Manuel Utilisateur :Compilateur Deca

Auteurs :
Mehdi El Idrissi El Fatmi
Khadija El Adnani
Alaa Jennine
Badr El Khoundafi
Yassine El Kouri

22/01/2025

Contents

1 Généralités

Ce document constitue le manuel utilisateur pour Deca qui est un sous-langage simplifié de Java. Le présent document a pour objectif de fournir une vue d'ensemble des fonctionnalités du compilateur Deca, accompagné d'instructions détaillées pour son utilisation. Il s'adresse à toute personne souhaitant écrire, compiler et exécuter des programmes Deca en exploitant les fonctionnalités offertes par notre compilateur.

Pour tester le compilateur, la commande suivante peut être utilisée : `decac -option fichier.deca`

Ce document couvrira les éléments suivants :

- Une présentation des différentes options disponibles pour la commande `decac`, permettant d'adapter son comportement selon les besoins de l'utilisateur.
- Une description détaillée de la base de tests fournie, qui permet de valider le comportement du compilateur et de garantir sa conformité avec les spécifications du langage.
- Une explication approfondie des limitations actuelles de notre compilateur, soulignant les fonctionnalités non supportées ou partiellement implémentées.
- Une présentation de l'extension ajoutée au compilateur pour enrichir les capacités du langage Deca.

En suivant ce manuel, vous serez en mesure de tirer pleinement parti des fonctionnalités de Deca tout en comprenant ses subtilités et ses contraintes.

2 Présentation du compilateur Deca

Le compilateur Deca repose sur une architecture modulaire organisée en trois étapes principales : l'analyse lexicosyntaxique, la vérification contextuelle et la génération de code assembleur. Ce découpage permet de séparer clairement les différentes responsabilités et de simplifier le développement, les tests, ainsi que l'extension du compilateur.

2.1 Analyse lexicosyntaxique

L'analyse lexicosyntaxique constitue la première étape du processus de compilation. Elle consiste à transformer le programme source écrit en Deca en un arbre de syntaxe abstraite (AST). Cette étape inclut :

- **La lexicographie** : Cette phase découpe le programme en unités lexicales ou lexèmes et identifie les mots-clés, opérateurs et identificateurs.
- **L'analyse syntaxique** : Elle vérifie que la structure des instructions respecte les règles grammaticales définies dans le langage Deca.

L'arbre généré par cette étape sert de base pour les vérifications ultérieures.

2.2 Vérification contextuelle

La deuxième étape, la vérification contextuelle, garantit la cohérence sémantique du programme,et la generation d'un arbre decore Cela inclut :

- **La vérification des types** : Chaque opération est vérifiée pour s'assurer qu'elle est effectuée entre des opérandes compatibles.
- **La gestion des identificateurs** : Cette passe vérifie que toutes les variables et méthodes utilisées sont correctement déclarées et accessibles dans leur contexte.
- **Les règles spécifiques aux classes et héritages** : Le compilateur s'assure que les relations entre classes respectent les contraintes du langage, par exemple l'interdiction de boucles d'héritage.

Durant cette étape, des informations de typage et des décorations supplémentaires sont ajoutées à l'AST pour préparer la génération de code.

2.3 Génération de code assembleur

La dernière étape consiste à transformer l'AST décoré en code assembleur exécutable sur une machine virtuelle abstraite. Pour notre projet, nous avons choisi d'implémenter une extension ARM, qui sera détaillée dans une section ultérieure de ce manuel. Cette extension permet de générer du code efficace et adapté aux architectures modernes.

Cette étape inclut :

- **La traduction des expressions et instructions** en instructions assembleur.
- **La gestion des registres et de la pile** pour l'exécution des programmes.

3 Compilation d'un programme Deca

La compilation d'un programme écrit en langage Deca se fait à l'aide de l'outil `decac`, qui permet de transformer un fichier source `.deca` en code assembleur,qui s'execute a l'aide de ima. Cette section détaille les commandes disponibles, ainsi que les options associées.

3.1 Utilisation de `decac`

Pour compiler un programme Deca, la commande de base est la suivante :

```
decac fichier.deca
```

Cette commande produit un fichier assembleur nommé `fichier.ass` dans le même répertoire. Ce fichier peut ensuite être exécuté sur une machine virtuelle avec l'outil `ima`.

```
ima fichier.ass
```

Si le programme contient des erreurs, `decac` affiche des messages d'erreur explicites indiquant leur nature et leur localisation dans le code. Ces messages permettent de corriger rapidement les problèmes détectés. Une section ultérieure de ce manuel est dédiée à l'explication des différents types de messages d'erreurs que le compilateur peut générer.

3.2 Options disponibles

Le compilateur `decac` supporte plusieurs options qui permettent de personnaliser le processus de compilation :

- `-b` : Affiche des informations sur le compilateur, notamment l'equipe ayant développé le projet, puis quitte immédiatement. Cette option est utile pour vérifier que le compilateur est correctement installé et pour identifier l'équipe ayant développé le projet.

  ```
  decac -b
  ```

  Exemple de sortie :

  ```
  Project developed by Group 26.
  ```

- `-p` : Effectue uniquement une analyse lexicosyntaxique. Cette option est utile pour détecter rapidement des erreurs dans le code.

  ```
  decac -p fichier.deca
  ```

- `-v` : Permet d'effectuer l'analyse lexicosyntaxique et contextuelle du programme, sans afficher l'arbre décoré ou le fichier assembleur généré.

  ```
  decac -v fichier.deca
  ```

- `-r n` : Limite le nombre de registres utilisés à `n` (entre 4 et 16).

  ```
  decac -r 8 fichier.deca
  ```

- `-d` : Active le mode débogage avec plusieurs niveaux de détails. L'option peut être répétée jusqu'à 4 fois pour augmenter les traces de débogage.

- `-n` : Désactive certains tests d'exécution liés aux erreurs sémantiques (ex. : division par zéro, débordement arithmétique, absence de `return`, conversion de type impossible, déréférencement de `null`) et aux dépassements des limites machine (ex. : débordement de mémoire).

  ```
  decac -n fichier.deca
  ```

- `-P` : Active la compilation parallèle lorsque plusieurs fichiers source sont fournis. Cette option permet de lancer simultanément la compilation de ces fichiers, réduisant ainsi le temps total de compilation

3.3 Utilisation avancée

Pour tester des cas spécifiques, il est recommandé d'utiliser les options `-d` et `-v` en combinaison, car elles permettent de suivre le processus de compilation étape par étape.

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

4 Limitations et Particularités du Compilateur

Bien que le compilateur Deca implémente la majorité des fonctionnalités nécessaires à la création de programmes efficaces, certaines limitations et particularités sont à noter, qui diffèrent parfois d'autres langages comme Java.

4.1 Limitations du Langage Deca

- **Déclaration des variables :** Une fois une instruction placée après une déclaration de variables, il n'est plus permis de déclarer de nouvelles variables dans le même bloc de code. Par exemple :

```
{
    int a = 5; // OK
    println(a);
    int b = 10; // Erreur : déclaration après une instruction
}
```

- **Déclaration des champs (fields) :** Si un champ est déclaré dans une superclasse, il est interdit de déclarer un champ avec le même nom dans une sous-classe. Par exemple :

```
class A {
    int x;
}
class B extends A {
    int x; // Erreur : redéclaration interdite
}
```

- **Redéfinition des méthodes :** Contrairement à Java, Deca interdit de modifier la signature d'une méthode lors de sa redéfinition dans une sous-classe. Par exemple :

```
class A {
    int m() {
        return 42;
    }
}
class B extends A {
    void m() { // Erreur : modification du type de retour
        println("Erreur !");
    }
}
```

4.2 Limitation de notre Implémentation

Notre implémentation du compilateur Deca a permis de compléter entièrement l'analyse lexicosyntaxique et contextuelle, y compris la génération de l'arbre décoré. Ces étapes fondamentales sont robustes et permettent de détecter efficacement les erreurs syntaxiques et sémantiques,Cependant, notre génération de code assembleur présente certaines limitations, bien que la majorite des fonctionnalités clés aient été correctement implémentées. Voici un résumé des aspects pris en charge et des limitations identifiées :

- **Langage Deca sans objet :** Une large partie du langage Deca sans objet a été prise en charge avec succès. Cependant, des imprécisions subsistent dans la gestion des nombres en virgule flottante, notamment dans les cas limites et les calculs nécessitant une précision accrue.
- **Langage Deca avec objet :**
  - Les classes simples sont entièrement gérées, incluant les opérations internes telles que l'accès aux champs, l'appel des méthodes, et les manipulations de leurs instances.
  - La gestion de la table des méthodes et son initialisation est correctement assurée, quel que soit le mode d'exécution du langage.

Remarque : Lors d'un push, une vérification concernant les déclarations de variables a été involontairement supprimée. Cette vérification permettait de détecter si une variable était de type 'void', ce qui constitue une erreur contextuelle. Néanmoins, ce cas est correctement pris en charge dans la gestion des déclarations de champs à l'intérieur des méthodes.

5 Messages d'erreurs

Notre compilateur a été conçu avec une gestion d'erreurs robuste et précise, permettant de guider les développeurs dans la correction de leurs codes. Les erreurs sont catégorisées de manière claire pour identifier rapidement la nature du problème et offrir des solutions appropriées. Voici les principales caractéristiques de notre gestion d'erreurs :**lexicales**, **syntaxiques**, **contextuelles** et **d'exécution du code assembleur**.

5.1 Erreurs Lexicales

Les erreurs lexicales surviennent lorsque le compilateur ne parvient pas à reconnaître un *token* valide dans le code source.

Le message d'erreur associé est généralement présenté sous la forme suivante :

```
fichier.deca :ligne:colonne :token recognition error at "token"
```

Ce message précise :

- le fichier concerné ;
- la position exacte de l'erreur (ligne et colonne) ;
- le *token* non reconnu.

Exemple

Supposons le code suivant dans un fichier `example.deca` :

```
int @variable = 5;
```

Dans cet exemple, le caractère `@` n'est pas un symbole valide pour nommer une variable. Cela entraînera une erreur lexicale. Le message affiché par le compilateur sera :

```
example.deca :1:5 :token recognition error at "@"
```

5.2 Erreurs Syntaxiques

Les erreurs syntaxiques apparaissent lorsque le code source ne respecte pas la structure grammaticale du langage.

Le message d'erreur associé peut être présenté sous l'une des formes suivantes :

```
fichier.deca :ligne:colonne:left-hand side of assignment is not an lvalue
```

ou

```
fichier.deca :ligne:colonne:no viable alternative at input '...'
```

Ces messages indiquent :

- une erreur dans la partie gauche d'une affectation ;
- une syntaxe incorrecte ou ambiguë ne correspondant à aucune règle définie.

Exemples

Supposons le code suivant dans un fichier `example1.deca` :

```
(1 + 1) = 2;
```

Le compilateur affichera :

```
example1.deca:1:0:left-hand side of assignment is not an lvalue
```

Supposons le code suivant dans un fichier `example2.deca` :

```
{int x=1}
```

Le compilateur affichera :

```
example2.deca:1:8no viable alternative at input '}'
```

5.3 Erreurs Contextuelles

Les erreurs contextuelles apparaissent lors de l'analyse sémantique du code. À ce stade, bien que le code puisse être lexicalement et syntaxiquement correct, il peut encore contenir des incohérences ou des violations des règles sémantiques définies par le langage.

Voici quelques exemples des erreurs contextuelles détectées par le compilateur Deca, accompagnés de leurs messages d'erreur :

Description de l'erreur	**Message d'erreur**
Une variable x est déclarée plusieurs fois.	`Variable 'x' is already declared in this scope`
Une variable ne peut pas avoir `void` comme type.	`cannot declared a variable with type void.`
L'expression retournée par une méthode ne correspond pas au type attendu.	`Type incompatible: attendu ' ', trouvé ' '`
Une expression n'est pas du type attendu (sauf `int` converti en `float`).	`Type incompatible: attendu ' ', trouvé ' '`
Seuls les types `string`, `int`, et `float` peuvent être imprimés.	`illegal argument for print`
Les opérandes d'une condition `if` ou `while` doivent être booléens.	`Type of the condition must be boolean`
Les opérandes d'une opération arithmétique **op** doivent être de type `int` ou `float`.	`Operation arithmetic : op failed : one or both of operands is not of type int or float`
Les opérandes d'une opération booléenne doivent être de type `boolean`.	`Boolean operation op failed: both operands should be of type boolean`
L'identifiant **id** n'est pas un type prédéfini ou une classe définie.	`Identifier id is not defined`
L'identifiant suivant `extends` est un type primitif.	`Super-class is not a valid class`
Une classe **Class** est déclarée plusieurs fois.	`class 'Class' already declared`
Un champ **field** est déclaré plusieurs fois ou porte le même nom qu'une méthode existante dans la meme classe.	`field 'field' already defined in the current class`
Un champ **field** porte le même nom qu'une méthode existante ou un champ dans une superclasse **superClass**.	`field 'field' already defined in superclass superClass`
Un champ ne peut pas avoir `void` comme type.	`Cannot declare a field with type void`
Une méthode **meth** est redéfinie dans la même classe ou porte le même nom qu'un champ existant.	`La methode meth est déjà déclaré dans l'environnement local`

Une méthod **meth** essaie d'écraser une autre avec une signature différente.	`la signature des paramètres de la méthode methn'est pas compatible avec celle de la méthode redéfinie.`
L'identifiant **id** suivant `new` n'est pas un nom de classe valide.	`Identifier 'id' is not defined`
Une méthode est appelée sans préciser la classe ou l'instance, en dehors d'une déclaration de classe.	`Cannot call method without an object outside a class context`
L'identifiant à gauche d'un appel de champ n'est pas une instance de classe.	`The left part of '.' must be an object`
L'identifiant **id** à droite d'un appel de champ ne correspond pas à un champ existant.	`The member 'id' is not defined in class or its superclasses`
Une instance de classe est utilisée alors qu'un type primitif est attendu.	`expected type found class`

5.4 Erreurs de génération de code

Le compilateur prend en compte plusieurs types d'erreurs susceptibles de survenir lors de l'exécution du code généré. Ces erreurs incluent :

- **Erreur d'entrée/sortie :** `Error: Input/Output error`. Cette erreur peut se produire lorsque le programme tente d'accéder à des ressources d'E/S non disponibles ou non autorisées.
- **Dépassement de capacité arithmétique :** `Error: Overflow during arithmetic operation`. Cette erreur se produit lorsqu'une opération dépasse la plage des valeurs autorisées pour le type concerné (par exemple, dépassement de la capacité d'un entier).
- **Division par zéro :** `Error: Division by zero`. Cette erreur est générée lorsqu'une division est effectuée avec un dénominateur égal à zéro.
- **Débordement de pile :** `Error: Stack Overflow`. Cette erreur peut survenir dans deux cas :
  - La pile est pleine, en raison d'une allocation excessive ou d'appels récursifs infinis.
  - Une erreur dans la gestion de la mémoire, conduisant à un dépassement non contrôlé.
- **Déréférencement de pointeur nul :** `Error: Null pointer dereference`. Cette erreur survient lorsqu'un programme tente d'accéder à une adresse mémoire via un pointeur non initialisé ou nul.

Ces erreurs sont détectées et gérées uniquement si les options du compilateur n'ont pas désactivé les vérifications associées (via `getNoCheck()`).

6 Utilisation de l'extension

6.1 Prérequis : Installation des Outils

Avant de commencer, installez les outils nécessaires en exécutant les commandes suivantes :

6.2 Prérequis : Installation des outils

Pour installer les outils nécessaires, exécutez les commandes suivantes :

Listing 1: Installation des outils nécessaires

```
sudo apt update
sudo apt install qemu-user          # Emulateur pour executer des programmes
sudo apt install gcc-arm-linux-gnueabi  # Compilateur GCC pour ARM
sudo apt install libc6-armel-cross  # Bibliotheque C pour l'architecture AR
```

6.3 Génération du code ARM

Pour générer un fichier assembleur ".s" ARM, utilisez l'option suivante du compilateur Deca :

decac -a fichier.deca;

6.4 Exécution du code ARM

Pour assembler, compiler et exécuter le programme ARM, utilisez le script suivant depuis le répertoire racine :

```
./src/test/script/script_arm.sh <fichier.s>
```

7 Conclusion

Ce document a pour objectif de guider les utilisateurs dans l'utilisation du compilateur Deca. En cas de problème ou pour toute question supplémentaire, veuillez contacter l'équipe de développement.