Manuel Utilisateur

Projet Génie Logiciel — ENSIMAG

Auteur : Faical EL GOUIJ

Équipe : Groupe 8, équipe 43
Members :
Fatima Azzahra Ardan,
Mohammed EL ARABI,
Faical EL GOUIJ,
Hamza MOUNTASSIR,
Ayoub TOUATI

Tuteur : M. NICOLLIN Xavier

Date : 19 Janvier 2026

1 Introduction

Ce manuel utilisateur présente l'utilisation du compilateur Deca réalisé dans le cadre du projet de génie logiciel. Il s'adresse à un utilisateur souhaitant compiler et exécuter des programmes écrits en langage Deca, sans entrer dans les détails de l'implémentation interne du compilateur.

Le document décrit le périmètre du langage effectivement supporté, ainsi que l'utilisation de l'outil decac en ligne de commande, incluant les options disponibles et les conventions relatives aux fichiers d'entrée et de sortie. Les messages d'erreur ne sont évoqués qu'à titre indicatif, sans chercher à fournir une description exhaustive de leur fonctionnement interne, aussi que les différents limitations que nous avonz rencontré.

Enfin, Le manuel présente l'extension réalisée dans le cadre du projet, en expliquant la procédure de compilation des fichiers fournis, avec ou sans activation des optimisations. Celle-ci est expliquée néanmoins pour les différents scripts et commandes qu'on traitera dans ce Manuel.

2 Périmètre du langage supporté

Le compilateur Deca couvre les différents sous-ensembles du langage définis dans le cadre du projet, dans la mesure où ceux-ci ont été validés par les tests réalisés. Les programmes relevant du sous-ensemble hello-world sont supportés, incluant les constructions élémentaires du langage ainsi que la génération et l'exécution du code correspondant sur la machine abstraite IMA.

Le sous-ensemble sans-objet est également pris en charge. Il comprend notamment les types primitifs, les expressions arithmétiques et booléennes, les structures de contrôle, ainsi que les déclarations et affectations de variables, conformément aux spécifications du langage Deca pour ce sous-ensemble. Ces fonctionnalités ont été testées et validées dans le cadre du projet.

Enfin, le compilateur supporte le langage complet avec objets. Les mécanismes de définition de classes, d'héritage, de méthodes et d'accès aux champs sont implémentés. Les fonctionnalités liées au typage dynamique, telles que les opérations de transtypage (cast) et les tests de type (instanceof), sont également disponibles. Leur bon fonctionnement correspond aux cas couverts par la campagne de tests réalisée.

Ainsi, le compilateur essentiel comme le compilateur complet permet la compilation et l'exécution des programmes relevant des sous-ensembles du langage effectivement couverts par les tests.

3 Utilisation du compilateur

3.1 Commande de base

Le compilateur `decac` peut être exécuté de différentes manières selon le type de traitement souhaité : compilation complète ou tests d'analyse lexical, syntaxique ou contextuelle.

- **Compilation complète :**
  - **Commande :**

```
./src/main/bin/decac fichier.deca
```

  - **Comportement :** compilation complète du fichier source.
  - **Fichiers générés :** fichier assembleur **.ass** si le programme est valide, placé dans le même répertoire que le fichier source.

- **Compilation et exécution IMA :**
  - **Commande :**

```
./src/test/script/launchers/test_gencode fichier.deca
```

  - **Cas d'échec inattendu :**

```
######### ATTENDU (.expected) après Normalisation ##########
42
############ OUTPUT (.out) après Normalisation ############
45
```

  - **Comportement :** compilation complète du fichier source, exécution à l'aide de IMA et comparaison avec un fichier .expected donné, dans le cas d'un test **invalid/** le .expected contient l'erreur qui doit être générée, et dans le cas **valid/** ou **perf/** le résultat attendu.
  - **Fichiers générés :** fichier assembleur **.ass** et fichier **.out** si le programme est valide (si ce n'est qu'à l'exécution qu'une erreur est levée le fichier `.ass` est généré quand même), placé dans le même répertoire que le fichier source.

- **Analyse lexicale :**
  - **Commande :**

```
./src/test/script/launchers/test_lex fichier.deca
```

  - **Exemple de cas d'échec :**

```
eca:2:16: token recognition error at: '@'
```

  - **Comportement :** affiche la liste des tokens reconnus par l'analyseur lexical.
  - **Fichiers générés :**aucun.

- **Analyse syntaxique :**
  - **Commande :**

```
./src/test/script/launchers/test_synt fichier.deca
```

  - **Exemple de cas d'échec :**

```
src/test/deca/syntax/invalid/AvecObjet/class_imbriquee.deca:10:3: missing '}'
    at 'class'
```

  - **Comportement :** produit l'arbre syntaxique abstrait non décoré si le programme est syntaxiquement correct.
  - **Fichiers générés :** aucun n'est produit.

- **Analyse contextuelle :**
  - **Commande :**

```
./src/test/script/launchers/test_context fichier.deca
```

– **Cas d'échec :**

```
DEBUG fr.ensimag.deca.tree.Program.verifyProgram(Program.java:47) - verify
    program: start
.....
DEBUG fr.ensimag.deca.tree.Main.verifyMain(Main.java:37) - verify Main: start
DEBUG fr.ensimag.deca.tree.Main.verifyMain(Main.java:41) - verify Main: end
src/test/deca/context/invalid/assign/id_noy_expr.deca:12:4: Identifier A is
    not defined in the current scope nor in its parent scopes
```

– **Comportement :** produit l'arbre syntaxique abstrait décoré si le programme est valide du point de vue sémantique.

– **Fichiers générés :** aucun n'est produit.

- **Gestion des erreurs :**
  – Si une erreur est détectée dans n'importe quel mode, le compilateur interrompt le traitement.
  – Un message d'erreur adapté est affiché selon le type d'analyse (lexicale, syntaxique ou contextuelle) et indique la partie du code source concernée.

3.2 Options disponibles

Règle générale pour utiliser les options:

`decac [[-p | -v] [-n] [-r X] [-d]* [-P] [-w] [-o] <fichier deca>...] | [-b]`

Le compilateur `decac` propose plusieurs options permettant de contrôler son comportement. Les principales options sont les suivantes :

- **-b** : Banner
  – **Rôle :** affiche une bannière indiquant notre nom d'équipe.
  – **Effet observable :** sortie textuelle suivante :

```
./src/main/bin/decac -b
Deca compiler - équipe gl43
```

  – **Compatibilité :** ne peut être combinée avec aucune autre option ni fichier test, et n'entraîne ni la génération de fichier assembleur, ni son analyse.

- **-p** : Parser
  – **Rôle :** arrête `decac` après l'étape de construction de l'arbre, et affiche la décompilation de ce dernier.
  – **Effet observable :** sortie textuelle représentant le code deca décompilé.

```
./src/main/bin/decac -p src/test/deca/syntax/valid/provided/hello.deca

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

  - **Compatibilité :** peut être utilisée seul ou combinée avec d'autres options à l'exception de **-v** et **-b**, et n'entraîne pas la génération de fichier assembleur.

- **-v** : Vérification
  - **Rôle :** arrête `decac` après l'étape de vérifications contextuelles.
  - **Effet observable :** aucune sortie en l'absence d'erreur, et un message contenant l'erreur contextuelle générée et l'ensemble de fichiers parcourus dans le cas contraire.

```
./src/main/bin/decac -v  src/test/deca/syntax/valid/test_op_arith.deca
/home/mohammed/ensimag/GL/Projet_GL/src/test/deca/syntax/valid/test_op_arith.deca:11:10:
    Incompatible assignment: expected  int, got string
```

  - **Compatibilité :** peut être combinée avec toute option de test à l'exception de **-p** et **-b**, et n'entraîne pas la génération de fichier assembleur.

- **-n** : No check
  - **Rôle :** désactive la génération du code de vérification des erreurs à l'exécution lors de la compilation.
  - **Effet observable :** le fichier assembleur `.ass` produit ne contient aucun branchement vers des labels de gestion d'erreurs (par exemple pour les débordements arithmétiques, les accès mémoire invalides ou les déréférencements nuls).

```
./src/main/bin/decac src/test/deca/codegen/valid/created/addition.deca
```

Avec -n :

```
; start main program
ADDSP #2
; Beginning of main declarations:
; Beginning of main instructions:
LOAD #4, R1
ADD #5, R1
WINT
WNL
; Main program
HALT
; end main program
```

Sans -n :

```
; start main program
TSTO #2 ; Taille maximale de la pile
ADDSP #2
BOV stack_overflow_error
; Beginning of main declarations:
; Beginning of main instructions:
LOAD #4, R1
ADD #5, R1
BOV overflow_error
```

```
WINT
WNL
; Main program
HALT
; end main program
stack_overflow_error:
WSTR "Error : Stack Overflow"
WNL
ERROR
overflow_error:
WSTR "Error : Overflow during arithmetic operation"
WNL
ERROR
io_error:
WSTR "Error : Input / Output Error"
WNL
ERROR
```

  - **Compatibilité :** peut être combinée avec toute option de test à l'exception de **-b**, et entraîne pas la génération de fichier assembleur.

- **-r X** : Registres
  - **Rôle :** limite les registres banalisés disponibles à R0 ... RX-1, avec $4 <= X <= 16$.
  - **Effet observable :** si le nombre de registres est invalide, un message d'erreur est produit.

```
./src/main/bin/decac -r 3 src/test/deca/syntax/valid/provided/hello.deca
Error during option parsing:
Invalid number of registers, must be between 4 and 16
```

  - **Compatibilité :** peut être combinée avec toutes les options de test à l'exception de **-p** et **-b** et entraîne pas la génération de fichier assembleur.

- **-o** : Optimisations
  - **Rôle :** active les optimisations lors de la compilation complète.
  - **Effet observable :** fichier assembleur `.ass` optimisé.

    Sans -o :

```
; start main program
TSTO #6 ; Taille maximale de la pile
ADDSP #6
BOV stack_overflow_error
; Beginning of main declarations:
; Beginning of main instructions:
LOAD #1, R1
ADD #0, R1
BOV overflow_error
ADD #0, R1
BOV overflow_error
...
```

    Avec -o :

```
; start main program
TSTO #3 ; Taille maximale de la pile
ADDSP #3
BOV stack_overflow_error
```

```
; Beginning of main declarations:
; Beginning of main instructions:
LOAD #1, R1
WINT
WNL
...
```

  - **Compatibilité :** Compatible avec toute autre option sauf **b** uniquement applicable lors de la compilation complète et entraîne pas la génération de fichier assembleur.

- **Autres options :**
  - Certaines options peuvent être combinées selon le mode d'exécution.
  - Il est recommandé de consulter la documentation pour connaître les combinaisons possibles et leurs effets.

4 Tests automatisés

Organisation et exécution :

Les fichiers de test sont regroupés par type d'analyse (lexicale, syntaxique, contextuelle ou génération de code). Pour chaque phase, plusieurs scripts sont fournis afin de permettre une exécution ciblée, exhaustive ou optimisée.

Remarque:

- Il pourra être nécessaire d'autoriser l'exécution de ces fichiers avant de les lancer à l'aide de la commande suivante:

```
chmod +x script
```

- Certains scripts de test sont hiérarchisés et peuvent invoquer d'autres scripts afin d'éviter la duplication de code et de centraliser la logique d'exécution des tests.
- Les scripts responsables **test__XXX__energy.sh** et les scripts exhaustifs **not__basic__XXX.sh** (explicités ci-dessous) produisent les **mêmes types d'affichages** pour un test donné. En conséquence, seuls des exemples de sorties sont fournis pour les scripts **not__basic__XXX.sh**, afin d'éviter toute redondance.

En particulier :

- Les scripts **test__...__energy.sh** peuvent appeler à la fois **not__basic-....sh** et **test__....sh**, selon les fichiers réellement impactés par les modifications détectées.
- Les scripts **not__basic-....sh** eux-mêmes invoquent **test__....sh** afin d'exécuter individuellement chaque test et d'en analyser précisément le résultat.

4.1 Analyse lexicale

- **test_lex.sh** : lance l'analyse lexicale sur un seul fichier Deca (paragraphe **3.1**).
  Comportement : affiche le succès ou l'échec du test lexical pour le fichier donné.
- **not_basic-lex.sh** : exécute l'ensemble des tests lexicaux.
  Commande :

```
./src/test/script/not_basic-lex.sh
```

Exemple d'affichage (Succès et Echec attendus) :

```
on teste le fichier : src/test/deca/lex/invalid/chaine_incomplete.deca
Echec lexical attendu : src/test/deca/lex/invalid/chaine_incomplete.deca
Erreur lexical produite :
src/test/deca/lex/invalid/chaine_incomplete.deca:10:0: token recognition error at:
    '"chaine pas finie,

Tests lexicaux VALIDES
on teste le fichier : src/test/deca/lex/valid/just_a_test.deca
Succes lexical attendu : src/test/deca/lex/valid/just_a_test.deca
```

Comportement : affiche le succès ou l'échec de chaque test, avec les erreurs levées par l'analyseur lexical en cas d'échec.
Remarques et explications

  - Les résultats sont affichés sur le terminal pour chaque test : succès attendu / inattendu, échec attendu / inattendu, et erreur générée par l'analyseur en cas d'échec (attendu ou inattendu).

- **test_lex_energy.sh** : lance les tests de manière "responsable" (explication dans le paragraphe suivant).
  Remarques et explications

  - Les résultats sont affichés sur le terminal pour chaque test : succès attendu / inattendu, échec attendu / inattendu, et erreur générée par l'analyseur en cas d'échec (attendu ou inattendu).

4.2 Analyse syntaxique

- **test_synt.sh** : lance l'analyse syntaxique sur un seul fichier Deca (paragraphe **3.1**).
- **not_basic-synt.sh** : exécute tous les tests syntaxiques (valides et invalides). **Commande :**

```
./src/test/script/not_basic-synt.sh
```

Exemple d'affichage (Succès et Echec attendus) :

```
    on teste le fichier :
        src/test/deca/syntax/invalid/sansObjet/invalid_incomplete_assign.deca
Echec syntaxique attendu :
    src/test/deca/syntax/invalid/sansObjet/invalid_incomplete_assign.deca
Erreur syntaxique produite :
src/test/deca/syntax/invalid/sansObjet/invalid_incomplete_assign.deca:8:9:
    mismatched input ';' expecting {'(', 'new', 'readFloat', 'readInt', STRING,
    INT, FLOAT, '-', '!', 'true', 'false', 'null', 'this', IDENT}
```

```
    on teste le fichier :
        src/test/deca/syntax/invalid/sansObjet/invalid_incomplete_assign.deca
on teste le fichier : src/test/deca/syntax/valid/test_op_arith.deca
Succes attendu : src/test/deca/syntax/valid/test_op_arith.deca
```

Exemple d'affichage (Underflow) :

```
    on teste le fichier : on teste le fichier :
        src/test/deca/syntax/invalid/sansObjet/float_underflow.deca
Erreur syntaxique attendu (overflow/underflow/crash) sur
    src/test/deca/syntax/invalid/sansObjet/float_underflow.deca
Exception in thread "main" java.lang.IllegalArgumentException: Float underflow
    at org.apache.commons.lang.Validate.isTrue(Validate.java:136)
```

Comportement : affiche succès / échec et l'erreur syntaxique levée si échec.
Remarques et explications

- Les résultats sont affichés sur le terminal pour chaque test : succès attendu / inattendu, échec attendu / inattendu, et erreur générée par l'analyseur en cas d'échec (attendu ou inattendu).

- **test_synt_energy.sh** : lance les tests "responsables" pour cette étape (explication au paragraphe suivant).
  Remarques et explications
  - Les résultats sont affichés sur le terminal pour chaque test : succès attendu / inattendu, échec attendu / inattendu, et erreur générée par l'analyseur en cas d'échec (attendu ou inattendu).

4.3 Analyse contextuelle

- **test_context.sh** : lance l'analyse contextuelle sur un seul fichier Deca (paragraphe **3.1**).
- **not_basic-context.sh** : exécute tous les tests contextuels.

  Commande :

```
./src/test/script/not_basic-context.sh
```

Exemple d'affichage (Succès et Echec attendus) :

```
on teste le fichier :
    src/test/deca/context/invalid/provided/affect-incompatible.deca
Echec attendu : src/test/deca/context/invalid/provided/affect-incompatible.deca
Erreur contextuelle produite :
src/test/deca/context/invalid/provided/affect-incompatible.deca:15:7: Incompatible
    assignment: expected  int, got boolean

PASSED=56

Tests contextuels VALIDES

on teste le fichier : src/test/deca/context/valid/cast/cast_same.deca
Succes attendu : src/test/deca/context/valid/cast/cast_same.deca
PASSED=57
```

Comportement : succès / échec et affichage des erreurs contextuelles en cas d'échec.
Remarques et explications

- Les résultats sont affichés sur le terminal pour chaque test : succès attendu / inattendu, échec attendu / inattendu, et erreur générée par l'analyseur en cas d'échec (attendu ou inattendu).

- **test_context_energy.sh** : lance les tests contextuels de manière responsable (explication au paragraphe suivant).
Remarques et explications

  - Les résultats sont affichés sur le terminal pour chaque test : succès attendu / inattendu, échec attendu / inattendu, et erreur générée par l'analyseur en cas d'échec (attendu ou inattendu).

4.4 Génération de code

Tous les scripts ci-dessous peuvent être lancés avec l'option -o pour activer l'ensemble des optimisations, mais pas autre que cette option.

- **test_gencode.sh** : teste la génération de code pour un seul fichier Deca, comparant la sortie à un fichier `.expected` (paragraphe **3.1**).

- **not_basic-gencode.sh** : exécute tous les tests de génération de code présents dans le répertoire correspondant.
Commande :

```
./src/test/script/not_basic-gencode.sh
```

Exemple d'affichage :

```
>> Lancement du test :
    src/test/deca/codegen/valid/oracle/AvecObjet/test_appels_methods_imbriques.deca
Succès attendu pour le fichier valid :
    src/test/deca/codegen/valid/oracle/AvecObjet/test_appels_methods_imbriques.deca
PASSED=84
>> Lancement du test : src/test/deca/codegen/valid/oracle/AvecObjet/test3.deca
Succès attendu pour le fichier valid :
    src/test/deca/codegen/valid/oracle/AvecObjet/test3.deca
PASSED=85
```

Comportement : affiche le succès ou l'échec de chaque test, avec les erreurs levées par l'analyseur lexical en cas d'échec.
Remarques et explications

- Les résultats sont affichés sur le terminal pour chaque test : succès attendu / inattendu, échec attendu / inattendu, et erreur générée par l'analyseur en cas d'échec (attendu ou inattendu).
- Si le fichier `.expected` est absent, le script affiche :

```
Fichier non trouvé
Fichier attendu manquant : fichier .expected
```

- **test_gencode_energy.sh** : exécute les tests de génération de code des répertoires **/valid/created/**, **/valid/oracle/**, **/valid/optim/**, **/invalid/**, et **/perf/** de la section **/codegen/** de manière responsable (explication au paragraphe suivant).
  Commande :

```
./src/test/script/test_gencode_energy.sh
```

  Remarques et explications

  - Les résultats sont affichés sur le terminal pour chaque test : succès attendu / inattendu, échec attendu / inattendu, et erreur générée par l'analyseur en cas d'échec (attendu ou inattendu).
  - Si le fichier `.expected` est absent, le script affiche :

```
Fichier non trouvé
Fichier attendu manquant : fichier .expected
```

- **test_differential_energy.sh** : exécute tous les tests de génération de code du répértoire **/valid/regpressure** de manière responsable (explication au paragraphe suivant).
  Commande (Par défaut, toutes les configurations de registres allant de 4 à 16 registres):

```
./src/test/script/test_differential_energy.sh
```

  Commande (nombre de registres précisé par l'utilisateur): exemple avec 4, 5 et 6 registres

```
./src/test/script/test_differential_energy.sh 4 5 6
```

  Remarques et explications

  - Ce script exécute un ensemble de tests qui exercent une forte pression sur les registres, en utilisant l'ensemble des registres disponibles. Il lance ainsi différents tests avec une variété de configurations, allant de 4 à 16 registres par défaut, ou bien selon un ensemble de nombres de registres spécifié par l'utilisateur.
  - Les résultats sont affichés sur le terminal pour chaque test : succès attendu / inattendu, échec attendu / inattendu, et erreur générée par l'analyseur en cas d'échec (attendu ou inattendu).
  - Si le fichier `.expected` est absent, le script affiche :

```
Fichier non trouvé
Fichier attendu manquant : fichier .expected
```

- **run_metamorphic_tests.sh** : Ce test non énergétiquement responsable (vu la rareté de ces tests) lance les tests du répertoire **valid/metamorphic**.
  Commande (par défaut, toutes les configurations de registres allant de 4 à 16) :

```
./src/test/script/test_differential_energy.sh
```

  Commande (nombre de registres précisé par l'utilisateur) : Exemple avec 4, 5 et 6 registres :

```
./src/test/script/test_differential_energy.sh 4 5 6
```

Remarques et explitions :

- Ce script exécute un ensemble de tests dont la validation est pertinente. Il lance deux fichiers .deca dont les noms sont presque identiques (par exemple : `mr_add_remove_parenths_1.deca` et `mr_add_remove_parenths_2.deca` à un indice près), et la validation se fait en comparant les sorties de ces deux fichiers, et donc il ne nécessite pas la présence des fichiers .expected.
- Les résultats sont affichés sur le terminal pour chaque test : OK / KO.

- **test_all_energy.sh** : exécute tous les tests responsables mentionnés ci-dessus pour une vérification globale.
  Commande :

```
./src/test/script/test_all_energy.sh
```

Remarques et explications

- Ce fichier a pour rôle d'afficher les résultats de tous les scripts précédemment mentionnés dans une succession organisée, afin de fournir une vue d'ensemble du statut des tests.

4.5 Erreurs à l'exécution

Le compilateur et la machine abstraite IMA gèrent plusieurs types d'erreurs à l'exécution. Chaque type correspond à une situation particulière survenant pendant l'exécution d'un programme compilé. Le tableau ci-dessous présente les principaux types d'erreurs, leur déclenchement et des fichiers exemples :

- **STACK_OVERFLOW** : déclenchée lorsque la pile dépasse sa capacité maximale, par exemple lors d'appels récursifs profonds ou d'allocation excessive de variables locales. Fichier exemple : **src/test/deca/codegen/invalid/stack_overflow_trigger.deca**
- **OVERFLOW** : déclenchée lors d'un débordement arithmétique, par exemple lorsqu'une addition ou une multiplication dépasse la capacité d'un entier. Fichier exemple : **src/test/deca/codegen/invalid/overflow_add.deca**
- **DIVISION_BY_ZERO** : déclenchée lorsqu'une division par zéro est effectuée. Fichiers exemples :
  - Division entière : **src/test/deca/codegen/invalid/div_0_in_and.deca**
  - Modulo : **src/test/deca/codegen/invalid/modulo_par_0.deca**
- **IO_ERROR** : déclenchée en cas d'erreur d'entrée/sortie, par exemple lors d'une lecture ou écriture invalide. Remarque : ces tests nécessitent une interaction utilisateur et ne sont pas automatisables.
- **TAS_PLEIN** : déclenchée lorsqu'il n'y a plus de place disponible dans le tas mémoire pour l'allocation dynamique. Fichier exemple : **src/test/deca/codegen/invalid/AvecObjet/heap_overflow.deca**
- **DEREFERENCEMENT_NULL** : déclenchée lorsqu'une opération tente d'accéder à un objet via une référence nulle. Fichier exemple : **src/test/deca/codegen/invalid/AvecObjet/dereferecement_null.deca**

- **CAST_ERROR** : déclenchée lorsqu'un transtypage (`cast`) impossible est tenté. Fichier exemple :
  src/test/deca/codegen/invalid/AvecObjet/cast_impossible.deca

Listing 1: Exemple de gestionnaire d'erreurs à l'exécution

```
public ImmediateString getErrorMessage(RuntimeErrorType errorType) {
    switch (errorType) {
        case STACK_OVERFLOW:
            return new ImmediateString("Error : Stack Overflow");
        case OVERFLOW:
            return new ImmediateString("Error : Overflow during arithmetic operation");
        case DIVISION_BY_ZERO:
            return new ImmediateString("Error : Division by zero is forbidden");
        case IO_ERROR:
            return new ImmediateString("Error : Input / Output Error");
        case TAS_PLEIN:
            return new ImmediateString("Heap Overflow");
        case DEREFERENCEMENT_NULL:
            return new ImmediateString("Dereferencing null pointer");
        case CAST_ERROR:
            return new ImmediateString("Impossible cast operation");
        default:
            throw new AssertionError("Unknown runtime error type");
    }
}
```

5 Description des tests responsables

Le compilateur inclut un système de tests automatisés permettant de vérifier rapidement le comportement des programmes Deca. Ces tests sont **responsables**, c'est-à-dire qu'ils n'exécutent que les tests nécessaires selon les modifications apportées.

- **But :** valider les différentes phases du compilateur (lexicale, syntaxique, contextuelle et génération de code) sur un ensemble de fichiers de test fournis.
- **Principe de fonctionnement :**
  - Les scripts relancent des test selon leur impact sur les compilateurs **importance** :
    - **Fichiers "cœur" (core)** : ce sont des fichiers Java qui contrôlent le type d'analyse traité. Toute modification d'un fichier cœur entraîne le **re-lancement de tous les tests**, car ces fichiers impactent le comportement global du compilateur.
    - **Fichiers Deca normaux** : si un fichier Deca est modifié sans toucher à un fichier cœur, **seul ce test est relancé**.
  - **Fichiers Deca supprimés** : si un fichier Deca est supprimé, ce test est ignoré avec l'affichage du message suivant :

    ```
    Fichier supprimé, test ignoré : nom du fichier
    ```

  - **Fichiers "cœur" (core)** : ce sont des fichiers dont la modification entraîne le **re-lancement de tous les tests**, car ils impactent le comportement global du compilateur.
    Chemins selon le type d'analyse :

* **Analyse lexicale :**

```
src/main/antlr4/fr/ensimag/deca/syntax/DecaLexer.g4
src/main/java/fr/ensimag/deca/syntax/*
```

* **Analyse syntaxique :**

```
src/main/antlr4/fr/ensimag/deca/syntax/*.g4
src/main/java/fr/ensIMAg/deca/syntax/*
```

* **Analyse contextuelle :**

```
src/main/java/fr/ensimag/deca/context/*
src/main/java/fr/ensimag/deca/syntax/*
src/main/antlr4/fr/ensimag/deca/syntax/*.g4
src/main/java/fr/ensimag/deca/tree/*
```

* **Génération de code :**

```
src/main/java/fr/ensimag/deca/codegen/*
src/main/java/fr/ensimag/deca/context/*
src/main/java/fr/ensimag/deca/syntax/*
src/main/antlr4/fr/ensimag/deca/syntax/*.g4
src/main/java/fr/ensimag/deca/tree/*
```

- Les changements mineurs comme l'ajout, la modification ou la supression d'un commentaire ou d'une ligne vide **ne déclenchent pas le relancement du test**.
- Les tests de génération de code sont particuliers :
  - Chaque test de génération (avec l'exception des tests metamorphiques du répo **codegen/test/deca/valid/metamporphic**) nécessite un fichier `.expected` portant **le même nom** que le fichier Deca testé.
  - Si le fichier `.expected` est absent, le script renvoie une alerte :

```
Fichier non trouvé
Fichier attendu manquant: fichier.expected
```

  - Toute modification d'un fichier `.expected` entraîne le relancement du test correspondant, afin de garantir que la sortie générée correspond bien à la sortie attendue.

- **Organisation et exécution :**
  - Les fichiers de test sont regroupés par type d'analyse (lexical, syntaxique, contextuelle ou génération de code).
  - Les tests peuvent être lancés automatiquement via un script fourni, par exemple : `./runTests.sh`.
  - En cas de divergence entre le résultat obtenu et le fichier `.expected`, le script signale précisément le ou les fichiers concernés.
- **Avantages :**
  - Réduit le temps d'exécution des tests en ne relançant que les tests réellement affectés par les modifications.
  - Assure une couverture minimale et ciblée des fonctionnalités.

  - Facilite la validation des modifications et des optimisations du compilateur.

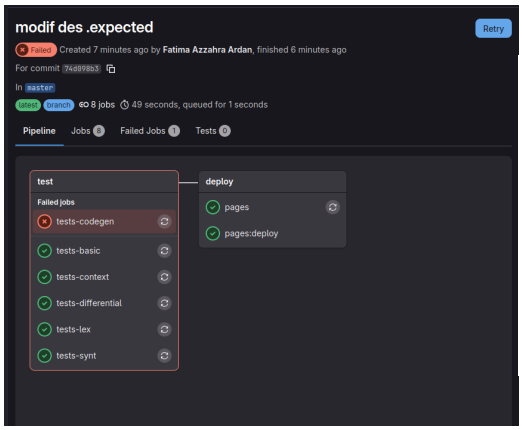
- **Compilation complète :**
  - Génération d'un fichier assembleur `.ass` pour chaque fichier source valide.
  - Nom du fichier généré identique au fichier source, extension `.ass`.
  - Placement du fichier dans le même répertoire que le fichier source.

6 Intégration Continue (CI) sur GitLab :

Nous avons mis en place une intégration continue sur GitLab, inspirée du projet CEP de l'année dernière. Cette CI s'appuie sur les tests responsables mentionnés précédemment pour lancer automatiquement les tests pertinents à chaque push.

Utilisation côté utilisateur : Traitez ces tests comme les autres tests énergétiques. Après un push, consultez le pipeline GitLab pour suivre l'exécution des tests et accéder aux logs détaillés.

Un `README` spécifique est fourni pour les compteurs : il redirige vers une page indiquant combien de tests ont réussi ou échoué pour chaque phase. Notez que ces compteurs reflètent uniquement les tests exécutés par le push concerné et peuvent afficher par exemple `0 passed/ 0 failed` si aucun test n'était impliqué par les modifications. Tous les log des différents job du pipeline sont donc directement accessibles à partir du Gitlab. Voilà un exemple du pipeline et de la page que le `README` peut rediriger vers à l'issue d'un push:



7 Intégration de Jacoco pour la couverture de code

Pour tester la couverture de code, nous utilisons Jacoco, un outil qui génère des rapports détaillés sur les parties du code testées et celles qui ne le sont pas.

7.1 Étapes pour lancer Jacoco

1. **Exécuter le script de tests :** Lancez le script `test_all.sh` pour exécuter tous les tests et générer le rapport Jacoco. Vous pouvez le faire en utilisant la commande suivante dans le terminal :

```
./src/test/script/test_all.sh
```

2. **Ouvrir le rapport Jacoco :** Une fois les tests exécutés, vous pouvez visualiser le rapport de couverture de code généré par Jacoco. Il suffit d'ouvrir le fichier HTML généré dans votre navigateur avec la commande suivante :

```
firefox target/site/jacoco/index.html
```

   Ce fichier contient des informations détaillées sur la couverture des tests, vous permettant de voir quelles parties du code ont été couvertes par les tests et celles qui ne l'ont pas été.

8 Limitations

Bien que les scripts automatisés soient conçus pour faciliter l'exécution des tests sur le compilateur Deca, il existe certaines limitations à prendre en compte :

- **Options des scripts :** Les scripts fournis n'acceptent pas une large variété d'options. Par exemple, des options telles que `-p`, `-v`, `-b`, etc., ne peuvent pas être combinées avec les scripts de test, ce qui limite leur flexibilité dans certains scénarios d'utilisation avancés.
- **Durée d'exécution des tests :** Certains scripts, notamment ceux de la forme **not_basic_XXX.sh**, exécutent l'ensemble des tests présents dans un répertoire donné. Cela peut entraîner des temps d'exécution relativement longs. Il est donc recommandé de les utiliser avec parcimonie.
- **Limitation liée à la pression sur les registres :** Le test **test_binary_tree.deca** dans **src/test/deca/codegen/valid/oracle/AvecObjet/** ne s'exécute pas correctement dans l'interpréteur `ima` lorsque le nombre de registres banalisés est limité à `-r 4`. Dans ce cas, une erreur d'exécution survient en raison d'une forte pression sur les registres. En revanche, ce test fonctionne correctement à partir de `-r 5`, ce qui indique une dépendance au nombre minimal de registres disponibles pour ce programme.
- **Gestion des commentaires et espaces dans les tests responsables :** Dans les scripts **test_XXX_energy.sh**, toute modification (commentaire ou espace) sur une ligne contenant du code entraîne le retestement du fichier. En revanche, les modifications portant uniquement sur des lignes de commentaires, d'espaces ou de lignes vides sont ignorées et ne déclenchent pas de nouveau test.

9 Optimisations

Des optimisations ont été intégrées au compilateur, notamment des améliorations au niveau de la gestion de la pile et des optimisations de code au moment de la génération, et au moment de l'analyse contextuelle. Ces optimisations peuvent être utilisées avec tous les scripts de génération de code, y compris la commande `decac` en utilisant l'option **-o**, comme mentionné précédemment.