PROJET GL39

GIRAULT MATHIS
GAUTHIER PAUL
FAUCHON JULES
STEVANT GUILHEM
AMARE BAPTISTE

Projet GL - GL39

Manuel Utilisateur
Version 1.1.0

Version History Summary

Issue	Description	Date
1.0.0	Version initiale	21/01/23
1.1.0	Correctif mineurs et ajout listing erreurs	22/01/23

Author: GL39

Date Produced: 23/01/23

1. Introduction

Ce manuel s'adresse aux utilisateurs du compilateur Deca du groupe GL39, qui ont déjà une connaissance des spécifications détaillées du langage Deca. Il décrit les limitations de l'implémentation du compilateur, les messages d'erreur qui peuvent être retournés à l'utilisateur, l'extension ARM de ce compilateur et les modes opératoires pour utiliser cette extension.

2. Utilisation du compilateur

Pour compiler votre programme codé en langage Deca (.deca) il vous faut vous rendre dans votre terminal à l'emplacement du fichier et effectuer la commander :

decac <fichier.deca>

→ Cela vous produira un fichier .**ass** assembleur qu'il vous sera ensuite possible d'exécuter avec ima par exemple.

Il faut savoir que des spécifications du compilateur sont implémentées, pour les choisir il vous suffit d'utiliser:

decac -X <fichier.deca>

En remplacant **-X** par :

- **-b : Cela affiche une bannière indiquant le nom de l'équipe**
- **-p : Cela 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)**
- **-r X: Cela permet de limiter le nombre de registres banalisés disponibles à R0 … R{X-1}, avec X compris entre 4 et 16.**
- **-v : Cela arrête decac après l'étape de vérifications (ne produit aucune sortie en l'absence d'erreurs)**
- **-n : Cela supprime les tests à l'éxécution spécifiés dans les points 11.1 et 11.3 de la sémantique de Deca.**
- **-d : Cela active les traces de debug. Répéter l'option plusieurs fois pour avoir plus de traces.**
- **-P : Si il y a plusieurs fichiers sources, lance la compilation des fichiers en parallèle (pour accélérer la compilation).**
- **-arm : Cela génère du code assembleur qui sera éxécutable sur un processeur ARM-V6** *(Extension présentée en partie 5)*

3. Limitations de l’implémentation

Il est important de noter que notre implémentation du compilateur Deca comporte certaines limitations qui peuvent affecter la compilation ou l'exécution du code généré. Les utilisateurs sont invités à consulter les informations suivantes pour éviter les erreurs potentielles :

- Notre compilateur ne prend pas en compte l’utilisation des conversions explicites (**cast**) et le test d’appartenance à une classe (**instanceof**).

  → Il n’est donc pas possible de forcer le compilateur à considérer une variable comme étant d'un type qui n'est pas le type déclaré et de tester l’appartenance à une classe dans vos programmes Deca.

  Ces limitations sont la cause du manque de temps en fin de projet qui nous a contraint à ne pas implémenter la partie B et C pour ces subdivisions du langage Deca.

4. Messages d'erreurs

Le compilateur peut retourner des messages d'erreur lors de la compilation du code si le fichier deca n'est pas valide, ou lors de l'exécution dans certains cas précis. Ces messages d'erreur peuvent indiquer des erreurs de syntaxe, des erreurs de vérification contextuelle ou des erreurs d'exécution. Les utilisateurs doivent être conscients des types d'erreurs qui peuvent survenir et de la configuration qui les provoque.

Le format des erreurs est normalisé et est composé de la manière suivante :

<nom de fichier.deca>:<ligne>:<colonne>: <description informelle du problème>

Par exemple :

test.deca:12:4: Identificateur "foobar" non déclaré

Voici, en fonction de leurs provenances, les erreurs qu'il est possible que l'utilisateur rencontre :

- **Erreurs lors de l'exécution:**

- 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

- Déréférencement de null

→ Dans ces cas, votre programme est incorrect.

—----------

- Entrée utilisateur de type non float sur un ReadFloat()

- Entrée utilisateur de type non int sur un ReadInt()

→ Dans ces cas, votre programme est correct mais l'utilisateur fournit des entrées incorrectes.

—----------

- Débordement mémoire Tas/pile

→ Dans ces cas, le programme est correct mais son exécution entraîne une erreur de débordement mémoire.

- **Erreurs Vérification contextuelle**

Toutes les erreurs liés à la vérification contextuelle sont listées dans un document externe nommé :

GL39_listing_erreurs_etape_B.pdf

- **Erreurs analyse lexicale et syntaxique**

Circular include

→ L'erreur se produit lorsque deux fichiers incluent mutuellement l'un l'autre, causant ainsi une boucle infinie.

Par exemple, si on prend les fichiers :

- A.deca → #include “B.deca”
- B.deca → #include “A.deca”

Alors l'exécution d’un des deux fichiers provoquera une erreur.

IncludeFileNotFound

→ L’erreur se produit lorsque un fichier spécifié est introuvable ou inaccessible.

Par exemple, vous écrivez un #include dans un programme avec un nom de fichier qui n’existe pas dans le path déclaré de votre terminal.

InvalidLValue

→ L’erreur se produit lorsque le fils gauche d’une affectation dans la règle assign_expr n’est pas une lvalue (voir la spécification du langage deca).

Par exemple, vous écrivez du code “Object.equals = x”. L’expression de gauche ne peut pas avoir une valeur assignée.

InvalidFloatInput

→ L’erreur se produit si un flottant est déclaré comme trop grand ou trop petit, et doit être arrondi vers l’infini ou vers 0.

Cette erreur sera lancée pour des flottants supérieurs à 3.4028235E38f ou inférieurs à 1.4E-45f.

InvalidIntInput

→ L’erreur se produit si un entier est déclaré comme trop grand (en positif ou négatif), et ne peut pas être codé sur 32 bits.

Cette erreur sera lancée pour des entiers supérieurs à 2^31 - 1 ou inférieur à -2^31.

DecaRecognitionException

→ L'erreur se produit si le fichier deca ne respecte pas la syntaxe concrète du langage deca.

Par exemple, un fichier qui contient uniquement '{', ou un fichier qui comprend une chaîne de caractères non fermée '"test'.

- **Decac internal error**

→ Cette erreur n'est pas censée être relevée car elle correspond à un mauvais fonctionnement du compilateur, donc une faille dans notre code. Si vous relevez cette erreur merci de contacter GL39.

5. Extension ARM

4.1 Description Extension

L'extension ARM est une extension rajoutée au compilateur Deca permettant de générer du code assembleur spécialement pour être exécuté sur un processeur **ARM** (Version 6). Cette extension est implémentée de manière à réutiliser les algorithmes de parser et lexer ainsi que de décoration et vérification de l'arbre implémentées pour le compilateur classique tout en ajoutant une nouvelle manière de générer du code assembleur.

Il est à noter que si l'utilisateur souhaite exécuter le code assembleur ARM, il lui faudra un environnement spécial capable de le faire, l'installation de cet environnement est décrite dans le document *GL39_documentation_extension*

4.2 Mode opératoire de l'extension

Pour utiliser les extensions du compilateur, les utilisateurs doivent utiliser les options de la commande decac en tapant :

decac -arm [nom programme deca]

Cela crée un fichier fichier.S qu'il est ensuite possible d'exécuter sous une architecture ARM.

Les détails sur les limitations et les potentielles erreurs sont décrites dans la suite de ce document.

4.3 Limitation de l’extension

L'extension ARM de notre compilateur Deca permet de générer du code pour les processeurs ARM. Cependant, il est important de noter que cette extension comporte certaines limitations qui peuvent affecter la compilation ou l'exécution du code généré :

- Notre extension ARM est capable uniquement de compiler des programmes Deca **Sans Objets.**

  → Il est impossible de compiler du langage Deca contenant des fonctions liées à la création de Classes Deca.

- Les **flottants** et les **entiers** ne sont pas gérés par l’extension ARM.

  → Il est possible d’effectuer des calculs avec les entiers mais il n’est pas possible de les afficher. Nous pouvons travailler avec les booléens. Les flottants ne sont pas traités.

- Notre compilateur ne prend pas en compte l’utilisation des conversions (**cast**) et le test d’appartenance à une classe (**instanceof**).

  → Il n’est donc pas possible de forcer le compilateur ARM à considérer une variable comme étant d'un type qui n'est pas le type déclaré et de tester l’appartenance à une classe dans vos programmes Deca.

- ReadInt ReadFloat non implémentés.

  → Le compilateur ARM ne gère pas les entrées utilisateurs de type entiers et flottants.

- La gestion du Modulo n’est pas implémentée sur l’extension ARM.

Notre compilateur Deca de base ne gèrent pas le cast et instanceof, ce qui fait que l’extension ARM hérite également de ces limitations.

L’utilisation des flottants nécessite une utilisation des registres différente que celle des entiers, implémentation que nous n’avons pas eu le temps d’implémenter.

Nous n’avons pas eu le temps d’implémenter ReadInt et ReadFloat en assembleur pour ARM.

5. Conclusion

Ce manuel a pour objectif de fournir aux utilisateurs les informations nécessaires pour utiliser correctement le compilateur Deca et ses extensions. Les utilisateurs sont invités à lire attentivement ce manuel avant d'utiliser le compilateur.