Projet Génie Logiciel : Manuel Utilisateur

Groupe 7 Equipe 34

BESSET Noé
FALL Rokhaya Yvette
GILLOT Aurélie
LESPINE Marilou
ROCHE Faustine

22 janvier 2024

Table des matières

1 Introduction

Ce document a pour but d'offrir une référence concise aux utilisateurs actuels et potentiels de notre compilateur, qui disposent déjà des spécifications détaillées du langage Deca. Elle englobe plusieurs aspects, visant à faciliter une utilisation optimale du compilateur.

Nous aborderons d'abord les limitations spécifiques à notre implémentation, mettant en lumière les portions du langage qui peuvent être insuffisamment ou incorrectement implémentées.

De plus, cette documentation comprendra une liste détaillée des messages d'erreur susceptibles d'être renvoyés à l'utilisateur, couvrant les erreurs de lexicographie, de syntaxe hors-contexte, de syntaxe contextuelle et d'exécution du code assembleur. Les configurations et les conditions spécifiques entraînant ces messages d'erreur seront clairement définies.

Nous aborderons enfin les modes opératoires pour utiliser nos extensions (options de la commande decac, configurations à utiliser, etc.)

2 Les limitations ou les points propres à l'implémentation de notre compilateur

Dans cette section, nous examinerons les limitations et les aspects spécifiques à l'implémentation de notre compilateur. Nous identifierons ces limitations, détaillerons les effets qu'elles peuvent avoir sur l'utilisateur final. Cette analyse permettra aux utilisateurs de mieux appréhender les fonctionnalités du compilateur et d'ajuster leurs attentes en conséquence.

Par manque de temps, nous n'avons pas implémenté les fonctionnalités Cast et InstanceOf pour la partie "génération de code assembleur".

Nous n'avons pas non plus géré l'underflow des float, lorqu'un float est trop proche de zéro. Cela peut causer des soucis et beaucoup de calculs si l'utilisateur souhaite diviser par un float très petit.

3 Les messages d'erreur

Cette section se consacre à l'examination des messages d'erreur générés par notre compilateur, couvrant divers types tels que les erreurs de lexicographie, de syntaxe hors-contexte, de syntaxe contextuelle et d'exécution du code assembleur. Notre objectif est de fournir une liste complètes des messages d'erreurs susceptibles d'être renvoyés à l'utilisateur. En comprenant ces

messages d'erreur et leurs origines potentielles, les utilisateurs pourront diagnostiquer plus efficacement les problèmes lors de la compilation, améliorant ainsi leur expérience globale avec le compilateur.

3.1 Analyseur syntaxique & Vérificateur de la syntaxe contextuelle

Les messages d'erreur sont listées ci-dessous (2 messages d'erreur lexicographiques et 35 erreurs contextuelles).

3.1.1 Erreurs lexicographiques

- **mismatched input 'else' expecting 'print', 'println', 'printx', 'printlnx', 'while', 'return', 'if', 'readInt', 'readFloat', 'true', 'false', 'this', 'null', 'new', IDENT, '(', '', ';', '-', '!', INT, FLOAT, STRING** : Ce message d'erreur est dû au **if_then_else**
- **fr.ensimag.deca.syntax.DecacErrorListner.syntaxError(DecacErrorListner.java : - no token, no LocationException => using input.getSourceName() : line=11 src/test/deca/lexer/invalid/mots_reserves/test16.deca :11 :5 : token recognition error at : '°'** : Ce message d'erreur a plusieurs explications. D'abord nous avons le fait que **Le caractère "°" n'est pas reconnu par le lexer**.
  Mais aussi nous avons **list_int**, **OP_ARITH**, **OP_BOOL**, **PRINT**.

3.1.2 Erreurs contextuelles

Les **Règles** correspondent aux règles de la syntaxe contextuelle du langage Deca.

1. **identifier *identifier name* not declared** : l'identificateur utilisé doit avoir été défini préalablement (**Règle(0.1)**).
2. **type not defined** : le type utilisé n'est pas défini (**Règle (0.2)**).
3. **class *class name* already declared** : une classe du même nom a déjà été déclarée (**Règle (1.3)**).
4. **current class does not extend a class type** et **superclass undefined** : la classe n'étend pas une classe valide (**Règle (1.3)**).
5. **field *field name* already declared** : un champ du même nom a déjà été déclaré (**Règle (2.5)**).
6. **the name of the field is already used in superclass, but not as a field name** : une superclasse de la classe courante contient déjà une méthode de ce nom (**Règle (2.5)**).

7. **field cannot be void type** : un champ ne peut pas être de type void (**Règle (2.5)**).
8. **method *method name* already declared , signature or type of the overrided method is not the same as the original method** : la **Règle(2.7)** explique ces erreurs.
9. **parameter cannot be void type** : Cette erreur peut être expliquée par la règle **Règle (2.9)**.
10. **identifier *variable name* already declared** : Nous pouvons l'expliquer par le fait que la variable ne peut pas être déclarée plusieurs fois (**Règle (3.16) et (3.17)**).
11. **type is void** : le type ne peut pas être void (**Règle (3.17)**).
12. **invalid return argument** : L'argument du return ne peut pas être un void (**Règle (3.24)**).
13. **incompatibility for assignment** : le type des deux valeurs ne sont pas compatibles (**Règle (3.28)**).
14. **not boolean condition** : l'expression doit être booléenne (**Règle (3.29)**).
15. **erreur in print argument** : l'argument du print() doit être int, float ou string (**Règle (3.31)**).
16. **invalid types for comparison** : on peut comparer seulement des int et des float ou deux booléens (**Regle (3.33)**).
17. **invalid types for boolean operation** : on peut faire des comparaisons booléennes seulement entre deux valeurs booléennes (**Règle (3.33)**)
18. **invalid types for arithmetic operation** : on peut faire des opérations seulement entre des int et des float (**Règle (3.33)**).
19. **the input value is not an integer** : la valeur entrée apès le readInt doit être un entier (**Règle (3.35)**).
20. **the input value is not a float** : la valeur entrée apès le readInt doit être un float (**Regle (3.36)**).
21. **invalid types for modulo** : le modulo peut seulement être effectué avec deux int (**Règle (3.37)**).
22. **the operand is not a Boolean expression** : on peut appliquer l'opérateur Not seulement sur un booléen (**Règle (3.37)**).
23. **the operand is neither a float nor an integer** : on peut appliquer l'opérateur UnaryMinus seulement sur un float ou un entier (**Règle (3.37)**).

24. **types *cast type* and *type expression* aren't cast compatible** : les deux types ne sont pas compatibles pour effectuer un cast (**Règle (3.39)**).
25. **Wrong types for 'instanceOf' (expected left operand as Object and right operand as Class)** : on peut appliquer instanceOf que si l'opérande de gauche est un Objet (Class ou Null) et l'opérande de droite est une Class (**Règle (3.40)**).
26. **'New' is only callable on a class** : on ne peut appeler 'New' que sur un objet de type Class (**Règle (3.42)**).
27. **'This' has no reference** : on ne peut appeler This que dans une classe (**Règle (3.43)**).
28. **The object on which the field is called must be a class** : on ne peut appeler un field que sur une classe (**Regle (3.65)**).
29. **Class unknown** : l'expression doit avoir une définition (**Règle (3.65)**).
30. **Field isn't defined** : le champ doit avoir une définition (**Règle (3.65)**).
31. **Protected field can't be reached** : le champ protected doit être appelé dans une classe sous type de la classe le contenant, et le type de l'expression doit être un sous type de la classe courante (**Règle (3.66)**).
32. **Class called is not a subtype of the currentClass** : la classe appelée n'est pas une superclasse de la clase courante (**Règle (3.66)**).
33. **method is only callable on a type Class object** : une méthode peut être appelée que sur une classe (**Règle (3.71)**).
34. **The method called doesn't exist** : méthode non définie (**Règle (3.71)**).
35. **number of parameters for the method should be *signature size* and not *...*** : due à la **Règle (3.74)** qui montre que le nombre de paramètres dans un appel de méthode doit être le même que celui de la méthode appelée.

3.2 Génération de code assembleur

Les messages d'erreur de la partie génération de code sont au nombre de 6. Nous allons les détailler ci-dessous.

- **Overflow during arithmetic operation** : Ce message apparait Lors d'une addition, une soustraction ou une multiplication. L'erreur correspond à un resultat de l'opération arithmétique qui dépasse la taille des registres.

Elle apparait aussi Lors d'un modulo ou une division, l'erreur correspond à une division par 0.

- **Stack Overflow** : Elle apparait lorsque l'on cherche à empiler plus d'éléments dans la pile que cell-ci le permet.
- **Input/Output error** : On voit cette erreur lorsque le programme attends un input de l'utilisateur, que ce soit un entier ou un flottant, mais que l'input fourni ne respecte pas le format attendu
- **Dereferencing a null object** : le programme tente d'acceder à un champ ou une méthode d'un objet alors que cet objet a pour adresse null.
- **Heap Overflow** : Ce message apparait lorsque l'on cherche à créer un nouvel objet qui ne peut pas rentrer dans l'espace restant du tas
- **Method $(nom_de_la_methode) from class $(nom_de_la_classe) is supposed to return something** : Elle apparait lorsqu'une methode termine son execution sans retourner de valeur alors qu'elle est censée en retourner une de type non void

4 Le mode opératoire pour l'utilisation de nos extensions

Le processus d'utilisation des extensions de notre compilateur Deca, ainsi que les options associées à la commande decac, joue un rôle essentiel dans la compilation réussie des programmes Deca. Cette section détaillera le mode opératoire pour exploiter nos extensions, en mettant en lumière les configurations à utiliser avec la commande decac.

4.1 Ligne de commande

Le programme principal, appelé "decac", représente un compilateur Deca complet. Pour spécifier le fichier source, utilisez des chemins de la forme *<répertoire/nom.deca>*, avec le suffixe "*.deca*" obligatoire. Le résultat de la compilation, sauf erreur, doit être enregistré dans un fichier *<répertoire/nom.ass>* situé dans le même répertoire que le fichier source.

La syntaxe d'utilisation de l'exécutable "decac" est la suivante :

decac [[-p — -v] [-n] [-r X] [-d] [-P] [-w] ¡fichier deca¿...] — [-b]*

La commande "decac" sans argument affiche les options disponibles. Elle peut être appelée avec un ou plusieurs fichiers sources Deca. Les options incluent la bannière (-b), l'analyse syntaxique (-p), la vérification (-v), la

suppression des tests à l'exécution (-n), la limitation des registres (-r X), le mode debug (-d), la compilation parallèle (-P), et les avertissements (-w).

4.2 Formatage des messages d'erreur

Les messages d'erreur, qu'ils soient lexicaux, syntaxiques, contextuels, ou liés à des limitations éventuelles du compilateur, suivent un format standard :

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

Par exemple, une erreur au 4ème caractère de la ligne 12 serait notée comme suit :

fichier.deca :12 :4 : Identificateur "foobar" non déclaré (règle 6.12).

Cette section a détaillé le mode opératoire et les configurations associées à l'utilisation des extensions de notre compilateur Deca. La clarté sur les options de la commande decac ainsi que le formatage des messages d'erreur contribuent à une expérience utilisateur transparente et facilitent le débogage des programmes Deca. La compréhension de ces aspects est essentielle pour tirer pleinement parti des fonctionnalités du compilateur.

5 Conclusion

En conclusion, cette documentation vise à fournir un guide complet, offrant aux utilisateurs une compréhension du compilateur Deca. Elle devrait être une ressource pour exploiter au mieux les fonctionnalités du compilateur tout en comprenant ses limites et en diagnostiquant les éventuelles erreurs rencontrées lors de la compilation.