de test fr.ensimag.deca.tree.ManualTestInitialGencode exécute cet embryon de génération de code sur un mini-exemple sans utiliser les étapes A et B. Ainsi, après avoir fait "mvn test-compile" à la racine du projet, lancer

```
mvn -q exec:java -Dexec.classpathScope="test" \
  -Dexec.mainClass="fr.ensimag.deca.tree.ManualTestInitialGencode"
doit afficher
---- From the following Abstract Syntax Tree ----
`> Program
   +> ListDeclClass [List with 0 elements]
      +> ListDeclVar [List with 0 elements]
      `> ListInst [List with 2 elements]
         []> Print
            `> ListExpr [List with 1 elements]
                [] > StringLiteral (Hello )
         \Pi
         []> Println
             `> ListExpr [List with 1 elements]
                [] > StringLiteral (everybody !)
---- We generate the following assembly code ----
; Main program
; Beginning of main instructions:
```

```
WSTR "Hello "
WSTR "everybody !"
WNL
HALT
```

4.3 Travail demandé et conseils

L'essentiel du travail consiste à ajouter les méthodes permettant la génération de code aux classes définissant les nœuds de l'arbre. Au final, la génération de code sera faite par le compilateur en appelant la méthode AbstractProgram.codeGenProgram(). Les méthodes des classes de l'arbre doivent rester très courtes : les portions de code non-triviales (gestion des registres et de la pile par exemple) seront factorisées dans d'autres classes, à ajouter dans le paquetage fr.ensimag.deca.codegen.

Pour les nœuds de l'arbre correspondant à des déclarations, le parcours d'arbre se contente en général de calculer l'emplacement mémoire correspondant à la déclaration, et décore les définitions en utilisant Definition.setOperand(). La génération de code a proprement parler utilisera getOperand() pour retrouver l'opérande à utiliser dans le code généré sans avoir à la recalculer.

5 Consignes particulières pour l'extension TRIGO

La classe Math ne fait pas partie du langage Deca. Si vous choisissez cette extension, elle doit être fournie dans la bibliothèque standard (cf. [BibliothequeStandard]). En terme d'implémentation, cela signifie que le compilateur Decac n'a rien de particulier à faire pour l'implémentation de cette classe, si ce n'est gérer correctement le #include (cf. 2.4) et éventuellement implémenter une optimisation de la génération de code des expressions de la forme "a + b*c" via l'instruction FMA (cf. discussion ci-dessous), ainsi que les méthodes asm. La classe Math sera implémentée en Deca, dans le fichier src/main/resources/include/Math.decah. On peut si besoin utiliser les méthodes écrites en assembleur (la spécification autorise l'ajout de méthodes à la classe Math pourvu que leur nom commence par le caractère '_').

Idéalement, les calculs devraient renvoyer l'arrondi au flottant représentable le plus proche de la valeur exacte de la fonction mathématique concernée, qui soit à l'intérieur du codomaine de cette fonction. C'est le mieux qu'on puisse faire avec une représentation en nombre flottant. En pratique, atteindre ce niveau de précision est très difficile, on pourra donc se contenter d'une précision un peu moins bonne, à condition de documenter correctement les limitations de votre implémentation.

Quelques pistes pour évaluer/améliorer la précision de votre classe Math:

- L'unité de précision élémentaire sur les nombres flottants est l'intervalle entre deux flottants représentables (« Unit in the last place », ou « ULP »). On peut remarquer que calculer l'arrondi correct de la valeur exacte est plus difficile que d'en calculer un arrondi à 1 ULP près.
- Certains bugs sont plus faciles à identifier avec une représentation graphique. Il peut être intéressant de tracer des courbes (vue d'ensemble, ou vue zoomée sur des points particuliers de la courbe) et de comparer avec les courbes obtenues avec d'autres logiciels.
- L'affichage d'un flottant en décimal fait un arrondi, et n'affiche pas la valeur exacte contenue dans le flottant. Un affichage hexadécimal (printx, printlnx) permet d'afficher tous les bits significatifs du flottant. Par exemple, 0x1.FFFFFEp0 et 0x1.0p1 sont deux flottants différents, mais tous deux affichés 2.00000e+00 en décimal. Inversement, la lecture d'un littéral flottant décimal se fait avec un arrondi par le compilateur, alors que les littéraux hexadécimaux peuvent représenter tous les flottants possibles.
- Considérer toutes les sources d'imprécision. En général, on utilise des formules qui sont mathématiquement approchées (approximation d'une série infinie par une somme finie), mais le fait que les opérations se fassent sur des valeurs flottantes au lieu de nombres réels apporte d'autres imprécisions (liées à l'informatique et non aux mathématiques).
- Typiquement, lorsque le calcul d'une fonction mathématique réelle est approximé par l'évaluation d'un polynôme, on calcule cette évaluation par la méthode de Horner. Pour améliorer la

précision de ce calcul dans un programme assembleur, il est bon d'utiliser l'instruction FMA (voir section 3.4 du document [MachineAbstraite]). Celle-ci permet en effet de réaliser le calcul d'une expression "a+(b*c)" (avec a, b et c expressions de type float) en n'ayant qu'une seule erreur d'arrondi, au lieu de 2 si on utilise un MUL suivi d'un ADD.

La quantité de travail à fournir sur la classe Math dépend fortement du niveau de précision recherché. Obtenir le meilleur arrondi possible pour toutes les valeurs d'entrée est un problème très difficile (la première version de la norme IEEE754, publiée en 1985, ne demandait pas ce niveau de précision car il n'y avait pas d'algorithme connu qui le permette à l'époque), et inversement obtenir une précision à quelques dizaines d'ULP pour un petit sous-ensemble des valeurs d'entrée possibles se fait en quelques dizaines de minutes. Nous attendons des étudiants Ensimag qu'ils soient capables de trouver le meilleur compromis entre ces deux extrêmes et qu'il soient capables d'argumenter et de justifier la précision qu'ils peuvent obtenir pour chaque fonction, en différenciant selon les sous-domaines du domaine de définition.

Deuxième partie Spécifications du compilateur decac

Lexicographie de Deca

Conventions de notation dans ce document

- Les éléments entre quotes (comme '0', '.') désignent les caractères ou séquences de caractères correspondants; '' désigne la séquence de caractères vide.
- Les mots notés en majuscules (comme LETTER, DIGIT) désignent des langages réguliers.
- Les noms de langages en italique (comme *SOUSREGLE*) sont utilisés dans d'autres règles, mais ne désignent pas des unités lexicales de Deca.
- Les opérateurs sur les langages utilisés sont les notations habituelles d'expressions régulières.
- On appelle « caractère de formatage » : la tabulation horizontale, le retour chariot ('\r'), et la fin de ligne ('\n'). Cette dernière est désignée par EOL.

Unités lexicales

Les unités lexicales de Deca sont les mots réservés, les identificateurs, les symboles spéciaux, les littéraux entiers, flottants (décimaux et hexadécimaux) et chaînes de caractères.

Mots réservés

Les séquences de lettres suivantes sont des mots réservés :

```
asm class extends else false if instanceof new null readInt readFloat print println printlnx printx protected return this true while
```

Identificateurs

```
LETTER = 'a' + ... + 'z' + 'A' + ... + 'Z'

DIGIT = '0' + ... + '9'

IDENT = (LETTER + '$' + '_')(LETTER + DIGIT + '$' + '_')*
```

Exception : les mots réservés ne sont pas des identificateurs.

Symboles spéciaux

Les caractères suivants, ainsi que les associations suivantes de deux caractères ont un sens particulier en Deca :

```
'<' '>' '=' '+' '-' '*' '/' '%' '.' ',' '(' ')' '{' '}'
'!' ';' '==' '!=' '>=' '<=' '&&' '||'
```

Littéraux entiers

```
POSITIVE_DIGIT = '1' + ... + '9'
INT = '0' + POSITIVE_DIGIT_DIGIT*
```

Une erreur de compilation est levée si un littéral entier n'est pas codable comme un entier signé positif sur 32 bits.

Littéraux flottants

Les littéraux flottants sont convertis en arrondissant si besoin au flottant IEEE-754 simple précision le plus proche. Une erreur de compilation est levée si un littéral est trop grand et que l'arrondi se fait vers l'infini, ou bien qu'un littéral non nul est trop petit et que l'arrondi se fait vers zéro.

Le suffixe f est autorisé mais ignoré, pour permettre une meilleure compatibilité de Deca avec Java.

Chaînes de caractères

On définit *STRING_CAR* comme l'ensemble de tous les caractères, à l'exception des caractères '"', '\' et de *EOL* (fin de ligne).

```
STRING = '"' (STRING_CAR + '\\"' + '\\\')* '"'
MULTI_LINE_STRING = '"' (STRING_CAR + EOL + '\\"' + '\\\')* '"
```

(Avec par convention : '\\"' représente un caractère '\' suivi d'un caractère '\"'. '\\\\' représente deux caractères '\' successifs)

Commentaires

Un commentaire « classique » commence par '/*' et termine par '*/'. Le commentaire s'arrête au premier '*/' suivant le début du commentaire.

Un commentaire « mono-ligne » est une suite de caractères (autres qu'une fin de ligne) qui commence par deux '/' successifs et s'étend jusqu'à la fin de la ligne ou du fichier.

Séparateurs

Les séparateurs de Deca sont ' ' (caractère d'espace) et les caractères de formatage (tabulation horizontale, fin de ligne ' \n' ou retour chariot ' \n' ') ou des commentaires.

Inclusion de fichier

Un programme Deca peut inclure un autre fichier avec la syntaxe

```
FILENAME = (LETTER + DIGIT + '.' + '-' + '_')+
INCLUDE = '#include' (' ')* '"' FILENAME '"'
```

Une inclusion de fichier ne produit pas directement de lexème, mais change temporairement le fichier d'entrée : le fichier passé en argument du #include est lu, puis la lecture reprend après le #include quand le fichier inclus est terminé.

Le fichier à inclure est cherché dans le même répertoire que le fichier source. S'il n'existe pas de fichier avec ce nom dans le répertoire, alors le fichier est cherché dans la bibliothèque standard fournie avec le compilateur, qui est décrite dans le document [BibliothequeStandard].

Syntaxe concrète de Deca

La syntaxe concrète de Deca est définie par la grammaire ci-dessous. Celle-ci utilise une syntaxe étendue avec expressions régulières proche des grammaires [ANTLR].

Conventions de la grammaire concrète

Chaque terminal de cette grammaire représente :

- 1. soit le lexème spécial EOF marquant la fin de fichier
- 2. soit un lexème concret comme 'while' ou ';'
- 3. soit le nom d'un langage régulier de lexèmes concrets, comme IDENT

La règle de membre gauche assign_expr serait plus clairement écrite sous la forme

```
assign_expr

→ or_expr

| lvalue '=' assign_expr

| value

→ ident
| select_expr '.' ident
| '(' lvalue ')'
```

Malheureusement, comme le langage **lvalue** est inclus dans le langage **or_expr**, une telle grammaire **[ANTLR]** induit une analyse très inefficace du langage **assign_expr**. En pratique, on vérifie donc que l'expression à gauche d'une affection est une **lvalue** grâce aux attributs d'arbre abstrait manipulés par l'analyseur syntaxique (c'est ce qui est exprimé par le commentaire dans la règle de **assign_expr**).

Les règles d'associativité et de précédence déterminant le parenthésage implicite des opérateurs se déduisent à partir de la structure d'arbre d'analyse engendrée par cette grammaire non-ambiguë.

Grammaire concrète

```
\begin{array}{l} \operatorname{prog} \\ \to \operatorname{list\_classes} \ \operatorname{main} \ \to \varepsilon \\ \mid \operatorname{block} \\ \end{array}
```

```
decl_var_set
  \rightarrow type list_decl_var ';'
list_decl_var
  \rightarrow decl_var (', 'decl_var)*
decl_var
   \rightarrow ident
     ('='\mathbf{expr})?
list_inst
   \rightarrow (inst)*
inst

ightarrow expr ';'
    | ';'
     'printlnx' '(' list_expr ')' ';'
    if_then_else
      'while' '(' expr ')' '{' list_inst '}'
      'return' expr ';'
if_then_else
   → 'if' '(' expr ')' '{' list_inst '}'
      ('else' 'if' '(' expr ')' '{' list_inst '}')*
      ('else' '{' list_inst '}')?
list_expr
   \rightarrow (expr
       (',' expr)*)?
\mathbf{expr}
   \rightarrow assign_expr
assign_expr
   \rightarrow or_expr (
        { condition: expression e must be a "lvalue" }
        '=' assign_expr
      \mid \varepsilon \rangle
or_expr
   \rightarrow and_expr
   or_expr '||' and_expr
and_expr
   \rightarrow eq_neq_expr
   and_expr '&&' eq_neq_expr
eq_neq_expr
   \rightarrow inequality_expr
    eq_neq_expr '==' inequality_expr
    eq_neq_expr '!=' inequality_expr
inequality_expr
   \rightarrow sum_expr
   | inequality_expr '<=' sum_expr
    inequality_expr '>=' sum_expr
    | inequality_expr '>' sum_expr
```

```
inequality_expr '<' sum_expr
    inequality_expr 'instanceof' type
sum_expr
  \rightarrow mult_expr
    sum_expr '+' mult_expr
    sum_expr '-' mult_expr
mult\_expr
   \rightarrow unary_expr
    mult_expr '*' unary_expr
    mult_expr '/' unary_expr
    mult_expr '%' unary_expr
{\bf unary\_expr}
   \rightarrow '-' unary_expr
    '!' unary_expr
    select_expr
select\_expr
   \rightarrow primary_expr
    select_expr '.' ident
        ('(' list_expr ')'
         \mid \varepsilon \rangle
primary_expr
   \rightarrow ident
    | ident '(' list_expr ')'
    | '(' expr ')'
     'readInt' '(' ')'
      'readFloat' '(' ')'
      'new' ident '(' ')'
      '(' type ')' '(' expr ')'
    literal
type

ightarrow 	ext{ident}
literal
   \rightarrow INT
    FLOAT
    STRING
      'true'
     'false'
      'this'
      'null'
ident
   \rightarrow IDENT
/**** Class related rules ****/
list\_classes
  \rightarrow (class_decl) *
class_decl
   → 'class' ident class_extension '{' class_body '}'
class_extension

ightarrow 'extends' \operatorname{ident}
```

```
| ε
class\_body
   \rightarrow (decl_method
       | decl_field_set)*
decl_field_set
   \rightarrow visibility type list_decl_field
visibility
   'protected'
list\_decl\_field

ightarrow \ \operatorname{decl\_field}
          (',' decl_field)*
decl\_field

ightarrow \ \mathbf{ident}
       ('=' expr)?
decl\_method
   → type ident '(' list_params ')' (block
       | 'asm' '(' multi_line_string ')' ';')
list\_params
    \rightarrow (param (',' param)*)?
{\bf multi\_line\_string}
    \rightarrow STRING
    | MULTI_LINE_STRING
param
  \rightarrow type ident
```

$[{f Syntaxe Abstraite}]$

Syntaxe abstraite de Deca

Par définition, la syntaxe abstraite de Deca est l'ensemble d'arbres **PROGRAM**. Tout programme Deca syntaxiquement correct (voir [Lexicographie] et [Syntaxe]) peut être représenté par un arbre de cet ensemble. Concrètement, cet ensemble d'arbres est défini par la grammaire de la section 2. Autrement dit, un mot engendré par cette grammaire encode un arbre qui représente lui-même la structure d'un programme Deca. La grammaire d'arbres est donnée avec un système d'attributs qui "décompile" chaque arbre de la syntaxe abstraite dans la syntaxe concrète. Voir [Decompilation].

1 Notations de la grammaire des arbres de syntaxe abstraite

Les non-terminaux de la grammaire sont en majuscules; ils représentent des « ensembles d'arbres ». L'axiome est le premier non-terminal, ici **PROGRAM**. Les noms de nœuds sont des terminaux de la grammaire et sont en <u>CamelCase</u> (souligné). Les deux autres terminaux de la grammaire sont les crochets « [» et «] » qui servent à délimiter des listes d'arbres. Comme détaillé ci-dessous, dans notre encodage des arbres, chaque paire de crochets « bien parenthésée » représente un nœud : ce nœud est éventuellement étiqueté par un nom qui, dans ce cas, précède le crochet ouvrant.

1.1 Syntaxe des règles de la grammaire d'arbres

Dans la grammaire, il y a deux types de non-terminaux : ceux qui représentent des ensembles de listes d'arbres (définis section 2.2) et les autres qui représentent des ensembles d'arbres dits "de base". (définis section 2.1). Pour ceux-ci, les règles de la grammaire sont de la forme

$$G \rightarrow D_1 \mid D_2 \mid \dots \mid D_n$$

avec $n \ge 1$, où G est le non-terminal partie gauche (définissant un ensemble d'arbres) et les D_i sont les alternatives de partie droite. La forme de chaque D_i est

- soit un unique non-terminal "A", auquel cas l'ensemble d'arbres défini par A est *inclus* dans celui défini par G;
- soit " $\underline{Xxx} \uparrow a_1 ... \uparrow a_k$ ", auquel cas l'arbre est réduit à une feuille qui contient k attributs, dont les types sont respectivement $a_1, ..., a_k$. Le type des attributs est détaillé en section 1.2;
- soit " $\underline{\mathtt{Xxx}} \uparrow a_1 \dots \uparrow a_k [F_1 \dots F_p]$ " (avec $p \ge 1$), auquel cas le nœud $\underline{\mathtt{Xxx}}$ a p enfants (sous-arbres), qui appartiennent respectivement aux ensembles d'arbres F_1, \dots, F_p .

Une propriété importante de cette grammaire est d'une part qu'un nom de nœud donné n'apparaît que dans une seule règle, et d'autre part qu'un non-terminal donné apparaît au plus une fois en tant que membre droit d'une règle d'inclusion. Ce critère syntaxique simple garantit la non-ambiguïté de la grammaire ainsi définie. ¹

Remarque: dans les représentations graphiques des arbres de syntaxe abstraite comme à la section 1.2 de [Exemple], il est important d'introduire un nœud spécifique pour chaque liste d'arbres, sinon la représentation graphique est potentiellement ambiguë.

^{1.} Cela garantit donc l'existence d'une bijection entre les arbres encodés par les mots reconnus par la grammaire, et les arbres d'analyse de ces mots. D'où le nom de « grammaire d'arbres ».

[SyntaxeAbstraite] 62

1.2 Attributs sur les noms de nœuds

Ci-dessus, un type d'attribut a_i est l'un des types suivants :

boolean type des booléens

int type des entiers 32 bits

float type des flottants simple précision qui respectent les conditions sur les littéraux énoncés dans [Lexicographie]

String type des chaînes de caractères

Symbol représentation concrète des identificateurs

Visibility type énuméré à 2 valeurs : public et protected

Le profil de chaque terminal avec attribut est donné ci-dessous :

```
BooleanLiteral↑boolean

DeclField↑Visibility

FloatLiteral↑float

Identifier↑Symbol

IntLiteral↑int

Print↑boolean

Println↑boolean

StringLiteral↑String

This↑boolean
```

Les attributs sont utilisés de la manière suivante : pour <u>BooleanLiteral</u>, <u>FloatLiteral</u>, <u>IntLiteral</u> et <u>StringLiteral</u>, l'attribut représente la valeur du litéral. Pour <u>Identifier</u>, l'attribut est le nom de l'identificateur. Pour <u>This</u>, l'attribut est vrai si et seulement si le nœud a été ajouté pendant l'analyse syntaxique sans que le programme source ne contienne le mot clé this (par exemple, m(); pour dire this.m();). Cet attribut n'est pas strictement nécessaire, mais il est utilisé dans la décompilation (cf. section [<u>Decompilation</u>]). Pour <u>Print</u> et <u>Println</u>, l'attribut booléen dit si les valeurs numériques doivent être affichée en hexa. Pour <u>DeclField</u>, l'attribut représente bien entendu la visibilité (public ou protégé) des champs.

2 Grammaire des arbres de syntaxe abstraite

2.1 Règles pour les ensembles d'arbres de base

```
PROGRAM →
Program[LIST_DECL_CLASS MAIN]

MAIN →
EmptyMain
| Main[LIST_DECL_VAR LIST_INST]

DECL_VAR →
DeclVar[IDENTIFIER IDENTIFIER INITIALIZATION]

IDENTIFIER →
Identifier↑Symbol

INITIALIZATION →
Initialization[EXPR]
| NoInitialization
```

[SyntaxeAbstraite] 63

```
\mathbf{EXPR} \rightarrow
     BINARY_EXPR
   LVALUE
   READ_EXPR
   STRING_LITERAL
   UNARY_EXPR
   | \underline{\texttt{BooleanLiteral}} \uparrow boolean
   Cast IDENTIFIER EXPR
   | FloatLiteral \( float \)
   InstanceOf EXPR IDENTIFIER
   <u>IntLiteral</u>↑int
     MethodCall EXPR IDENTIFIER LIST_EXPR
     New IDENTIFIER
     Null
   <u>This</u>↑boolean
BINARY\_EXPR \rightarrow
     OP_ARITH
   OP_BOOL
   OP_CMP
   Assign LVALUE EXPR
OP\_ARITH \rightarrow
     Divide[ EXPR EXPR]
   Minus EXPR EXPR
   | Modulo EXPR EXPR
   | Multiply EXPR EXPR
   Plus EXPR EXPR
OP\_BOOL \rightarrow
     And EXPR EXPR
   or EXPR EXPR
OP\_CMP \rightarrow
     OP_EXACT_CMP
   OP_INEQ
OP EXACT CMP \rightarrow
     Equals[ EXPR EXPR]
   NotEquals EXPR EXPR
OP_INEQ \rightarrow
     Greater EXPR EXPR
   GreaterOrEqual EXPR EXPR
   Lower EXPR EXPR
   LowerOrEqual EXPR EXPR
LVALUE \rightarrow
     IDENTIFIER
   | Selection EXPR IDENTIFIER
READ_EXPR \rightarrow
     \underline{\texttt{ReadFloat}}
```

[SyntaxeAbstraite] 64

```
ReadInt
STRING\_LITERAL \rightarrow
      \underline{StringLiteral} \uparrow String
UNARY\_EXPR \rightarrow
      ConvFloat EXPR
     \underline{\mathsf{Not}}[\mathbf{\ EXPR}]
   UnaryMinus EXPR
INST \rightarrow
      EXPR
   PRINT
   <u>IfThenElse</u> EXPR LIST_INST LIST_INST
     NoOperation
   Return EXPR
   While EXPR LIST_INST
\mathbf{PRINT} \ \rightarrow
      Print boolean [LIST_EXPR]
   Println boolean LIST_EXPR
/****
            Class related rules
                                        ****/
DECL\_CLASS \rightarrow
      DeclClass | IDENTIFIER IDENTIFIER LIST_DECL_FIELD LIST_DECL_METHOD
DECL FIELD \rightarrow
      DeclField Visibility | IDENTIFIER IDENTIFIER INITIALIZATION |
DECL\_METHOD \rightarrow
      DeclMethod[ IDENTIFIER IDENTIFIER LIST_DECL_PARAM METHOD_BODY]
DECL\_PARAM \rightarrow
      DeclParam[ IDENTIFIER IDENTIFIER]
\mathbf{METHOD\_BODY} \ \rightarrow
      MethodAsmBody STRING_LITERAL
     MethodBody LIST_DECL_VAR LIST_INST
```

2.2 Règles pour les ensembles de listes d'arbres

```
\begin{split} & \textbf{LIST\_DECL\_CLASS} \, \rightarrow \, [\, \textbf{DECL\_CLASS*} \,] \\ & \textbf{LIST\_DECL\_FIELD} \, \rightarrow \, [\, \textbf{DECL\_FIELD*} \,] \\ & \textbf{LIST\_DECL\_METHOD} \, \rightarrow \, [\, \textbf{DECL\_METHOD*} \,] \\ & \textbf{LIST\_DECL\_PARAM} \, \rightarrow \, [\, \textbf{DECL\_PARAM*} \,] \\ & \textbf{LIST\_DECL\_VAR} \, \rightarrow \, [\, \textbf{DECL\_VAR*} \,] \\ & \textbf{LIST\_EXPR} \, \rightarrow \, [\, \textbf{EXPR*} \,] \\ & \textbf{LIST\_INST} \, \rightarrow \, [\, \textbf{INST*} \,] \end{split}
```

Décompilation des arbres de syntaxe abstraite

Ce document précise le lien entre la syntaxe concrète de [Syntaxe] et la syntaxe abstraite de [SyntaxeAbstraite] et participe donc à la spécification de l'analyseur syntaxique (voir le théorème 1). Il donne aussi la spécification de l'option -p du compilateur decac (voir section 3).

1 Notations de la grammaire attribuée de décompilation

Tous les non-terminaux ont un unique attribut qui est une chaîne de caractère synthétisée : c'est le texte du programme Deca provenant de la décompilation de l'arbre.

Les ensembles d'arbres correspondant à des listes, comme **LIST_INST**, ont un non-terminal spécifique qui permet de concentrer le traitement sur ces ensembles de listes dans une seule règle. Ces règles ont elles-mêmes une syntaxe spécifique pour exploiter la construction générique basée sur l'opérateur * des expressions régulières. Elles sont détaillées en section 1.3.

1.1 Règles associées aux ensembles d'arbres de base

Les règles de la grammaire de décompilation sont calquées sur celles de la grammaire d'arbre. Elles sont de la forme "G\reftr \rightarrow D_1 | D_2 | ... | D_n" où \rightarrow r est le nom de l'attribut en « résultat ». La forme de chaque D_i est

- 1. soit "A\r", auquel cas le résultat de la décompilation se propage sans changement;
- 3. soit "Xxx↑r", où Xxx est un nom de noeud avec un unique attribut parmi ceux de la section 1.2 de [SyntaxeAbstraite]. Son attribut est implicitement converti en chaîne de caractères à la décompilation. Cette conversion généralement « évidente » n'est pas détaillée dans ce document, hormis le cas des attributs de type Visibility. Ainsi, public est converti en ε, tandis que protected est converti en 'protected'.
- 4. soit " $\underline{\mathsf{Xxx}} a_1 ... \uparrow a_k [F_1 \uparrow x_1 ... F_p \uparrow x_p]$ { $r := \mathtt{calcul}$ }", auquel cas la décompilation respective des fils est appelée $x_1 ... x_p$, et le résultat r de la décompilation de l'arbre est celui de \mathtt{calcul} en fonction des a_i et des x_j .

1.2 Syntaxe des calculs dans les règles

On utilise les notations suivantes sur les mots : ε pour le mot vide et l'opérateur binaire « . » pour la concaténation. Les littéraux sont notés entre apostrophes comme 'println'.

On utilise aussi un opérateur if-then-else à la C « $(cond ? w_1 : w_2)$ » où cond est une expression booléenne et w_1 et w_2 deux mots. Les expressions booléennes sont limitées à l'accès dans un attribut booléen et à l'égalité sur les mots.

1.3 Règles associées aux ensembles de listes d'arbres

Les règles de la grammaire sont de la forme "LIST_A\(\gamma\) $r \to \{r := init\}[(A\(\gamma\) x \ \{r := iter\})*]$ " où A est un ensemble d'arbres de base, et init et iter sont deux calculs dont la sémantique est la suivante :

- au départ l'affection "r := init" est effectuée (init est une constante).
- puis, l'affection "r := iter" est itérée sur tous les éléments de la liste, dans l'ordre, de gauche à droite. Ici, iter dépend du r précédent et du résultat x de la décompilation de l'élément.

Cette sémantique est similaire à celle des grammaires attribuées ANTLR avec expressions régulières.

En fait, dans la grammaire d'arbres de Deca, il n'y a que 2 façons de décompiler les listes d'arbres (voir section 4.2):

- soit en faisant la simple concaténation de la décompilation des sous-arbres éléments;
- soit en insérant le littéral ',' entre chaque décompilation.

2 Correspondance entre la syntaxe abstraite et la syntaxe concrète

La propriété suivante constitue une spécification de l'analyseur syntaxique : celui-ci doit réaliser ce théorème, c'est-à-dire construire l'arbre dont ce théorème affirme l'existence.

Théorème 1 (spécification de l'analyseur syntaxique) Pour tout programme Deca syntaxiquement correct, il existe un arbre de la syntaxe abstraite qui se décompile dans un programme Deca syntaxiquement correct « équivalent ».

Cette équivalence de programme correspond à l'égalité syntaxique des suites de lexèmes concrets modulo

- 1. l'équivalence de parenthésage ¹ dans les expressions;
- 2. l'équivalence entre deux littéraux qui représentent une valeur identique, même si leur encodage de cette valeur est différent (par exemple dans FLOATDEC et FLOATHEX);
- 3. la « complétion » des suites de if-then-else-if sans une dernière branche "else", par une branche "else" avec une liste d'instructions vide :

```
if (cond) { x=1 }
est complété en
if (cond) { x=1 } else { }
```

4. l'équivalence entre les if-then-else-if-else et les if-then-else avec accolades systématiques dans les branches "else" :

```
if (cond1) { x=1 } else if (cond2) { y=2 } else { z=3 } est équivalent à if (cond1) { x=1 } else { if (cond2) { y=2 } else { z=3 } }
```

5. la transformation des "decl_var_set" en "(type decl_var ';')*" équivalents (c-à-d. en préservant l'ordre de déclaration des variables) :

```
int x=1, y=x;
est transformé en
```

^{1.} Deux expressions dérivant du non-terminal **expr** de **[Syntaxe]** sont dites équivalentes pour le parenthésage ssi on peut rendre leurs arbres d'analyses identiques en « insérant » uniquement

[—] des « noeuds de parenthésage » : noeuds associés à la règle primary_expr → '(' expr ')';

 [—] et des « noeuds d'inclusion » : noeuds associés à un pas de réécriture parmi select_expr → primary_expr, ou unary_expr → select_expr, ... ou expr → assign_expr.

```
int x=1;
int y=x;
```

6. la transformation des "decl_field_set" en "(visibility type decl_field ';')*" équivalents (c-à-d. en préservant l'ordre de déclaration des champs et en propageant la visibilité sur chaque déclaration):

```
protected int x, y=x;
est transformé en
protected int x;
protected int y=x;
```

- 7. la « complétion » des déclarations de classes sans extends avec "extends Object".
- 8. l'échange de place (dans une déclaration de classe) entre déclarations de méthodes et déclarations de champs (mais en préservant l'ordre entre déclarations de champs).

Par exemple, considérons le programme Deca syntaxiquement correct (mais contextuellement incorrect) "{ (this)+x; }". Il correspond à l'arbre abstrait donné à la figure 1. En effet, la décompilation de cet arbre donne "{(this+x);}", et ces deux programmes peuvent tous deux se réécrire dans le programme "{((this)+x);}" en insérant uniquement des parenthèses superflues.

Remarquons au passage que dans l'arbre de la figure 1, si on remplace <u>This</u>false par <u>This</u>ftrue, alors la décompilation produit le programme suivant qui n'est pas syntaxiquement correct $\{(+x);\}$. Ce comportement étrange de <u>This</u>ftrue permet de faire correspondre l'arbre de syntaxe abstraite "<u>MethodCall[This</u>ftrue <u>Identifier</u>m []]" à l'appel de méthode "m()" (avec this implicite) de la syntaxe concrète.

Ainsi, la grammaire d'arbres « simplifie » certaines constructions quitte à permettre des arbres qui ne correspondent pas à des programmes syntaxiquement corrects. En pratique, on peut négliger ces arbres car l'analyseur syntaxique ne pourra pas les produire (sauf bogue)!

Par ailleurs, on peut montrer ce deuxième théorème ci-dessous. En conjonction avec le théorème 1, il entraîne que la *composée* de *l'analyseur syntaxique* par *la décompilation* est idempotente (pour l'égalité des chaînes de caractères). Cette dernière propriété est exploitée dans les tests automatiques (voir section 1.9 de [Tests]) pour détecter des bogues sur "decac -p".

Théorème 2 (sobriété de la décompilation) Étant donnés deux programmes Deca syntaxiquement corrects obtenus par décompilation d'arbre, si ces deux programmes sont équivalents au sens du théorème 1, alors ils correspondent à deux chaînes de caractères identiques.

Pour finir, un noeud de l'arbre n'a pas de correspondance dans la syntaxe concrète : c'est le noeud <u>ConvFloat</u> qui est utilisé pour dénoter la conversion d'une expression entière en flottant dans [ArbreEnrichi]. La décompilation n'est donc pas injective.

3 Spécification de l'option -p

L'option -p de decac doit décompiler l'arbre abstrait obtenu par l'analyse syntaxique lorsque le programme en entrée est syntaxiquement correct (cf. [Decac]). La décompilation de decac -p doit s'inspirer de celle spécifiée section 4, mais en indendant le programme décompilé de manière à le rendre plus lisible. Un exemple de gestion d'indentation est donné dans le code fourni du noeud "While". La décompilation de decac -p doit rester sobre (cf. théorème 2) : il est vivement conseillé de tester l'idempotence de la composée de l'analyseur syntaxique par la décompilation.

```
Notation de la grammaire
                                         Notation de test_synt
                                       (cf. [ExempleSansObjet])
Program
                              `> [1, 0] Program
                                +> ListDeclClass [List with 0 elements]
   Main
                                 `> [1, 0] Main
                                    +> ListDeclVar [List with 0 elements]
                                    `> ListInst [List with 1 elements]
                                       []> [1, 8] Plus
          Plus
                                           +> [1, 3] This
               This false
                                           `> [1, 9] Identifier (x)
               Identifier x
```

FIGURE 1 – Deux notations du même arbre

4 Grammaire attribuée de décompilation

4.1 Règles pour les ensembles d'arbres de base

```
PROGRAM\uparrow r \rightarrow
          \underline{\mathtt{Program}}[\ \mathbf{LIST\_DECL\_CLASS} \uparrow class_s \ \ \mathbf{MAIN} \uparrow main] \ \{\ r := class_s.main\}
MAIN\uparrow r \rightarrow
          \underline{\mathtt{EmptyMain}}\{\,r:=\varepsilon\}
      | \underline{\text{Main}}[ \underline{\textbf{LIST\_DECL\_VAR}} | var_s \underline{\textbf{LIST\_INST}} | \{ r := `\{`.var_s.inst_s.'\}' \}
\mathbf{DECL\_VAR} \uparrow r \rightarrow
         DeclVar
                IDENTIFIER↑type
                IDENTIFIER↑name
                INITIALIZATION†init
         ] \{ r := type.' `.name.init.';' \}
IDENTIFIER\uparrow r \rightarrow
          Identifier \uparrow r
INITIALIZATION\uparrow r \rightarrow
          Initialization [EXPR \uparrow e] \{ r := ' = ' \cdot e \}
      | NoInitialization\{r := \varepsilon\}
\mathbf{EXPR} \uparrow r \rightarrow
          BINARY_EXPR\uparrow e \{ r := '('.e.')' \}
      | LVALUE\uparrow r
      | READ_EXPR\uparrow r
      | STRING_LITERAL↑r
      UNARY_EXPR\uparrow e \{ r := '('.e.')' \}
         BooleanLiteral\uparrow r
         \underline{\mathtt{Cast}}[\mathbf{IDENTIFIER}\uparrow type \ \mathbf{EXPR}\uparrow e] \{ \ r := \text{`('}.type.') \ \text{('}.e.')' \}
         FloatLiteral\uparrow r
```

```
<u>InstanceOf</u> [EXPR\uparrow e IDENTIFIER\uparrow type] { r := '('.e.') instanceOf '.type.')'}
          IntLiteral\uparrow r
      <u>MethodCall</u>
                 EXPR↑obj
                 \textbf{IDENTIFIER} \uparrow meth
                 \mathbf{LIST} \underline{\phantom{}} \mathbf{EXPR} \uparrow param_s
          \{ r := (obj = \varepsilon ? \varepsilon : obj.'.').meth.'('.param_s.')' \}
      New IDENTIFIER class r := 'new' \cdot class \cdot ()'
      | Null{ r := 'null'}
      This impl\{r := (impl? \varepsilon : 'this')\}
BINARY_EXPR\uparrow r \rightarrow
          OP\_ARITH \uparrow r
      | \mathbf{OP}_{\mathbf{BOOL}} \uparrow r
      | \mathbf{OP}_{\mathbf{CMP}} \uparrow r
      Assign LVALUE\uparrow lval EXPR\uparrow e \{ r := lval.' = '.e \}
OP ARITH\uparrow r \rightarrow
          \underline{\mathsf{Divide}}[\ \mathbf{EXPR} \uparrow e_1 \ \ \mathbf{EXPR} \uparrow e_2] \ \{\ r := e_1.'/'.e_2\}
      \underline{\text{Minus}}[\mathbf{EXPR}\uparrow e_1 \ \mathbf{EXPR}\uparrow e_2] \{ r := e_1.\text{'--'}.e_2 \}
      | \underline{\mathsf{Modulo}}[ \underline{\mathbf{EXPR}} \uparrow e_1 \underline{\mathbf{EXPR}} \uparrow e_2 ] \{ r := e_1. \%. e_2 \}
       | \quad \underline{\text{Multiply}}[ \ \mathbf{EXPR} \uparrow e_1 \ \ \mathbf{EXPR} \uparrow e_2 ] \ \{ \ r := e_1. \text{'*'}. e_2 \} 
      Plus EXPR\uparrow e_1 EXPR\uparrow e_2 \{ r := e_1.'+'.e_2 \}
OP\_BOOL\uparrow r \rightarrow
          And [\mathbf{EXPR} \uparrow e_1 \ \mathbf{EXPR} \uparrow e_2] \{ r := e_1. \& \& . e_2 \}
      \underline{\mathsf{or}}[\mathbf{EXPR}\uparrow e_1 \ \mathbf{EXPR}\uparrow e_2] \{ r := e_1.' | 1'.e_2 \}
\mathbf{OP} \mathbf{CMP} \uparrow r \rightarrow
          OP\_EXACT\_CMP\uparrow r
      | OP_INEQ\uparrow r
OP EXACT CMP\uparrow r \rightarrow
          Equals \begin{bmatrix} \mathbf{EXPR} \uparrow e_1 & \mathbf{EXPR} \uparrow e_2 \end{bmatrix} \{ r := e_1.'=='.e_2 \}
      NotEquals EXPR\uparrow e_1 EXPR\uparrow e_2 | { r := e_1.'!='.e_2 }
OP_INEQ\uparrow r \rightarrow
          Greater EXPR\uparrow e_1 EXPR\uparrow e_2 | { r := e_1.'>'.e_2 }
      GreaterOrEqual EXPR\uparrow e_1 EXPR\uparrow e_2 \{ r := e_1.' >= '.e_2 \}
      Lower EXPR\uparrow e_1 EXPR\uparrow e_2 | { r := e_1.'<'. e_2 }
      LowerOrEqual EXPR\uparrow e_1 EXPR\uparrow e_2 \{ r := e_1 . <= '.e_2 \}
LVALUE\uparrow r \rightarrow
          IDENTIFIER\uparrow r
      Selection EXPR\uparrow obj IDENTIFIER\uparrow field { r := obj.'.'.field}
READ EXPR\uparrow r \rightarrow
          \underline{ReadFloat}\{r := 'readFloat()'\}
      | ReadInt{r := 'readInt()'}
STRING LITERAL\uparrow r \rightarrow
          StringLiteral\uparrow r
```

```
UNARY_EXPR\uparrow r \rightarrow
         ConvFloat [EXPR↑r]
         \underline{\text{Not}}[\mathbf{EXPR}\uparrow e] \{ r := '!'.e \}
      | \underline{\text{UnaryMinus}}[\underline{\mathbf{EXPR}} \uparrow e] \{ r := '-' \cdot e \} 
INST\uparrow r \rightarrow
         EXPR\uparrow e \{ r := e.'; ' \}
        \mathbf{PRINT} \uparrow r
     | IfThenElse
               EXPR\(\frac{1}{2}\)cond
              LIST_INST\uparrow then_s
              LIST_INST\uparrow else_s
         | \{ r := 'if('.cond.') \{'.then_s.'\} \text{ else } \{'else_s.'\}' \}
     NoOperation \{r := '; '\}
      Return [\mathbf{EXPR}\uparrow e] { r := \text{'return'}.e.';'}
     While EXPR \uparrow cond LIST_INST \uparrow inst<sub>s</sub> \ \{\ r := 'while ('.cond.') \{'.inst_s.'\}'\}
PRINT\uparrow r \rightarrow
         \underline{\mathtt{Print}} \uparrow x [ \underline{\mathtt{LIST}} \underline{\mathtt{EXPR}} \uparrow e_s ] \{ r := (x ? '\mathtt{printx}(' : '\mathtt{print}(').e_s.'); ' \}
     Println\(\gamma\) [LIST_EXPR\(\frac{e}{e}\)] \{ r := (x ? \text{'printlnx}(' : '\text{println}(').e_s.');'\}
/****
                  Class related rules
                                                             ****/
\mathbf{DECL\_CLASS} \uparrow r \rightarrow
         <u>DeclClass</u>
               IDENTIFIER↑name
               IDENTIFIER↑super
               LIST_DECL_FIELD\fields
               LIST_DECL_METHOD\uparrow method<sub>s</sub>
         \{r := \text{`class'}.name.' \text{ extends'}.super.' \{\text{'.field}_s.method}_s.'\}'\}
\mathbf{DECL\_FIELD} \uparrow r \rightarrow
         DeclField\u00e9visib[
               IDENTIFIER↑type
              IDENTIFIER† field
              INITIALIZATION↑init
         \{ r := visib.', '.type.', '.field.init.';' \}
DECL_METHOD\uparrow r \rightarrow
         DeclMethod
              IDENTIFIER↑type
              IDENTIFIER↑name
               LIST_DECL_PARAM\(\)param_s
               METHOD\_BODY \uparrow body
         \{ r := type.', name.'('.param_s.')'.body \}
DECL PARAM\uparrow r \rightarrow
         <u>DeclParam</u>[IDENTIFIER\uparrow type IDENTIFIER\uparrow name] { r := type.' '.name}
METHOD BODY\uparrow r \rightarrow
         \underline{\texttt{MethodAsmBody}}[\mathbf{STRING}\underline{\mathbf{LITERAL}} code] \{ r := 'asm('.code.'); '\}
         \underline{\texttt{MethodBody}}[\ \underline{\textbf{LIST\_DECL\_VAR}} \\ var_s \ \underline{\textbf{LIST\_INST}} \\ inst_s] \ \{\ r := `\{`.var_s.inst_s.'\}'\}
```

4.2 Règles pour les ensembles de listes d'arbres

$[{f Syntaxe Contextuelle}]$

Syntaxe contextuelle du langage Deca

1 Introduction

Ce document décrit la syntaxe contextuelle du langage Deca, c'est-à-dire la notion de programme Deca « bien typé ». Tout programme Deca bien typé doit être compilé en un programme assembleur dont le comportement à l'exécution respecte la sémantique du programme source décrite dans [Semantique]. Réciproquement un programme Deca mal typé doit être rejeté par le compilateur avec un message d'erreur approprié.

La vérification contextuelle d'un programme Deca nécessite trois passages sur le programme. En effet, on peut, dans une déclaration de champ ou de paramètre d'une méthode, faire référence à une classe dont la déclaration apparaît *après* son utilisation. Pour cette raison, la première passe consiste à vérifier uniquement le nom des classes et la hiérarchie de classes. Lors de la deuxième passe, on vérifie les déclarations des champs et la signature des méthodes. D'autre part, les méthodes peuvent être mutuellement récursives. Par conséquent, on vérifie le corps des méthodes au cours d'une troisième passe.

Pour chaque passe, les vérifications à effectuer sont spécifiées formellement par une grammaire attribuée exprimée sur la syntaxe abstraite. Autrement dit, chacune de ces grammaires définit un ensemble d'arbres **program** (l'ensemble des arbres acceptées par cette grammaire attribuée) qui est un sous-ensemble de **PROGRAM**.

Par définition, les programmes Deca « bien typés » sont les programmes dont l'arbre de syntaxe abstraite est accepté successivement par chacune des grammaires attribuées de la syntaxe contextuelle.

2 Domaines d'attributs

Dans cette partie sont définis les domaines d'attributs et les opérations sur les attributs.

2.1 Définition des domaines

On définit d'une part des domaines de base, et d'autre part des domaines construits à partir des domaines de base à l'aide des opérations suivantes :

- $D_1 \times D_2$: produit cartésien des domaines D_1 et D_2 ;
- $D_1 \rightarrow D_2$: domaine des fonctions partielles de D_1 vers D_2 ; le domaine de définition d'une fonction f est noté dom(f);
- D^* : ensemble des listes d'éléments de D;
- $D_1 \cup D_2$: union des domaines D_1 et D_2 ;
- $f(D_1, \ldots D_n)$ est l'ensemble des termes de la forme $f(d_1, \ldots d_n)$, avec $(d_1, \ldots d_n) \in D_1 \times \ldots D_n$.

Soit Symbol le domaine des identificateurs.

Soit Type le domaine des types du langage Deca. Les types du langage Deca comportent <u>void</u>, <u>boolean</u>, <u>float</u>, <u>int</u> et <u>string</u>. De plus, à chaque classe A du programme correspond un type <u>type class</u>(A). On a également un type null qui représente le type du littéral Null.

```
Type \triangleq \{ \underline{\text{void}}, \underline{\text{boolean}}, \underline{\text{float}}, \underline{\text{int}}, \underline{\text{string}}, \underline{\text{null}} \} \cup \underline{\text{type\_class}}(\underline{\text{Symbol}})
```

Visibilité est le type énuméré suivant :

```
Visibility \triangleq \{ protected, public \}
```

Les identificateurs sont de différentes natures. Dans le domaine TypeNature, on distingue les identificateurs de type ou de classe. Dans le domaine ExpNature, on distingue les identificateurs de champ, de paramètre, de variable et de méthode : c'est-à-dire tous les identificateurs qui peuvent apparaître dans une expression Déca.

```
TypeNature \triangleq \{\underline{\text{type}}\} \cup \underline{\text{class}}(\text{Profil})
ExpNature \triangleq \{\underline{\text{param}}, \underline{\text{var}}\} \cup \underline{\text{method}}(\text{Signature}) \cup \underline{\text{field}}(\text{Visibility}, \text{Symbol})
```

Une nature $\underline{\text{field}(\text{public}, name)}$ représente un champ public de la classe name; $\underline{\text{field}(\text{protected}, name)}$ représente un champ protected de la classe name.

Une définition est un couple (nature, type).

```
\begin{array}{ll} \text{TypeDefinition} & \triangleq & \text{TypeNature} \times \text{Type} \\ \text{ExpDefinition} & \triangleq & \text{ExpNature} \times \text{Type} \end{array}
```

Un environnement est une fonction partielle des identificateurs vers les définitions. On distingue les environnements de types et les environnements d'identificateurs dans les expressions.

```
\begin{array}{ll} EnvironmentType & \triangleq & Symbol \rightarrow TypeDefinition \\ EnvironmentExp & \triangleq & Symbol \rightarrow ExpDefinition \end{array}
```

Une signature est une liste (ordonnée) de types.

```
Signature \triangleq Type*
```

Une extension de classe est le nom de sa super-classe, ou 0 pour la classe Object qui n'a pas de super-classe.

```
Extension \triangleq Symbol \cup {0}
```

Un profil de classe est constitué du nom de sa super-classe et d'un environnement qui contient les définitions des champs et méthodes de la classe.

```
Profil \triangleq Extension \times EnvironmentExp
```

Operator est l'ensemble des opérateurs du langage. Les règles (3.49) à (3.63) donnent la correspondance entre les noms d'opérateurs de la syntaxe abstraite et ceux du domaine Operator (utilisés pour spécifier le typage des opérateurs).

```
Operator \triangleq \{\text{plus}, \text{minus}, \text{mult}, \text{divide}, \text{mod}, \text{eq}, \text{neq}, \text{lt}, \text{gt}, \text{leq}, \text{geq}, \text{and}, \text{or}, \text{not}\}
```

2.2 Opérations et prédicats sur les domaines d'attributs

Notations

- Une fonction partielle f du domaine $D_1 \to D_2$ peut être notée par son graphe, c'est-à-dire " $\{d_1 \mapsto v_1, \ldots d_n \mapsto v_n\}$ ", où dom $(f) = \{d_1, \ldots d_n\}$ et $\forall i \in \{1, \ldots n\}$, $f(d_i) = v_i$. La fonction nulle part définie est donc notée " $\{\}$ ".
- Un élément du domaine $D_1 \times D_2$ est noté " (d_1, d_2) ", ou, s'il n'y a pas d'ambiguïté, " d_1, d_2 ", avec $d_1 \in D_1$ et $d_2 \in D_2$.

— Un élément du domaine D^* est noté " $[d_1, d_2, \dots d_n]$ ", où les d_i sont des éléments de D. La séquence vide est notée "[]".

On note $e \cdot l$ l'ajout en tête de l'élément e dans la liste l.

$$e \cdot [e_1, e_2, \dots e_n] = [e, e_1, e_2, \dots e_n].$$

On note l@m la concaténation des listes l et m.

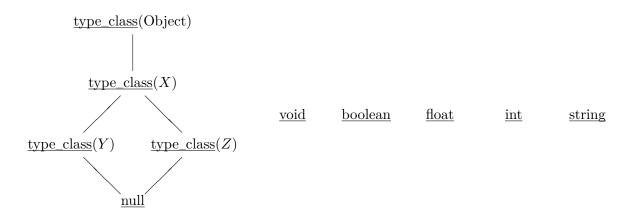
$$[e_1, e_2, \dots e_n] @ [f_1, f_2, \dots f_p] = [e_1, e_2, \dots e_n, f_1, f_2, \dots f_p].$$

Relation de sous-typage

Soit un environnement de types env (de EnvironmentType). La relation de sous-typage relative à l'environnement env est définie de la façon suivante :

- Pour tout type T, T est un sous-type de T.
- Pour toute classe A, $\underline{\text{type_class}}(A)$ est un sous-type de $\underline{\text{type_class}}(\text{Object})$.
- Si une classe B étend une classe A dans l'environnement env, alors $\underline{\text{type_class}}(B)$ est un sous-type de $\underline{\text{type_class}}(A)$.
- Si une classe C étend une classe B dans l'environnement env et si $\underline{\text{type_class}}(B)$ est un sous-type de T, alors $\underline{\text{type_class}}(C)$ est un sous-type de T.
- Pour toute classe A, null est un sous-type de type_class(A).

Par exemple, si on a une classe X et deux classes Y et Z qui sont sous-classes de X, la relation de sous-typage a la forme suivante :



Si T_1 est un sous-type de T_2 relativement à l'environnement env, on note subtype (env, T_1, T_2) .

Compatibilité pour l'affectation

Soit un environnement de types env (de EnvironmentType). On peut affecter à un objet de type T_1 une valeur de type T_2 si l'une des conditions suivantes est satisfaite :

- T_1 est le type <u>float</u> et T_2 est le type <u>int</u>;
- subtype(env, T_2, T_1).

On note assign_compatible(env, T_1, T_2).

Compatibilité pour la conversion

Soit un environnement de types env (de EnvironmentType). On peut convertir une valeur de type T_1 en une valeur de type T_2 si T_1 n'est pas le type \underline{void} et l'une des conditions suivantes est satisfaite :

- assign_compatible(env, T_1, T_2);
- assign_compatible(env, T_2, T_1).

On note cast_compatible (env, T_1, T_2) .

Signature des opérateurs

On définit trois opérations partielles : type_unary_op, type_arith_op et type_binary_op, qui permettent de calculer respectivement le type du résultat d'un opérateur unaire, d'un opérateur arithmétique binaire et d'un opérateur binaire quelconque.

```
type\_unary\_op : Operator \times Type \rightarrow Type
type\_unary\_op(\underline{minus}, \underline{int})
                                                          int
                                                   \triangleq
type_unary_op(minus, float)
                                                          float
type\_unary\_op(\underline{not}, \underline{boolean})
                                                          boolean
type_arith_op : Type \times Type \rightarrow Type
type_arith_op(int, int)
                                                     int
                                              \triangleq
type_arith_op(int, float)
                                                     <u>float</u>
type_arith_op(\underline{float},\underline{int})
                                                     <u>float</u>
type_arith_op(\underline{float},\underline{float})
                                              \triangleq
                                                     float
type\_binary\_op: Operator \times Type \times Type \rightarrow Type
type_binary_op(op, T_1, T_2) \triangleq type_arith_op(T_1, T_2),
     si op \in \{\text{plus}, \text{minus}, \text{mult}, \text{divide}\}
type\_binary\_op(mod, int, int) \triangleq int
type_binary_op(op, T_1, T_2) \triangleq boolean,
      si op \in \{\underline{eq}, \underline{neq}, \underline{lt}, \underline{gt}, \underline{leq}, \underline{geq}\}
      et (T_1, T_2) \in \text{dom}(\text{type\_arith\_op})
type_binary_op(op, T_1, T_2) \triangleq boolean,
      si\ op \in \{\underline{eq}, \underline{neq}\}
      et (T_1 = \underline{\text{type\_class}}(A) \text{ ou } T_1 = \underline{\text{null}})
      et (T_2 = \underline{\text{type\_class}}(B) \text{ ou } T_2 = \underline{\text{null}})
On autorise donc la comparaison de deux objets de classes A et B quelconques.
type_binary_op(op, boolean, boolean) \triangleq boolean,
      si op \in \{\underline{and}, \underline{or}, \underline{eq}, \underline{neq}\}
```

Le prédicat type_instanceof_op indique si on peut appliquer l'opération <u>InstanceOf</u>.

```
type_instanceof_op : Type × Type → Type
type_instanceof_op(T_1, T_2) \triangleq boolean,
si (T_1 = \underline{\text{type\_class}}(A) ou T_1 = \underline{\text{null}})
et T_2 = \underline{\text{type\_class}}(B)
```

2.3 Environnements

Définition des environnements prédéfinis

Les identificateurs prédéfinis du langage Deca sont définis par deux environnements : env_types_predef dans EnvironmentType qui définit les types prédéfinis, et env_exp_object de EnvironmentExp qui est associé à la classe prédéfinie Object.

```
\begin{array}{lll} & \text{env\_exp\_object} & \triangleq & \{ & \text{equals} \mapsto (\underline{\text{method}}([\underline{\text{type\_class}}(\text{Object})]), \, \underline{\text{boolean}}) \\ & \text{env\_types\_predef} & \triangleq & \{ & \text{void} \mapsto (\underline{\text{type}}, \, \underline{\text{void}}), \\ & & \text{boolean} \mapsto (\underline{\text{type}}, \, \underline{\text{boolean}}), \\ & & \text{float} \mapsto (\underline{\text{type}}, \, \underline{\text{float}}), \\ & & \text{int} \mapsto (\underline{\text{type}}, \, \underline{\text{int}}), \\ & & \text{Object} \mapsto (\underline{\text{class}}(0, \, \underline{\text{env\_exp\_object}}), \, \underline{\text{type\_class}}(\underline{\text{Object}})) & \} \end{array}
```

Opérations sur les environnements

On définit deux opérations sur les environnements : l'empilement de deux environnements, noté /, et l'union disjointe de deux environnements, notée \oplus . L'opération \oplus est partielle sur les environnements. Dans les définitions ci-dessous, le domaine Environment est soit EnvironmentType, soit EnvironmentExp.

```
— Empilement

/: Environment × Environment → Environment

\forall x \in \text{Symbol}, \ (env_1/env_2)(x) \triangleq env_1(x), \quad \text{si } x \in \text{dom}(env_1),

\triangleq env_2(x), \quad \text{si } x \notin \text{dom}(env_1) \text{ et } x \in \text{dom}(env_2).

— Union disjointe

⊕: Environment × Environment → Environment

env_1 \oplus env_2 \text{ n'est pas défini si dom}(env_1) \cap \text{dom}(env_2) \neq \varnothing.

Sinon, \forall x \in \text{Symbol}, \ (env_1 \oplus env_2)(x) \triangleq env_1(x), \quad \text{si } x \in \text{dom}(env_1)

\triangleq env_2(x), \quad \text{si } x \in \text{dom}(env_2).
```

3 Conventions d'écriture

Les grammaires de chaque passe utilisent le même ensemble de terminaux que celui de [SyntaxeAbstraite]. Les non-terminaux de ces grammaires sont soit tout en minuscules, soit tout en majuscules, avec les règles suivantes.

- Un non-terminal en majuscule correspond exactement à un ensemble d'arbres associé au même non-terminal dans la grammaire attribuée de [SyntaxeAbstraite]. Les règles associées à la définition de cet ensemble d'arbres ne sont pas répétées.
- Un non-terminal en minuscule qui a exactement son homologue en majuscule dans la grammaire attribuée de [Decompilation] définit un sous-ensemble de son homologue.
- Les non-terminaux en minuscule qui n'ont pas d'homologue en majuscule, mais qui ont un homologue en <u>CamelCase</u> correspondant à un terminal, représentent des ensembles d'arbres dont la racine est ce nom de nœud.
- Les autres non-terminaux (en minuscule) ne correspondent pas forcément à un ensemble d'arbres : ils peuvent parfois correspondre à des ensembles de "morceaux" d'arbres (en général, un commentaire introduit leur signification).

Par ailleurs, les non-terminaux préfixés par "list" désignent toujours des listes d'arbres.

Enfin, on utilise aussi les notations suivantes sur les attributs :

- les attributs synthétisés sont préfixés par ↑;
- les attributs hérités sont préfixés par ↓.

3.1 Affectation des attributs

Pour toute règle, les attributs synthétisés du non-terminal en partie gauche et les attributs hérités des non-terminaux en partie droite doivent être affectés. Ces affectations peuvent être effectuées de deux manières différentes : 1. explicitement en utilisant une clause *affectation*; 2. implicitement par une expression fonctionnelle.

1. Affectation explicite de la forme affectation v := exp. Par exemple, la règle (0.1)

```
identifier \downarrow env\_exp \uparrow def
\rightarrow \quad \underline{Identifier} \uparrow name
affectation \quad def := env\_exp(name)
```

signifie qu'à l'attribut synthétisé $\uparrow def$ du non-terminal **identifier** est affecté la valeur $env_exp(name)$.

2. Affectation implicite par une expression fonctionnelle. L'exemple précédent peut également s'écrire :

```
identifier \downarrow env\_exp \uparrow env\_exp(name) \rightarrow <u>Identifier</u> \uparrow name.
```

3.2 Conditions sur les attributs

Les valeurs d'attributs, pour une règle de grammaire, peuvent être contraintes. Ces contraintes peuvent être exprimées de trois manières différentes : 1. explicitement par une condition logique sur les valeurs d'attributs ; 2. lors de l'affectation des attributs, en imposant l'existence d'une valeur (dans le cas des opérations partiellement définies) ; 3. implicitement en contraignant par filtrage les valeurs possibles d'attributs.

1. Utilisation d'une clause **condition** P, où P est une condition logique. Si P est faux, la clause n'est pas respectée. Par exemple, la règle (3.28)

```
rvalue \downarrow env\_types \downarrow env\_exp \downarrow class \downarrow type_1

\rightarrow expr \downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type_2

condition assign_compatible(env_types, type_1, type_2)
```

impose que les types $type_1$ et $type_2$ sont compatibles pour l'affectation.

NB: les noms "rvalue" et "lvalue" viennent de la règle (3.32) sur Assign.

2. Par affectation : toute valeur d'attribut doit être définie. Par exemple,

```
identifier \downarrow env\_exp \uparrow def
\rightarrow \underline{Identifier} \uparrow name

affectation def := env\_exp(name)
```

contraint $env_{-}exp(name)$ à être défini.

3. Par filtrage : on impose une forme particulière pour un attribut hérité dans une partie gauche de règle, ou pour un attribut synthétisé pour une partie droite de règle. Par exemple, la règle (3.29)

```
 \begin{array}{ccc} \mathbf{condition} \downarrow env\_types \downarrow env\_exp \downarrow class \\ \rightarrow & \mathbf{expr} \downarrow env\_types \downarrow env\_exp \downarrow class \uparrow \underline{\mathbf{boolean}} \end{array}
```

impose que la valeur de l'attribut synthétisé de \mathbf{expr} soit le type $\underline{\mathbf{boolean}}$.

```
Dans la règle (3.73)
```

```
rvalue_star \downarrow env\_types \downarrow env\_exp \downarrow class \downarrow [\ ]
\rightarrow \quad \varepsilon
```

on impose que la signature héritée en partie gauche soit la signature vide ([]).

3.3 Marqueur spécial '_' de noms inutiles

Le symbole '_' est destiné à remplacer un nom de valeur qui n'a qu'une occurrence (et n'est donc pas « utilisé »). Autrement dit ce symbole n'est pas lui-même un nom (chaque occurrence de ce symbole correspond à un nouveau nom inutilisé). Par exemple, la condition de la règle (1.3)

condition
$$env_types(super) = (\underline{class}(_), _)$$

signifie la même chose que

condition
$$env_types(super) = (\underline{class}(profil), type)$$

avec profil et type deux noms qui n'apparaissent nulle part ailleurs dans le contexte.

3.4 Égalité exprimant une définition

Certaines égalités apparaissant dans les conditions expriment une définition de nouveaux noms. C'est typiquement le cas, quand des noms impliqués dans l'égalité ne sont pas contraints ailleurs dans la règle. Par exemple, ces noms apparaîssent uniquement comme attributs hérités dans le membre droit.

Pour faciliter la lecture de ce type d'égalité, on utilise l'opérateur \triangleq qui indique que l'égalité réalise une définition des symboles apparaissant dans son membre gauche. Cet opérateur a bien la même sémantique que =. Il sert juste à faciliter la lecture des règles en donnant une indication sur l'effet du =.

Par exemple, dans la règle (2.3), la condition utilise une égalité qui réalise une définition de env_exp_super :

$$condition$$
 $(class(_,env_exp_super),_) \triangleq env_types(super)$

3.5 Règles avec syntaxe spéciale pour les non-terminaux préfixés par list

Comme dans la grammaire de [**Decompilation**], les règles portant sur les ensembles de listes d'arbres ont une syntaxe spécifique étendant légèrement celle décrite dans la section 1.3 de [**Decompilation**]. En effet,

- elles peuvent maintenant comporter plusieurs attributs hérités ou synthétisés :
- lorsqu'il n'y a pas d'attribut synthétisé, il n'y a pas d'action d'affectation;
- l'itération de l'affectation des attributs synthétisés peut être implicite. Ainsi, dans la règle cidessous (numérotée (1.2)), env_types_r reçoit initialement la valeur de l'attribut hérité env_types , puis à chaque itération sa valeur courante est passée comme attribut hérité de $decl_class$, et il reçoit en sortie de l'itération la valeur de l'attribut synthétisé par ce non-terminal.

```
list\_decl\_class \downarrow env\_types \uparrow env\_types_r
```

```
 \rightarrow \{env\_types_r := env\_types\} \\ [(\mathbf{decl\_class} \downarrow env\_types_r \uparrow env\_types_r)^*]
```

On aurait aussi pu écrire cette règle en utilisant un nouvel attribut tmp:

```
list\_decl\_class \downarrow env\_types \uparrow env\_types_r
```

```
\rightarrow \{env\_types_r := env\_types\} \\ [(\mathbf{decl\_class} \downarrow env\_types_r \uparrow tmp \{env\_types_r := tmp\})^*]
```

Les règles sur les non-terminaux préfixés par **list** n'utilisent pas de conditions explicites (les conditions explicites sont exprimées dans le non-terminal portant sur les éléments). Par contre, les opérations partielles (comme \oplus) expriment des conditions implicites (cf. section 3.2).

4 Règles communes aux trois passes de vérifications contextuelles

Deux règles, communes aux trois passes de vérifications contextuelles, sont détaillées ici.

4.1 Identificateurs dans les expressions

$$identifier \downarrow env_exp \uparrow def$$

$$\rightarrow \quad \underline{Identifier} \uparrow name$$

$$affectation \quad def := env_exp(name)$$

$$(0.1)$$

On doit trouver une définition associée au nom name dans l'environnement env_exp.

4.2 Identificateurs de types

$$type \downarrow env_types \uparrow type \\ \rightarrow \underline{Identifier} \uparrow name \\ condition \quad (_, type) \triangleq env_types(name)$$
 (0.2)

5 Grammaire attribuée spécifiant la passe 1

Lors de la passe 1, on vérifie le nom des classes et la hiérarchie de classes. On construit un premier environnement, qui contient env_types_predef, ainsi que les noms des différentes classes du programme. Cet environnement contient un profil incomplet pour chaque classe du programme : celui-ci contient la super-classe, mais ne contient pas l'environnement des champs et méthodes de la classe.

```
list\_decl\_class \downarrow env\_types \uparrow env\_types_r 
\rightarrow \{env\_types_r := env\_types\} 
[(decl\_class \downarrow env\_types_r \uparrow env\_types_r)^*]
(1.2)
```

L'identificateur super doit être un identificateur de classe préalablement déclaré. L'environnement associé à la classe est laissé vide ($\{\}$), et sera complété pendant la passe suivante.

6 Grammaire attribuée spécifiant la passe 2

Lors de la passe 2, on vérifie les champs et la signature des méthodes des différentes classes. On hérite de l'environnement construit au cours de la passe 1 (attribut env_types₀ hérité par le non-terminal **program**). On construit un environnement qui contient les types prédéfinis et les classes du programme : le profil de chaque classe contient le nom de sa super-classe et l'environnement des champs et méthodes de la classe.

6.1 Programmes

```
list\_decl\_class \downarrow env\_types \uparrow env\_types_r 
\rightarrow \{env\_types_r := env\_types\} 
[(decl\_class \downarrow env\_types_r \uparrow env\_types_r)^*]
(2.2)
```

6.2 Déclarations de classes

Le corps d'une classe (non-terminaux list_decl_field et list_decl_method) est analysé dans l'environnement env_types , qui est l'environnement créé par la passe 1, complété au fur et à mesure avec les définitions complètes de classes. Le non-terminal list_decl_method hérite du nom de la super-classe, qui servira à vérifier la signature d'une méthode redéfinie, règle (2.7). Le non-terminal list_decl_field hérite en plus du nom de la classe lui-même (class), qui servira à construire la définition des champs, règle (2.5). Leurs attributs synthétisés env_exp_m et env_exp_f contiennent respectivement l'environnement des méthodes et celui des champs de la classe.

Pour la règle suivante, si la passe 1 a été correctement effectuée, alors $name \in \text{dom}(env_types)$. De plus, dans env_types , super est associé à une définition de classe. La condition sert donc simplement à récupérer l'environnement env_exp_super associé à la classe super. Ainsi, l'empilement $\{name \mapsto new_def\}/env_types$ complète la définition de name laissée incomplète par la passe 1. En pratique, une implémentation pourra simplement ajouter les nouvelles définitions à l'environnement contenu dans la définition de classe construite en passe 1. Il n'est pas nécessaire de créer une nouvelle définition de classe et l'empilement d'environnement peut être fait dès la création de la définition de classe en passe 1.

6.3 Déclarations de champs

```
\mathbf{list\_decl\_field} \downarrow env\_types \downarrow super \downarrow class \uparrow env\_exp_r 
\rightarrow \quad \{env\_exp_r := \{\}\} 
[ (\mathbf{decl\_field} \downarrow env\_types \downarrow super \downarrow class \uparrow env\_exp 
\{env\_exp_r := env\_exp_r \oplus env\_exp\})^* ]
(2.4)
```

Dans la règle suivante, l'attribut *type* contient le type du champ déclaré, L'attribut *visib* vaut <u>protected</u> si le champ est protégé, et vaut <u>public</u> sinon. Par ailleurs, si l'identificateur *name* est déjà défini dans l'environnement des expressions de la super-classe, alors ce doit être un identificateur de champ.

$$\begin{aligned} \operatorname{\mathbf{decl_field}} \downarrow env_types \downarrow super \downarrow class \uparrow \{name \mapsto (\operatorname{\underline{field}}(visib, class), type)\} \\ & \to \quad \underbrace{\operatorname{\mathbf{DeclField}} \uparrow visib}[\\ & \quad \operatorname{\mathbf{type}} \downarrow env_types \uparrow type \\ & \quad \operatorname{\underline{\mathbf{Identifier}}} \uparrow name \\ & \quad \operatorname{\mathbf{INITIALIZATION}} \\ \end{bmatrix} \\ & \quad \operatorname{\mathbf{condition}} \quad type \neq \underbrace{\operatorname{void}}_{} \text{ et }, \\ & \quad \operatorname{\mathbf{si}} \left\{ \begin{array}{c} (\operatorname{\underline{\mathbf{class}}}(_, env_exp_super), _) \triangleq env_types(super) \\ \text{ et } env_exp_super(name) \text{ est défini} \\ \text{ alors } env_exp_super(name) = (\underline{\operatorname{field}}(_, _), _). \\ \end{array} \right. \end{aligned}$$

6.4 Déclarations de méthodes

$$list_decl_method \downarrow env_types \downarrow super \uparrow env_exp_r$$

$$\rightarrow \{env_exp_r := \{\}\}$$

$$[(decl_method \downarrow env_types \downarrow super \uparrow env_exp$$

$$\{env_exp_r := env_exp_r \oplus env_exp\})^*]$$

$$(2.6)$$

Si une méthode est redéfinie, alors celle-ci :

- doit avoir la même signature que la méthode héritée;
- doit avoir pour type de retour un sous-type du type de retour de la méthode héritée.

L'attribut synthétisé du non-terminal **list_decl_param** permet de construire la signature de la méthode à partir du type des paramètres.

$$list_decl_param \downarrow env_types \uparrow sig$$

$$\rightarrow \{sig := []\}$$

$$[(decl_param \downarrow env_types \uparrow type \{sig := sig @ [type]\})^*]$$

$$(2.8)$$

$$\frac{\text{decl_param} \downarrow env_types \uparrow type}{\rightarrow \quad \text{DeclParam}[\text{ type } \downarrow env_types \uparrow type \text{ Identifier } \uparrow_]}{condition \quad type \neq \text{void}} \tag{2.9}$$

7 Grammaire attribuée spécifiant la passe 3

Lors de la passe 3, on vérifie les blocs, les instructions, les expressions et les initialisations. On construit un environnement qui contient les champs et les méthodes, ainsi que les paramètres des méthodes et les variables locales.

La plupart des non-terminaux ont un attribut hérité *class* qui représente la classe dans laquelle les déclarations ou les instructions apparaissent. L'attribut *class* a pour valeur 0 lorsque les déclarations ou les instructions apparaissent dans le programme principal. Cet attribut permet :

- de vérifier que <u>This</u> n'apparaît pas dans le programme principal et de construire son type, règle (3.43);
- de vérifier que, dans une sélection de champ protégé, le type de l'expression est un sous-type de la classe analysée, règle (3.66).

7.1 Programmes

L'environnement hérité env_types du non-terminal **program** est l'environnement synthétisé en passe 2.

$$list_decl_class \downarrow env_types \\ \rightarrow [(decl_class \downarrow env_types)^*]$$
(3.2)

$$\begin{array}{ccc} \mathbf{main} \downarrow env_types & & & & & \\ & \rightarrow & \underline{\mathtt{EmptyMain}} & & & \\ & \rightarrow & \underline{\mathtt{Main}} \ \mathbf{bloc} \downarrow env_types \downarrow \{\} \downarrow \{\} \downarrow \theta \downarrow \underline{\mathtt{void}} & & & \\ \end{array} \tag{3.4}$$

Le non-terminal **bloc** définit un sous-ensemble de "[LIST_DECL_VAR LIST_INST]" (c'est un ensemble de paires d'arbres) et permet de factoriser le traitement des nœuds <u>Main</u> et <u>MethodBody</u>. Dans le cas de <u>MethodBody</u>, **bloc** hérite d'un environnement contenant les déclarations de champs et de méthodes de la classe courante, et d'un autre contenant les déclarations de paramètres. Dans le cas de <u>Main</u>, ces deux environnements sont vides. L'attribut *class* vaut 0 et l'attribut *return* vaut <u>void</u> car on analyse le programme principal.

7.2 Déclarations de classes

Si les deux premières passes ont été effectuées correctement, env_types contient forcément une classe de nom class. Cette condition sert donc simplement à récupérer l'environnement des champs et méthodes de la classe.

7.3 Déclarations de champs

$$list_decl_field \downarrow env_types \downarrow env_exp \downarrow class \rightarrow [(decl_field \downarrow env_types \downarrow env_exp \downarrow class)^*]$$
(3.6)

Ci-dessous, le non-terminal **rvalue** correspond au sous-ensemble des expressions compatibles avec le type type.

```
initialization \downarrow env\_types \downarrow env\_exp \downarrow class \downarrow type (3.8)

\rightarrow \quad \underline{\text{Initialization}} \left[ \text{ rvalue } \downarrow env\_types \downarrow env\_exp \downarrow class } \downarrow type \right]
\rightarrow \quad \text{NoInitialization}  (3.9)
```

7.4 Déclarations de méthodes

```
list_decl_method \downarrow env\_types \downarrow env\_exp \downarrow class (3.10)

→ [ (decl_method \downarrow env\_types \downarrow env\_exp \downarrow class)* ]
```

Pour analyser une méthode (règle (3.11)), on analyse d'abord les paramètres : on construit l'environnement des paramètres. Puis, dans le non-terminal **bloc**, on complète cet environnement avec les déclarations locales, avant d'analyser les instructions dans l'environnement résultant de ces déclarations.

```
\begin{array}{ll}
\operatorname{list\_decl\_param} \downarrow env\_types \uparrow env\_exp_r \\
\to & \{env\_exp_r := \{\}\} \\
& \left[ (\operatorname{decl\_param} \downarrow env\_types \uparrow env\_exp \\
& \{env\_exp_r := env\_exp_r \oplus env\_exp\}\}^* \right]
\end{array} \tag{3.12}
```

```
\begin{aligned} \mathbf{decl\_param} \downarrow env\_types \uparrow \{ name \mapsto (\underline{param}, type) \} \\ &\rightarrow \underline{\quad DeclParam} \left[ \mathbf{type} \downarrow env\_types \uparrow type \ \underline{\quad Identifier} \uparrow name \ \right] \end{aligned} \tag{3.13}
```

L'attribut return contient le type de retour de la méthode. Cet attribut est utilisé par le non-terminal **bloc** pour vérifier les instructions Return, règle (3.24).

7.5 Déclarations de variables

Dans les règles suivantes, env_exp_sup représente l'environnement de la classe englobante ou l'environnement vide dans le programme principal. L'environnement env_exp est l'environnement contenant les variables déclarées (et les paramètres). L'initialisation des variables est analysée dans l'environnement env_exp/env_exp_sup .

```
list\_decl\_var \downarrow env\_types \downarrow env\_exp\_sup \downarrow env\_exp \downarrow class \uparrow env\_exp_r 
\rightarrow \{env\_exp_r := env\_exp\} 
[(decl\_var \downarrow env\_types \downarrow env\_exp\_sup \downarrow env\_exp_r \downarrow class 
\uparrow env\_exp_r)^*]
(3.16)
```

7.6 Blocs et instructions

Les non-terminaux bloc, list_inst et inst ont plusieurs attributs hérités :

- env_types qui représente l'environnement des types;
- env_exp_sup représente l'environnement de l'éventuelle classe englobante (pour **bloc** uniquement);
- *env_exp* qui représente l'environnement (éventuellement vide) des paramètres dans le cas de **bloc**, ou l'environnement (tout court) pour les autres;
- class qui représente le nom de la classe où apparaît l'instruction, si l'instruction apparaît dans une classe, et 0 sinon;
- return qui représente le type de retour de la méthode, si l'instruction apparaît dans une méthode, et <u>void</u> sinon.

```
bloc \downarrow env\_types \downarrow env\_exp\_sup \downarrow env\_exp \downarrow class \downarrow return \rightarrow [

\begin{array}{c} \textbf{list\_decl\_var} \downarrow env\_types \downarrow env\_exp\_sup \downarrow env\_exp \\ \downarrow class \uparrow env\_exp\_r \\ \textbf{list\_inst} \downarrow env\_types \downarrow env\_exp\_r/env\_exp\_sup \downarrow class \\ \downarrow return \end{array}
```

```
list_inst \downarrow env_types \downarrow env_exp \downarrow class \downarrow return 
\rightarrow [(inst \downarrow env_types \downarrow env_exp \downarrow class \downarrow return)^*] 
(3.19)
```

Ci-dessous, le non-terminal **print** correspond juste à l'ensemble des 2 noms de nœuds {Print,Println}. Le non-terminal list_exp_print correspond aux listes de paramètres effectifs imprimables.

```
 \begin{array}{c} \rightarrow \quad \underline{ \mbox{IfThenElse}[ } \\ & \quad \mbox{condition} \downarrow env\_types \downarrow env\_exp \downarrow class \\ & \quad \mbox{list\_inst} \downarrow env\_types \downarrow env\_exp \downarrow class \downarrow return \\ & \quad \mbox{list\_inst} \downarrow env\_types \downarrow env\_exp \downarrow class \downarrow return \\ \mbox{]} \\ \rightarrow \quad \underline{ \mbox{NoOperation} } \\ \rightarrow \quad \underline{ \mbox{Return}} \left[ \begin{array}{c} \mathbf{rvalue} \downarrow env\_types \downarrow env\_exp \downarrow class \downarrow return \end{array} \right] \\ \mathbf{condition} \\ \rightarrow \quad \underline{ \mbox{Vhile}} \left[ \\ \quad \mbox{condition} \downarrow env\_types \downarrow env\_exp \downarrow class \\ \quad \mbox{list\_inst} \downarrow env\_types \downarrow env\_exp \downarrow class \downarrow return \end{array} \right] \\ (3.25) \\ \end{array}
```

$$\mathbf{print} \qquad \rightarrow \quad \underline{\mathbf{Print}} \tag{3.26}$$

$$\rightarrow$$
 Println (3.27)

7.7 Expressions

Le non-terminal **rvalue** correspond aux sous-ensembles des expressions compatibles avec le type $type_1$.

rvalue
$$\downarrow env_types \downarrow env_exp \downarrow class \downarrow type_1$$
 (3.28)
 $\rightarrow \exp r \downarrow env_types \downarrow env_exp \downarrow class \uparrow type_2$
condition assign_compatible(env_types, type_1, type_2)

Le non-terminal condition correspond aux sous-ensembles des expressions booléennes.

condition
$$\downarrow env_types \downarrow env_exp \downarrow class$$
 $\rightarrow \exp r \downarrow env_types \downarrow env_exp \downarrow class \uparrow boolean$ (3.29)

Le non-terminal **exp_print** correspond aux ensembles d'expressions imprimables.

$$list_exp_print \downarrow env_types \downarrow env_exp \downarrow class \rightarrow [(exp_print \downarrow env_types \downarrow env_exp \downarrow class)^*]$$
(3.30)

Ci-dessous, le non-terminal **op_bin** (respectivement **op_un**) contient l'ensemble des nœuds correspondant à un opérateur binaire (resp. unaire) de Operator. Ces deux non-terminaux synthétisent l'opérateur correspondant. Le non-terminal **literal** contient l'ensemble des littéraux (plus la constante <u>Null</u>). Il retourne le type de ce littéral.

 \rightarrow op_bin $\uparrow op$

(3.33)

```
expr \downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type_1
                                          expr \downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type_2
             affectation type := type\_binary\_op(op, type_1, type_2)
                                  lvalue \downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type
                                                                                                                                              (3.34)
                                   \underline{\mathtt{ReadInt}}
                                                                                                                                              (3.35)
             affectation
                                 type := \underline{int}
                                                                                                                                              (3.36)
                                   ReadFloat
             affectation
                                 type := float
                                  op\_un \uparrow op [expr \downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type_1]
                                                                                                                                              (3.37)
             affectation type := type\_unary\_op(op, type_1)
                            \rightarrow literal \uparrow type
                                                                                                                                              (3.38)
                                  <u>Cast</u>
                                                                                                                                              (3.39)
                                          type \downarrow env\_types \uparrow type
                                          expr \downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type_2
                condition cast_compatible(env\_types, type_2, type)
                                   <u>InstanceOf</u>
                                                                                                                                              (3.40)
                                          expr \downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type_1
                                          type \downarrow env\_types \uparrow type_2
             affectation type := type_instanceof_op(type_1, type_2)
                                  method_call \downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type
                                                                                                                                              (3.41)
                                  \underline{\text{New}} \left[ \text{ type } \downarrow env\_types \uparrow type \right]
                                                                                                                                              (3.42)
                condition type = \underline{type\_class}(\underline{\hspace{0.1cm}})
                                  This
                                                                                                                                              (3.43)
                condition class \neq 0
             affectation \quad type := \underline{type\_class}(class)
literal ↑int
                                                                                                                                              (3.44)
                                   IntLiteral
\mathbf{literal}\uparrow \underline{\mathrm{float}}
                                                                                                                                              (3.45)
                                  <u>FloatLiteral</u>
literal ↑string
                                  \underline{StringLiteral}
                                                                                                                                              (3.46)
literal ↑boolean
                                  BooleanLiteral
                                                                                                                                              (3.47)
                            \rightarrow
literal ↑null
                                   Null
                                                                                                                                              (3.48)
op_bin ↑divide
                                  Divide
                                                                                                                                              (3.49)
\mathbf{op\_bin} \uparrow \underline{\mathrm{minus}}
                                                                                                                                              (3.50)
                                  Minus
op_bin ↑mod
                                                                                                                                              (3.51)
                                   Modulo
op_bin ↑mult
                                                                                                                                              (3.52)
                                  Multiply
op_bin ↑plus
                                                                                                                                              (3.53)
                                  Plus
                            \rightarrow
op_bin ↑and
                                   And
                                                                                                                                              (3.54)
```

7.8 Expressions affectables (en partie à gauche d'une affectation)

Le non-terminal **field_ident** (resp. **lvalue_ident**) correspond à l'ensemble des noms de champ (resp. identificateurs affectables) dans le contexte. Ces deux non-terminaux synthétisent le type associé à l'identificateur.

```
lvalue \downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type
                                                                                                                                                               (3.64)
                                \rightarrow lvalue_ident \downarrow env\_exp \uparrow type
                                       Selection
                                                                                                                                                               (3.65)
                                               expr \downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type\_class(class_2)
                                              field_ident \downarrow env\_exp_2 \uparrow \underline{\text{public}} \uparrow \underline{\hspace{0.5cm}} \uparrow type
                  condition (\underline{class}(\underline{\ },env\_exp_2),\underline{\ }) \triangleq env\_types(class_2)
                                      Selection
                                                                                                                                                               (3.66)
                                               expr \downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type\_class(class_2)
                                               field\_ident \downarrow env\_exp_2 \uparrow protected \uparrow class\_field \uparrow type
                                      (class(\_, env\_exp_2), \_) \triangleq env\_types(class_2)
                  condition
                                       et subtype(env\_types, \underline{type\_class}(class_2), \underline{type\_class}(class))
                                       et subtype(env\_types, \underline{type\_class}(class), \underline{type\_class}(class\_field))
```

La règle (3.66) expriment les contraintes relatives à la visibilité des champs protégés. L'intuition et la raison d'être de ces contraintes sont détaillées dans la section 9.

lvalue_ident ↓
$$env_exp \uparrow type$$
 (3.67)
→ identifier ↓ $env_exp \uparrow (\underline{field}(_,_), type)$
→ identifier ↓ $env_exp \uparrow (\underline{param}, type)$ (3.68)
→ identifier ↓ $env_exp \uparrow (var, type)$ (3.69)

L'attribut synthétisé *visib* de **field_ident** vaut <u>public</u> si le champ est public, et vaut <u>protected</u> sinon. L'attribut synthétisé *class* est le nom de la classe où le champ est déclaré. L'attribut synthétisé *type* représente le type du champ.

$$\begin{array}{ccc} \mathbf{field_ident} \downarrow env_exp \uparrow visib \uparrow class \uparrow type \\ & \rightarrow & \mathbf{identifier} \downarrow env_exp \uparrow (\underline{\mathbf{field}}(visib, class), type) \end{array}$$

7.9 Appels de méthode

L'objet sur lequel est invoqué la méthode doit être d'un type correspondant à une classe $class_2$. Le non-terminal **method_ident** correspond à l'ensemble des identificateurs correspondant à un nom de méthode dans l'environnement env_exp . Il synthétise la signature sig et le type de retour type de cette méthode. Le non-terminal **rvalue_star** définit un sous-ensemble de "**EXPR***" correspondant aux suites de paramètres effectifs compatibles avec la signature sig retournée par **method_ident**.

$$\mathbf{rvalue_star} \downarrow env_types \downarrow env_exp \downarrow class \downarrow [\]$$

$$\rightarrow \quad \varepsilon$$

$$(3.73)$$

$$\mathbf{rvalue_star} \downarrow env_types \downarrow env_exp \downarrow class \downarrow (type \cdot sig)$$

$$\rightarrow \mathbf{rvalue} \downarrow env_types \downarrow env_exp \downarrow class \downarrow type$$

$$\mathbf{rvalue_star} \downarrow env_types \downarrow env_exp \downarrow class \downarrow sig$$

$$(3.74)$$

8 Profils d'attributs des symboles non-terminaux et terminaux

8.0 Profils communs aux trois passes

Identificateurs dans les expressions

 $\label{eq:continuous} \begin{array}{l} \mathbf{identifier} \downarrow & \mathbf{EnvironmentExp} \uparrow & \mathbf{ExpDefinition} \\ \underline{\mathbf{Identifier}} \uparrow & \mathbf{Symbol} \end{array}$

Identificateurs de types

type ↓EnvironmentType ↑Type

8.1 Passe 1

program ↑EnvironmentType
list_decl_class ↓EnvironmentType ↑EnvironmentType
decl_class ↓EnvironmentType ↑EnvironmentType

8.2 Passe 2

Programmes

 $\begin{array}{l} \textbf{program} \downarrow EnvironmentType \uparrow EnvironmentType \\ \textbf{list_decl_class} \downarrow EnvironmentType \uparrow EnvironmentType \\ \end{array}$

Déclarations de classes

decl_class ↓EnvironmentType ↑EnvironmentType

Déclarations de champs

Déclarations de Méthodes

list_decl_method ↓EnvironmentType ↓Symbol ↑EnvironmentExp
decl_method ↓EnvironmentType ↓Symbol ↑EnvironmentExp
list_decl_param ↓EnvironmentType ↑Signature
decl_param ↓EnvironmentType ↑Type

8.3 Passe 3

Programmes

program ↓EnvironmentType
list_decl_class ↓EnvironmentType
main ↓EnvironmentType

Déclarations de classes

decl_class ↓EnvironmentType

Déclarations de champs

 $\begin{tabular}{ll} \textbf{list_decl_field} $$\downarrow$ EnvironmentType $$\downarrow$ EnvironmentExp $$\downarrow$ Symbol \\ \textbf{decl_field} $$\downarrow$ EnvironmentType $$\downarrow$ EnvironmentExp $$\downarrow$ Symbol \\ \textbf{initialization} $$\downarrow$ EnvironmentExp $$\downarrow$ Symbol $$\downarrow$ Type \\ \end{tabular}$

Déclarations de méthodes

 $\label{list_decl_method} $$ $\operatorname{LenvironmentType} $$ $\operatorname{LenvironmentExp} $$ $\operatorname{Symbol} $$ $\operatorname{decl_method} $$ $$ $\operatorname{LenvironmentExp} $$ $\operatorname{Symbol} $$ $\operatorname{list_decl_param} $$ $\operatorname{LenvironmentExp} $$ $\operatorname{decl_param} $$ $\operatorname{LenvironmentExp} $$ $\operatorname{decl_param} $$ $\operatorname{LenvironmentExp} $$ $\operatorname{LenvironmentExp} $$ $\operatorname{method_body} $$ $\operatorname{LenvironmentExp} $$ $$

Déclarations de variables

 $\label{list_decl_var} $$ $$ LenvironmentType $$ EnvironmentExp $$ EnvironmentExp $$ $$ EnvironmentExp $$ EnvironmentExp $$ EnvironmentExp $$ EnvironmentExp $$ EnvironmentExp $$ EnvironmentExp $$ $$ EnvironmentExp $$ $$ EnvironmentExp $$ $$ $$ EnvironmentExp $$ Environmen$

Blocs et instructions

 $\begin{tabular}{l} bloc \downarrow EnvironmentType \downarrow EnvironmentExp \downarrow EnvironmentExp \downarrow Symbol \downarrow Type \\ list_inst \downarrow EnvironmentType \downarrow EnvironmentExp \downarrow Symbol \downarrow Type \\ inst \downarrow EnvironmentType \downarrow EnvironmentExp \downarrow Symbol \downarrow Type \\ \end{tabular}$

Expressions

```
rvalue ↓EnvironmentType ↓EnvironmentExp ↓Symbol ↓Type
condition ↓EnvironmentType ↓EnvironmentExp ↓Symbol
list_exp_print ↓EnvironmentType ↓EnvironmentExp ↓Symbol
exp_print ↓EnvironmentType ↓EnvironmentExp ↓Symbol
expr ↓EnvironmentType ↓EnvironmentExp ↓Symbol ↑Type
literal ↑Type
op_bin ↑Operator
op_un ↑Operator
```

Expressions affectables

```
lvalue ↓EnvironmentType ↓EnvironmentExp ↓Symbol ↑Type
lvalue_ident ↓EnvironmentExp ↑Type
field_ident ↓EnvironmentExp ↑Visibility ↑Symbol ↑Type
```

Appels de méthode

```
method_call ↓EnvironmentType ↓EnvironmentExp ↓Symbol ↑Type
method_ident ↓EnvironmentExp ↑Signature ↑Type
rvalue_star ↓EnvironmentType ↓EnvironmentExp ↓Symbol ↓Signature
```

9 Note sur les champs protégés

La règle (3.66), qui porte sur les champs protégés en Deca, est relativement délicate à comprendre. Elle dit la chose suivante :

- 1. le type de l'expression doit être un sous-type de la classe courante;
- 2. le type de la classe courante doit être un sous-type de la classe où le champ protégé est déclaré.

Ces deux règles sont inspirées du point 6.6.2.1 des spécifications Java. La principale différence est que Deca n'a pas de notion de package, et s'inspire des règles de visibilité de deux classes Java situées dans des packages différents.

La condition (2) est la plus simple : il faut "qu'on soit dans une sous-classe". La condition (1) permet en fait de ne pas pouvoir détourner la condition (2).

Prenons l'exemple suivant :

```
class A {
   protected int x;
}

class X {
   void m()
   {
      A a = new A();
      println(a.x) ; // Erreur contextuelle : x est protege
   }
}
```

On utilise une classe A dans la classe X et on cherche à accéder au champ x de a. Comme il est protégé, on n'a pas le droit (condition (2)). On pourrait essayer de contourner cette interdiction de la façon suivante.

On déclare une classe B, sous classe de A,

```
class B extends A {
    int getX(A a) {
        return a.x;
    }
}

et on modifie la classe X:

class X {
    void m() {
        A a = new A();
        B b = new B();
        println(b.getX(a)); // Ok du point de vue de la condition (2)
    }
}
```

Si on pouvait faire cela, n'importe quelle classe pourrait accéder au champ de A... En fait, on n'a pas le droit de faire cela, grâce à la condition (1), dans la classe B, on a :

10 Implémentation de l'environnement

Dans cette partie, on montre sur un exemple comment les environnements env_types et env_exp peuvent être implémentés.

Un environnement est une liste chaînée de tables d'associations identificateur \mapsto définition.

La figure 1 montre les environnements prédéfinis env_types_predef et env_exp_object.

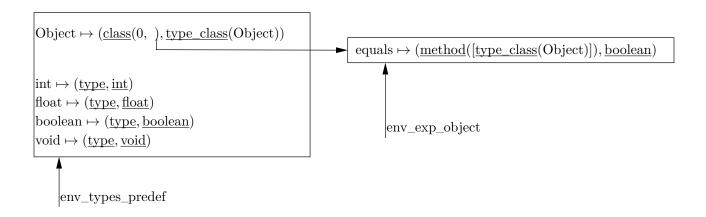


FIGURE 1 – Environnements prédéfinis env_types_predef et env_exp_object.

Considérons le programme Deca suivant.

```
class A {
  protected int x ;
  void setX(int x) {
     this.x = x;
   int getX() {
     return x ;
   }
   void init() {
     x = 0;
}
class B extends A {
   int y ;
   void init() {
     setX(0);
     y = 0;
}
class C extends A {
   int z ;
  void init() {
     setX(0);
     z = 1;
   }
}
```

La figure 2 montre l'environnement env_types construit à partir des classes du programme, ainsi que l'environnement env_exp d'analyse de chaque classe : env_exp_A , env_exp_B et env_exp_C . Cet environnement env_exp est l'environnement d'analyse du corps de la classe (attribut hérité de **corps_class**).

La figure 3 montre l'environnement env_exp_{setX} d'analyse de la méthode setX de la classe A. Cette méthode a un paramètre x de type <u>int</u>. Cet environnement correspond à l'attribut hérité env_exp de **bloc**.

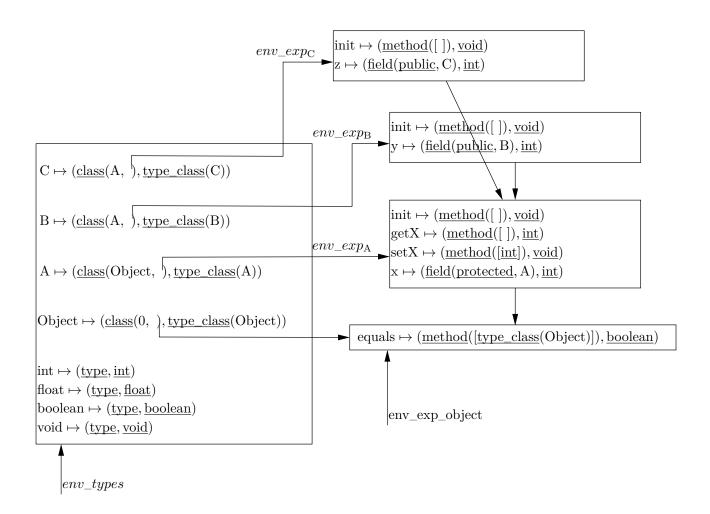


Figure 2 – Environnements env_exp d'analyse du corps des classes

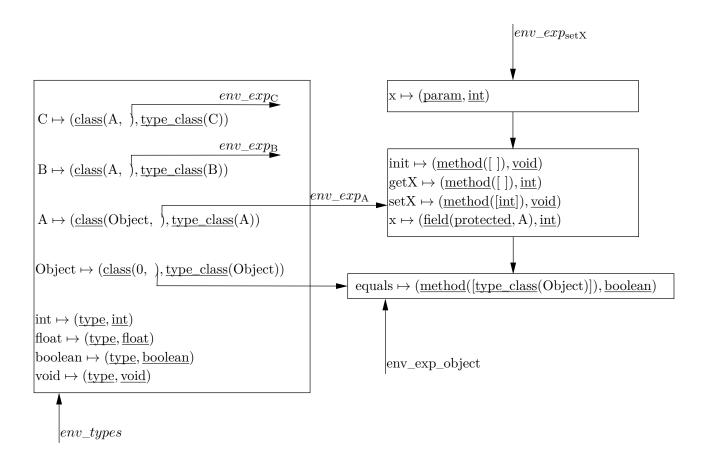


FIGURE 3 – Environnement env_exp_{setX} d'analyse du corps de la méthode setX de A

[BibliothequeStandard]

Bibliothèque standard du langage Deca

Remarque IMPORTANTE La bibliothèque standard varie selon l'extension choisie. Seules les équipes ayant choisi une extension proposant une bibliothèque (ou ayant besoin d'en ajouter une) sont concernées par ce chapitre. Notamment, les extensions TAB (fonctionnalités sur tableaux et matrices) et TRIGO (fonctions trigonométriques) enrichiront la bibliothèque standard.

La bibliothèque standard de Deca est implémentée sous la forme de fichiers source Deca, en utilisant l'inclusion de fichier (#include), qui est ici utilisée comme un moyen de partager du code entre plusieurs programmes Deca. C'est une façon pour l'utilisateur de pallier le fait que la machine abstraite servant à l'exécution de programme Deca ne permette pas l'édition de liens de programmes compilés. Contrairement aux langages comme le C, l'usage en Deca est d'écrire du code (classes et méthodes) dans les fichiers inclus (en C, c'est une mauvaise pratique d'écrire autre chose que des déclarations et des macros dans des fichiers .h, le corps des fonctions est lié au programme utilisateur par l'éditeur de liens).

Un inconvénient évident de la manière de faire de Deca est que tout le code de bibliothèque est recompilé pour chaque programme, alors qu'une vraie chaîne de compilation devrait permettre une compilation séparée. Cette solution est néanmoins suffisante dans le cadre de notre projet.

1 Recherche de fichiers dans la bibliothèque standard

Lorsqu'un fichier est inclus via la directive #include (voir la règle INCLUDE du document [Lexicographie]), le fichier à inclure est dans un premier temps cherché dans le répertoire contenant le fichier source. Si la recherche dans ce répertoire échoue, le fichier est recherché dans la bibliothèque standard fournie avec le compilateur. L'emplacement de la bibliothèque standard est laissé libre à l'implémentation, mais l'utilisateur ne doit pas avoir à le spécifier à la compilation.

2 Extension TRIGO: le fichier Math.decah

Pour les équipes ayant choisi l'extension TRIGO, la bibliothèque standard devra contenir un fichier Math.decah qui définit une classe Math avec les méthodes suivantes :

— float ulp(float f)
— float sin(float f)
— float cos(float f)
— float asin(float f)
— float atan(float f)

La méthode ulp (« Unit in the Last Place ») est identique à la méthode Math.ulp(float) en Java. Les quatre dernières méthodes calculent les fonctions mathématiques usuelles du même nom, en radians. Leur implémentation doit chercher la meilleure approximation possible permise par la représentation des flottants simple précision, à l'intérieur du codomaine de la fonction mathématique. Typiquement, la méthode asin doit retourner un flottant dans $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ (où π est le réel mathématique et pas une approximation en flottant de ce nombre). Par ailleurs, un appel à la méthode asin avec

un argument qui n'est pas dans l'intervalle [-1.0, 1.0] provoque l'arrêt de l'exécution du programme (similaire à un débordement arithmétique).

Par exemple, le programme Deca suivant est correct, il doit afficher cos(0.0) = 1.00000e+00:

```
#include "Math.decah"
{
    Math m = new Math();
    println("cos(0.0) = ", m.cos(0.0));
}
```

Convention obligatoire Le fichier Math.decah peut éventuellement fournir d'autres classes que la classe Math, et cette classe peut fournir des méthodes et des champs qui ne sont pas dans la liste ci-dessus, pourvu que ces nouveaux identificateurs commencent par le caractère _. Ceci permet à l'utilisateur de pouvoir écrire du code utilisant Math.decah sans risque de conflit de noms pourvu que ses propres identificateurs ne commencent pas par _.

3 Autres extensions utilisant la bibliothèque standard

Pour les autres extensions utilisant la bibliothèque standard, les classes et leur contenu ne sont pas imposés puisque cela fait partie intégrante du travail de définition de l'extension. Il est attendu qu'une spécification précise soit fournie, notamment dans le manuel utilisateur.

Vous pourrez utiliser les conventions de nommage suggérées ci-dessus pour l'extension TRIGO.

Sémantique de Deca

La sémantique de Deca n'est pas entièrement détaillée : on se référera à la sémantique de Java pour les constructions non évoquées dans les paragraphes qui suivent. Pour le calcul flottant, on s'appuiera sur la norme IEEE-754 selon le mode de représentation utilisé par la machine abstraite, c'est à dire pour des flottants en simple précision sur 32 bits.

Un programme sémantiquement correct peut s'arrêter anormalement à l'exécution en cas de dépassement des limites de la machine support de l'exécution ou d'erreur de l'utilisateur lors de la lecture d'une valeur.

Un programme est dit sémantiquement incorrect si une erreur survient lors de son exécution. Un compilateur est tenu, en l'absence d'options spécifiques, de produire du code qui provoquera l'erreur à l'exécution.

Remarque: un compilateur peut émettre un message d'avertissement s'il peut statiquement détecter qu'une erreur va arriver à l'exécution. Il est néanmoins tenu de produire du code (qui doit commencer à s'exécuter normalement, puis provoquer une erreur à l'endroit indiqué).

1 Initialisation des variables et champs

Les initialisations de variable ont lieu dans l'ordre de déclaration. Une variable non initialisée n'a pas de valeur définie : accéder à une variable avant qu'elle ne soit affectée d'une valeur est un comportement indéfini.

Remarque : La sémantique de Java est moins permissive, car les programmes qui "pourraient potentiellement" accéder à des variables n'ayant pas de valeur définie sont rejetés à la compilation.

Lors de l'allocation d'un objet, les champs sont initialisés dans l'ordre de déclaration. Un champ non initialisé, ou accédé avant d'être initialisé, a la valeur par défaut 0 pour entier, 0.0 pour un flottant, false pour un booléen, ou null pour un objet.

2 Instruction « new »

L'instruction new <classe>(); alloue dynamiquement un objet de la classe <classe> et l'initialise (cf. 1). L'objet n'est jamais désalloué.

3 Instruction « return »

L'exécution d'une instruction return achève l'exécution du corps de la méthode possédant ce return. Étant donné une méthode qui retourne une valeur d'un type différent de void, l'exécution du corps de cette méthode qui se termine sans passer par une instruction return est incorrecte.

4 Ordre d'évaluation

Les opérandes des opérations arithmétiques binaires, de comparaison et d'affectation sont évalués de gauche à droite.

Les expressions booléennes sont évaluées paresseusement de gauche à droite. Cela signifie que lorsqu'on évalue C1 && C2, on évalue d'abord C1, puis si C1 est vrai, on évalue C2. C2 n'est pas évalué si C1 est faux. De même, lorsqu'on évalue C1 || C2, on évalue d'abord C1, puis si C1 est faux, on évalue C2. C2 n'est pas évalué si C1 est vrai.

5 Débordements lors de l'évaluation des expressions

Une division entière ou un reste par 0 provoque une erreur.

Il n'y a pas de débordement pour les opérations arithmétiques sur les entiers : les calculs sont fait modulo 2^{32} .

Un débordement sur une opération arithmétique sur des flottants provoque une erreur.

Une division flottante par 0.0 provoque une erreur.

Lorsqu'une erreur à l'exécution survient, le programme devra afficher un message d'erreur explicite, et quitter la machine abstraite avec l'instruction ERROR.

Remarque : La sémantique de Java est plus permissive sur les opérations en flottants, les débordements ne provoquent pas d'erreur à l'exécution mais retournent des valeurs flottantes spéciales autorisées par la norme IEEE-754.

6 Procédures d'affichage

Un appel à print sans argument n'a aucun effet. Un appel à println sans argument a pour effet de passer à la ligne.

```
print(E); écrit sur la sortie standard la valeur de E.
```

```
print(E1, E2, ... En); est équivalent à print(E1); print(E2); ... print(En);.
println(E1, E2, ... En); est équivalent à print(E1, E2, ... En); println();.
```

printx et printlnx sont respectivement équivalents à print et println à une exception près : les flottants sont affichés en hexadecimal avec printx et printlnx, et en décimal avec print et println.

7 Appels de méthodes

Les paramètres des méthodes sont passés par valeur, ce qui signifie que la valeur du paramètre effectif est recopiée dans le paramètre formel au début de l'appel.

Les paramètres d'une méthode sont évalués de gauche à droite. Lors d'un appel de méthode de la forme X.m(Y1, Y2, ... Yn), on évalue successivement X, puis Y1, puis Y2, ... puis Yn.

8 Méthode « equals » de la classe Object

Sa sémantique est équivalente à celle de :

```
public boolean equals (Object other) {
   return this == other;
}
```

9 Opérateur de conversion de type : (type) (valeur)

L'opérateur de conversion de type (transtypage, ou « cast » en anglais) s'écrit (type) (valeur). Contrairement à Java, les parenthèses sont obligatoires autour de valeur. Il a essentiellement deux utilisations :

- (int)(valeur_flottante) convertit la valeur flottante en arrondissant à l'entier vers 0 le plus proche (il n'y a donc aucun débordement possible lors de cette opération).
- (UneClasse) (v) vérifie que la valeur v est effectivement de type UneClasse. Si le type statique de v dérive de UneClasse, la conversion réussit trivialement. Sinon, il faut vérifier que le type dynamique de v dérive bien de UneClasse. Quand la conversion échoue, une erreur est levée. Quand elle réussit, la valeur v est renvoyée.

10 Méthodes écrites en assembleur

Pour une méthode écrite en assembleur (via la syntaxe asm("<portion d'assembleur>")), le code généré est constitué simplement de la portion d'assembleur précédée de l'étiquette correspondant au début de méthode. Le compilateur n'ajoute aucune instruction assembleur : l'éventuelle sauvegarde de registre ou le retour de fonction via l'instruction RTS sont à la charge du programmeur. Aucune vérification n'est faite par le compilateur sur la portion d'assembleur : si cette portion ne respecte pas les conventions de liaison ou la syntaxe de l'assembleur, alors le résultat de la compilation peut être un code assembleur incorrect.

11 Liste des catégories d'erreurs à l'exécution

En l'absence d'options spécifiques, les cas 11.1, 11.2 et 11.3 provoquent un arrêt de l'exécution du programme. Le cas 11.4 provoque un comportement indéfini.

11.1 Programmes incorrects

- division entière (et reste de la division entière) par 0;
- débordement arithmétique sur les flottants (inclut la division flottante par 0.0);
- absence de return lors de l'exécution d'une méthode;
- conversion de type impossible;
- déréférencement de null.

11.2 Programmes corrects, dont l'exécution dépend de l'utilisateur

— erreur de lecture (valeur entrée pas dans le type attendu).

11.3 Programmes corrects, dont l'exécution dépasse les limites de la machine

— débordement mémoire (pile ou tas).

11.4 Programmes erronés

- accès à des variables non initialisées;
- utilisation d'une méthode écrite en assembleur non compatible avec l'assembleur généré par le compilateur.

[Decac]

Description du compilateur decac

1 Ligne de commande

Le programme principal « decac » est un compilateur Deca complet. On permettra de désigner le fichier d'entrée par des chemins de la forme <répertoires/nom.deca> (le suffixe .deca est obligatoire); SAUF ERREUR, LE RÉSULTAT DOIT ÊTRE DANS UN FICHIER <répertoires/nom.ass> situé dans le même répertoire que le fichier source.

La syntaxe d'utilisation de l'exécutable decac est :

```
decac [[-p | -v] [-n] [-r X] [-d]* [-P] [-w] <fichier deca>...] | [-b]
```

La commande decac, sans argument, affichera les options disponibles. On peut appeler la commande decac avec un ou plusieurs fichiers sources Deca.

On définira les options suivantes à la commande decac :

•	-b	(banner)	:	affiche une bannière indiquant le nom de l'équipe
	-p	(parse)	:	arrête decac après l'étape de construction de
				l'arbre, et affiche la décompilation de ce dernier
				(i.e. s'il n'y a qu'un fichier source à
				compiler, la sortie doit être un programme
				deca syntaxiquement correct)
•	- ∆	(verification)	:	arrête decac après l'étape de vérifications
				(ne produit aucune sortie en l'absence d'erreur)
	-n	(no check)	:	supprime les tests à l'exécution spécifiés dans
				les points 11.1 et 11.3 de la sémantique de Deca.
•	-r X	(registers)	:	limite les registres banalisés disponibles à
				$R0 \ldots R\{X-1\}, \text{ avec } 4 \le X \le 16$
•	-d	(debug)	:	active les traces de debug. Répéter
				l'option plusieurs fois pour avoir plus de
				traces.
•	-P	(parallel)	:	s'il y a plusieurs fichiers sources,
				lance la compilation des fichiers en
				parallèle (pour accélérer la compilation)

N.B. Les options '-p' et '-v' sont incompatibles.

On pourra ajouter une option '-w' autorisant l'affichage de messages d'avertissement (« warnings ») en cours de compilation. Il sera également possible d'ajouter une option pour fixer le mode d'arrondi si cela s'avère utile.

En l'absence des options '-b', '-p' et de l'éventuelle option '-w', une exécution de decac ne doit produire AUCUN AFFICHAGE si la compilation réussit. L'exécution de decac ne doit en aucun cas lire d'entrée sur son entrée standard. Il est impératif de respecter les conventions sur les arguments de

Decac 104

la commande, car les exécutables decac seront testés automatiquement à la fin du projet.

L'option '-b' ne peut être utilisée que sans autre option, et sans fichier source. Dans ce cas, decac termine après avoir affiché la bannière.

Si un fichier apparaît plusieurs fois sur la ligne de commande, il n'est compilé qu'une seule fois.

2 Formatage des messages d'erreur

test.deca:3:1: Caractère '#' non autorisé

Les messages d'erreur (lexicales, syntaxiques, contextuelles, et éventuelles limitations du compilateur) doivent être formatées de la manière suivante (cette règle est également indispensable pour l'évaluation automatique de votre compilateur par les enseignants) :

```
<nom de fichier.deca>:<ligne>:<colonne>: <description informelle du problème>
    Comme par exemple (erreur au 4ème caractère de la ligne 12) :
fichier.deca:12:4: Identificateur "foobar" non déclaré (règle 6.12)
ou bien (erreur au début de la ligne 3) :
```

Il ne faut pas d'espaces entre le nom de fichier et les ':', ni autour des numéros de lignes et de colonnes.

$[{f Machine Abstraite}]$

Définition de la Machine Abstraite et de son langage d'assemblage

Ce document ne concerne que la spécification de la machine. Une implémentation logicielle de cette machine est fournie par le programme ima donné dans le document [Ima].

1 Données et mémoires

Les types des valeurs manipulées sont les entiers, les flottants, les adresses (séparées en « adresses code » et « adresses mémoire »). La machine abstraite gère les nombres sur 32 bits, et utilise la représentation de la norme IEEE-754 pour les flottants.

La « mémoire physique » (sous ce terme sont englobés registres, caches, RAM...) de la machine est logiquement partagée en 3 zones :

- La zone registres. Elle est constituée des registres banalisés R0 .. R15. Ils peuvent contenir des valeurs de tout type, et peuvent être lus ou modifiés.
- La zone code. Elle contient les instructions du programme. À cette zone est associé un registre spécialisé, PC (compteur ordinal), qui contient les adresses successives des instructions à exécuter (appelées « adresses code »). PC ne peut être ni lu ni modifié explicitement. PC+1 (resp. PC-1) est l'adresse de l'instruction suivant (resp. précédant) celle d'adresse PC.
- La zone mémoire, partagée en une zone pile et une zone tas. La zone mémoire est constituée de mots. À chaque mot est associée une « adresse mémoire ». Seuls certains mots sont consultables et modifiables; on les appelle « mots adressables ». En zone pile, il s'agit des N mots constituant la pile (N n'est pas fixé a priori). En zone tas, il s'agit des mots constituant les blocs alloués par l'instruction NEW. Le nombre de mots allouables en zone tas n'est pas fixé a priori. Chaque mot adressable peut contenir une valeur de tout type et peut être lu ou modifié. À la zone pile sont associés trois registres spécialisés, qui ne peuvent contenir que des adresses de la zone pile (mais pas nécessairement de la pile):

```
GB (base globale) : contient à tout instant l'adresse précédant celle du premier mot de la pile.
```

LB (base locale).

SP (pointeur de pile).

Les adresses des mots de la pile sont comprises entre GB+1 et GB+N.

Les éléments de mémorisation (registres banalisés et mots mémoires) sont « typés » dynamiquement. Initialement, tout est « indéfini » ; lors d'une modification d'un élément, le type de données est aussi mémorisé. Lors d'une opération, il y a vérification de compatibilité de type. Une valeur particulière, appelée null, est de type « adresse mémoire ». Elle représente une « absence d'adresse ».

Avant l'exécution de la première instruction :

— Le contenu de la pile ainsi que des registres RO .. Rn est indéfini.

[MachineAbstraite] 106

- GB = LB = SP sont initialisés à la même valeur (par le chargeur).
- Aucun mot n'est alloué en zone tas.
- PC est initialisé (par le chargeur) à l'adresse de la première instruction à exécuter dans la zone code.

2 Modes d'adressages

On dispose de 6 modes d'adressage suivants :

```
registre direct Rm (m dans 0 .. n)
```

- registre indirect avec déplacement d(XX), où d est entier et $XX \in \{GB, LB, SP, Rm\}$ (qui doit contenir une adresse mémoire autre que null). Le mode d'adressage d(XX) désigne l'adresse mémoire (contenu de XX)+d.
- registre indirect avec déplacement et index d(XX, Rm), où d est entier, XX ∈ {GB, LB, SP, Rp} (qui doit contenir une adresse mémoire autre que null), et Rm doit contenir un entier. Le mode d'adressage d(XX, Rm) désigne l'adresse mémoire (contenu de XX)+(contenu de Rm)+d. Pour les deux modes d'adressage indirect, le déplacement d est un entier en notation Deca, éventuellement précédé d'un signe + ou -.
- immédiat #d, où soit d est un littéral entier ou flottant en notation Deca éventuellement précédé d'un signe + ou (auquel cas la valeur désignée est l'entier ou le flottant correspondant), soit d est la séquence de 4 caractères null (donc le mode d'adressage est #null), auquel cas la valeur désignée est l'adresse mémoire null.
- étiquette etiq, ou etiq est une étiquette de programme en langage d'assemblage. etiq désigne l'adresse code de la première instruction qui suit l'étiquette.
- **chaîne** "...", une chaîne de caractères en notation Ada : chaîne délimitée par des guillemets doubles. Un littéral guillemet double s'écrit avec deux guillemets doubles.

Par exemple, WSTR "je dis ""bonjour"" affichera je dis "bonjour".

3 Instructions

Les instructions (dénotées InstructionMA dans la grammaire de la section 4) sont classées par catégories dans les sections suivantes :

- 3.1 Transfert de données
- 3.2 Allocation mémoire
- **3.3** Comparaison de valeurs
- **3.4** Opérations arithmétiques
- 3.5 Contrôle
- 3.6 Entrées-Sorties
- 3.7 Divers

Les instructions ont 0, 1 ou 2 opérandes (« source » puis « destination »). Avant l'exécution de chaque instruction, PC est incrémenté de 1.

La description des instructions suit les conventions suivantes :

Les notations d(XX) et d(XX, Rm) ne sont autorisées pour des dval

[MachineAbstraite] 107

```
que si l'adresse désignée est une adresse d'un mot adressable.
```

V[dadr] n'a de sens que si A[dadr] est l'adresse d'un mot adressable.

```
L \leftarrow Val est une affectation : la valeur Val est rangée dans L. L est soit un registre, soit l'adresse d'un mot adressable. Dans ce dernier cas, Val est rangée dans le mot d'adresse L.
```

Les codes condition sont :

```
EQ (égal)

GT (strictement supérieur)

GE (supérieur ou égal)

OV (débordement)

NE (différent)

LT (strictement inférieur)

LE (inférieur ou égal)
```

Ils sont positionnés à vrai ou faux par certaines instructions. On les note génériquement "cc" dans les instructions Scc et Bcc (ex. : SEQ, BOV, etc.).

Les codes de comparaison EQ, NE, GT, LT, GE, LE sont toujours positionnés simultanément, et leurs valeurs satisfont toujours les axiomes :

La valeur initiale des codes condition est indéterminée, mais elle satisfait les axiomes.

Pour indiquer qu'une instruction positionne les codes condition, on écrit "CC : suivi de OV et/ou CP (CP pour ComParaison).

- La valeur des codes de comparaison est relative au résultat de la comparaison pour l'instruction CMP (voir ci-dessous). Pour un transfert d'une valeur V (entière, flottante ou adresse mémoire), une instruction arithmétique dont le résultat est V, ou une lecture d'une valeur V (entière ou flottante), la valeur des codes de comparaison est la même qu'après une instruction CMP #(0 ou 0.0 ou null), V.
- Lorsqu'une instruction arithmétique sur nombre entier positionne $\tt OV$ à vrai, l'opération est tout de même effectuée. Le résultat est celui de l'opération en arithmétique modulo $\tt 2^{32}$.
- Lorsqu'une instruction autre qu'arithmétique sur des entiers positionne OV à vrai, son effet est indéterminé. Pour les instructions ADD, SUB, DIV et MUL sur nombres flottants, OV vaut vrai si et seulement si le résultat n'est pas codable sur un flottant (après approximation éventuelle). Pour DIV, OV vaut également vrai dans le cas d'une division par 0.0 (quel que soit le dividende, y compris 0.0). Les autres cas de positionnement de OV sont expliqués ci-dessous.

[MachineAbstraite] 108

3.1 Transfert de données

LEA dadr, Rm : Rm <- A[dadr]

PEA dadr : V[SP]+1 <- A[dadr] ; SP <- V[SP] + 1

Note : CP non positionné si transfert d'adresse code

3.2 Allocation mémoire

NEW dval, Rm : alloue un bloc de V[dval] mots contigus dans le tas et range dans Rm l'adresse du début du bloc (les adresses des mots du bloc vont donc de O(Rm) à d-1(Rm)). d doit être un entier naturel, avec ou sans le signe +.

CC : OV = (allocation impossible)

DEL Rm : libère le bloc d'adresse V[Rm] précédemment alloué par NEW. Rm <- indéfini CC : OV = (adresse invalide)

3.3 Comparaison de valeurs (entre 2 entiers, 2 flottants ou 2 adresses mémoire)

3.4 Opérations arithmétiques (soit entre entiers, soit entre flottants)

Opérations arithmétiques spécifiques aux entiers

```
QUO dval, Rm
                : Rm <- V[Rm] / V[dval]
                                                  CC : OV = (V[dval] = 0), CP
                  (quotient entier)
                : Rm <- V[Rm] rem V[dval]
                                                  CC : OV = (V[dval] = 0), CP
REM dval, Rm
                  (reste entier)
Scc Rm
                : si (cc = vrai) alors Rm <- 1 sinon Rm <- 0
SHL Rm
                : shift left
                                                   CC : OV, CP
SHR Rm
                                                   CC : CP
                : shift right
```

Les "shifts" gauche et droit sont des opérations de décalage arithmétique : elles peuvent s'appliquer à des entiers négatifs et elles préservent le signe de l'opérande. Faire un décalage sur une valeur réelle n'a pas de sens et provoque une erreur.

Opérations arithmétiques spécifiques aux flottants

```
FMA dval, Rm : Rm \leftarrow V[Rm] * V[dval] + V[R1] CC : OV, CP
```

FMA est l'opération de Fusion Multiplication Addition. Elle permet de n'effectuer qu'un seul arrondi au lieu de deux. Cette opération est ternaire. Elle s'appuie sur deux opérandes. La troisième opérande est implicite et correspond au registre R1.

Opérations arithmétiques de conversion entre entiers et flottants

FLOAT dval, Rm : conversion entier->flottant CC : OV = (V[dval] non

Rm <- CodageFlottant(V[dval]) codable sur un flottant)</pre>

INT dval, Rm : conversion flottant->entier CC : OV = (V[dval] non

codable sur un entier)

Rm <- Signe(V[dval]) * PartieEntiere(ValAbsolue(V[dval]))</pre>

Gestion des nombres flottants

SETROUND_mode : Positionner le mode d'arrondi pour les opérations

flottantes faites après cette opération. Les

opérations possibles sont :

SETROUND_TONEAREST : arrondi à la valeur la plus proche

(mode d'arrondi initial par défaut)

SETROUND_UPWARD : arrondi à la valeur supérieure SETROUND_DOWNWARD : arrondi à la valeur inférieure

SETROUND_TOWARDZERO : arrondi vers zero

3.5 Contrôle

BRA dval : branchement inconditionnel

PC <- V[dval]

Bcc dval : branchement conditionnel

si (cc = vrai) alors

PC <- V[dval]</pre>

BSR dval : $SP \leftarrow V[SP] + 2$; $V[SP] - 1 \leftarrow V[PC]$; $V[SP] \leftarrow V[LB]$;

LB <- V[SP] ; PC <- V[dval]

RTS : $PC \leftarrow C[V[LB]-1]$; $SP \leftarrow V[LB]-2$; $LB \leftarrow C[V[LB]]$;

Note: V[dval] doit être une adresse code

3.6 Entrées-Sorties

erreur de syntaxe)

RFLOAT : R1 <- flottant lu CC : CP, OV (idem RINT)

WINT : écriture de l'entier V[R1]

WFLOAT : écriture du flottant V[R1] en décimal (arrondi si besoin)

WFLOATX : écriture du flottant V[R1] en hexadécimal (exact)

WSTR "..." : écriture de la chaîne (notation Ada)

WNL : écriture newline

RUTF8 : R1 <- code entier du caractère UTF-8 lu CC : CP WUTF8 : écriture du caractère UTF-8 dont le code entier est V[R1]

Note: RINT et RFLOAT attendent un retour chariot de l'utilisateur du terminal (pour le laisser éventuellement corriger avec la touche "backspace"). Au contraire, RUTF8 lit le premier caractère non-consommé (sans attendre de retour chariot) et laisse la plupart des touches de contrôle non-interprétées (e.g. sauf

Ctrl-C et Ctrl-D); attention, certaines touches du clavier (e.g. les flèches) correspondent à plusieurs caractères. RUTF8 utilise 0 comme code de fin de fichier (e.g. sur un Ctrl-D).

3.7 Divers

```
ADDSP #d
                 : SP \leftarrow V[SP] + d
                   d doit être un entier naturel, avec ou sans le signe +.
SUBSP #d
                 : SP \leftarrow V[SP] - d
                   d doit être un entier naturel, avec ou sans le signe +.
TSTO #d
                 : test débordement pile.
                                              CC : OV = (V[SP] + d > V[GB] + N)
                   d doit être un entier naturel, avec ou sans le signe '+'.
HALT
                 : arrêt du programme
                 : similaire à HALT, mais ima termine avec un status d'erreur.
ERROR
SCLK
                 : R1 <- nombre entier de secondes
                   depuis le 01/01/2001 à 0h00 *au lancement* de ima
CLK
                 : R1 <- nombre flottant de secondes
                                                                       CC:CP, OV
                   depuis le lancement de ima
```

3.8 Temps d'exécution des instructions (en nombre de cycles internes)

LOAD	2							DIV	40					
STORE	2							INT	4					
LEA	0							BRA	5					
PEA	4							Всс	5	(cc	vrai)	4	(cc	faux)
PUSH	4							BSR	9					
POP	2							RTS	8					
NEW	16							DEL	16					
ADD	2							RINT	16					
SUB	2							RFLOAT	16					
SHL	2							WINT	16					
SHR	2				W]	FLOAT	et	WFLOATX	16					
OPP	2					RUT:	F8 (et WUTF8	16					
MUL	20							WSTR	16					
CMP	2							WNL	14					
QUO	40							ADDSP	4					
REM	40							SUBSP	4					
FLOAT	4							TST0	4					
Scc	3	(cc	vrai)	2	(cc	faux)	HALT	1					
ERROR	1							SCLK	2					
SETROUND_mode 20					CLK	16								
FMA	21	L												

Il faut ajouter le cas écheant le temps d'accès aux opérandes :

```
Modes d'adressage Temps

Rm 0
d(XX) 4
d(XX,Rm) 5
#d 2
etiq 2
"..." 2 * longueur de la chaîne
```

4 Syntaxe du langage d'assemblage

La syntaxe suit les principes suivants :

- l'espace et la tabulation sont des séparateurs.
- on peut insérer des lignes blanches où on veut.
- les commentaires sont constitués du caractère ';' et du reste de la ligne (caractères imprimables et tabulations).
- une étiquette est positionnée en faisant suivre son nom de ':'.
- on place une instruction par ligne, éventuellement suivie de commentaires.

Les codes opération et les noms des registres peuvent être en majuscules ou minuscules. Il est d'usage d'indenter les instructions par rapport aux étiquettes.

La grammaire concrète de l'assembleur est la suivante (notation ANTLR) où le non-terminal instruction représente le langage des instructions données ci-dessus.

```
programme : ligne* EOF;
ligne : (ETIQUETTE ':')* instruction? '\n';
```

La lexicographie des étiquettes et des commentaires est la suivante (notation ANTLR)

```
LETTRE: 'a' .. 'z' | 'A' .. 'Z';

CHIFFRE: '0' .. '9';

ETIQUETTE: LETTRE (LETTRE | CHIFFRE | '_' | '.')*

COMM_CAR: ('\t' | ' ' .. '~') // caracteres imprimables et tabulation

COMMENTAIRE: ';' COMM_CAR*
```

Exemples d'étiquettes : Ceci_Est_1_etiquette.0, En.Voici_42._autres_.

On ne distingue pas majuscules et minuscules. Une étiquette ne doit pas avoir un nom de codeopération ou de registre.

5 Exemple de programme assembleur : la factorielle récursive

```
; Programme principal : Lecture d'un entier positif, impression de sa factorielle
; Rem : il faudrait tester le debordement de pile
  WSTR "n = "
  RINT
  PUSH R1
  BSR fact.
  SUBSP #1
  WSTR "fact = "
  LOAD RO, R1
  WINT
  WNL
  HALT
; La fonction factorielle d'un entier >= 0. Par convention, le
; resultat est dans RO. -2(LB) designe le parametre.
Fact.
                       ; recuperation du parametre effectif n
  LOAD - 2(LB), R1
  BNE sinon_fact
  LOAD #1, RO
                       ; pour n = 0, n! = 1
  BRA fin.fact
```

```
sinon_Fact:
   SUB #1, R1
                     ; on empile n-1
   PUSH R1
   BSR fact.
                       ; on calcule (n-1)!
   SUBSP #1 ; on depile le parametre MUL -2(LB), RO ; on multiplie le resultat par n
Fin.fact:
   RTS
                        ; le resultat est dans RO
Une autre écriture possible de la fonction factorielle :
; Une utilisation de Scc : prise en compte de l'observation 0! = 1! = 1
fact. :
              LOAD -2(LB), R1 ; recuperation du parametre effectif n
              SUB #1, R1
              SLE RO
                                  ; n! = 1 si (n-1 \le 0)
              BLE fin_fact.42
                                ; on empile n-1
; on calcule (n-1)!
              PUSH R1
              BSR fact.
             SUBSP #1 ; on depile le parametre

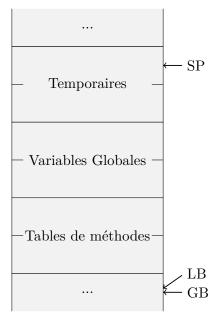
MUL -2(LB), RO ; on multiplie le resultat par n
fIN_fACT.42: RTS
```

$[{f Conventions Liaison}]$

Conventions de liaison pour la Machine Abstraite

Le code produit par un compilateur doit respecter certaines conventions imposées par le système de la machine cible, afin notamment de permettre l'écriture d'un logiciel en utilisant plusieurs langages de programmation.

1 État de la pile lors de l'exécution du programme principal



Les tables des méthodes sont écrites au bas de la pile et sont décrites en section 3 du document [Gencode].

2 Bloc d'activation d'une méthode

Les registres R0 et R1 sont des registres scratch. Cela signifie que leur valeur peut être modifiée par un appel de méthode. Les registres R2, R3,... etc. ne sont pas scratch, donc le code d'une méthode doit sauvegarder et restaurer les registres que celle-ci utilise. R0 sert à stocker le résultat d'une méthode.

Les paramètres d'une méthode sont empilés de droite à gauche. Lors de l'appel X.m(Y1, Y2, ... Yn), on empile d'abord Yn, puis ..., puis Y2, puis Y1, puis X. Le paramètre implicite X a donc toujours -2(LB) comme adresse. Y1 a comme adresse -3(LB), Y2 a comme adresse -4(LB)... etc.

La figure 1 illustre ces conventions.

3 Nommage des étiquettes

Le compilateur ne doit générer aucune étiquette dont le nom commence par la séquence de caractères

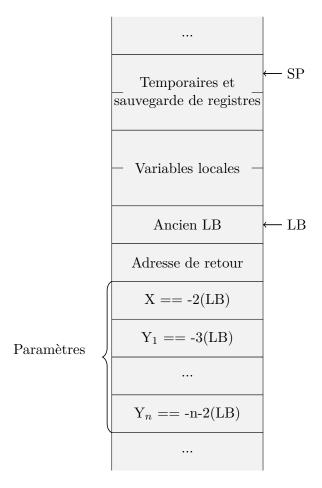


FIGURE 1 – Bloc d'activation d'une méthode

'user.'. En revanche, un programme Deca peut utiliser des étiquettes dont le nom commence par 'user.' dans des méthodes écrites en assembleur.

Troisième partie

Compléments sur les outils et les méthodes de Génie Logiciel

Environnement de développement pour le projet Génie Logiciel

1 Organisation en répertoires

Le document [SeanceMachine] vous explique comment récupérer le répertoire Projet_GL/ dans lequel vous allez travailler. Ce répertoire contient :

docs/: documentations (manual utilisateur, documentation de conception, ...) de votre projet.

planning/: diagrammes de Gantt (planning prévisionnel et réalisé).

src/ : les fichiers sources de votre projet :

main/: les fichiers sources du compilateur

java/, antlr4/: fichiers source Java et ANTLR4, utilisant la disposition habituelle de Java (un répertoire par paquetage).

bin/: script(s) permettant de lancer le programme Java en ligne de commande.

assembly/: fichiers permettant la génération d'une version empaquetée du programme (intéressant pour distribuer une version propre au client, mais pas utile pendant le développement)

resources/: fichiers nécessaires à l'exécution du programme, qui seront copiés dans target/classes pendant la compilation.

test/: les fichiers sources permettant de tester le projet.

java/: fichiers Java pour le test du projet (tests unitaires JUnit, lanceurs pour différentes méthodes main, ...)

deca/: programmes source Deca à tester sur le compilateur

script/: scripts de tests (typiquement, scripts lançant le compilateur sur les programmes contenus dans le répertoire deca)

resources/: fichiers nécessaires à l'exécution des tests, qui seront copiés dans target/test-classes

examples/: Contient quelques exemples de code pour vous aider. Ne correspond à aucun rendu.

target/: répertoire contenant les fichiers générés (fichiers .class, fichiers .jar, Javadoc, ...).

2 Travail en parallèle et gestion de versions

Dans toute cette partie, on suppose que l'utilisateur a positionné \$GITREPO et édité son fichier ~/.gitconfig (cf. [SeanceMachine]).

Chacun des membres d'une équipe travaille sur son compte personnel et possède donc une arborescence Projet_GL/. Il faut évidemment "synchroniser" les différentes arborescences d'une équipe afin qu'il n'y ait pas de divergence entre les arborescences : si un étudiant crée ou modifie un fichier, celui-ci doit être rendu disponible pour ses co-équipiers afin qu'ils puissent l'utiliser.

Pour réaliser ce mécanisme de synchronisation, on utilise le logiciel libre Git. Le principe général en est le suivant : dans chaque répertoire de travail, en plus des fichiers sur lesquels vous allez travailler,

Git conserve tout l'historique du projet et l'utilise pour faire automatiquement les fusions entre les versions des différents coéquipiers. En plus de chaque répertoire de travail (un par étudiant), un dépôt (repository en anglais) est préalablement créé pour chaque équipe, contenant l'arborescence initiale de Projet_GL/.

Ce document présente l'utilisation de Git depuis la ligne de commande. Une autre solution est d'utiliser un IDE pour réaliser les mêmes opérations. Netbeans et Eclipse permettent tous les deux d'utiliser Git sans installation supplémentaire dans leurs dernières versions. Si on utilise Git en ligne de commande, il faut prendre soin de rafraîchir le projet depuis l'IDE après toutes les commandes qui modifient des fichiers (comme git pull).

Chaque membre de l'équipe crée sa propre arborescence Projet_GL/ dans son compte, par la commande

```
git clone "$GITREPO" Projet_GL
```

Un étudiant souhaitant transmettre des modifications aux autres membres de son équipe doit mettre à jour le dépôt, par la commande :

```
git commit -a # enregistrer les changements localement
git push # envoyer les changements dans le dépôt partagé
```

Un étudiant souhaitant récupérer les modifications intervenues dans le dépôt le fait au moyen de la commande :

```
git pull
```

Un fichier nouvellement créé à ajouter au dépôt doit être déclaré par la commande :

```
git add nom-du-fichier
```

On peut ajouter tous les fichiers d'un répertoire avec :

```
git add nom-du-repertoire
```

(le fichier sera effectivement enregistré dans le dépôt au prochain 'commit')

A chaque équipe est associé un compte particulier ("compte Git") sur depots.ensimag.fr. Les comptes Git s'appellent depotgl01 à depotgl60 (s'il y a 60 équipes). Ce sont sur ces comptes que se trouvent les dépôts Git partagés. Chaque dépôt contient essentiellement les fichiers sources, les fichiers de test et la documentation.

Outre la possibilité de travail à plusieurs, Git offre un mécanisme d'archivage des versions. La commande 'git commit -a' enregistre dans le dépôt la nouvelle version (on parle indifféremment de 'version', de 'révision', ou de 'commit') des fichiers modifiés, mais les révisions précédentes y sont encore présentes. Les révisions successives d'un fichier sont identifiées par une nombre hexadécimal (40 chiffres hexa, mais on peut en général abréger l'identifiant en ne donnant que les premiers caractères). Il est possible de récupérer une révision quelconque d'un fichier. Il est également possible de consulter les différences entre deux révisions d'un fichier, ainsi que l'historique des révisions d'un fichier.

Si on veut travailler sur une version temporaire sans impacter la version "officielle" du compilateur pendant un certain temps, on peut créer des branches. La notion de branche est essentielle sur des projets de grande taille, mais vous pouvez éventuellement l'ignorer dans le cadre du projet GL. La manière la plus simple de créer des branches est d'utiliser la commande 'git commit -a' sans faire de 'git push' après. Ceci permet de garder une série de commits dans son répertoire local. Si on veut récupérer les modifications du dépôt partagé, on peut faire des 'git pull'. Une fois satisfait par la série de commits locaux, on peut la mettre à disposition des autres, avec 'git push' (précédé d'un 'git pull' si besoin). On peut aussi créer plusieurs branches dans le même dépôt, mais cette notion n'est pas présentée ici (une documentation sur l'utilisation des branches pour le projet GL est disponible sur la page BugBusters « Gérer des branches avec Git » Nous supposerons donc que vous n'utilisez qu'une seule branche, qui par défaut s'appelle 'master'.

En fin de projet, la dernière révision de la branche 'master' du dépôt sera recupérée par les enseignants pour tester le compilateur. Si vous avez travaillé sur une branche elle sera ignorée par les enseignants. Si vous n'avez pas fait de branche, la version prise en compte est le dernier commit envoyé via 'git push' (avant l'heure limite de rendu).

Une question délicate est de savoir quand lancer 'git commit -a', 'git push' et 'git pull'. Il faut à la fois éviter que les versions de travail des membres de l'équipe ne divergent trop ('commit/push'-'pull' pas assez fréquents) et éviter de polluer le travail des autres avec de nombreuses erreurs ('commit' et surtout 'push' trop rapide). Une politique acceptable est de convenir que 'git pull' doit être fait le plus souvent possible, mais que 'commit' et 'push' ne doivent être fait que lorsque le programme se compile sans erreur et qu'il n'y a pas de régression (i.e. lorsqu'il passe au moins tous les tests que la version précédente passait).

Un utilisateur avancé prendra soin de faire des 'commits' petits, regroupant une et une seule fonctionnalité ajoutée. La commande 'git diff HEAD' lancée juste avant le commit doit montrer un diff lisible, et qui 'montre' clairement les modifications faites par le programmeur. Le message de commit qui l'accompagne résume le contenu, justifie pourquoi ce changement est nécessaire et pourquoi il a été implémenté de cette manière. Les pages BugBusters « Écrire de bons messages de commit avec Git » et « Maintenir un historique propre avec Git » vous donne d'avantage de conseils pour travailler proprement avec Git.

L'utilisation de Git ne remplace en aucun cas la communication d'informations à l'intérieur de l'équipe. Il est clair qu'il est toujours important de savoir qui est en charge de quoi. Il est également toujours déconseillé que deux étudiants travaillent simultanément sur la même méthode, ou quand c'est possible, d'éviter de travailler sur le même fichier (cf. la séance machine).

3 Utilisation de Maven, fichier pom.xml

3.1 XML en 5 minutes

Le fichier de configuration de Maven est au format XML. Vous n'avez pas besoin de tout savoir sur XML, mais voici quelques éléments pour pouvoir lire un fichier XML.

XML est un langage de balise, qu'on peut voir comme une représentation textuelle d'un arbre. Un fichier XML commence par spécifier son type et son encodage :

```
<?xml version="1.0" encoding="UTF-8"?>
cproject xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

La suite du fichier est une succession de balises (très similaire à HTML). Une balise ouvrante s'écrit <nomBalise> et elle est suivie par une balise fermante </nomBalise>.

Par exemple, la portion de XML suivante :

se lit : définir une section project dans laquelle se trouve une section dependencies, dans laquelle se trouvent deux sections dependency. La première contient une section groupId contenant le texte « org.antlr », et la seconde une section du même nom mais contenant le texte « log4j ».

Les commentaires s'écrivent <!-- comme ceci -->, et si on a besoin d'utiliser les caractères < et > ailleurs que pour des balises et commentaires, on peut les échapper avec < et > (« lower than » et « greater than »).

3.2 Introduction à Maven

Pour compiler le projet, lancer les tests et générer des documentations, nous utiliserons l'outil Maven (commande mvn). Maven se base sur un fichier de configuration appelé pom.xml (Project Object Model) situé à la racine du projet. En se basant sur ce fichier, l'outil va permettre entre autres :

- De **gérer les dépendances externes de notre projet**. Il serait fastidieux de devoir télécharger et placer dans le \$CLASSPATH chacune des bibliothèques Java que nous utilisons. Maven permet de faire tout cela automatiquement : on déclare les dépendances dans le pom.xml, et il les télécharge à la première utilisation (dans le répertoire \$HOME/.m2/ par défaut) et positionne le \$CLASSPATH correctement.
- De compiler les fichiers source de notre projet. On pourrait comparer cet aspect à l'outil make, mais avec une philosophie différente : Maven sépare la description du projet et les actions à effectuer pour compiler. La description du projet est faite par l'utilisateur dans le fichier pom.xml, et les actions à effectuer pour compiler sont en général déjà écrite dans des plugins (essentiellement écrits en Java). Dans notre cas, il suffit donc d'appeler les plugins maven-compiler-plugin et antlr4-maven-plugin pour que Maven s'occupe de compiler nos fichiers Java en ANTLR (en respectant les dépendances entre fichiers, en plaçant les fichiers générés aux bons endroits...). Si le pom.xml est correctement écrit, la commande mvn compile va donc télécharger tout ce dont elle a besoin, et compiler le projet.
- D'organiser fichiers et répertoires de manière standardisée Pour éviter que chaque projet adopte des conventions différentes des voisins, Maven propose une hiérarchie de répertoire (« Standard Directory Layout » ¹). Par exemple, les sources Java seront dans src/main/java/, les sources ANTLR dans src/main/antlr4/, les fichiers Java utilisés uniquement pour les tests dans src/test/java/, le répertoire target/ sera réservé aux fichiers générés, ...
- De gérer le **lancement des tests automatisés**. Maven permet de lancer une commande ou un programme Java arbitraire (via le plugin **exec-maven-plugin**), ou de lancer des tests JUnit automatiquement. La commande mvn verify va si besoin recompiler le projet et lancer les tests. Pour plus de détails, lire [Tests].
- De **générer des rapports et documentation**. Avec les plugins appropriés (section <reporting> du pom.xml), la commande mvn site va générer un site web avec les rapports de différents outils (documentation Javadoc, couverture des tests Jacoco, analyse statique avec Findbugs, ...) dans le répertoire target/site/. Pour plus de détails, lire [Jacoco].
- De permettre une intégration avec différents environnements de développement (IDE). Un IDE a besoin de connaître les emplacements des fichiers sources, fichiers générés, et des dépendances externes (par exemple, pour proposer une complétion intelligente sur les classes qu'elles contiennent). Le fichier pom.xml fournit toutes ces informations. Netbeans sait nativement utiliser pom.xml, Eclipse peut le faire si on installe le plugin m2e (on peut aussi générer des fichiers de configuration pour eclipse avec mvn eclipse:eclipse). Le fichier pom.xml est donc partagé entre les développeurs (il est ajouté au dépôt Git), et chaque développeur du projet peut choisir l'IDE de son choix (en général, il ne va pas ajouter les fichiers de configurations de son IDE comme .classpath, .project, ... au dépôt Git, car ces fichiers peuvent contenir des préférences personnelles, des noms de fichiers qui n'existent que pour cet utilisateur, ...).

^{1.} http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

3.3 Maven, cycle de vie et cibles

Plus généralement, Maven est parfois décrit comme un outil de gestion du cycle de vie du logiciel ². Il définit les phases suivantes pour le cycle de vie par défaut (on appelle en général Maven via une commande du type mvn phase, toujours depuis le répertoire qui contient pom.xml):

validate : vérifie que le projet est correct et que les informations nécessaires sont disponibles.

compile : compile les sources du projet.

test-compile : compile les sources du projet et des tests (i.e. contenu des répertoires src/tests/*/.

test : fait passer les premiers tests, n'ayant pas besoin que le logiciel soit déployé ou empaqueté (« packaged »). Ces tests incluent les test unitaires (comme ceux utilisant JUnit).

package : empaquette les fichiers compilés dans un format distribuable (par exemple, un fichier JAR incluant les dépendances, un fichier .zip, ...)

integration-test : lance les tests d'intégration. Contrairement à la phase test, on peut utiliser la version empaquetée du projet, et tester ses interactions avec d'autres projets.

verify : vérifie que le paquetage a un niveau de qualité suffisant (par exemple avec des plugins fournissant des métriques de qualité du code).

install : installe le package dans le dépôt Maven local (\$HOME/.m2/), pour pouvoir l'utiliser comme dépendance pour d'autres projets.

deploy : copie le paquetage final sur un dépôt et distant, pour qu'il puisse être utilisé par d'autres développeurs et projets.

Chaque phase de ce cycle de vie dépend de la précédente, donc, par exemple, la commande mvn deploy va lancer tous les tests et vérifications avant de déployer le projet (ce qui devrait éviter de déployer une version buggée par erreur). De la même manière, la commande mvn verify recompile systématiquement le projet, ce qui évite de tester une vieille version compilée si les sources ont déjà évolué. Pour le projet GL, seules les phases compile, test et package seront vraiment utilisées.

D'autres cycles de vies sont définis, avec des phases qui n'ont pas de dépendances avec le cycle par défaut :

site: génère un site web dans target/site/.

clean: supprime les fichiers générés de target.

Des plugins peuvent fournir des « cibles » (« goals »), qui peuvent être ou non rattachés à une phase du cycle de vie du projet. Par exemple, les cibles exec:exec et exec:java du plugin exec-maven-plugin permettent de lancer respectivement une commande et un programme Java quelconque. On peut les utiliser en ligne de commande avec:

```
# Lancer la commande shell: echo Hello World
mvn exec:exec -Dexec.executable=echo -Dexec.args='Hello World'

# Lancer le compilateur Decac avec l'option -b
mvn exec:java -Dexec.mainClass=fr.ensimag.deca.DecacMain -Dexec.args=-b
```

mais on peut aussi les « rattacher » à une phase du cycle de vie, c'est-à-dire demander à ce qu'elles soient exécutées systématiquement quand le cycle de vie est exécuté. Le fichier pom.xml fourni vous donne quelques exemples, comme :

^{2.} http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

Cette portion du fichier dit qu'on va utiliser le plugin exec-maven-plugin, définit l'exécution basic-lex comme étant l'exécution du script ./src/test/script/basic-lex.sh et rattache cette exécution à la phase test, ce qui fait que mvn test va lancer cette commande (et rapporter un échec si la commande termine sur un statut différent de 0).

4 Résumé des commandes utiles pour le projet

Les différentes commandes suivantes se trouvent pour partie dans Projet_GL/src/test/script/, qu'il faudra donc avoir dans le \$PATH.

4.1 Compilation

```
mvn compile : compilation du programme
mvn verify : lancement des tests
mvn clean : suppression de tous les fichiers générés
mvn site : générer la documentation Javadoc et divers rapports
```

4.2 Converture avec Jacoco

Lancement manuel des test et génération du rapport :

mvn jacoco:instrument: instrumente les classes pour enregistrer la couverture.

Lancement manuel d'une série de tests.

mvn jacoco:restore-instrumented-classes : désinstrumente les classes pour les restaurer dans leur version initiale.

mvn jacoco:report : génère le rapport de couverture dans target/site/jacoco.

Lancement automatique:

mvn verify : compile le projet, instrumente les classes, lance l'ensemble des tests, désintrumente les classes et génère le rapport dans target/site/jacoco.

4.3 Gestion du dépôt Git

Voir [SeanceMachine] pour un tutoriel Git adapté au contexte du projet.

```
GITREPO=ssh://depotgl42@depots.ensimag.fr/~/git/
git clone "$GITREPO" Projet_GL/
```

(à faire normalement une seule fois, au début, depuis votre home-directory) Installe la version initiale du projet depuis le dépôt partagé.

git pull

Met à jour le dépôt local à partir de la dernière révision enregistrée dans le dépôt distant. Il faut en général avoir fait un "git commit -a" avant.

git push

Envoie les révisions (commits) locales au dépôt distant.

- (1) git commit -a
- (2) git commit fichier_1 ...

Met à jour le dépôt local à partir (1) de l'ensemble de l'arbre de travail (2) des fichiers ou répertoires (et leurs sous-répertoires) fichier_1, ... Un message de 'log' est demandé pour renseigner les modifications intervenues.

git add fichier_1 ...

Ajoute les fichiers ou répertoires fichier_1, ... à la liste des fichiers gérés par Git. Ils feront donc partie du prochain 'commit'.

- (1) git rm fichier_1 ...
- (2) git rm -r repertoire_1/ ...

Supprime les (1) fichiers fichier_1 ou (2) tous les fichiers contenus dans le répertoire repertoire_1/, ... de la liste des fichiers gérés par Git. Ils ne feront donc plus partie du prochain 'commit'.

git mv fichier_source fichier_cible

Renomme le fichier fichier source en fichier fichier cible. Si on fait un 'git rm' suivi d'un 'git add', Git détecte aussi que c'est un renommage.

git status -a

Affiche le statut des copies locales de tous les fichiers du dépôt local (i.e. le répertoire Projet_GL/ et ses sous-répertoires).

- (1) git \log
- (2) git log fichier_1 ...

Affiche les informations sur les révisions successives (1) de tous les fichiers de l'arbre de travail (2) des fichiers ou répertoires (et leurs sous-répertoires) fichier_1, ... (identifiants de révisions et messages de 'log' entrés lors des 'commit')

gitk

'gitk' fonctionne comme 'git log' (mêmes arguments), mais affiche l'historique sous forme graphique.

- (1) git diff HEAD
- (2) git diff HEAD -- .
- (3) git diff HEAD -- fichier_1 ...

Affiche la différence entre la dernière révision du dépôt local (HEAD) et la version de travail pour (1) tous les fichiers de l'arbre de travail (2) tous les fichiers du répertoire courant (et ses sous-répertoires) (3) les fichiers ou répertoires (et leurs sous-répertoires) fichier_1, ...

git show revision_id

Affiche le commit désigné par revision_id, avec son message et la liste des différences qu'il apporte (dans le même format que 'git diff'). L'identifiant revision_id peut être un identifiant en hexa (court ou long), mais aussi une représentation symbolique. Par exemple, HEAD est le dernier commit, HEAD^ est l'avant dernier, HEAD~3 est l'avant-avant-avant dernier, etc.

git diff revision_id -- fichier_1 ...

Affiche la différence entre l'état du fichier fichier_1 à la révision désignée par revision_id, et sa version courante.

- (1) git diff revision_id_1..revision_id_2
- (2) git diff revision_id_1..revision_id_2 fichier_1 ...

Affiche la différence entre les deux révisions considérées (données par leur identificateur, comme ci-dessus) pour (1) tous les fichiers du dépôt. (2) les fichiers ou répertoires (et leurs sous-répertoires) fichier_1, ...

git gui

Lance une interface graphique qui permet de réaliser la plupart des actions utiles. Attention, cette interface est puissante mais pas évidente à utiliser.

Récupération d'anciennes versions avec Git :

git stash

Met de côté les changements de l'arbre de travail, et revient à la dernière version commitée. Si on change d'avis et qu'on veut ré-appliquer ces changements, on pourra faire 'git stash apply'.

- (1) git checkout HEAD -- .
- (2) git checkout HEAD -- fichier_1

Récupère les dernières révisions des fichiers du dépôt en écrasant les versions locales pour (1) tous les fichiers du répertoire courant et de ses sous-répertoires (2) les fichiers ou répertoires (et leurs sous-répertoires) fichier_1, ... Contrairement à 'git stash', les versions courantes sont perdues.

- (1) git show revision_id:fichier_1
- (2) git show revision_id:fichier_1 > fichier_1

Affiche le contenu du fichier 'fichier_1' tel qu'il était à la révision 'revision_id'. La commande (2) redirige la sortie vers le fichier_1, et donc écrase le contenu courant de ce fichier avec son ancien contenu.

git revert revision_id

Crée un nouveau commit qui annule les modifications introduites par revision_id. À utiliser pour annuler une modification que vous avez déjà 'commitée'.

4.4 Divers

planner

permet de réaliser des plannings avec l'outil planner de gnome.

$[{f Seance Machine}]$

Séance de prise en main de l'environnement du projet

Mise en place

Cette séance machine est prévue pour être réalisée sur les PCs Linux de l'Ensimag. Au démarrage de la séance, choisissez plusieurs machines proches les unes des autres par équipe. Pour les premières étapes, vous pouvez au choix travailler à deux sur la même machine, ou chacun sur sa machine. Pour les manipulations sur Git (section 6), il faudra vous répartir en deux demi-équipes sur deux machines adjacentes. Chaque demi-équipe se connecte sur le compte de l'un de ses membres. Dans tous les cas, chaque étudiant doit avoir réalisé les sections 2 et 3 sur son compte.

En fin de séance, vous pourrez adapter les consignes des sections 2 et 3 sur votre machine personnelle. Ça demande éventuellement d'installer des logiciels supplémentaires comme mvn, ima, etc. Ces manipulations sont décrites sur la page https://projet-gl.pages.ensimag.fr/environnement/machine_perso/. Dans tous les cas, l'environnement de référence reste celui des PCs de l'école, il faudra donc tester votre projet sur ces machines régulièrement et avant chaque rendu.

1 Lire l'actualité du projet sur le site du Projet GL

Un certain nombre d'informations importantes sur le projet sont disponibles sur le site web hébergé sur gitlab. La page d'accueil du projet est :

```
https://projet-gl.pages.ensimag.fr/
```

Une page particulière est à consulter régulièrement :

```
https://projet-gl.pages.ensimag.fr/actualites/
```

Chargez maintenant cette page dans votre navigateur (il peut y avoir eu des corrections depuis l'impression des polycopiés...). Pour être sûr de ne rien rater de l'actualité de cette page, nous vous proposons deux solutions :

- Faire de cette page votre page d'accueil (sous Firefox, menu « Edit », « Preferences », « Main », et sous le champ « Home page », cliquer sur « Use current page »).
- Suivre cette page via votre lecteur RSS favori. Par exemple, sous Thunderbird, on peut recevoir les nouvelles versions de cette page comme on recevrait des emails.

2 Création des répertoires du projet

On suppose dans les explications ci-dessous que vous placez votre projet dans \$HOME/ensimag/-GL/Projet_GL. Vous pouvez bien entendu placer votre projet dans n'importe quel autre répertoire en adaptant les consignes.

Avant tout, il faut connaître votre numéro d'équipe et votre numéro de groupe (qui a été attribué par les enseignants). Pour cela, exécutez la commande suivante :

```
grep " $LOGNAME " /matieres/4MMPGL/GL/global/doc/liste-equipes.txt
```

La sortie devrait être une ligne de texte, indiquant votre numéro de groupe et d'équipe.

```
Lancez la commande git clone suivante pour récupérer le squelette du projet. Dans l'URL, changer « 42 » par votre numéro d'équipe et 8 par votre numéro de groupe : cd git clone git@gitlab.ensimag.fr:gl2024/gr9/gl44 ensimag/GL/Projet_GL chmod -R go-rwx ensimag/GL/Projet_GL ls cd ensimag/GL/Projet_GL ls
```

La commande git clone ne doit pas demander de mot de passe. Si elle en demande un, c'est que vous n'avez pas fourni votre clé publique ssh à gitlab. Cette clé est généralement dans le fichier :

```
$HOME/.ssh/id_rsa.pub
```

Pour entrer votre clé publique sous gitlab, connectez vous sur le site https://gitlab.ensimag.fr. Allez dans le menu settings (en haut à droite), puis sélectionnez l'entrée SSH keys (bandeau de gauche).

3 Modification du path et des variables d'environnement

3.1 Dans le fichier .bashrc

Vérifiez (ou faites en sorte) que la variable PATH contient, au début et dans cet ordre :

- le répertoire global contenant ima (c-à-d. sur pcserveur.ensimag.fr/matieres/4MMPGL/GL/global/bin)
- \$HOME/ensimag/GL/Projet_GL/src/main/bin
- \$HOME/ensimag/GL/Projet_GL/src/test/script
- \$HOME/ensimag/GL/Projet_GL/src/test/script/launchers

Par exemple, ajoutez en fin de fichier .bashrc les lignes

```
JAVA_HOME=/usr/lib/jvm/java-1.18.0-openjdk-amd64
M2_HOME=/opt/maven
MAVEN_HOME=$M2_HOME
PATH=$M2_HOME/bin:"$PATH"
PATH=/matieres/4MMPGL/GL/global/bin:"$PATH"
PATH=$JAVA_HOME/bin:"$PATH"
PATH=$HOME/ensimag/GL/Projet_GL/src/main/bin:"$PATH"
PATH=$HOME/ensimag/GL/Projet_GL/src/test/script:"$PATH"
PATH=$HOME/ensimag/GL/Projet_GL/src/test/script/launchers:"$PATH"
export PATH
```

3.2 Relire le fichier .bashrc

```
source ~/.bashrc
```

3.3 Vérifier que les commandes suivantes donnent les résultats annoncés

```
which mvn ⇒ /opt/maven/bin/mvn
which java ⇒ /usr/lib/jvm/java-16-openjdk-amd64/bin/java
which ima ⇒ /matieres/4MMPGL/GL/global/bin/ima
```

3.4 Tester IMA

IMA est la machine virtuelle cible du compilateur à réaliser pendant le projet (cf. [MachineAbstraite]). Vérifier que la commande "ima -v" (dans ce cas elle répond quelque chose comme "ima nom_de_version").

Pour faire le projet sur votre machine personnelle, vous devrez l'installer en copiant l'archive "/matieres/4MMPGL/GL/ima_sources.tgz". Après extraction, soit le binaire "ima" fonctionne directement, soit vous devrez le recompiler à partir des sources. 1

3.5 Configuration de Git

Il faut maintenant configurer l'outil Git (si ce n'est pas déjà fait) :

```
emacs ~/.gitconfig # ou son éditeur préféré à la place d'Emacs !
```

Le contenu du fichier .gitconfig (à créer s'il n'existe pas) doit ressembler à ceci :

La section [user] est obligatoire. Merci d'utiliser votre vrai nom et votre adresse officielle Ensimag ici, et d'utiliser la même configuration sur toutes les machines sur lesquelles vous travaillez.

La ligne editor de la section [core] définit votre éditeur de texte préféré (par exemple, emacs, nano, vim, gvim -f,... mais évitez gedit qui vous posera problème ici)². Cette dernière ligne n'est pas obligatoire; si elle n'est pas présente, la variable d'environnement VISUAL sera utilisée; si cette dernière n'existe pas, ce sera la variable d'environnement EDITOR.

la section [diff] et la section [color] sont là pour rendre l'interface de Git plus jolie.

La section [push] vous évitera des warnings sur certaines versions de Git, et rend le comportement de git push mieux adapté à notre manière de travailler.

4 Compilation (avec gestion des dépendances) et exécution

Le projet est compilé avec l'outil Maven (commande Unix mvn). Le fichier de configuration de Maven est pom.xml, et l'outil utilise une ensemble de plugins (téléchargés automatiquement à la première exécution) pour compiler (et tester) les différents fichiers de notre projet. Cette section vous propose quelques manipulations pour un premier contact avec l'outil, et vous pourrez trouver plus d'informations dans la section 3 du document [Environnement]. cd ensimag/GL/Projet_GL

```
ls src # On trouve : deux répertoires main/ et test/
ls src/main/ # On trouve : un répertoire par langage (java, antlr, ...)
```

^{1.} Voir détails sur : https://projet-gl.pages.ensimag.fr/environnement/machine_perso/.

^{2.} Si un gedit est déjà lancé, la commande git commit va se connecter au gedit déjà lancé pour lui demander d'ouvrir le fichier, et le processus lancé par git va terminer immédiatement. Git va croire que le message de commit est vide, et abandonner le commit. Il semblerait que gedit -s -w règle le problème, mais cette commande est disponible seulement avec les versions ≥ 3.1.2 de gedit, donc pas sur les machines de l'école, mais peut-être sur vos portables

```
ls examples/
                       # On trouve : des exemples de code pour
                                 vous aider à démarrer.
cd examples/tools
                       # On va dans le projet d'exemple
                       # On trouve : README.md, pom.xml, src
ls
ls target
                       # On trouve : rien (rien n'est compilé, le
                                  répertoire peut ne pas exister)
echo -n | test_lex
                      # Échoue : le projet n'est pas compilé
# se remettre à la racine de votre projet
cd ../..
                       # Compilation du programme (contenu de src/main)
mvn compile
mvn test-compile
                       # Compilation des tests (contenu de src/test)
                         et du programme
On voit apparaître quelque chose qui ressemble à ceci :
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building Example Maven project using various tools
[INFO] task-segment: [compile]
[INFO] -----
[INFO] [resources:resources {execution: default-resources}]
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO] [compiler:compile {execution: default-compile}]
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 6.760s
[INFO] Finished at: Mon Jan 1 00:00:00 MET 2015
[INFO] Final Memory: 9M/107M
[INFO] ------
Si cela ne marche pas, vérifier le fichier .bashrc (cf. section 3).
                       # On trouve : classes (fichiers .class générés)
ls target
                       # generated-sources (vide pour l'instant)
echo -n | test_lex
                      # Affiche "Reading from stdin ..."
mvn compile
                       # Recompilation
On voit apparaître quelque chose qui ressemble à :
[INFO] Scanning for projects...
[INFO] Nothing to compile - all classes are up to date
[INFO] -----
[INFO] BUILD SUCCESSFUL
```

On voit que tout est à jour, et donc rien n'est recompilé.

```
mvn clean  # supprime les .class et autres fichiers générés
mvn compile  # recompile
```

5 Choix et prise en main d'un IDE

De manière à faciliter le développement, nous vous recommandons d'utiliser un IDE. Historiquement, l'IDE recommendé pour le projet GL était Netbeans ³, car premier IDE bien intégré avec Maven. Mais cet IDE semble de moins en moins utilisé et les plugins ANTLRWorks 2, fortement utilisés en projet GL, ne semblent pas avoir été portés sur les dernières versions de Netbeans. Un autre IDE historique est Eclipse (IDE historique du développement en Java) ⁴. Aujourd'hui ces IDEs historiques sont supplantés par d'autres – peut-être mieux adaptés pour développer dans d'autres langages que Java. Notamment, nous conseillons maintenant d'utiliser soit VS Code ⁵ — disponible sur les machines de l'Ensimag via la commande code — soit Intellij IDEA ⁶ — disponible sur les machines de l'Ensimag via la commande idea.sh. Bien sûr, sur votre machine perso, vous pouvez utiliser d'autres IDE que ceux disponibles à l'Ensimag, selon votre choix.

5.1 Installer les plugins Maven et ANTLR4 pour votre IDE

Chaque IDE vient avec sa façon d'installer les plugins. Par exemple, pour VS Code, il suffit d'ouvrir une ligne de commande via le raccourci "Ctrl+P" et de copier-coller la ligne ci-dessous pour installer le plugin Maven :

```
ext install vscjava.vscode-maven
```

De même avec le plugin ANTLR4 (le générateur d'analyseur syntaxique utilisé dans le projet, cf. [ANTLR]) :

```
ext install mike-lischke.vscode-antlr4
```

De façon générale, vous trouverez les plugins spécifiques à votre IDE sur le web. Par exemple, des plugins ANTLR4 pour différents IDE sont indiqués ici : https://www.antlr.org/tools.html.

5.2 Premiers pas dans l'IDE de votre choix

Après installation des plugins nécessaires au projet, il faut prendre un peu temps pour apprendre à maîtriser votre IDE. Ceux-ci fournissent en général une aide en ligne. Le reste de ce document explique cette prise en main pour les IDE Netbeans et Eclipse (i.e. les choix historiques) sur un "mini-projet" examples/tools, dédié à ça, dans un sous-répertoire de Projet_GL . On vous conseille de chercher à adapter ces manipulations avec l'IDE de votre choix.

5.3 Premiers pas avec NetBeans

La version de NetBeans installée à l'Ensimag est la 12.6. Lancez NetBeans (commande unix netbeans), ouvrez maintenant le projet d'exemple. Pour cela, utilisez menu « File » \rightarrow « Open project », puis sélectionnez le répertoire examples/tools (en ligne de commande, on peut faire netbeans --open examples/tools). Remarque : tout répertoire contenant un fichier |pom.xml| est interprété par NetBeans comme un projet et apparaît dans le sélecteur de projet avec une icône | m_a |. Double-cliquer sur ce répertoire provoque l'ouverture du projet correspondant. Si on souhaite naviguer dans ce répertoire pour ouvrir un sous-projet, il faut alors l'ouvrir en sélectionnant le triangle précédant son nom.

```
3. https://netbeans.apache.org/
```

^{4.} https://eclipseide.org

^{5.} https://code.visualstudio.com/

^{6.} https://www.jetbrains.com/idea/

Une fois le projet ouvert, on peut naviguer dans les sources avec la barre latérale de gauche. On peut lancer la compilation avec le bouton « Build project » dans la barre d'outils en haut (ou via la touche f11), et l'exécution via le bouton « Run project » (touche f6).

Pour passer un argument au programme, faites un clic droit sur le projet dans la barre latérale de gauche, puis « Properties ». Choisir la section « Run », puis entrez l'argument (par exemple, toto) dans le champ « Arguments ». Relancez l'exécution, vous devriez voir un résultat différent (c'est un bug volontaire de l'exemple, nous le corrigerons plus tard).

Pour apprendre à naviguer dans les sources du projet GL avec Netbeans, il est conseillé de consulter la présentation http://www-prima.inrialpes.fr/reignier/GL.

5.4 Premiers pas avec Eclipse

Lancez Eclipse (commande unix eclipse). Contrairement à NetBeans, Eclipse ne propose pas un support natif de maven. Deux solutions sont possibles : l'installation d'un plugin supplémentaire (solution privilégiée) et la génération des fichiers projets d'eclipse à partir du projet mvn (transformation du projet maven en projet eclipse natif).

Utilisation d'un plugin

La première solution consiste à augmenter eclipse par l'ajout d'un plugin afin de lui permettre de gérer les projets maven. Plusieurs plugins sont disponibles. Nous vous recommendons d'utiliser m2e. Allez dans le menu « Help » \rightarrow « Eclipse MarketPlace ». Faîtes une recherche sur maven et installez le plugin Maven integration for eclipse (juno and newer) 1.4 (seul m2e est nécessaire, m2e-slf4j est optionnel).

Un projet maven importé sous NetBeans est géré par NetBeans en utilisant directement maven pour le recompiler, l'exécuter etc. Eclipse, à travers le plugin m2e, a une approche différente. Il analyse le contenu du fichier pom.xml, en déduit les actions associées et traduit ces différentes actions vers ses propres mécanismes internes de gestion de projet.

Une fois le plugin m2e installé, le projet peut être importé par le menu « File » \to « Import ». Choisir la sous section « Maven » \to « Existing Maven Projects ».

Attention : toute modification de la configuration du projet doit également être effectuée au niveau du fichier pom.xml. Cette modification est ensuite propagée au niveau d'Eclipse par un clic droit sur le projet suivi de « Maven » \rightarrow « Update Project ».

mvn eclipse:eclipse

Si vous n'êtes pas en mesure d'ajouter le plugin précédent, vous pouvez générer un projet eclipse natif à partir du projet maven. Cette transformation est réalisée par la commande mvn eclipse:eclipse. Le projet peut alors être importé normalement. Attention : si vous souhaitez ajouter de nouvelles dépendances à votre projet (utilisation de jar supplémentaires par exemple), il faut le faire au niveau du fichier pom.xml et ne pas oublier de regénérer le projet eclipse pour propager la modification.

Le projet étant importé, on peut naviguer dans les sources du projet avec la colonne latérale de gauche. Par défaut, eclipse recompile automatiquement un projet lorsque vous sauvegardez un fichier. Vous n'avez donc pas besoin d'action de recompilation explicite.

Le projet peut être lancé en sélectionnant dans l'explorateur de projet (colonne de gauche) le fichier java contenant le main par un clic droit permettant d'ouvrir le menu contextuel, puis « Run As » et « Java Application ». On peut passer des arguments au programme en choisissant « Run As » et « Run Configurations », onglet « Arguments ». Sur notre exemple, ajouter un argument produit un résultat différent (c'est un bug volontaire de l'exemple, nous le corrigerons plus tard).

Remarque : Ctrl F11 permet de relancer directement la dernière exécution.

5.5 Déboguer

Traces de debug et log4j

Ouvrez dans NetBeans ou Eclipse le fichier ./src/main/java/tools/Main.java, qui est le programme principal de notre mini-projet (examples/tools). Ce petit programme comporte un bug : la variable name n'est pas positionné dans la branche else. On peut vérifier à l'exécution qu'en passant un argument à notre script, on a une exception non-rattrapée. Si vous avez suivi les instructions cidessus avec l'IDE que vous avez choisi, vous avez déjà vu le bug (en passant un argument à la classe Main). Vous pouvez le reproduire en ligne de commande :

La pile affichée se lit « Une exception a été levée depuis la méthode Validate.notNull déclarée à la ligne 192 de Validate.java, qui a été appelée par une autre méthode notNull déclarée ligne 178 de Validate.java, appelé par la méthode sayItTo, ... ».

Pour mieux comprendre l'exécution du programme, l'auteur a inséré des traces (les appels à LOG.info() LOG.trace(), ...). Ces traces utilisent la bibliothèque log4j, décrite avec plus de détails en section 5 de [ConventionsCodage]. Ce sont ces appels qui produisent les lignes commençant par INFO, DEBUG et FATAL dans l'exemple ci-dessus. Par rapport à un affichage avec System.out.println(), il y a au moins deux avantages :

- Les traces log4j fournissent plus d'information. Ici, chaque trace donne son niveau d'information, le nom complet de la classe et de la méthode dans laquelle elle a été appelée, avec le numéro de ligne dans le fichier source.
- Les traces log4j sont activables et désactivables sans modifier le code Java. Essayez de modifier le fichier ./src/main/resources/log4j.properties, et modifiez la ligne log4j.rootLogger=INFO, CONSOLE en remplaçant INFO par TRACE. Recompilez (mvn compile) puis relancez le programme. En augmentant le niveau de trace, on vient d'activer la ligne LOG.trace("name = " + name);, qui produit maintenant l'affichage

```
TRACE [tools.Main.main(Main.java:37)] - name = null

(Ici, on se rend compte qu'il y a un problème car name ne devrait pas être nul)
```

La solution présentée ci-dessus permet au programme de produire plus d'affichages et peut permettre de comprendre l'origine d'un problème. Il ne faut néanmoins pas en abuser : cette méthode est intrusive dans le code, et si on ajoute trop de traces, on est rapidement submergé par des milliers de lignes de sortie à chaque exécution. Une autre solution consiste à utiliser un debugger.

Debug avec NetBeans

Au lieu de lancer le programme normalement, relancez-le dans le debugger (le bouton « Debug Project » est à côté du bouton « exécuter », et l'action est accessible au clavier via Control+F5). Si vous avez bien ajouté un argument à l'exécution (cf 5.3), vous devriez reproduire le bug dans le debugger.

Pour rattraper l'exception au moment où elle est levée, il faut configurer NetBeans : menu « Tools » \rightarrow « Options » \rightarrow icône « Java » \rightarrow onglet « Java debugger », cochez la case « stop on uncaught exceptions ». Relancez l'exécution dans le debugger (si besoin, terminez l'exécution précédente avec le bouton « Finish debugger session » en forme de carré rouge, puis utilisez « Debug Project »). Cette fois-ci, l'exécution s'arrête au lancement de l'exception. La ligne en cours d'exécution au moment de l'arrêt apparaît en couleur et avec un marqueur « call stack line » en forme de triangle dans la marge. On peut naviguer dans la pile d'exécution avec Alt+PageUp et Alt+PageDown (les équivalents de « up » et « down » avec GDB). Un onglet « Variables » est disponible en bas de la fenêtre NetBeans, et permet pour chaque cadre de pile de visualiser la valeur des variables. Ici, on peut vérifier que la valeur de la variable name est null (et nous avons fait ceci sans modifier le code et sans utiliser les traces).

Comme tout bon debugger, celui intégré à NetBeans permet de placer des points d'arrêts (clic dans la marge sur le numéro de ligne correspondant, et le point d'arrêt apparaîtra avec un carré rouge), d'avancer pas à pas dans l'exécution (une barre d'outils apparaît en haut pendant que le debugger tourne), ...

Debug avec Eclipse

Relancez le programme en utilisant cette fois-ci l'entrée « Debug As ... » du menu contextuel (sur le source Java contenant le main). Le debugger peut également être lancé directement en reprenant les paramètres de la dernière exécution en utilisant le raccourci « F11 ». Eclipse bascule alors dans la perspective "Debug" en réorganisant les fenêtres ouvertes et leurs dimensions. Une fois le debug terminé, vous pouvez retrouver votre perspective "Java" en la sélectionnant dans les boutons disponibles en haut à droite de l'interface.

Par défaut, le debugger d'Eclipse s'arrête sur une exception lancée et non rattrapée. Ce comportement peut être modifié si nécessaire dans le menu « Run » \rightarrow « Add Java Exception Breakpoint . . . ». Votre programme va donc s'arrêter à l'intérieur de la classe Validate dont les codes sources ne sont pas disponibles. Vous pouvez alors naviguer dans la pile d'exécution (à la souris ou en utilisant directement les flèches du clavier) pour retourner dans la première classe vous appartenant : DireBonjour. La ligne de code correspondante apparaît alors en surlignée. Vous pouvez visualiser la valeur des variables et constater que name est null.

Il est également possible de positionner des points d'arrêt (clic dans la marge de la ligne considérée). On peut également avancer en pas à pas et mettre fin au débug en cliquant sur le carré rouge de la barre d'icones.

6 Utilisation de Git

Si vous n'êtes pas à l'aise avec l'utilisation de Git (commandes commit, push, pull, add, rm, mv, status, ...), lire et réaliser les manipulations proposées par le document [Git] (très similaire à ce que vous avez vu en première année si vous étiez à l'Ensimag). Dans tous les cas, voici quelques rappels et précisions sur l'utilisation de Git dans le cadre du projet. Un récapitulatif des commandes utiles est également disponible dans [Environnement].

Conseils pratiques

- Ne *jamais* s'échanger des fichiers sans passer par Git (email, scp, clé USB), sauf si vous savez *vraiment* ce que vous faites.
- Ne faites pas de modifications autres que celles nécessaires (e.g. ajouter/supprimer des espaces, des lignes blanches, modifier l'indentation) sauf si vous êtes certains que vos collègues ne sont pas en train de modifier les mêmes fichiers. En particulier en utilisant un IDE: ne le laissez pas reformater votre code à tout va (mais formatez-le correctement du premier coup!). On peut tout de même utiliser les fonctionnalités de reformatage de code, mais prennez l'habitude de ne

le faire que sur le code que vous venez d'écrire (Sous NetBeans : sélectionnez la portion de code à reformater, puis Alt-Shift-F).

- Toujours utiliser git commit avec l'option -a.
- Faire un git push après chaque git commit -a, sauf si on veut garder ses modifications privées. Il peut être nécessaire de faire un git pull avant un git push si des nouvelles révisions sont disponibles dans le dépôt partagé.
- Faire des git pull régulièrement pour rester synchronisés avec vos collègues. Il faut faire un git commit -a avant de pouvoir faire un git pull (ce qui permet de ne pas mélanger modifications manuelles et fusions automatiques).
- Ne faites jamais un « git add » sur un fichier binaire généré : si vous le faites, attendez-vous à des conflits à chaque modification des sources! Git est fait pour gérer des fichiers sources, pas des binaires générés.

Si vous vous sentez suffisamment à l'aise avec Git, vous pouvez aussi lire les pages BugBusters « Maintenir un historique propre avec Git » et « Gérer des branches avec Git », qui vous expliqueront que la vie n'est pas si simple, et que la puissance de Git vient de git commit sans -a, des git commit sans git push, ...

7 Tout nettoyer

mvn clean

8 Regarder l'exemple ANTLR (Calculette)

```
cd ensimag/GL/Projet_GL
cd examples/calc
cat README.txt
mvn compile
echo "2 + 2 * 4 - 1" | mvn -q exec:java -Dexec.mainClass=calc.Main
```

La commande mvn - q exec: java permet de lancer une commande Java sans avoir à écrire de script ni à se soucier de positionner le \$CLASSPATH correctement. Ici, on doit obtenir le résultat 2 + 2 * 4 - 1 = 9.

Parcourir les fichiers CalcLexer.g4 et CalcParser.g4 du répertoire src/main/antlr4/calc/pour voir comment un lexer et un parser ANTRL sont écrits. Le fichier src/main/java/calc/Main.java est le programme principal qui montre comment instancier et lancer le lexer et le parser. Les autres fichiers Java correspondent à une structure d'arbre abstrait très simple représentant l'expression analysée.

Vous n'avez rien à faire sur cet exemple, mais il peut aider à comprendre comment utiliser ANTLR et comment représenter un arbre abstrait. Voir le document [ANTLR] pour les explications détaillées sur ANTLR et sur cet exemple.

9 Tester l'outil de planification planner

```
cd ensimag/GL/Projet_GL
cd planning
planner &
```

Ouvrir le fichier Planning.planner. Il s'agit d'un exemple de réalisation d'un début de planning : vous devez le modifier et le compléter. Une fois que vous aurez un planning prévisionnel convaincant, copiez-le avec par exemple : cp Planning.planner Realisation.planner pour obtenir un planning effectif, que vous modifierez tout au long du projet.

Pour plus d'information sur planner, voir [GuidePlanner].

10 Travailler sur l'analyseur lexical

L'utilisation dans un projet d'ANTLR nécessite l'ajout d'un plugin pour sa prise en charge par l'IDE (sans plugin, l'IDE considèrera les fichiers source ANTLR comme de simples fichiers textes). Si votre IDE ne met pas de coloration syntaxique sur les fichiers ANTLR, c'est que le plugin n'est pas installé. Voir section 5.1.

Des explications sur le travail à réaliser en analyse lexicale sont données section 2 du document [Consignes]. Modifiez ./src/main/antlr4/fr/ensimag/deca/syntax/DecaLexer.g4.

—	commentaires et autres séparateurs
—	littéraux
	mots clés
_	

Testez par exemple avec ./src/test/script/launchers/test_lex (nécessite une compilation avec mvn test-compile).

Utilisation basique de Git

Ce document présente les concepts de base avec Git, destiné à des débutants avec Git, ou bien à des étudiants l'ayant déjà utilisé mais désirant réviser leurs bases.

Nous allons maintenant faire quelques modifications au projet d'exemple, pour nous entraîner avec Git. Si vous êtes déjà très à l'aise avec Git, vous pouvez sauter cette section (relisez quand même la section « Conseils pratiques » ci-dessous).

Pour cette section, on suppose que l'équipe est coupée en deux demi-équipes (faites les manipulations à plusieurs sur une machine), qui travaillent sur des machines adjacentes. Nous nommerons les deux demi-équipes Alice et Bob.

Dans le fichier src/main/java/tools/Main.java, Alice ajoute son nom sur la ligne Cauthor, et remplace print par println dans la fonction computeAnswer en bas du fichier. Bob ajoute également son nom sur la ligne Cauthor, mais ne touche pas au print d'Alice, et remplace le 43 par 42 à la fin de computeAnswer.

Les deux demi-équipes lancent maintenant :

Ce qui nous intéresse ici est la ligne « modified : .../Main.java » (la distinction entre « Changed but not updated » et « Changes to be committed » dépasse le cadre de ce cours), qui signifie que vous avez modifié Main.java, et que ces modifications n'ont pas été enregistrées dans le dépôt. On peut vérifier plus précisément ce qu'on vient de faire :

```
git diff HEAD
```

Comme Alice et Bob ont fait des modifications différentes, le diff affiché sera différent, mais ressemblera dans les deux cas à :

```
diff --git a/.../tools/Main.java b/.../tools/Main.java
index 1520f3e..79a0446 100644
--- a/examples/tools/src/main/java/tools/Main.java
+++ b/examples/tools/src/main/java/tools/Main.java
@@ -8,41 +8,40 @@ import org.apache.log4j.Logger;
    *
    * A FAIRE : Alice et Bob rajoutent leurs noms sur la ligne suivante
```

```
*
- * @author ... and ...
+ * @author Alice and ...
*/
public class Main {
```

Les lignes commençant par '-' correspondent à ce qui a été enlevé, et les lignes commençant par '+' à ce qui a été ajouté par rapport au précédent commit. Si vous avez suivi les consignes ci-dessus à propos du fichier .gitconfig, vous devriez avoir les lignes supprimées en rouge et les ajoutées en vert.

Maintenant, Alice et Bob font :

```
git commit -a  # Enregistrement de l'état courant de  # l'arbre de travail dans le dépôt local.
```

L'éditeur est lancé et demande d'entrer un message de 'log'. Ajouter des lignes et d'autres renseignements sur les modifications apportées à Main.java (on voit en bas la liste des fichiers modifiés). Un bon message de log commence par une ligne décrivant rapidement le changement, suivie d'une ligne vide, suivie d'un court texte expliquant *pourquoi* la modification est bonne. Vous trouverez plus d'informations sur le sujet sur la page du projet GL « Écrire de bons messages de commit avec Git ».

On voit ensuite apparaître :

```
[master 2483c22] Ajout de mon nom et correction d'un bug
1 files changed, 2 insertions(+), 12 deletions(-)
```

Ceci signifie qu'un nouveau « commit » (qu'on appelle aussi parfois « revision » ou « version ») du projet a été enregistrée dans le dépôt. Ce commit est identifié par une chaîne hexadécimale (« 2483c22 » dans notre cas).

On peut visualiser ce qui s'est passé avec les commandes

```
gitk
                    # Visualiser l'historique graphiquement
et
git gui blame Main.java
                             # voir l'historique de chaque
                              # ligne du fichier Main.java
   On va maintenant mettre ce « commit » à disposition des autres utilisateurs.
   SEULEMENT Bob fait :
git push
                    # Envoyer les commits locaux dans
                    # le dépôt partagé
Pour voir où on en est, les deux équipes peuvent lancer la commande :
gitk
                    # afficher l'historique sous forme graphique
ou bien
git log
                    # afficher l'historique sous forme textuelle.
A PRESENT, Alice peut tenter d'envoyer ses modifications :
git push
On voit apparaître :
To git@gitlab.ensimag.fr:gl2024/gr9/gl44
                     master -> master (non-fast forward)
error: failed to push some refs to 'git@gitlab.ensimag.fr:gl2024/gr9/gl44'
```

To prevent you from losing history, non-fast-forward updates were rejected Merge the remote changes (e.g. 'git pull') before pushing again. See the 'Note about fast-forwards' section of 'git push --help' for details.

L'expression « non-fast forward » (qu'on pourrait traduire par « absence d'avance rapide ») veut dire qu'il y a des modifications dans le dépôt vers lequel on veut envoyer nos modifications et que nous n'avons pas encore récupérées. Il faut donc fusionner les modifications avant de continuer.

L'utilisateur Alice fait donc :

```
git pull
```

Après quelques messages sur l'avancement de l'opération, on voit apparaître :

```
Auto-merging examples/tools/src/main/java/tools/Main.java
CONFLICT (content): Merge conflict in examples/tools/src/main/java/tools/Main.java
Automatic merge failed; fix conflicts and then commit the result.
```

Ce qui vient de se passer est que *Bob* et *Alice* ont fait des modifications au même endroit du même fichier dans les commits qu'ils ont fait chacun de leur côté, et Git ne sait pas quelle version choisir pendant la fusion : c'est un conflit, et nous allons devoir le résoudre manuellement. Allez voir Main.java.

On peut remarquer que Git a fusionné automatiquement les modifications faite à la fonction computeAnswer : le print est bien devenu println, et le 43 est devenu un 42. Quand une équipe est bien organisée et évite de modifier les mêmes endroits en même temps, ce cas est le plus courant : les développeurs font les modifications, et le gestionnaire de versions fait les fusions automatiquement.

Par contre, la ligne **@author** a été modifiée en parallèle par deux développeurs, et ne peut être fusionnée automatiquement. Git propose les deux versions, et demande à l'utilisateur de choisir la bonne ou de fusionner manuellement :

```
<<<<< HEAD:examples/tools/src/main/java/tools/Main.java
  * @author Alice and ...
======
  * @author ... and Bob
>>>>>> 2483c22:examples/tools/src/main/java/tools/Main.java
```

Les lignes entre <<<<< et ====== contiennent la version de votre commit (qui s'appelle HEAD). les lignes entre ====== et >>>>> contiennent la version que nous venons de récupérer par « pull » (nous avions dit qu'il était identifié par la chaîne 2483c22, vous avez sans doute une version plus longue mais équivalente dans votre fichier).

Il faut alors « choisir » dans Main.java la version qui convient (ou même la modifier). Ici, on supprimera les marqueurs de conflits et on éditera le résultat pour le faire ressembler à :

```
* A FAIRE : Alice et Bob rajoutent leurs noms sur la ligne suivante
 * @author Alice and Bob
 */
   Si Alice fait à nouveau
git status
On voit apparaître:
# On branch master
# Your branch and 'origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
# Unmerged paths:
    (use "git add/rm <file>..." as appropriate to mark resolution)
#
#
#
        both modified:
                             src/main/java/tools/Main.java
#
```

Si on n'est pas sûr de soi après la résolution des conflits, on peut lancer la commande :

```
git diff  # git diff sans argument, alors qu'on avait  # l'habitude d'appeler 'git diff HEAD'

Après un conflit, Git affichera quelque chose comme:

diff --cc examples/tools/src/main/java/tools/Main.java
index 9b4821a,ffbfc38..0000000
--- a/examples/tools/src/main/java/tools/Main.java
+++ b/examples/tools/src/main/java/tools/Main.java
@@@ -8,7 -8,7 +8,7 @@@ import org.apache.log4j.Logger
    *
    * A FAIRE : Alice et Bob rajoutent leurs noms sur la ligne suivante
    *
    * @author Alice and ...
    - * @author ... and Bob
++ * @author Alice and Bob
```

(les '+' et les '-' sont répartis sur deux colonnes, ce qui correspond aux changements par rapport aux deux « commits » qu'on est en train de fusionner. Si vous ne comprenez pas ceci, ce n'est pas très grave!)

Après avoir résolu manuellement les conflits à l'intérieur du fichier, on marque ces conflits comme résolus, explicitement, avec git add :

```
$ git add src/main/java/tools/Main.java
$ git status
# On branch master
# Your branch and 'origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
#
# Changes to be committed:
#
# modified: src/main/java/tools/Main.java
#
```

On note que Main. java n'est plus considéré « both modified » (i.e. contient des conflits non-résolus) par Git, mais simplement comme « modified ».

Quand il n'y a plus de fichier en conflit, il faut faire un commit (comme « git pull » nous l'avait demandé) :

```
git commit -a
```

*/

public class Main {

Un éditeur s'ouvre, et propose un message de commit du type « Merge branch 'master' of ... », on peut le laisser tel quel, sauver et quitter l'éditeur.

NB : si il n'y avait pas eu de conflit, ce qui est le cas le plus courant, « git pull » aurait fait tout cela : télécharger le nouveau commit, faire la fusion automatique, et créer si besoin un nouveau commit correspondant à la fusion.

On peut maintenant regarder plus en détails ce qu'il s'est passé :

gitk

Pour *Alice*, on voit apparaître les deux « commit » fait par *Bob* et *Alice* en parallèle, puis le « merge commit » que nous venons de créer avec « git pull ». Pour *Bob*, rien n'a changé.

La fusion étant faite, Alice peut mettre à disposition son travail (le premier commit, manuel, et le « merge commit ») avec :

```
git push
et Bob peut récupérer le tout avec :
git pull
(cette fois-ci, aucun conflit, tout se passe très rapidement et en une commande)
   Les deux utilisateurs peuvent comparer ce qu'ils ont avec :
gitk
ils ont complètement synchronisé leur répertoires. On peut également faire :
git pull
git push
Mais ces commandes se contenteront de répondre Already up-to-date. et Everything up-to-date.
   A PRESENT, Alice crée un nouveau fichier, Toto. java, contenant par exemple:
class Toto { }
Alice fait
git status
On voit apparaître:
# On branch master
# Untracked files:
#
    (use "git add <file>..." to include in what will be committed)
        Toto.java
nothing added to commit but untracked files present (use "git add" to track)
Notre fichier Toto. java est considéré comme « Untracked » (non suivi par Git). Si on veut que
Toto. java soit ajouté au dépôt, il faut l'enregistrer (git commit ne suffit pas) : git add Toto. java
   Alice fait à present :
git status
On voit apparaître:
# On branch master
# Changes to be committed:
#
    (use "git reset HEAD <file>..." to unstage)
#
        new file:
                      Toto.java
#
Alice fait à présent (-m permet de donner directement le message de log) :
git commit -m "ajout de Toto.java"
```

On voit apparaître:

```
[master b1d56e6] Ajout de Toto.java
 1 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 Toto.java
Toto. java a été enregistré dans le dépôt. On peut publier ce changement :
git push
Bob fait à présent :
git pull
Après quelques messages informatifs, on voit apparaître :
Fast forward
 Toto.java |
                 1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 Toto.java
Le fichier Toto. java est maintenant présent chez Bob.
   Bob crée à présent un nouveau fichier temp-file.txt, puis fait :
git status
On voit maintenant apparaître:
# On branch master
# Untracked files:
    (use "git add <file>..." to include in what will be committed)
#
#
        temp-file.txt
nothing added to commit but untracked files present (use "git add" to track)
Si Bob souhaite que le fichier temp-file.txt ne soit pas enregistré dans le dépôt (soit « ignoré » par
Git), il doit placer son nom dans un fichier .gitignore dans le répertoire contenant temp-file.txt.
Concretement, Bob tape la commande
emacs .gitignore
et ajoute une ligne
temp-file.txt
puis sauve et on quitte. Pour que tous les utilisateurs du dépôt bénéficient du même fichier .gitignore,
Bob fait:
git add .gitignore
Bob fait a nouveau
git status
On voit apparaître:
# On branch master
# Changes to be committed:
    (use "git reset HEAD <file>..." to unstage)
#
#
#
        new file:
                      .gitignore
```

Quelques remarques:

— Le fichier temp-file.txt n'apparaît plus. C'était le but de la manoeuvre. Une bonne pratique est de faire en sorte que « git status » ne montre jamais de « Untracked files » : soit un fichier doit être ajouté dans le dépôt, soit il doit être explicitement ignoré. Cela évite d'oublier de faire un « git add ».

- En général, on met dans les .gitignore les fichiers générés (*.o, fichiers exécutables, ...), ce qui est en partie fait pour vous pour ce projet.
- Le fichier .gitignore vient d'être ajouté (ou bien il est modifié si il était déjà présent). Il faut à nouveau faire un commit pour que cette modification soit disponible pour tout le monde.

A ce stade, vous devriez avoir les bases pour l'utilisation quotidienne de Git. Bien sûr, Git est bien plus que ce que nous venons de voir, et nous encourageons les plus curieux à se plonger dans le manuel utilisateur et les pages de man de Git pour en apprendre plus. Au niveau débutant, voici ce qu'on peut retenir :

Les commandes

git commit -a enregistre l'état courant du répertoire de travail.
git push publie les commits
git pull récupère les commits publiés
git add, git rm et git mv permettent de dire à Git quels fichiers il doit gérer.
git status pour voir où on en est

[Tests]

Validation du compilateur Deca

Une part importante de l'effort de développement d'un compilateur (et de tout logiciel en général) consiste à s'assurer que le code que l'on écrit implante fidèlement les fonctionnalités que l'on attend de ce compilateur. Pour réaliser cette tâche (appelée « validation »), plusieurs techniques ayant pour but d'accroître la confiance que l'on a dans le fonctionnement du compilateur peuvent être utilisées : test, revue de code, analyses statiques sur le code, preuves et vérifications formelles, etc.

Dans ce document on s'intéresse à la validation du compilateur par les tests et l'on donne quelques repères en ce qui concerne la conception et l'utilisation d'un jeu de tests et des consignes à respecter pour l'organisation de vos tests.

Dans le cadre du projet GL, un accent particulier est mis sur cette forme de validation. Vous serez évalués sur divers aspects du génie logiciel, dont la qualité de votre compilateur pour compiler des programmes Deca, mais aussi la qualité des tests que vous aurez faits pour produire un bon compilateur. Il est donc TRÈS IMPORTANT de :

- consacrer une part significative de votre activité au test
- respecter les conventions que nous vous imposons pour le nommage et l'architecture de vos tests

1 Conseils généraux sur les tests

1.1 À propos de tests

D'une façon générale, les tests ont pour but d'évaluer la qualité d'un système, pour nous d'un logiciel qui est le compilateur Deca. Il existe différents facteurs de la qualité, dont la correction fonctionnelle (le système fait bien ce qu'on attend de lui) qui est la principale caractéristique que vos tests doivent vérifier, mais aussi ses performances (comme la rapidité à fournir un résultat, mais aussi son efficience dans l'utilisation des ressources), et d'autres facteurs qu'on ne regardera pas dans le cadre du projet (comme l'utilisabilité à travers l'ergonomie, la sécurité etc). Nous allons nous intéresser principalement aux tests de correction fonctionnelle (appelé aussi tests de conformité puisqu'il faut que le logiciel soit conforme aux spécifications).

Un test a pour but non pas de prouver que le compilateur fonctionne correctement mais plutôt d'exhiber le plus grand nombre d'erreurs possibles, s'il y en a, ou de s'assurer qu'il en reste le moins possible. Pour ceci il faut que l'ensemble des tests conçus couvre la plus grande partie possible du code écrit et de son fonctionnement. Au minimum, on pourrait espérer que les tests sollicitent au moins une fois chaque instruction du compilateur. Ceci n'est pas forcément possible à faire, mais on doit au moins procéder de manière systématique. Lors de la conception d'un test, il faut avoir toujours en tête les questions suivantes :

- Quels sont les documents spécifiant la partie du code que je veux tester?
- Quelle partie du code de mon compilateur je veux tester?
- Avec quelles données en entrée je peux exécuter la partie du code en question?
- Est-ce que le test que je suis en train d'écrire fait bien exécuter la partie du code qui m'intéresse?

En procédant ainsi, on a de meilleures chances d'écrire des tests utiles, peu redondants, et on a une meilleure idée de la couverture de code réalisée par le jeu de tests.

[Tests] 144

1.2 Conception de tests

Le compilateur Deca du projet est décomposé en trois étapes. Chaque étape contient ses propres structures de données, algorithmes et... erreurs. On peut donc concevoir un jeu de tests différent pour chaque étape.

En ce qui concerne l'étape A, les tests portent essentiellement sur la construction de l'arbre abstrait (document [SyntaxeAbstraite]). Une façon d'éviter les recouvrements consiste à créer un test pour chaque non-terminal de la grammaire d'arbres, sans oublier tous les cas concernant les listes. D'autres tests concernant la lexicographie (par exemple chaînes non valides) doivent aussi être élaborés.

En ce qui concerne l'étape B, les tests portent aussi bien sur le parcours de l'arbre abstrait pour effectuer les vérifications, que sur la gestion des environnements, les décorations et les enrichissements de l'arbre (documents : [SyntaxeContextuelle], [ArbreEnrichi]). On peut distinguer deux catégories de tests : une qui teste les vérifications à tous les nœuds de l'arbre abstrait (pour avoir une grande couverture) et une autre qui teste tous les cas d'erreur possibles pouvant être rencontrés lors de cette étape.

En ce qui concerne l'étape C, les tests portent sur les différentes constructions Deca, s'exécutant correctement ou provoquant une erreur à l'exécution, ainsi que sur les séquences d'instructions assembleur (documents : [Semantique], [Gencode]). Ayant en tête l'instruction assembleur que l'on veut produire on induira le programme Deca qui la fait générer. Les tests peuvent aussi porter sur des cas de figures « exotiques ». Par exemple : très grandes expressions nécessitant beaucoup de registres/temporaires pour tester la politique d'allocation des registres, procédures récursives divergentes pour tester les débordements de pile, etc.

Associé au test de cette étape C, il y a l'étape d'exécution du code produit. On peut concevoir des petits programmes Deca qui contiennent plus que quelques instructions (par exemple des petits jeux, ... etc.), pour voir si les résultats obtenus à l'exécution sont cohérents.

Il ne faut pas oublier de tester toutes les options obligatoires (et facultatives) du compilateur car elles imposent implicitement des contraintes supplémentaires sur les différentes étapes (par exemple s'assurer que quand on appelle decac avec l'option « -r 4 » on ne génère pas du code manipulant des registres parmi R4, R5, ..., R15).

De manière générale on peut identifier, pour chaque test, un certain nombre d'attributs qui peuvent aussi guider la conception :

- Le contexte d'utilisation du test : identification de l'étape concernée par le test (étape A, B ou C, exécution du code généré).
- La nature du test : le programme-test doit être accepté/refusé par le compilateur, c'est-à-dire qu'il correspond à un programme Deca valide (respectant la syntaxe et la sémantique du langage) ou invalide. Les tests ont pour but, soit de faire générer par le compilateur le code correct, pouvant s'exécuter normalement ou provoquer une erreur à l'exécution (pour tester que le compilateur est conforme à ses spécifications), soit de provoquer une erreur de compilation (pour tester si le compilateur se comporte de manière cohérente lors de la découverte d'une erreur et respecte les messages d'erreur spécifiés).
- La manière d'utiliser le test (si le test affiche ses propres résultats ou s'il faut vérifier sa validité à la main) : on peut concevoir des tests qui se vérifient automatiquement. Par exemple le test suivant porte sur l'utilisation de la conversion de type et affiche tout seul le résultat du test :

```
{
  float f;
  print ("test que float f = 1 fait que f == 1.0 : ");
  f = 1;
  if (f == 1.0) {
    print("ok");
  } else {
    print("ERRONE");
  }
```

}

Nous vous indiquons dans les conventions comment catégoriser vos tests pour que nous puissions les évaluer automatiquement.

1.3 Exécution des tests

Au fur et à mesure que les jeux de tests sont conçus, il faut les placer dans l'environnement du projet et les faire passer sur le compilateur (ou la partie du compilateur concernée).

Pour exécuter les tests de l'étape A, on peut utiliser les programmes test_synt et test_lex fournis par défaut.

Pour automatiser l'exécution de test_synt, test_lex, test_context et decac sur un ensemble précis de tests on peut utiliser le script suivant :

```
#!/bin/sh
```

```
for i in ./src/test/deca/syntax/valid/*.deca
do
    echo "$i"
    # Remplacer <executable> par test_synt ou test_lex
    # ou test_context ou decac
    <executable> "$i"
done
```

Si on suppose que le répertoire ./src/test/deca/syntax/valid/ contient au moins un fichier .deca, le script exécutera automatiquement et de manière séquentielle <executable> sur les fichiers Deca du répertoire.

On peut diriger la sortie de chaque exécution vers un fichier résultat pour avoir une trace synthétique de toutes les exécutions des tests passés en paramètre. Ceci permet en particulier de comparer systématiquement les résultats obtenus lorsque l'exécutable a été modifié (tests dits de non régression). On peut le faire en remplaçant la ligne <executable> "\$i" par

```
<executable> "$i" > "${i%.deca}".res
```

qui créera un fichier .res pour chaque fichier .deca considéré, contenant le résultat de l'exécution de l'exécutable.

1.4 Exemple de convention de nommage

Pour archiver les résultats attendus des tests, il faut décider dans quel fichier stocker quel résultat. Par exemple, le « résultat » d'un test peut être un message d'erreur (sortie du compilateur), un fichier assembleur, ou bien le résultat de l'exécution du fichier assembleur.

Une convention possible est (pour le fichier source test.deca):

test.lis messages de votre compilateur decac sur le fichier test.deca. Utilisé en étapes A et B seulement.

test.ass en étape C, programme interprétable par IMA. Utilisé en étape C seulement.

test.res résultat de l'exécution du fichier test.ass par IMA

N.B. Le contenu du fichier .lis (pour « listage » de compilation) peut dépendre de la façon dont vous avez testé : avec test_synt, il contient des affichages d'arbres par exemple; mais si vous avez utilisé le compilateur decac, appelé sans option -p ni -w, il devrait être vide (ou le fichier .lis peut ne pas exister) pour les programmes valides. Le fichier .lis n'est pas créé par la commande decac (ou test_synt, ou test_context), mais c'est votre script de test qui stockera la sortie standard (et la sortie d'erreur) produite par votre compilateur (ou test_synt etc.) dans un tel fichier. De même .res correspondra à la sortie produite par l'exécution sous IMA de votre fichier .ass.

1.5 Nom des tests

Dans l'exemple d'utilisation du script automatique, on donne aussi un exemple de nommage des tests à éviter! En fait, si on donne aux tests des noms qui sont très uniformes et sans liaison avec leur contenu on oublie très vite ce que chaque test fait! Il faut donc trouver un bon compromis entre des noms à rallonge comme test_types_incompatibles_operation_binaire_etapeB_invalide.deca et des noms sibyllins comme test1.deca.

Dans tous les cas, il est obligatoire, dès l'étape A, que les noms des tests soient suffixés par « .deca ». Ceci permettra de les faire passer par le compilateur final 'decac' (qui n'accepte que des fichiers ayant un tel suffixe).

1.6 Automatisation des tests

La section précédente propose une méthode simple pour exécuter plusieurs tests automatiquement. Mais il est aussi essentiel de vérifier que les tests se sont bien passés (ce qu'on appelle « l'oracle »).

Il faut donc améliorer le petit script proposé, probablement en écrire plusieurs, pour lancer et v'erifier l'exécution du test. On vous demande de respecter la convention suivante :

Un script de test pour lequel au moins un test a échoué doit renvoyer un statut différent de 0 (par exemple, « exit 1 »), un script de test n'ayant détecté aucun échec doit renvoyer 0 (fin du script, ou « exit 0 »).

A partir de là, il est très simple d'utiliser la cible « mvn test ». Si vous avez écrit deux scripts de tests, disons un-test.sh et un-autre-test.sh, il vous suffit d'ajouter les sections correspondantes dans le fichier pom.xml

```
. . .
  <artifactId>exec-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>un-test</id>
      <configuration>
        <executable>./src/test/script/un-test.sh</executable>
      </configuration>
      <phase>test</phase>
      <goals><goal>exec</goal></goals>
    </execution>
    <execution>
      <id>un-autre-test</id>
      <configuration>
        <executable>./src/test/script/un-autre-test.sh</executable>
      </configuration>
      <phase>test</phase>
      <goals><goal>exec</goal></goals>
    </execution>
  </executions>
</plugin>
. . .
```

et de taper « mvn test ». Votre projet sera recompilé, puis les deux scripts seront exécutés, et Maven vous dira si quelque chose s'est mal passé. À la fin du projet, « mvn test » devra faire passer tous les tests sur votre compilateur (ce qui peut prendre plusieurs minutes si vous avez beaucoup de tests). Remarque : « mvn test » exécute l'ensemble de vos tests mais laisse les classes instrumentées par Jacoco pour la mesure de couverture. Il est préférable d'utiliser « mvn verify » qui réalise la phase

test au préalable (équivalent à un « mvn test ») mais désinstrumente également les classes et génère le rapport de couverture. Pour plus détails, lire [Jacoco].

Quelques scripts d'exemples sont fournis dans le répertoire src/test/script/ de votre arborescence initiale. Vous pouvez vous en inspirer pour en écrire de meilleurs (il est conseillé de conserver les scripts originaux dans votre dépôt Git et dans le fichier pom.xml, pour vous assurer que ce qui est testé par les scripts d'exemples est effectivement testé sur la version finale de votre compilateur). Un bon point de départ pour écrire des scripts shells est disponible sur EnsiWiki:

```
http://ensiwiki.ensimag.fr/index.php/Script_Shell
```

La commande « mvn test » doit impérativement être non-interactive (i.e. exécuter tous les tests sans que l'utilisateur ne doive taper quoi que ce soit au clavier, que les tests réussissent ou échouent). Ceci est indispensable pour que l'équipe enseignante puisse évaluer votre base de tests.

Par défaut, Maven n'exécutera pas les tests via le plugin exec-maven-plugin si les tests JUnit ont échoué (cf. section suivante). Il arrive cependant qu'on souhaite exécuter ces tests même si un test unitaire a échoué (par exemple en début de projet vu que les enseignants vous fournissent quelques tests unitaires qui ne passent pas dans le squelette). On peut demander à Maven d'exécuter les scripts de tests même quand les tests unitaires échouent avec la commande suivante :

```
mvn test -Dmaven.test.failure.ignore
```

1.7 Automatisation des tests unitaires et JUnit

Un test unitaire est un test qui cible une petite partie du code du programme à tester. On peut par exemple tester unitairement une classe, voire une méthode, en l'isolant du reste du programme. Pour fixer les idées, disons que l'on cherche à tester une classe monpaquetage.MaClasse.

Le langage Java permet d'avoir plusieurs méthodes main par projet, donc une solution pour écrire un test unitaire est d'écrire une classe (disons monpaquetage.MonTestManuel) contenant une méthode main qui va appeler les méthodes de monpaquetage.MaClasse, et vérifier que les résultats sont ceux attendus. Ensuite, on pourra lancer le test en ligne de commande avec

```
java monpaquetage.MonTestManuel
```

ou bien automatiser le lancement avec Maven dans le pom.xml avec le plugin exec-maven-plugin :

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>exec-maven-plugin</artifactId>
      <version>1.2.1
      <executions>
        <execution>
          <id>unit-test-maclasse</id>
          <configuration>
            <mainClass>monpaquetage.MonTestManuel</mainClass>
            <classpathScope>test</classpathScope>
          </configuration>
          <phase>test</phase>
          <goals><goal>java</goal></goals>
        </execution>
```

Sur cet exemple, on utilise la cible java du plugin, qui va lancer un programme Java (dans la même JVM que Maven). On demande à positionner le \$CLASSPATH correctement pour les tests avec classpathScope) (i.e. inclure les dépendances qui ont été déclarées avec <scope>test</scope> parce

qu'elles ne sont utilisées que par les tests), et on rattache l'exécution à la phase de cycle de vie test, ce qui fait que mvn test lancera systématiquement ce test, et s'arrêtera si le test échoue (par exemple en levant une exception non-rattrapée).

Dans l'exemple ci-dessus, en utilisant Maven, la classe MaClasse doit être implémentée dans le fichier src/main/java/monpaquetage/MaClasse.java, et le test dans src/test/java/monpaquetage/MonTestManuel.java.

La solution ci-dessus a l'avantage d'être utilisable en Java et sans outil supplémentaire. En pratique, il est en général plus pratique d'utiliser un outil dédié aux tests unitaires comme JUnit, qui est très facile à apprendre et offre un certain nombre d'avantages (intégration avec Maven et les IDE comme Netbeans et Eclipse, génération de rapports plus ou moins détaillés, ...). La suite de cette section présente rapidement JUnit, version 4.

Un test JUnit ressemble à ceci (exemple tiré de votre squelette de code) :

```
package fr.ensimag.deca.tools;
import fr.ensimag.deca.tools.SymbolTable.Symbol;
import static org.junit.Assert.*;
import org.junit.Test;
/**
 * Example JUnit class. A JUnit class must be named TestXXX or YYYTest
 * JUnit will run each methods annotated with @Test.
public class SymbolTest {
    @Test
    public void testSymbol() {
        SymbolTable t = new SymbolTable();
        Symbol s1 = t.create("foo");
        // Check that two objects are equal (using Object.equals())
        assertEquals(s1.getName(), "foo");
        Symbol s2 = t.create("foo");
        // Check that two objects are the same (same reference)
        assertSame(s1, s2);
    }
    @Test
    public void multipleTables() {
        SymbolTable t1 = new SymbolTable();
        SymbolTable t2 = new SymbolTable();
        Symbol s1 = t1.create("foo");
        Symbol s2 = t2.create("foo");
        assertEquals(s1.getName(), s2.getName());
        assertNotSame(s1, s2);
    }
}
```

Les points importants sont :

- Un test unitaire JUnit est une classe, dont le nom commence ou termine par Test. Avec Maven, le fichier source doit se trouver dans src/test/java.
- Une méthode est considérée comme un cas de test, et sera appelée par JUnit, si elle est décorée avec l'annotation @Test (qui nécessite import org.junit.Test; en haut de fichier).
- Pour tester que le résultat obtenu correspond au résultat attendu, on utilise les méthodes

assert*¹ (qui nécessitent import static org.junit.Assert.*; en haut de fichier). Les méthodes assert* ont en général deux variantes : l'une qui fait simplement le test, et l'autre avec un argument supplémentaire (premier argument) qui décrit ce que fait l'assertion. Par exemple, assertEquals("descriptif", valeurAttendue, valeurObtenue) teste l'égalité de valeurAttendue et valeurObtenue, si le test échoue, va afficher générer un rapport contenant "descriptif".

Une fois les tests JUnit écrits, on peut les exécuter :

- Depuis son IDE. Sous Netbeans, clic-droit sur le fichier, puis choisir « test file » dans le menu. Sous Eclipse, clic-droit sur le fichier (ou sur un paquetage pour en lancer plusieurs d'un coup), puis « Run As \rightarrow JUnit test » (Control+Alt+x puis t).
- Via Maven, en lançant simplement « mvn test » (la commande affiche une section T E S T S signalant les éventuels échecs, en redirigeant l'utilisateur vers le rapport détaillé). Il n'y a rien à ajouter au pom.xml: le plugin JUnit est déjà là et trouve tout seul la liste des test à exécuter.

Le lancement depuis l'IDE est pratique pour lancer des tests individuellement, obtenir un retour rapide sur les échecs (il suffit de cliquer sur la ligne de l'échec pour avoir le retour aux sources sur l'assertion qui a échoué), et le lancement depuis Maven permet de s'assurer que tous les tests ont bien été lancés (on pourrait par exemple faire en sorte qu'un git push vérifie que les tests passent avant de mettre à jour le dépôt partagé).

JUnit propose bien d'autres fonctionalités, en général accessible avec des annotations en plus de **©Test**. Lire la documentation sur le site de JUnit ², et les exemples fournis dans votre squelette de code.

1.8 Utilisation des tests

Une base de tests bien automatisée permet non seulement de valider le produit final, mais aussi (et surtout!) d'être utilisée tout au long du projet. En fait, il est même conseillé d'écrire les cas de tests et l'ensemble de scripts avant de coder la fonctionnalité à tester. Ainsi, la personne qui écrit le code aura un retour immédiat sur la correction de ce qu'elle écrit. On parle alors de développement piloté par les tests (« Test Driven Development » ou TDD).

Un point très important est que ce qui a été testé une fois doit être re-testé régulièrement, pour éviter que des fonctionnalités existantes ne soient cassées plus tard. Là encore, une bonne automatisation est la clé : si une validation manuelle du résultat d'un test est nécessaire, il faut archiver le résultat validé manuellement, et la prochaine exécution de la base de tests peut vérifier que le résultat est toujours conforme à sa référence avec une simple comparaison (commande « diff » par exemple).

Avec tout ceci, la base de tests peut être exécutée très régulièrement, par exemple avant chaque commit, et certainement plusieurs fois par jours.

1.9 Quelques exemples d'oracles automatiques

Nous avons vu plus haut que vos scripts, et en particulier l'exécution de la commande « mvn test » devaient vérifier automatiquement si l'exécution d'un test était correcte ou non (on parle d'oracle).

Voici quelques exemples d'oracles simples à implémenter de manière automatique :

- Vérifier qu'un programme qui doit lever une erreur à la compilation la lève effectivement (si possible en vérifiant le numéro de ligne, et au minimum en vérifiant que le numéro de ligne est différent de 0).
- Vérifier que la compilation de tous les programmes Deca corrects de la base ne produit pas de sortie à l'écran.
- Vérifier qu'aucun programme de test ne lève d'exception Java non-rattrapée à la compilation.
- Vérifier qu'aucun programme de test ne lève d'erreur ligne 0.

^{1.} http://junit.sourceforge.net/javadoc/org/junit/Assert.html

^{2.} http://www.junit.org/

— La décompilation d'un arbre abstrait est une opération idempotente : soit P1 un programme source, P2 la décompilation de l'arbre abstrait correspondant à P1, P3 la décompilation de l'arbre abstrait correspondant a P2; on doit avoir P3 = P2. Une application simple de cette propriété consiste à utiliser la commande Unix « diff » pour vérifier que P3 = P2.

À ces oracles entièrement automatiques, il faut ajouter la comparaison avec un fichier de référence mentionnée plus haut. Cette technique demande une validation manuelle, mais reste sans doute la plus efficace.

1.10 Pour aller plus loin : Mockito pour les tests unitaires

L'utilisation de Mockito n'est pas réellement nécessaire pour le projet GL, mais c'est un outil très utilisé en complément de JUnit pour les tests unitaires. Voici donc quelques explications pour les plus curieux d'entre vous.

Tester unitairement une classe qui n'a aucune dépendance est assez facile. La situation se complique quand on souhaite tester une classe qui dépend d'autres classes : un bon test unitaire ne doit utiliser que la classe en question et pas ses dépendances. Par exemple, si on souhaite tester la classe Plus, il faut pouvoir l'instancier, et le constructeur de Plus a besoin de deux instances de AbstractExpr. On pourrait utiliser d'autres nœuds de l'arbre comme bouchons, comme par exemple

```
Plus plus = new Plus(new IntLiteral(42), new IntLiteral(12));
```

Mais cette solution n'est pas vraiment un test unitaire : on utilise la classe IntLiteral alors qu'on devrait utiliser uniquement Plus. Une deuxième solution est de créer une classe dérivant de AbstractExpr uniquement pour le test. Cette solution est implémentée dans le fichier src/test/java/fr/ensimag/deca/context/TestPlusWithoutMock.java. Elle est conceptuellement correcte, mais demande d'écrire beaucoup de code pour un test très simple.

bibliothèques sont spécialisées dans la création d'objets utilisées uniquement comme bouchons pendant les tests unitaires, qu'on appelle des « mocks ». En une bibliothèque très utilisée est Mockito. Un exemple vous est fourni dans src/test/java/fr/ensimag/deca/context/TestPlusPlain.java. L'instanciation du nœud Plus se fait maintenant comme ceci:

```
AbstractExpr left = Mockito.mock(AbstractExpr.class);
when(left.verifyExpr(compiler, null, null)).thenReturn(INT);
AbstractExpr right = Mockito.mock(AbstractExpr.class);
when(right.verifyExpr(compiler, null, null)).thenReturn(INT);
Plus t = new Plus(left, right);
```

Tout ce qu'on sait, et tout ce qu'on a besoin de savoir sur les fils gauches et droits de notre nœuds, c'est qu'ils dérivent de AbstractExpr, et que leur méthodes verifyExpr renvoient INT.

On peut aller plus loin, par exemple pour factoriser la création de mocks pour plusieurs tests unitaires, voir par exemple src/test/java/fr/ensimag/deca/context/TestPlusAdvanced.java, et bien sûr le site de Mockito: https://code.google.com/p/mockito/.

1.11 Mesure de couverture

Il existe des outils permettant de mesurer le taux de couverture d'un programme par un jeu de tests (Jacoco par exemple, cf. [Jacoco]).

2 Gestion des risques et gestion des rendus

Un être humain étant nécessairement imparfait, on ne peut pas chercher à faire réellement un projet sans erreurs. Par contre, il est important de distinguer les conséquences potentielles d'une erreur dans la gestion du projet, la manière de programmer et de tester.

Par exemple, une erreur d'interprétation subtile de la grammaire attribuée ne sera visible que sur quelques programmes Deca, alors qu'un point-virgule manquant dans un fichier source rendra votre compilateur non-compilable, et par conséquent inutilisable. Sur cet exemple, l'erreur subtile serait difficile à corriger, mais avec un risque faible. Le point-virgule manquant est trivial à éviter et à corriger, mais le risque sur la qualité du projet est énorme (en gestion de projet, on parle parfois d'« analyse coût-bénéfice »).

En parcourant la partie II et les documents donnant des consignes de la partie I (typiquement, [A-Rendre]...) de la documentation, vous pouvez dresser une liste des dangers les plus critiques. Par exemple, le document [Introduction] mentionne des dates de rendus, un danger évident étant d'oublier une de ces dates. Le document [ExempleSansObjet] donne des exemples simples sur lesquels il faudrait *vraiment* que le compilateur fonctionne correctement, le danger étant simplement de perdre sa crédibilité auprès du client (ou de l'enseignant dans notre cadre)... Accordez une importance toute particulière au document [Decac].

Pour chacun de ces dangers, il faudra décider d'une ou plusieurs actions pour limiter ou éviter ce danger (dans l'exemple un peu simpliste des dates de rendus, « utiliser un agenda » peut être une réponse). Par « action », on entend quelque chose de concret et facilement vérifiable (on doit pouvoir répondre à la question « cette action a-t-elle été faite? » par oui ou par non). Par exemple « travailler sérieusement » est un principe général, mais pas une action.

Pour vous aider avec les bugs les plus simples mais les plus dévastateurs dans une base de tests automatisée, l'équipe enseignante vous fournit un script common-tests.sh (du répertoire src/test/script), activé par défaut dans « mvn test ». Faites en sorte d'être absolument certain d'exécuter ce script avant les rendus.

Nous vous demandons de consacrer une section de la documentation de validation à cette analyse des risques (c'est un sous-ensemble de ce qu'on appelle le « plan de gestion des risques » en gestion de projet). Cette section doit être écrite et discutée en suivi avant le rendu intermédiaire.

Un certain nombre d'actions ne peuvent pas être totalement automatisées (un des dangers délicats à traiter est la présence de bug dans votre infrastructure de tests!). Ces actions devront donc être réalisées occasionnellement au cours du projet, et surtout juste avant les rendus. Par exemple, vous testez probablement régulièrement le contenu de votre clone Git, mais avez-vous la *certitude* que le contenu du dépôt partagé marche également (ce n'est pas le cas par exemple si vous avez oublié un git add sur un fichier)?

Même avec une bonne base de tests, il faudra donc réaliser un certain nombre d'actions avant chaque rendu (c'est ce qu'on appelle la « gestion des mises en production », ou plus couramment en anglais « release management process »). Cette procédure doit être documentée dans votre documentation de validation, sous la forme d'une checklist d'actions à effectuer (avec pour chaque action, une explication rapide de l'utilité de l'action). Comme pour la gestion des risques, cette partie de la documentation doit être écrite et discutée en suivi avant le premier rendu.

3 Consignes à respecter impérativement

La qualité de la base de tests formée des fichiers deca (dans le répertoire src/test/deca/ du dépôt) sera évaluée sur le compilateur de référence des enseignants, essentiellement de manière automatique. Le résultat de cette évaluation aura un impact déterminant sur la perception qu'aura l'enseignant de votre démarche de validation (et donc sur la note finale au projet). Il est pour cela essentiel que votre base de tests respecte la classification utilisée par l'évaluateur automatique des enseignants, qui est décrite ci-dessous.

3.1 Classification des tests ".deca" (i.e. répertoires où placer ces tests)

D'une façon générale, les tests ".deca" afférents à chaque étape se trouvent dans des sous-répertoires de ~/Projet_GL/src/test/deca/nom d'étape>. Les seuls caractères autorisés dans les noms de répertoires et de fichiers de tests sont : [a-zA-Z0-9._/+-].

Plus précisément, dans chaque tel répertoire **src/test/deca/*/**, on trouve au moins deux sous-répertoires :

src/test/deca/nom d'étape/invalid/ contient des fichiers .deca ne contenant pas des programmes Deca valides.

- En étape A, cela veut dire que le fichier .deca contient des erreurs lexicales ou syntaxiques (c'est-à-dire, un test pour lequel « decac -p » doit renvoyer une erreur).
- En étape B, cela veut dire que le fichier .deca contient un programme Deca dont la syntaxe hors-contexte est correcte, mais qui viole des règles de la grammaire attribuée, pour lesquels le compilateur doit donc signaler des erreurs.
- En étape C, cela veut dire que le fichier .deca contient un programme qui respecte la grammaire, mais dont l'exécution va provoquer des erreurs (par exemple, une division par 0), qui doivent être signalées.

src/test/deca/nom d'étape/valid/ contient des fichiers .deca contenant des programmes Deca valides (i.e. ne levant pas d'erreur) pour l'étape courante. Par exemple, les tests lexicalement corrects mais syntaxiquement incorrects peuvent se trouver dans le répertoire src/test/deca/syntax/invalid/ mais pas dans src/test/deca/syntax/valid/.

En étape C, le répertoire ~/Projet_GL/src/test/deca/codegen/ contient, outre invalid et valid, deux autres sous-répertoires :

src/test/deca/codegen/interactive/ contient tous les fichiers .deca contenant des programmes valides faisant appel aux fonctions readInt et readFloat.

src/test/deca/codegen/perf/ doit être utilisé pour des tests de performances, une fois que le compilateur est valide. Il ne doit contenir que des programmes valides, dont on pourra évaluer la performance au moyen de l'option "-s" de l'interpréteur ima.

En effet, pour tester les fonctions de lecture readInt et readFloat, on ne peut simplement lancer l'exécutable, il faut ensuite fournir en entrée des valeurs.

Pour que nous puissions évaluer automatiquement vos tests :

- vous devez réserver le répertoire interactive/ aux tests faisant des lectures
- vous devez au maximum utiliser des tests qui n'ont pas besoin de faire de lecture, et que vous placerez donc dans valid/ et invalid/.

En fait, la plupart des aspects du langage peuvent être testés sans lecture, en utilisant des valeurs fournies dans le programme. Seul le test des fonctions de lecture elles-mêmes, readInt et readFloat, nécessite un traitement particulier. En outre, cela vous évite d'avoir à implémenter en étape C les méthodes de lecture dès le début.

Vous pouvez créer des sous-répertoires de valid, invalid, interactive et perf si vous le souhaitez : les fichiers de tests dans ces sous-répertoires doivent juste respecter la classification donnée ci-dessus (ils seront pris en compte par l'évaluateur automatique des enseignants).

Précisons que les fichiers présents dans les répertoires décrits ci-dessus doivent correspondre à des tests en *boîte noire* : leur classification doit être valable sur n'importe quelle implémentation correcte. Vous appliquerez cette remarque aussi dans le cadre de l'extension TRIGO : les tests de la classe Math utilisent la spécification donnée et leur classification doit être valable sur n'importe quelle classe Math (pas seulement la vôtre).

Pour utiliser des tests en *boîte blanche* (spécifiques à votre implémentation) ou qui ne rentrent pas dans les catégories précédentes (par exemple, les programmes Deca avec des variables non-initialisées), il est conseillé de créer d'autres sous-répertoires au même niveau que valid, etc. Ils ne seront pas pris en compte par l'évaluateur automatique des enseignants. Par exemple, pour tester les extensions de la [BibliothequeStandard], il est conseillé d'introduire deux sous-répertoires dans gencode : extension/valid et extension/invalid.

3.2 Commentaires imposés

Il est demandé de commenter les tests (en début de fichier), à l'aide de commentaires Deca commençant en première colonne. Ces commentaires résument les caractéristiques du test, dans un format commun à l'ensemble des tests.

Par exemple, pour le programme de test de la conversion entier-flottant du paragraphe 1.2 :

```
// Description :
// test de la conversion entier -> flottant
//
// Resultats :
// ok
```

En ce qui concerne les étapes A et B, on recommande d'avoir pour tous les tests invalides une en-tête de fichier contenant :

```
// Resultats :
// Erreur contextuelle
// Ligne <No_de_ligne> : <message d'erreur de votre compilateur>
```

Cela peut vous permettre de vérifier que votre compilateur produit bien une erreur à la ligne indiquée. Il est par ailleurs fortement recommandé que le message d'erreur de votre compilateur fasse référence au numéro de la règle (ou des règles) de la grammaire attribuée qui est impliquée.

3.3 Documentation de validation

Vous rendrez en fin du projet (cf. [A-Rendre]) une documentation sur la validation de votre compilateur, qui comprendra au moins :

- Descriptif des tests
 - types de tests pour chaque étape/passe (tests unitaires, tests système, ...)
 - organisation des tests
 - objectifs des tests, comment ces objectifs ont été atteints.
- Les scripts de tests
 - comment faire passer tous les tests
- Gestion des risques et gestion des rendus (cf. section 2)
- Résultats de Jacoco
- Méthodes de validation utilisées autres que le test

Ne pas inclure l'ensemble des jeux de tests dans la documentation. Ils figurent déjà dans le dépôt Git.

Une bonne documentation de validation doit être écrite pour permettre à quelqu'un de reproduire la validation (i.e. de lancer la base de tests et d'interpréter les résultats) et de la continuer (i.e. d'identifier les faiblesses de la couverture des tests, ajouter de nouveaux tests, ...). Attention à ne pas la rédiger comme un journal de bord!

Utilisation de Jacoco pour la mesure de couverture

Jacoco est un outil qui permet de calculer la couverture d'un jeu de tests sur un programme. On peut savoir quelle instruction a été exécutée (ou non), quelles branches d'une instruction conditionnelle (if, while) ont été prises, estimer la complexité du code etc.. Nous allons utiliser Jacoco via le plugin Maven, ce qui nous évite d'avoir à installer l'outil explicitement et nous donne une intégration dans le cycle de vie du programme.

1 Découverte et utilisation manuelle de Jacoco

Le principe de Jacoco est d'instrumenter le code binaire des classes Java afin de mesurer les différentes traces d'exécution. Cette instrumentation peut être réalisée à la volée lors du chargement des classes par la machine virtuelle. Cela repose sur la notion d'agent. Un agent java est une classe java s'appuyant sur l'API d'instrumentation Java et ayant la capacité d'inspecter et de modifier le code binaire chargé au sein d'une JVM. Jacoco propose son propre agent java pour mesurer le couverture du code

L'instrumentation peut également être réalisée hors ligne par une phase explicte d'instrumentation des fichiers binaires *.class. C'est ce mode que nous utiliserons dans ce projet.

Soit le programme Java « hello » suivant (une version un peu plus longue se trouve dans le répertoire examples/tools/ de votre projet) :

```
public static void main(String[] args) {
    SayHello sayHello;
    if (args.length == 0) {
        sayHello = new SayHello();
    } else {
        sayHello = new DireBonjour();
    }
    sayHello.sayIt();
}
```

On compile ce programme (mvn compile) puis on instrumente les fichiers .class générés : mvn jacoco:instrument. Cette commande va générer de nouveaux fichiers .class dans le répertoire ./target/generated-classes/jacoco/. Ces classes sont ensuite permutées avec les classes initiales (réperpertoire ./target/classes). Quand on exécutera l'application, les classes instrumentées enregistreront les instructions exécutées dans un fichier ./target/jacoco.exec (illisible par un être humain), et l'outil Jacoco pourra générer un rapport lisible (au formats HTML, XML et CSV).

Lancez l'exécution du programme à l'aide du script run.sh. On peut le lancer sans argument, il va exécuter la branche « then » de notre if. Le test est bien entendu incomplet, on va le vérifier avec Jacoco.

Avant de pouvoir générer le rapport, il faut désintrumenter les classes et les restaurer dans leur état initial. Cela se fait par la commande mvn jacoco:restore-instrumented-classes.

```
mvn jacoco:report
firefox target/site/jacoco/index.html
```

Ouvrir la page de rapport de couverture de la classe Main, et vérifiez que la branche else n'est pas couverte (elle apparaît en rouge). Pour les lignes couvertes, elles sont marquées en vert. Pour les instructions conditionnelles, Jacoco donne en plus la couverture des branches : un if n'est considéré couvert que si les deux branches (then et else) ont été prises au moins une fois. Une seule branche étant couverte, le test apparaît en jaune, indiquant que la couverture est partielle.

Relancez run. sh avec un argument (en ayant pris soin de réinstrumenter le code avant). L'exécution produit un résultat différent, et si on régénère le rapport de couverture, on peut voir que la classe est maintenant totalement couverte.

2 Couverture pour les tests automatisés

L'intérêt principal de la mesure de couverture est d'évaluer la couverture d'une base de tests automatisée. Les différentes étapes précédentes sont intégrées dans le fichier pom.xml qui vous est fourni. Leur enchaînement s'obtient en invoquant la phase verify. La phase verify par ses dépendance va déclencher si besoin la compitation (compile), l'instrumentation Jacoco puis les tests unitaires (test), les tests d'intégration, la désintrumentation et la génération du rapport (verify).

Attention: lancer mvn test à la place de mvn verify va bien permettre d'exécuter les tests unitaires mais ne produira pas de rapport et va laisser les classes instrumentées au lieu des classes originales. Un nouveau mvn test provoquera alors une erreur lorsque Jacoco tentera d'instrumenter des classes qui le sont déjà. Pour repartir sur une base propre, il faut alors d'abord executer un mvn clean. Il est donc préférable d'exécuter systématiquement un mvn verify en phase de tests.

Dans notre exemple, on peut donc lancer :

```
mvn clean
mvn verify
firefox target/site/jacoco/index.html
```

On peut voir que les tests unitaires ne testent pas du tout la classe Main, qu'ils ont une couverture raisonnable sur les classes DireBonjour et SayHello, mais que la méthode DireBonjour.getOtherMessage() n'est pas couverte. On peut corriger ceci en ajoutant un test. Par exemple, il y a un test qu'il suffit de décommenter dans ./src/test/java/tools/DireBonjourTest.java. Relancer mvn verify et vérifier que la couverture a augmenté.

3 Méthode

Jacoco est un très bon outil pour compléter une base de tests existante. La commande mvn verify permet d'identifier les « trous » dans la couverture. Comme cette commande lance tous les tests, elle prend souvent assez longtemps à s'exécuter. On peut donc ajouter des tests pour combler chaque trou, lancer les nouveaux tests à la main (en oubliant d'instrumenter les classes au préalable) pour vérifier rapidement que les nouveaux tests sont bien efficaces. Une fois les nouveaux tests lancés, désintrumentez le code et générez le rapport. Une fois les trous comblés, on relance mvn clean verify pour vérifier que les tests sont bien lancés par la base de tests automatisée.

Il est possible de s'imposer un objectif minimal en terme de couverture pour un build réussi de votre projet. Cela correspond à la section check du plugin Jacoco du fichier pom.xml. Dans l'exemple fournit, une couverture minimale de 20% sur les instructions est nécessaire (ce n'est pas vraiment pas exigeant ...). Ces contraintes peuvent être sur d'autres critères. Se référer à la documentation.

4 Remarque

Le fichier pom.xml fournit à la racine de votre projet ne calcule pas par défaut la couverture des tests. Ce comportement est contrôlé par la propriété jacoco.skip dont la valeur est fixée à true en tête du fichier. La commande mvn verify ne produira donc aucune couverture. Pour la calculer, il faut

fixer la valeur de la proriété à false. Ceci peut être réalisé directement depuis la ligne de commande : mvn -Djacoco.skip=false verify.

[ProgrammationDefensive]

Programmation défensive

1 Introduction

Une méthode peut en général être spécifiée avec des préconditions (i.e. des conditions que les entrées de la fonction doivent satisfaire), et des post-conditions (i.e. conditions que les sorties de la fonction doivent satisfaire). Par exemple, une fonction « racine carré » peut être spécifiée comme ceci :

```
/**
 * precondition : x >= 0
 * post-condition : sqrt(x) * sqrt(x) = x
 * (plus or minus the rounding errors)
 */
float sqrt(float x)
```

On parle parfois aussi de « contrat » de la fonction avec des suppositions (assume) et des garanties (guarantees).

Une violation de cette spécification (pré/post-conditions) peut venir d'au moins deux problèmes :

- La fonction est appelée avec des entrées incorrectes (par exemple, l'utilisateur a écrit sqrt(-1)). Si c'est le cas, le problème se trouve du côté de l'appelant, et il faut lui signaler. Ce cas se produit quand la pré-condition (assume) est violée.
- La fonction contient une erreur. Si c'est le cas, on souhaite en général voir la condition violée pour pouvoir corriger le bug (ou signaler le bug au développeur quand on vient de trouver un bug dans une méthode dont on n'est pas l'auteur). Ici, c'est la post-condition (guarantee) qui est violée.

Faire de la programmation défensive, c'est vérifier explicitement ces pré et post-conditions : même si on sait que ces conditions devraient être satisfaites, on le vérifie tout de même pour trouver les erreurs rapidement.

Il existe différentes techniques pour faire de la programmation défensive en Java. Dans tous les cas, on interrompt le programme en levant une exception. L'examen de la pile d'exécution permet de retrouver facilement l'appel de l'opération qui n'a pas respecté la précondition.

Écrire du code défensif prend un peu plus de temps au départ, mais l'expérience montre qu'on gagne beaucoup de temps sur la mise-au-point. On peut citer deux raisons à cela :

- 1. Avec le « contrat » écrit dans le code, le programmeur réfléchit davantage sur la spécification et sur la cohérence entre l'implémentation et la spécification (en particulier, lorsque l'implémentation évolue).
- 2. En cas de bogue, l'exception levée aide à diagnostiquer l'origine du bogue : viol de précondition ou viol de postcondition, à telle ligne, etc.

2 Test de précondition

Les opérations ont un domaine de définition : elles ne sont pas définies pour certaines valeurs. On dit que ces opérations comportent des préconditions. Cela peut se produire :

- lorsque la fonction mathématique correspondante est partielle (exemple : la division par zéro n'est pas définie);
- lorsque l'algorithme utilisé présuppose des conditions sur les paramètres (exemple : recherche dichotomique dans un tableau trié).

Lorsque la précondition n'est pas respectée, le comportement de l'opération est indéfini.

2.1 Déclenchement explicite d'une exception

En Java, il est recommandé de lever l'exception IllegalArgumentException quand un argument d'une méthode (surtout pour une méthode publique) ou d'un constructeur ne vérifie pas une précondition.

On peut tester explicitement la précondition et lever une exception spécifique. Exemple :

```
/**
 * precondition : x >= 0
 */
float sqrt(float x) {
   if (x < 0) {
      throw new IllegalArgumentException("x should be positive");
   }
   // ...
}</pre>
```

Pour signaler un problème de cohérence interne du compilateur on pourra utiliser l'exception DecacInternalError (ou l'exception AssertionError).

2.2 Utilisation de la classe Validate d'apache commons

Une classe très simple mais bien pratique permet d'avoir un peu moins de code à écrire pour faire la même chose : org.apache.commons.lang.Validate. Le code ci-dessus devient :

```
import org.apache.commons.lang.Validate;
...
/**
 * precondition : x >= 0
 */
float sqrt(float x) {
    Validate.isTrue(x >= 0, "x should be positive");
    // ...
}
```

La classe Validate fournit (entre autres) des méthodes isTrue, isFalse, notNull, notEmpty. Chacune de ces méthodes a une variante à un argument (sur lequel la vérification est faite), et une avec un deuxième argument de type String, qui est un message à donner à l'utilisateur si la validation échoue.

3 Vérification des post-conditions et des invariants

La vérification des post-conditions, et éventuellement des invariants (par exemple, invariant de boucle) est un peu différente. Si on a bien testé, ces vérifications ne devraient jamais être violées par un utilisateur, et deviennent inutile dans la version finale du produit. Si jamais une de ces conditions était violée, il ne faudrait pas reprocher à l'utilisateur d'avoir mal utilisé notre code.

On utilise donc un mécanisme différent. Les assertions Java sont bien adaptées ici:

- Les assertions sont désactivables (en fait, elles sont même désactivées par défaut), donc si les vérifications sont trop coûteuses, on peut s'en passer dans la version finale.
- Une violation d'assertion lève l'exception AssertionError, qui dérive de Error, ce qui rend explicite le fait qu'elle ne doit pas être rattrapée.

Pour utiliser une assertion en Java, on écrit simplement :

```
assert (condition);
```

ou bien:

```
assert condition;
```

Les assertions sont activées par l'option -ea ou -enableassertions au lancement de java (pas à la compilation, contrairement au C). On peut activer les assertions seulement pour certaines classes si nécessaire.

Si les assertions sont désactivées, la condition n'est pas évaluée. Il est donc important de ne *jamais* avoir d'effet de bord dans une assertion :

Pour plus d'information, lire la page « Programming With Assertions » disponible à l'url http://docs.oracle.com/javase/1.4.2/docs/guide/lang/assert.html.

4 Remarque sur la récupération des exceptions

Les exceptions qui sont levées lorsqu'une condition (pré ou post) n'est pas respectée ne sont pas destinées (au moins dans la phase de mise au point) à être récupérées. En effet, si on récupère une telle exception, on ne peut plus savoir quelle procédure l'a levée. On s'interdira, a fortiori, de récupérer toutes les exceptions avec une clause try ... catch(Exception e) { dans un moniteur d'exception.

```
try {
    ...
} catch (InternalError e) { // interdit
    System.out.println("Une erreur interne s'est produite");
} catch (Exception e) { // interdit aussi
    System.out.println("Une erreur inexpliquee s'est produite");
}
```

Par contre, certaines procédures spécifient qu'elles lèvent des exceptions dans des cas précis. Ces exceptions sont destinées à être récupérées, puisqu'il s'agit d'un fonctionnement normal du programme.

Par exemple, la méthode Java File.createNewFile() peut lever l'exception IOException si quelque chose empêche le fichier d'être créé. On peut l'utiliser de la manière suivante :

```
try {
    boolean createNewFile = new File("toto").createNewFile();
} catch (IOException ex) {
    // Faire quelque chose pour recuperer l'erreur
    // ce n'est pas un bug du programme d'arriver ici,
    // seulement un cas particulier a traiter.
}
```

Remarque : en Java, cette distinction correspond aux exceptions non contrôlées (celles qui dérivent de RuntimeException, et qui ne devraient pas être levées) et aux exceptions contrôlées (qui font partie de la spécification de la méthode). Chaque méthode doit déclarer les exceptions contrôlées qu'elle peut lever dans la clause throws de la signature de la méthode.

5 Coût de la programmation défensive

Effectuer de la programmation défensive peut modifier la complexité d'un programme. Par exemple, si on teste que le tableau est trié avant d'effectuer une recherche dichotomique, la complexité de la procédure de recherche devient linéaire.

Dans ce cas, une fois la phase de mise au point achevée, on souhaite supprimer les tests défensifs. Pour les assertions, nous avons déjà vu comment les désactiver. Dans le cas des tests de préconditions, on ne les supprime que si le coût n'est pas négligeable. Pour ne pas devoir modifier le code, on peut utiliser une constante booléenne DEFENSIVE qui indique si on veut faire de la programmation défensive ou non. Dans la phase de mise au point, DEFENSIVE est vrai, dans la phase finale DEFENSIVE est faux.

Dans l'exemple ci-dessus, la procédure isSorted() est appelée uniquement lorsque DEFENSIVE est vrai.

```
private static final boolean DEFENSIVE = true;

void m(MyTable t) {
    if (DEFENSIVE) {
        Validate.isTrue(isSorted(t));
    }
    // ...
}
```

$[{f Conventions Codage}]$

Conventions et Styles de Codage

1 Introduction

La qualité d'un logiciel n'est pas limitée à la partie visible de l'utilisateur. Pour écrire un logiciel maintenable, il est nécessaire de se fixer un certain nombre de règles pour que le code soit de bonne qualité. Certaines règles sont des bonnes pratiques largement reconnues pour obtenir un code lisible (par exemple, trouver des noms expressifs pour les fonctions), d'autres sont plutôt des conventions (par exemple, de combien doit-on indenter l'intérieur d'un bloc entre accolades?). Si certaines conventions peuvent paraître arbitraires, il est tout de même important de les respecter pour obtenir un code homogène, surtout quand on travaille à plusieurs.

Ce document vous propose quelques conseils et consignes sur la manière d'écrire votre code. Il ne cherche pas l'exhaustivité, à vous d'ajouter vos propres conventions pour le compléter!

2 Types abstraits de données

On définit un « objet abstrait » comme l'encapsulation d'un état (un ensemble d'informations, qui peuvent être elles-mêmes des objets abstraits). L'état est encapsulé au sens ou il n'est manipulable qu'à travers certaines opérations associées à l'objet. Un « type abstrait de données » (TAD) est le type d'un ensemble d'objets abstraits; sa définition fournit l'ensemble des opérations disponibles sur les objets en masquant leur représentation ainsi que l'implantation des opérations.

On classifie les opérations disponibles en plusieurs catégories :

Les constructeurs permettent de créer de nouveaux objets;

Les mutateurs permettent de modifier un objet existant;

Les sélecteurs permettent d'accéder à une partie de l'objet;

Les testeurs permettent d'interroger la structure d'un objet (l'objet a-t-il une certaine propriété?)

Le langage Java fournit un certain nombre de fonctionnalités pour gérer des types abstraits de données. Les notions de classes et de paquetages permettent de grouper les données et méthodes, et les règles de visibilités (public, private, ...) permettent de décider qui aura accès à quoi.

On implémentera un type abstrait de données de la manière suivante :

- Le type abstrait est défini par une classe Java
- Les données contenues dans chaque objet sont des champs privés de la classe. On s'interdira d'avoir des champs publics dans une classe (sauf éventuellement pour des constantes, i.e. des champs déclarés avec le modificateur final).
- Les sélecteurs sont des méthodes dont le nom commence par get, sans argument, et qui renvoient la donnée recherchée. Dans le cas d'un sélecteur renvoyant une valeur booléenne, ou pour les testeurs, le nom de la méthode peut commencer par is au lieu de get.
- Les mutateurs sont des méthodes ne renvoyant rien, prenant généralement un argument, et dont le nom commence par set.

```
public class TimeOfDav {
   private int hour;
    private int minute;
   public int getHour() {
       return hour;
    public void setHour(int hour) {
        if (hour < 0 || hour >= 24) {
            throw new IllegalArgumentException("Bad value for hour");
        this.hour = hour;
   public int getMinute() {
        return minute;
    public void setMinute(int minute) {
        if (minute < 0 || minute >= 60) {
            throw new IllegalArgumentException("Bad value for minute");
        this.minute = minute:
    }
    public TimeOfDay(int hour, int minute) {
        if (hour < 0 || hour >= 24) {
            throw new IllegalArgumentException("Bad value for hour");
        if (minute < 0 || minute >= 60) {
            throw new IllegalArgumentException("Bad value for minute");
        this.hour = hour;
        this.minute = minute;
    }
7
```

FIGURE 1 – Implémentation possible d'une classe TimeOfDay à deux champs

Par exemple, le type « heure du jour » pourrait s'écrire comme dans la figure 1.

Un avantage immédiat d'encapsuler les données (i.e. d'avoir des champs privés et des accesseurs/mutateurs pour y accéder) est qu'on peut facilement garantir des propriétés sur les données représentées. Ici, les mutateurs et le constructeur vérifient que les heures et les minutes ont bien une valeur raisonnable. Un autre intérêt est que cette implémentation permet de changer la représentation des données sans changer l'interface publique. Par exemple, si on doit manipuler un très grand nombre d'objets du type TimeOfDay, il peut être judicieux de stocker sa valeur sur un seul entier (par exemple, stocker le nombre total de minutes depuis le début de la journée), et on ne souhaite surtout pas avoir à modifier tous les utilisateurs de la classe pour le faire : il suffit de modifier les accesseurs getMinute et getHour et les mutateurs setMinute et setHour, et le constructeur, pour leur faire faire les divisions et modulo appropriés.

Il n'est pas toujours souhaitable d'écrire des accesseurs et des mutateurs pour chaque champ d'une classe. Par exemple, on peut souhaiter avoir des objets immutables (pour éviter une modification accidentelle par exemple), et donc interdire les modifications des champs en ne fournissant pas de mutateurs. On peut aussi décider de faire faire toutes les manipulations de données directement à la classe elle-même, et du coup se passer de sélecteurs. Par exemple, si les seules opérations qu'on souhaite faire avec TimeOfDay sont l'afficher à l'écran, et calculer une durée en faisant une différence de deux objets, on pourra ajouter une méthode display() et une méthode computeDuration(TimeOfDay otherDate), et se passer de getHour et getMinute.

3 Règles de mise en forme du code

On respectera autant que possible les règles de codage de Java : voir le document « Code Conventions for the Java Programming Language » disponible à l'adresse : http://www.oracle.com/technetwork/java/codeconv-138413.html

En particulier:

- La largeur d'indentation est de 4 caractères. Il est fortement recommandé d'indenter uniquement avec des espaces, mais il est aussi possible d'utiliser des tabulations. Dans ce cas, une tabulation doit impérativement représenter 8 espaces. Attention, ce n'est pas le cas par défaut sous Eclipse, mais on peut obtenir un réglage correct via « Window -> Preferences -> General -> Editors -> Text Editors -> champ 'Displayed tab width :' » (qui vaut 4 par défaut, à passer à 8). NetBeans indente avec des espaces uniquement et a la bonne largeur de tabulation par défaut.
- Un nom de classe commence toujours par une majuscule, un nom de champ ou de méthode commence toujours par une minuscule. Par exemple :

```
class MaClasse {
   int monChamp;
   void maMethode();
}
```

- Les constantes de classes (static final) ont un nom entièrement en majuscules, les mots sont séparés par des tirets bas (_). Par exemple : public static final MA_CONSTANTE = 42;.
- Le placement des espaces est illustré par l'exemple suivant :

— On utilise toujours des accolades avec les constructions if, while et for, même si elles ne sont pas obligatoires.

Le squelette de code qui vous est fourni est en anglais (noms de variables et commentaires), à une exception près : les commentaires vous donnant des instructions sont en français (ils commencent par // A FAIRE:); ils devraient disparaître assez rapidement. Vous êtes encouragés à écrire votre code en anglais également. Pour un vrai projet, c'est important car on ne sait jamais ce que le code qu'on écrit aujourd'hui va devenir (l'entreprise qui le développe va-t-elle embaucher un non-francophone? Va-t-elle sous-traiter à une société de service étrangère? Va-t-elle se faire racheter par une entreprise étrangère? ...), et il est très difficile de traduire du code source après coup (traduire des noms de classes et méthodes publiques revient à casser l'API).

La question de la langue à utiliser dans les messages d'erreurs est différente, puisqu'elle impacte l'utilisateur (et non seulement les développeurs). Un vrai compilateur utiliserait une bibliothèque pour s'adapter à la langue de l'utilisateur (comme ResourceBundle ou GNU Gettext). Pour le projet, vous pouvez écrire les messages en français ou en anglais.

4 Fonctionnalités du langage Java

4.1 Itération sur une collection

Depuis Java 1.5, on peut parcourir une collection avec une boucle for (cette version est parfois appelée « for each ») :

```
for (TypeElement elem : collection) {
    // Code utilisant 'elem'
}
```

C'est équivalent, mais beaucoup plus lisible que :

```
for (Iterator < Type Element > i = collection.iterator(); i.hasNext(); ) {
    Type Element elem = i.next();
    // Code utilisant 'elem'
}
```

On préférera la première écriture dans ce projet.

4.2 Transtypage (cast)

L'utilisation du transtypage (((Type) expression)) est à éviter au maximum pour deux raisons. D'une part, un transtypage peut échouer (typiquement, transtyper vers une classe dérivée ne marche que si le type dynamique de l'expression est le bon), donc cette écriture pose la question de la gestion de l'exception ClassCastException.

Mais surtout, l'utilisation du transtypage est en général signe d'un code orienté objet mal conçu. En général, il est préférable d'utiliser à la place le polymorphisme ou la généricité.

Par exemple:

Le code à l'intérieur du **if** est spécifique à la classe **Voiture**, donc il aurait été préférable de placer ce code dans la classe **Voiture** (qui aurait utilisé **this** et qui aurait été du bon type sans transtypage). Sur cet exemple, le polymorphisme aurait donc été préférable.

Un autre exemple 1 :

```
ListeDObjets liste = ...;
Object o = liste.get(...);
MaClasse c = (MaClasse) o;
```

Le problème ici est qu'on a une collection mal typée : elle peut contenir n'importe quel objet Java. Si on se trompe en ajoutant un élément, on ne s'en rendra compte qu'à l'exécution. En utilisant la généricité, on peut écrire :

```
List < MaClasse > liste = ...;
MaClasse c = liste.get(...);
```

5 Affichages, traces, et debug

Il est parfois pratique pendant le développement et le débugage d'écrire ce que fait le programme à l'écran pendant son exécution. On parle de « traces », ou « traces de débug ».

^{1.} Qui était malheureusement la manière standard d'utiliser les collections avant Java 1.5

Un programmeur maladroit pourrait parsemer son code de System.out.println(...), mais c'est une très mauvaise idée, car il est trop facile d'oublier de supprimer ces traces. Dans le cas d'un compilateur dont la sortie représente les erreurs de compilation, c'est particulièrement grave, car l'utilisateur verrait les affichages de débug mélangés avec ses erreurs. Par ailleurs, si on utilise beaucoup les traces de debug, on risque aussi d'avoir une très grande quantité d'affichages, et il devient alors nécessaire de pouvoir contrôler finement quels affichages on souhaite voir.

On pourrait créer notre propre système d'affichage de traces, par exemple :

```
class TraceDebug {
   private static final int LEVEL = 5;
   public static void trace(int level, String message) {
      if (level <= LEVEL) {
         System.out.println("trace: " + message);
      }
   }
}</pre>
```

On pourrait alors utiliser TraceDebug.trace(1, "mon message"); au lieu de System.out. On pourrait contrôler la quantité d'affichages en modifiant la valeur de la constante LEVEL (0 pour n'avoir aucun affichage) et en recompilant.

En fait, il existe déjà plusieurs systèmes pour faire ce genre de choses. La bibliothèque standard de Java fournit un paquetage java.util.logging (depuis Java 1.4), et nous allons utiliser la bibliothèque log4j, très utilisée et un peu plus flexible que la bibliothèque standard.

Pour utiliser log4j, il faut dans un premier temps instancier un « logger » :

```
// Pour pouvoir utiliser log4j:
import org.apache.log4j.Logger;

public class LogClass {
    // Instantiation du logger, une fois par classe.
    private static final Logger LOG = Logger.getLogger(LogClass.class);
    // ...
}
```

Une fois le logger instancié, on peut l'utiliser dans les méthodes de la classes :

```
LOG.trace("Trace Message!");
LOG.debug("Debug Message!");
LOG.info("Info Message!");
LOG.warn("Warn Message!");
LOG.error("Error Message!");
LOG.fatal("Fatal Message!");
```

Chacune des méthodes trace, debug, info, warn, error et fatal peut produire un affichage de debug. On choisit un niveau à partir duquel les traces sont affichées. Par exemple, si le niveau est « info », on verra les affichages de info, warn, error et fatal, mais ceux des méthodes trace et debug sont désactivées.

Pour choisir le niveau d'affichage, on peut utiliser la méthode setLevel de chaque logger, ou le fichier de configuration log4j.properties. Un fichier d'exemple vous est fournis : src/main/resources/log4j.properties. Ce fichier contient beaucoup de commentaires, lisez-les pour comprendre ce que vous pouvez faire avec log4j. On peut aussi activer et désactiver les traces depuis Java. C'est ce que decac -d fait (le squelette de code de CompilerOptions.java vous met sur la voie sur la manière de l'implémenter correctement), et cela vous permet d'activer les traces pour une exécution, sans avoir à modifier les sources.

Pour plus de renseignements, lire la documentation de log4j : http://logging.apache.org/log4j/1.2/ ou l'un des nombreux didacticiels qu'on peut trouver sur Internet.

6 Représentation de l'arbre abstrait avec le patron « interprète »

L'implémentation de l'arbre abstrait est faite en utilisant le patron de conception « interprète » ² (« interpreter design pattern » en anglais). Le principe est de représenter la grammaire d'arbres (cf. section 2 de [SyntaxeAbstraite]) par une hiérarchie de classes obtenue très systématiquement à partir de la grammaire d'arbres. ³

6.1 Esquisse du lien entre la hiérarchie de classes et la grammaire d'arbres

La correspondance entre la hiérarchie de classes et la grammaire d'arbres est basée sur les idées suivantes :

- Un arbre est codé par un objet instance d'une classe abstraite appelée Tree. Les fils éventuels de cet arbre sont eux-mêmes des instances de Tree figurant dans des champs de cet objet.
- Un type de nœud (par exemple, le terminal <u>Plus</u>) est codé par une classe concrète dont les instances représentent des arbres ayant ce nœud comme racine.
- Un non-terminal (qui représente un ensemble d'arbres) correspond à une classe abstraite dont les instances représentent des arbres de cet ensemble.
- Une règle de réécriture donnée dans la grammaire d'arbres exprime l'inclusion de l'ensemble d'arbres en membre droit dans l'ensemble d'arbres en membre gauche. Elle correspond donc à définir la classe (abstraite) en membre gauche de la règle comme la super-classe de la classe en membre droit de la règle (qui est soit la classe abstraite du membre droit si celui-ci est réduit à un non-terminal, soit la classe concrète du terminal en première position du membre droit).
- Les listes d'arbres sont codées à partir d'une structure générique de listes.

Par exemple, la grammaire d'arbres autorise la suite de dérivations

$EXPR \rightarrow BINARY_EXPR \rightarrow OP_ARITH \rightarrow Plus[EXPR EXPR]$

La hiérarchie de classes comportera donc une classe AbstractExpr (qui hérite de Tree) pour représenter le non-terminal EXPR de la grammaire, dont hérite une classe AbstractBinaryExpr pour représenter BINARY_EXPR, dont hérite une classe AbstractOpArith pour représenter OP_ARITH, dont hérite une classe concrète Plus pour représenter le terminal Plus. Un objet de type Plus a deux champs contenant des objets de type AbstractExpr.

L'intérêt de ce patron est que le typage de Java garantit que les objets instances de Tree correspondent bien à des arbres valides de la grammaire d'arbres. Autrement dit, on utilise le typage de Java comme algorithme vérifiant que tout objet de type Tree correspond bien à un mot de la grammaire d'arbres. Plus généralement, le compilateur Java vérifie statiquement que l'implémentation du compilateur decac manipule bien les arbres de syntaxe abstraite de façon conforme à la grammaire d'arbres 4. On évite ainsi « par construction » des bogues relatifs à une manipulation incorrecte des arbres.

6.2 Construction de la hiérarchie de classes à partir de la grammaire d'arbres

On précise ici le principe de la correspondance exprimée ci-dessus, en explicitant en particulier les règles de nommage des classes en fonction des noms des symboles de la grammaire.

— Chaque non-terminal correspondant à un ensemble d'arbres de base est implémenté par une classe abstraite en Java. Le nom de la classe est obtenu en préfixant le non-terminal par Abstract, en supprimant les caractères "_", et en appliquant les règles de minuscule/majuscules des noms de classes en Java (CamelCase : une majuscule en début de chaque mot).

^{2.} http://en.wikipedia.org/wiki/Interpreter_pattern

^{3.} En fait, la grammaire d'arbres donnée en [SyntaxeAbstraite] est extraite automatiquement à partir des sources Java du « decac » enseignant, par un programme Java qui inspecte la hiérarchie de classes par réflexion et vérifie qu'elle correspond bien à une grammaire d'arbres.

^{4.} à condition que l'implémentation ne fasse que du « upcast »

- Chaque terminal de la grammaire en <u>CamelCase</u> (correspondant à un nom de nœud) est implémenté par une classe concrète de même nom en Java avec un unique constructeur. La liste des arguments du constructeur de cette classe correspond à la liste des éventuels attributs, suivis des non-terminaux en "argument" du nom de nœud (c'est-à-dire les fils du nœud). Chacun des arguments du constructeur correspond lui-même à un champ privé de la classe concrète (initialisé avec les arguments du constructeurs).
- Chaque règle de la grammaire d'arbres correspond à l'héritage de la super-classe (abstraite) en membre gauche de la règle par la classe (abstraite ou concrète) en membre droit. Les classes qui n'ont pas de super-classe avec cette règle (c'est-à-dire les classes abstraites qui correspondent à un non-terminal qui n'apparaît pas en tant que membre droit réduit à un unique non-terminal) ont la classe abstraite Tree comme super-classe.
- Chaque ensemble de listes d'arbres, appelé "LIST_A" (avec A non-terminal) dans [SyntaxeAbstraite], est implémenté par une classe concrète héritant de la classe TreeList (instanciée sur la classe abstraite correspondant au non-terminal A). Le nom de la classe suit les mêmes règles que pour le nom de la classe correspondant à A, sauf que le préfixe Abstract est remplacé par le préfixe List.

Par exemple, considérons la règle suivante de [SyntaxeAbstraite]:

```
INST \rightarrow \underline{While}[EXPR\ LIST\_INST]
```

Elle correspond à la définition de classe suivante (ici les noms des champs de la classe sont repris des noms d'attributs de la grammaire attribuée de [Decompilation]):

```
public class While extends AbstractInst {
    private final AbstractExpr cond;
    private final ListInst insts;

public While(AbstractExpr condition, ListInst insts) {
        this.cond = cond;
        this.insts = insts;
    }

....
}
```

6.3 Implémentation d'un parcours d'arbres à partir d'une grammaire attribuée

Chaque parcours d'arbres dans le patron interpréteur correspond à un ensemble spécifique de méthodes virtuelles pour lesquelles chaque classe concrète doit fournir une implémentation. Cette façon d'implémenter les parcours d'arbres peut être mise en correspondance avec l'écriture d'une grammaire attribuée. En effet, le profil de chaque méthode spécifique au parcours, déclarée abstraite au niveau d'une certaine classe abstraite, va correspondre au profil du non-terminal de cette classe abstraite dans la grammaire attribuée. Chaque implémentation de cette méthode spécifique (dans les classes qui dérivent de la classe abstraite) va correspondre à une règle de la grammaire attribuée.

Par exemple, la décompilation des arbres (correspondant grosso-modo à la grammaire attribuée de [Decompilation]) est implémentée par une unique méthode virtuelle déclarée abstraite au niveau de la classe abstraite Tree (car tous les non-terminaux ont le même profil dans cette grammaire attribuée) :

```
public abstract void decompile(IndentPrintStream s);
```

Ici, le profil de cette méthode de décompilation diffère de la spécification donnée en [Decompilation], puisqu'elle prend en entrée un flux d'impression s qu'elle modifie par effets de bord, plutôt que de retourner en sortie une chaîne de caractères. Pour coller à la spécification, on aurait pu choisir le

profil "String decompile()". Mais, pour des raisons d'efficacité (décompilation à coût linéaire), il est préférable de changer les types de données. Malgré ce changement de profil, l'implémentation de decompile peut quand même rester fidèle à la grammaire attribuée de [Decompilation], ⁵ puisque la règle

```
INST\uparrow r \rightarrow \text{While}[\text{EXPR}\uparrow cond \text{LIST\_INST}\uparrow inst_s] \{ r := \text{'while}(\text{'}.cond.') \{\text{'}.inst_s.'\}' \}
```

correspond à implémenter decompile dans la classe concrète While par le code suivant (on omet ici les détails de gestion d'indentation que vous trouverez dans le code fourni) :

```
@Override
public void decompile(IndentPrintStream s) {
    s.print("while(");
    cond.decompile(s);
    s.println("){");
    insts.decompile(s);
    s.print("}");
}
```

Dans le cas d'un parcours correspondant à une grammaire attribuée, comme celles de [Syntaxe-Contextuelle], où les non-terminaux n'ont pas tous le même profil (cf. section 8), chaque classe abstraite a potentiellement besoin de déclarer une méthode virtuelle abstraite dont le profil correspond à celui du non-terminal associé à cette classe. Dans ce cas, une classe donnée peut hériter de plusieurs méthodes spécifiques au parcours avec plusieurs profils différents : pour les distinguer, il suffit par exemple de suffixer le nom de chaque méthode spécifique du parcours par le suffixe de la classe abstraite où la méthode est déclarée abstraite. Par exemple, pour la passe 3 des vérifications contextuelles, on utilise plusieurs méthodes, dont verifyExpr qui renvoie un Type (déclarée abstraite dans AbstractExpr puis implémentée dans toutes les classes qui en dérivent) et verifyInst qui ne renvoie rien (déclarée abstraite dans AbstractInst puis implémentée dans toutes les classes qui en dérivent). Comme AbstractExpr a AbstractInst comme super-classe, ceci permet d'implémenter verifyInst dans AbstractExpr à partir de verifyExpr en ignorant son résultat, exactement comme c'est spécifié dans la règle (3.20).

En gros, on peut associer à chaque non-terminal X de chaque grammaire de [SyntaxeContextuelle] une méthode "verifyX" ayant un profil correspondant dans une classe Y, de sorte que chaque règle de membre gauche X correspond à une implémentation de "verifyX" dans une sous-classe de Y. Cette méthode "verifyX" est typiquement déclarée dans la classe abstraite Y correspondant à la catégorie d'arbres traitée par X. Par exemple, on a vu ci-dessus que les non-terminaux inst et expr en passe 3 de [SyntaxeContextuelle] correspondent respectivement à la méthode verifyInst déclarée dans AbstractInst et verifyExpr déclarée dans AbstractExpr. Un autre exemple est le non-terminal rvalue qui correspond à la méthode verifyRValue de AbstractExpr.

Pour un exemple de code utilisant ce patron de conception, voir par exemple la calculette, fournie dans examples/calc/ et décrit dans [ANTLR].

6.4 Les styles de programmation à éviter

La technique décrite dans la section 6.3 a plusieurs avantages sur les autres approches qu'on pourrait imaginer pour implémenter des traitements sur les arbres.

Efficacité. La sélection du traitement à effectuer sur chaque nœud s'effectue à coût constant (c'est un appel de méthode).

Sûreté. Cette technique ne nécessite aucun « downcast » : la vérification des types est statique ce qui assure qu'un maximum de bogues est signalé par votre compilateur Java ou votre IDE, avant de lancer le moindre test. Par exemple, si vous ajoutez un nouveau noeud de type AbstractInst

^{5.} Formellement, si r est le résultat de la décompilation d'un arbre t dans la spécification, alors l'implémentation de "t.decompile(s)" esquissée ici, réalise "s.print(r)".

mais que vous oubliez d'implémenter le traitement verifyInst, cela sera signalé par le compilateur Java : la méthode abstraite verifyInst n'est pas implémentée dans le nouveau noeud.

Si vous utilisez une autre approche pour les parcours d'arbres, vous risquez d'écrire du code qui soit à la fois moins efficace et moins sûr. On en donne deux contre-exemples ci-dessous.

Premier contre-exemple, on cherche à réaliser un traitement par cas sur une instruction inst de type AbstractInst en testant explicitement son type dynamique :

```
if (inst instanceof While) {
    AbstractExpr expr = ((While) inst).cond ;
    ...
} else if (inst instanceof Return) {
    AbstractExpr expr = ((Return) inst).e ;
    ...
} ...
```

Deuxième contre-exemple, on cherche à réaliser un traitement par cas sur une instruction <code>expr</code> de type <code>AbstractBinaryExpr</code> en testant le nom de l'opérateur binaire :

Remarquons pour finir qu'il est facile de réécrire ces deux mauvais exemples dans le bon style :

- 1. introduire une méthode abstraite dans AbstractInst (respectivement AbstractBinaryExpr);
- 2. implémenter cette méthode dans les classes concrètes en utilisant le code de chacun des cas du if (respectivement switch).

[GuidePlanner]

Outil « Planner » pour gnome

Site web: http://live.gnome.org/Planner

Ce document vise à donner une introduction rapide à l'outil planner. Pour une utilisation plus avancée, consulter la documentation. Des exemples vous sont fournis dans le répertoire Projet_GL/planning. Il ne s'agit pas de modèles pour établir votre planning, seulement pour vous montrer ce qu'il est possible de faire avec planner, et les durées indiquées pour les tâches sont arbitraires.

L'onglet de droite permet de naviguer facilement entre les vues. On commence par aller sur la vue « ressource », pour définir les ressources disponibles. Ensuite, on définit les tâches dans la vue « tâche », puis on génèrera les dépendances simplement avec la vue « gantt ». La vue « utilisation ressource » permet de vérifier que les ressources ne sont pas sur- ou sous-attribuées.

1 Définir le calendrier

Dans le menu projet, cliquer sur « gérer les calendriers », pour définir les horaires de travail pour les jours travaillés ou non, ainsi que la répartition des jours travaillés dans la semaine. Il est aussi possible de définir d'autres types de jour (par exemple : « journée : on est à la bourre ») avec les horaires qui vont bien (suite de l'exemple : 8 :00 - 24 :00).

Pour pouvoir prévoir des tâches pendant les week-ends, il faut notamment spécifier que ces journées ne sont pas chômées (c'est-à-dire, sans période de travail). Cela peut être réalisé en rajoutant des heures de travail pour les jours de type « chômé » ou en sélectionnant un autre type de jour pour ces journées. Sans cela, aucune tâche ne commencera ou ne s'arrêtera pendant le week-end.

2 Création de ressource

Un clic droit dans la vue de ressource permet d'ajouter une ressource. Un second clic droit sur la ressource permet de l'éditer :

- ajout d'un nom et d'une abréviation pour la vue gantt
- adresse électronique éventuellement. On peut aussi directement cliquer sur une case pour l'éditer.

3 Création des tâches

Un clic droit dans l'écran des tâches permet de créer une tâche. Là encore un second clic droit permet de l'éditer, mais aussi de créer une sous-tâche. On peut modifier la durée d'une tâche, en utilisant le format suivant :

```
ENTIER w | ENTIER d | ENTIER h (pour week, day et hour)
```

La durée d'une tâche est divisée par le nombre de personnes y participant. Pour éviter cet effet sur certaines tâches, il faut éditer chacune de ces tâches, et cocher « durée fixe » dans l'onglet général. C'est aussi là qu'on modifiera l'état d'avancement de la tâche, et l'onglet ressource permet l'assignation des ressources.

[GuidePlanner] 174

4 Création des dépendances

On passe dans la vue « gantt ». La barre d'outil propose un outil pour zoomer ou dézoomer. Pour créer une dépendance, il suffit de faire un 'drag and drop' depuis la tâche prédécesseur vers la tâche dépendante. On peut faire des dépendances plus fines, celles de base sont : la tâche dépendante ne peut être commencée qu'après la fin de la tâche dont elle dépend. Il est possible de dire :

- on ne commence la tâche 2 qu'après avoir commencé la tâche 1;
- on ne peut finir la tâche 2 qu'après avoir fini la tâche 1;
- on ne peut finir la tâche 2 qu'une fois que la tâche 1 est commencée.

Pour cela, il faut éditer la tâche et rajouter la dépendance à la main dans l'onglet « dépendances ». En cas d'erreur, sélectionner la tâche en faute, en choisir l'action « délier » : cela supprimera toute

En cas d'erreur, sélectionner la tâche en faute, en choisir l'action « délier » : cela supprimera toutes ses dépendances.

La vue « ressource utilisation » permet de vérifier la répartition du temps de travail, avec le code couleur :

```
rouge: trop de travail;vert: pas de travail;bleu: peut travailler plus;
```

— gris : bonne utilisation de la ressource.

5 Exportation

Pour obtenir des rendus en pdf, choisir « imprimer », puis sélectionner l'imprimante pdf (il peut être nécessaire de cocher des vues dans l'onglet « select views » de la fenêtre de dialogue d'impression). Un export HTML est également disponible dans le menu Fichier (choisir « exporter > html »). On peut aussi bien évidemment imprimer ou sauvegarder le projet.

Fraude, propriété intellectuelle et logiciels

1 Introduction

Chaque personne participant au projet GL accepte et signe en début de projet la charte de travail en équipe [Suivi-SHEME-I] et de « Fraude interdite et sanctionnée pour les TP et projets ». Cette charte ¹ est nécessaire pour défendre les droits de propriété intellectuelle des enseignants et de l'Ensimag sur le matériel de conception préparatoire mis à la disposition de leurs étudiants dans une finalité pédagogique (en entreprise : finalité économique ou sociétale).

Le but de ce document est de vous apporter un éclairage juridique et stratégique sur des notions étroitement liées au génie logiciel autour de la fraude ou des sanctions (en entreprise : la contrefaçon, le plagiat ou plus généralement les risques, garanties ou responsabilités liés à la production), la (ré)utilisation licite de droits de propriété intellectuelle (en entreprise : l'intelligence économique ou la veille technologique pour l'agrégation de composants logiciels exploitables ou la production outillée de code logiciels comme avec [ANTLR]) et l'acquisition ou la défense de ces droits (en entreprise : secret et savoir-faire industriels, brevet et autres titres de propriété intellectuelle déterminants pour la compétitivité et l'innovation, dans un environnement concurrentiel).

Ce document ne se substitue pas à un audit juridique et autres expertises. Dans le projet GL (et plus tard), il ne faut pas oublier les possibilités d'analyse (internes ou par des concurrents) et la capacité d'autres outils de génie logiciel de détection automatique de fraude ou d'incompatibilité de licence ou d'activité anormale dans un système de production (et leurs évolutions : ces méthodes et outils s'améliorent sans cesse, avec les capacités de mémorisation ou d'accès à l'ensemble des projets connus et avec les capacités de calcul sur la totalité du processus de production logicielle).

En résumé, il n'est pas nécessaire de maitriser le contenu de ce document pour faire le projet GL si vous respectez les consignes, mais ces notions vous seront probablement utiles plus tard.

2 Protection du logiciel

Le logiciel est protégé par le droit d'auteur. En France, le logiciel en tant que tel ne peut pas faire l'objet d'un brevet. Le brevet pourra être déposé sur la méthode ou le procédé lié(e) au logiciel.

Article L112-2 CPI ² : « Sont considérés notamment comme œuvres de l'esprit au sens du présent code : (...) 13 ° les logiciels, y compris le matériel de conception préparatoire. »

^{1.} Dans les projets libres, l'équivalent est la charte de contributeur; dans les projets en entreprise, l'équivalent est le contrat de travail et réglement intérieur ou autres conventions ou contrats (pour les stagiaires : convention de stage et cession des droits en contrepartie de l'accueil), ouvrant les accès en contribution au projet.

^{2.} Code de la Propriété Intellectuelle (CPI) http://www.legifrance.gouv.fr/affichCode.do?cidTexte=LEGITEXT00006669414.

3 Notion d'auteur du logiciel

L'auteur du logiciel est celui qui le crée, c'est-à-dire celui qui écrit des lignes de code originales mais également celui qui participe au matériel de conception préparatoire (les spécifications) et à la documentation. Il n'y a besoin d'aucune formalité pour protéger le logiciel, à la différence d'une invention brevetable ou d'une marque qui doit faire l'objet d'un dépôt de titre auprès de l'INPI en France.

Un dépôt du logiciel auprès de l'Agence pour la Protection des Programmes ³ est néanmoins recommandé pour se constituer une preuve de l'existence du logiciel à une date donnée.

4 Le titulaire des droits

Article L111-1 CPI : « L'auteur d'une œuvre de l'esprit jouit sur cette œuvre, du seul fait de sa création, d'un droit de propriété incorporelle exclusif et opposable à tous. »

Il faut néanmoins distinguer l'auteur du titulaire des droits sur le logiciel, qui peuvent être deux personnes différentes.

Il existe ainsi une spécificité en matière de logiciel lorsque l'auteur est salarié. Dans ce cas, les droits portant sur le logiciel sont automatiquement dévolus à l'employeur. L'employeur est alors le titulaire des droits, l'auteur salarié gardant sa qualité d'auteur.

Article L113-9 CPI : « Les droits patrimoniaux sur les logiciels et leur documentation créés par un ou plusieurs employés dans l'exercice de leurs fonctions ou d'après les instructions de leur employeur sont dévolus à l'employeur qui est seul habilité à les exercer (...). »

5 Les droits d'auteur

Le droit d'auteur se scinde en deux catégories : les droits patrimoniaux et les droits moraux.

5.1 Les droits moraux et leur application aux logiciels

Les droits moraux sont perpétuels, inaliénables, insaisissables et imprescriptibles, et se composent du droit à la paternité et au respect de son œuvre, du droit de divulgation de l'œuvre, du droit de repentir ou de retrait ⁴.

Mais dans le cas d'un logiciel, la jurisprudence a déterminé que le droit moral de l'auteur se réduisait seulement au droit au nom.

5.2 Les droits patrimoniaux

Les droits patrimoniaux sont cessibles, saisissables, prescriptibles et transmissibles. Le titulaire des droits patrimoniaux en bénéficient durant sa vie + 70 ans 5 .

Ces droits patrimoniaux, en matière de logiciel, sont les suivants ⁶:

- le droit de reproduction Dans la mesure où le chargement, l'affichage, l'exécution, la transmission ou le stockage du logiciel nécessitent une reproduction, ces actes ne sont possibles qu'avec l'autorisation du titulaire des droits.
- 3. L'Agence pour la Protection des Programmes (APP) http://www.app.asso.fr/ attribue un numéro IDDN aux logiciels (équivalent du ISBN pour les livres).
 - 4. articles L121-1 et suivants CPI
 - 5. 70 ans à compter du 1er janvier de l'année suivant la divulgation de l'œuvre si le titulaire est une personne morale
 - 6. article L122-6 CPI, et ses exceptions article L122-6-1

— le droit d'adaptation

Consiste dans la traduction, l'adaptation, l'arrangement ou toute autre modification d'un logiciel et la reproduction du logiciel en résultant.

— le droit de mise sur le marché

Consiste dans la mise sur le marché à titre onéreux ou gratuit, y compris la location, du ou des exemplaires d'un logiciel par tout procédé.

Mais il existe des exceptions, c'est-à-dire que pour réaliser les actions suivantes, il n'est pas nécessaire d'obtenir l'autorisation du titulaire des droits :

— le droit de réaliser les actes nécessaires pour permettre l'utilisation du logiciel Cela comporte notamment le droit de corriger les erreurs.

Toutefois, le titulaire des droits peut se réserver par contrat le droit de réaliser ces actes.

— le droit de réaliser une copie de sauvegarde

Uniquement lorsque celle-ci est nécessaire pour préserver l'utilisation du logiciel et doit être réalisée par la personne ayant le droit d'utiliser le logiciel.

— le droit d'observer, étudier et tester

Consiste dans l'observation, l'étude ou le test du fonctionnement du logiciel afin de déterminer les idées et principes qui sont à la base du logiciel.

— le droit de reproduire ou traduire le logiciel à des fins d'interopérabilité

Ces actes doivent être accomplis par la personne ayant le droit d'utiliser le logiciel (ou pour son compte) et uniquement si les informations nécessaires à l'interopérabilité n'ont pas déjà été rendues accessibles.

Ces actes doivent être limités aux parties du logiciel nécessaires à l'interopérabilité.

Les informations ainsi obtenues ne peuvent être utilisées à d'autres fins que l'interopérabilité, ne doivent pas être communiquées à des tiers (sauf si cela est nécessaire à l'interopérabilité) et ne peuvent être utilisées pour la production d'un logiciel similaire ou pour tout autre acte portant atteinte au droit d'auteur.

6 L'exploitation des droits patrimoniaux

Il faut faire la distinction entre deux actes juridiques différents :

— la vente du support matériel du logiciel

Il s'agit d'un acte de vente classique.

L'acheteur est propriétaire du support mais ne peut pas pour autant utiliser le logiciel comme il le souhaite (restriction par un fichier licence à la lecture du support par exemple).

— la concession d'un droit d'utilisation du logiciel

Ceci fait l'objet d'un contrat particulier par lequel le titulaire des droits va lister un certain nombre d'actes que l'utilisateur pourra faire ou ne pas faire.

Tout acte d'exploitation des droits patrimoniaux d'un logiciel nécessite un contrat :

— le contrat de licence

Ce document octroie au licencié le droit d'utiliser et/ou d'exploiter le logiciel selon des conditions négociées entre le titulaire des droits et le licencié.

Le titulaire des droits reste propriétaire du logiciel.

— le contrat de cession

Ce document a pour objet de céder un certain nombre de droits à un cessionnaire.

Le cessionnaire devient propriétaire des droits cédés par le titulaire dans les conditions fixées par le contrat.

Dans tous les cas, le contrat doit obligatoirement comporter un certain nombre de mentions 7.

^{7.} L'article L131-3 CPI précise que la transmission des droits de l'auteur est subordonnée à la condition que chacun des droits cédés fasse l'objet d'une mention distincte dans l'acte et que le domaine d'exploitation des droits cédés soit délimité quant à son étendue et à sa destination, quant au lieu et quant à la durée.

7 Cas d'exploitation spécifique : les logiciels libres

Le logiciel libre est un logiciel diffusé sous une licence dite libre ⁸ qui donne à chacun le droit de l'utiliser, de l'étudier, de le modifier, de le reproduire et de le diffuser.

Logiciel libre ne veut pas dire logiciel sans droits (ni sans responsabilités) : seul le titulaire des droits peut autoriser l'utilisateur par le biais d'un texte de licence à exploiter son logiciel.

Il existe plusieurs types de licences libres :

- licences copyleft (fort copyleft : CeCILL ⁹ , GPL faible copyleft : LGPL, CeCILL-C)

 Licences à fort copyleft : Ces licences imposent que toute redistribution du logiciel, modifié ou non, se fasse sous les termes de la licence initiale. Elles sont dites « contaminantes ».

 Licences à faible copyleft : généralement il s'agit de licences permissives en matière d'aggrégation de composants. Elles ont été créées en premier lieu à destination des bibliothèques logicielles.
- licences de type BSD (Licence Apache, MIT, BSD, CeCILL-B)

 Ce sont des licences sont très permissives et elles offrent une grande liberté d'utilisation du logiciel. Habituellement, seule la citation des auteurs est demandée.

— le domaine public

Le logiciel appartient à tous. Les droits patrimoniaux disparaissent et le logiciel peut alors être utilisé librement. Cette notion est théorique car aucun logiciel n'est encore tombé dans le domaine public. Cependant, on trouve des logiciels « domaine public » qui sont en réalité des logiciels dont les auteurs ont renoncé à faire valoir leurs droits. Mais cette notion n'a aucun équivalent en droit français.

8 La défense des droits

La défense des droits et les sanctions sont proportionnées aux enjeux économiques et sociétaux.

Les sanctions pénales :

Article L335-2 CPI (applicable aux logiciels) : « Toute édition d'écrits, de composition musicale, de dessin, de peinture ou de toute autre production, imprimée ou gravée en entier ou en partie, au mépris des lois et règlements relatifs à la propriété des auteurs, est une contrefaçon et toute contrefaçon est un délit.

La contrefaçon en France d'ouvrages publiés en France ou à l'étranger est punie de trois ans d'emprisonnement et de 300 000 euros d'amende. Seront punis des mêmes peines le débit, l'exportation et l'importation des ouvrages contrefaisants.

Lorsque les délits prévus par le présent article ont été commis en bande organisée, les peines sont portées à cinq ans d'emprisonnement et à 500~000 euros d'amende. »

Les sanctions civiles :

Article L331-1-3 CPI (et suivants) : « Pour fixer les dommages et intérêts, la juridiction prend en considération les conséquences économiques négatives, dont le manque à gagner, subies par la partie lésée, les bénéfices réalisés par l'auteur de l'atteinte aux droits et le préjudice moral causé au titulaire de ces droits du fait de l'atteinte.

Toutefois, la juridiction peut, à titre d'alternative et sur demande de la partie lésée, allouer à titre de dommages et intérêts une somme forfaitaire qui ne peut être inférieure au montant des redevances ou droits qui auraient été dus si l'auteur de l'atteinte avait demandé l'autorisation d'utiliser le droit auquel il a porté atteinte. »

Les autres sanctions (y compris non judiciaires) : pédagogiques ou liées à l'image de marque ou à la renommée de l'entreprise ou de la personne.

^{8.} Il n'y a pas de définition immuable, par exemple: http://www.gnu.org/philosophy/free-sw.fr.html#History.

^{9.} CeCILL: CEA CNRS Inria Logiciel Libre, famille de licences libres françaises http://www.cecill.info.

Quatrième partie

Compléments sur la compilation du langage Deca

Exemple avec objet illustrant les étapes de compilation

Ce document donne un exemple complémentaire de [ExempleSansObjet]. Il illustre notamment la « sémantique de partage » des décorations de l'arbre enrichi mentionné dans [ArbreEnrichi]. Il illustre aussi les aspects spécifiques aux programmes Deca objet dans les étapes B et C du compilateur.

```
// Un exemple de programme Deca
2
3
    class A {
4
       protected int x ;
5
       int getX() {
6
          return x ;
 7
8
       void setX(int x) {
9
          this.x = x;
10
    }
11
12
13
       A = new A();
14
15
16
       a.setX(1);
17
       println("a.getX() = ", a.getX());
18
```

1 Étape d'analyse syntaxique

1.1 Analyse lexicale

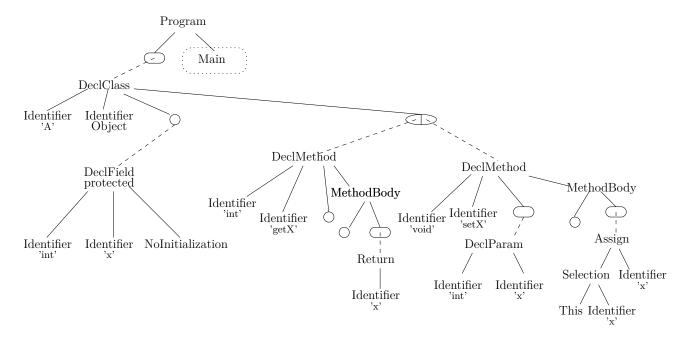
Séquence des lexèmes, avec numéros de ligne, noms des identificateurs et valeurs des littéraux.

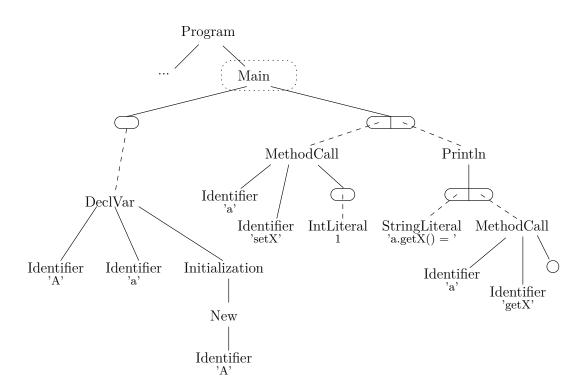
CLASS	IDENT	OBRACE	PROTECTED	IDENT	ID	ENT SE	MI	IDENT	IDENT		
3	3 'A'	3	4	4 'int	;' 4	'x' 4		5 'int'	5 'getX		
OPAREN	IT CPARE	ENT OBRA	CE RETURN	I IDEN	T SE	MI CBR	ACE	IDENT	IDEN	Г ОР	PARENT
5	5	5	6	6 'x'	' 6	7		8 'void	l' 8 'se	tX' 8	
IDENT	IDENT	CPARENT	OBRACE	THIS	DOT	IDENT	EQU.	ALS II	ENT SEI	MI CBI	RACE
8 'int'	' 8 'x'	8	8	9	9	9 'x'	9	9	'x' 9	10	
•											
CBRACE	OBRACE	IDENT	IDENT	EQUALS	NEW	IDENT	OP.	ARENT	CPARENT	SEMI	IDENT
11	13	14 'A'	14 'a'	14	14	14 'A'	14		14	14	16 'a'
DOT	IDENT	OPARENT	INT CF	ARENT	SEMI	PRINTL	ИО	PARENT	STRING		COMMA
16	16 'setX'	16	16 1 16		16	16	1	6	17 'a.ge	tX() = '	17

IDENT	DOT	IDENT	OPARENT	CPARENT	CPARENT	SEMI	CBRACE
17 'a'	17	17 'getX'	17	17	17	17	18

1.2 Arbre abstrait

L'arbre abstrait correspondant au programme est le suivant :





2 Étape de vérifications contextuelles et décorations

Abréviations:

- D : Definition

— Т: Туре

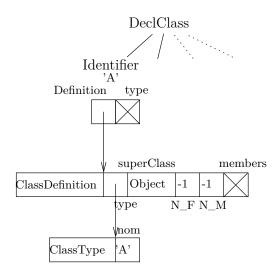
- N_F : numberOfFields - N_M : numberOfMethods

— op : Operand
— sig : Signature

L'étape de vérifications contextuelles est réalisée en trois passes (il y a donc trois parcours de l'arbre abstrait).

2.1 Passe 1

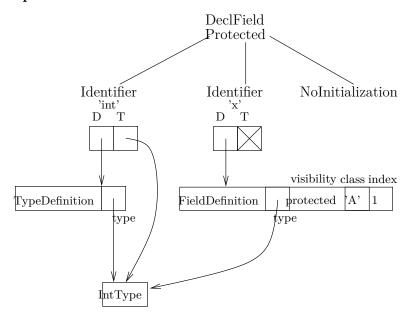
Au cours de la première passe (spécifiée en section 5 de [SyntaxeContextuelle]), on vérifie le nom des classes et la hiérarchie de classes.



2.2 Passe 2

Au cours de la deuxième passe (spécifiée en section 6 de [SyntaxeContextuelle]), on vérifie les déclarations de champs et la signature des méthodes, et on met à jour leur index; on construit également l'environnement members de chaque classe, et on met à jour les numberOfFields et numberOfMethods des ClassDefinitions.

Déclaration du champ x

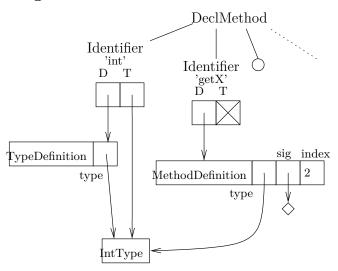


L'identificateur x est décoré avec la Definition (field(protected, A), int). 1

Le champ type associé à x est nul car il s'agit d'une occurrence de déclaration et non d'une occurrence d'utilisation.

Le champ Index est 1, car il n'y a pas de champ dans Object.

Déclaration de la méthode getX



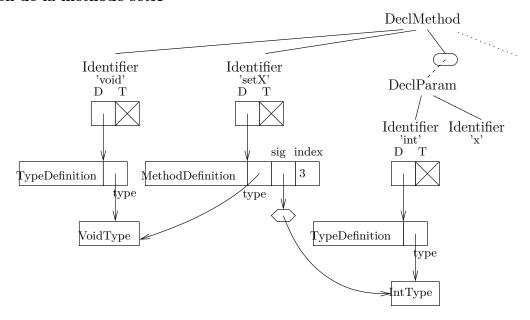
L'identificateur getX est décoré avec la Definition ($\underline{\text{m\'ethode}}([\]),\ \underline{\text{int}}).$

Le champ signature (Sig) est une liste vide car la méthode getX n'a pas de paramètre.

Le champ index est 2, car il y a déjà une méthode dans Object (equals); getX est donc la deuxième méthode de A.

^{1.} La notation (<u>field(protected, A), int</u>) est celle de la grammaire attribuée. L'objet Java correspondant est de type FieldDefinition avec visibility = PROTECTED, containingClass est une référence vers la ClassDefinition de la classe A, et type est une instance de IntType. La notation de la grammaire attribuée est utilisée car plus concise.

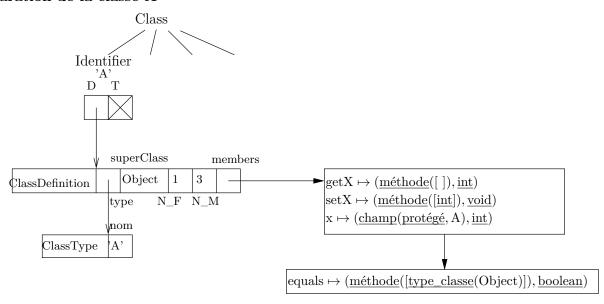
Déclaration de la méthode setX



L'identificateur setX est décoré avec la Definition (méthode([int]), void). Son index est 3.

Les types des paramètres (ici, le type $\underline{\mathrm{int}}$) sont analysés, afin de construire la signature de la méthode. Les paramètres (ici, le paramètre x) seront analysés et décorés lors de la troisième passe. La signature est cette fois une liste à un élément.

Déclaration de la classe A

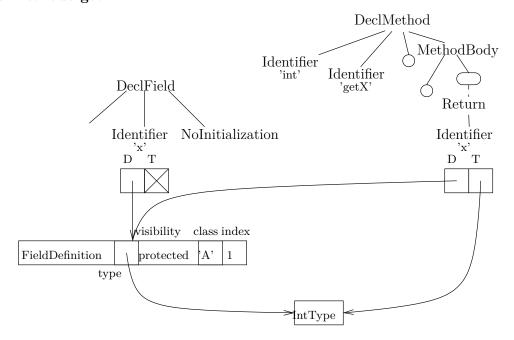


Ses champs numberOfFields et numberOfMethods valent respectivement 1 et 3.

2.3 Passe 3

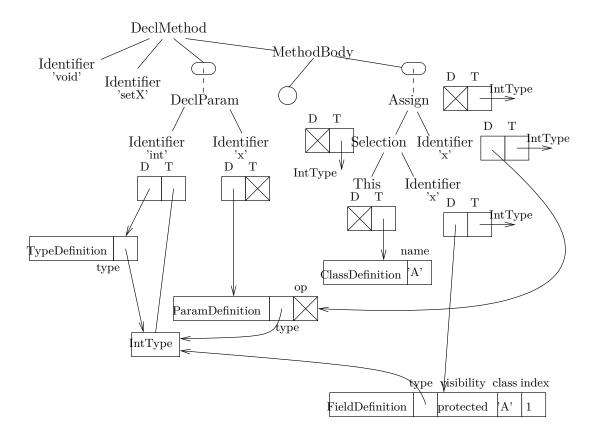
Au cours de la troisième passe (voir section 7 de [SyntaxeContextuelle]), on vérifie les initialisations et le corps des méthodes.

Corps de la méthode getX

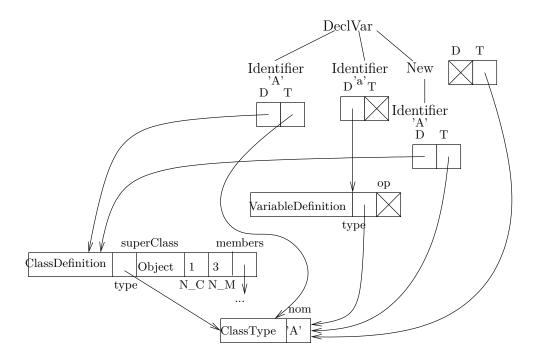


Ici, le type de x est rempli car il s'agit d'une occurrence d'utilisation.

Corps de la méthode set X



Déclaration et initialisation de la variable a



3 Étape de génération de code

Au cours de la génération de code, les Operandes des Definitions de variables et paramètres sont mis à jour au fur et à mesure qu'on rencontre leur déclaration avec des adresses de la forme d(GB) ou d(LB) (voir [ConventionsLiaison]).

élément	Operand
paramètre x	-3(LB)
variable a	7(GB)

Un premier parcours des classes permet de créer pour chacune le tableau de ses méthodes et de le remplir avec des opérandes qui sont les étiquettes des méthodes (voir section 3 du document [Gencode]). Pour la classe A le tableau est :

1	code.Object.equals
2	code.A.getX
3	code.A.setX

Une fois le tableau rempli, on peut générer le code de la construction de la table des méthodes de la classe (voir [Gencode]).

État de la pile dans l'appel a.setX(1)

$\mathtt{SP},\mathtt{LB}\longrightarrow$	0(GB)	(≡ ancien LB)
51, 22	PC	
-2(LB)	@ de a (dans le tas)	
-3(LB)	1	
7(GB)	@ de a (dans le tas)	variable globale
6(GB)	code.A.setX	table des
5(GB)	code.A.getX	méthodes
4(GB)	code.Object.equals	de A
3(GB)	1(GB)	$(\equiv @ \text{ de la table de la superclasse})$
2(GB)	code.Object.equals	table des méthodes de Object
1(GB)	null	$(\equiv \text{pas de superclasse})$
$\overline{\hspace{1.5cm} \texttt{GB} \longrightarrow}$		

Programme assembleur (ajouter le code de Object.equals)

```
TSTO #11
    BOV pile_pleine
    ADDSP #7
      Construction des tables des methodes
 -----
; Construction de la table des methodes de Object
    LOAD #null, RO
    STORE RO, 1 (GB)
    LOAD code.Object.equals, RO
    STORE RO, 2 (GB)
; Construction de la table des methodes de A
    LEA 1 (GB), RO
    STORE RO, 3 (GB)
    LOAD code.Object.equals, RO
    STORE RO, 4 (GB)
    LOAD code.A.getX, RO
    STORE RO, 5 (GB)
    LOAD code.A.setX, RO
    STORE RO, 6 (GB)
           Code du programme principal
; -----
; new ligne 13
    NEW #2, R2
    BOV tas_plein
    LEA 3 (GB), RO
    STORE RO, 0 (R2)
    PUSH R2
    BSR init.A
    POP R2
    STORE R2, 7 (GB)
; Appel de methode ligne 16
    ADDSP #2
    LOAD 7 (GB), R2
```

```
STORE R2, 0 (SP)
    LOAD #1, R2
    STORE R2, -1 (SP)
    LOAD 0 (SP), R2
    CMP #null, R2
    BEQ dereferencement_null
    LOAD 0 (R2), R2
    BSR 3 (R2)
    SUBSP #2
; Instruction println ligne 17
    WSTR "a.getX() = "
; Appel de methode ligne 17
    ADDSP #1
    LOAD 7 (GB), R2
    STORE R2, 0 (SP)
    LOAD O (SP), R2
    CMP #null, R2
    BEQ dereferencement_null
    LOAD 0 (R2), R2
    BSR 2 (R2)
    SUBSP #1
    LOAD RO, R1
    WINT
    WNL
    HALT
                    Classe A
 -----
; ----- Initialisation des champs de A
init.A :
    LOAD #0, RO
    LOAD -2 (LB), R1
    STORE RO, 1 (R1)
; ----- Code de la methode getX dans la classe A ligne 5
code.A.getX :
    TSTO #1
    BOV pile_pleine
; Sauvegarde des registres
    PUSH R2
; Instruction return ligne 6
    LOAD -2 (LB), R2
    LOAD 1 (R2), R2
    LOAD R2, R0
    BRA fin.A.getX
    WSTR "Erreur : sortie de la methode A.getX sans return"
    WNL
    ERROR
fin.A.getX :
; Restauration des registres
    POP R2
    RTS
```

```
; ----- Code de la methode setX dans la classe A ligne 8
code.A.setX :
   TSTO #2
   BOV pile_pleine
; Sauvegarde des registres
   PUSH R2
   PUSH R3
; Affectation ligne 9
   LOAD -2 (LB), R2
   CMP #null, R2
   BEQ dereferencement_null
   LOAD -3 (LB), R3
   STORE R3, 1 (R2)
fin.A.setX :
; Restauration des registres
   POP R3
   POP R2
   RTS
; -----
    Message d'erreur : dereferencement de null
; -----
dereferencement_null :
   WSTR "Erreur : dereferencement de null"
   WNL
   ERROR
; -----
       Message d'erreur : pile pleine
pile_pleine :
   WSTR "Erreur : pile pleine"
   WNL
   ERROR
; -----
; Message d'erreur : allocation impossible, tas plein
: -----
tas_plein :
   WSTR "Erreur : allocation impossible, tas plein"
   WNL
   ERROR
```

ANTLR: ANother Tool for Language Recognition

1 ANTLR: vue d'ensemble

1.1 Principes généraux

ANTLR est un outil pour générer des reconnaisseurs de langages. En général, on découpe le travail en au moins deux étapes :

l'analyseur lexical (ou lexer) reconnaît les « mots » du langage. Il découpe la suite de caractères qu'il lit sur son entrée en une suite de lexèmes (« lexèmes » aussi appelés « jetons » ou « tokens » en anglais). Par exemple, si on cherche à reconnaître 12+42 comme une expression arithmétique, on peut découper ces 5 caractères en un jeton de type « nombre » 12, puis un jeton + et un jeton pour 42.

l'analyseur syntaxique (ou parser) reconnaît les « phrases » du langage. Il utilise la suite de jetons produite par le lexer pour analyser globalement la structure du langage. Par exemple, notre

chaîne 12+42 pourrait être interprétée comme l'arbre $\begin{pmatrix} + \\ \\ 12 \\ 42 \end{pmatrix}$ valeur 54.



Avec ANTLR, en général, on implémente le lexer dans un fichier source, et le parser dans un autre fichier source. Ces fichiers sont des fichiers spécifiques à ANTLR, on les nomme avec une extension .g4 (en ANTLR4) et leur syntaxe est définie ci-dessous. Chaque fichier source est compilé par ANTLR en un fichier Java (le lexer correspond à une classe, le parser à une autre). Ces fichiers Java seront compilés comme les autres fichiers Java du programme pour donner des fichiers .class.

1.2 Pourquoi ANTLR?

Parmi les points forts d'ANTLR4, on peut citer :

- L'utilisation d'analyseurs ALL(*), capables de traiter n'importe quelle grammaire hors-contexte ne contenant pas de récursion-gauche indirecte.
- La possibilité d'avoir une bonne gestion des erreurs (messages d'erreur précis, possibilité de rattraper les erreurs, ...) et un code d'analyseur généré à la fois lisible et efficace.
- L'aspect multi-langage : ANTLR est écrit en Java, mais peut générer des analyseurs en Ada95, ActionScript, C, C#, Java, JavaScript, Objective-C, Perl, Python, Ruby et Standard ML!
- L'utilisation du même outil pour gérer l'analyse lexicale et syntaxique (on peut au choix séparer les deux aspects dans des fichiers source différents ou combiner les deux)
- La possibilité (non-utilisée dans le projet) de générer automatiquement un arbre abstrait, et d'y appliquer automatiquement certains traitements.
- Le code généré par ANTLR est fait pour être lisible et débuggable. Quand quelque chose se passe mal, le programmeur a les outils pour résoudre les problèmes.

1.3 Avertissement au lecteur

Cette documentation a été écrite pour le projet génie logiciel de l'Ensimag, et se concentre sur une manière particulière d'utiliser ANTLR. On fait volontairement l'impasse sur des sujets pourtant intéressants comme :

- Les « tree parsers », qui permettent de bien séparer le code Java et les grammaires ANTLR. On pourrait écrire notre analyseur en 3 passes : un lexer qui produit des jetons, un parser qui produit un arbre abstrait (AST) automatiquement d'après les règles, et un parser d'arbre qui transformerait l'AST ANTLR en notre représentation de l'arbre abstrait.
- Le rattrapage d'erreur, qui permet à un analyseur de continuer après la première erreur, pour signaler plusieurs erreurs en une compilation par exemple. Faire du rattrapage d'erreur sans utiliser les parsers d'arbres n'est pas facile puisqu'il faut que les actions de chaque règle de grammaire prennent en compte les cas où une sous règle a échoué puis rattrapé une erreur (et typiquement renvoyé un pointeur nul).
- L'utilisation d'ANTLR avec un langage autre que Java.
- Les outils de déboggage des grammaires (celle du projet GL étant donnée, vous n'avez pas à la "débogguer").

2 Structure d'un fichier source ANTLR

La syntaxe des fichiers source ANTLR reprend quelques idées de la syntaxe de Java :

- Les commentaires se font /* comme ceci */ ou // comme celà.
- Les espaces, tabulations et retours à la ligne sont de simples séparateurs et ne sont pas significatifs.

Un fichier source ANTLR est constitué de deux parties : la première partie du fichier donne un certain nombre de méta-données, utilisés dans la génération de code. La seconde partie est une suite de règles qui décrivent le langage à analyser.

Un fichier source commence par une ligne déclarant le type de fichier. Pour un analyseur lexical, on écrit :

```
lexer grammar NomDeClasse;
```

Pour un analyseur syntaxique, on écrit :

```
parser grammar NomDeClasse;
```

Le code généré pour le lexer sera une classe de nom NomDeClasse.

Cette ligne est suivie d'une suite de sections déclarées avec la syntaxe nom-de-section { contenu } .

La section options permet de donner un certain nombre de directives à ANTLR. Elle se présente sous la forme

```
options {
   nom-option = valeur;
   autre-option = valeur;
   ...
}
```

Les options valides pour les deux types d'analyseurs (lexical et syntaxique) sont :

language : le langage de programmation à utiliser pour le code généré. La valeur par défaut est Java. super-Class : super-classe à utiliser pour le code généré (i.e. le code sera généré dans une classe qui hérite de la classe spécifiée par l'option superClass)

tokenVocab : vocabulaire d'entrée à utiliser. Cette option est typiquement utilisée par les parsers, auquel cas on met ici le nom du lexer depuis lequel on va lire des jetons.

La section **@header** est une portion de code Java qui sera ajoutée en tête du fichier Java généré. En général, on l'utilise pour déclarer le nom du paquetage dans lequel la classe doit être générée, et si besoin utiliser la directive **import** pour déclarer les dépendances de cette classe. Cet en-tête est de la forme :

```
@header {
   package nom.du.paquetage;
}
```

La section @members est une portion de code Java qui sera insérée à l'intérieur de la classe générée. Comme son nom l'indique, on peut l'utiliser pour déclarer des membres (champs ou méthodes) pour la classe générée.

3 Analyse lexicale (Lexer) avec ANTLR

3.1 Corps du fichier : les règles

Le corps du fichier source du lexer est une suite de règles, en général de la forme :

```
NOM_REGLE: expression;
```

Pour les règles de lexer, il est impératif que le nom (NOM_REGLE) commence par une majuscule.

ANTLR va compiler l'ensemble de ces règles en un gros automate, qui va décider quelle règle appliquer en fonction de l'entrée. Plus précisément, quand une règle reconnaît un préfixe de l'entrée du programme, le comportement par défaut est "consommer" ce préfixe (il ne sera plus là à la prochaine invocation du lexer), et de produire un jeton, c'est à dire un objet du type org.antlr.runtime.Token qui contient entre autres une valeur représentant la règle qui vient d'être appliquée (ici, NOM_REGLE). Ce comportement est suffisant si le but est de fournir une séquence de jetons au parser, mais on peut parfois souhaiter faire autre chose. On peut faire ceci en ajoutant une action à la règle, c'est à dire un morceau de code Java entre accolades :

```
NOM_REGLE : expression { action };
Par exemple :
ESPACE : ' ' { System.out.println("J'ai reconnu un espace"); } ;
```

Dans les actions des règles, on peut faire appel aux méthodes définies par l'API du runtime ANTLR. Le effet, dans le cadre du projet, le code des actions a lieu au sein de la classe DecaLexer générée par ANTLR, qui hérite de AbstractDecaLexer, qui hérite elle-même de la classe Lexer du paquetage org.antlr.runtime. Typiquement, il peut être utile d'invoquer dans une action la méthode skip() héritée de la classe Lexer, ce qui permet de ne pas générer de jeton pour la règle courante. Voir par exemple la règle ci-dessous (qui décrit les commentaires d'un script Perl ou shell):

```
COMMENT : '#' (~('\n'))* { skip(); };
```

Une autre méthode utile de la classe Lexer est getText() qui permet de récupérer un objet String correspondant à la chaîne de caractères reconnue par la règle. Sur le même principe, on peut aussi utiliser des méthodes héritées de AbstractDecaLexer comme doInclude (cf. section 2.4 du document [Consignes]).

Pour structurer les expressions, on peut utiliser des « fragments » de règles :

```
fragment NOM : expression;
```

^{1.} Voir le javadoc en ligne sur http://www.antlr.org/api/Java/org/antlr/v4/runtime/package-summary.html.

Ces règles peuvent être utilisées dans d'autres règles (on peut les voir comme des macros), mais ne produisent pas de jeton. Par exemple, plutôt que d'écrire :

```
NOMBRE : ('0' .. '9')+;
on peut écrire :
fragment CHIFFRE : '0' .. '9';
NOMBRE : CHIFFRE+;
```

3.2 Syntaxe des expressions régulières

Les constructions disponibles dans les expressions régulières sont les suivantes :

```
c': le caractère c (exemple : c') pour reconnaître le caractère =)
```

```
'\n': un caractère « ligne suivante » (LF)
```

'\r': un retour chariot (CR)

'\t': une tabulation

'\\' : un antislash

'\'': un guillemet simple.

'chaine': la chaîne de caractère (exemple : 'if' pour reconnaître le mot clé if).

.: n'importe quel caractère (y compris '\n' et '\r').

expr1 expr2 : l'expression expr1 suivie de expr2. Par exemple, 'i', 'f' est équivalent à 'if'.

'c1' .. 'c2' : n'importe quel caractère entre c1 et c2. Par exemple, 'a' .. 'z' pour reconnaître n'importe quelle lettre minuscule.

(expr) : reconnaît l'expression expr (parenthèses de groupement).

expr1 | expr2 : reconnaît soit l'expression expr1, soit l'expression expr2. Par exemple, ('a' .. 'z' | 'A' .. 'Z') pour reconnaître une lettre minuscule ou majuscule.

- expr*: reconnaît soit la chaîne vide, soit l'expression expr, soit l'expression expr répétée un nombre quelconque de fois. Par exemple : ('0' .. '9')* pour reconnaître une séquence possiblement vide de chiffres. Par défaut, l'étoile est « gloutonne », c'est à dirque qu'ANTLR va chercher à reconnaître une chaîne la plus grande possible, donc à répéter expr un maximum de fois. On peut changer ce comportement en suffixant l'étoile par un point d'interrogation, soit "*?". Par exemple, sur l'entrée /* foo */ bar */, l'expression '/*' .*? '*/' va reconnaître /* foo */, alors que l'expression '/*' .* '*/' aurait reconnu la chaîne /* foo */ bar */.
- expr+: reconnaît l'expression expr répétée au moins une fois. En d'autres termes, expr+ est équivalent à expr=expr*.
- \sim expr: un caractère qui ne correspond pas à l'expression expr. L'expression expr doit être limitée à un caractère. Par exemple : \sim ('"' | '\n') pour reconnaître un caractère, autre qu'un guillemet double ou un retour à la ligne.

En dehors des guillemets, les espaces et retour chariots ne sont pas significatifs. Par exemple, la règle suivante

3.3 Résolution des ambiguïtés

Il peut arriver qu'une même chaîne en entrée soit reconnue par plusieurs règles. Dans ce cas, ANTLR applique deux règles pour résoudre l'ambiguïté :

Principe de la plus longue correspondance : chaque règle de l'analyseur va tenter de reconnaître une chaîne la plus longue possible. Par exemple, sur la grammaire suivante :

```
ELSE : 'else';
ELSEIF : 'elseif';
IF : 'if';
SPACE : ' ';
```

Sur l'entrée else if, il n'y a pas d'ambiguïté donc les règles ELSE, SPACE et IF sont choisies. Sur l'entrée elseif, on aurait pu imaginer reconnaître ELSE puis IF, mais le principe de plus longue correspondance va forcer la règle ELSEIF.

Première règle prioritaire : Dans le cas où le principe de la plus longue correspondance ne permet pas de résoudre l'ambiguïté (deux règles reconnaissent un préfixe de l'entrée de même longueur), la première règle déclarée dans le fichier source ANTLR est prioritaire. Par exemple, sur la grammaire suivante :

```
IF : 'if';
SPACE : ' ';
IDF : 'a' .. 'z' *;
```

Sur l'entrée ifoo, il y a ambiguïté entre IF et IDF, mais IDF permet de reconnaître un préfixe plus long, donc c'est elle qui l'emporte. Sur l'entrée if oo, les deux règles reconnaissent le préfixe if (puis un espace). Ici, c'est la première règle qui l'emporte, donc IF. Si on changeait l'ordre des règles comme ceci :

```
IDF : 'a' .. 'z' *;
IF : 'if';
SPACE : ' ';
```

alors c'est IDF qui l'emporterait, mais dans ce cas, la règle IF ne peut plus jamais être activée, et ANTLR lève une erreur à la compilation de la grammaire.

Un cas particulier intéressant pour écrire une règle qui n'est activée que quand aucune autre ne reconnaît l'entrée, on peut écrire par exemple DEFAULT: . ; en fin de fichier. La règle reconnaît n'importe quel caractère, et reconnaît toujours un préfixe de taille 1, donc elle n'est prioritaire vis-à-vis d'aucune des règles ci-dessus.

4 Analyse syntaxique (Parser) avec ANTLR

4.1 Règles de grammaires

Comme pour le lexer, un fichier de parser est constitué d'un en-tête puis d'un ensemble de règles. Les règles de parser doivent avoir un nom commençant par une minuscule. Les règles définissent une grammaire hors-contexte au format EBNF. Une règle de grammaire peut faire appel à une autre règle de grammaire ou à des règles de lexer. Par exemple :

```
sum_expr : mult_expr (PLUS mult_expr)*;
mult_expr: INTEGER;
```

Cette règle définie un non-terminal sum_expr qui se ré-écrit en le non-terminal mult_expr (défini dans une autre règle de parser) suivi d'un nombre quelconque de fois la succession du terminal (donc du jeton) PLUS et d'une autre instance de mult_expr. Si PLUS et INTEGER ont été définis par des

règles de lexer pour reconnaître respectivement + et un nombre entier, cette grammaire permettra de reconnaître 42 ou 42+3+4 comme des sum_expr.

La syntaxe des membres droits des règles de parser est très similaire à celle des lexers. Bien sûr, les constructions comme '0'..'9' qui font référence aux caractères ne s'appliquent plus (puisque l'entrée n'est plus une séquence de caractères mais une séquence de jetons). Les constructions les plus utiles sont :

```
(expr1 | expr2) : reconnaît une séquence reconnue soit par expr1, soit par expr2. (expr)* : expr, répétée un nombre quelconque (possiblement nul) de fois. (expr)+ : expr, répétée au moins une fois. (expr)? : reconnaît soit \varepsilon (la chaîne vide), soit expr.
```

4.2 Principe de l'analyse syntaxique et structure du code généré

Le principe de la génération de code d'ANTLR est le suivant : le parser est généré dans une classe (nommée d'après le nom de la grammaire). Pour chaque non-terminal de la grammaire, une méthode est générée dans cette classe, et appeler cette méthode va reconnaître (et consommer) une séquence de jetons sur l'entrée du parser. Quand un terminal apparaît en partie droite de règle, le parser vérifie que le prochain jeton sur l'entrée est le bon, et consomme ce jeton. Quand un non-terminal apparaît, le parser fait un appel récursif sur la méthode correspondante à ce non-terminal. Par exemple, pour la grammaire suivante qui reconnaît les suites de jeton de la forme $A^nC^*B^nEOF$ (où EOF est le jeton spécial marquant la fin de fichier)

```
entree: boucle EOF ;
boucle: A boucle B | C* ;
```

le code généré ressemblera à celui donné ci-dessous (où les appels de méthodes enterOuterAlt et setState servent au parser à mettre-à-jour des structures de données internes)

```
public final ... entree() throws RecognitionException {
  enterOuterAlt(...);
    setState(...); boucle();
    setState(...); match(EOF);
  }
}
public final ... boucle() throws RecognitionException {
  switch (_input.LA(1)) {
    case A:
      enterOuterAlt(...);
      {
        setState(...); match(A);
        setState(...); boucle();
        setState(...); match(B);
      }
      break;
    case EOF:case B:case C:
      enterOuterAlt(...);
      {
        setState(...);
```

Sur cette grammaire, on peut en effet sélectionner l'alternative à appliquer en regardant uniquement le premier jeton non consommé de l'entrée (on dit « 1 jeton de Look-Ahead » d'où LA(1) dans le code). Ici, l'ensemble de jetons $\{EOF, B, C\}$ qui sélectionne l'alternative boucle $\rightarrow C*$ en regardant le premier jeton non consommé, s'appelle le directeur LL(1) de cette alternative. Le calcul des directeurs LL(1) a été étudié dans les cours de Théorie des Langages.

Une grammaire est LL(1) ssi les directeurs LL(1) des alternatives d'une même règle sont 2 à 2 disjoints : c'est bien le cas de la grammaire précédente. Cette propriété garantit d'une part la génération d'un parser efficace (linéaire en fonction du nombre de jetons), et d'autre part que la grammaire est non-ambiguë : à chaque suite de jetons en entrée, il y a (au plus) un seul arbre d'analyse associé (celui-ci correspond à l'arbre d'appels de méthodes du parser généré).

ANTLR4 accepte en fait n'importe quelle grammaire hors-contexte qui ne contient pas de récusion indirecte à gauche. Il accepte en particulier la grammaire des palindromes ci-dessous :

```
palind: A palind A | B palind B | A | B | /* epsilon */;
```

Sur une telle grammaire, l'algorithme de sélection des alternatives est plus complexe : il peut scanner l'ensemble de la suite de jetons et la pile des appels récursifs. L'analyseur généré invoque cet algorithme via la méthode getInterpreter().adaptivePredict qui prend la décision.

```
public final ... palind() throws RecognitionException {
  switch (getInterpreter().adaptivePredict(...)) {
    case 1:
      enterOuterAlt(...);
      {
        setState(...); match(A);
        setState(...); palind();
        setState(...); match(A);
      }
      break;
    case 2:
      enterOuterAlt(...);
      {
        setState(...); match(B);
        setState(...); palind();
        setState(...); match(B);
      }
      break;
    case 3:
      enterOuterAlt(...);
```

```
setState(...); match(A);
}
break;
...
}
```

Cette expressivité a toutefois un prix. Premièrement, ANTLR4 ne peut pas garantir la non-ambiguïté de la grammaire (mais l'analyseur généré peut détecter les ambiguïtés éventuelles sur une entrée donnée). Deuxièmement, l'analyseur généré par ANTLR peut être assez inefficace : par exemple, l'analyseur généré pour les palindromes est quadratique alors qu'on pourrait facilement écrire un analyseur linéaire à la main! Voir aussi la discussion sur assign_expr dans [Syntaxe].

Précisons que ces problèmes ne sont pas dûs à une mauvaise conception de ANTLR4 mais aux difficultés fondamentales du traitement des grammaires hors-contextes. Premièrement, la non-ambiguïté est une propriété indécidable des grammaires hors-contextes (il ne peut pas exister d'algorithme qui répond parfaitement à la question). Deuxièmement, il est impossible de générer automatiquement des analyseurs optimaux (pour un générateur d'analyseur donné, il existe une grammaire pour laquelle on connaît un meilleur analyseur que celui qui a été généré). Remarquons toutefois que d'autres outils (ANTLR3, YACC) font un compromis différent : ils ne savent traiter qu'un plus petit sous-ensemble de grammaires hors-contextes, mais garantissent la non-ambiguïté des grammaires qu'ils acceptent ainsi qu'un parsing efficace.

Dans le cadre du projet GL, les enseignants ont vérifié que la grammaire donnée dans [Syntaxe] est non-ambiguë et que l'analyseur généré est raisonnablement efficace.

4.3 Actions d'une règle de grammaire, attributs hérités et synthétisés

Comme pour les règles de lexer, on peut ajouter des actions, c'est-à-dire des portions de code Java entre accolades. On peut ajouter des actions n'importe où dans la règle, et le code de l'action sera inséré dans le code généré à l'endroit correspondant. Par exemple :

Vu que le code généré ne fait pas de backtrack, les actions peuvent contenir des effets de bord, et ce n'est pas un problème de les exécuter au fur et à mesure. Quand la grammaire n'est pas ambiguë, on peut comprendre sans se tromper l'ordre des actions en fonction du texte d'entrée.

On peut ajouter des attribut synthétisés aux non-terminaux en ajoutant derrière le nom de la règle la déclaration "returns [type1 nom1, type2 nom2]". Dans le code généré, les attributs synthétisés d'une règle appelée toto seront des champs d'un objet de type TotoContext retourné par la méthode toto. La valeur de l'attribut synthétisé nom1 est spécifiée dans la (ou les) action(s) de la règle, en affectant la valeur à renvoyer à la pseudo-variable \$nom1. Par exemple :

```
quarante_deux returns[int val] : INT { $val = 42; }

Cette valeur peut bien sûr être utilisée depuis d'autres règles. Par exemple :

expr returns[int val] :
    sum_expr { $val = $sum_expr.val; }
    ;

sum_expr returns[int val] :
    e=mult_expr {$val = $e.val;}
    (PLUS e2=mult_expr {$val = $val + $e2.val;})*
```

```
mult_expr returns[int val] :
    INTEGER {$val = Integer.parseInt($INTEGER.text);}
    :
```

On peut accéder à la valeur synthétisée par un symbole terminal ou non-terminal de deux manières : soit avec \$nom-symbole.nom-valeur-de-retour (par exemple, \$sum_expr.val), soit en donnant un nom à cette instance du symbole (par exemple, e=mult_expr pour donner le nom e), puis en utilisant ce nom au lieu du nom du symbole (par exemple, \$e.val, qui permet de faire référence au premier mult_expr de la règle sans ambiguïté).

Dans le cas d'un terminal comme INTEGER ci-dessus, la valeur synthétisée text est une chaîne de caractères correspondant au lexème concret lu par l'analyseur lexical lorsqu'il a généré le jeton INTEGER. Ainsi, ci-dessus, \$INTEGER.text est une chaîne de caractères qui représente un entier.

Une action peut lever explicitement une erreur de syntaxe en lançant l'exception RecognitionException (ou une classe qui en dérive). Sur notre exemple, il aurait fallu faire cela dans le cas où Integer.parseInt échoue).

On peut également ajouter des attributs hérités (qui correspondent à des arguments de la méthode générée) avec la syntaxe suivante :

```
sum_expr returns[int val] :
    mult_expr {$val = $mult_expr.val;}
    plus_mult_expr[$val] {$val = $plus_mult_expr.after; }
    ;

plus_mult_expr[int before] returns[int after] :
    PLUS mult_expr { $after = $before + $mult_expr.val; }
    ;
}
```

Pour déclarer des variables locales ayant comme portée l'ensemble de la règle et/ou faire des initialisations, ANTLR propose une section @init en début de règle :

```
expr returns[int val]
@init {
    int i;
    $val = 0;
}
    : other_expr { i = 42; $val = $val + 1; }
```

4.4 Attributs synthétisés prédéfinis de localisation

En dehors de l'attribut text déjà mentionné, chaque terminal possède d'autres attributs synthétisés pour connaître sa position dans le fichier d'entrée (line, position, etc.). Dans le cadre du projet, vous n'avez pas à priori besoin d'y accèder directement, mais plutôt au travers de la méthode setLocation définie dans la classe AbstractDecaParser (dont hérite la classe DecaParser générée par ANTLR).

```
protected void setLocation(Tree tree, Token token);
```

De plus, chaque non-terminal a un attribut synthétisé start qui désigne le premier jeton de la suite des jetons reconnue par l'analyseur. Le squelette d'analyseur fourni montre comment exploiter ceci pour associer une position dans le fichier d'entrée (c'est-à-dire un objet de type Location) à chaque nœud de l'arbre abstrait (cf. utilisations de « setLocation » dans DecaParser.g4).

4.5 Gestion des erreurs

Une erreur de syntaxe lance l'exception RecognitionException. Le reconnaisseur va lancer cette exception automatiquement si l'entrée ne satisfait pas la grammaire, ou bien on peut la lancer manuellement depuis une action. Par défaut, chaque règle rattrape cette exception avec la construction

ci-dessous où <u>errHandler</u> est un objet de type <u>ANTLRErrorStrategy</u> dont les méthodes <u>reportError</u> et <u>recover</u> décide respectivement comment rapporter les erreurs à l'utilisateur, et comment rattraper une erreur (c'est-à-dire comment tenter de continuer l'analyse en présence d'erreur).

```
try {
    // corps de la règle
}
catch (RecognitionException re) {
    _errHandler.reportError(this, re);
    _errHandler.recover(this, re);
}
```

Dans le cadre du projet GL, on utilise le gestionnaire d'erreur par défaut (appelé DefaultErrorStrategy) qui notifie un objet DecacErrorListner sur sa méthode syntaxError en cas d'erreur. Cette méthode se charge alors de stopper le mécanisme de rattrapage d'erreur par défaut, afin d'arrêter l'analyse syntaxique sur la première erreur de syntaxe.

Pour lever une exception dans les actions DecaParser.g4, vous devez utiliser une exception dérivant de la classe DecaRecognitionException. Le squelette fourni en donne un exemple, qui lève l'exception InvalidLValue lorsque le fils gauche d'une affectation dans la règle assign_expr n'est pas une lvalue.

5 Exemple de programme utilisant ANTLR : calculette

Un exemple minimaliste de programme utilisant ANTLR est donné dans le répertoire examples/calc/ de votre projet. C'est une « calculette » très simple, qui gère les nombres entiers, les opérateurs +, - et * (avec priorité entre les opérateurs). Le programme lit une expression (par exemple 1+2*3) sur son entrée standard, et affiche le résultat (par exemple, 1 + 2 * 3 = 7) sur sa sortie standard.

Le fichier CalcLexer.g4 (fig. 1) implémente le lexer de notre langage. Il lit des caractères sur son entrée, et produit une suite de jetons. Le fichier CalcParser.g4 (figs. 2 et 3) implémente le parser qui consomme ces jetons et produit un arbre abstrait (instance de la classe AbstractExpr en Java). Les classes générées par ces deux fichiers sont instanciées dans le fichier Main.java (fig. 4).

```
lexer grammar CalcLexer;
// fragment rules are used by other rules, but do not produce tokens:
fragment DIGIT : '0' .. '9';
// Actual rule that will produce a token INT when matching the regular
// expression "DIGIT+":
INT : DIGIT+;
PLUS : '+' ;
MINUS : '-';
TIMES : '*';
// Ignore spaces, tabs, newlines and whitespaces
       ( ' '
        | '\t'
        | '\r'
        | '\n'
        ) {
              skip(); // avoid producing a token
          }
    ;
```

FIGURE 1 - CalcLexer.g4 : lexer pour la calculette

```
parser grammar CalcParser;

options {
    // Use the vocabulary generated by the accompanying
    // lexer. Maven knows how to work out the relationship
    // between the lexer and parser and will build the
    // lexer before the parser. It will also rebuild the
    // parser if the lexer changes.
    tokenVocab = CalcLexer;
}

// Toplevel rule: matches an expression followed by the end of file
// (EOF). The "returns" declaration says which Java type is produced
// by this rule.
expr returns[AbstractExpr tree]
```

FIGURE 2 – CalcParser.g4 : parser pour la calculette (en-tête du fichier)

```
expr returns[AbstractExpr tree]
   : sum_expr EOF {
                   // between braces ({}), an action for this rule,
                   // i.e. a piece of Java code that will be executed
                   // while matching the rule. Must assign a value to
                   // $tree to produce the value declared in the
                   // returns clause above.
                   $tree = $sum_expr.tree;
           }
    ;
sum_expr returns [AbstractExpr tree]
   : e=mult_expr {
           $tree = $e.tree;
     (PLUS e2=mult_expr {
           // action inside (expr)* => will be executed once for each
           // match of "expr".
           $tree = new Plus($tree, $e2.tree);
       }
      | MINUS e2=mult_expr {
           $tree = new Minus($tree, $e2.tree);
       }
     ) *
    ;
mult_expr returns [AbstractExpr tree]
   : e=literal {
           $tree = $e.tree;
     (TIMES e2=literal \{
           $tree = new Times($tree, $e2.tree);
       }
     ) *
literal returns[IntLiteral tree]
    : INT {
           try {
               $tree = new IntLiteral(Integer.parseInt($INT.text));
           } catch (NumberFormatException e) {
               // The integer could not be parsed (probably it's too large).
               // set $tree to null, and then fail with the semantic predicate
               // management.
               $tree = null;
       } {$tree != null}?
```

FIGURE 3 - CalcParser.g4: parser pour la calculette (partie « règles » du fichier)

```
package calc;
// ANTLR generated code uses the ANTLR runtime
import org.antlr.v4.runtime.ANTLRInputStream;
import org.antlr.v4.runtime.CommonTokenStream;
 st Example program using ANTLR. This is a simple calculator that
 * manages integer values, +, - and * operators (with correct
 * management of operators precedence).
public class Main {
    public static void main(String[] args) throws Exception {
        System.out.println("Enter expression (end with Ctrl-d):");
        // Instantiate lexer and parser, connected together:
        CalcLexer lexer =
            new CalcLexer(new ANTLRInputStream(System.in));
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        CalcParser parser = new CalcParser(tokens);
        // Launch the parser
        AbstractExpr expression = parser.expr().tree;
        if (parser.getNumberOfSyntaxErrors() > 0) {
            System.out.println("Cannot compute expression because of syntax error (;
        } else {
            // Display the result (e.g. Result: 1 + 2 * 3 = 7)
            System.out.println("Result : "
                    + expression.toString() + " = "
                    + expression.value());
        }
    }
}
```

FIGURE 4 - Main. java : Programme Java qui instancie lexer et parser

Arbres enrichis et décorés

L'étape d'analyse syntaxique produit un arbre abstrait primitif (voir [SyntaxeAbstraite]); l'étape de vérifications contextuelles enrichit cet arbre et le décore à l'aide d'informations contextuelles correspondant à certains attributs des grammaires attribuées de Deca (voir [SyntaxeContextuelle]).

1 Enrichissement avec le nœud ConvFloat

Ce nœud indique qu'il y a une conversion du type entier au type flottant Deca.

```
Exemple Dans le contexte de déclarations
   int i;
   float f;
l'instruction
   f = f + i;
est représentée par l'arbre primitif
Assign [
  Identifier \f"f"
  Plus [
     Identifier ↑"i"
  ]
1
Après enrichissement, l'arbre devient :
Assign [
  \underline{\texttt{Identifier}} {\uparrow} \texttt{"f"}
  Plus [
     ConvFloat [
       Identifier | | '"i"
     ]
  ]
]
```

2 Les décors

Un décor correspond à certains attributs des grammaires attribuées qui sont mémorisés dans l'arbre abstrait, afin de faciliter l'étape de génération de code (cf. [Gencode]). Une illustration détaillée sur un exemple est présenté dans [Exemple].

Il y a deux types de décorations :

- 'Definition', qui représente la définition d'un identificateur (de type, de méthode, de variable, etc.), qui est rattaché à tous les nœuds <u>Identifier</u>.
- 'Type', qui représente le type d'une expression, qui est rattaché à tous les nœuds racines de sousarbres de la classe définie par EXPR dans la grammaire d'arbres (de [SyntaxeAbstraite]).
 N.B. 'Type' sera rattaché aux nœuds <u>Identifier</u> correspondant à une occurrence d'utilisation d'identificateur dans une expression, mais n'est pas nécessaire pour une définition d'identificateur.

En Passe 1, on crée les Definition des classes : ce sont des objets de type ClassDefinition.

En Passe 2, on crée les Definition des champs et des méthodes avec leur index (cf. paramètre des constructeurs), et on met à jour les champs numberOfFields et numberOfMethods (via les mutateurs appropriés) de la définition associée à leur classe. Le rôle de ces champs index dans FieldDefinition et dans MethodDefinition est détaillé dans [Gencode].

En Passe 3, on crée les autres Definition.

On utilisera au maximum la « sémantique de partage » : les objets de type Definition et Type sont partagés autant que possible. Par exemple, les nœuds <u>Identifier</u> correspondant à la même définition ont tous la même Definition et le même Type. Ceci est utilisé par [Gencode].

3 Un exemple complet

Programme source

```
class A {
1
       protected int x ;
2
3
       int getX() {
          return x ;
       void setX(int x) {
6
          this.x = x;
7
8
    }
9
10
    {
11
       A = new A();
12
       a.setX(1);
13
       println("a.getX() = ", a.getX());
14
    }
15
```

Arbre décoré

```
`> [1, 0] Program
  +> ListDeclClass [List with 1 elements]
     []> [1, 0] DeclClass
         +> [1, 6] Identifier (A)
          | definition: type defined at [1, 0], type=A
          +> [builtin] Identifier (Object)
          | definition: type (builtin), type=Object
          +> ListDeclField [List with 1 elements]
             [] > [2, 17] [visibility=PROTECTED] DeclField
                 +> [2, 13] Identifier (int)
                 | definition: type (builtin), type=int
                 +> [2, 17] Identifier (x)
                 | definition: field defined at [2, 17], type=int
          1
                 `> NoInitialization
          `> ListDeclMethod [List with 2 elements]
             [] > [3, 3] DeclMethod
             || +> [3, 3] Identifier (int)
             || | definition: type (builtin), type=int
             || +> [3, 7] Identifier (getX)
             П
               | definition: method defined at [3, 3], type=int
             \Pi
                +> ListDeclParam [List with 0 elements]
                `> [3, 14] MethodBody
             \Pi
             \Pi
                   +> ListDeclVar [List with 0 elements]
             \Pi
                    `> ListInst [List with 1 elements]
             \Pi
                       [] > [4, 6] Return
             \Pi
                           `> [4, 13] Identifier (x)
             definition: field defined at [2, 17], type=int
             [] > [6, 3] DeclMethod
                 +> [6, 3] Identifier (void)
                 | definition: type (builtin), type=void
                 +> [6, 8] Identifier (setX)
                 | definition: method defined at [6, 3], type=void
                 +> ListDeclParam [List with 1 elements]
                 | []> [6, 13] DeclParam
                        +> [6, 13] Identifier (int)
                        | definition: type (builtin), type=int
                        `> [6, 17] Identifier (x)
                           definition: parameter defined at [6, 13], type=int
                 '> [6, 20] MethodBody
                    +> ListDeclVar [List with 0 elements]
                    `> ListInst [List with 1 elements]
                       []> [7, 13] Assign
                           type: int
                           +> [7, 10] Selection
                           | type: int
                           | +> [7, 6] This
                             | type: A
                              `> [7, 11] Identifier (x)
                                 definition: field defined at [2, 17], type=int
                           `> [7, 15] Identifier (x)
                              definition: parameter defined at [6, 13], type=int
```

```
`> [11, 0] Main
  +> ListDeclVar [List with 1 elements]
     [] > [12, 5] DeclVar
         +> [12, 3] Identifier (A)
         | definition: type defined at [1, 0], type=A
         +> [12, 5] Identifier (a)
         | definition: variable defined at [12, 5], type=A
         `> [12, 7] Initialization
            `> [12, 9] New
                type: A
                > [12, 13] Identifier (A)
                  definition: type defined at [1, 0], type=A
   `> ListInst [List with 2 elements]
      [] > [13, 9] MethodCall
     || type: void
     || +> [13, 3] Identifier (a)
     || | definition: variable defined at [12, 5], type=A
         +> [13, 5] Identifier (setX)
         | definition: method defined at [6, 3], type=void
      | |
         `> ListExpr [List with 1 elements]
     | |
             []> [13, 10] Int (1)
      | |
                 type: int
      []> [14, 3] Println
          `> ListExpr [List with 2 elements]
             []> [14, 11] StringLiteral (a.getX() = )
             || type: string
             [] > [14, 32] MethodCall
                type: int
                 +> [14, 26] Identifier (a)
                 | definition: variable defined at [12, 5], type=A
                 +> [14, 28] Identifier (getX)
                 | definition: method defined at [3, 3], type=int
                 `> ListExpr [List with 0 elements]
```

Génération de code pour le langage Deca

1 Génération de code pour le langage Deca « sans objet »

La génération de code pour un programme Deca « sans objet » peut être réalisée en une passe (d'autres passes peuvent être ajoutées pour optimiser le code généré). La forme générale du code généré à partir d'un programme Deca peut être la suivante (celle-ci peut aussi être généralisée pour générer du code objet, cf. section 2.2) :

```
; Début du programme principal
  TSTO #d1; taille maximale de la pile
  BOV pile_pleine
  ADDSP #d2; variables globales
; Code du programme principal
  HALT
; Messages d'erreurs
pile_pleine:
  WSTR "Erreur: ..."; Message d'erreur adéquat
  WNL
  ERROR
; Autres messages d'erreurs
```

Comme cela apparaît dans ce squelette, il est nécessaire de générer du code qui teste le débordement de la pile via l'instruction TSTO. Un tel débordement est en effet possible avec un programme sans objet utilisant trop de variables et de temporaires : voir l'état de la pile des programmes sans objet en section 4 du document [ExempleSansObjet]. Des explications sur la façon de générer correctement cette instruction TSTO sont données en section 2.2.

La passe de génération de code commence par la liste des déclarations de variables pour associer une adresse dans la pile à chaque variable globale. Les adresses des variables globales sont de la forme 1(GB), 2(GB), 3(GB).... Associer une adresse à chaque variable consiste à modifier le champ operand de sa définition via la méthode setOperand: voir les classes VariableDefinition et ExpDefinition dans le code fourni. Cette adresse pourra être récupérée via la méthode getOperand dans les phases ultérieures de la génération de code (génération de code des expressions, génération de code des affectations), grâce au partage des objets de type Definition dans l'arbre abstrait décoré (voir document [ArbreEnrichi]). Pendant ce passage sur les déclarations de variables, il faut aussi générer le code des initialisations explicites de variables (voir section 6).

Ensuite, la passe de génération traite la liste des instructions, dont les expressions arithmétiques (voir section 7.1), les expressions booléennes (voir section 7.2) et les structures de contrôles (voir section 8).

2 Génération de code pour le langage Deca complet

La génération de code pour un programme Deca peut être réalisée en deux passes (d'autres passes peuvent être ajoutées à ces deux passes pour optimiser le code généré). Cette section détaille le rôle de ces 2 passes et la forme générale du code généré. Le détail des différentes composantes de la génération de code est donné à partir de la section 3.

2.1 Rôle de chaque passe

Passe 1

La première passe consiste à parcourir les classes du programme en s'intéressant à leurs méthodes, de façon à générer du code pour construire la *table des méthodes* (« virtual methods table », souvent abbrégé « vtable » en anglais). Cette première passe comporte plus précisément les points suivants :

- construction du tableau des étiquettes des méthodes;
- génération de code permettant de construire la table des méthodes.

Passe 2

La deuxième passe comporte les points suivants :

- codage des champs de chaque classe (initialisation);
- codage des méthodes de chaque classe (déclarations et instructions);
- codage du programme principal (déclarations et instructions).

Cela nécessite de coder les expressions du langage Deca.

2.2 Forme générale du code

La forme générale du code généré à partir d'un programme Deca peut être la suivante :

- ; Partie principale :
- ; Code de la construction de la table des méthodes de chaque classe
- ; Code du programme principal

HALT

- ; Messages d'erreurs
- ; Pour chaque classe:
- ; code de l'initialisation des champs
- ; code des méthodes

Dans la suite, le mot « bloc » désigne soit le programme principal, soit une méthode, soit un sous-programme d'initialisation.

Il est nécessaire de générer du code qui teste le débordement de la pile. Plutôt que d'effectuer un test de débordement (instruction TSTO #d) chaque fois qu'on effectue un empilement, il est conseillé, pour produire du code plus efficace, de ne générer qu'une seule instruction TSTO #d par bloc.

Comme l'argument d du TSTO n'est connu qu'à la fin de la génération de code du bloc, alors que l'instruction TSTO doit être placée parmi les premières instructions du bloc, on utilisera la possibilité offerte par le paquetage pseudocode d'ajouter une instruction en début de bloc. L'instruction TSTO pourra ainsi être ajoutée après avoir généré le bloc de code correspondant, en utilisant les informations récoltées pendant la génération du bloc pour générer l'opérande correct.

Au cours de la génération de code d'un bloc, on calculera donc le nombre maximal d d'empilements nécessaires. Ce nombre dépend :

- du nombre de registres sauvegardés en début de bloc;
- du nombre de variables du bloc;
- du nombre maximal de temporaires nécessaires à l'évaluation des expressions;
- du nombre maximal de paramètres des méthodes appelées (chaque instruction BSR effectuant deux empilements).

Pour calculer ces informations « au vol », on utilisera un paquetage qui fournira des primitives utilisées de façon systématique lors de la génération de code.

2.3 Format de la partie principale

Le format du code construisant les tables des méthodes, du programme principal et des messages d'erreurs est le suivant.

```
TSTO \#d_1; taille maximale de la pile
BOV pile_pleine
ADDSP \#d_2; tables des méthodes + variables globales
; Code de la construction de la table des méthodes de chaque classe
; Code du programme principal
HALT

pile_pleine:
    WSTR "Erreur : ..."; Message d'erreur adéquat
    WNL
    ERROR
; Autres messages d'erreurs
```

2.4 Format d'une méthode

Le format d'une méthode m dans une classe A est le suivant.

```
 \begin{array}{l} \operatorname{code}.A.m \colon \\ \operatorname{BOV}\ \mathsf{pile\_pleine} \\ \operatorname{ADDSP}\ \mathsf{\#}d_1\ ; \ \ \mathsf{taille}\ \mathsf{maximale}\ \mathsf{de}\ \mathsf{la}\ \mathsf{pile} \\ \operatorname{ADDSP}\ \mathsf{\#}d_2\ ; \ \ \mathsf{variables}\ \mathsf{locales} \\ ; \ \ \mathit{Code}\ \mathsf{de}\ \mathsf{la}\ \mathsf{sauvegarde}\ \mathsf{des}\ \mathsf{registres} \\ ; \ \ \mathit{Code}\ \mathsf{de}\ \mathsf{la}\ \mathsf{m\'ethode} \\ ; \ \ \mathit{Message}\ \mathsf{d'erreur}\ \mathsf{si}\ \mathsf{on}\ \mathsf{sort}\ \mathsf{d'une}\ \mathsf{m\'ethode}\ \mathsf{retournant}\ \mathsf{un}\ \mathsf{r\'esultat}\ \mathsf{de}\ \mathsf{type} \\ ; \ \ \ \mathsf{diff\'erent}\ \mathsf{de}\ \mathsf{void}\ \mathsf{autrement}\ \mathsf{que}\ \mathsf{par}\ \mathsf{une}\ \mathsf{instruction}\ \mathsf{return} \\ \mathsf{ERROR} \\ \mathsf{fin}.A.m \colon \\ ; \ \ \mathit{Code}\ \mathsf{de}\ \mathsf{la}\ \mathsf{restauration}\ \mathsf{des}\ \mathsf{registres} \\ \mathsf{RTS} \\ \end{array}
```

2.5 Format d'un sous-programme d'initialisation des champs

Le format du sous-programme d'initialisation des champs d'une classe A est le suivant.

```
init.A:

TSTO #d_1; taille maximale de la pile

BOV pile_pleine

; Code de la sauvegarde des registres

; Code de l'initialisation des champs

; Code de la restauration des registres

RTS
```

3 Construction de la table des méthodes

À chaque classe du programme est associée une table des m'ethodes, qui est une suite de mots de la pile qui contient :

- un pointeur sur la table des méthodes de la super-classe;
- un pointeur vers le code de chaque méthode de la classe.

Considérons par exemple le programme suivant :

```
class A {
    void m() { }
    void p() { }
}
class B extends A {
    void p() { }
    void q() { }
}
```

La table des méthodes de la classe A est représenté figure 1.

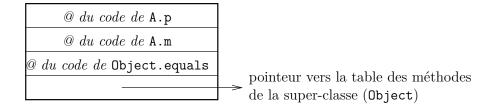


FIGURE 1 – Table des méthodes de la classe A

On stocke ces tables des méthodes dans la pile. L'état de la pile après la construction des tables des méthodes pour les classes Object, A et B est représenté figure 2. On choisit d'associer à une méthode t d'une classe C l'étiquette code.C.t qui débutera le code de cette méthode. Par exemple, code.A.m représente l'étiquette de début du code de la méthode m définie dans la classe A.

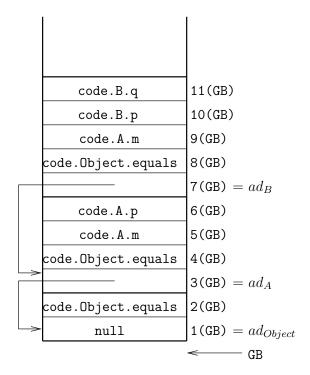


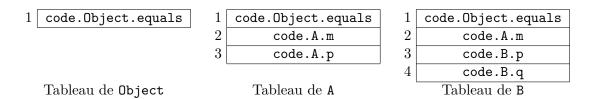
FIGURE 2 - État de la pile après la construction des tables des méthodes de Object, A et B

L'adresse de la table des méthodes de la classe Object est $ad_{Object} = 1$ (GB), celle de A est $ad_A =$

```
3(GB), et celle de B est ad_B = 7(GB).
```

Pour permettre de construire cette structure, on associe à chaque classe un tableau des étiquettes de méthodes qui associe à la *i*-ème méthode de chaque classe (en considérant également les méthodes héritées) l'étiquette de son code.

Les tableaux associés aux classes Object, A et B sont les suivants :



Lors d'un héritage ou d'une redéfinition, les deux méthodes doivent avoir le même indice dans les deux tableaux. Par exemple, A.m et B.m (héritée de A.m) sont à l'indice 2, et de même, A.p et B.p (redéfinition de A.p) sont à l'indice 3. L'indice d'une méthode correspond à l'index de la méthode, qui a été calculé et rangé dans sa MethodDefinition au cours de l'étape B (vérification/décoration).

Le code correspondant à la construction des tables des méthodes pour Object et A est le suivant.

```
; Code de la table des méthodes de Object
LOAD #null, RO
STORE RO, 1(GB)
LOAD code.Object.equals, RO
STORE RO, 2(GB)
; Code de la table des méthodes de A
LEA 1(GB), RO
STORE RO, 3(GB)
LOAD code.Object.equals, RO ; Héritage de la méthode equals de Object
STORE RO, 4(GB)
LOAD code.A.m, RO
STORE RO, 5(GB)
LOAD code.A.p, RO
STORE RO, 6(GB)
; Code de la table des méthodes de B
LEA 3(GB), RO
STORE RO, 7(GB)
LOAD code.Object.equals, RO
                             ; Héritage de la méthode equals de Object
STORE RO, 8(GB)
LOAD code.A.m, RO ; Héritage de la méthode m de A
STORE RO, 9(GB)
                   ; Redéfinition de la méthode p
LOAD code.B.p, RO
STORE RO, 10(GB)
LOAD code.B.q, RO
STORE RO, 11(GB)
```

L'adresse de la table des méthodes doit être stockée en mémoire de l'exécutable du compilateur, de manière à pouvoir être retrouvée facilement lors des phases ultérieures de la génération de code (notamment les appels de méthodes), à partir de la ClassDefinition de la classe. Sur l'exemple, ces adresses sont 1(GB) pour Object, 3(GB) pour A et 7(GB) pour B.

4 Codage des champs

4.1 Représentation d'un objet

Considérons la classe C suivante.

```
class C {
   int x;
   int y=1;
   int getX() {
      return x;
   }
   void incrX() {
      x = x + 1;
   }
}
```

Un objet de la classe C est représenté par une structure allouée dans le tas contenant :

- un pointeur sur la table des méthodes de C;
- une case mémoire pour le champ x;
- une case mémoire pour le champ y.

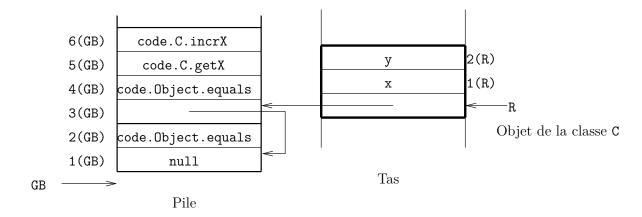


FIGURE 3 – Représentation mémoire d'un objet de la classe C

La figure 3 représente un objet de la classe C. Cet objet est alloué dans le tas. Si son adresse est dans le registre R, alors O(R) dénote l'adresse du pointeur sur la table des méthodes de C, O(R) dénote l'adresse du champ O(R) dénote l'adresse du champ O(R) dénote l'adresse du champ O(R) denote l'adresse du

En plus de la classe C définie précédemment, on définit la classe D de la façon suivante.

```
class D extends C {
  int x=3;
  int z;
}
```

La classe D hérite donc des champs x et y de C. La figure $\frac{4}{2}$ représente un objet de la classe D. On peut remarquer que la déclaration int x; dans D définit un nouveau champ x et que la place du champ x de C est conservée dans la structure.

Pour chaque champ, son déplacement (qui correspond à sa position dans la structure de l'objet) est exactement son index, qui a été rangé dans sa FieldDefinition lors de l'étape B (vérification/décoration). La taille de l'objet est, elle, directement reliée au nombre de champs, rangé dans le champ numberOfFields de la ClassDefinition de sa classe en étape B.

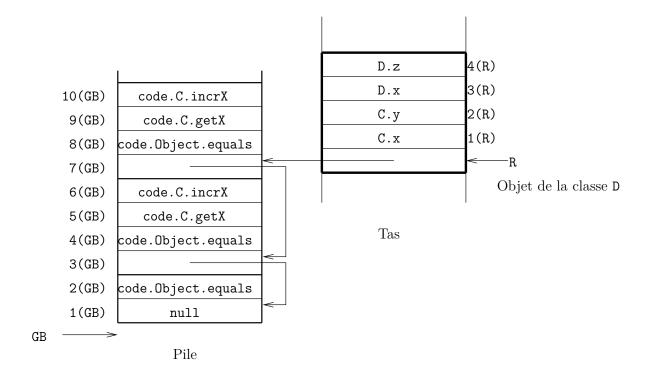


FIGURE 4 – Représentation mémoire d'un objet de la classe D

4.2 Code d'une sélection de champ

c.x = 2; print(c.x);

> WNL ERROR

Soit c un objet de la classe C précédente, supposons que l'adresse de c est rangée en 42(LB). Soit l'extrait de programme suivant :

```
Le code correspondant est le suivant :
; Affectation
      LOAD 42(LB), R2
                                        ; objet null dans sélection de champ?
      CMP #null, R2
      BEQ dereferencement.null
      LOAD #2, R3
      STORE R3, 1(R2)
; Instruction print
      LOAD 42(LB), R2
      CMP #null, R2
                                        ; objet null dans sélection de champ?
      BEQ dereferencement.null
      LOAD 1(R2), R2
      LOAD R2, R1
      WINT
   ; [...]
     Message d'erreur en cas de déréférencement de null
   dereferencement.null:
```

On remarque qu'il faut tester si l'objet qui est en partie gauche de la sélection est null.

WSTR "Erreur : dereferencement de null"

On peut également noter que l'instruction CMP #null, R2 est ici superflue parce que les codes conditions correspondant à cette comparaison sont déjà positionnés par l'instruction LOAD qui précède. Néanmoins, son utilisation peut éventuellement permettre de simplifier la génération automatique de code en évitant de distinguer des cas particuliers.

4.3 Initialisation des champs

Lorsqu'un nouvel objet est construit, ses champs doivent être initialisés, soit à la valeur zéro, faux ou null, si la déclaration du champ ne comporte pas d'initialisation explicite, soit à la valeur spécifiée, si la déclaration comporte une initialisation explicite. De plus, les champs hérités devront être initialisés comme cela est spécifié dans la super-classe.

L'initialisation des champs va être codée par un sous-programme. On choisit d'associer au sous-programme d'initialisation d'une classe C l'étiquette init.C. L'objet à initialiser devra être passé en paramètre à ce sous-programme et doit donc être empilé avant l'appel. Ce paramètre a donc -2(LB) comme adresse.

Par exemple, l'initialisation d'un objet de la classe C pourra être réalisée par le code suivant :

```
; Initialisation des champs de C
init.C :
    ; Initialisation de x
LOAD #0, R0
LOAD -2(LB), R1 ; R1 contient l'adresse de l'objet
STORE R0, 1(R1) ; 1(R1) est l'adresse de x
    ; Initialisation de y
LOAD #1, R0
LOAD -2(LB), R1 ; R1 contient l'adresse de l'objet
STORE R0, 2(R1) ; 2(R1) est l'adresse de y
RTS
```

Si le code qui réalise les initialisations utilise des registres parmi R2, R3, ..., ces registres devront être sauvegardés au début du sous-programme, et restaurés avant la sortie du sous-programme.

L'état de la pile lors de l'exécution du sous-programme d'initialisation init.C est représenté figure 5.

Pour initialiser un objet d'une sous-classe, on commence par initialiser les champs hérités, puis on initialise les champs propres de la classe. Ceci dit, si les champs hérités sont initialisés par des appels de méthodes qui sont elles-mêmes redéfinies dans la sous-classe alors l'initialisation des champs hérités peut indirectement accéder/modifier ceux de la sous-classe. Pour être sûr de respecter la sémantique, il est donc plus prudent de commencer par mettre tous les nouveaux champs à 0, avant l'initialisation des champs hérités, et de finir par les éventuelles initialisations explicites des nouveaux champs. Par exemple, l'initialisation d'un objet de la classe D pourra être réalisée par le code suivant :

```
; Initialisation des champs de D
init.D:
TSTO #3 ; Test de débordement de pile
BOV pile_pleine
LOAD -2(LB), R1 ; R1 contient l'adresse de l'objet
; Initialisation de D.x
LOAD #0, R0
STORE R0, 3(R1) ; 3(R1) est l'adresse de D.x
; Initialisation de D.z
STORE R0, 4(R1) ; 4(R1) est l'adresse de D.z
```

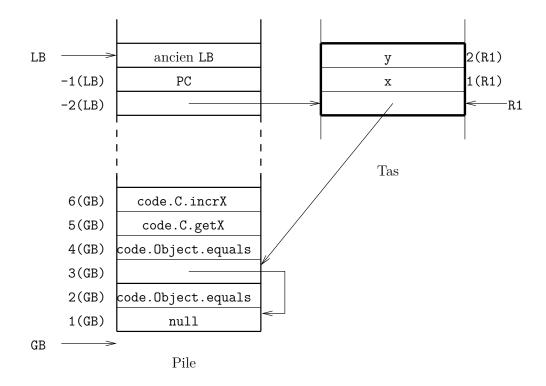


FIGURE 5 – État de la pile lors de l'appel de init.C

```
; Appel de l'initialisation des champs hérités de C
PUSH R1 ; empile l'objet à initialiser
BSR init.C ; appel de l'initialisation de la super-classe
SUBSP #1 ; on remet la pile dans son état initial
; initialisation explicite de D.x
LOAD -2(LB), R1 ; R1 contient l'adresse de l'objet
LOAD #3, R0
STORE R0, 3(R1) ; 3(R1) est l'adresse de D.x
RTS
```

5 Codage des méthodes

Considérons par exemple la classe suivante :

```
class P {
   int x;
   int y;
   void move(int a, int b) {
      x = x + a;
      y = y + b;
   }
}
```

L'état de la pile lors d'un appel à la méthode move est représenté figure 6. Le paramètre implicite de la méthode (l'objet qui invoque la méthode) a donc pour adresse -2(LB), le premier paramètre a a pour adresse -3(LB), et le deuxième paramètre b a pour adresse -4(LB). On remarque donc que les paramètres doivent être empilés de *droite à gauche*.

Le code de la méthode move a la forme suivante.

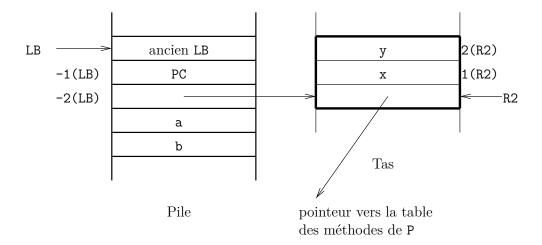


FIGURE 6 – État de la pile lors de l'appel de move

```
Code de la méthode move
code.P.move :
   TSTO #2
                     ; Test de débordement de pile
   BOV pile_pleine
   ; Sauvegarde des registres
   PUSH R2
   PUSH R3
   ; x = x + a;
   LOAD -2(LB), R2
                     ; R2 := this
   LOAD - 2(LB), R3
                    ; R3 := this
   LOAD 1(R3), R3
                       R3 := this.x
                     ; R3 := this.x + a
   ADD -3(LB), R3
   STORE R3, 1(R2)
                     ; this.x := R3
   ; y = y + b;
                    ; R2 := this
   LOAD -2(LB), R2
   LOAD -2(LB), R3
                     ; R3 := this
   LOAD 2(R3), R3
                     ; R3 := this.y
   ADD -4(LB), R3
                       R3 := this.y + b
   STORE R3, 2(R2)
                     ; this.y := R3
fin.P.move :
   ; Restauration des registres
   POP R3
   POP R2
   RTS
```

Les registres utilisés dans le corps de la méthode doivent être sauvegardés au début du code de la méthode, et restaurés à la fin. On introduit une étiquette pour la fin du code d'une méthode. Le codage d'une instruction return effectue un branchement sur cette étiquette.

Considérons la classe Q suivante.

```
class Q {
   float abs(float x) {
      if (x < 0.0) {
         return -x;
      }
   }</pre>
```

}

La fonction abs est censée calculer la valeur absolue de son argument, mais le cas $x \ge 0$ a été oublié.

Le paramètre explicite x de abs a pour adresse -3(LB). L'instruction return effectue un branchement à la fin de la méthode. Le résultat de la méthode abs est dans le registre RO. Le code de la méthode a la forme suivante.

```
; Code de la méthode abs
code.Q.abs :
    ; Instruction if
    LOAD -3(LB), RO
    CMP #0.0, RO
    BGE etiq_fin.1
    ; return -x
    OPP -3(LB), RO
    BRA fin.Q.abs
etiq_fin.1 :
    WSTR "Erreur : sortie de la methode Q.abs sans return"
    WNL
    ERROR
fin.Q.abs :
    RTS
```

On peut remarquer que si on sort de la fonction sans être passé par une instruction return, il y a une erreur à l'exécution.

6 Codage des déclarations

Le codage des déclarations consiste à

- 1. associer une adresse à chaque variable. Les variables locales aux méthodes ont des adresses de la forme 1(LB), 2(LB), 3(LB).... Les variables globales ont des adresses de la forme k(GB).... (la plus petite valeur de k dépend de la taille des tables de méthodes des classes).
- 2. coder les initialisations explicites de variables, en utilisant le codage des expressions. Les variables qui n'ont pas d'initialisation explicite sont laissées non initialisées.

7 Codage des expressions

7.1 Codage des expressions arithmétiques

Le langage Deca impose que les expressions binaires soient évaluées de gauche à droite. Pour évaluer une expression binaire de la forme e_1 op e_2 , on évalue donc e_1 , on conserve sa valeur dans un registre, puis on évalue e_2 et on applique enfin l'opérateur op. Si l'expression est complexe, on a donc besoin de plusieurs registres pour conserver les sous-expressions intermédiaires. Dans certains cas, on ne dispose pas de suffisamment de registres, il faut alors allouer des variables temporaires.

Des appels de méthodes peuvent avoir lieu au cours de l'évaluation de l'expression. Par conséquent, il ne faut pas conserver de valeur intermédiaire dans les registres R0 et R1, dont la valeur peut être modifiée par ces appels (ce sont des registres « scratch »). On utilise donc les registres R2, R3... pour évaluer les sous-expressions.

On considère par exemple le programme suivant.

```
{
  int x = 1;
  int y = 2;
  x = 3 * x - y / 2;
}
```

Dans ce programme, x a pour adresse 3(GB) et y a pour adresse 4(GB). Le code correspondant à l'instruction 3 * x - y / 2 a la forme suivante :

```
LOAD #3, R2 ; R2 := 3 MUL 3(GB), R2 ; R2 := 3 * x LOAD 4(GB), R3 ; R3 := y QUO #2, R3 ; R3 := y / 2 SUB R3, R2 ; R2 := R2 - R3 STORE R2, 3(GB)
```

Si on ne dispose que du registre R2, on doit utiliser une variable temporaire pour stocker 3 * x. La sauvegarde et la restauration d'une valeur peuvent être faites à l'aide des instructions PUSH et POP. Cela donne le code suivant :

Dans le cadre de l'extension TRIGO, pour améliorer la précision de la classe Math, il peut être intéressant d'adapter l'algorithme esquissé ci-dessus pour utiliser l'instruction FMA. Voir section 5 du document [Consignes].

7.2 Codage des expressions booléennes

En Deca, les opérateurs booléens && et | | sont évalués paresseusement de gauche à droite. Cela signifie que pour évaluer e_1 && e_2 , on évalue d'abord e_1 . Ensuite, si e_1 est faux, le résultat est faux (e_2 n'est pas évalué). Si e_1 est vrai, alors e_2 est évalué, et le résultat vaut e_2 .

On code les expressions booléennes par des *flots de contrôle*. Avec cette approche, on considère qu'une expression booléenne correspond à une suite de lignes de code comportant un ou plusieurs branchements à une certaine étiquette E (cf. figure 7).

Les deux valeurs booléennes correspondent

- au branchement à l'étiquette E (rupture de séquence);
- à la poursuite des instructions (continuation en séquence).

On a deux possibilités:

- 1. Le branchement est effectué si l'expression booléenne C est vraie. Le code correspondant est noté $\langle Code(C, vrai, E) \rangle$.
- 2. Le branchement est effectué lorsque l'expression booléenne C est fausse. Le code correspondant est noté $\langle Code(C, faux, E) \rangle$.

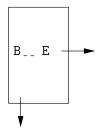


FIGURE 7 – Une expression booléenne codée par un flot de contrôle

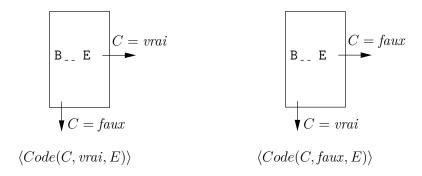


Figure 8 – Codes possibles pour une expression booléenne C

Un intérêt de cette approche est que l'opérateur unaire de négation booléenne **ne génère pas** pas de code additionnel par rapport à son fils. Ainsi, lorsque C est une expression booléenne,

$$\langle Code(!C, vrai, E) \rangle \equiv \langle Code(C, faux, E) \rangle$$
 $\langle Code(!C, faux, E) \rangle \equiv \langle Code(C, vrai, E) \rangle$

Autrement dit, il y a élimination des négations à la volée : celles-ci sont traitées directement aux feuilles de l'arbre abstrait.

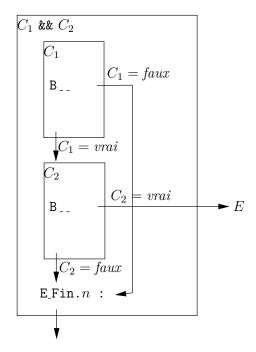
Détaillons le code à produire pour les autres expressions booléennes possibles.

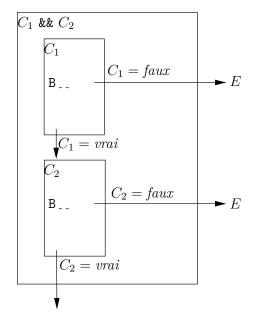
```
 \langle Code(\mathsf{true}, \mathit{vrai}, E) \rangle \equiv \mathsf{BRA} \; \mathsf{E} \qquad \langle Code(\mathsf{true}, \mathit{faux}, E) \rangle \equiv \varepsilon \; \; (\mathsf{pas} \; \mathsf{de} \; \mathsf{code})   \langle Code(\mathsf{false}, b, E) \rangle \equiv \langle Code(!\mathsf{true}, b, E) \rangle   \langle Code(C_1 \; \&\& \; C_2, \mathit{vrai}, E) \rangle \equiv \langle Code(C_1 \; \&\& \; C_2, \mathit{faux}, E) \rangle \equiv \langle Code(C_1, \mathit{faux}, E) \rangle   \langle Code(C_1, \mathit{faux}, E) \rangle   \langle Code(C_2, \mathit{vrai}, E) \rangle   \langle Code(C_2, \mathit{faux}, E) \rangle   \langle Code(C_1 \; | \; | \; C_2, b, E) \rangle \equiv \langle Code(!(!C_1 \; \&\& \; !C_2)), b, E) \rangle
```

Remarquons que le code produit par $\langle Code(\mathtt{false} \ \&\& \ C, \ b, \ E) \rangle$ est efficace à l'exécution (un seul branchement), par contre il contient du code "mort" (le code pour C). Ce code mort peut éventuellement être éliminé lors d'une passe ultérieure sur le code assembleur.

Pour les identificateurs il est nécessaire de stocker leur valeur en mémoire. On suppose que la valeur faux est codée par #0. On note @idf l'adresse de l'identificateur idf.

```
 \begin{split} \langle Code(idf,\,vrai,\,E)\rangle \equiv & & \langle Code(idf,\,faux,\,E)\rangle \equiv \\ \text{LOAD @}idf,\,\,\text{RO} & \text{LOAD @}idf,\,\,\text{RO} \\ \text{CMP #O, RO} & \text{CMP #O, RO} \\ \text{BNE }E & \text{BEQ }E \end{split}
```





```
FIGURE 9 – \langle Code(C_1 \&\& C_2, vrai, E) \rangle
```

FIGURE $10 - \langle Code(C_1 \&\& C_2, faux, E) \rangle$

Pour les opérateurs de comparaison (==, !=, <, <=...), il est nécessaire d'évaluer la partie gauche, puis la partie droite, de générer une instruction CMP, et enfin d'effectuer un branchement conditionnel.

Considérons par exemple le programme suivant.

```
{
  int x;
  x = readInt();
  if (x >= 0 && x <= 5) {
    print("x est dans l'intervalle [0, 5]");
  } else {
    print("x n'est pas dans l'intervalle [0, 5]");
  }
}</pre>
```

L'adresse de x est 3(GB). Le code de l'instruction if est le suivant (on applique ici le codage décrit en section 8.1).

```
LOAD 3(GB), R2
CMP #0, R2
BLT E_Sinon.1
LOAD 3(GB), R2
CMP #5, R2
BGT E_Sinon.1
; Instructions alors
WSTR "x est dans l'intervalle [0, 5]"
BRA E_Fin.1
E_Sinon.1:
; Instructions sinon
WSTR "x n'est pas dans l'intervalle [0, 5]"
E_Fin.1:
```

7.3 Codage des appels de méthodes

Le langage Deca impose que les arguments d'une méthode soient évalués de gauche à droite, en commençant par le paramètre implicite, et soient empilés de droite à gauche, en terminant par le paramètre implicite.

On considère par exemple l'instruction p.move(1, 2). On suppose que p a pour adresse 42(GB) et que move est la première méthode de classe P. Le code obtenu peut être le suivant :

```
ADDSP #3
                      ; On réserve de la place pour les trois paramètres
   LOAD 42(GB), R2
   STORE R2, O(SP)
                     ; On empile le paramètre implicite (p)
   LOAD #1, R2
   STORE R2, -1(SP); On empile le premier paramètre (1)
   LOAD #2, R2
   STORE R2, -2(SP); On empile le deuxième paramètre (2)
   LOAD O(SP), R2
                      ; On récupère le paramètre implicite
   CMP #null, R2
                        On teste s'il est égal à null
   BEQ dereferencement.null
   LOAD O(R2), R2
                      ; On récupère l'adresse de la table des méthodes
   BSR 1(R2)
                      ; Appel de la méthode move (première méthode de la classe)
   SUBSP #3
                      ; On dépile les trois paramètres
; [...]
  Message d'erreur en cas de déréférencement de null
dereferencement.null:
   WSTR "Erreur : dereferencement de null"
   WNT.
   ERROR
```

On remarque qu'il faut tester si le paramètre implicite (p) est différent de null.

7.4 Codage de new

Le codage d'une expression new A() consiste à construire un structure d'objet dans le tas (cf. figures 3, 4...). Il faut pour cela connaître la taille d de la structure à allouer : c'est le nombre total des champs de A auquel on ajoute 1 pour stocker l'adresse de la table des méthodes.

Le code consiste à allouer d mots dans le tas, à stocker dans le premier mot de cette structure l'adresse ad_A de la table des méthodes de A, à effectuer l'initialisation et à stocker l'adresse de l'objet dans le registre courant.

On suppose ici qu'on cherche à produire du code qui stocke l'adresse de l'objet alloué dans le registre R2. Le code de l'instruction new A() est donc de la forme suivante :

```
NEW #d, R2 ; Allocation d'une structure de taille d dans le tas BOV tas_plein ; Test si l'allocation est possible LEA ad_A, R0 ; On stocke l'adresse de la table des méthodes STORE R0, O(R2) ; de A dans le premier mot de l'objet créé PUSH R2 BSR init. A ; Initialisation de l'objet créé POP R2
```

7.5 Codage de instanceof

On considère le programme suivant.

```
class A { }
class B extends A { }
class C extends B { }
{
    A a;
    B b;
    a = new C();
    if (a instanceof B) {
        b = (B)(a);
        println("ok");
    }
}
```

La figure 11 représente l'objet a dans le tas. Comme a est une instance de C, il contient l'adresse de la table des méthodes de C $ad_C = 7(GB)$.

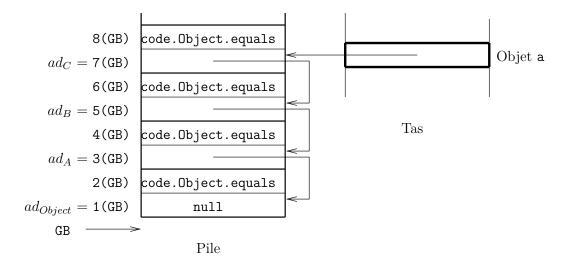


FIGURE 11 – Représentation de l'objet a et de la pile

Pour réaliser le test a instanceof B, il faut déterminer si la classe de l'objet a est un sous-type de B. Cela peut être fait en parcourant la chaîne de pointeurs de ad_C vers ad_{Object} et en testant si l'une des adresses rencontrées est égale à ad_B . Ici, c'est le cas, donc le test doit être évalué à vrai.

7.6 Codage des conversions de types

Il faut distinguer les différents cas possibles.

- 1. Conversion de la forme (T)(e) où T est le type boolean, int ou float et e est une expression de type T. Il suffit de coder l'expression, la conversion est l'opération identité.
- 2. Conversion de la forme (float)(e), où e est une expression de type int. On code l'expression, puis on effectue une opération FLOAT.
- 3. Conversion de la forme (int)(e), où e est une expression de type float. On code l'expression, puis on effectue une opération INT.
- 4. Conversion de la forme (B) (a), où B est une classe et a une expression d'un type correspondant à une classe A.

On teste si a est une instance de B, ou bien la valeur null. Si c'est le cas, l'expression (B) (a) a pour valeur a. Sinon, la conversion est incorrecte : on affiche un message d'erreur et le programme s'arrête.

8 Codage des structures de contrôle

8.1 Conditionnelles

```
\langle Code(\text{if }(C_1) \ \{\ I_1\ \} \ \text{else if }(C_2) \ \{\ I_2\ \} \ \dots \ \text{else if }(C_p) \ \{\ I_p\ \} \ \text{else} \ \{\ I\ \}) \rangle \ \equiv \ \langle Code(\text{if }(C_1) \ \{\ I_1\ \} \ \text{else }(C_2) \ \{\ I_2\ \} \ \dots \ \text{else }(C_p) \ \{\ I_p\ \} \ \text{else} \ \{\ I\ \}) \rangle
             \langle Code(C_1, faux, E_Sinon.n_1) \rangle
             \langle Code(I_1) \rangle
             {\tt BRA \ E\_Fin.} n
        E_Sinon.n_1:
             \langle Code(C_2, faux, E_Sinon.n_2) \rangle
             \langle Code(I_2) \rangle
             {\tt BRA \ E\_Fin.} n
        E_Sinon.n_2:
             \langle Code(C_p, faux, E\_Sinon.n_p) \rangle
             \langle Code(I_p) \rangle
             {\tt BRA E\_Fin.} n
        E_Sinon.n_p:
             \langle Code(I) \rangle
        E_Fin.n:
\langle Code(\text{if }(C_1) \ \{\ I_1\ \} \ \text{else if }(C_2) \ \{\ I_2\ \} \ \dots \ \text{else if }(C_p) \ \{\ I_p\ \}) \rangle \ \equiv
             \langle Code(C_1, faux, E_Sinon.n_1) \rangle
             \langle Code(I_1) \rangle
            {\tt BRA} \ {\tt E\_Fin.} n
        E_Sinon.n_1:
             \langle Code(C_2, faux, E_Sinon.n_2) \rangle
             \langle Code(I_2) \rangle
             {\tt BRA} \ {\tt E\_Fin.} n
        E_Sinon.n_2:
             \langle Code(C_p, faux, E_Fin.n) \rangle
             \langle Code(I_p) \rangle
        E_Fin.n:
```

8.2 Boucles tant que

```
 \begin{split} \langle Code(\texttt{while }(C) \ \{ \ I \ \} \ ) \rangle \ \equiv \\ \text{BRA E\_Cond.} n \\ \text{E\_Debut.} n \ : \\ \langle Code(I) \rangle \\ \text{E\_Cond.} n \ : \\ \langle Code(C, \mathit{vrai}, \texttt{E\_Debut.} n) \rangle \end{split}
```

Descriptif d'utilisation de l'interpréteur ima

Le programme ima est un interpréteur-metteur au point (débogueur) de l'assembleur de la machine abstraite décrit dans le document [Machine Abstraite].

1 Appel de ima

L'appel de l'interpréteur s'effectue en exécutant la commande

```
ima [options] nom_de_fichier_assembleur
```

Sans option, le programme est directement assemblé et exécuté.

Options (les trois premières sont exclusives) :

2 Utilisation de ima en mode metteur au point

L'appel d'ima avec l'option -d réalise l'analyse, l'assemblage et le chargement du fichier assembleur. Si aucune erreur n'a été détectée, on entre dans l'environnement d'interprétation-mise au point. Diverses commandes sont proposées sous la forme du menu suivant :

```
| d : Démarrer l'exécution jusqu'au premier point d'arrêt |
| c : Continuer l'exécution jusqu'au point d'arrêt suivant |
| a : Ajouter un point d'arrêt |
| e : Enlever un point d'arrêt |
| s : Initialiser l'exécution en mode pas à pas |
| x : Exécuter l'instruction courante |
| i : Afficher l'instruction courante |
| p : Afficher tout le programme avec les points d'arrêt |
| l : Afficher le programme entre deux lignes, avec les points d'arrêt |
| r : Contenu des registres et des codes-condition |
| m : Contenu de la pile entre 2 adresses relatives a GB |
| b : Contenu d'un bloc dont l'adresse (+- depl) est dans un registre |
```

L'invite

(ima)

permet d'entrer une des commandes, dont la description suit.

- d Initialise les registres et la mémoire, puis commence l'exécution du programme, soit jusqu'à la première erreur d'exécution, soit jusqu'au premier point d'arrêt rencontré s'il n'y a pas d'erreur, soit jusqu'à la fin du programme (instructions HALT et ERROR) si aucune erreur ni aucun point d'arrêt n'a été rencontré. La commande d peut être demandée à tout moment, provoquant la réinitialisation et la réexécution.
- c Lorsque l'exécution est interrompue par un point d'arrêt ou lorsqu'on est en mode pas à pas (voir ci-dessous), continue l'exécution du programme de la même façon qu'avec la commande d.
- a Permet de positionner un point d'arrêt. On doit indiquer un numéro de ligne du fichier source. Le point d'arrêt est positionné sur la première instruction rencontrée dans le texte à partir de la ligne indiquée. Par exemple, supposons que le fichier assembleur contient les lignes

```
1
            LOAD #3, R1
2
     ; Commentaire
3
    etiquette :
4
5
            LOAD #1, RO; Comment
6
            PUSH R1
7
            NEW #4, R2
8
            LEA -3(R2), R2
9
    etiq :
            CMP RO, R1
```

- Un point d'arrêt demandé en ligne 2, 3, 4 ou 5 est positionné en ligne 5.
- Un point d'arrêt demandé en ligne 9 est positionné en ligne 9.

Si on lance l'exécution avec ces deux points d'arrêt, l'interprète exécute LOAD #3, R1 puis s'arrête sur la ligne 5, avant son exécution, car c'est un point d'arrêt. La ligne du point d'arrêt est affichée avec son numéro, sous la forme :

```
5 : LOAD #1, RO
```

On peut alors continuer l'exécution par la commande c. Les lignes 5, 6, 7 et 8 sont alors exécutées, et l'interprète s'arrête sur la ligne 9, en affichant

```
9 : CMP RO, R1
```

On peut alors continuer avec c; la ligne 9 est exécutée, et on obtient ensuite le message d'erreur

```
** IMA ** ERREUR ** Plus d'instructions !!
```

car l'exécution ne peut pas continuer et l'interprète n'a pas rencontré d'instruction HALT ou ERROR.

e Permet de supprimer un point d'arrêt précédemment positionné. La logique de la commande est la même que pour la comande a : sur le programme précédent avec ses deux points d'arrêt en lignes 5 et 9, enlever un point d'arrêt en ligne 2, 3, 4 ou 5 enlève celui de la ligne 5, enlever un point d'arrêt en ligne 6, 7 ou 8 n'a pas d'effet, enlever un point d'arrêt en ligne 9 enlève celui de la ligne 9.

s Initialise les registres et la mémoire, et s'arrête sur la première instruction en l'affichant, par exemple sur l'exemple précédent :

1 : LOAD #3, R1

De même que d, s peut être demandée à tout moment.

x Exécute l'instruction sur laquelle s'est arrêté l'interprète, soit parce que c'est un point d'arrêt, soit parce que l'instruction précédente a été exécutée par la commande x. L'instruction suivante est ensuite affichée. Par exemple, après s, x exécute LOAD #3, R1 et affiche

```
5 : LOAD #1, RO
```

Le x suivant exécute cette dernière instruction, et affiche

```
6: PUSH R1
```

On peut bien sûr à tout moment continuer l'exécution par c.

- i Réaffiche la prochaine instruction à exécuter (utile lorsqu'elle a disparu de l'écran suite à des commandes r , m, p ou l).
- p Affiche le programme avec les numéros de ligne, la position de PC (sous la forme -->) et les points d'arrêt (sous la forme ##). Par exemple, sur l'exemple avec les points d'arrêt spécifiés plus haut, avant toute exécution ou après une commande s, on obtient l'affichage

```
-->
        1|
                            LOAD #3, R1
        21
                     ; Commentaire
        31
                   etiquette :
        4 I
     ## 5|
                            LOAD #1, RO; Comment
        6 I
                            PUSH R1
        7|
                            NEW #4, R2
                            LEA -3(R2), R2
        8|
     ## 91
                            CMP RO, R1
                   etiq:
```

Apres une commande d, on obtient pour p le résultat

```
1 |
                      LOAD #3, R1
   21
               ; Commentaire
   3|
              etiquette :
   41
## 5|
                      LOAD #1, RO; Comment
   61
                      PUSH R1
   7|
                      NEW #4, R2
                      LEA -3(R2), R2
   81
## 91
              etiq:
                      CMP RO, R1
```

- 1 Idem p, mais permet d'indiquer un intervalle de lignes.
- r Affiche la valeur des registres SP, GB, LB et des R_i. Affiche également les codes-condition positionnés à vrai. Les valeurs d'adresse mémoire en zone pile sont indiquées relativement à GB et LB. Les valeurs d'adresse mémoire en zone tas sont indiqués comme des déplacements relatifs à l'adresse du premier mot d'un bloc alloué (par NEW); l'affichage mentionne ce déplacement ainsi que la taille du bloc. Pour les valeurs d'adresse de code, voir ci-dessous la commande m. Sur l'exemple, avant toute exécution, on obtient

```
: @ zpile : O(GB) , O(LB)
SP
   : @ zpile : O(GB) , O(LB)
    : @ zpile : O(GB) , O(LB)
RO
   : <indefini>
                                            : <indefini>
R2
    : <indefini>
                                            : <indefini>
                                        RЗ
R4
    : <indefini>
                                        R5
                                           : <indefini>
   : <indefini>
                                        R.7
                                            : <indefini>
R.6
R8
    : <indefini>
                                        R9
                                           : <indefini>
R10 : <indefini>
                                        R11 : <indefini>
R12 : <indefini>
                                        R13 : <indefini>
R14 : <indefini>
                                        R15 : <indefini>
Codes-condition vrais : NE GE GT
```

Noter les codes-conditions initialement vrais.

Après d et c, on obtient pour la commande r l'affichage

```
: @ zpile : 1(GB) , 1(LB)
    : @ zpile : O(GB) , O(LB)
GB
   : @ zpile : O(GB) , O(LB)
RO
                                       R1
                                           : 3
R2
    : @ bloc (taille 4) depl -3
                                       R3
                                           : <indefini>
   : <indefini>
                                       R5
                                           : <indefini>
   : <indefini>
R.6
                                       R7
                                            : <indefini>
R8
   : <indefini>
                                       R9
                                           : <indefini>
R10 : <indefini>
                                       R11 : <indefini>
R12 : <indefini>
                                       R13 : <indefini>
R14 : <indefini>
                                       R15 : <indefini>
Codes-condition vrais : NE GE GT
```

m Affiche le contenu de la pile entre deux adresses, désignées par des déplacements relativement à GB. Indique le cas échéant où pointent SP et LB. Une valeur d'adresse mémoire est indiquée sous la même forme que pour l'affichage des valeurs des registres, une valeur d'adresse de code est présentée sous la forme

```
@ code ligne 47
```

Sur l'exemple, après d et c, la commande m, en indiquant les déplacements 1 et 5 produit :

```
5 | <indefini>
4 | <indefini>
3 | <indefini>
2 | <indefini>
SP ----> 1 | 3
```

b Affiche le contenu d'un bloc (précédemment alloué par NEW). Le bloc choisi est désigné par un registre banalisé, qui doit contenir une adresse en zone tas, déplacement (éventuellement non nul) par rapport à l'adresse du premier mot d'un bloc alloué. Sur l'exemple, après d et c, on obtient pour la commande b et le registre 2 :

```
0 | <indefini>
1 | <indefini>
2 | <indefini>
3 | <indefini>
```

t Affiche le temps d'exécution du programme depuis le dernier démarrage. Le temps total d'exécution est dans tous les cas affiché à la fin du programme si aucune erreur n'a été rencontrée.

- ? Affiche le menu.
- ${f h}$ Affiche le menu.

 ${\bf q}\,$ Sort de l'interpréteur-metteur au point.