MINISTÈRE
DE L'ENSEIGNEMENT
SUPÉRIEUR
ET DE LA RECHERCHE
Liberté
Égalité
Fraternité

GRENOBLE INP UGA Ensimag

PROJET GÉNIE LOGICIEL 2023-2024

DOCUMENTATION UTILISATEUR

Ingénieur 2ème année

(Environ 12 pages)
Deadline: 22 Janvier 2024 à 20h00

Professeur encadrante:

Karine ALTISEN

Equipe projet :

- Stéphane KOUADIO
- Julian COUX
- Breno MORAIS
- Loan GATIMEL
- Hugo MERCIER

SOMMAIRE

INTRODUCTION

Bienvenue dans le manuel de l'utilisateur du compilateur Deca. Nous sommes ravis de vous accompagner dans la découverte de notre outil, conçu pour simplifier et optimiser le processus de transformation des programmes Deca en code Java exécutable.

Que vous soyez novice en programmation ou développeur chevronné, ce guide a pour objectif de rendre votre expérience avec le langage Deca et notre compilateur à la fois intuitive et productive. Nous avons conçu ce manuel en mettant l'accent sur la clarté et la pertinence des informations fournies, afin de faciliter votre utilisation du compilateur et de maximiser votre efficacité.

Dans les pages suivantes, vous découvrirez une description détaillée du compilateur du point de vue utilisateur. Nous détaillerons les commandes et options essentielles pour une utilisation optimale, ainsi que les divers messages d'erreurs que vous pourriez rencontrer et comment les interpréter. Nous aborderons également les éventuelles limitations de notre compilateur et fournirons des informations cruciales sur l'extension. En dernière partie vous trouverez une explication détaillée sur comment tester, de manière automatique, ce compilateur, si toutefois vous cherchez à lui apporter des modifications.

Nous sommes convaincus que ce manuel vous guidera de manière complète tout au long de votre utilisation du compilateur Deca. Si vous avez des questions ou des retours, n'hésitez pas à vous référer à ce document pour trouver les informations nécessaires. Nous vous souhaitons une expérience de développement agréable et fructueuse avec le compilateur Deca.

1. Description du compilateur du point de vue de l'utilisateur

Le compilateur, élaboré en Java, simplifie le processus de compilation et d'exécution du code Deca en générant un fichier assembleur (.ass). Ce fichier est ensuite interprété par la machine abstraite IMA pour produire la sortie standard.

Les fonctionnalités clés de notre compilateur incluent :

1. **Support Intégral du Langage :** Le compilateur prend en charge toutes les fonctionnalités du langage Deca, assurant une traduction précise vers du code Java.
2. **Rapidité de Compilation :** Il offre des temps de compilation rapides, optimisant ainsi l'efficacité du développement.
3. **Messages d'Erreur Informatifs :** En cas d'erreur, des messages détaillés et des suggestions sont fournis pour faciliter le débogage.
4. **Messages d'Erreur Clairs :** Les erreurs de compilation sont accompagnées de messages détaillés, facilitant l'identification rapide des problèmes par les utilisateurs.
5. **Tests Automatisés :** Une série de tests et de scripts sont disponibles pour valider le bon fonctionnement du compilateur. Cette approche automatisée permet une évaluation rapide et complète, contribuant à garantir la robustesse du compilateur Deca.

Cette conception orientée utilisateur assure une expérience de développement fluide. Le fichier assembleur généré (.ass) peut être interprété par IMA pour produire la sortie standard, permettant une exécution efficace du programme Deca sur notre machine abstraite.

2. Commandes et options

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, affichera les options disponibles.
On peut appeler la commande decac avec un ou plusieurs fichiers sources Deca.

Les options disponibles avec la commande **decac** sont définies comme suit :

- **b (banner)** : Affiche une bannière indiquant le nom de l'équipe.

- **p (parse)** : Interrompt **decac** après la construction de l'arbre syntaxique et affiche la décompilation de ce dernier. Si un seul fichier source est compilé, la sortie doit être un programme Deca syntaxiquement correct.
- **v (vérification)** : Arrête **decac** après l'étape de vérification et ne produit aucune sortie en l'absence d'erreur.
- **n (no check)** : Supprime les tests à l'exécution.
- **r X (registers)** : Limite les registres banalisés disponibles à R0 ... R{X-1}, avec 4 <= X <= 16.
- **d (debug)** : Active les traces de débogage. Répéter l'option plusieurs fois pour obtenir plus de traces.
- **P (parallel)** : Si plusieurs fichiers sources sont présents, lance la compilation des fichiers en parallèle pour accélérer le processus.
- **w (warnings) :** Autorise l'affichage de messages d'avertissement en cours de compilation.

Notons que les options **-p** et **-v** sont incompatibles.

L'option **-b** ne peut être utilisée que indépendamment, sans autre option et sans fichier source. Dans ce cas, **decac** se termine après avoir affiché la bannière. Si un fichier apparaît plusieurs fois sur la ligne de commande, il n'est compilé qu'une seule fois.

- On pourra ajouter une option '-w' autorisant l'affichage de messages d'avertissement (« warnings ») en cours de compilation.
- L'option '-b' ne peut être utilisée que sans autre option, et sans fichier source. Dans ce cas, decac termine après avoir affiché la bannière.
- Si un fichier apparaît plusieurs fois sur la ligne de commande, il n'est compilé qu'une seule fois

3. Messages d'erreurs

Les messages d'erreur (lexicales, syntaxiques, contextuelles, et éventuelles limitations du compilateur) sont formatées de la manière suivante :

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

Comme par exemple, erreur au 4ème caractère de la ligne 12 :

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

ou bien, erreur au début de la ligne 3 :

test.deca:3:1: Caractère '#' non autorisé

Dans notre programme, nous avons veillé à mettre l'essentiel des messages d'erreur et commentaires en anglais. Ainsi, notre programme peut être lu, compris et exécuté par n'importe qui.

a. Erreurs lexicales

Les erreurs lexicales sont des erreurs qui surviennent lors de l'analyse lexicale dans lesquelles le code source est divisé en tokens. Elles sont généralement dues à :

- des caractères ou des séquences de caractères qui ne sont pas reconnus comme des tokens valides du langage.

left-hand side of assignment is not an lvalue

b. Erreurs syntaxiques

Les erreurs syntaxiques surviennent lorsque le code ne respecte pas les règles de grammaire du langage, souvent dues à une organisation incorrecte des tokens. Parmi les erreurs les plus fréquentes, on trouve les oublis de point-virgule, les parenthèses mal fermées, les structures de contrôle mal formées, et autres.

Voici les listes d'erreurs que vous pourriez rencontrer :

- Lorsqu'un mauvais argument est passé en argument, nous avons un message d'erreur de la forme:

<nom de fichier.deca>:<ligne>:<colonne>: Wrong number of arguments in Signature

- Lorsqu'un type incompatible est passé en argument, nous avons un message d'erreur de la forme :

<nom de fichier.deca>:<ligne>:<colonne>: Incompatible Type in Signature

- Lorsqu'un fichier inclus n'est pas trouvé, un message d'erreur de la forme suivante est retourné.

<nom de fichier.deca>:<ligne>:<colonne>: include file not found

c. Erreurs contextuelles

Les erreurs contextuelles sont des types spécifiques d'erreurs sémantiques. Elles se produisent lorsque le code source viole les règles liées au contexte d'utilisation des éléments du langage, telles que la portée des variables, les règles de type et les contraintes d'usage spécifiques au langage.

- Utilisation de variables non déclarées
- Conflits de types
- Réaffectation à des constantes
- Appel de fonctions incorrect
- Opération sur des types non supportés

Exemple :

- Dans la classe **Cast**, on gère l'erreur qui peut se produire lorsqu'on a une incompatibilité au niveau du cast avec le message du type :
  <nom de fichier.deca>:<ligne>:<colonne>: type incompatible in Cast

- Dans la classe **DeclClass**, on gère l'erreur qui peut se produire lorsque nous déclarons une classe qui a déjà été déclarée auparavant. Et pour cela, le message d'erreur se présente comme suit :

  <nom de fichier.deca>:<ligne>:<colonne>: Error, Class already declared in DeclClass

- Lorsque dans une classe, nous faisons référence à une super classe qui n'est elle même pas déclaré ou inexistante, nous avons le message de la forme :

  <nom de fichier.deca>:<ligne>:<colonne>: superClass is not a valid Class in DeclClass

- Dans **DeclField**, lors de la passe 2 avec classe dont le type des attributs n'a pas été déclaré on obtient :
  <nom de fichier.deca>:<ligne>:<colonne>: type is Void in DeclField

- Dans **DeclField,** pour la déclaration des champs d'une classe, il ne faut pas qu'il soit déjà déclaré :

<nom de fichier.deca>:<ligne>:<colonne>: Error, field already declared in DeclField

- Dans **DeclField**, lors de l'utilisation d'un attribut d'une classe qui n'est pas défini, nous avons le message d'erreur de la forme :

<nom de fichier.deca>:<ligne>:<colonne>: env_exp_super(name) not defined in DeclField

- Dans la classe **DeclVar**, lorsqu'une méthode est déclaré avec la même signature qu'une autre, nous avons une erreur de la forme suivante:

<nom de fichier.deca>:<ligne>:<colonne>: Error, field already declared in DeclMethod

- Dans la classe **DeclVar**, lorsque le type d'une variable n'existe pas ou est nul le message d'erreur suivant est retourné:
  <nom de fichier.deca>:<ligne>:<colonne>: Error of type in DeclVar

- Dans la classe **DeclVar**, lorsqu' une variable, fonction, ou un autre élément est déclaré plus d'une fois dans le même contexte, entraînant une exception DoubleDefException.
  <nom de fichier.deca>:<ligne>:<colonne>: Error, type already declared in Declvar

- Dans la classe **Identifier**, lorsqu'un type invalide est fourni, un message d'erreur est généré de la forme suivante :

<nom de fichier.deca>:<ligne>:<colonne>: <Type> is an invalid type in Identifier

- Pour ce qui concerne une variable utilisé qui n'a pas été initialisé, nous implémentons dans la classe **Initialization** un message d'erreur de la forme :

<nom de fichier.deca>:<ligne>:<colonne>: not the Type expected in Initialization

- Pour ce qui concerne les méthodes d'une classe, lorsque des variables sont utilisé à l'intérieur et non déclaré, nous avons des message de la forme:

<nom de fichier.deca>:<ligne>:<colonne>: Liste des déclarations de variables est null

- Pour ce qui est d'une absence d'instruction, nous avons des message d'erreur de la forme :

<nom de fichier.deca>:<ligne>:<colonne>: Liste des déclarations des instruction est null

Lorsqu'une méthode doit renvoyer un élément mais qu'il manque le return, un message d'erreur est retourné sous la forme:

<nom de fichier.deca>:<ligne>:<colonne>: Method without Return

d. Erreurs de Runtime

Les erreurs de runtime sont des erreurs qui ne sont pas détectées durant la compilation mais apparaissent lors de l'exécution du programme compilé. Elle ont toutes été implémenté dans la classe **Program**

Exemple :

- Lorsqu'on ne peut pas allouer d'espaces dans une pile, un message d'erreur de la forme suivante est retourné :

Erreur: Pile pleine

- Lorsqu'on veut accéder à une méthode d'une classe non initialisé, un message d'erreur de la forme suivante est retourné :

Erreur: Déréférencement de null

- Input/Output error

Erreur: Input/Output error

- Lorsqu'on ne peut pas allouer d'espaces dans le tas, un message d'erreur de la forme suivante est retourné :

Erreur: Tas plein

- En cas de division par zéro, nous avons un message d'erreur de la forme :

Erreur: Division par zéro

- Flottant imprécis

Erreur: Résultat n'est pas codable sur un flottant

4. Implémentation et Limitations

Notre compilateur prend en charge la compilation de différents types de programmes, notamment ceux sans objet et ceux avec objet.

Il autorise l'inclusion de fichiers externes avec include, l'affichage de valeurs, la réalisation de calculs complexes, la création de classes et méthodes, etc…

En particulier, pour les programmes sans objet, l'implémentation du compilateur autorise une utilisation complète de ces programmes. Nous avons veillé à ce que le compilateur traite tous les types d'erreurs et reconnaisse toutes les subtilités du langage. Ainsi, quel que soit votre programme deca, le compilateur devrait être en mesure de le compiler avec succès ou de vous informer en cas d'erreur.

En ce qui concerne le langage orienté objet, l'implémentation du compilateur autorise la création de classes et de méthodes, ainsi que l'héritage entre les classes. Ce qui garantit une prise en compte des principales subtilités du langage objet de Deca. Nous avons également implémenté la gestion de l'espace mémoire (le tas et la pile), pour offrir la meilleure efficacité possible en maîtrisant le nombre de registres utilisés.

Pour améliorer l'automatisation des tests, nous pourrions nous inspirer des scripts déjà existants afin de mettre en place, à l'avenir, une automatisation des tests globale pour l'ensemble du compilateur.

En ce qui concerne les opérations de conversion (Cast), notre prise en charge est actuellement limitée à la conversion d'un float en int et inversement. Cependant, il est important de noter que la gestion d'erreur pour d'autres types de conversion n'est pas optimale pour le moment, ce qui pourrait entraîner un crash du programme. De plus, pour les opérations de conversion liées aux classes, leur implémentation n'a pas encore été réalisée.

Cependant, bien que la compilation de la plupart des programmes normaux soit possible, certains points restent à améliorer. Notamment, un des problèmes les plus évidents avec la gestion des erreurs concerne le test de la pile. Chaque bloc devrait vérifier si la pile peut gérer les opérations, mais notre implémentation a un problème: le test de la pile ne prend en compte que les opérations immédiates dans le bloc, ignorant complètement les opérations dans la pile lorsqu'on se trouve à l'intérieur d'une méthode. Ces lacunes dans la gestion des erreurs peuvent être des points critiques à adresser pour renforcer la fiabilité du compilateur.

5. Extension TRIGO du compilateur

Notre compilateur possède une bibliothèque standard qui définit une classe Math. Cette classe implémente les fonctions trigonométriques. La liste des fonctions implémentées est la suivante :

- float sin(float f)
- float cos(float f)
- float asin(float f)
- float atan(float f)
- float ulp(float f)

Ces fonctions sont implémentées dans le fichier **Math.decah** dans le répertoire ***src/main/resources/include***

L'implémentation des méthodes cherche la meilleure approximation possible permise par la représentation des flottants simple précision, à l'intérieur du codomaine de la fonction mathématique.

Pour inclure la bibliothèque standard,il faut utiliser la commande #include qui permet de partager du code entre plusieurs programmes.

Une documentation de l'extension de nôtre compilateur décrivant les méthodes employées, la précision et le mode d'emploi de ces fonctions trigonométriques est disponible.

6. Comment tester le compilateur

Afin de garantir le bon fonctionnement de notre compilateur, nous avons mis à disposition de l'utilisateur une série de tests et de scripts. Ces tests, conçus pour valider le bon fonctionnement du compilateur, sont le fruit d'un travail approfondi. Répartis dans le répertoire ***/src/test/deca/...***, ils ont été soigneusement élaborés pour couvrir chaque étape critique de la compilation.

Ces scénarios de validation et d'invalidation ont été essentiels pour assurer la qualité de notre compilateur Deca. Leur exécution méthodique a joué un rôle déterminant, évaluant la performance du compilateur dans divers contextes. Explorez-les pour une utilisation optimale, consultez la documentation pour des détails sur leur exécution et interprétation. Ces tests sont au cœur de notre engagement envers la qualité du compilateur Deca.

Dans le cadre de notre compilateur, les tests fournis pour les trois étapes (syntaxe, contexte et génération de code) sont organisés de manière similaire. Les tests

destinés à la validation sont regroupés dans des répertoires 'valid', tandis que ceux conçus pour déclencher des erreurs sont regroupés dans des répertoires 'invalid'.

a. Utilisation du script Principal

Pour faciliter la visualisation de ce qui se passe dans chaque étape de la compilation, nous mettons à disposition du client un script *./scriptTest.py*, un exécutable python qui prend en argument un fichier .deca et réalise individuellement les 3 étapes de la compilation et les affiches sur le terminal.
Pour l'utiliser, il sera d'abord nécessaire de rendre exécutable le script avec la commande : ***chmod +x scriptTest.py***, qui est à exécuter à la racine du compilateur.

Ensuite, ce script prend en paramètre un des fichier de test, par exemple :

./scriptTest.py src/test/deca/context/…/test_complet_ultime.deca

ou (si la commande chmod n'a pas été exécuté)

python3 scriptTest.py src/test/deca/context/…/test_complet_ultime.deca

Cela permet d'avoir une visualisation rapide du programme et s'il y a une erreur de pouvoir l'identifier rapidement et précisément peu importe l'endroit où elle se trouve.
Tous nos fichiers de test ont été ajouté au script disponible qui eux même ont été ajouté au fichier ***pom.xml***, ce qui permet de tous les lancer simplement en exécutant la commande : ***mvn test***

b. Script de validation automatique

Nous avons également mis au point plusieurs scripts permettant de valider automatiquement la compilation d'un programme deca.

- **Validation automatique Lexer**

À la racine du compilateur, il y a un script permettant de tester la première étape du compilateur et en particulier le lexer.
Pour le lancer, rien de plus simple :

./lexer_test.py src/test/deca/context/…/test_complet_ultime.deca

ou

python3 lexer_test.py src/test/deca/context/…/test_complet_ultime.deca

Ce script prend en entrée un fichier .deca, avec au début de ce fichier, en commentaire, les tokens attendus.

Le script exécute le fichier *.deca* avec notre compilateur puis compare la sortie attendue (en commentaire) à celle obtenue.
Cela permet d'automatiser au maximum les tests, et d'imaginer dans l'avenir pouvoir tester automatiquement un important jeu de tests en quelques secondes et en un coup d'œil.
Ce script permet essentiellement de tester la partie d'analyse syntaxique, du Lexer. Cette technique de test pourra être étendue à toutes les étapes à l'avenir.

Voici sous quel format doivent être écrit les fichiers *.deca* à tester :

```
// Tokens attendus: OBRACE, IDENT, EQUALS, FLOAT, SEMI, TIMES, OPARENT, INT,
MINUS, CPARENT, CBRACE
// Fichier source avec des nombres à virgule flottante
{
   float price = 19.99;
   float discount = 0.2;
   float final_price = price * (1 - discount);
}
```

- **Validation automatique Génération de code**

Un autre script permet de valider automatiquement les tests pour la partie de génération de code en vérifiant que la sortie attendue est similaire à celle obtenue après avoir **compilé** le fichier deca puis appelé **ima** sur le fichier assembleur généré.
Ce script en bash exécute la commande decac avec le fichier .deca passé en paramètre puis récupère dans une variable 'résultat' la sortie obtenue avec ima. Ensuite il appelle un script python permettant d'extraire le résultat attendu d'un fichier de test et compare ainsi les deux résultats

Pour lancer ce script il suffit de taper comme commande à la racine du projet ***mvn test*** ou :

./src/test/script/basic-gencode.sh

Voici sous quel format doivent être écrit les fichiers *.deca* à tester :

```
// Description : affichage d'un entier et d'un flottant
//
// Résultats:
// 15
// 7.50000e+00
//
// Historique :
//    crée le 01/01/2024

{
   int entier = 15;
   float flottant = 7.5;
   println(entier);
```

```
   print(flottant);
}
```

CONCLUSION

Cette documentation vise à offrir une compréhension détaillée et accessible du compilateur Deca, en mettant particulièrement l'accent sur ses aspects pratiques et son application dans des projets de programmation. Les informations présentées sont conçues pour armer les utilisateurs des connaissances nécessaires à une utilisation efficace de ce compilateur, soulignant sa flexibilité et ses capacités avancées. Nous espérons que ce document deviendra une référence précieuse, facilitant une exploration enrichissante du langage Deca tant sur le plan académique que dans divers contextes de développement.