# Compilateur Deca : Manuel Utilisateur

Équipe 58

January 25, 2017

## 1 Introduction au compilateur

Le compilateur Deca prend en argument un fichier source en Deca et le compile pour donner un fichier en code assembleur.
Ce compilateur gère toutes les étapes de la compilation, c'est-a-dire le parsing (analyse lexical et syntaxique) du fichier source, l'analyse contextuelle du code et la génération du code assembleur. Plusieurs options sont disponibles pour le compilateur

## 2 Exécution de decac

Pour lancer le compilateur, il faut utiliser la commande `decac`. Cette commande possède différentes options d'exécution:

- `decac -b`
  Affiche la bannière de l'équipe qui a programmé ce compilateur.
- `decac -p <ficher deca>`
  Parse le fichier passé en argument et affiche sa décompilation.
- `decac -v <fichier deca>`
  Vérifie contextuellement le fichier deca. Affiche une erreur si il y en a.
- `decac -n <fichier deca>`
  Exécute le fichier deca en ignorant les erreurs de débordement à l'exécution.
- `decac -r X <fichier deca>`
  Execute le fichier deca en limitant les registres disponibles à `R{0}...R{X}`. `X doit être compris entre 4 et 16.`
- `decac -d <fichier deca>`
  Augmente le niveau de debug lors de l'exécution. Répéter l'option jusqu'à 4 fois pour augmenter les traces de debug.
- `decac -P <fichier(s) deca>`
  Si il y a plusieurs fichiers sources, les compile en parallèle.

Plusieurs options peuvent être appelées simultanément, `-p` et `-v` sont cependant incompatibles. Le fichier `.ass` obtenu en sortie du compilateur éxecuté avec la commande `ima <fichier assembleur>`.

# 3 Erreurs levées par le compilateur

## 3.1 Erreurs lexicales

Voici les erreurs qui peuvent être levées par l'analyse lexicale:

- `token recognition error`
  Est levée si le compilateur ne reconnaît pas le lexème.

## 3.2 Erreurs syntaxiques

- `RecognitionException / FailedPredicateException`
  Echec de l'analyse syntaxique. Est levée si le parser a réduit les possibilités à une seule, et que le code fourni ne convient pas. Le prochain symbole attendu sera fourni.

- `NoViableAltException`
  Echec de l'analyse syntaxique. Est levée s'il y avait plusieurs possibilités à l'endroit de l'échec, mais qu'aucune ne convenait au code. Les prochaines symboles possibles seront fournies.

- `InvalidLValue`
  Echec de l'analyse syntaxique. Est levée si ce dernier repère une tentative de définir une forme qui n'a pas à l'être. Exemple : (1+1) = 3; causera cette erreur, car (1+1) n'est pas une variable ou autre forme qu'il est sensé d'assigner à une valeur.

## 3.3 Erreurs contextuelles

Voici les erreurs qui peuvent être levées par l'analyse contextuelle :

- `type or class undefined`
  Est levée si un identifier de type est attendu, mais l'identifier trouvé n'est pas un type prédéfini ou une classe qui a été définie.

- `class extends type`
  Est levée si, lors de la déclaration d'une classe, l'identifier suivant `extends` est un type (`int,float,boolean,void`);

- `class already defined`
  Est levée si une même classe est déclarée plusieurs fois.

- `field already defined`
  Est levée si un champ est déclaré plusieurs fois ou déclaré avec le même nom qu'une méthode existante.

- `field cannot be void`
  Est levée si un champ a pour type `void`.

- `method already defined`
  Est levée si une méthode est déclarée plusieurs fois dans la même classe ou déclarée avec le même nom qu'un champ existant.

- `method overrides method with different signature`
  Est levée si une méthode essaie d'écraser une méthode avec une différente signature (différent type, différent nombre de paramètres ou différent types de paramètres).

- `parameter cannot be void`
  Est levée si un paramètre d'une méthode a pour type `void`.

- `parameter already defined`
  Est levée si plusieurs paramètres d'une même méthode ont le même nom.

- `variable already defined`
  Est levée si une variable est déclarée plusieurs fois.

- `variable cannot be void`
  Est levée si une variable a pour type `void`.

- `return called in void method`
  Est levée si `return` est appelée dans une méthode qui a pour type `void`.

- `type of expression must match method type`
  Est levée si l'expression renvoyée n'est pas du même type que la méthode.

- `expected type found class`
  Est levée si l'expression est une instance de classe alors qu'une expression de type `int,float` ou `boolean` est attendue.

- `incompatible class type`
  Est levée si la classe attendue n'est pas une super-classe de la classe renvoyé. Ce cas est le seul qui permet d'initier l'instance d'une classe avec une différente classe.

- `expected different type`
  Est levée si l'expression trouvée n'est pas du type attendu, sauf si l'expression est de type `int` et le type attendu est `float`. Dans ce dernier cas, on converti l'`int` en `float`.

- `only string, int and float expressions can be printed`
  Est levée si un `void`,`null` ou une instance de classe est mis en paramètre de `print` ou `println`.

- `condition must be boolean`
  Est levée si la condition dans un `if` ou `while` n'est pas booléen.

- `operands must be int or float`
  Est levée si les opérandes d'un opération arithmétique (ou d'une comparaison) ne sont pas numériques.

- `operand must be int or float`
  Est levée si l'opérande du moins unaire n'est pas numérique.

- `operands must be boolean`
  Est levée si les opérandes d'une opération booléenne n'est pas `boolean`.

- `operand must be boolean`
  Est levée si l'opérande de `Not(!)` n'est pas booléen.

- `operands must have same type`
  Est levée si les opérandes d'une comparaison n'ont pas le même type. Notamment, on ne peut pas comparer un `int` et un `float`.

- `incompatible types for cast`
  Est levée si expression est `Cast` dans un type incompatible a son propre type. Le seul `Cast` de type accepté est le passage de `int` à `float`.

- `type cast to class`
  Est levée si une expression de type `int,float` ou `boolean` est `Cast` dans une `class`.

- `class cast to type`
  Est levée si l'instance d'une classe est `Cast` en `int,float` ou `boolean`.

- `classes incompatible for cast`
  Est levée si l'instance d'une classe est `Cast` dans une classe incompatible. Cela se produit quand les deux classes ne sont pas dans la même hiérarchie de classes.

- `identifier is not a class`
  Est levée si l'identifier suivant `new` n'est pas un nom de classe.

- `cannot call this in main`
  Est levée si `this` est appelé en dehors d'une déclaration de classe.

- `left operand not instance of a class`
  Est levée si l'identifier a gauche d'un appel de champ n'est pas l'instance d'une classe.

- `no such field in class`
  Est levée si l'identifier à droite d'un appel de champ ne correspond pas à un champ de la classe à gauche.

- `identifier is not a field`
  Est levée si l'identifier à droite d'un appel de champ correspond à une méthode.

- `field is protected`
  Est levée si on essaie d'appeler un champ `protected` en dehors de la classe à laquelle il appartient.

- `expression not instance of a class`
  Est levée si l'identifier à gauche d'un appel de méthode n'est pas l'instance d'une classe.

- `direct method call in main`
  Est levée si une méthode est appelée sans préciser la classe ou l'instance de classe, en dehors d'une déclaration de classe.

- `identifier is not a method`
  Est levée si l'identifier a droite d'un appel de méthode correspond a un champ.

- `number of parameters does not match signature`
  Est levée si une méthode est appelée avec un nombre de paramètres différent du nombre de paramètre dans sa signature.

- `parameter type does not match signature`
  Est levée si le type d'un paramètre dans un appel de méthode ne correspond pas au type de paramètre dans la signature de la méthode.

- `class type in call not subclass of class type in signature`
  Est levée si, lorsqu'une méthode demande une instance de classe en paramètre, la classe appelée n'est pas une sous-classe de la classe donnée en signature.

## 3.4 Erreurs à l'éxecution

- `Erreur : pile pleine"`
  Est levée si il est impossible pour le programme de reserver la pile dont il a besoin avec l'operation assembleu `TSTO`

- `Error: Input/Output error`
  Est levée si :

  - la valeur reçue pour un entier sur l'entrée utilisateur standard est supérieur à $2^{31} - 1$ ou inférieur à -$2^{31}$

  - la valeur reçue pour un flottant sur l'entrée utilisateur standard est supérieur à `1.175494351 E { 38` ou inférieur à `3.402823466 E + 38` en valeur absolue

  - la valeur reçue n'est pas conforme au type attendu lors de la lecture

  - affichage d'une variable du mauvais type. N'est pas censé être soulevé.

- `Error: Arithmetic Overflow`
  L'erreur est soulevé lors de l'overflow de variables flottantes. Il est possible que cela provienne de:

  - Multiplication de deux flottant dépassant la taille limite.

  - Division d'un flottant par un flottant très petit provocant le dépassement de taille du résulat.

- `Error: Heap OverFlow`
  Est levée lorsque il y a dépassement de tas. C'est à dire que le programme a essayé de d'allouer plus de mémoire qu'il ne lui est possible. Normalement avec la machine ima cette valeur pour une allocation de 10002 plage 4 octets

# 4 Choix d'implémentation et erreur à la compilation

## 4.1 Choix d'implémentation

L'implémentation est assez proche du squelette fourni. La différene majoritaire est que la méthode `codeGen` retourne une variable `DVal` qui nous permet d'optimiser l'utilisation des registres. Nous avons aussi pris le partie de généraliser l'appelle à certaine classe pour la génération du code. Ainsi le package `codeGen` offre des classes permetants une génération de code général. Tel que `codeGenBinaryInstructionDValtoReg`.

## 4.2 Comportement inattendu

Lors d'appel de plusieurs intialisations ( c'est à dire qu'une initialisation apelle l'initialisation d'une autres classe.) nous avons remarqué que le code généré sauvegarde plus de registre que ceux utilisé lors de l'apelle de la méthode.
Il existe aussi un problème si l'apelle d'une méthode besoin d'accéder à la pile. En effet la gestion de l'overflow n'est pas adapté à cet environement. Pour le corrigé il nous aurait suffit de modifier rapidement le code de `DecacCompiler` pour ajouter un compteur de sauvgarde.

## 4.3 Oublies

Nous avons oublié de codé les déclaration de variables dans les méthodes

# 5 Extensions

Cette partie a pour but de présenter succintement le fonctionnement des extensions implémentées dans le compilateur Deca. Elles sont au nombre de deux : DeadStore et ConstantFolding

### 5.0.1 Deadstore

Pour activer le deadstore sur un programme, il suffit de rajouter l'option -o1 pendant la compilation du fichier Decac, en tapant en ligne de commande decac -o1 fichier.deca. Le fichier assembleur obtenu sera ainsi optimisé avec les spécificités du deadstore. Le Deadstore permet de supprimer les variables initialisées mais non utilisées par la suite dans votre programme. Par exemple, si vous écrivez :

```
int a=1;
int b=2;
int c;
c=5*b+1;
```

Le compilateur aura détruit de façon interne la variable "a" du programme, car elle n'est pas utilisée ni dans la liste des instructions ni après son initialisation.

### 5.0.2 ConstantFolding

Pour activer le ConstantFolding sur un programme, il suffit de rajouter l'option -o2 pendant la compilation du fichier decac, en faisant par exemple decac -o2 fichier.deca. Le fichier assembleur obtenu sera ainsi optimisé avec les spécificités de ConstantFolding. Le ConstantFolding permet de remplacer le calcul de constantes par le résultat directement. Par exemple si vous écrivez :

```
int a=5;
int b=10;
int c=15;
int d=5+5+5+5+5;
```

Le compilateur effectuera en interne le calcul de 5+5+5+5+5, et le remplacera par son résultat, ici 25. A la fin, le fichier assembleur sera adapté pour ce programme deca :

```
int a=5;
int b=10;
int c=15;
int d=25;
```

Le calcul est rendu plus rapide pour le compilateur, et pourra accélérer grandement les longs calculs de constantes (attention, le calcul avec des variables n'est pas pris en compte)

### 5.0.3 limitations

Pour le Deadstore, l'optimisation ne prend pas en charge le cas où il y a des calculs lors des initialisations. pour que le deadstore entre en action, la variable ne doit pas être utilisée au-delà des initialisations, au niveau des instructions
Pour le ConstantFolding, le code n'est pas optimisé, seul le calcul avec des constantes l'est.