GRENOBLE INP PHELMA
GRENOBLE INP ENSIMAG

Manuel utilisateur du compilateur Decac

DIJS Thomas FALGAYRAC Loic HO-SUN Jules
WANG Caroline NOIRY Sylvain
groupe GL 54

Phelma SEOC/ Ensimag ISI-MMIS
January 24, 2022

Table des matières

1 Utilisation du compilateur Decac

1.1 Introduction

Le langage Deca est un langage de programmation orienté objet basique dérivé du langage Java. L'outil Decac est un compilateur spécifique à ce langage qui permet de générer le code assembleur correspondant à un programme deca.

Ce manuel utilisateur explique comment utiliser le compilateur decac, présente les différentes options à la compilation, montre les limites du compilateur et explique comment interpréter les messages d'erreur qu'il peut renvoyer.

1.2 Exemple d'utilisation

L'utilisation du compilateur decac se fait avec la ligne de commande ci-dessous:

Pour compiler un seul fichier :

decac [option] [monFichier.deca]

ou, pour compiler plusieurs fichiers :

decac [option] [fichier1.deca] ... [fichierX.deca]

Nous rappelons que les fichiers à compiler doivent être de type ".deca". Un fichier assembleur ".ass" est alors généré et peut être exécuté par la machine IMA à condition qu'il n'y ait pas eu d'erreur à la compilation.

1.3 Liste des options à la compilation

-b (banner) : affiche une bannière indiquant le numéro de l'équipe et le nom des personnes composant l'équipe

-p (parse) : arrête decac après l'étape de construction de l'arbre, et affiche la décompilation de ce dernier. Par exemple, s'il n'y a qu'un fichier source à compiler, la sortie doit être un programme deca syntaxiquement correct.

-v (verification) : arrête decac après l'étape de vérification ; 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 $R_0 ... R_{X-1}$, avec $4 \leq X \leq 16$.
Si X n'est pas spécifié, le compilateur renvoie: *Merci d'indiquer un nombre de registres*; si X ne correspond pas, il renvoie: *Le nombre de registres doit être compris entre 4 et 16 inclus*

-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. Lorsqu'elles sont demandées en même temps, le message d'erreur *Les options -p et -v sont incompatibles entre elles*

2 Messages d'erreur

2.1 Erreurs lexicographiques

Message	Tests	Interprétation
token recognition error at : "..."	include_inreadable.deca	Le code écrit ne correspond à la structure d'aucun token du lexer

Chemin des tests : /src/test/deca/syntax/invalid/lexer

2.2 Erreurs syntaxiques

Message	Tests	Interprétation
extraneous input '..' expecting '..'	Class.deca equals2.deca ineq_inv.deca manque_op_mult.deca manque_op_plus.deca manque_op_div.deca	Un des tokens de la suite de tokens reconnus par le lexer ne suit aucune règle syntaxique du parser
mismatched input '..' expecting {..}	equals.deca manque_op_moins.deca manque_op_mod.deca	Un des tokens de la suite de tokens reconnus par le lexer ne suit aucune règle syntaxique du parser
missing '...' at '...'	manque_acolades.deca manque_parentheses.deca	Il manque un token à l'endroit indiqué pour qu'une règle du parser puisse être appliquée
no viable alternative at input '...'	manque_pts_virgules.deca	Le parser ne trouver pas de règle appliquable au code deca écrit à l'endroit indiqué

Chemin des tests : /src/test/deca/syntax/invalid/parser

2.3 Erreurs de contexte

Une erreur de contexte apparaît lorsque la syntaxe est bonne mais que le code mis bout-à-bout ne correspond pas. Ces erreurs sont repérées lors de l'étape B, avant la décoration de l'arbre. Si aucune erreur n'est repérée, le code va passer à l'étape suivante et être traduit en assembleur. Une détection d'erreur au cours de cette étape permet donc d'éviter de générer du code inutilement.

Message	**Tests**	**Interprétation**
float is expected	affect_boolean_to_float.deca affect_float_to_boolean.deca	Un nombre flottant est demandé
int is expected	affect_float_to_int.deca affect_boolean_to_int.deca wrong_assignation.deca wrong_assignation3.deca	Un nombre entier est demandé
boolean is expected	affect_int_to_boolean.deca wrong_assignation2.deca	Un booléen est demandé
'..' is expected	invalid_new_2.deca	Un objet de la classe '..' est demandé
Var must not be void	affect_void.deca	Un type vide a été affecté à la variable
Unary minus operands needs to be either an int or a float	invalid_minus.deca	L'utilisation d'un moins unaire nécessite un nombre entier ou flottant
'..' needs to be a boolean	invalid_not.deca if_int.deca if_float.deca	Un booléen est nécessaire dans l'utilisation d'un if, d'un while ou d'un not
'..' n'est pas défini	use_not_defined.deca pt_virgule.deca Class_wrong2.deca	La variable utilisée n'est pas initialisée
'..' is not a type	null.deca Instanceof_wrong.deca Class_wrong8.deca	La déclaration de variable n'est pas correcte
Both binary boolean operators need to be a boolean	bool_incorrects.deca invalid_op_bool.deca invalid_op_bool2.deca	Les opérateurs utilisés doivent être booléens
Both binary arithmetic operators need to be either an int or a float	invalid_op_arith.deca invalid_op_arith1.deca div_bool.deca	Il faut des nombres (flottants ou entiers)
There must be two integers for the modulo	modulo_float.deca modulo_float2.deca modulo_2.deca	L'utilisation du modulo nécessite 2 entiers.
'..' is a predefined type, can't be a var name	print_int.deca	Le nom d'une variable correspond à un type
'..' is not a class	invalid_new.deca	Utilisation d'une classe sans l'avoir implémentée
'..' has a wrong list of param	invalid_method_call.deca	Le type des variables d'entrée est incorrect
'..' is already defined at '..'	redefine.deca test.deca	La variable utilisée n'est pas initialisée
'..' is a class name defined at '..', can't be a var name	var_class.deca	'..' est une classe et ne peut donc pas être le nom d'une variable
'..' is a parameter already defined at '..'	paramVerify.deca	Ce paramètre a déjà été défini auparavant
What is printed needs to be either an int, a float or a string	print_boolean.deca println_boolean.deca	L'argument de print / println doit être un entier, un flottant ou un string

Chemin des tests : /src/test/deca/syntax/invalid/context

2.4 Erreurs d'exécution

La totalité des erreurs d'exécutions renvoient le message "Erreur:" suivi par la cause de l'erreur. Les vérifications d'erreurs à l'exécution peuvent être désactivées avec l'option -n, mais cela pourrait se traduire en un comportement indéfini si le programme Deca n'est pas correcte. Une erreur d'exécution peut être:

- Une division ou un reste de la division par l'entier 0
- Un débordement sur une opération arithmétique flottante (celles sur des entiers sont effectuées modulo 2^{32})
- Une division flottante par 0.0
- Un dépassement de pile
- Un dépassement de tas

Une erreur à l'exécution renvoie un message d'erreur explicite qui la traduit pour l'utilisateur. Tous les messages d'erreur qui concernent la partie *exécution* commencent par *Erreur:*. Voici la liste des messages d'erreur:

Message	**Interprétation**
Erreur: division par 0	Une division par 0 est repérée dans le code
Erreur: le tas est plein	L'espace mémoire alloué au tas est insuffisant
Erreur: la valeur entrée est du mauvais type	La valeur entrée par l'utilisateur est du mauvais type
Erreur: déréferencement de null	Un symbole de type null est déréférencé
Erreur: dépassement de pile	L'espace mémoire alloué à la pile est insuffisant

Chemin des tests : /src/test/deca/syntax/invalid/codegen

3 Optimisation du compilateur

Nous avons choisi d'optimiser notre compilateur grâce à un ensemble de méthodes que nous traiterons rapidement ci-dessous. Ainsi, suivant l'option choisie, le temps de compilation peut-être légèrement plus long mais le temps d'exécution est quant à lui réduit.

3.1 Utilisation

Pour utiliser l'optimisation, il est possible d'utiliser les options de compilation suivantes:

- Aucune option ou -O0 engendre la compilation du fichier sans aucune optimisation particulière. Une bonne architecture de la génération de code permet d'avoir des performances tous de même très respectables.
- -O1 fait uniquement le calcul d'expressions constantes à la volée. Le temps de compilation ajouté est négligeable, mais le gain de performance peut-être conséquent.
- -O2 fait toutes les optimisations implémentées en utilisant un graphe de contrôle de flot. Une mise en forme SSA est appliquée, ce qui permet de ne n'avoir qu'une affectation par variable du graphe. Puis, les deux principales optimisations appliquées sur le graphe sont la propagation des constantes et la suppression de code mort. Cette optimisation n'est faite qu'à l'intérieur du main ou d'une méthode. Cette option permet d'avoir le gain en temps d'exécution le plus important, mais est aussi assez coûteuse en temps de compilation. De plus certaines variables peuvent être supprimées, donc l'écriture en mémoire du résultat n'est pas garanti.
- -O2g fait les mêmes optimisations que -O2 et génère en plus un fichier .dot pour graphviz, afin que le graphe puisse être lu visuellement ; il n'est donc plus nécessaire de le dessiner à la main.

L'utilisation de ces options engendre un temps de calcul plus long mais réduit très fortement le temps de compilation. Nous avons réussi à diviser jusqu'à 4 le nombre de cycles nécessaires pour les exécuter.

3.2 Gain de temps et d'énergie

L'utilisation des options de compilation précédentes permet de réduire très fortement le nombre de cycles d'horloges nécessaires pour exécuter le programme. Le nombre de cycles d'horloges gagnés peut être très rapidement calculé via les programmes tests disponibles via le chemin donné en énoncé. Cette optimisation s'inscrit également dans le cadre d'une démarche environnementale: moins un programme aura besoin de cycles d'horloges pour fonctionner, moins il aura besoin d'énergie. Néanmoins, un compilateur ne peut pas faire des miracles, il est donc primordial de penser un programme écrit en Deca en ayant en tête des algorithmes efficaces. La plateforme d'exécution IMA étant orientée embarqué de au vu de ses performances, les opérations de multiplication et de division sont assez lourdes, alors que les opérations mémoires sont assez rapides et peu sensibles à la localité des données.

4 Les limites du compilateur

Pour mieux comprendre l'analyse du code du point de vue du compilateur, voici une brève explication de son fonctionnement en 3 étapes :

- **Passe 1 : vérification du nom et de la hiérarchie des classes**
  Nous vérifions que le nom des classes définies n'est pas "Object" puis on les récupère pour pouvoir les utiliser lors de l'analyse des passes suivantes
- **Passe 2 : vérification des champs et de la signature des méthodes des différentes classes**
  Nous pouvons redéfinir un champ dans une classe par un nom qui a déjà été utilisé dans une classe définie au-dessus dans la hiérarchie. Si une méthode est redéfinie elle doit avoir la même signature que la méthode héritée et doit avoir pour type de retour un sous-type du type de retour de la méthode héritée
- **Passe 3 : vérification des méthodes, des instructions et des initialisations**
  Nous vérifions si les variables utilisée sont bien définies (mais pas si elles sont initialisées) puis chaque instruction ou appel à une méthode est fait son la spécification du langage deca.
  Nous vérifions que "This" n'apparaît pas dans le programme principal.
  Nous vérifions que, dans une sélection de champ protégé, le type de l'expression est un sous-type de la classe analysée

4.1 Limites de l'analyse lexicale et syntaxique

- De la même manière qu'en Java, les attributs sont "public" par défaut, la seule autre visibilité disponible dans le langage deca est "protected".
- Les constructeurs de classe ne sont pas implémentés, nous privilégierons donc des getters pour pouvoir initialiser les variables à l'intérieur des classes.
- au sein d'une de classe ***A***, l'utilisation d'une méthode ou d'un champ ***x*** ne peut se faire qu'à l'aide d'une sélection : ***this.x***.

4.2 Limites de l'analyse contextuelle

- La conversion des types des variables (cast) fonctionne seulement pour des conversions implicites.
- Quelques erreurs n'ont pas été gérées :
  - Lorsqu'une méthode est utilisée mais pas implémentée
  - Lorsqu'un constructeur est appelé sans le new
- Des erreurs qui devraient se lancer mais qui ne le sont pas :
  - Lorsqu'une classe ***MaClasse***, appelle directement une méthode ; ***MaClasse.maMethode()***.

4.3 Limites de la compilation

- Lorsque le nom d'une classe appelle une méthode, le message d'erreur renvoyé ne fait pas correspondre la position et n'est pas clair.
- Les classes utilisent une méthode "void init()" pour initialiser la structure, toute redéfinition de cette méthode a la responsabilité d'initialiser la structure.

5 Propriété intellectuelle

En France, les droits d'auteurs des logiciels sont protégés par l'**Article L112-2 CPI 2 :** *Sont considérés notamment comme œuvres de l'esprit au sens du présent code : (. . .) 13les logiciels, y compris le matériel de conception préparatoire.*