Manuel Utilisateur

Compilateur Deca - GL31

Table de matières

Introduction

Bienvenue dans le manuel utilisateur du Compilateur Deca, un projet passionnant qui vous plonge dans le monde captivant de la compilation et de la création de langages de programmation. Ce manuel a été conçu pour vous accompagner tout au long de votre expérience avec notre compilateur Deca, en vous fournissant des informations détaillées, des conseils pratiques et des ressources essentielles pour tirer le meilleur parti de cet outil.

I. Présentation du compilateur Deca

Le compilateur Deca vise à créer un compilateur pour le langage Deca, un sous-ensemble de Java, avec quelques variations. Notre objectif est d'offrir une solution robuste, respectant les normes de qualité les plus strictes. Le compilateur Deca se décompose en trois étapes clés, chacune contribuant à la transformation du code source Deca en un programme exécutable sur une machine abstraite. Ces étapes sont:

Étape A : Analyse Lexicale et Syntaxique, Construction de l'Arbre Abstrait

A. Analyse lexicale pour identifier les "mots" dans le programme source.
B. Analyse syntaxique pour vérifier la structure grammaticale correcte.
C. Construction simultanée de l'arbre abstrait, représentation structurée du programme.

Étape B : Vérifications Contextuelles et Décoration de l'Arbre Abstrait

D. Réalisation de vérifications contextuelles selon la syntaxe contextuelle de Deca.
E. Modification et décoration de l'arbre abstrait en préparation pour l'étape C.
F. Trois passes sur l'arbre abstrait pour vérifier la déclaration des identificateurs, les types d'expressions, etc.

Étape C : Génération de Code

G. Génération de code assembleur pour une machine abstraite.
H. Utilisation d'un interprète de machine abstraite pour exécuter le code généré.
I. Deux passes sur l'arbre abstrait pour construire la table des méthodes des classes et coder le programme

II. Comment utiliser le compilateur

1. Utilisation du compilateur

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

- Options disponibles

1. **decac :** afficher les informations d'utilisation
2. **-b** (banner) : Affiche une bannière indiquant le nom de l'équipe qui a fait le projet
3. **-p** (parse) : Arrête decac après l'étape de construction de l'arbre et affiche la décompilation de ce dernier. Si un seul fichier source est fourni, la sortie doit être un programme Deca syntaxiquement correct.
4. **-v (**vérification) : Arrête decac après l'étape de vérification. Aucune sortie n'est générée en l'absence d'erreur.
5. **-n** (no check) : supprime les tests à l'exécution spécifiés dans la "remarque" de la sémantique de Deca.
6. **-r X** (registers) : limite les registres banalisés disponibles à R0 ... R{X-1}, avec 4 <= X <= 16
7. **-P** (parallel) : s'il y a plusieurs fichiers sources, lance la compilation des fichiers en parallèle (pour accélérer la compilation).

remarque :

- Pour les noms des fichiers , le suffixe .deca est obligatoire.
- Les options '-p' et '-v' sont incompatibles.
- L'option -P va compiler tous les fichiers, mais s'il y a une erreur dans l'un des fichiers, la génération de code des autres fichiers ne sera pas produite.
- L'exécution avec l'option -n permet permet de sauter les vérification suivant suivant :
  - débordement arithmétique sur les flottants (inclut la division flottante par 0.0)
  - absence de return lors de l'exécution d'une méthode
  - division entière (et reste de la division entière) par 0.
- s il ya une erreur dans la command decac , les informations d'utilisation seront affiché

☐ Après la génération de code par decac un file de suffix .ass sera produit , on exécute ce fichier par : **ima file.ass**

.

2. Validation du compilateur par les tests

Pour s'assurer que le compilateur marche, un ensemble de tests manuels a été écrit (spécifique à chaque étape de la compilation), ainsi que quelques scripts pour le lancement et la vérification automatique de ces tests.

Pour voir le contenu des fichiers de tests écrits, on peut se rendre dans le répertoire *src/test/deca/* puis choisir l'étape qui nous intéresse . Les scripts de test quant à eux se trouvent dans *src/test/script*.

Tous les scripts de tests écrits ont été ajoutés dans le pom.xml. Ainsi, par la commande "*mvn verify"* à la racine du projet, tous les scripts de test seront lancés.
Une autre possibilité serait d'exécuter la commande "*mvn -Djacoco.skip=false verify"* pour lancer les test en traçant la couverture , puis "*firefox target/site/jacoco/index.html*" pour visualiser le résultat.

pour lancer manuellement un test sur un fichier deca il faut :

- se rendre dans le répertoire de test concerné : par exemple pour un test valide de la partie B(sur la syntaxe contextuelle) , se trouvant à la racine du projet, faire dans le terminal un cd *src/test/deca/context/valid/created/objet/* .
- lancer le test concerné (pour notre cas *test_context <nom du fichier>*)

pour lancer automatiquement un script particulier on peut :

- se rendre dans *src/test/script* puis lancer le test avec : ./<nom du script> (par exemple *./all_synt.sh*)

Note: il est possible d'avoir une erreur du type : Permission denied lors du lancement d'un script. Dans ce cas, il faudrait exécuter au préalable *chmod +x <nom du script>* pour donner la permission nécessaire.

Pour les tests supposés valides, il réussit s'il affiche le nom du fichier accompagné de *passed* ou *reussi* dans le terminal. Pour les tests supposés invalides , le script affiche *Échec attendu pour le [test context | test_synt] dans <nom du fichier>*.

III. Limitation du compilateur

1. Pour les print/println , seuls les entiers et les flottants ont la possibilité d'être affichés, ainsi que les chaînes de caractères . Par contre , on n' a pas la possibilité d'afficher des booléens
2. L'option -d du compilateur qui active les traces du debug n'a pas été implémenté.
3. Les entiers ne doivent pas contenir plus de 31 chiffre , sinon le Parser affiche l'erreur suivante : "`OverFlow : int number is too long"`
4. Concernant les classes , les champs doivent être déclarés public ou protected . Attribuer un nom de méthode à un champ de classe ne génère pas d'erreur comme il le devrait .

5. De la même manière que le int , on peut déclarer autant de chiffre après la virgule si l'on souhaite mais seuls les premiers chiffres après la virgule seront pris en compte.
6. On ne peut pas déclarer un float trop petit , le parser génère une erreur :" Overflow , Float number is too small "
7. Les opérations arithmétiques concernant des floats génèrent l'erreur "Overflow during arithmetic operation lorsqu'un certain seuil a été franchi .
8. Quand un programme utilise trop de pile , une erreur de débordement de la pile est généré
9. L'option -n du compilateur est incomplète et ne répond pas complètement au cahier des charges.

IV. Messages d'erreur

1. Erreur de Lexicographie

Les erreurs lexicales surviennent lorsque le compilateur est incapable de générer le token correspondant à l'information fournie. Ces erreurs seront signalées dans les fichiers .lis, générés automatiquement après l'exécution du test automatique.

Le format de l'erreur est le suivant :`.<file:/emplacement/fichier><line:a><col:b>: erreur de reconnaissance du token`

Où :

- `a` est le numéro de la ligne où l'erreur se produit dans le fichier.
- `b` est le numéro de la colonne où l'erreur se produit dans le fichier.

2. Erreurs Syntaxiques

Les erreurs syntaxiques sont des erreurs de structure dans le code source. Elles sont détectées lors de la compilation ou de l'interprétation du code, et peuvent être causées par une variété de facteurs, tels que :

- Une faute de frappe
- Une erreur de ponctuation
- Une utilisation incorrecte d'un mot-clé ou d'une fonction
- Une structure de contrôle incorrecte

Recognition Exception

Une "Recognition Exception" est une erreur syntaxique spécifique qui se produit lorsque le compilateur ou l'interpréteur ne parvient pas à reconnaître un élément du code source. Cette erreur peut être causée par une faute de frappe, une erreur de ponctuation ou une erreur de structure de contrôle.

3. Erreurs Contextuelles

Pendant la compilation des programmes deca ou de l'exécution du code assembleur généré, des messages d'erreur peuvent être retournés si ceux-ci ne respectent pas certaines règles établies dans la syntaxe ou la sémantique de deca. Ci-dessous les erreurs principales qui peuvent être renvoyées ainsi que les configurations qui les provoquent.

Configuration	Exemple	Message d'erreur
Variable non déclarée	{int a = x;}	"x" is not declared
Type de variable invalide	{void a;}	"Invalid type to declare variable"
Double définition d'une variable	{int a; int a;}	"double definition of variable : a "

Invalid assign	class A {
	}
	
	class B extends A {
	}
	
	{
	
	A a ;
	B b ;
	b = a;
	}
		Invalid assign : the two expressions has not the same type

appel de méthode incorrecte	class A {
	int x ;
	int getX() {
	return x ;
	}
	void setX(int x)
	{
	this.x = x ;
	}
		methodcall : appel implicite de methode dans le main

	}
	
	{
	A a = new A() ;
	setX() ;	

cast de classe	class A {
	int x;
	
	}
	
	class B {
	
	}
	
	{
	
	A a ;
	B b ;
	a = (A)(b);
	}
		“invalid cast”

cast d'un type en un autre	class A {
	int x ;
	void setX(int x)
	{
	}
	}
	
	{
	A a = new A() ;
	int x = (int) (a.setX(5)) ;
	
	}
		<line:13><col:10>:Invalid initialization for int field/variable

Nom de classe invalide	classe int{}	“la classe ne peut avoir ce nom”
Redéfinition d’une méthode pas respectée	class A{ void m(){}} class B extends A { void m(int x){} }	“override of method m is not respected
Classe inexistante	classe B extends A{}	“the class A is not defined”
Champ redéfini comme une méthode	class A{int x;} class B extends A { void x(){} }	“x is already defined in superclasses but not as a method”

appel implicite d'une méthode	class A{ void m(){}} {m();}	"appel implicite de méthode dans le main"
Appel d'une méthode avec la signature incorrecte	class A{ void m(){}} {A a; a.m(1);}	"Incorrect Method signature"

champ protege	class A {
	protected int x;
	}
	class X {
	int m()
	{
	A a = new A();
	return a.x ;
	}
	}
		Erreur contextuelle : x est protegé

nom de classe invalide	class null {
	
	}
		class_null.deca:6:0: la classe ne peut avoir ce nom

return incompatible	class A {
	int m(){
	return true;
	}
	}
		"The return type and method type are not compatible"

4. Erreur d'exécution du code assembleur

- **Error : Erreur Cast** : Cela se produit lorsque le changement de type d'une classe à une autre échoue, probablement parce que la classe en question n'est pas une sous-classe de l'autre (ou une instance). Voici un exemple :

```
class A {}
class B extends A {  }
{
    A a = new A();
    B b=(B)(a);
}
```

- **Matherror : ZerDivision** : Cela se produit lorsqu'on essaye de faire une division par Zéro. Voici un exemple :

```
class A{
        int x = 7/0 ;
}
```

- **Return Error** : Sortie de méthode sans return :cela se produit lorsqu'on fait un appel à une méthode de type de retour différent de void et qu'on ne passe pas par l'instruction return. Voici un exemple:

```
class A{
        int fct(){
            int x = 1;
        }
}
{
  A  a = new A();
   a.fct();
}
```

- **StackOverflow Error** : se produit lorsque l'espace mémoire de tas est insuffisant

```
class A
{
        A a = new A();
}
```

```
{
        A a = new A();
}
```

V. Conclusion

La réalisation du compilateur Deca fut non seulement un outil technique, qui nous a permis d'approfondir nos connaissances dans le domaine de la compilation, des normes de qualité les plus élevées en programmation, mais surtout une aventure de travail en équipe et de respect des livrables itératifs attendu produit. Nous avons certaines difficultés au cours de ce projet, mais nous sommes ravis d'avoir partagé cette expérience passionnante avec notre équipe de développement. Nous sommes impatients de présenter ce produit qui fut le résultat d'un travail acharné.