ANALYSE DES IMPACTS ÉNERGÉTIQUES

EQUIPE 20

GROUPE 4

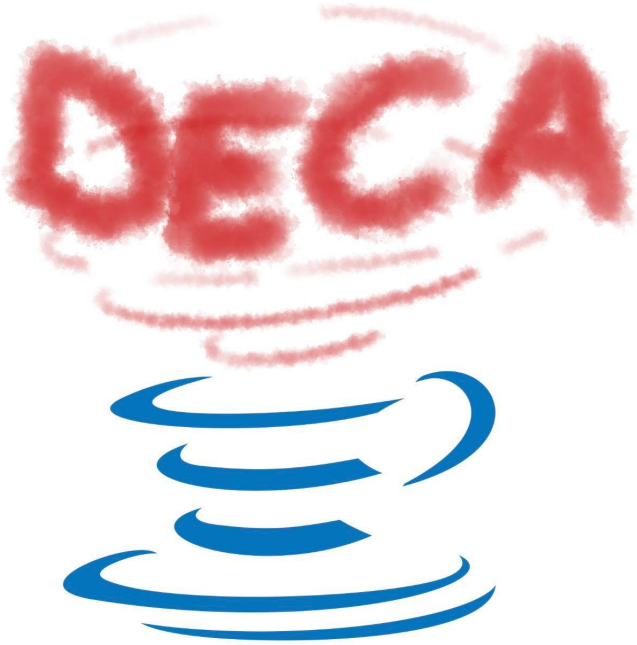

Table des matières

SECTION 1

Introduction

En tant qu'ingénieur du numérique responsable, nous nous devons d'être conscient de l'impact social et environnemental de notre travail sur la société. Nous rédigeons donc ce rapports dans le but de prendre conscience des impacts environnementaux du projet et pourquoi pas de trouver des façons d'amoindrir cet impact dans nos futurs projets.
Le projet a pour but de développer un compilateur du langage Deca vers une machine abstraite Ima, même si notre projet n'aura que peu d'impact à notre échelle, il est important de prendre conscience qu'un tel outil pourrait être utilisé de très nombreuses fois pour la compilation et le développement de logiciels de grande envergure. Ainsi, le code générer par compilateur se doit d'être le plus optimisé possible pour d'un côté être efficace mais aussi pour limiter un maximum l'impact sur l'environnement.
Prenons un exemple simple, il est possible de voir lors de la génération du code assembleur de type de code :

```
LOAD #1, R2
LOAD R2, R3
```

Ce code est générer automatisque en 2 étapes, donc il sera toujours fonctionnel, mais il semble évident que ce code ne devrait pas s'écrire ainsi mais plutôt :

```
LOAD #1, R3
```

Ce type d'instruction assembleur étant très souvent appelé, diviser par 2 leur nombre n'est pas négligeable car cela pour diviser par deux la consommation énergétique du code généré. D'un autre côté, en plus de devoir générer du code optimal, la façon dont celui-ci est généré peut aussi être optimisée pour permettre une compilation moins gourmande en énergie.

SECTION 2

Efficience du procédé de fabrication

2.1 Compilation de la partie contextuelle

La partie sans objet du code ne présente que peu de perte d'efficacité pour le procédé de fabrication. Il ne s'agit que de simples tests, et les listes de déclaration de variables sont parcourues en temps linéaire. Il est toutefois à noter que dans le cas où chaque type ne correspond qu'à au plus une seule variable, le parcours de la classe `DeclVarSet`, puis celui de `ListDeclVar` et `DeclVar` n'est pas optimal. Il est donc préférable de tester un programme avec :

```
int i = 1, j = 2;
```

plutôt que :

```
int i = 1;
int j = 2;
```

La partie avec objet présente plus de perte d'efficacité par rapport aux autres classes. En effet, pour chaque classe et pour chaque méthode, il faut allouer une place mémoire pour une variable `EnvironmentExp` propre à ces objets. Il faut de plus, pour chaque objet manipulé ou créé dans une classe, vérifier sa validité, principalement en vérifiant son existence et sa compatibilité avec l'ensemble des champs et des méthodes définies dans une éventuelle classe supérieure. Comme une classe hérite des attributs de la classe mère, il fallait soit copier tous les attributs non redéfinis de la classe mère dans la classe fille, ce qui prenait de la mémoire, soit parcourir récursivement l'arborescence des classes lors d'un appel pour y rechercher un champ ou une méthode, ce que engendrait une perte d'efficience du code. Nous avons donc choisi en priorité l'efficacité du compilateur, cette solution est alors plus coûteuse en mémoire. Cependant, pour

éviter également de perdre trop de temps dans la vérification de la classe supérieure, nous avons factorisé du code dans `DeclClass.java` pour ne pas avoir à vérifier de nouveaux certaines conditions que l'on savait vraies après l'appel d'une première fonction `vérify`, lors des premières passes. La partie génération de code a été suffisamment optimisée pour ne pas présenter de perte de temps et de mémoire trop importantes.

SECTION 3

Efficience du code produit

3.1 Tests et scripts de validation

Pour valider le code écrit tout au long du projet, nous avons créé une batterie de tests permettant de traiter différents aspects devant être mis en lumière. Ces tests se répartissent entre les tests pour les parties lexer, syntaxe, contexte et génération de code se trouvant dans `src/test/deca/`, puis les tests sensés être valides et les tests sensés être invalides. Voici un tableau répertoriant le nombre de tests de chaque partie :

Tests :	syntaxe	context	codegen
Valides :	9	57	71
Invalides :	2	57	11

Cependant, afin d'éviter des dépenses énergétiques liées au projet trop élevées, nous avons décidé de ne lancer les tests qu'à chaque grosse mise à jour sur GitLab. Cela avait pour but de se forcer à ne pas lancer l'exécution de l'ensemble du programme de vérification plusieurs fois par jour. Pour lancer l'ensemble des tests, nous avons en tout début de projet utilisés des scripts artisanaux permettant de parcourir tous les fichiers tests d'un répertoire et de lancer des scripts fournis dessus, comme `test_lex`, `test_synt` ou `test_context`.

Enfin, pour ce qui est du code produit, les résultats par rapport aux autres groupes se sont avérés assez satisfaisant, et assez proches, si ce n'est parfois meilleurs que le compilateur de référence : `VieuxCompdeProf`. Voici les scores et classements nous permettant d'évaluer l'efficience de notre code :

Tests :	Syracuse42	ln2	ln2_fct
Résultats :	1288	15004	19001
Résultats profs :	1340	15194	18102
Classement :	4	6	9