Projet Génie Logiciel
Notes de cours

Ensimag

Décembre 2023

Projet Génie Logiciel : introduction

Projet GL

Ensimag
Grenoble INP

9 décembre 2025

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.

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

Le compilateur

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

Le compilateur - cf. I-[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.

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/time`)
- 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 efficients
- Qualité de l'analyse réalisée (document en fin de projet)
  - Le point le plus important pour valider vos compétences en DD.

Extension

- Développer le compilateur pour le langage Deca *essentiel*
  - Partie précisément spécifiée et guidée
  - Les spécifications doivent être strictement respectées
  - ⇒ 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
  - ⇒ 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)

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 12 janvier 2026 à 12h00
  - Pris en compte dans la note finale

Déroulement du projet

- Récupérations des projets : programmes et tests du compilateur

  date Lundi 19 janvier 2026 à 16h00.
- Rétrospective collective

  date Mardi 20 janvier 2026.
- Soutenance [Soutenance]

  date Du jeudi 22 janvier au vendredi 23 janvier 2026.
- Extension

  date Au moment de la soutenance

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 19 janvier 2026 à 20h00
- Bilan
  - Bilan collectif sur la gestion d'équipe et de projet

  date Mercredi 21 janvier 2026 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

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

Environnement de développement

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

Hiérarchie des répertoires

```
/matieres/4MMPGL/GL/
  global/
    bin/              ← machine abstraite ima et autres utilitaires
    doc/              ← documentation (y compris polycopiés fournis)
    Makefile
    Sources/          ← 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 calculette
    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:gl2026/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
  - 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`

Projet Génie Logiciel

Étape A

Analyse lexicale, syntaxique et construction de l'arbre abstrait

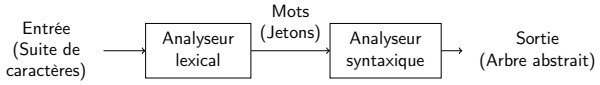
Projet GL

Ensimag
Grenoble INP

9 décembre 2025

Vue d'ensemble de l'étape A

- Étape A
  - Analyse lexicale
  - Analyse syntaxique
  - Construction de l'arbre abstrait
- Une passe sur le programme source Deca



Analyse lexicale

Partitionner une suite de caractères en une suite de mots.

- la suite de caractères :
  - = programme source en Deca
- les « mots » :
  - = unité lexicale
- à chaque unité lexicale est associé :
  - un « jeton » ou « token »

Analyse lexicale - Exemple

Prenons la suite de caractères : « $x = 2.5 * (y + 1);$ »
La suite de jetons correspondante est :

IDENT	['x']
EQUALS	['=']
FLOAT	['2.5']
TIMES	['*']
OPARENT	['(']
IDENT	['y']
PLUS	['+']
INT	['1']
CPARENT	[')']
SEMI	[';']

Analyse syntaxique

- L'analyse syntaxique permet de déterminer si une suite de mots est une phrase du langage.

⇒ Est-ce qu'une suite de jetons correspond à un programme Deca syntaxiquement correct ?

Outil
- La *grammaire hors-contexte* définit la syntaxe concrète du langage Deca, autrement dit l'ensemble des programmes syntaxiquement corrects (cf. II-[Syntaxe])

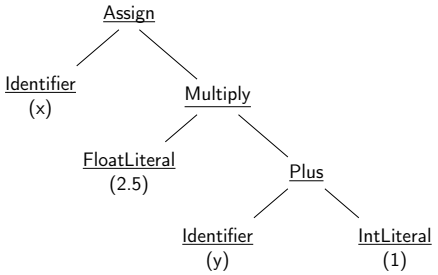
Construction de l'arbre abstrait

L'arbre abstrait est :

- une représentation structurée du programme Deca ;
- construit lors de l'analyse syntaxique.

Construction de l'arbre abstrait - Exemple

L'arbre abstrait correspondant à l'instruction : « $x = 2.5 * (y + 1);$ »



ANTLR : ANother Tool for Language Recognition

- ANTLR permet de générer des reconnaisseurs pour des langages :
  - analyseurs lexicaux (« *lexer* » en anglais),
  - analyseurs syntaxiques (« *parser* » en anglais).
- ANTLR est multi-langages : permet de générer des analyseurs en Ada 95, C, C#, Java, JavaScript, Perl, Python, Standard ML... etc.
- Analyse descendante ALL(*)
  Généralisation de l'analyse descendante LL(1)

Lexicographie

- La lexicographie de Deca décrit l'ensemble des mots du langage Deca.
- Description complète : II-[Lexicographie]
- Chaînes de caractères
  - STRING_CAR est l'ensemble de tous les caractères, à l'exception des caractères '"', '\' et de la fin de ligne.
  - STRING = '"' (STRING_CAR + '\"' + '\\')* '"'
    NB : dans poly, caractère '\' doublé comme en ANTLR.
- Commentaires :
  - Un commentaire commence par '/*' et termine par '*/'. Le commentaire s'arrête au premier '*/' suivant le début du commentaire.
  - Un commentaire est une suite de caractères (autre qu'une fin de ligne) qui commence par '//' et s'étend jusqu'à la fin de la ligne ou du fichier.

Syntaxe

- La syntaxe concrète de Deca décrit l'ensemble des phrases (ou programmes) correctes du langage Deca.
- Description complète de la syntaxe : grammaire de Deca cf. II-[Syntaxe].
- Grammaire utilisant une syntaxe étendue proche des grammaires ANTLR[1] : grammaire hors-contexte classique, avec en plus
  - des répétitions : « * »,
  - des parties optionnelles : « ? ».

1. Style « EBNF » : Extended Backus-Naur Form

Syntaxe

- Règle particulière :
  - au lieu du très inefficace (mais bien plus lisible)
    assign_expr
    → **or_expr**
    | **lvalue** '=' **assign_expr**
  - on utilise
    assign_expr
    → e=**or_expr** (
    { condition : expression e must be a "**lvalue**" }
    '=' **assign_expr**
    | ε)
  - e est l'expression correspondant à **or_expr**.
  - La condition sur e doit être vérifiée lors de l'analyse syntaxique (condition sur l'arbre construit pour la partie gauche).

Exercice

```
{
   int i = 1;
   println(i, " : ok");
}
```

Question

Le programme Deca est-il syntaxiquement correct ?

Exercice

Question

Écrire un programme qui définit une classe Compteur, avec un champ x de type int, une méthode incr() et un programme principal qui fait fonctionner le compteur.

Correction

Structure d'un fichier source ANTLR

- Analyseur lexical :
  lexer grammar *nomDeClasse*;
  - L'analyseur lexical généré est une classe de nom *nomDeClasse*;
- Analyseur syntaxique :
  parser grammar *nomDeClasse*;
  - L'analyseur syntaxique sera une classe de nom *nomDeClasse*;
- options {
  nom-option = *valeur*;
  nom-option2 = *valeur2*;
  ...
  }
  - language : le langage de programmation (défaut : Java)
  - superClass : le code généré est une classe qui hérite de cette classe
  - tokenVocab : le vocabulaire d'entrée à utiliser (le nom de l'analyseur lexical dans le cas d'un analyseur syntaxique)

Structure d'un fichier source ANTLR

- @header { ... }
  - Portion de code (Java) ajouté en tête du fichier généré
  - Exemple :
    @header {
    import *nom.du.paquetage*;
    }
- @members { ... }
  - Portion de code (Java) ajoutée dans la classe générée.
  - Peut être utilisé pour ajouter des champs et des méthodes dans la classe générée.
- La dernière partie du fichier est constituée de *règles* qui sont appliquées pour réaliser l'analyse lexicale ou syntaxique.

Règles pour l'analyse lexicale

- Une *règle* est de la forme :
  `NOM_REGLE : expression-régulière;`
- Le nom de la règle doit commencer par une majuscule.
- Exemple : `PLUS : '+';`
- Lorsque l'expression régulière est reconnue, un jeton correspondant au nom de la règle est renvoyé.
- Le jeton est un objet de type `org.antlr.runtime.Token`.
- `PLUS: [@6,13:13='+',<45>,1:13]`
  - 6 : 6ème jeton reconnu
  - 13:13 : la chaîne '+' commence au 13ème caractère du fichier source, et se termine sur ce même 13ème caractère.
  - <45> : code du jeton
  - 1 : numéro de ligne
  - 13 : numéro de colonne

Action associée à une règle

- Portion de code Java qui est effectuée avant de renvoyer le jeton.
  `NOM_REGLE : expression-régulière { action } ;`
  `ESPACE : ' ' { System.out.println("espace reconnu"); };`
- Fonction utile : `skip();`
  Permet de ne pas renvoyer le jeton correspondant
  `ESPACE : ' ' { skip(); } ;`
- Autre fonction utile : `getText();`
  Permet de récupérer le texte source correspondant
- Fragment de règle : règle qui peut être utilisée dans d'autres règles, et ne produisant pas de jeton (macros)
  `fragment CHIFFRE : '0' .. '9';`
  `NOMBRE : CHIFFRE+;`

Syntaxe des expressions régulières

`'c'`	le caractère *c*
`'\n'`	passage à la ligne (LF)
`'\r'`	retour chariot (CR)
`'\t'`	tabulation
`'\\'`	le caractère « \ »
`'\''`	le caractère « ' »
`'chaine'`	la chaîne de caractères
`.`	caractère quelconque (y compris une fin de ligne)
expr1 expr2	*expr1* suivie de *expr2*
`'c1'..'c2'`	caractères compris entre les caractères *c1* et *c2*
`( expr )`	expression *expr*
expr1 `|` *expr2*	expression *expr1* ou *expr2*
expr`*`	expression *expr* répétée entre 0 et *n* fois
expr`+`	expression *expr* répétée entre 1 et *n* fois
`~`*expr*	tout caractère sauf *expr* (*expr* ne doit reconnaître que des caractères)

Résolution des ambiguïtés

Principe de la plus longue correspondance

- Pour chaque règle, l'analyseur lexical tente de reconnaître la chaîne la plus longue possible.
- Exemple

```
ELSE : 'else';
ELSEIF : 'elseif';
IF : 'if';
SPACE : ' ';
```

  - `else if` ↦ `ELSE SPACE IF`
  - `elseif` ↦ `ELSEIF` (et non `ELSE IF`)
- Si deux règles peuvent s'appliquer pour reconnaître deux chaînes de la même longueur : la première règle est prioritaire.
- Exemple :
  `DEFAULT : .;`
  en fin de fichier pour reconnaître « tous les autres caractères ».

Traitement des commentaires Deca

- Si on définit les commentaires de la façon suivante :
  `COMMENT : '/*' .* '*/' { skip(); } ;`
  Si en entrée, on a « `/* foo */ bar */` », l'analyseur lexical va reconnaître « `/* foo */ bar */` », alors qu'il faudrait s'arrêter au premier « `*/` ».
- Une solution
  On utilise "*?" l'étoile « non gloutonne » en écrivant :
  `COMMENT : '/*' .*? '*/'`
  `{ skip(); } ;`
  github.com/antlr/antlr4/blob/master/doc/wildcard.md
  Exercice : définir l'étoile non gloutonne à partir des opérateurs classiques des expressions régulières

Travail à réaliser pour l'analyse lexicale

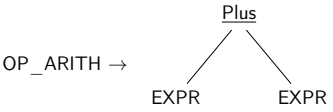
- Pour l'analyse lexicale, travail à réaliser :
  - Compléter le fichier `src/main/antlr4/fr/ensimag/deca/syntax/DecaLexer.g4`
  - La classe `DecaLexer` hérite de la classe `AbstractDecaLexer`.
  - L'inclusion de fichier est traitée en analyse lexicale. Utiliser la méthode `doInclude` de `AbstractDecaLexer.java`.
- Utiliser le script `test_lex` pour faire des tests.

Arbre abstrait

- Arbre abstrait : représentation structurée du programme Deca
- Les constructions d'arbres sont décrites par une *grammaire d'arbres*,
  - définit la *syntaxique abstraite* du langage Deca.
- Référence : II-[SyntaxeAbstraite]

Grammaire d'arbres

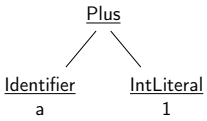
- Vocabulaire terminal : nœuds de l'arbre
  Program, EmptyMain, Main, Identifier...
- Vocabulaire non terminal :
  PROGRAM, MAIN, DECL_VAR, IDENTIFIER...
- Règles :
  OP_ARITH → Plus [EXPR EXPR] ≡



IDENTIFIER → Identifier ↑ *Symbol*

Grammaire d'arbres

Par exemple, l'expression « $a + 1$ » correspond à l'arbre :



Codage des *listes d'arbres* avec une racine "anonyme" :

PROGRAM → Program [LIST_DECL_CLASS MAIN]
LIST_DECL_CLASS → [DECL_CLASS*]

Exercices

Question

Construire l'arbre abstrait correspondant aux programmes suivants.

Exercices

- Programme 1 :

```
// rien !!!
```

- Programme 2 :

```
{}
```

- Programme 3 :

```
class A extends B
{ int x = 1; }
```

- Programme 4 :

```
void setX(int x) {
    this.x = x;
}
```

- Programme 5 :

```
{
    A a = new A();
    println("a.getX() = ",
            a.getX());
}
```

- Programme 6 :

```
if (a == 1) {
    x = y;
} else if (a == 2) {
    x = t;
}
```


Classes Java pour la grammaire d'arbres

- Classes définies dans `src/main/java/fr/ensimag/deca/tree/`.
- Patron de conception « *Interprète* » (cf. III-[ConventionsCodage])
- Non terminaux : classe abstraites Java

Non terminal	classe Java
PROGRAM	AbstractProgram
MAIN	AbstractMain
DECL_VAR	AbstractDeclVar
IDENTIFIER	AbstractIdentifier

- Terminaux : classes concrètes Java

terminal	classe Java
Program	Program
EmptyMain	EmptyMain
DeclVar	DeclVar
Identifier	Identifier

- Règle $X \to Y \quad \longmapsto$ (abstract) class Y extends X
- Règle $X \to Y\,[A\ B] \quad \longmapsto$ la classe Y (ou une classe de base) contient des champs de types A et B.

Classes Java pour la grammaire d'arbres

Exemple : les expressions

- Grammaire d'arbres :

  EXPR → BINARY_EXPR
  BINARY_EXPR → OP_ARITH | ...
  OP_ARITH → *Plus*[EXPR EXPR] | ...

- Classes :

```
abstract class AbstractExpr {...}

abstract class AbstractBinaryExpr extends AbstractExpr {
    AbstractExpr leftOperand;
    AbstractExpr rightOperand;
}

abstract class AbstractOpArith extends AbstractBinaryExpr {...}

class Plus extends AbstractOpArith {...}
```


Parcours d'arbre avec le patron interprète

Exemple : les expressions

```
abstract class AbstractBinaryExpr extends AbstractExpr {
    AbstractExpr leftOperand, rightOperand;
    public void decompile(IndentPrintStream s) {
        s.print("(");
        leftOperand.decompile(s);
        s.print(" " + getOperatorName() + " ");
        rightOperand.decompile(s);
        s.print(")");
    }
    abstract protected String getOperatorName();
}

class Plus extends AbstractOpArith {
    protected String getOperatorName() { return "+"; }
}
```

Factoriser le code « le plus haut possible »
dans la hiérarchie de classes

Classe abstraite `Tree`

- Toutes les classes pour les arbres sont définies dans le paquetage `fr.ensimag.deca.tree`.
- On a une classe abstraite `Tree` dont héritent toutes les autres classes.
- À chaque arbre est associé un objet de type `Location`, qui comporte
  - un nom de fichier (`fileName`),
  - un numéro de ligne (`line`),
  - un numéro de colonne (`positionInLine`).
- Permet de stocker la position d'un élément correspondant à un nœud de l'arbre dans le fichier source.
  (Utile pour les messages d'erreur).

Règles des grammaires ANTLR

- Exemple de règles

```
sum_expr : mult_expr (PLUS mult_expr)*;
mult_expr: INTEGER;
```

  - `mult_expr` dérive vers un entier ;
  - `sum_expr` dérive vers une suite de `mult_expr` séparées par des `+` ;
  - reconnaît par exemple 42, 3+42, 12+28+1.
- Les non-terminaux commencent par une minuscule ; les terminaux (jetons produits par le lexer) sont en majuscule.
- Construction des règles
  - `(expr1 | expr2)` : `expr1` ou `expr2` ;
  - `(expr)*` : `expr`, répétée entre 0 et n fois ;
  - `(expr)?` : `expr`, 0 ou 1 fois.

Principe de l'analyse syntaxique

- `parser grammar DecaParser;`
  - ANTLR génère un analyseur syntaxique dans la classe `DecaParser`
- Pour chaque non-terminal de la grammaire, ANTLR génère une méthode dans la classe `DecaParser`.
  - Un appel à cette méthode reconnaît et consomme une séquence de jetons produits par l'analyse lexicale.
- Quand apparaît en partie droite de règle
  - un terminal : l'analyseur syntaxique vérifie que le jeton est correct et consomme ce jeton ;
  - un non-terminal : l'analyseur syntaxique effectue un appel récursif à la méthode correspondante.

Exemple 1

- Pour les règles

```
nombre_negatif : MINUS nombre;
nombre : INT;
```

ANTLR produit un code qui ressemble à

```
public final ... nombre_negatif()
                 throws RecognitionException {
    ...
    // NomDuFichier.g4:24:16: ( MINUS nombre )
    match(MINUS);
    nombre();
    ...
}

public final ... nombre()
                 throws RecognitionException { ... }
```

- L'exception `RecognitionException` est levée si une erreur de syntaxe

Exemple 2

- Pour une règle de la forme `expr : expr1 | expr2;`
  ANTLR produit un code qui ressemble à

```
public final ... expr()
                 throws RecognitionException {
    switch (choisir_branche) {
    case 1:
        expr1(); break;
    case 2:
        expr2(); break;
    }
}
```


Actions associées à une règle

- Action associée à une règle : portion de code Java entre accolades
  - Ce code est inséré dans le corps des méthodes d'analyse
  - Exemple :

```
prog : debut { System.out.println("j'ai vu debut"); }
       suite { System.out.println("j'ai vu suite"); }
     ;
```

  - Après la reconnaissance de `debut`, affiche `"j'ai vu debut"` ; après la reconnaissance de `suite`, affiche `"j'ai vu suite"`.

Résultat de la méthode associée à un non-terminal

- La méthode associée au non-terminal `toto` retourne un objet de type `TotoContext`
- On peut ajouter des attributs synthétisés qui seront des champs de cet objet : *non_terminal* `returns[`*type1 nom1, type2 nom2*`]`
- Exemple

```
expr returns[int val] :
   sum_expr { $val = $sum_expr.val; }
   ;
sum_expr returns[int val] :
   e=mult_expr {$val = $e.val;}
   (PLUS e2=mult_expr {$val = $val + $e2.val;})*
   ;
mult_expr returns[int val] :
   INTEGER {$val = Integer.parseInt($INTEGER.text);}
   ;
```

- `$INTEGER.text` représente la chaîne de caractères qui correspond à l'entier reconnu.

Paramètres de la méthode associée à un non-terminal

- Exemple

```
sum_expr returns[int val] :
   mult_expr {$val = $mult_expr.val;}
   (plus_mult_expr[$val] {$val = $plus_mult_expr.after;})*
   ;

plus_mult_expr[int before] returns[int after] :
   PLUS mult_expr { $after = $before + $mult_expr.val; }
   ;
```

- Remarque
  L'objet en retour contient un uplet des *attributs synthétisés* ; les paramètres correspondent à des *attributs hérités* d'une *grammaire attribuée*.

  `plus_mult_expr[int before] returns[int after]`
  $\equiv$ *plus_mult_expr* $\downarrow$*before* $\uparrow$*after*

Section @init

- Déclaration de variables locales / initialisations
- Exemple

```
expr returns[int val]
@init {
   int i;
   $val = 0;
}
   : expr1 { i = 42; $val = $val + 1 }
   | expr2 { i = 43; $val = $val + 2 }
```

- Les déclarations/initialisations s'appliquent à toute la méthode `int expr()`.

Gestion des erreurs

- En cas d'erreur de syntaxe : l'analyseur lève l'exception `RecognitionException`.
- On peut également lever cette exception explicitement dans une règle.
- Par défaut, chaque méthode d'analyse générée rattrape l'exception avec la construction :

```
try {
   // corps de la regle
} catch (RecognitionException re) {
   _errHandler.reportError(this, re);
   _errHandler.recover(this, re);
}
```

Gestion des erreurs dans le cadre du projet

- Dans le cadre du projet, on s'arrête à la première erreur détectée (étapes B et C). En étape A, on laisse le rattrapage d'erreur d'ANTLR travailler sur les cas triviaux.
- Pour l'analyse syntaxique, pour s'arrêter à la première erreur de syntaxe non-triviale, on initialise l'objet `_errHandler` à `new DefaultErrorStrategy()` (dans le constructeur `AbstractDecaParser`).

Construction de l'arbre

- Au cours de l'analyse syntaxique, on construit l'arbre abstrait du programme Deca source.
- Fichier source : `DecaParser.g4`, qui génère `DecaParser.java`.
- La classe `DecaParser` étend la classe abstraite `AbstractDecaParser`.

Règles de DecaParser.g4

```
prog returns[AbstractProgram tree]
    : list_classes main EOF {
        // Verification des attributs
        // (pas indispensable, mais aide au debug)
        assert($list_classes.tree != null);
        assert($main.tree != null);

        // Construction de l'arbre
        $tree = new Program($list_classes.tree,
                            $main.tree);

        // Initialisation du champ 'location'
        // de l'arbre $tree a partir du jeton $main
        setLocation($tree, $main.start);
      }
    ;
```


Règles de DecaParser.g4

```
prog returns[AbstractProgram tree]
    : list_classes main EOF { ... $tree = ...; };

main returns[AbstractMain tree]
    : /* epsilon */ {
        $tree = new EmptyMain();
      }
    | block {
        assert($block.decls != null);
        assert($block.insts != null);
        $tree = new Main($block.decls, $block.insts);
        setLocation($tree, $block.start);
      }
    ;
```


Règles de DecaParser.g4

```
block returns[ListDeclVar decls, ListInst insts]
    : OBRACE list_decl list_inst CBRACE {
       assert($list_decl.tree != null);
       assert($list_inst.tree != null);
       $decls = $list_decl.tree;
       $insts = $list_inst.tree;
      }
    ;

list_decl returns[ListDeclVar tree]
        // tree est ici une liste d'arbres
  @init {
     $tree = new ListDeclVar();
  }
     : decl_var_set[$tree]*
     ;

...
```


Travail à réaliser pour l'analyse syntaxique

cf. I-[Consignes]

- Pour l'analyse syntaxique, travail à réaliser :
  - Compléter le fichier `src/main/antlr4/fr/ensimag/deca/syntax/DecaParser.g4`
  - Compléter le paquetage des constructeurs de l'arbre `src/main/java/fr/ensimag/deca/tree/*.java`
  - Utiliser le script `test_synt` pour faire des tests.

Notion d'« erreur » dans le projet

On distingue plusieurs sortes d'erreurs :

- Erreurs du programme Deca
  - Programme Deca incorrect, doit être rejeté.
- Limitations du compilateur : programme Deca peut-être correct, mais le compilateur ne peut pas le compiler
  - limitation du langage ou des outils utilisés,
  - portion du langage non implémentée.
- Avertissements (« warnings ») : programme correct, mais le compilateur a détecté un problème.
  - Le compilateur DOIT générer du code.
  - Il PEUT afficher un message *UNIQUEMENT si l'option -w est utilisée.*
- Erreurs internes :
  - Le compilateur a détecté une erreur dans lui-même
  - (pas une erreur du programme Deca).

Erreurs du programme Deca

- Chaque étape du compilateur, ainsi que la ligne de commande, peut provoquer une erreur.
- On s'arrête à la première erreur détectée, on ne fait pas de récupération d'erreur (choix imposé).
- Programmation : utilisation d'exceptions Java
- Lorsqu'une erreur est détectée, un message d'erreur est affiché.
- Format des messages d'erreur (imposé : à RESPECTER STRICTEMENT)
  `<nom-fichier>:<no-ligne>:<no-colonne>: <description>`
  Exemple : `fichier.deca:12:4: caractère '#' non autorisé`
  - pas d'espace entre le nom du fichier et `':'`,
  - pas d'espace autour des numéros de ligne et de colonne.

Hiérarchie d'exceptions fournies

Exception
- ◁— RuntimeException
  - ◁— RecognitionException (erreurs étape A)
    - ◁— IncludeFileNotFound
    - ◁— InvalidLValue
    - ◁— CircularInclude
  - ◁— DecacInternalError (erreurs internes)
- ◁— LocationException
  - ◁— Contextual error (erreurs étape B)
- ◁— CLIException (erreur ligne de commande)
- ◁— DecacFatalError (erreur de lecture du fichier source ou d'écriture du fichier cible)

- D'autres exceptions peuvent être définies.

Projet Génie Logiciel

Étape B

Analyse contextuelle

Projet GL

Ensimag
Grenoble INP

9 décembre 2025

Grammaires attribuées

- Les grammaires attribuées
  - permettent de définir une classe de langages plus grande que les grammaires hors-contexte,
  - décrivent des calculs dirigés par la syntaxe.
  - permettent d'associer une interprétation à la syntaxe (sémantique dénotationnelle).
- On associe à chaque terminal et non terminal d'une grammaire hors-contexte des *attributs* (sortes de paramètres)
- Attributs *typés* (domaine de valeurs)
- Types : entier, réel, chaîne de caractères, ensemble, fonction... etc.

Attributs

- On distingue :
  - les attributs hérités X↓*att*
    Valeur dépendant du contexte dans lequel X est dérivé.
    Transmis du père vers le fils dans l'arbre de dérivation.
    cf. paramètres « in » en Ada, paramètres d'une fonction.
  - les attributs synthétisés X↑*att*
    Valeur dépendant des règles appliquées pour dériver X
    Transmis du fils vers le père dans l'arbre de dérivation.
    cf. paramètres « out » en Ada, ou valeur(s) de retour d'une fonction.
- Description du calcul des attributs : pour chaque règle de la forme
  $$X \to Y_1 Y_2 \dots Y_n$$
  on définit :
  - les attributs hérités de Y_i en fonction des attributs de X, Y_1, ... Y_n ;
  - les attributs synthétisés de X en fonction des attributs de X, Y_1, ... Y_n.

Exemple : langage $L = \{a^n b^n c^n \ ; \ n \in \mathbb{N}\}$

- On considère $L_1 = \{a^n b^n c^p \ ; \ n, p \in \mathbb{N}\}$
- Grammaire qui engendre L_1 :
  $$\begin{array}{lcl} S & \to & A\ C \\ A & \to & \varepsilon \mid a\ A\ b \\ C & \to & \varepsilon \mid c\ C \end{array}$$
- Variante, avec style EBNF :
  $$\begin{array}{lcl} S & \to & A\ C \\ A & \to & \varepsilon \mid a\ A\ b \\ C & \to & c* \end{array}$$

Exemple : langage $L = \{a^n b^n c^n \ ; \ n \in \mathbb{N}\}$

- Définition de L à l'aide d'une grammaire attribuée
- Attributs synthétisés :
  $A{\uparrow}n$, $n : \mathbb{N}$ (nombre de 'a' de la chaîne)
  $C{\uparrow}n$, $n : \mathbb{N}$ (nombre de 'c' de la chaîne)
- Grammaire attribuée qui engendre L :
  $$\begin{array}{lcl} S & \to & A{\uparrow}n\ \ C{\uparrow}p \\ & & \text{condition } n = p \\ A{\uparrow}0 & \to & \varepsilon \\ A{\uparrow}n+1 & \to & a\ A{\uparrow}n\ b \\ C{\uparrow}0 & \to & \varepsilon \\ C{\uparrow}n+1 & \to & c\ C{\uparrow}n \end{array}$$

Exemple : langage $L = \{a^n b^n c^n \ ; \ n \in \mathbb{N}\}$

- Autre grammaire attribuée pour L
- Attributs synthétisé et hérité :
  $A{\uparrow}n$, $n : \mathbb{N}$ (nombre de 'a' de la chaîne)
  $C{\downarrow}n$, $n : \mathbb{Z}$ (nombre de 'c' que le contexte impose pour C)
- Grammaire attribuée qui engendre L :
  $$\begin{array}{lcl} S & \to & A{\uparrow}n\ \ C{\downarrow}n \\ A{\uparrow}0 & \to & \varepsilon \\ A{\uparrow}n+1 & \to & a\ A{\uparrow}n\ b \\ C{\downarrow}n & \to & \varepsilon \\ & & \text{condition } n = 0 \\ C{\downarrow}n & \to & c\ C{\downarrow}n-1 \end{array}$$

Exemple : langage $L = \{a^n b^n c^n \ ; \ n \in \mathbb{N}\}$

- Encore une autre grammaire attribuée pour L (avec petites extensions de syntaxe).
- Attributs synthétisé et hérité :
  $A{\downarrow}n$, $n : \mathbb{Z}$ (nombre de 'a' que le contexte impose pour A)
  $C{\uparrow}n$, $n : \mathbb{N}$ (nombre de 'c' de la chaîne C)
- Grammaire attribuée qui engendre L :
  $$\begin{array}{lcl} S & \to & A{\downarrow}n\ \ C{\uparrow}n \\ A{\downarrow}0 & \to & \varepsilon \\ A{\downarrow}n & \to & a\ A{\downarrow}n-1\ b \\ C{\uparrow}n & \to & \{\ n := 0\ \}\ (\mathrm{c}\ \{\ n := n+1\ \})* \end{array}$$

Propriétés contextuelles d'un programme

- Propriétés *contextuelles* d'un programme :
  ne peuvent pas être décrites par une grammaire hors-contexte
  - déclaration et utilisation des identificateurs ;
  - typage des expressions.
- Propriétés contextuelles : décrites par des règles contextuelles.
- Nécessaire pour définir la *sémantique statique* du langage (cf. slide suivant).
- Règle contextuelle non respectée ⇒ message d'erreur contextuelle

Sémantique Statique $\triangleq$ Syntaxe Contextuelle

- Syntaxe Contextuelle $\triangleq$ ensemble des programmes pour lesquels le compilateur doit produire un exécutable (sauf limitations).

  NB : la sémantique (dynamique) dépend souvent du *typage statique*. Exemple Java : la sémantique de "`o.equals(x)`" dépend du *type dynamique* de "o" **et** du *type statique* de "x".

- Nécessite une définition rigoureuse pour "compatibilité" des compilateurs. **Vous ne pouvez pas l'inventer !**

  Exemple : ce programme est-il accepté par les compilateurs Java ?

```
class A {
  void f(){
    A A = new A();
    A.f();
  }
}
```


Exemple

- Langage à structure de bloc

```
                        -- niveau 1
declare                 -- niveau 2
   x, y : integer;
begin
   x := 1;
   declare              -- niveau 3
      x : boolean;
   begin
      x := true;
      y := 1;
   end;                 -- niveau 2
   x := x + y;
end;                    -- niveau 1
```


Règles contextuelles du langage

- Les identificateurs `integer` et `boolean` sont des identificateurs de type prédéfinis.
- Les identificateurs `true` et `false` sont des identificateurs de constantes booléennes prédéfinis.
- Un identificateur ne peut pas être déclaré plus d'une fois au même niveau.
- Tout identificateur utilisé dans les instructions doit :
  - être préalablement déclaré (soit dans le même bloc, soit dans un bloc englobant)
  - être utilisé conformément à sa déclaration.

Type, nature, définition et environnement

- Types du langage : entier ou booleen
  Type $\triangleq$ { entier, booleen }
- Nature des identificateurs : type, variable ou constante
  Nature $\triangleq$ { var, type, const }
- Définition d'un identificateur : nature et type
  Définition $\triangleq$ Nature $\times$ Type
- Symbol : domaine des identificateurs
- Environnement : associe à un identificateur sa définition
  Environnement $\triangleq$ Symbol $\rightarrow$ Définition (fonction partielle)
- $env(x)$: définition associée à $x \in$ Symbol dans env.
- $dom(env)$: domaine de l'environnement env (ensemble des identificateurs auquels est associée une définition)

Environnements

- Environnement prédéfini :

$$\text{Predef} \triangleq \{ \begin{array}{lcl} \text{integer} & \mapsto & (\underline{\text{type}}, \underline{\text{entier}}), \\ \text{boolean} & \mapsto & (\underline{\text{type}}, \underline{\text{booleen}}), \\ \text{true} & \mapsto & (\underline{\text{const}}, \underline{\text{booleen}}), \\ \text{false} & \mapsto & (\underline{\text{const}}, \underline{\text{booleen}}) \end{array} \}$$

- Traitement de la structure de bloc : *empilement* d'environnements

env_3	niveau 3
env_2	niveau 2
Predef	niveau 1

  - $env_2 = \{ \text{x} \mapsto (\underline{\text{var}}, \underline{\text{entier}}), \text{y} \mapsto (\underline{\text{var}}, \underline{\text{entier}}) \}$
  - $env_3 = \{ \text{x} \mapsto (\underline{\text{var}}, \underline{\text{booleen}}) \}$

Opérations sur les environnements

Soit deux environnements env_1 et env_2.

- Union disjointe de deux environnements
  - $env_1 \oplus env_2$ n'est pas défini, si $dom(env_1) \cap dom(env_2) \neq \varnothing$
  - $(env_1 \oplus env_2)(x) \triangleq \begin{cases} env_1(x) \text{ si } x \in dom(env_1) \\ env_2(x) \text{ si } x \in dom(env_2) \end{cases}$

  Permet de traiter les déclarations d'identificateurs qui sont au même niveau.

- Empilement de deux environnements
  - $(env_1/env_2)(x) \triangleq \begin{cases} env_1(x) \text{ si } x \in dom(env_1) \\ env_2(x) \text{ si } x \notin dom(env_1) \text{ et } x \in dom(env_2) \\ \text{indéfini sinon} \end{cases}$

Syntaxe abstraite du mini-langage à blocs

PROG	$\rightarrow$	BLOC
BLOC	$\rightarrow$	`Bloc`[LIST_DECL LIST_INST]
LIST_DECL	$\rightarrow$	[DECL*]
DECL	$\rightarrow$	`Decl`[IDF TYPE]
TYPE	$\rightarrow$	IDF
LIST_INST	$\rightarrow$	[INST*]
INST	$\rightarrow$	BLOC | `Assign`[IDF EXP]
EXP	$\rightarrow$	IDF | `Num` | `Plus`[EXP EXP]
IDF	$\rightarrow$	`Idf`

Profils des symboles de la syntaxe contextuelle

- BLOC $\downarrow env$ $\quad env$: Environnement (environnement englobant)
- LIST_DECL $\downarrow env\_glob \uparrow env$
  env_glob : Environnement (environnement englobant)
  env : Environnement (environnement des déclarations)
- DECL $\downarrow env\_glob \uparrow env$
- IDF $\downarrow t \uparrow env$ $\quad t$: Type
- TYPE $\downarrow env\_glob \uparrow t$ $\quad t$: Type
- LIST_INST $\downarrow env$
- INST $\downarrow env$
- EXP $\downarrow env \uparrow t$ $\quad t$: TYPE
- `Idf` $\uparrow nom$ $\quad nom$: Symbol

Règles de la syntaxe contextuelle

PROG → BLOC $\downarrow$*Predef*
BLOC $\downarrow$*env_glob* → Bloc [LIST_DECL $\downarrow$*env_glob* $\uparrow$*env*
LIST_INST $\downarrow$*env*/*env_glob*]

LIST_DECL $\downarrow$*env_glob* $\uparrow$*env* →
{ *env* := ∅ }[(DECL $\downarrow$*env_glob* $\uparrow$*env*$_1$ { *env* := *env* ⊕ *env*$_1$ })*]
condition implicite : *dom*(*env*) initial et *dom*(*env*$_1$) disjoints

DECL $\downarrow$*env_glob* $\uparrow$*env* → Decl[IDF $\downarrow t$ $\uparrow$*env* TYPE $\downarrow$*env_glob* $\uparrow t$]

IDF $\downarrow t$ $\uparrow\{nom \mapsto (\text{var}, t)\}$ → Idf $\uparrow$*nom*

TYPE $\downarrow$*env_glob* $\uparrow t$ → Idf $\uparrow$*nom*
condition : $(\text{type}, t) \triangleq env\_glob(nom)$
condition implicite additionnelle : $nom \in dom(env\_glob)$

Règles contextuelles (suite)

LIST_INST $\downarrow$*env* → [(INST $\downarrow$*env*)*]

INST $\downarrow$*env* → BLOC $\downarrow$*env*
→ Assign[Idf $\uparrow$*nom* EXP $\downarrow$*env* $\uparrow t$]
condition : $env(nom) = (\text{var}, t)$

EXP $\downarrow$*env* $\uparrow t$ → Idf $\uparrow$*nom*
condition : $(nat, t) \triangleq env(nom)$ et $nat \in \{\text{var}, \text{const}\}$

EXP $\downarrow$*env* $\uparrow$entier → Num
→ Plus[EXP $\downarrow$*env* $\uparrow t_1$ EXP $\downarrow$*env* $\uparrow t_2$]
condition : $t_1 = t_2 = \text{entier}$

Les programmes suivants sont-ils acceptés ?

1.
```
begin
   false := true;
end;
```

2.
```
declare
   integer : integer;
begin
   integer := integer + 1;
end;
```

3.
```
declare
   integer : integer;
begin
   declare
      x : integer;
   begin
      x := 1;
   end;
end;
```


Introduction

- cf. II-[SyntaxeContextuelle]
- Vérification contextuelle de Deca : nécessite trois passes sur le programme.
- Exemple

```
class A {
   B b;
}

class B {
   A a;
}
```

```
class Parcours {
   void parcoursA(A a) {
      if (a != null) {
         parcoursB(a.b);
      }
   }
   void parcoursB(B b) {
      if (b != null) {
         parcoursA(b.a);
      }
   }
}
```


Trois passes sur le programme

- Déclaration de champ ou méthode : référence à une classe qui apparait après. Ex : « `B b` » dans la classe « `A` »
  - Passe 1 : on vérifie le nom des classes et la hiérarchie de classes
- Remarque : Deca, contrairement à Java, impose que les super-classes soient déclarées avant les sous-classes. Par exemple

```
class D extends C { }
class C { }
```

  est un programme Deca incorrect.
- Méthodes mutuellement récursives (`parcoursA` et `parcoursB`)
  - Passe 2 : on vérifie les déclarations de champs et les signatures des méthodes.
  - Passe 3 : on vérifie le corps des méthodes, les expressions d'initialisation des champs, et le programme principal.

Exemple

- Les trois passes sur l'exemple :

```
class A {
   B b;
}

class B {
   A a;
}
```

```
class Parcours {
   void parcoursA(A a) {
      if (a != null) {
         parcoursB(a.b);
      }
   }
   void parcoursB(B b) {
      if (b != null) {
         parcoursA(b.a);
      }
   }
}
```

- Passe 1 : `class A ; class B ; class Parcours ;`
- Passe 2 : `B b ; A a ; parcoursA(A a) ; parcoursB(B b) ;`
- Passe 3 : corps de `parcoursA` et `parcoursB`

Domaines d'attributs de la grammaire attribuée de Deca

- Symbol : ensemble des identificateurs Deca
- Type $\triangleq$ {void, boolean, float, int, string, null}
  $\cup$ type_class(Symbol)
  À chaque classe A du programme correspond un type type_class(A).
- Visibility $\triangleq$ {protected, public}
- TypeNature $\triangleq$ {type} $\cup$ class(Profil)
- ExpNature $\triangleq$ {param, var} $\cup$ method(Signature)
  $\cup$ field(Visibility, Symbol)
  - field(public, A) : champ public d'une classe A
  - field(protected, A) : champ protégé d'une classe A
- Signature d'une méthode : liste (ordonnée) des types de ses paramètres
  Signature $\triangleq$ Type*

Domaines d'attributs de la grammaire attribuée de Deca

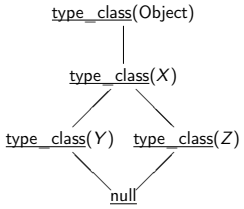
- Extension : nom de la super-classe, ou 0 pour Object
  Extension $\triangleq$ Symbol $\cup$ {0}
- Profil d'une classe : nom de la super-classe et environnement des champs et méthodes de la classe
  Profil $\triangleq$ Extension $\times$ EnvironmentExp
- Definition d'un identificateur : sa nature et son type
  TypeDefinition $\triangleq$ TypeNature $\times$ Type
  ExpDefinition $\triangleq$ ExpNature $\times$ Type
  - type de l'objet pour un identificateur param, var ou field
  - type du résultat pour un identificateur de méthode
- Environnement : associe à un identificateur sa définition
  EnvironmentType $\triangleq$ Symbol $\rightarrow$ TypeDefinition (fonction partielle)
  EnvironmentExp $\triangleq$ Symbol $\rightarrow$ ExpDefinition (fonction partielle)

Relation de sous-typage

- *env* de EnvironmentType
- Relation de sous-typage relative à *env* :
  - Pour tout type T, T est un sous-type de T.
  - Pour toute classe A, type_class(A) est un sous-type de type_class(Object).
  - Si une classe B étend une classe A dans l'environnement *env*, alors type_class(B) est un sous-type de type_class(A).
  - Si une classe C étend une classe B dans l'environnement *env* et si type_class(B) est un sous-type de T, alors type_class(C) est un sous-type de T.
  - Pour toute classe A, null est un sous-type de type_class(A).
- Notation : subtype(*env*, T_1, T_2)
  T_1 est un sous-type de T_2 relativement à *env*

Relation de sous-typage

```
class X { };
class Y extends X { };
class Z extends X { };
```



void boolean float int string

Autres opérations

- cf. II-[SyntaxeContextuelle] (section 2)
- Compatibilité pour l'affectation
- Compatibilité pour la conversion
- Compatibilité des opérations unaires et binaires

Environnements prédéfinis

- env_types_predef : types prédéfinis (dans EnvironnementType)

  env_types_predef $\triangleq$ {
  void $\mapsto$ (type, void),
  boolean $\mapsto$ (type, boolean),
  float $\mapsto$ (type, float),
  int $\mapsto$ (type, int),
  Object $\mapsto$ (class(0, env_exp_object), type_class(Object))
  }

- env_exp_object : environnement des champs et méthodes de Object (dans EnvironnementExp)

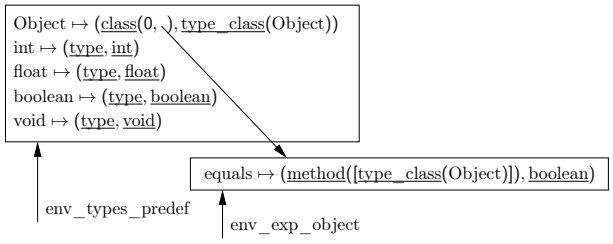
  env_exp_object $\triangleq$ {
  equals $\mapsto$ (method([type_class(Object)]), boolean) }

Implémentation des environnements

II-[SyntaxeContextuelle] section 10

- valeurs de EnvironmentExp comme listes chaînées de tables d'associations

  identificateur $\mapsto$ définition
- Environnements prédéfinis env_types_predef et env_exp_object.



Exemple

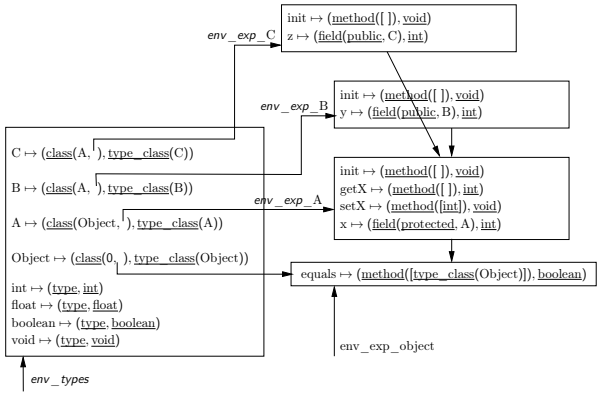
II-[SyntaxeContextuelle] section 10

```
class A {
   protected int x;
   void setX(int x) {
      this.x = x;
   }
   int getX() {
      return x;
   }
   void init() {
      x = 0;
   }
}
```

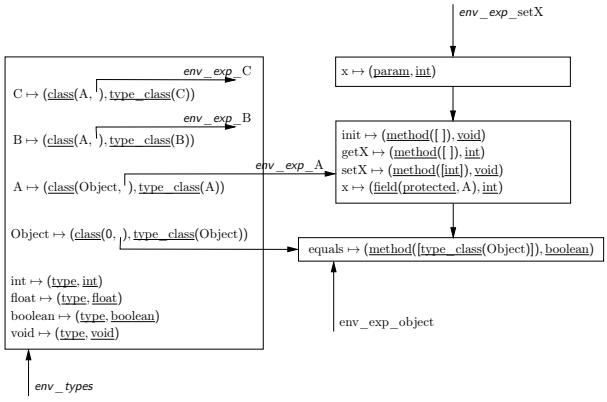
```
class B extends A {
   int y;
   void init() {
      setX(0);
      y = 0;
   }
}

class C extends A {
   int z;
   void init() {
      setX(0);
      z = 1;
   }
}
```


Environnement *env_exp* du corps des classes



Environnement d'analyse de la méthode *setX*



Conventions d'écriture dans la grammaire attribuée de Deca
Affectation des attributs

- Affectation explicite de la forme ***affectation*** $v := exp$.
  Règle (0.1)
  identifier $\downarrow env\_exp \uparrow def$
  $\rightarrow$ `Identifier` $\uparrow name$
  affectation $def := env\_exp(name)$
- Affectation implicite par expression fonctionnelle
  identifier $\downarrow env\_exp \uparrow env\_exp(name)$
  $\rightarrow$ `Identifier` $\uparrow name$

Conditions sur les attributs

- Clause condition
  Règle (3.28)
  rvalue $\downarrow env\_types \downarrow env\_exp \downarrow class \downarrow type_1$
  $\rightarrow$ **expr** $\downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type_2$
  condition assign_compatible($env\_types, type_1, type_2$)
- Affectation : toute valeur d'attribut doit être définie
  identifier $\downarrow env\_exp \uparrow def$
  $\rightarrow$ `Identifier` $\uparrow name$
  affectation $def := env\_exp(name)$
  contraint $env\_exp(name)$ à être défini.

Conditions sur les attributs

- Filtrage d'un attribut synthétisé en partie droite
  Règle (3.29)
  condition $\downarrow env\_types \downarrow env\_exp \downarrow class$
  $\rightarrow$ **expr** $\downarrow env_types \downarrow env_exp \downarrow class \uparrow$boolean
  impose que la valeur de l'attribut synthétisé de **expr** soit le type boolean.
- Filtrage d'un attribut hérité en partie gauche
  Règle (3.73)
  rvalue_star $\downarrow env\_types \downarrow env\_exp \downarrow$[ ]
  $\rightarrow \varepsilon$
  impose que la signature héritée en partie gauche soit la signature vide ([ ]).

Grammaires attribuées de la syntaxe contextuelle

- Vérifications contextuelles de Deca :
  trois passes sur le programme.
- Les règles contextuelles sont spécifiées à l'aide de trois grammaires attribuées.

Règles communes aux trois passes

- **identifier** $\downarrow env\_exp \uparrow def$ (0.1)
  $\rightarrow$ `Identifier` $\uparrow name$
  affectation $def := env\_exp(name)$

  On doit trouver une définition associée au nom *name* dans l'environnement *env_exp*.

  type $\downarrow env\_types \uparrow type$ (0.2)
  $\rightarrow$ `Identifier` $\uparrow name$
  condition $(\_, type) \triangleq env\_types(name)$

Passe 1

- Vérification du nom des classes et de la hiérarchie de classes
- Construction de l'environnement env_types, qui contient env_type_predef et les noms des différentes classes du programme
- **program** $\uparrow env\_types$ (1.1)
  $\rightarrow$ `Program`**[**
  list_decl_class $\downarrow env\_types\_predef \uparrow env\_types$
  MAIN]

Passe 1

- **list_decl_class** $\downarrow env\_types \uparrow env\_types_r$ (1.2)
  $\rightarrow$ **{**$env\_types_r := env\_types$**}**
  [(**decl_class** $\downarrow env\_types_r \uparrow env\_types_r$)* **]**

  À partir de l'environnement *env_types* hérité, on calcule l'environnement env_types_r résultant de la déclaration des classes.

  Si la liste de classes est vide, l'environnement résultant env_types_r est l'environnement *env_types* hérité.

Passe 1

- **decl_class** $\downarrow env\_types$ (1.3)
  $\uparrow \{name \mapsto (\underline{class}(super, \{\}), \underline{type\_class}(name))\} \oplus env\_types$
  $\rightarrow$ `DeclClass`**[**
  `Identifier` $\uparrow name$ `Identifier` $\uparrow super$
  LIST_DECL_FIELD LIST_DECL_METHOD
  ]
  condition $env\_types(super) = (\underline{class}(\_), \_)$
  - On récupère le nom de la super-classe *super*.
  - On vérifie que *super* fait partie de *env_types* et que c'est bien un nom de classe (condition).
  - L'environnement des types résultat est *env_types* auquel on ajoute la définition de la nouvelle classe.
  - Condition implicite (due à $\oplus$) : $env\_types(name)$ doit être non-défini.
  - Remarque : le profil de chaque classe est incomplet ; il ne contient que le nom de la super-classe mais pas l'environnement des champs et les méthodes (profil laissé vide {} et complété en passe 2).

Exercices

Question

Les programmes Deca suivants sont-ils corrects ?

1.
```
class A {
   B b;
}
```

2.
```
class A {
   void x;
}
```

3.
```
{ A a; }
```


Erreurs contextuelles

- Grammaires attribuées de Deca :
  spécification de la syntaxe contextuelle du langage
- Examen systématique de toutes les règles du langage :
  Identification de toutes les erreurs contextuelles possibles
- Contraintes potentielles
  - condition
  - filtrage d'un attribut hérité en partie gauche
  - filtrage d'un attribut synthétisé en partie droite
  - opération partielle

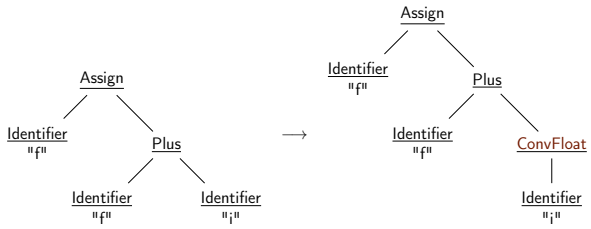
Examen systématique des règles

(0.1)
- raison : opération partielle
- message : « identificateur non déclaré »

Examen systématique des règles

Enrichissement de l'arbre abstrait

- cf. IV-[ArbreEnrichi]
- Ajout de Nœud ConvFloat
- Exemple : f = f + i ;
  avec les déclarations “float f ;” et “int i ;”



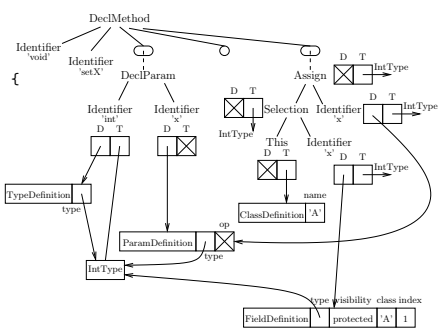
Décoration de l'arbre abstrait

- Décors : informations supplémentaires stockées dans l'arbre abstrait
- Permettent de faciliter la génération de code (étape C)
- Deux types de décors :
  - `Definition` : associés aux nœuds `Identifier`
    (Nature et type de l'identificateur)
  - `Type` : associés aux nœuds qui dérivent de EXPR dans la grammaire d'arbres
    (Type de l'expression)
- Remarque :
  - Il n'est pas nécessaire que les occurrences de déclaration d'un identificateur aient un `Type` associé.
  - Les occurrences d'utilisation d'un identificateur ont un `Type` associé en plus de sa `Definition`.

Décoration de l'arbre : Exemple

cf. IV-[Exemple] (section 2.3 : corps de la méthode setX)

```
void setX(int x) {
   this.x = x;
// this.x :
//   champ x
// x :
//   parametre x
}
```



Index des champs et des méthodes

- À chaque `Definition` de champ et de méthode est associé un index (numéro du champ ou de la méthode dans la classe).
- À chaque `Definition` de classe sont associés un nombre de champs (numberOfFields) et un nombre de méthodes (numberOfMethods).
- Ces informations doivent être mise à jour lors de l'étape B, en passe 2.

Index des champs et des méthodes

Exemple

```
class A {                    class B extends A {
   int x;                       int z;
   void f() {}                  void f() {}
   int y;                       void h() {}
   void g() {}                  int x;
}                            }
```

- Object :
  - 0 champ
  - 1 méthode : Object.equals (index 1)
- A :
  - 2 champs : A.x (index 1) A.y (index 2)
  - 3 méthodes : Object.equals (index 1) A.f (index 2) A.g (index 3)
- B :
  - 4 champs : A.x (index 1) A.y (index 2) B.z (index 3) B.x (index 4)
  - 4 méthodes : Object.equals (index 1) B.f (index 2) A.g (index 3) B.h (index 4)

Index des champs et des méthodes

Exemple

```
class A {                    class B extends A {
   int x;                       int z;
   void f() {}                  void f() {}
   int y;                       void h() {}
   void g() {}                  int x;
}                            }
```

Remarques

- Dans B, « void f() {} » redéfinit la méthode f
  - elle garde donc le même index que A.f
  - B a 4 méthodes et non 5
- Dans B « int x ; » déclare un *nouveau* champ B.x
  - B a 4 champs (A.x, A.y, B.z, B.x)
- Liaison dynamique sur les méthodes et non sur les champs.

Sémantique de partage

- Les `Definitions` sont partagées.
- Toutes les occurrences d'un même identificateur sont décorées avec la même `Definition`.
- cf. IV-[Exemple] (section 2.3)

Principaux répertoires concernés

cf. I-[Consignes]

- `src/main/java/fr/ensimag/deca/tree/`
  - Classes qui définissent les arbres
  - Méthodes de parcours de l'arbre abstrait
- `src/main/java/fr/ensimag/deca/context/`
  - Fichiers sources Java concernant l'étape B
- `src/test/java/fr/ensimag/deca/context/`
  - Fichiers Java de test concernant l'étape B
- `src/test/deca/context/`
  - Fichiers Deca de test

Classes fournies

cf. I-[Consignes]

- Classes pour les types :
  - Classe abstraite Type, dont dérivent les classes StringType, VoidType, BooleanType, IntType, FloatType, NullType et ClassType.

  Compléter le code des méthodes
  - boolean sameType(Type otherType) ;
  - boolean isSubClassOf(ClassType potentialSuperClass) ;
- Classes pour les définitions :
  - Classe abstraite Definition ;
  - Classes VariableDefinition, ParamDefinition, ClassDefinition, FieldDefinition, MethodDefinition...
- Classe Signature (pour la signature des méthodes)
- Classe EnvironmentExp (squelette fourni, à implémenter)
- Exception ContextualError, levée lorsqu'on détecte une erreur contextuelle (on s'arrête à la première erreur contextuelle).

Parcours de l'arbre abstrait

- Trois parcours à effectuer
- Parcours basés sur la syntaxe abstraite du langage II-[SyntaxeAbstraite]
- Une méthode abstraite verifyXyz pour chaque non terminal XYZ (ou classe AbstractXyz) de la grammaire d'arbres.
  - Dans la classe AbstractProgram :

    ```
    abstract void verifyProgram(DecacCompiler compiler)
       throws ContextualError;
    ```
  - Dans la classe AbstractMain :

    ```
    abstract void verifyMain(DecacCompiler compiler)
       throws ContextualError;
    ```
  - ...
- Ces méthodes abstraites sont ensuite implémentées dans les sous-classes (Program, EmptyMain, Main...)

Parcours de l'arbre abstrait

- Codage des attributs de la grammaire attribuée :
  - paramètres pour les attributs hérités ;
  - résultat de méthode pour un attribut synthétisé.
- Exceptions
  - env_exp : en général, dans la grammaire, on trouve un attribut hérité (pour la valeur « avant ») et un attribut synthétisé (pour la valeur « après » application de la règle). Dans l'implémentation, on utilise un seul objet de type EnvironmentExp, que l'on mute.
  - env_types : on peut stocker un environnement dans les objets de type DecacCompiler, de cette façon il n'est pas nécessaire de le passer en paramètres dans la plupart des méthodes.
  - On peut avoir besoin de paramètres supplémentaires pour les décorations.

Travail à effectuer pour l'étape B

cf. I-[Consignes]

- Compléter la classe EnvironmentExp.
- Classe de test de la classe EnvironmentExp
- Implantation des trois parcours de l'arbre
  - Implantation des méthodes verifyXyz dans les classes Java qui définissent l'arbre abstrait
  - Décoration et enrichissement de l'arbre
- On fournit le script `test_context`, qui appelle la classe `ManualTestContext` qui permet de tester l'analyse contextuelle.

Software Engineering: Basics

Projet GL

Ensimag
Grenoble INP

December 9, 2025

Software Engineering

Parallel with:

- Military engineering (oldest one): fortresses, siege & war machines
- Civil engineering: buildings, bridges
- Mechanical engineering
- Electrical engineering
- Chemical engineering...
- Software engineering & bio-engineering: the newest ones

Engineering: engineers are experts who

- master scientific and technical bases
- are able to design and guarantee quality
- organize the tasks and processes

Dual aspects of Software Engineering

Software engineering of *products*

- Tools and methods to make software *artefacts*
- Artefacts can be: source code, binary code, data structures or repositories, architecture diagrams etc.
- Example of tools: text editor (vim, Atom), IDE (Visual Studio), compiler, debugger, source code generator (ANTLR), model checkers (Avispa), link editor, configuration management (make, Maven), test harnesses, coverage analyzers (Jacoco), etc.

Software engineering of *processes*

- Methods to organize the production tasks
- In the case of software: mostly work of humans (developers)
- Backbone organization: known as (software development) "(life)cycles"
- Examples: Waterfall lifecycle, V lifecycle, Agile development

Ensimag Engineers

Engineers: are experts who

- master scientific and technical bases: **Computer science**
- are able to design and guarantee quality **Projet GL**
- organize the tasks and processes **SHEME, project Management**

What you will learn and practice (main skills)

- Software tools: git, mvn, antlr, log4j, IDE, gitlab, junit, jacoco...
- Languages & artefacts: Java, scripts, grammars, architecture diagrams, tests...
- Management methods: Agile, gantt planners, reporting...
- Working in a team, adapting to peoples' strengths and weaknesses

Ensimag Future Engineers

What you may start to discover (and learn further at Ensimag)

- Floating point computation
- Ecological impact of digital world
- Testing methodologies (more advanced than project)
- "Bowels" of computing: bytecode, binary (FP), link editing...
- ...

Quality Criteria for Software

- Criteria for the user
  - Reliable: Gives the expected result,
  - Robust: Doesn't crash, behaves reasonably in unexpected conditions,
  - Effective: Gives the result quickly,
  - Efficient: Uses a minimum of resources
  - User-friendly: Easy to use.
- Main focus for us: **Reliable**
- Secondary focus: Efficient (w.r.t. energy)

Quality Criteria for Software

- Criteria for the developer
  - Readable: Easy to read, to understand. Well documented,
  - Maintainable: Easy to modify, to fix,
  - Portable: Runs on various systems,
  - Extensible: Easy to improve,
  - Reusable: Can be adapted to other applications.
- Third focus for you: Readable

Software Lifecycle Stages

- Requirement analysis and definition
- Analysis and design
- Coding/Debugging
- Validation
- Evolution and Maintenance

Software Stages: Requirements

- Requirement analysis and definition
  - high level specifications
  - feasibility study

Projet GL:

- Decac compiler: specifications are ready (just read them)
- Extension: specs. are negociated with teachers

In real life, discussing requirements with customers is an important task (time consuming and critical).

Software Stages: Design

- Analysis and design
  - Detailed specification
    (for us, this is booklet part II)
  - Architecture
    (for us, 3 stages, Java packages, ...)
  - Detailed design (algorithms, data-structures)

Software Stages: Coding

- Coding: translating algorithms into programming language
- Debugging: compiling and exercising the code to check it

Beware: testing is NOT debugging

- Debugging done by developer to check whether the lines of code are actually written as he/she meant.
- Testing done by testing team to check whether program behaves as specified.

Software Stages: Validation

- Validation: make sure the program "works"
  - Static analysis and proof
  - Code review (very efficient)
  - Tests (essential)

Software Stages: Maintenance

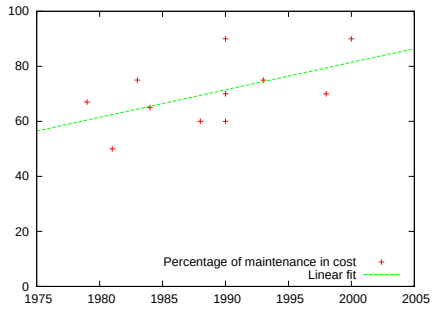
- Evolution and maintenance:
  - Corrective maintenance (Bug fixing)
  - Adaptive maintenance (Porting, ...)
  - Evolutionary maintenance (New features, ...)

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

Effort distribution

http://users.jyu.fi/~koskinen/smcosts.htm

- Part of Maintenance in Total Cost:



⇒ better optimize for maintainability than for initial development

Effort distribution in *Initial* Development

As a rule of thumb ...

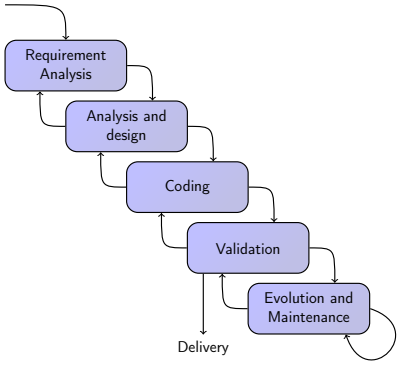
- Initial development:
  - Requirement analysis, architectural design: 40%
  - Coding, debugging: 20%
    - /! debugging is not part of validation.
    - Validation starts when the code looks correct.
  - Validation: 40%

And for our project

- Analysis, architectural design: 15% (reading, splitting into packages, ...)
- Detailed design, coding, debugging: 20%
- Validation: 40% (including scripting)
- Extension: 25%, of which 40% on analysis

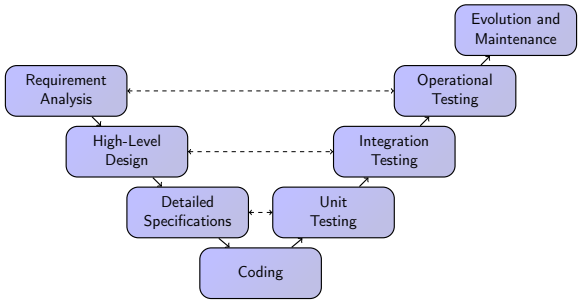
And time will also be used by management, synchronizing with team and with professors.

Waterfall Lifecycle - Historic



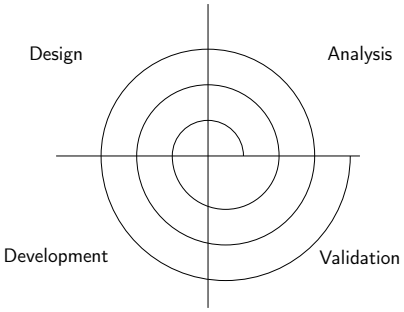
V Lifecycle - Large critical systems

Stresses Validation



Incremental (Spiral) Lifecycle

Typically in Agile development



Incremental Lifecycle

- Guiding principle: divide the program in small amounts of analyzed, coded, and tested features.
- Advantages:
  - Early discovery of problems,
  - Early availability of prototypes (essential to get feedback from the client),
  - Helps continuous validation,
  - Allows time-based releases, as opposed to feature-based releases.

Specifying the increment

- Informally
- With a set of rules in Deca's grammar
- With a set of tests

⇒ Test Driven Development

```
while true loop
    write tests
    make sure they don't pass
    implement feature
    debug until test pass
    commit and push
end loop
```


Examples of increment

- First goals:
  - Compile the empty program
  - Compile a hello-world
- Without objects:
  - Simple expressions (2+2, 2-2, ...)
  - Variables (int, float)
  - Control-structures (if/while)
- Objects:
  - Objects without methods
  - Methods (definitions and calls)

Planning should be driven by language subsets,
not by stage/passes (B1, B2, B3, C1, C2)

Incremental Lifecycle in our Project

- Hardly applicable to stage A (too short)
- Mandatory for stages B and C (type-checking and code generation).
- Avoids big-bang validation right before the final deadline
- Avoids half-done or untested features at the final deadline
- Necessary to get the intermediate delivery on time

⚠ prefer complete and well-tested compiler
on a subset of the language
to a buggy feature-complete compiler.

Validation

- Validation means making sure the program is
  - Correct
  - Robust
  - Efficient
  - Readable
  - Usable
  - Documented

Reminder: focus in our project is on Correctness (reliability).

Validation Techniques

- Use of static analysis tools
  (typing, coding style, absence of overflows, ...)
- Formal proof: costly, rarely used except for critical systems.
  Example: "meteor" subway in Paris, developed with "Atelier B",
  CompCERT C compiler proved in coq.
- Code review: one person reviews the code written by another, and checks the readability and correctness of the code.
- Test: run the program with different test-cases, and check that the results correspond to the expected results.

Testing

- Testing is the main validation technique.
- Objective: "show" that the program is correct, or find defects.
- Cannot "prove" the correctness, can indeed only exhibit defects.

Phases

- Test objective: select the feature to test
- Write test-cases
- Execute tests
- Observing, assessing and recording the result (oracle)
- Fix defects
- Evaluation: was the test sufficient?

Test objective and test cases

- Test objective: select the feature to test
- Examples:
  - Test stage A, test stage B (imprecise),
  - Test passe 2 of stage B,
  - Test type-checking of declarations,
  - Test rule 2.9.
- Select relevant data to accomplish the test objective.
- Exhaustive test usually impossible (infinite)

Execute tests and observe the result

- We execute the program with inputs and get outputs,
- Oracle: Verify that the output matches the expected output,
- Fix defects if some are found.

Evaluation: was the test sufficient?

- Is the test-suite sufficient? ...
- ... or shall we continue testing?
- How can we "measure" the effectiveness of a test-suite?
- ⇒ one answer is the notion of coverage (details follow).

Types of Tests: Overview

Unit tests Test small parts of the system,

Integration tests Check that the components work well together,

System tests Test the system under real conditions,

Acceptance tests Tests to run before any release (useful when the test-suite is not 100% automated),

Black-box tests Tests for an objective based on the *specification* of the program,

Glass-box tests Tests for an objective based on the *implementation* of the program,

(Non-)Regression tests Check that what *used to* work *still* works.

Example of program to test: factorial

```
public class Util {
    public static int fact(int n) {
        if (n == 0) return 1;
        else return n * fact(n - 1);
    }
}
```

```
import java.io.*;
public class FactMain {
    public static void main(String[] args) {
        BufferedReader stdin = new BufferedReader(
                new InputStreamReader(System.in));
        System.out.print("Enter a value: ");
        try {
            int v = Integer.parseInt(stdin.readLine());
            System.out.println("fact(v) = " + Util.fact(v));
        } catch (Exception e) {
            System.out.println("Input error");
        }
} }
```


Unit tests

« Tests unitaires »

- Test a small portion of code (one method, one class, ...)
- Example: test for the class `EnvironmentExp`
- Advantages:
  - Can be executed before building the whole system,
  - Finds errors more easily than testing the whole system,
  - Can test conditions hard to reach in normal executions,
  - Debugging unit-test is easy.
- Drawbacks:
  - Requires drivers to call the code under test (example: class `TestEnvironmentExp`),
  - May require stubs to replace the portions needed by the code under test.

Unit test for factorial

The manual way

```
class FactUnit {
    static void assertTrue(boolean c) {
        if (c) {
            System.out.println("ok");
        } else {
            throw new RuntimeException();
        }
    }
    public static void main(String[] args) {
        assertTrue(Util.fact(0) == 1);
        assertTrue(Util.fact(1) == 1);
        assertTrue(Util.fact(3) == 6);
    }
}
```


Unit test for factorial

Using JUnit (cf. III-[Tests])

```
import static org.junit.Assert.*;
import org.junit.Test;

public class FactTest {
    @Test
    public void testFact() {
        assertEquals(Util.fact(0), 1);
        assertEquals(Util.fact(1), 1);
        assertEquals(Util.fact(3), 6);
    }
}
```

- JUnit provides:
  - A library of assertions (`assertTrue`, `assertFalse`, `assertEquals...`),
  - A launcher that runs all methods decorated with `@Test` in classes named `Test...` or `...Test`,
  - Integration with Maven (`mvn test`), IDE (Right-click → Test file with Netbeans)...

Integration tests

« Tests d'intégration »

- Test for a set of methods, classes, or packages
- Examples: `test_synt`, `test_context`

Example system test for FactMain

```
#! /bin/sh

echo "Enter a value: fact(v) = 24" > expected
# we test both input/output and computation of fact(4).
echo 4 | java FactMain > actual

if ! diff expected actual; then
    # exit with error if expected and actual differ.
    exit 1
fi

# we should try "echo -1 | java FactMain" too ...
```


Black-box Tests

« Tests boîte noire »

- Black-box test = functional tests,
- Based on specifications of the program,
- Can, and should be written before coding,
- Preferably not written by the implementer of the code under test, otherwise
  - Ambiguities in the specifications are interpreted the same way,
  - Missing functionality will hardly be detected.
- Example: from the attribute grammar of Deca, one can
  - Identify the possible errors,
  - Write the list of error messages,
  - Prepare black-box tests for stage B,
  - Write part of the user manual.
  - before writing a single line of code!

Glass-box Tests

« Tests boîte transparente (ou blanche) »

- Glass-box test = structural tests
- Based on the implementation of the program.
- Goal: cover as much as possible of the program source code.
- Example:
  Dictionary implemented with a hash-table ⇒ test the colliding cases and the non-colliding ones.

Regression Tests

« Tests de non-regression »

- Re-execute the tests after each modification of the program,
- Check that the new result matches the old ones,
- Example: use "`diff old new`"

Code Coverage

« Couverture de code »

- Goal: "everything in the code must have been tested"
- What does it mean?
  - Each instruction has been executed?
  - Each variable took all the possible values?
  - ...?
- No perfect coverage metric in a finite world.

Statement coverage

« Couverture des instructions »

- Definition: a statement is covered when at least one test-case triggers its execution.
- Coverage ratio: number of statements covered/number of statements.
- Goal: cover 100% of the code
- (except dead code, as a result of defensive programming)

Branch Coverage

« Couverture des arcs »

- Definition: Branch = path from an instruction to the next.
- Example:

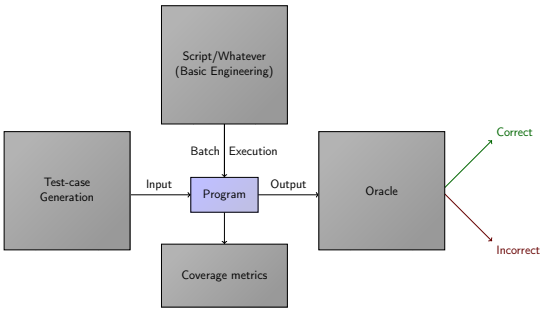
```
I1 ;
if (C1) {
  I2 ;
}
I3 ;
```

⇒ Instruction coverage achieved by one execution if C1 is true. Does not cover I1 → I3.

Jacoco: Statement Coverage Measure

- Compile the program with the right options (see III-[Jacoco]),
- Execute the test-suite,
- Jacoco tells which line of code has been executed, which hasn't.
- ⇒ essential to finish the validation
  or some lines of code have not even been tried!
- Add extra tests to increase coverage,
- However 100% usually not reachable (dead code, esp. with defensive programming)

General Considerations on Testing



Execution automation

- Minimal launcher:

```
#!/bin/sh
for i in *.deca
do
    echo "$i"
    # replace <executable> with test_synt or
    # test_lex or test_context or decac
    <executable> "$i"
done
```


Oracle: Checking the Result

- Automatic oracle essential:
  - Manual checking of output is boring and error-prone
  - Regression testing almost impossible without automatic oracle.
- Many ways to manage oracles:
  - Manual validation the first time, diff the next times,
  - Comparison of two implementations,
  - Assertions, defensive programming,
  - Approximation (example: casting out 9).

Regression and Efficiency

Regression testing is heavily used in s/w industry, esp. with Continuous Integration (every commit triggers tests to avoid a regression).
BUT automated execution of large test suites takes a lot of CPU and energy.

- Automated regression testing is necessary to ensure Reliability
- Requires smart scripts to focus on impacted parts.

⇒ You will have to manage conflicting goals:

- Ensure highest reliability (primary goal)
- While being conservative about energy (scondary goal)

Your approach to solving this conflict will be reported (and graded) in your report on energy.

Mandatory Conventions for our Project

- Directories:
  - Deca tests must be in sub-directories of `src/test/deca/syntax`, `src/test/deca/context` and `src/test/deca/codegen`.
  - Each directory must have
    - ⋆ `valid/`: Deca program correct with respect to the current stage.
    - ⋆ `invalid/`: Deca program triggering a compiler error in the current stage.
  - `src/test/deca/codegen` has in addition:
    - ⋆ `interactive/`: all interactive tests.
      `valid/` and `invalid/` must not contain any `read` instruction.
    - ⋆ `perf/`: performance tests, to assess the number of ima cycles used executing them. They should all be valid programs.

Mandatory Conventions for our Project

cf. III-[Tests]

- File name extensions:
  - .deca: Deca source files,
  - .ass: Generated (archived) assembly files.

Mandatory Conventions for our Project

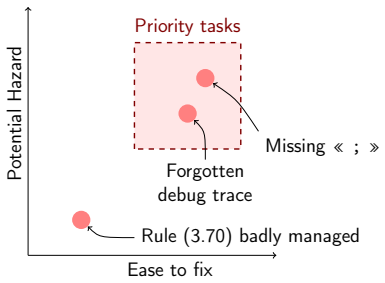
- Automation: `mvn test`
- Test-suite must be automated as much as possible (scripts).
  - Scripts must exit with 0 if the test succeeds, with another value (exit 1) if the test fails.
  - Add test scripts in pom.xml file (cf III-[Tests], section 1.6).
  - Example scripts are provided, but are minimalistic.
- Must be non-interactive by default (both for success and failures)
- (for info: the teacher's test infrastructure is >2000 lines of shell-script).
- `/bin/sh` is the suggested language for test automation (perl, python are other good candidates).
- Resources for shell-scripts available on EnsiWiki.

Test Suite in the Software Engineering Project

- Test suite is an important part of the grade.
  - It weighs almost the same as the compiler
  - If your compiler is perfect, but tests are absent, you get half the points.
  - And you will get the same if you do not write a single line of code for decac but have excellent tests
  - $\Rightarrow$ share your efforts accordingly.
  - You can and should write tests before writing a single line of code.
- Grading takes into account:
  - Coverage of the test-suite
  - Test-case layout (conventions above)
  - Automation

See `III-[Tests]`

Risks management: cost-benefit



Risks management & release process

- Risk assessment and control:

Document	Danger	Action
`I-[Introduction]`	Miss a deadline	Use an agenda
`IV-[Example]`	Fail on provided example	Test it!
`II-[Decac]`	...	...
...		

- Release Process
  - Checklist of actions to perform before a release
  - Should prevent all major risks (i.e. compiler unusable with respect to the teachers' testsuite, grossly mis-classified testsuite...)

See `III-[Tests]`, section 2.

Roles

- Designer
- Developer
- Reviewer
- Tester
- Documentation writer
- Scrum master etc.

Tips (see resources on Project Management on Chamilo):

- Ideally, try all the possible roles.
- Practically, in this project, more efficient to choose depending on your individual skills.
- However, a big task should never be assigned to a single person, as this single person can fail (lack of skills, health problem...)
- At any time, the team should be able to re-assign tasks quickly.

Provisional Schedule

« Planning prévisionnel »

- Make a provisional schedule at the beginning of the project.
- Use "`planner Planning.planner`"[1] in `Projet_GL/planning` to modify your schedule
- Specify your increments and distribute the tasks between the members of the team.
- (You may use an alternative to Gantt charts, e.g. burndown chart, for day-to-day planning)

[1] or your favorite alternative, as long as it can create PDF files

Actual Schedule

« Planning effectif »

- Make the actual schedule of your daily work.
- You must explain the differences between your estimated and effective schedules (evaluation takes into account your explanations, not the differences themselves)
- Use "`planner Realisation.planner`" in `Projet_GL/planning` to modify your schedule.

Typical Efforts for Projet GL

According to parts of project

- Stage A lex+synt: 10%
- Stage B ctx verif: 20%
- Stage C gencode: 45%
- Extension: 25%

According to type of activity (see Rule of thumb, adapted for Projet GL)

- Analysis & Design: 25% (mostly for stage C and extension)
- Coding & debugging: 20%
- Validation (reviews & tests): 35%
- Documentation & Management (incl. lectures) : 20%

Activity Report

You are expected to make an activity report for each "progress meeting". Specify:

- what has been done since the last "progress meeting",
- the current differences between your effective and estimated schedules.

We advise you to keep a detailed count of how many hours were spent on each task (including tasks such as meetings, preparations etc): this will help you for the final report ("bilan"), and also for your own feedback and planning.

Progress Meetings in our Project

Reminders

- `I-[Suivis]`
- 3 meetings, 30 minutes each
- 20 minutes "progress report"
  - You convince the teacher that the progress is good,
  - Must be prepared.
- 10 minutes "technical support"
  - ⇒ The teacher can help you.
- first two meetings with a "SHEME" teacher.

First Progress Meeting

- See `I-[Suivi-SHEME1]`
- Prepare a short document presenting your team and your organization,
- Prepare a provisional schedule (See `III-[GuidePlanner]`).
- Present a proposition of an extension (2 pages)
  - analysis
  - draft specification of the extension

Execution Traces

- Traces can be useful to debug a program
- *must* be easy to remove
  ⇒ **never** use `println` for debugging.
- Implementation (the manual way - prefer log4j):

```
class TraceDebug {
    private static final int LEVEL = 5;
    public static void trace(int level,
                             String message) {
        if (level <= LEVEL) {
            System.out.println("trace: " + message);
        }
    }
}
```

Usage: `TraceDebug.trace(4, "Message");`

Log4j library

cf. III-[ConventionsCodage]

- ```
  import org.apache.log4j.Logger;
  public class LogClass {
      // Instantiation of logger, done once for each class.
      private static final Logger LOG =
          Logger.getLogger(LogClass.class);
      // ...
      LOG.trace("Trace Message!");
      LOG.debug("Debug Message!");
      LOG.info("Info Message!");
      LOG.warn("Warn Message!");
      LOG.error("Error Message!");
      LOG.fatal("Fatal Message!");
  ```
- To choose the level:
  - method `setLevel` of each logger,
  - configuration file `log4j.properties`
    (in `src/test/resources/` and `src/main/resources/`).
- `warn` level ⇒ messages corresp. to `warn`, `error` and `fatal` displayed.

Good Practices and Coding Style

- Keep methods short (≈ 1 screen)
- Do not write long lines (80 characters max)
- Indent consistently with 4 spaces (if using tabs, 1 tab = 8 spaces)
  ⚠ Not the default with Eclipse :-(
- Class names start with an uppercase letter, method and variable names start with a lowercase letter
- Comment your code to explain *why* the code is how it is, not *what* it does
- Comment your method headers (javadoc) to explain what methods are doing (pre/post conditions, ...)
- cf. III-[ConventionsCodage], section 3

Defensive Programming

- See `III-[ProgrammationDefensive]`
- Method preconditions: conditions that the method arguments must satisfy:
  - partial functions,
  - conditions that the arguments must satisfied, so that the algorithm works correctly.
    Example: dichotomic search in a sorted array.
- Method postconditions: conditions that must be satisfied after a method call.
- Invariant: condition that is always satisfied
  (loop invariant, class invariant)
  ```

Defensive Programming

- Defensive programming: explicit check of preconditions, postconditions and invariants.
- Allows the programmer to detect and correct bugs at a lower cost.
- When an assertion is violated, the program is stopped by raising an exception.

Checking Preconditions

- Use of the class `Validate` from apache commons
- Example:

```
import org.apache.commons.lang.Validate;
// ...
/**
 * precondition : x >= 0
 */
float sqrt(float x) {
    Validate.isTrue(x >= 0 ,
                    "x should be positive");
    // ...
}
```

- Methods: `isTrue`, `isFalse`, `notNull`, `notEmpty`.

Checking postconditions and invariants

- Use of assertions
- Assertions are enabled during development, and disabled during final testing and release.
- In Java: assertions are disabled by default, enable with `java -enableassertions`
- Syntax:
  `assert` *`condition`*;
- Violating an assertion raises the `AssertionError` exception (deriving from `Error`).

Checked and unchecked exceptions

Two types of exceptions in Java:

- unchecked exceptions
  - derive from `RuntimeException` or `Error`;
  - are the result of a programming problem (e.g. `NullPointerException`), or other unrecoverable error (e.g. `OutOfMemoryError`);
  - should not be caught.
  - `Validate` and `assert` raise unchecked exceptions.
- checked exceptions
  - are part of the specification of the method (`throws` clause);
  - must be caught by the caller.

Méthodes agiles : Eléments de la méthode Scrum

Projet GL

Ensimag
Grenoble INP

9 décembre 2025

Introduction

Scrum : méthode agile

- cycle de vie itératif
- succession d'itérations (appelées *sprints*)
- ensemble d'incréments (*user stories* ou scénarios utilisateur)
  ⇒ petit nombre de fonctionnalités vues du point de vue de l'utilisateur, pour lesquelles on réalise :
  - analyse
  - conception
  - implémentation
  - validation
- livraison (démo) à la fin de chaque sprint

Rôles

Trois rôles distincts en Scrum :

- *Product Owner* : représentant du client (ou client sur site)
  - Dans le cadre du projet GL : l'enseignant joue partiellement ce rôle.
- *Scrum Master* : personne qui s'assure de la bonne application de la méthode Scrum
  - Dans le cadre du projet GL : un des membres de l'équipe
- Membre de l'équipe de développement

Scénarios utilisateur
User stories

- Le travail est organisé sous forme de scénarios utilisateur
  - Fonctionalités voulues par le client, en utilisant la terminologie du client
  - À chaque scénario est associé une priorité, qui détermine l'ordre dans le quel chaque scénario sera developpé.
  - À chaque scénario sont associés un certain nombre de tests qui montrent que le scénario est bien implémenté.
- Liste des scénarios utilisateur = *product backlog*
- Il peut être utile d'avoir en plus des scénarios techniques (*technical stories*).

Planification d'une itération
Sprint planning

- Horaire pour la mêlée quotidienne
- Définir le but du sprint
- Définir les scénarios qui font partie du sprint (*sprint backlog*)
- Définir pour chaque scénario :
  - portée
  - importance
  - estimation du coût (en nombre de « points »)
- Découper chaque scénario en tâches, en attribuant à chaque tâche un nombre de points
- Prévoir une date et un horaire pour la démonstation (en fin de sprint)
- Estimation des coûts : par l'équipe en utilisant par exemple le jeu du Poker (*planning poker*)

Tableau des tâches pour un sprint

User stories	Tâches à faire	Tâches en cours	Tâches finies
US1(24)	T1(2) T2(7) T3(5) T4(10)		
US2(10)			
US3(20)			
US4(24)			

On doit définir ce que signifie qu'une tâche est « finie ».

Mêlée quotidienne
Daily Scrum

- Une réunion tous les jours (matin) à la même heure
- Courte durée (15 min)
- On reste debout (*Stand-up meeting*)
- Chaque membre de l'équipe décrit ce qu'il a fait la veille et ce qu'il fera le jour même.
- Mise à jour du tableau des tâches

Plannings prévisionnel et effectif

Méthode agile
⇒ Pas de planning prévisionnel détaillé des tâches en début de projet

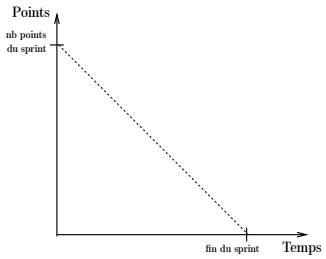
Planning prévisionnel comportant :

- les différents sprints
  - la durée de chaque sprint est fixe, on ne "rallonge" pas les sprints en cas de retard
- les rendus
- les suivis

Planning effectif

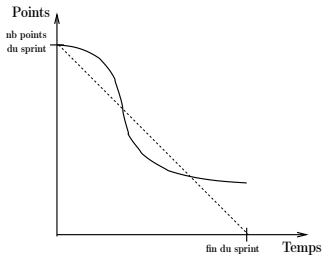
- Maintenir une trace des activités
- Qui fait quoi, sur quelle durée ?
- Estimer la vitesse de développement de l'équipe
- Bilan en fin de projet

Calcul de la vitesse
Burndown chart



Estimation de la vitesse de développement

Calcul de la vitesse
Burndown chart



Vitesse effective de développement

Fin de sprint

Démonstration au client
- Prise en compte des retours du client

Calcul de la vitesse de développement
- Permet de mieux prévoir la quantité de travail possible pour le sprint suivant

Rétrospective
- Identifier :
  - ce qui s'est bien passé
  - ce qui aurait pu être mieux
  - ce qui aurait pu être fait différemment
- Identifier des actions concrètes pour que le sprint suivant se passe mieux

Qualité

- En cas de retard :
  - On ne cherche pas à développer l'ensemble des fonctionnalités attendues.
  - On préfère un logiciel incomplet mais fiable, plutôt qu'un logiciel complet mais complètement bogué.
- « On ne sacrifie pas la qualité ».

Projet Génie Logiciel

Étape C

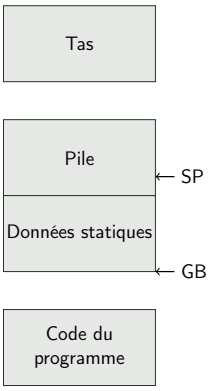
Génération de code

Projet GL

Ensimag
Grenoble INP

9 décembre 2025

Organisation de la mémoire à l'exécution dans IMA



données statiques variables globales, littéraux du programme, tables des méthodes, et valeurs temporaires

pile variables locales, paramètres de méthodes

tas objets alloués dynamiquement (pas utilisé dans le langage sans-objet)

GB (« Global Base ») Registre particulier qui pointe sur le fond de pile

SP (« Stack Pointer ») Pointe sur la dernière case occupée de la pile

Analogie/Différences avec l'assembleur x86 (Pentium)

- zone statique IMA = sections `.data`, `.rodata` et `.bss` en ELF x86.
- SP ≈ %esp (%rsp en 64 bits)
- GB n'a pas d'équivalent en x86

Principes de la génération de code pour expressions

- Évaluation de gauche à droite (sémantique de Deca)
- Calcul du résultat dans un registre banalisé $\geq$ R2, en utilisant d'autres registres pour sauvegarde des résultats de sous-expressions.
- Registres R0 et R1 = registres scratch (modifiables par les appels de méthodes), pas utilisés pour sauvegarder le résultat de sous-expressions.
- **RMAX** autorisé : `X-1` si option `-r X`, ou 15 sinon.
- Si plus de registre disponible, utiliser `PUSH` et `POP` pour sauvegarde des résultats sur la pile (c-à-d. des "temporaires").
  NB : attention à prise en compte dans calcul du `TSTO` (cf. plus loin).

Exemple de l'affectation

```
<codeExp( Assign[ Ident↑symb  e] , n)>
  := <codeExp( e,n)> // calcul de e dans Rn (avec n ∈ 2..MAX)
     STORE Rn, @symb
```


Génération de code naïve pour expressions arithmétiques

Opérande d'une expression atomique

```
<dval( IntLiteral↑n)> := #n
<dval( Identifier↑symb)> := @symb
<dval( _[ _ _] )> := ⊥
```

Mnémonique d'un opérateur binaire

```
<mnemo( Plus)> := ADD
<mnemo( Minus)> := SUB
<mnemo( Mult)> := MUL
```

Code pour calculer *e* dans R*n* (utilisant uniquement R0 et R*n* ... RMAX)

```
<codeExp( e,  n)>
avec <dval( e)>≠⊥
  := LOAD <dval( e)>, Rn

<codeExp( op[ e1  e2] , n)>
avec <dval( e2)>≠⊥
  := <codeExp( e1 , n)>
     <mnemo( op)> <dval( e2)>, Rn
```

```
<codeExp( op[ e1  e2] , n)>
avec <dval( e2)>=⊥ et n=MAX
  := <codeExp( e1 , n)>
     PUSH Rn ; sauvegarde
     <codeExp( e2 , n)>
     LOAD Rn, R0
     POP Rn  ; restauration
     <mnemo( op)> R0, Rn
```

```
<codeExp( op[ e1  e2] , n)> avec <dval( e2)>=⊥ et n<MAX
  := <codeExp( e1 , n)>
     <codeExp( e2 , n+1)>
     <mnemo( op)> Rn+1, Rn
```


Exercice 1 : évaluation d'expression

On considère le programme suivant :

```
{
    int x = 1;
    int y = 2;
    int z;
    z = 2 * x - 3 * y;
}
```

Question

Dessiner la pile

Question

Écrire l'assembleur généré

Documentation

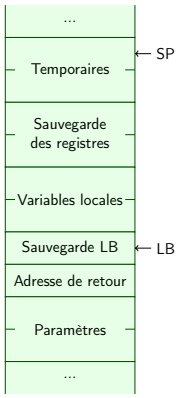
- Sémantique de Deca : II-[Semantique]
- Machine abstraite : II-[MachineAbstraite]
- Conventions de liaison : II-[ConventionsLiaison]
- Algorithmes de génération de code : IV-[Gencode]
- Outil fourni : `ima` (interprète de la machine abstraite), plus un metteur au point (voir IV-[Ima])
- À faire : I-[Consignes] (étape C)

Algorithmes et implantation de la génération de code

IV-[Gencode]

- Génération de code en deux passes :
  - Passe 1 : construction de la table des méthodes de chaque classe
    - Pas utile pour le langage sans-objet.
    - cf. IV-[Exemple]
  - Passe 2 : génération de code pour
    - le programme principal
    - chaque classe : initialisation des objets et codage des méthodes
    - messages d'erreurs pour les erreurs à l'exécution
- Parcours d'arbre en utilisant le patron « interprète », basé sur la grammaire d'arbres.

Bloc d'activation d'un appel de procédure



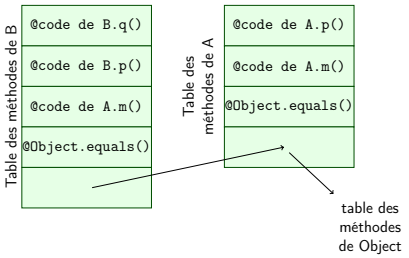
SP Pointeur de Pile, qui pointe sur le sommet de pile. (analogie avec l'assembleur x86 : SP ≈ %esp, ou %rip en 64 bits)

LB base locale, qui permet d'accéder aux paramètres, variables locales et temporaires dans la pile. (analogie avec l'assembleur x86 : LB ≈ %ebp, ou %rbp en 64 bits)

Table des méthodes

```
class A {
   int x;
   int y;
   void m() { }
   void p() { }
}

class B extends A {
   int x;
   int z;
   void p() { }
   void q() { }
}
```



Exercice 2 : programme d'exemple

```
class Point2D {
   int x; // Abscisse
   int y; // Ordonnee
   // Deplace ce point
   // de a en diagonale.
   void diag(int a) {
      x = x + a;
      y = y + a;
   }
}

class Point3D extends Point2D {
   int z; // Hauteur
   // On redefinit la methode diag,
   // pour tenir compte de z
   void diag(int a) {
      x = x + a;
      y = y + a;
      z = z + a;
   }
}
```

```
{
   Point2D p1, p2;
   Point3D p3;

   p1 = new Point2D();
   p1.diag(1);

   p3 = new Point3D();
   p3.diag(2);

   p2 = p3;
   p2.diag(3);

   println("p3.z = ", p3.z);
}
```

a. Qu'affiche le programme ?
b. Dessiner l'état de la pile et du tas
c. Ecrire le code généré pour ce programme Deca

Exercice 2(a) : Sémantique du programme

```
{
   Point2D p1, p2;
   Point3D p3;

   p1 = new Point2D(); // p1 initialise a zero :
                       // p1.x = 0; p1.y = 0
   p1.diag(1);         // p1.x = 1; p1.y = 1;

   p3 = new Point3D(); // p3 initialise a zero :
                       // p3.x = 0; p3.y = 0; p3.z = 0
   p3.diag(2);         // p3.x = 2; p3.y = 2; p3.z = 2

   p2 = p3;    // p2 et p3 representent le meme point
   p2.diag(3); // Liaison dynamique : Appel de Point3D.diag
               // p2.x = 5; p2.y = 5; p2.z = 5
   println("p3.z = ", p3.z); // p3.z = 5
}
```


Problème du TSTO

Lignes à ajouter aux extraits de code ci-dessus pour tester les débordements de pile :

```
code.Point2D.diag :
   TSTO #2
   BOV pile_pleine

code.Point3D.diag :
   TSTO #2
   BOV pile_pleine

init.Point3D :
   ; pour pouvoir appeler
   ; init.Point2D:
   TSTO #3
   BOV pile_pleine
```

```
; construction de la
; table des méthodes
; + programme principal
TSTO #15 ; (11 pour le
         ;     "ADDSP #11"
         ; + 4 pour le programme
         ;     principal)
BOV pile_pleine
```


Récapitulatif des erreurs à l'exécution

II [Semantique]

- erreurs de débordement :
  - pile
  - tas
  - arithmétique (sur flottants, inclus la division par 0.0)
- division entière par 0 et reste de la division entière par 0
- sortie de méthode sans passer par "return"
- conversion de type impossible
- déréférencement de null
- erreur de lecture (`RINT` et on ne tape pas un entier ; `RFLOAT` et on ne tape pas un flottant)
- accès à des variables non initialisées (non traité : si on y accède `ima` donne un message d'erreur).

Les calculs sur les flottants

Projet GL

Ensimag
Grenoble INP

9 décembre 2025

Pourquoi s'intéresser aux flottants ?

- Un objectif fort du projet GL est de comprendre comment les machines traduisent nos algorithmes
- Les flottants ne sont qu'un type, mais très utilisé dans toutes sortes d'applications
  - Codage (audio et vidéo), jeux vidéos, graphismes
  - Localisation, planification de trajets
  - Contrôle de processus, analyse de capteurs, logiciels embarqués,
  - Calcul scientifique, etc.
- Le calcul flottant est plein de difficultés et de bizarreries
- Besoin évident pour l'extension TRIGO, mais aussi TAB, OPTIM etc.

Pour le développement durable, il est important de réduire le calcul flottant qui représente un élément important de la consommation d'énergie

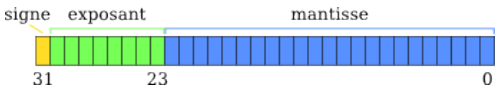
Un projet digne des Ensimag

Ensimag = Maths + Info

- Héritage de Jean Kuntzmann : l'informatique grenobloise fondée sur le calcul numérique
- Témoignage d'industriels : *"Un Ensimag, c'est un ingénieur qui sait que 1 plus 1 ne fait pas forcément 2"*
  - 1.1 + 1.1 n'est PAS égal à 2.2
- Différence infime ? Mais la conséquence peut être "catastrophique"
- Exemple de test incorrect :

```
float x, y = 1.1; float z = 2.2;
if (x + y == z) {
  ...
```


Les flottants Deca



- 1.0
  - Signe : $s = 0$
  - Exposant+127 : $e = 0111\ 1111$
  - Mantisse : $m = 00000000000000000000000$
  - $1.0 = (-1)^s \times 2^{e-127} \times (1 + m \times 2^{-23})$
- Précision sur 1.0 : dernier bit significatif
  $ulp(1.0) = 2^{-23} \approx 1.19209 \times 10^{-7}$
- Mais attention : le flottant juste en dessous de 1.0 vaut $x = 1 - 2^{-24}$
  $ulp(x) = 2^{-24}$

Répartition des flottants parmi les réels

- Tous les flottants sont des rationnels $\Rightarrow \pi$ n'est pas un flottant.
- >40% des flottants sont des entiers : tous ceux supérieurs (en valeur absolue) à 2^{23}
- >40% des flottants sont inférieurs (en valeur absolue) à 2^{-23}
- Il n'y a que 18% entre ces deux "extrêmes" 2^{-23} et 2^{23} ($\approx 8.4E6$)
- Densité très variable
  - Très forte autour de 0 (jusqu'à 2^{-149})
  - Et dans les grands entiers (eux-mêmes très espacés)

Importance de raisonner en binaire

- Seule la représentation binaire des flottants est exacte
  - Et l'affichage par ima en décimal est tronqué.
- La précision de vos résultats doit s'exprimer selon ulp
- ulp = Unit of Least Precision
  - La précision relative est 2^{-23}
  - ulp = précision absolue
- Exemples
  - $ulp(1) = 2^{-23}$
  - $ulp(2^{-23}) = 2^{-46}$
  - $ulp(10,000,000) = 1 \Rightarrow$ le flottant suivant est l'entier $10,000,001$
  - $ulp(2^{30}) = 128$: autour de 1 milliard, l'écart entre flottants successifs est de l'ordre de la centaine. Et on est encore très loin des plus grands flottants représentables (il en reste encore 38%).

Exemple $sin(\pi)$

- π n'est pas rationnel, donc n'est pas un flottant. Or vos fonctions trigonométriques ne peuvent être appliquées qu'à des arguments flottants.
- Soit
  - π^- le flottant immédiatement inférieur à π.
  - $\Delta = \pi - \pi^-$.
- On a
  - $\Delta < 2^{-22} = ulp(\pi^-)$ (car $2 < \pi < 4$ donc exposant 1)
  - $sin(\pi^-) < 2^{-22}$ (car $sin(\pi - \Delta) = sin(\Delta) \leq \Delta$)
  - Votre calcul devrait être précis :
    - ★ à 2^{-23} en valeur RELATIVE si possible,
    - ★ donc à 2^{-45} (puisque $2^{-23}\Delta < 2^{-45}$).

Consignes pour l'extension TRIGO

- Votre bibliothèque sera testée séparément par nos tests : ne vous basez pas sur des particularités de votre compilateur.
- *Vos tests* ne doivent pas utiliser de particularités de votre bibliothèque, car ils seront testés avec une bibliothèque standard.
- Respectez bien les domaines et codomaines des fonctions trigonométriques.
- Documentation attendue pour la classe Math : 10 à 20 pages décrivant vos *choix d'algorithmes*, analysant leur *précision* (cf vos cours de Méthodes Numériques), décrivant l'implantation et la validation.

Recommandations

Extension :

- $\approx$ 20 à 25% de l'effort du projet... Et 20% de la note :
  - $\Rightarrow$ presqu'autant que le compilateur (25%) et que les tests (22,5%)
- TRIGO : Spécifiée, mais pas guidée ; analyse et conception à faire intégralement.

Organisation

- Commencer dès le début du projet.
- Choix d'algorithmes adaptés au contexte (flottants 32 bits, capacités IMA...)
- Gros travail en phase amont : algorithmes, analyse de convergence et précision etc. Vous ne pouvez pas attendre que votre compilateur compile du code objet (rush de la dernière semaine).
- Impliquer plusieurs étudiants (comme la plupart des autres tâches) : validation croisée etc.