Manuel Utilisateur
Pour Deca

GL22(G5)

January 22, 2024

Contents

1 Introduction

1.1 Projet Deca: Langage Java Sans Caféine

Le projet Deca émerge comme une exploration unique du monde du langage de programmation, tirant son nom de l'idée décaféinée dans l'univers Java. Deca se présente comme un langage décaféiné, un sous-ensemble, ou sous-langage de Java conçu pour simplifié la complexité du langage tout en restant fidèle à la robustesse du langage parent.

Dans ce contexte, notre équipe s'est lancée dans l'élaboration d'un compilateur pour Deca, cherchant à créer une expérience de programmation épurée et professionnelle.

Bienvenue dans l'univers serein de Deca, où la simplicité rencontre la sophistication, et où chaque ligne de code est délicieusement décaféinée.

1.2 Une Compilation Complète et Optimisée

Dans le cadre du projet Deca, notre objectif est clair : développer un compilateur capable de gérer tous les aspects du langage Deca. Cela comprend la prise en charge du côté sans objet, du côté avec objet, et la gestion de concepts cruciaux tels que le lexing[1], le parsing[2], la construction de l'arbre abstrait, jusqu'à la génération du code assembleur.

Notre mission ne se limite pas à la fonctionnalité brute. Nous nous engageons à optimiser chaque étape du processus de compilation.

Le résultat ? Un code assembleur généré efficacement, propre et bien structuré. Notre approche vise à simplifier le développement en offrant un compilateur qui prend en charge les particularités de Deca, tout en générant un code source facile à comprendre et à maintenir.

2 Fonctionnement et Limitations

2.1 Decac: Compilateur pour Deca

Le programme **decac** est un compilateur pour le langage Deca. Pour tirer pleinenement parti de **decac**, familiarisons-nous avex ses principales options:

- `-b (banner)` : Affiche une bannière avec le nom de l'$_{\text{(la meilleur)}}$ équipe. **GL 22**

[1] https://fr.wikipedia.org/wiki/Analyse_lexicale
[2] https://fr.wikipedia.org/wiki/Analyse_syntaxique

- `-p (parse)` : Construit l'arbre de l'analyse syntaxique et affiche la décompilation correspondante. Utilisez cette option si vous souhaitez vérifier la syntaxe d'un programme Deca sans passer à l'étape de compilation.

  ```
  decac -p mon_programme.deca
  ```

- `-v (verification)` : Effectue des vérifications sans produire de sortie en l'absence d'erreurs. Utile pour garantir la qualité de votre code avant la compilation.

  ```
  decac -v mon_programme.deca
  ```

- `-n (no check)` : Supprime les tests à l'exécution. À utiliser si vous préférez éliminer les vérifications supplémentaires à l'exécution.

  ```
  decac -n mon_programme.deca
  ```

- `-r X (registers)` : Limite les registres banalisés disponibles à R0 ... RX-1, avec 4 <= X <= 16. Contrôle le nombre de registres utilisés lors de la compilation.

  ```
  decac -r 8 mon_programme.deca
  ```

- `-d (debug)` : Active les traces de débogage. Répétez cette option plusieurs fois pour obtenir plus de détails de débogage.

  ```
  decac -d -d mon_programme.deca
  ```

- `-P (parallel)` : Si plusieurs fichiers sources sont présents, lance la compilation en parallèle pour accélérer le processus.

  ```
  decac -P mon_programme1.deca mon_programme2.deca
  ```

- `-op (optimize)` : Optimise le code généré. Cette option peut allonger le temps de génération du code.

  Exemple :

  ```
  decac -op mon_programme.deca
  ```

N'oubliez pas que les options `-p` et `-v` sont incompatibles. Utilisez ces options judicieusement pour optimiser votre expérience avec le compilateur Deca.

Vous retrouvez ces spécifications aussi en exécutant decac seule. Cela, si vous comprenez bien la langue de Shakespeare.

2.2 Limitations du Compilateur Decac

Le compilateur Deca ne permet pas d'effectuer une conversion (cast) entre deux variables de type prédéfini ou de type classe.

2.3 Partie Sans-objet

Le compilateur autorise la création d'un `main` vide, ce qui offre une flexibilité particulière aux programmes ne nécessitant pas de logique initiale dans la fonction principale. Cette fonctionnalité répond à des scénarios spécifiques où une entrée minimale est requise.

Le compilateur est conçu pour fonctionner de manière robuste même en cas de pénurie de registres. Cette résilience garantit une exécution stable du programme, même dans des environnements où les ressources sont limitées.

Le projet met en œuvre une optimisation intelligente des registres, notamment pour les opérations arithmétiques. Cela se traduit par une utilisation minimale de l'espace mémoire, améliorant ainsi l'efficacité du code généré.

Une gestion efficace des déclarations de variables de types tels que `int`, `float`, `boolean`, etc., est assurée.

Le compilateur prend en charge plusieurs éléments essentiels du langage Deca. Cela inclut la possibilité d'inclure des instructions courantes telles que print, d'effectuer des opérations arithmétiques de base, de créer des boucles while, et de gérer des branchements conditionnels, d'ajouter des commentaires dans le programme et de récupérer des entiers (int) et des flottants (float) à partir de la ligne de commande en utilisant **`readfloat()`** et **`readint()`**. Ces fonctionnalités fournissent une base solide pour la programmation sans objet.

2.4 Partie Avec-objet

Dans la seconde phase du développement du compilateur Deca, axée sur le côté orienté objet, plusieurs fonctionnalités ont été introduites pour prendre en charge les concepts de classes et d'héritage dans le langage Deca. Cependant, il est important de noter certaines limitations et caractéristiques spécifiques.

Le compilateur est capable d'interpréter un programme Deca qui introduit des classes. Ces classes peuvent être appelées et initialisée.

Le concept d'héritage est pris en charge, permettant aux classes de hériter des caractéristiques, notamment les méthodes et les champs, d'autres classes existantes.

Le compilateur autorise l'introduction de champs pour les classes, permettant aux développeurs de définir des champs spécifiques à chaque classe. De plus, la gestion de la visibilité des champs, telle que **`public`** ou **`protected`**.

Le compilateur autorise également l'introduction de méthodes avec des paramètres. Chacun des paramètres a un type prédéfini (sauf le type void) ou le type d'une classe déjà définie dans le programme. Ces méthodes peuvent renvoyer des objets de type prédéfini ou de type classe.

Dans la section principale (main) du programme, l'utilisateur peut initialiser des variables de types de classe déjà définis dans le programme en utilisant **`new nomDeClasse()`** .

De plus, le compilateur peut interpréter l'opérateur **`instanceof`** appliqué entre une variable de type classe et un type de classe déjà défini dans le programme, comme illustré dans l'exemple suivant :

```
class A{}
{
    A a = new A();
    boolean b = a instanceof A;
}
```

3 Messages d'erreur

3.1 Erreurs Lexicales

Les erreurs lexicales se produisent lorsque le compilateur rencontre des tokens qui ne sont pas reconnus ou invalides. Ces erreurs sont détectées lors de l'analyse lexicale. Les erreurs lexicales incluent :

- **Existence de tokens invalides :** Utilisation de tokens non conformes à la lexicographie de Deca.
- **Existence de mots réservés invalides :** Erreur de saisie des mots réservés dans la lexicographie de Deca.

3.2 Erreurs Syntaxiques

La syntaxe de Deca est basée sur un ensemble de règles grammaticales qui doivent être respectées par l'utilisateur.
Les erreurs syntaxiques se produisent lorsque l'utilisateur viole l'une de ces règles en saisissant un code syntaxiquement incorrect. Une liste non exhaustive de ces erreurs syntaxiques est donnée ci-dessous:

- **Parenthèses ou accolades manquantes :**
  Exemple 1 :

```
void m(int x, boolean y
```

  Exemple 2 :

```
if (x == 1) {
    println("gagné");
```

- **Points virgules manquants :**
  Exemple :

```
x = x + 1
```

- **Utilisation incorrecte des mots réservés :**
  Exemple : Utilisation incorrecte de ifThenElse :

```
if (x == 1) {
    println("Gagné !");
}
else {
    println("Perdu !");
}
else {
    println("Erreur !");
}
```

3.3 Erreurs Contextuelles

Les erreurs contextuelles se produisent lorsque l'utilisateur écrit un code syntaxiquement correct, mais qui viole la sémantique de Deca. Ainsi, un programme qui comprend certaines de ces erreurs est rejeté par le compilateur avec un message d'erreur bien approprié.
Une liste de ces erreurs avec des examples est donnée ci-dessous:

- **Typage incorrect des expressions unaires:**

```
    float f = 5.5;
    int a = !f;
Erreur : ! unary operation cannot occur with float !
```

- **Déclaration d'une variable déjà définie:**

```
    float f = 5.5;
    float f = 3.6;
Erreur : Name f is already defined in localEnv !
```

- **Assigner deux variables de types incompatibles:**

```
    boolean f = 2;
Erreur : assign_compatible condition in rvalue no-terminal fails !:
Trying to assign int to boolean
```

- **La condition de if n'est pas booléenne:**

```
    int x = 2;
    if (x = 2){
       print("La valeur de x est:", x);
    }
Erreur : Condition in ifThenElse loop must be of type boolean:
int was given !
```

- **Utilisation d'une variable non préalablement déclarée:**

```
    int f = x + 1.3;
Erreur : Expression 'x' is not defined in the local environment
```

- **Opérations binaires entre deux types d'objets incompatibles:**

```
    boolean x = true;
    int f = x + 1.3;

Erreur : + operation cannot occur between boolean and float !
```

- **Affichage d'une expression d'un type autre que Int, Float et String:**

```
    boolean x = true;
    print(x);
Erreur : Printable expressions can only be
of type Int or Float or String: boolean was given !
```

- **Utilisation d'une classe non définie:**

```
class A{}
{
   B b = new A();
}
Erreur : Type: B is undefined !
```

- **Conversion invalide d'un type vers un autre type:**

```
    int x = 147;
    int k = (void)(x);
Erreur : Can not cast int to void !
```

- **Déclaration d'un attribut et d'une méthode de même nom dans une classe:**

```
class A{
   int m = 3;
   int m(int k){
       return 2*k;
   }
}
Erreur : Un champ et une méthode ont le même nom dans la classe A !
```

- **Déclaration d'une classe déjà définie:**

```
class A{}
class A{}
Erreur : Class A is already defined !
```

- **Déclaration d'un attribut de type Void:**

```
class A{
   void k;
}
Erreur : Type of field must not be of Void type !
```

- **Déclaration d'un paramètre de type Void:**

```
class A{
   void k(void b){}
}
Erreur : Type of param b must not be of Void type !
```

- **Déclaration d'une méthode dans une classe où le nom de la méthode existe déjà en tant que champ dans la superclasse :**

```
class A{
   int k = 6;
}
class B extends A{
   int k(){
       return 147;
   }
}
Erreur : k can't be a method because
it is already defined in super class and is not a method !
```

- **Utilisation d'une méthode avec un nombre incorrect d'arguments:**

```
class B {
   void k(int s){
        s = 4;
   }
}
{
   B b = new B();
   b.k();
}
Erreur : Number of method arguments is not respected,
this method accepts 1 argument(s), 0 arguments were given !
```

- **Accès à un attribut de visibilité protected:**

```
class A{
   protected int k = 6;
}
{
   A b = new A();
   int x = b.k;
}
Erreur : Cannot get access to field k from current class
```

- **Appel à this en dehors d'une classe:**

```
{
    A b = new A();
    int x = this.b;
}
Erreur : This cannot be called outside a class !
```

- **Type de retour d'une méthode est Void:**

```
class B {
   void k(){
        int x = 3;
        return x;
   }
}

Erreur : Return Type should not be void !
```

- **Effectuer la sélection sur un objet de type différent de Class:**

```
{
   int x = 3;
```

```
    println(x.k);
}
Erreur : The object to which you apply
the selection must be of type Class: int was given !
```

3.4 Erreurs d'exécution

Les erreurs d'exécution dans deca sont des erreurs qui se produisent lors de la phase du génération du code assembleur. Une liste de ces erreurs est donnée ci-dessous:

- **Stack Overflow erreur:**

  ```
  Error: Stack Overflow
  ```

- **Erreur de Division par zéro:**

  ```
  Error: Division by Zero
  ```

- **Erreurs de input dans readfloat() et readint():**

  ```
  //Pour le readint()
  Error: Invalid Integer Input
  //Pour le readfloat()
  Error: Invalid Float Input
  ```

- **Erreur de remplissage complet du stack:**

  ```
  Error : The Stack is Full
  ```

- **Erreur d'utilisation de l'opération Modulo avec 0:**

  ```
  Error: Modulo by Zero
  ```

- **Erreur de dépassement pour la taille des nombres flottants.**

  ```
  Error: Overflow
  ```

4 Extension: OPTIM

4.1 Amélioration des Performances

L'extension optim du compilateur Deca apporte des améliorations significatives aux performances du code généré. Elle intègre plusieurs algorithmes

d'optimisation visant à rendre les programmes plus efficaces et à réduire leur empreinte.

a été conçue avec une philosophie axée sur l'amélioration des performances sans altération du comportement du code source d'entrée.

4.2 Algorithmes d'Optimisation

L'extension optim intègre plusieurs algorithmes d'optimisation pour améliorer les performances du code généré. Voici une énumération des algorithmes inclus :

5 Extension Optim - Algorithmes d'Optimisation

L'extension optim intègre plusieurs algorithmes d'optimisation pour améliorer les performances du code généré. Voici une énumération des algorithmes inclus :

1. **Dead Code Elimination (DCE):**
   Applique l'algorithme DCE pour éliminer les portions de code qui ne contribuent pas au résultat final du programme, réduisant ainsi la taille du code généré.

2. **Loop Inversion:**
   Applique l'algorithme d'inversion de boucle pour maximiser l'efficacité dans les structures de boucles while.

3. **Liveness Analysis (Réduction aux Boucles While):**
   Optimise l'accès aux variables dans les boucles while (hot spots), réduisant ainsi les temps d'accès aux données cruciales.

4. **Strength Reduction:**
   Applique la réduction de la force pour remplacer les opérations coûteuses par des opérations moins coûteuses, préservant la sémantique du programme. L'déee est de remplacer les multiplications et divisions avec des puissances de 2 par des shift.

5. **Constant Folding and Propagation:**
   Effectue le repliement et la propagation constants pour évaluer les expressions au moment de la compilation, réduisant la nécessité d'évaluations répétées.