Documentation de conception
Compte-rendu intermediaire

Naima Amalou Jiayun Po Zaineb Tiour An Xian Yidi Zhu

Janvier 2020

Table des matières

Table des figures

Etape A :

La conception de l'étape A est portée sur l'analyse lexicographique et syntaxique du programme par l'outil ANTLR. L'étape A est la première étape à faire pour filtrer les mots écrits par l'utilisateur dans un programme.

Analyse lexicographique :

L'analyse lexicographique du programme est construite telle que certains mots-clés ou symboles spéciaux dans le programme sont bien reconnus par le compilateur dès le début. Les lettres, nombres ou symboles construits par défaut sont bien classés soit étant un identificateur, une traduction littérale d'un entier, une traduction littérale d'un flottant, une chaîne de caractères ou bien un commentaire. Chaque reconnaissance de mot génère un token ou un lexème par l'ANTLR.

Le tableau ci-dessous montre les mots réservés et les symboles spéciaux :

asm	instanceof	new	true	class	If
println	printlnx	while	extends	null	else
printx	print	readInt	protected	false	this
readInt	protected	false	readFloat	return	, (virgule)
{	}	||	<	>	!=
>=	<=	=	+	-	==
*	/	%	. (point)	(	&&
)	;	!			

FIGURE 1.1 – Tableau regroupant les mots réservés et les symboles spéciaux.

Un identifiant de variable, classe, méthode ou type est forcément commencée par une lettre. Une littérale entière ou flottant est un ensemble de nombre, sauf le cas d'une valeur hexadécimale de type float. Une valeur hexadécimale de type float peut contenir la lettre 'A' à 'F' et la lettre 'X'.

Une chaîne de caractères commence et finit par des guillemets. Un commentaire est soit

délimité par "/* */" où commencé par le symbole "//" .
Les espaces vides, tabulations, retours à la ligne et les commentaires sont ignorés pendant l'analyse lexicographique. L'include d'un fichier est reconnu dans l'analyse lexicographique.
Lexème include doit être suivi avec le nom d'un fichier délimité par des guillemets.
Include "FILENAME"

Comment définir un lexème :

Dans le fichier DecaLexer.g4, un lexème est écrit avec tout en majuscules et le le mot réservé est écrit entre guillemets, sensible aux majuscules et minuscules.
E.G : LEXÈME : "Lexème" ;
le mot fragment est utilisé pour assigner temporairement une suite de caractères.

Analyse Syntaxique :

L'analyse syntaxique est en fait un parser qui produit un arbre abstrait automatiquement d'après les règles ou grammaires.
Les grammaires données nous permettent de définir une classe de langage plus grande. Par exemple, un programme est défini d'après le réglé sur la déclaration des classes sont définies avant un programme principal.
Toutes les classes qui ne respectent pas se règle automatiquement entraînent une erreur syntaxique. De même, l'analyse syntaxique est faite sur l'outil ANTLR. Les grammaires sont définies auparavant. [cf. pages 55 à 58 du poly projet GL].

Les points ci-dessous sont les remarques importantes pendant la conception :

- Une table de symbole (SymbolTable) est déclarée dans la partie @members du fichier DecaParser.g4 pour rassembler tous les identifiants déclarés dans le programme.
- Une règle (non définie dans la lexicographique) est écrit en minuscule.
- Une règle (non définie dans la lexicographique) est écrit en minuscule.
- Chaque règle est associée à une suite de grammaires et un bloc avec des actions à effectuer. NOM_REGLE : grammaire {action}
- L'interpellation de la constructor de classe Java est souvent nécessaire au niveau du fils de l'arbre ou bien quand il y a une liste ou opération à traiter.
- La fonction cette location est appelée à certains blocs d'action pour repérer l'endroit du mot.
- Si un mot non terminal est associé à un return, il est impérativement d'assigner la variable sortie à une valeur.
- Si une expression dans la grammaire est suivi avec un ' ?'cela représente que l'expression peut exister ou ne pas exister dans la grammaire.
- Si 2 expressions sont séparées avec un '|' est que la grammaire peut soit prendre l'expression 1 ou l'expression 2.

- Le symbole '*' signifie que l'expression peut répéter plusieurs fois.

- Le symbole '+' signifie que l'expression peut répéter plusieurs fois mais doit exister au moins une fois.
- Une expression avec un ' ' veut dire que la grammaire prend n'importe quelle expression sauf l'expression énoncée

Décompilation :

Après avoir découpé le code en petits morceaux, une méthode de décompilation est implantée en java pour la reconstruction du code. La sortie de la décompilation doit être exactement égale au code lequel l'utilisateur est entré. Il n'y a pas de grand détail sur la méthode de décompilation sauf qu'il faut faire attention à l'affichage de point virgule et l'ordre d'appellation.

Etape B :

L'étape B permet de faire la vérification contextuelle d'un programme Deca correct syntaxiquement. C'est l'étape qui suit l'analyse lexicale et syntaxique. Elle permet d'ajouter des informations au programme Deca qui sont trés utiles pendant la génération du code.

Analyse contextuelle :

Environnements :

On distingue entre deux types d'environnements : environementExp et environnementsType.

Une partie de **l'environmentExp** a été fournie, nous avons complété cette classe en ajoutant comme attribut un HashMap qui permet de lier chaque symbole par sa définition expression dans l'environnement courant.
Pour avoir la possibilité d'empilement pour les environnements Exp, chaque EnvironmentExp admet un parent EnvironmentExp, le premier environment créer dans le Main de programme n'admet pas du parent.

L' environmentsExp de chaque classe est créée au moment de la création de la classe , qui sera remplie au fur et à mesure pendant la vérification contextuelle pendant la passe 1 et la passe 2.
Pour chaque nouveau environment expression créer nous pouvons déclarer des nouvelles définitions, toutefois une variable déjà déclarée renvoie une erreur pendant la vérification contextuelles.

Nous avons considéré l'empilement de deux environments env1 et env2 comme une mis-à-jour de l'environnement env1 avec les définitions de l'env2 .

En ce qui concerne **l'environmentType** :

Pour chaque programme deca, il existe un seul et unique environmentType , créé au moment de la vérification contextuelles , l'environmentType est ajouté comme attribut dans le compiler, pour faciliter son accès pendant la vérification contextuelles. cet environment contient une Map qui lié chaque symbole par sa définition.

La classe Operators :
Nous avons définie cette classe pour associer chaque opérateur avec sa représentation cela permet de factoriser le code au niveau des opération binaire. Dans la classe AbstractBinaryOption nous avons ajouté des méthodes pour calculer le type à la sortie d'une opération binaire valide et qui renvoie une erreur dans le cas échéant.

Les relations de sous typages :

Nous avons ajouté ces relations dans la classe Type pour tester les opérations possible entre deux types : sous-type, cast et assign.

Les méthodes verifyXYZ :

Au début, nous avons commencé à compléter chaque méthode verifyXYZ à partir de "Program" en suivant l'ordre des branches dans l'arbre. A cause des dépendances entre certaines noeuds, après avoir complété les méthodes qui sont les bases pour la suite comme la vérification de la déclaration des variables et le début de la vérification de l'instruction, nous avons décidé de vérifier les feuilles au lieu des noeuds.

- **Program :** pour le language sansObjet, il faut vérifier le contenu de main, qui est composé de variables et instructions.

- **Main :** nous devons vérifier la déclaration de variables et la list de instruction. Pour "main" qui retourne rien, (équivalent de void), nous avons réfléchi de créer un voidType comme le paramètre qui correspond au type prévu retourné.

- **DeclVar :** d'abord, nous devons éviter la variable de type void.

- **ListInst.**

- **Expression :** nous devons redéfinir la méthode verifyExp pour les class héritant de AbstractExpr. La vérification de 'instruction pour les expressions est effectivement de vérifier le format d'expression différente.

- **Opération binaire :** L'idée principale est d'opérer sur deux valeurs de même type.
  Nous avons construit une nouvelle class qui s'appelle Operator, qui sert en particulier à vérifier les opérations binaires.
  Il y a une nouvelle méthode qui permet de renvoyer le type attendu par rapport à les types de variables et la nature de opérateur.
  Pour l'opération arithmétique qui est spécifique, nous devons vérifier que les variables sont nécessairement de type int ou float. En respectant les règles, la variable de type int doit être transformé en type float dans les cas spécifiques.

- **Affectation.**

- **Print.**

- **Littéral.**

Etape C :

La partie C est l'étape finale qui nous permet de générer du code assembleur à partir d'un arbre décoré éventuellement fourni par l'étape B.

Génération du code :

Principalement, cette partie consiste à suivre l'arboressance de l'arbre et de définir les méthodes codeGenInst et codeGenExpr là où il faut. Clairement, ceci implique la création de plusieur autres classes qui permettent de générer la dépendance entre les différentes méthodes.

Les méthodes principalement écrites et qui sont override dans les sous classes selon le besoin sont les suivantes :

- codeGenExpr : permet de gérer la déclaration des variables.
- codeGenInit : permet de gérer les codes assembleur pour la plupart des instructions.
- codeGenInstNot : permet de gérer les codes assembleur pour les instructions de quelques cas spécifiques. Par exemple : le Or dans IfthenElse, inst pour while.
- codeGenPrint : permet de gérer les prints de phrases et les autres variables.
- codeGenPrintX : permet de print float en hexadécimal.

On retrouve aussi les méthodes abstraites suivantes :

- public abstract void mnemop(DecacCompiler compiler, GPRegister rightOperand) : fonction de operation binaire avec codeGenexpr, pour initialiser les variables.
- public abstract void codeGenOp(DecacCompiler compiler, Label label) : fonction de operation binaire avec codeGenInst.
- public abstract void codeGenNot(DecacCompiler compiler, Label label) : fonction de operation binaire avec codeGenInstNot.

Pour faire toute ces méthodes correctement en maitrisant la gestion des registres et la gestion des erreurs, on avait besoin de créer soit des classes comme SPManager et GB manager où bien des méthodes simples pour gérer les labels et l'affichage des erreurs.

- GBmanager : une classe qui gére principalement les registres.

- SPmanager : une classe qui retourne le stack pointer et qui permet de calculer le nombre de case mémoire nécessaire à reserver pour un programme donnée.

- Gestion des labels : la partie pour gérer les labels est principalement utilisé pour IfThenElse et While. Elle permet d'enregistrer par ordre les labels qu'on va vouloir réutiliser aprés surtout dans ifthenelse et while.

- Gestion de l'initialisation des variables avec un getAddr et un setAdrr : ces méthodes nous permettent d'enregistrer les variables qu'on a deja initialisé dans une liste de variable, si une variable non inittialisé a été utilisée, on génére une erreur.