

# Projet Génie Logiciel : introduction

Projet GL

Ensimag  
Grenoble INP

6 décembre 2024



# Sommaire

- 1 Buts du projet
- 2 Vue d'ensemble du langage et du compilateur
- 3 Développement durable
- 4 Extension
- 5 Déroulement et planification
- 6 Ethique et fraude
- 7 Environnement de développement

# Objectifs : cf. [Introduction]

## ● Génie Logiciel

- ▶ Écrire un logiciel fiable dans le temps imparti ;
- ▶ Comprendre et respecter un cahier des charges, des spécifications ;
- ▶ Travailler en équipe et s'organiser ;
- ▶ Expérimenter des techniques agiles de développement : développement dirigé par les tests, intégration continue, programmation par paires ;
- ▶ Utiliser des outils d'aide au développement : Maven, Git, Jacoco

## ● Compilation

- ▶ Application du cours de théorie des langages : écrire un compilateur pour le langage Deca (un « mini-Java », sous-ensemble de Java avec quelques variations) ;
- ▶ Utiliser des générateurs d'analyseurs lexicaux et syntaxiques (ANTLR) ;
- ▶ Comprendre la façon dont les calculs sont traduits par les machines, par exemple sur les flottants.

# Sommaire

- 1 Buts du projet
- 2 Vue d'ensemble du langage et du compilateur
- 3 Développement durable
- 4 Extension
- 5 Déroulement et planification
- 6 Ethique et fraude
- 7 Environnement de développement

# Sommaire de cette section

## 2 Vue d'ensemble du langage et du compilateur

- Le langage Deca
- Le compilateur

# Le langage Deca

Deca est un sous-ensemble de Java, avec quelques variations.

On peut déclarer :

- des classes,
  - ▶ des champs,
  - ▶ des méthodes,
- un programme principal.

# Le langage Deca

## « Sous-langages » de Deca

langage Hello-world affichage des chaînes de caractères ;

langage sans-objet on ne peut pas déclarer de classes, donc pas d'attributs et pas de méthodes.

On ne peut donc avoir qu'un programme principal ;

langage essentiel sauf les conversions (cast) et les tests d'appartenance à une classe (instanceof) ;

langage complet tout !

```
hello_world.deca
```

```
{ println("Hello world !"); }
```

## Fichiers inclus

- On peut inclure des fichiers grâce à la construction `#include "fich.decah"`.
- Exemple (pas très propre!)

```
hello.decah
```

```
{ println("Hello",
```

```
world.decah
```

```
"world !"); }
```

```
hello_world.deca
```

```
#include "hello.decah"
```

```
#include "world.decah"
```



# Bibliothèque standard

## cf. II-[BibliothèqueStandard]

- Recherche d'un fichier inclus :
  - ▶ dans le répertoire courant,
  - ▶ puis dans la bibliothèque standard (par ex. pour la classe Math),  
emplacement `src/resources/include/` de la hiérarchie imposée.
- Code Deca sous forme de fichiers à inclure `.decah`
- Utile pour développer certaines extensions

# Sommaire de cette section

- 2 Vue d'ensemble du langage et du compilateur
  - Le langage Deca
  - Le compilateur

# Le compilateur

- Langage de programmation du compilateur : Java
- Langage source : Deca
- Langage cible : langage d'assemblage pour une « machine abstraite »

# Le compilateur - cf. l-[ExempleSansObjet]

Le compilateur comporte trois étapes :

- étape A :
  - ▶ analyse lexicale
  - ▶ analyse syntaxique
  - ▶ construction de l'arbre abstrait
- étape B :
  - ▶ vérifications contextuelles,
  - ▶ décoration de l'arbre abstrait
- étape C :
  - ▶ génération de code.

Chaque étape comporte entre 1 et 3 passes sur le programme.

## Etape A

- L'étape A comporte :
  - ▶ analyse lexicale,
  - ▶ analyse syntaxique,
  - ▶ construction de l'arbre abstrait.
- Le programme source Deca est une suite de caractères.
- L'analyse lexicale consiste à reconnaître les « mots ».
- L'analyse syntaxique consiste à vérifier que la suite de mots est une « phrase » correcte du langage.
- En même temps qu'on effectue l'analyse syntaxique, on construit l'« arbre abstrait » du programme source Deca (représentation structurée du programme sous la forme d'un arbre).
- L'étape A s'effectue en une seule passe sur le programme source.

## Etape B

- Le but de l'étape B est de :
  - ▶ réaliser des vérifications contextuelles ;
  - ▶ modifier et décorer l'arbre abstrait du programme pour préparer l'étape C.
- La syntaxe contextuelle de Deca (cf. [SyntaxeContextuelle]) définit les programmes Deca contextuellement corrects

### Outil *Grammaire attribuée de Deca*

- L'étape B est réalisée en 3 passes.
  - ⇒ 3 parcours de l'arbre abstrait.

## Etape B

- Exemple de vérification contextuelle :
  - ▶ vérifier que les identificateurs sont correctement déclarés, et utilisés conformément à leur déclaration ;
  - ▶ vérifier que les expressions sont correctement typées.
- Notion d'*environnement*
  - ▶ À chaque identificateur est associée sa « définition ».
- Trois parcours de l'arbre abstrait
  - ▶ première passe : vérifier le nom des classes,
  - ▶ deuxième passe : vérifier les champs et signatures des méthodes,
  - ▶ troisième passe : vérifier le corps des méthodes.
- Pendant un parcours, on décore
  - ▶ les identificateurs avec leur « définition »,
  - ▶ les expressions avec leur « type ».

## Etape C

- L'étape C consiste à
  - ▶ générer le code exécutable
- On génère du code assembleur pour une « machine abstraite », proche du 68000.
- On peut exécuter ce code grâce à un « interprète de machine abstraite » (IMA).
- L'étape C s'effectue en deux passes :
  - = deux parcours de l'arbre abstrait
- Les deux parcours :
  - ▶ première passe : construire la table des méthodes des classes ;
  - ▶ deuxième passe : coder le programme.



# Sommaire

- 1 Buts du projet
- 2 Vue d'ensemble du langage et du compilateur
- 3 Développement durable
- 4 Extension
- 5 Déroulement et planification
- 6 Ethique et fraude
- 7 Environnement de développement

# Développement durable

## Analyse énergétique de votre projet

- Efficacité du code produit
  - ▶ Un compilateur est potentiellement utilisé pour produire de nombreux logiciels
  - ▶ Produire du code optimisé
- Efficacité du procédé de fabrication de votre compilateur
  - ▶ Coût des compilations
  - ▶ Coût de la validation (exécution des tests)

## Pistes possibles d'analyse

- Informations générales sur la consommation des ordinateurs
- Consommation de votre propre projet (par ex. /usr/bin/times)
- Pour l'exécution des programmes générés : considérer le nombre de cycles indiqué par la machine abstraite ima

D'autres analyses sont les bienvenues !

**A vous de montrer que votre génération est éco-consciente.**

# Développement durable : évaluation

- Efficacité énergétique du code produit
  - ▶ Des exemples de test de performance sont fournis
  - ▶ Un palmarès pour se mesurer aux autres équipes au cours du projet
  - ▶ Evaluation par les enseignants sur d'autres tests
  - ▶ Critère important pour extension OPTIM et quelques autres
- Efficience globale de votre développement
  - ▶ Démarche globale pour réduire les phases "gourmandes" en énergie
  - ▶ Conception de scripts de test efficaces
- Qualité de l'analyse réalisée (document en fin de projet)
  - ▶ Le point le plus important pour valider vos compétences en DD.

# Sommaire

- 1 Buts du projet
- 2 Vue d'ensemble du langage et du compilateur
- 3 Développement durable
- 4 Extension**
- 5 Déroulement et planification
- 6 Ethique et fraude
- 7 Environnement de développement

# Extension

- Développer le compilateur pour le langage Deca *essentiel*
  - ▶ Partie précisément spécifiée et guidée
  - ▶ Les spécifications doivent être strictement respectées
  - ▶  $\Rightarrow$  75% du projet
- Extension (au choix)
  - ▶ Partie peu spécifiée, et très peu guidée
  - ▶ Recherches bibliographiques à effectuer
  - ▶ Spécifications à négocier avec les enseignants
  - ▶ Analyse, conception et implémentation très peu guidées
  - ▶ Méthode de validation à déterminer et à présenter
  - ▶ Validation à effectuer
  - ▶ Documentation à rendre
  - ▶  $\Rightarrow$  25% du projet

# Extension

- Choisir l'extension dès maintenant (semaine 1)
- Premier suivi : choix de l'extension finalisé avec les enseignants
- Commencer à travailler (étude bibliographique, spécifications)
- Deuxième suivi : présentation et négociation des spécifications avec les enseignants

N.B. Vos enseignants n'ont pas forcément beaucoup plus d'idées que vous sur comment réaliser ces extensions.

Ce sera à vous d'être créatifs et force de proposition.

## Extensions proposées

- [TRIGO] Bibliothèque de fonctions trigonométriques et calcul flottant
- [HISTOIRE/ACONIT] Génération de code pour machine historique, en partenariat avec l'association ACONIT
- [ARM] Génération de code pour l'architecture ARM
- [BYTE] Génération de bytecode Java
- [OPTIM] Mise en oeuvre de techniques classiques en compilation pour optimiser le code engendré
- [TAB] Extension de Deca avec des tableaux et bibliothèque de calcul matriciel
- [LINK] Compilation séparée et édition de liens
- [ETUD] Extension proposée par une équipe d'étudiants

## Extension [TRIGO]

- Classe Math
  - ▶ fichier `Math.decah`
  - ▶ dans la bibliothèque standard
- Fonctions attendues (spécification à respecter)
  - ▶ `float sin(float f)`
  - ▶ `float cos(float f)`
  - ▶ `float asin(float f)`
  - ▶ `float atan(float f)`
  - ▶ `float ulp(float f)`
- Algorithmes de calcul de ces fonctions
- Possibilité d'utiliser l'assembleur
- Défis
  - ▶ Exigences de précision (au presque dernier bit près ou mieux)
  - ▶ Efficacité des algorithmes (en place mémoire et temps de calcul)



# Extension [ARM]

- Génération de code pour l'architecture ARM
- Type d'architecture le plus répandu actuellement
- Deux étapes C
  - ▶ machine abstraite
  - ▶ processeur ARM
- Défis
  - ▶ double back-end (génération de code)
  - ▶ pas d'environnement (E/S etc.), débogage

## Extension [BYTE]

- Génération de bytecode Java
- Deux étapes C
  - ▶ machine abstraite
  - ▶ bytecode Java
- Étudier le bytecode Java
- Utiliser une bibliothèque de manipulation de bytecode
- Tester efficacement le code généré
- Défis
  - ▶ principe de machine différent
  - ▶ environnement E/S, débogage...
  - ▶ pouvoir exécuter sur la JVM un programme constitué de classes compilées avec votre compilateur et des classes compilées avec javac

# Extension [OPTIM]

- Optimisation du code généré (en particulier pour l'énergie)
  - ▶ Étudier les techniques classiques d'optimisation
  - ▶ Implémenter les algorithmes
  - ▶ Évaluer les résultats
- Défis
  - ▶ Techniques complexes, analyses dataflow non fournies
  - ▶ Ambitions à négocier avec enseignants

# Extension [TAB]

- Étendre le langage Deca avec des tableaux
  - ▶ Syntaxe hors-contexte
  - ▶ Syntaxe abstraite (grammaire d'arbres)
  - ▶ Syntaxe contextuelle (grammaire attribuée)
  - ▶ Sémantique (comportement à l'exécution)
- Implémenter les étapes A, B et C pour les tableaux
- Proposer une bibliothèque de calcul matriciel
- Défis
  - ▶ Formaliser grammaire contextuelle et sémantique des tableaux
  - ▶ Bibliothèque à négocier avec les enseignants

## Extension [LINK]

- Permettre la compilation séparée en Deca : fichiers objet
- Édition de liens pour faire un exécutable (assembleur IMA)
- Lors de la compilation séparée, on n'a pas toutes les informations pour générer le code
  - ▶ nécessité de conserver des informations symboliques dans les fichiers objet
  - ▶ l'édition de liens permet de résoudre ces liens symboliques
- Défis
  - ▶ Définir un format de code objet (ad hoc)
  - ▶ Intégrer génération symbolique pour ima

## Extension [HISTOIRE]

- Générer du code pour une machine ancienne
- Travail avec une association externe (ACONIT : Association pour un CONservatoire de l'Informatique et de la Télématique)
- Exemple de cible : les premiers Mac (code 68000)
- Défis
  - ▶ Code pour une vraie machine, avec ses limites, et sans bibliothèque
  - ▶ Idéalement : pouvoir exécuter le code Deca sur une vieille machine “nue”
  - ▶ Possibilité de participer à démos externes (fête de la science, expositions etc)

# Sommaire

- 1 Buts du projet
- 2 Vue d'ensemble du langage et du compilateur
- 3 Développement durable
- 4 Extension
- 5 Déroulement et planification
- 6 Ethique et fraude
- 7 Environnement de développement

# Déroulement du projet

- Stage

- ▶ 6h40 de vidéos de présentation du projet à visionner
- ▶ 6h00 de cours en début de période + 1h30 séance machine
- ▶ Amphi « Git avancé » d'1h30 (optionnel)
- ▶ Séance « Sobriété du code » (optionnel)

- Suivis [Suivis]

- ▶ 3 séances : 30 minutes pour chaque équipe
- ▶ Pris en compte dans la note finale

- Rendu intermédiaire [RenduIntermediaire]

- ▶ Compilateur de programmes Deca sans-objet

date Lundi 13 janvier 2025 à 12h00

- ▶ Pris en compte dans la note finale

- Récupérations des projets : programmes et tests du compilateur et de l'extension

date Lundi 20 janvier 2025 à 16h00.

- Rétrospective collective

date Mardi 21 janvier 2025.

- Soutenance [Soutenance]

date du jeudi 23 janvier au vendredi 24 janvier 2025.



# Planification du projet

Le travail doit être organisé :

- Découper le projet en tâches à réaliser :
    - ▶ Hello-world, sans objet, essentiel
    - ▶ Extension
    - ▶ Etapes A,B,C
    - ▶ Analyse, conception, implémentation, validation⇒ Diagramme de tâches
  - Prendre en compte :
    - ▶ Liens d'antériorité
    - ▶ Parallélisme⇒ Diagramme de PERT
  - Planifier les tâches :
    - ⇒ Planning de Gantt
- Outil planner [SeanceMachine]
- A faire :
    - ▶ planning prévisionnel, pour le premier suivi
    - ▶ planning effectif, à chacun des suivis
    - ▶ charte d'équipe, au premier suivi SHEME

## Documentation à rendre [A-Rendre]

- Documentation utilisateur (environ 12 pages)
  - ▶ Description du compilateur du point de vue de l'utilisateur
  - ▶ Commandes et options
  - ▶ Messages d'erreurs
  - ▶ Limitations
  - ▶ Utilisation de l'extension

date Lundi 22 janvier 2024 à 20h00

- Bilan
  - ▶ Bilan collectif sur la gestion d'équipe et de projet

date Mercredi 24 janvier 2024 à 9h00

## Documentation à rendre [A-Rendre]

- Documentation de conception (environ 10 à 15 pages)
  - ▶ La conception architecturale des étapes B et C
  - ▶ Les algorithmes et structures de données spécifiques

date le jour de la soutenance

- Documentation de validation (environ 10 à 15 pages)

### [Tests]

- ▶ Descriptif des tests
- ▶ Scripts de tests
- ▶ Gestion des risques et gestion des rendus
- ▶ Couverture des tests (résultats de Jacoco)
- ▶ Méthodes de validation autres que le test

date le jour de la soutenance

# Documentation à rendre [A-Rendre]

- Documentation de l'extension (20 à 30 pages)

- ▶ Analyse bibliographique
- ▶ Analyse et conception
- ▶ Choix d'algorithmes
- ▶ Méthode de validation
- ▶ Validation de l'implémentation

date le jour de la soutenance

## Documentation à rendre [A-Rendre]

- Documentation sur les impacts énergétiques du projet et de ses retombées (4 à 10 pages)
    - ▶ Moyens mis en œuvre pour évaluer la consommation énergétique de votre projet
    - ▶ Discussion sur vos choix de génération de code
    - ▶ Discussion sur vos choix de processus de validation
    - ▶ Prise en compte de l'impact énergétique de votre extension
    - ▶ **Toute autre analyse pertinente est également bienvenue**
- date le jour de la soutenance

# Sommaire

- 1 Buts du projet
- 2 Vue d'ensemble du langage et du compilateur
- 3 Développement durable
- 4 Extension
- 5 Déroulement et planification
- 6 Ethique et fraude
- 7 Environnement de développement

## Ethique professionnelle, responsabilité d'ingénieurs

- 1 équipe = 1 micro-entreprise en concurrence avec les autres
- toutes vos productions (code, tests, doc...) doivent être ORIGINALES
- copie = vol de Propriété Intellectuelle = délit
- équipe : solidaire ; si un ingénieur d'une entreprise fournit des codes volés, c'est toute l'entreprise qui peut couler ; même les employés sans rapport se retrouveront au chômage

Particularités projet GL :

- Interdiction de copier des tests ou code ou doc *même en Open Source*
  - Documents : exploitation possible en citant TOUTES ses sources

Fraude dans un projet = 0 pour TOUTE l'équipe

- année non validée
- + conseil discipline

## Concrètement : cas de fraude

- *consulter* ou utiliser des fichiers ou portions d'autres équipes
- utiliser ou avoir dans son compte des projets d'années antérieures (y compris ses propres fichiers pour un redoublant)
  - ▶ Si vous en avez dans votre ordinateur, *supprimez-les*
  - ▶ Ne faites aucune recherche Web de projets GL
- laisser ses fichiers accessibles à d'autres (par ex. co-loc) (fraude passive)

**ATTENTION : 1 seule ligne de code copiée ou 1 seul test copié = 0 au projet GL**

Mieux vaut avoir 10 ou 11 (les plus basses notes du projet GL) que 0.  
En cas de doute : consulter vos enseignants



# Sommaire

- 1 Buts du projet
- 2 Vue d'ensemble du langage et du compilateur
- 3 Développement durable
- 4 Extension
- 5 Déroulement et planification
- 6 Ethique et fraude
- 7 Environnement de développement

# Environnement de développement

- Développement sous Linux
  - ▶ machines de l'école
  - ▶ **machines personnelles**

# Hierarchie des répertoires

/matieres/4MMPGL/GL/

global/

bin/

doc/

Makefile

Sources/

← machine abstraite ima et autres utilitaires

← documentation (y compris polycopiés fournis)

← sources de la machine abstraite

## Hiérarchie des répertoires (2)

Projet_GL/	← votre répertoire de projet
docs/	← vos docs de projet
examples/	← les exemples fournis
calc/	← l'exemple de la calculatrice
tools/	← un exemple de projet utilisant maven, junit, validate et Jacoco
plannings/	← les plannings prévisionnel et effectif du projet

## Hiérarchie des répertoires (3)

src/	← les sources du projet
main/	
antlr4/	← les sources des analyseurs lexical et syntaxique
bin/	← programme principal decac
config/	
java/fr/ensimag/	
deca/	← les sources du compilateur
tree/	← arbre abstrait
syntax/	← utilitaires étape A
context/	← utilitaires étape B
codegen/	← utilitaires étape C
tools/	← classes utilitaires communes
ima/pseudocode/	← instructions de la machine abstraite
resources/	
include/	← fichiers inclus

## Hiérarchie des répertoires (4)

src/	← les sources du projet
test/	← ce qui concerne les tests du compilateur
deca/	← les tests deca
syntax/	← les tests deca concernant l'étape A
context/	← les tests deca concernant l'étape B
codegen/	← les tests deca concernant l'étape C
java/fr/ensimag/deca	← les tests "unitaires"
tree/	← les tests "unitaires" concernant l'arbre abstrait
syntax/	← les tests "unitaires" concernant l'étape A
context/	← les tests "unitaires" concernant l'étape B
codegen	← les tests "unitaires" concernant l'étape C
script/	← les scripts shell de test
target/	← les fichiers générés
classes/	← les fichiers .class générés
generated-sources/	← les fichiers java générés (analyse lexicale et syntaxique)

# Travail en parallèle et gestion de versions

- Chaque membre d'une équipe :
  - ▶ travaille sur son compte personnel
  - ▶ possède une arborescence `Projet_GL`
- Synchronisation
  - Outil `Git`
    - ▶ permet synchronisation
    - ▶ sauvegarde de versions (« commits »)
- Chaque équipe a son compte `Git`
  - ▶ stocke les versions successives des fichiers du projet

# Utilisation de Git

- Au départ :
  - ▶ `git clone git@gitlab.ensimag.fr:gl2024/g8/gl42 Projet_GL`
- En cas de modification que l'on souhaite conserver :
  - ▶ `git commit -a`
- Pour envoyer un commit sur le dépôt :
  - ▶ `git push`
- Pour récupérer les commits des coéquipiers depuis le dépôt :
  - ▶ `git pull`
- Pour ajouter un fichier ou un dossier sur le dépôt :
  - ▶ `git add nom_fichier`
  - ▶ `git add nom_dossier`



# Utilisation de Git

- Rendu intermédiaire et fin de projet :
  - ▶ les enseignants récupèrent la dernière révision avant la date et l'heure limite sur la branche principale.
- Pour plus d'infos :
  - ▶ [Environnement]
  - ▶ le site du projet (<https://projet-gl.pages.ensimag.fr/git/>)
  - ▶ le manuel Git

## Conseils sur l'utilisation de Git

- Pour tous :
  - ▶ Ne **jamais** échanger de fichiers autrement que via Git (email, clé USB...), sauf si vous savez *vraiment* ce que vous faites

## Conseils sur l'utilisation de Git

- Pour tous :

- ▶ Ne **jamais** échanger de fichiers autrement que via Git (email, clé USB...), sauf si vous savez *vraiment* ce que vous faites
- ▶ Ne faites pas de changements inutiles sur votre code. Ne laissez pas votre IDE ou éditeur reformater du code autre que celui que vous venez d'écrire (sous NetBeans : sélectionnez la portion de code à reformater, puis Alt-Shift-F)

## Conseils sur l'utilisation de Git

- Pour tous :
  - ▶ Ne **jamais** échanger de fichiers autrement que via Git (email, clé USB...), sauf si vous savez *vraiment* ce que vous faites
  - ▶ Ne faites pas de changements inutiles sur votre code. Ne laissez pas votre IDE ou éditeur reformater du code autre que celui que vous venez d'écrire (sous NetBeans : sélectionnez la portion de code à reformater, puis Alt-Shift-F)
- Si vous n'êtes pas à l'aise avec Git :
  - ▶ Toujours utiliser `git commit` avec l'option **-a**
  - ▶ Faire un **git push** après chaque commit
  - ▶ Faire des **git pull** régulièrement
- Si vous êtes à l'aise avec Git : maintenir un historique propre, apprendre à utiliser `git add -p`, `git rebase [-i]`, ... (cf. Site Projet GL + amphi « Git avancé »)

# Maven

- Maven est un outil qui permet de construire un logiciel Java à partir de ses sources.
- Comparable à l'outil make sous Unix.
- Utilisation d'un *Project Object Model* (POM), qui permet de décrire un projet logiciel, ses dépendances avec des modules externes et l'ordre à suivre pour sa construction.
  - ▶ Fichier `Projet_GL/pom.xml`
- Fonctionne en réseau : Maven télécharge automatiquement les programmes externes requis.

# Commandes Maven

- Compilation
  - ▶ `mvn compile`
  - ▶ (dans le répertoire `Projet_GL`)
- Exécution du compilateur sur un fichier Deca
  - ▶ `./src/main/bin/decac test/deca/.../fichier.deca`
- Compilation et exécution des tests
  - ▶ `mvn test-compile`
  - ▶ `mvn verify`
  - ▶ (dans le répertoire `Projet_GL`)
- Nettoyage
  - ▶ `mvn clean`
  - ▶ efface les fichiers générés
- Génération de rapports (FindBugs, PMD, Jacoco...) et documentation (JavaDoc)
  - ▶ `mvn site`
  - ▶ `firefox target/site/index.html`