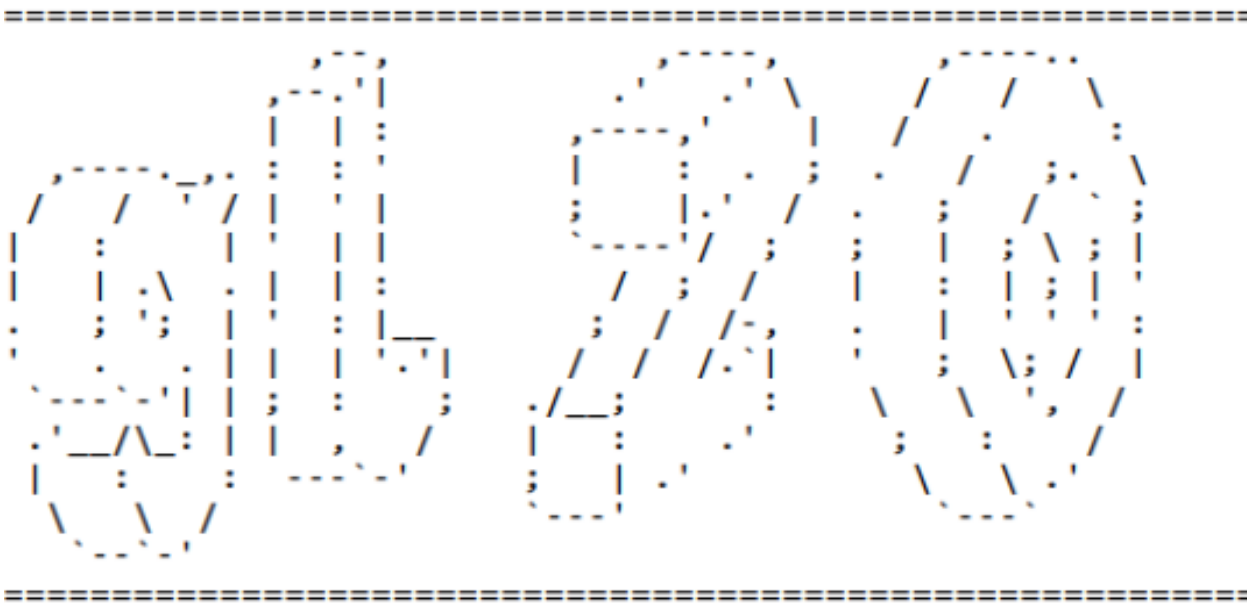
Documentation validation du compilateur Deca
- Groupe GL20



Melvin Bonamour
Steve Lening
Mathieu Meunier
Paul Grassi
Emmanuel Loroux

Table des matières

I- Descriptif des tests

- **types de tests pour chaque étape/passe**
- **organisation des tests**
- **objectifs des tests, comment ces objectifs ont été atteints.**

II- Les scripts de tests

- **Organisation générale**
- **Options des scripts**

III - Résultats de Jacoco

I- Descriptifs des tests

Préambule:

Tous les tests sont en langage deca et se trouvent dans les répertoires suivants:

test >

deca >

[nom_de_la_partie] >

invalid >

added >

provided >

valid >

added >

provided >

[nom_de_la_partie] correspond à la syntaxe, au contexte ou à gencode (génération code)

Objectif:

L'objectif de cette documentation est de présenter l'ensemble de la validation mise en place pour s'assurer de la fiabilité de notre compilateur.

Les tests sont soit valides soit invalides p

Partie A

La partie A réservée à la syntaxe possède deux launchers propres: test_lex et test_synt.

Test_lex renvoie les tokens reconnus par le langage deca. Test-synt fait un premier arbre syntaxique.

Les tests sont alors divisés en deux catégories (valides et invalides). Les tests valides sont valides par rapport à la syntaxe de la partie A (par exemple un programme possédant une division par zéro sera considéré comme valide par la partie A.

Note : Certains tests invalide ne déclencheront pas d'erreur par le test du lexeur.

Partie B

La partie B est quant à elle, réservée au launcher test_context et sert donc au spécificité contextuelle du langage déca. Une grande partie des tests de l'étape B est alors dans les tests invalides car toutes les erreurs contextuelles doivent être testées.

Partie C

Cette dernière partie finalisant le compilateur possède une gamme de test plus générale et ayant des thèmes plus explicites. A ce stade, seuls les tests valides sont implémentés car dès que le contexte est valide, la génération du code doit pouvoir se faire.
Les mêmes tests sont utilisés pour l'extension du Bytecode java

II- Les scripts de tests

Préambule :

Tous les scripts de tests sont en shell et sont formatés de la manière suivante:

./complex-*[partie testée]*.sh *[-options]*

Les scripts font un listing de tous les tests dans un dossier particulier

Objectif:

Ces scripts de tests ont été conçus pour faciliter le débogage de nos programmes. Ces scripts lancent tous les tests invalides puis valides avec des détails plus ou moins importants.

Utilisation :

Il y a au total quatre scripts différents:
Les trois premiers testent une étape particulière de la compilation à l'aide de tests unitaires (complex-lex.sh; complex-synt.sh; complex-context.sh). Le dernier teste la génération du code assembleur et Bytecode à l'aide des tests systèmes (complex-gencode.sh).

Scripts des tests unitaires.

Pour chaque test, le script va indiquer si la sortie attendue est atteinte (échec pour les tests invalides et succès pour les tests valides)

La reconnaissance des erreurs dans le script se fait par la détection d'un message d'erreur lors de l'exécution du launcher associé à l'étape de la compilation. Ainsi s'il reconnaît cette suite de caractère

nom_du_test.deca:nombre:

reconnaissable grâce à

grep -q -e "$(basename "$i"):[0-9][0-9]*:"

le script comprendra qu'une erreur a été commise.

Script des tests systèmes.

Pour chaque test, le script va d'abord écraser tout fichier assembleur résiduel du test s'il était auparavant généré. Puis va compiler le test en utilisant decac.
Comme certains fichiers avec des readInt ou readFloat existent dans le jeu de tests système. Le fichier assembleur est exécuté avec une entrée prévue pour passer tous nos tests interactifs, pour se faire l'utilisation de \\n permet d'écrire une entrée ultérieure pour un autre read du programme.
Chaque tests sont alors comparés au résultat attendu pré écrit dans les fichiers deca des tests dans l'en tête, entre les flags ***Resultats:*** et ***FIN Resultats***
L'option -B permet une génération de code en Bytecode Java au lieu de l'assembleur. (Cependant cette option est plus décrite dans la validation de l'extension.)

Verbose

Pour chaque script, l'option verbose (-v) permet d'afficher le détail des exécutions de chaque test. Ainsi il indiquera la liste des tokens pour le script du lexer test-lex.sh, l'arbre généré pour test-synt.sh, l'arbre décoré pour test-context.sh et le résultat du code assembleur pour test-gencode.sh.
Exemple: quelques lignes de complex-syntax.sh :

```
----------------------------------------
Test de chaine_incomplete.deca
Echec attendu pour test_synt
----------------------------------------
```

```
----------------------------------------
Test de div_par_0.deca
Succes attendu de test_synt
----------------------------------------
```

Exemple: quelques lignes de complex-syntax.sh -v:

```
----------------------------------------
Test de chaine_incomplete.deca
Reading from stdin ...
<unknown>:10:1: Erreur de syntaxe : Un token inconnu à la position indiquée empêche la compilation
Echec attendu pour test_synt
----------------------------------------
```

```
----------------------------------------
Test de div_par_0.deca
Reading from stdin ...
`> [11, 0] Program
   +> ListDeclClass [List with 0 elements]
   `> [11, 0] Main
      +> ListDeclVar [List with 1 elements]
      |  []> [12, 12] DeclVar
      |      +> [12, 8] Identifier (int)
      |      +> [12, 12] Identifier (x)
      |      `> [12, 16] Initialization
      |         `> [12, 16] Divide
      |            +> [12, 16] Int (4)
      |            `> [12, 18] Int (0)
      `> ListInst [List with 0 elements]
Succes attendu de test_synt
----------------------------------------
```

III Résultat Jacoco

Grâce aux scripts préalablement présentés. Leur insertion dans pom.xml permet d'utiliser Jacoco afin d'observer la couverture des tests sur l'ensemble du projet.
Pour l'utiliser il suffit de faire:

- **mvn compile**
- **mvn clean**
- **mvn verify**

Jacoco nous donnait une couverture de test de 60%, ce résultat peut être améliorer en affinant les détails sur les erreurs de syntaxe.