

Compilateur Decac

Document de validation

TEIMUR ABU ZAKI, ADRIEN BOUCHET, TROY
FAU, PAUL MARTHELOT, HUGO ROBERT

1 Arborescence des répertoires de tests

De façon à gérer plus efficacement la granularité des tests de non-régression (et donc leur temps d'exécution), nous avons cherché à dupliquer l'arborescence des fichiers Deca dans un répertoire nommé `results/` qui sert de stockage des résultats sous format `.lis` et `.res` ou `.ARMres`. Nous avons fait de même dans le répertoire `script/deca/` qui contient les scripts de tests automatiques.

Autrement dit, pour chacun des répertoires `src/test/deca/étape/validité/`, nous avons créé un répertoire associé pour les sauvegardes : `src/test/scripts/deca/étape/validité/` et un répertoire de scripts : `src/test/results/étape/validité/`.

Nous avons également ajouté un étage dans cette arborescence. Celle-ci correspond au sous-langage traité.



FIGURE 1 – Arborescence des tests, résultats et scripts décrite dans ce chapitre

Enfin, un répertoire `tmp/` a été ajouté pour le lancement des scripts. Il convient de le nettoyer régulièrement si ce n'est pas fait par les scripts.

2 Tests effectués

2.1 Tests en boîte noire de Decac : les launchers et les programmes Deca

Dans l'objectif de vérifier la qualité de notre compilateur et des lanceurs générés, nous avons rédigé des programmes Deca.

Chacun de ces tests visait à montrer une fonctionnalité ou une règle particulière du compilateur et a été validé manuellement (sauf pour la partie B de SansObjet) avant de servir comme témoins de non-régression. Nous avons par ailleurs subdivisé les tests en plusieurs répertoires HelloWorld/, SansObjet/ et Objet/ pour bien séparer les différents sous-langages vérifiés.

Les tests ainsi rédigés ne respectent pas convention de nommage unique mais parfois, certaines abréviations peuvent apparaître dans les noms. Nous les détaillerons dans chacune des parties.

2.1.1 Étape A

Il fallait ici tester les deux analyseurs (lexical et syntaxique) ainsi que la décompilation. Tous les tests sur l'analyseur lexical sont présents dans la partie invalide des test syntaxiques de façon à ne pas interférer avec le reste des tests valides pour l'analyseur.

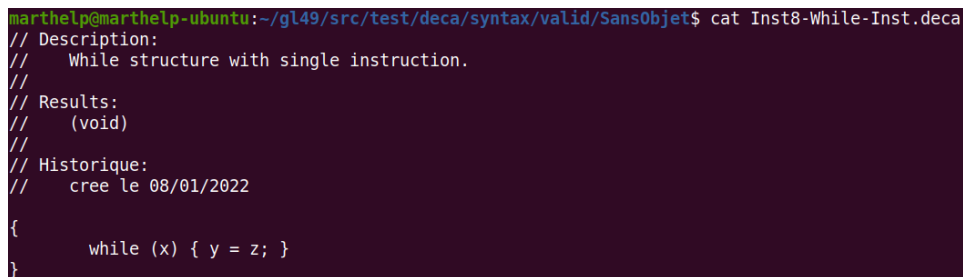
- **Analyseur lexical :** La plupart des tests ont été effectués sur les symboles pris ou non en compte par Deca par la convention ISO/CEI 646. Une autre partie des tests est dédiée au traitement des caractères spéciaux : fins de ligne, divers caractères d'échappements (en prêtant une attention particulière aux guillemets), commentaires, espaces et tabulations, etc. Les tests qui comportent des noms de la forme OOC ou OCO font référence à la façon dont les symboles sont agencés (O : *open*, C : *closing*). Vous trouverez tous ces tests dans la partie invalide pour syntax.

[illegible]

FIGURE 2 – Exemple de test invalide pour l’analyseur lexical

- **Analyseur syntaxique :** Les tests de l’analyseur ont été écrits de sorte à pouvoir parcourir au maximum l’arbre de la grammaire décrit dans les spécifications. Par exemple, la règle “père → filsA | filsB” donne lieu à au moins deux tests valides nommés père-filsA.deca et père-filsB.deca (ou père-i.deca où i = 1, 2 fait référence au numéro de la règle) Les tests invalides correspondants à une erreur (opérande en trop, manquantes ou inversées) sont nommés de la forme père-erreur.deca. Cela nous permettait d’identifier au mieux les tests déjà validés. Les tests de l’analyseur vont d’ailleurs droit au but : on cherche à écrire des programmes dérivant le plus vite possible sur la règle concernée. A mesure de la construction des tests pour la grammaire, on peut donc valider que chaque noeud est bien analysé et donc valider des sous-arbres de la grammaire.

Enfin, lorsque chaque noeud est testé, on peut valider que l’analyse donne bien lieu à la grammaire spécifiée. Nous avons également traité le cas spécial des include dans cette partie en générant des include pour des fichiers inexistantes ou des includes faisant appel à leur propre fichier.



```
marthelp@marthelp-ubuntu:~/gl49/src/test/deca/syntax/valid/SansObjet$ cat Inst8-While-Inst.deca
// Description:
//   While structure with single instruction.
//
// Results:
//   (void)
//
// Historique:
//   cree le 08/01/2022
{
    while (x) { y = z; }
}
```

FIGURE 3 – Exemple de test valide pour l’analyseur syntaxique

- **Décompilation et idempotence :** Nous vérifions que la composée de l’arbre d’analyseur syntaxique est cohérente en passant par la démonstration de la bonne sobriété de la décompilation de notre Decac. Chaque programme de test valide pour l’analyseur syntaxique est traité puis décompilé en le programme P2. On relance ensuite Decac sur ce nouveau programme pour obtenir P3 et on vérifie que P2=P3 grâce à la commande diff de shell.

2.1.2 Étape B

Dans cette partie, nos tests cherchent à identifier les différentes règles de syntaxe contextuelle : pour chaque règle un fichier de test au moins a été rédigé. En particulier, de nombreux tests ont été effectués pour gérer la compatibilité des types sur chacune des opérations binaires.

Pour faire cela d’une manière exhaustive, nous avons décidé de créer des scripts Python générant des fichiers tests .deca. Cette génération s’appuyait sur tous les

```

marthelp@marthelp-ubuntu:~/gl49/src/test/deca/syntax/valid/SansObjet$ decac -p IfThenElse-IfElse.deca
{
    if((x>3)) {
        (x = 2);
    } else {
        (x = 3);
    }
}
marthelp@marthelp-ubuntu:~/gl49/src/test/deca/syntax/valid/SansObjet$ decac -p IfThenElse-IfElse.deca > tmp.deca
marthelp@marthelp-ubuntu:~/gl49/src/test/deca/syntax/valid/SansObjet$ decac -p tmp.deca
{
    if((x>3)) {
        (x = 2);
    } else {
        (x = 3);
    }
}
marthelp@marthelp-ubuntu:~/gl49/src/test/deca/syntax/valid/SansObjet$ decac -p tmp.deca > tmp2.deca
marthelp@marthelp-ubuntu:~/gl49/src/test/deca/syntax/valid/SansObjet$ diff tmp.deca tmp2.deca

```

FIGURE 4 – Test manuel de l’idempotence

types définis dans Deca avec lesquelles nous pouvons effectuer des opérations binaires ou unaires. Ainsi, nous avons généré un test pour chaque combinaison possible d’expression binaire { opérateur, opérande gauche, opérande droite } et pour chaque possibilité d’opération unaire opérateur, opérande. Les scripts Python utilisaient des dictionnaires capables de faire la vérification contextuelles, et donc de prévoir les éventuelles erreurs.

```

marthelp@marthelp-ubuntu:~/gl49/src/test/deca/context/valid/SansObjet$ cat DeclVar-Lower-boolean-float-int.deca
// Description:
// Declaration of a boolean variable using < operation between a float variable and int .
//
// Results:
// nothing
//
// Historique:
// cree le 11/01/2022
//
{
    float x = 10.01;
    int y = 3;
    boolean z = x < y ;
}

```

FIGURE 5 – Exemple de test valide généré automatiquement pour l’analyse contextuelle

Le reste des tests a été généré manuellement pour avoir plus de contrôle sur les règles les plus compliquées et couvrir plus de règles.

2.1.3 Étape C

La distinction entre les tests valides et invalides de la partie C est assez forte. En effet, la partie valide cherchait à mettre à mal la génération de code : nous voulions surtout savoir si le résultat renvoyé par l’exécutable était bon. Les tests soumis à notre decac étaient de plus en plus difficiles (ajouts de blocs imbriqués, ajout de plusieurs variables, opérations parenthésées, etc.) de façon à pouvoir expliciter les faiblesses de notre compilateur.

On notera d’ailleurs que certains tests sont destinés à forcer les PUSH et POP sur nos registres selon le nombre de registres donnés. De fait, une mauvaise gestion

```

marthelp@marthelp-ubuntu:~/gl49/src/test/deca/context/invalid/Objet$ cat SameDeclaration-Field-After
-Method.deca
// Description:
//   Using same ident as method for field
//
// Results:
//   Line 13 : Rule (2.6) Identifier already used
//
// History:
//   created on 17/1/2022
class A {
    void x() {
    }
    protected int x;
}

```

FIGURE 6 – Exemple de test invalide pour l’analyse contextuelle (règle 3.72)

des registres ou de la pile pouvait plus facilement être identifié. En parallèle de cela, nous avons réalisé des tests manuels grâce à l’interpréteur-metteur au point d’IMA.

Par comparaison, la partie invalide cherchait surtout à explorer les limites de la machine abstraite pour pouvoir générer un certain nombre d’erreurs à l’exécution. Par exemple, pour pouvoir générer un débordement de pile, nous avons estimé à plus de entre 10 000 et 100 000 le nombre de PUSH nécessaires. Il fallait alors générer un programme deca à plus de 10000 variables pour pouvoir tester la bonne traitement de l’erreur (même s’il était beaucoup plus facile de générer un fichier .ass contenant un TSTO adéquat). Nous avons également cherché à produire des débordements arithmétiques pour les flottants, des divisions par zéro, des méthodes à signature non nulle ou sans retour (que ce soit en Deca ou en assembleur).

Aucune convention de nommage n’a été respectée ici mais la plupart du temps les noms des tests sont explicites décrivent bien l’aspect de la génération de code concerné (cast, equals, instanceof, etc.).

Très peu de tests interactifs ont été réalisés car ils concernaient seulement ReadInt et ReadFloat. En revanche, ces tests ont été lancés manuellement avec des entrées suffisamment diverses pour pouvoir couvrir les cas un peu particuliers tels que la conversion d’un *int* en *float*.

2.1.4 Extension ARM

Pour l’extension, nous avons exploité les mêmes tests que pour la partie C SansObjet, il suffisait alors de comparer les résultats générés pour ARM aux résultats précédemment validés par l’étape C.

Cependant, en l’absence de metteur au point, nous n’avons pas pu effectuer de débogage pour la partie ARM. La seule chose qu’il était possible de générer était la structure du programme dans la mémoire. Nous n’avons donc aucun contrôle sur l’état des registres à l’instant *t*.

```

marthelp@marthelp-ubuntu:~/gl49/src/test/deca/codegen/valid/Objet$ cat DoubleSelection.deca
// Description:
//   Program testing the good management of double selection
//   on fields
//
// Results:
//   1
//
// History:
//   Created on 20/1/2022

class A {
    int x = 1;
}

class C {
    A a = new A();
}

{
    C c = new C();
    println((c.a).x);
}

marthelp@marthelp-ubuntu:~/gl49/src/test/deca/codegen/invalid/Objet$ cat NoReturn-Asm.deca
// Description:
//   Class with an assembly int function without RTS at the end
//
// Result:
//   ERROR: no_return
//
// History:
//   created on 17/1/2022

class A {
    int x;

    int function()
    asm ( "LOAD #1, R0
" );
}

{
    A a = new A();
    int x = a.function();
}

```

FIGURE 7 – En haut : test valide ; en bas : test invalide

2.2 Tests unitaires

2.2.1 Les options de compilation

Pour faire fonctionner decac selon les spécifications, il fallait à la fois pouvoir analyser les options données en argument de la commande mais aussi réaliser les fonctionnalités sous-jacentes. Si l'analyse des options fait l'objet de plusieurs tests unitaires (sur plusieurs ensembles artificiels d'options compatibles ou non), ce n'est pas vraiment le cas pour le reste des options qui ont été testées manuellement.

Pour l'affichage de l'utilisation de decac ou de la bannière, ces tests se font très rapidement en lançant decac (ou decac -b).

Pour l'option de parsing et de vérification, elles avaient déjà été implémentées

```

[INFO] --- exec-maven-plugin:3.0.0:exec (ARM-tests) @ Deca ---
Start of valid generated ARM code tests for SansObjet
[EXPECTED OUTPUT] : BinaryOp-multiple-float
[EXPECTED OUTPUT] : Cast-Boolean-Boolean
[EXPECTED OUTPUT] : Cast-ConvFloat
[EXPECTED OUTPUT] : Cast-DeclVar
[EXPECTED OUTPUT] : Cast-Float-Float
[EXPECTED OUTPUT] : Cast-SameType
[EXPECTED OUTPUT] : Cast-SameTypeDeclVar
[EXPECTED OUTPUT] : Cast
[EXPECTED OUTPUT] : Div-NN
[EXPECTED OUTPUT] : Div-NP

```

FIGURE 8 – Affichage des scripts d’exécution pour ARM sur les tests de la partie C

avant et nous n’avons pas cherché à les tester.

Pour l’option `-P` de lancement de `decac` sur plusieurs fichiers, en parallèle, nous avons lancé `decac` puis `decac -P` sur un répertoire de test contenant un grand nombre de fichiers (le script associé est `src/test/script/time-parallel.sh`). En comparant les temps d’exécution de compilation et d’utilisation du CPU, on se rend compte que le parallélisme est bien géré. L’option de non-vérification des erreurs a également été testé par comparaison des résultats des exécutables générés en présence de l’option `-n` aux résultats déjà sauvegardés.

2.2.2 Les tests d’intégration

D’autres tests unitaires pour les parties les plus sensibles ont été rédigées en utilisant les bibliothèques JUnit Jupiter et Mockito. Les tests en questions concernent : *EnvironmentExp* et *EnvironmentType*, classes gérant l’association des symboles et de leurs définitions, *ContextTools*, regroupant les fonctions utilisées pour vérifier la compatibilité des opérandes dans les opérations binaires et *Register*, qui gère les registres libres ou non, avec un certain nombre de variables PUSH et POP dessus. Ces tests nous ont permis de vérifier le bon fonctionnement des parties avant de les insérer dans le reste du compilateur.

3 Scripts de tests

Pour faire passer tous les tests, il suffit de lancer la commande `mvn verify` à la racine du projet.

Tous les tests précédemment décrits seront alors automatiquement lancés (après compilation automatique du projet). Grâce à la configuration initiale du fichier `pom.xml` à la racine, nous pouvons y voir au lancement, dès le début, les tests Junit qui sont validés.

Cependant, nous y avons inclus de nouvelles règles pour nos scripts faits-maison qui sont lancés après. Ces scripts permettent de lancer tous les `test_lex/synt`,

`test_context` et `decac` fournis (respectivement pour les parties A, B et C) sur tous les fichiers de `src/test/deca` non interactifs et non fournis (cf. 2.1). Les résultats ainsi générés sont placés dans le répertoire `src/test/tmp/`.

Ensuite, on compare les résultats à ceux qui ont été validés manuellement et stockés dans le dossier `src/test/results`. *Pour plus de détails sur la conception des scripts, se référer à l'annexe.*

De plus, nous testons également automatiquement l'idempotence sur les programmes Deca en générant pour chacun d'entre eux deux composées consécutives de l'arbre abstrait de l'analyseur syntaxique et en les soumettant ces deux résultats temporaires à la commande `diff`.

D'autres scripts de tests pour les options `decac` ont été décrits dans la partie 2.2.1.

Il est important de préciser que tous ces scripts ont été validés par deux personnes dans l'équipe pour permettre au restant de l'équipe de les utiliser sans crainte. Ils ont alors permis de trouver des failles plus rapidement lors du développement mais également de valider le produit au moment des rendus.

4 Gestion des risques et des rendus

Cette partie a fait l'objet d'un rendu pour le suivi n°2 mais nous l'incluons tout de même ici.

4.1 Risques logistiques

4.1.1 Date des rendus et suivis

Le risque de ne pas être prêt pour les dates des rendus et des suivis a été identifié comme le risque majeur du projet. Nous avons donc organisé notre planning en tenant compte des dates des rendus, et en se donnant un peu d'avance par rapport à ces dates. Celles-ci sont prises en compte par notre Trello, qui envoie des notifications à l'approche des rendus.

4.1.2 Manque de coordination et partage des tâches

Pour que chacun d'entre nous puisse se situer dans le projet et savoir ce qu'il doit faire, nous effectuons une réunion de groupe tous les matins, ce qui nous permet d'échanger, de faire le point sur ce qui a été fait et de déterminer les tâches de chacun pour la journée.

4.2 Risques techniques

4.2.1 Non-compilation des fichiers sources

En compilant régulièrement notre compilateur après avoir fait des modifications, nous veillons à ce qu'il ne contient pas des erreurs critiques qui empêchent

cette compilation, ce qui permet de corriger les erreurs d'oubli de point-virgule, de parenthèse, etc.

4.2.2 Erreurs lors de l'ajout de fonctionnalités

En faisant tourner nos tests, nous veillons à ce que le compilateur arrive à s'exécuter. En faisant tourner `common-tests.sh`, nous évitons les bugs les plus simples mais les plus dévastateurs dans une base de tests automatisée.

4.2.3 Tests erronés ou incorrects

Pour les tests sur des programmes Deca, nous effectuons une validation manuelle des tests la première fois qu'ils sont lancés et avant de stocker les résultats. Pour les tests automatiques comme les tests unitaires avec JUnit, nous effectuons souvent une revue de code pour s'assurer que les tests font bien ce que nous voulons.

4.2.4 Régression du compilateur

Une fois une série de tests validée, nous stockons leur résultat, et à chaque étape dans le projet (telle que le définit notre Trello), nous effectuons des tests de régression pour s'assurer que le compilateur fonctionne correctement sur les tests déjà validés.

4.3 Risques humains

4.3.1 Conflits

Pour éviter tout conflit, nous débattons posément lorsque des membres du groupe ne sont pas d'accord sur la marche à suivre. Nous travaillons le plus possible en présentiel tous les cinq, et nous mangeons régulièrement ensemble au RU pour échanger en dehors du travail et créer du lien dans l'équipe.

4.3.2 Démoralisation

Pour éviter qu'une personne ne perde le moral seule face à un problème difficile, tout membre du groupe peut demander de l'aide ou des suggestions aux autres membres du groupe.

4.3.3 Covid-19, maladies et indisponibilités

Si l'un des membres du groupe doit s'arrêter de travailler pour cause de maladie, les autres membres du groupe doivent pouvoir prendre le relais. Pour assurer cela, tous les membres du groupe connaissent un minimum tous les aspects du projet, quelque soit leur rôle précis dans le projet, pour pouvoir rapidement prendre la relève si besoin est.

5 Résultats de Jacoco

Après lancement de `mvn verify`, le rapport Jacoco nous donnait une couverture de 82%, extension comprise.

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		78%		57%	517	737	445	2,068	242	367	1	48
fr.ensimag.deca.tree		90%		77%	135	760	196	2,175	49	524	0	86
fr.ensimag.ima.pseudocode		70%		50%	85	206	118	451	39	143	3	37
fr.ensimag.deca		74%		72%	44	131	101	341	7	58	1	5
fr.ensimag.ima.pseudocode.instructionsARM		39%		n/a	50	89	103	180	50	89	15	40
fr.ensimag.deca.context		83%		86%	45	202	52	287	29	129	1	23
fr.ensimag.ima.pseudocode.instructions		66%		n/a	22	62	38	111	22	62	16	54
fr.ensimag.deca.tools		86%		66%	10	54	13	115	3	42	0	6
fr.ensimag.deca.codegen		98%		50%	4	20	1	99	1	17	0	3
Total	4,689 of 25,693	81%	528 of 1,602	67%	912	2,261	1,067	5,827	442	1,431	37	302

FIGURE 9 – Index du rapport Jacoco peu avant le rendu final

Cependant, avant la validation finale du produit, nous étions à 78% : nous avons pu nous rendre compte que certaines fonctions de génération de code n'étaient pas appelées alors qu'elles auraient pu l'être facilement. En ce sens, Jacoco nous a permis d'étendre notre base de tests et ainsi confirmer la validité de notre compilateur.

Après étude du rapport, les parties majeures non couvertes qui restaient ont été associées aux cas de lancement d'erreur du compilateur, des méthodes `toString` servant au débogage, ou à des instructions IMA ou ARM non utilisées. Nous avons jugé bon de laisser ces vestiges de débogage et d'instructions pour les éventuelles personnes qui reprendraient le projet.

6 Autres méthodes de validation utilisées

6.1 Revue de code

D'autres méthodes de validation plus manuelles ont été utilisées : tout d'abord l'analyseur lexical et l'analyseur syntaxique ont été relus par les membres de l'équipe n'ayant pas touché au code. Pour l'étape de génération de code, nous avons procédé différemment : le testeur et le codeur ont travaillé conjointement pour identifier les erreurs et les corriger rapidement. Lorsque nous avons commencé à travailler sur la partie ARM, nous avons également pu retrouver des erreurs liées à la non-couverture de la génération de code pour la partie C.

6.2 Validation de la conception

A chaque ajout d'une structure de données nécessaire au développement, au moins deux personnes du groupe se sont accordées sur la façon dont procéder. Cela comprenait la réalisation de l'architecture en commun (entrées et sorties des programmes, implémentation, manière d'utiliser la nouvelle structure), la validation par tous les membres ayant contribué à l'architecture puis la réalisation des tests

et des codes en même temps. Cela nous a permis de gagner du temps sur l'apprentissage des codes écrits par autrui.

6.3 Validation collective avant les rendus

Avant chaque rendu, nous re-clonions le dépôt distant puis lançons la commande `mvn verify` de sorte à rendre un code qui passait tous les tests, quitte à supprimer certaines fonctionnalités. Par exemples, le rendu intermédiaire ne contenait rien pour l'extension ARM alors que les fonctionnalités étaient bien en cours de développement.

Annexe - Conception des scripts

Si vous avez bien suivi l'arborescence des répertoires décrite dans la partie 1, nous pouvons nous placer dans l'un des répertoires de scripts correspondant à la partie que nous voulons tester. Admettons que ce soit les tests invalides de la partie A pour le sous-langage SansObjet. Nous nous plaçons donc dans le dossier `src/test/scripts/deca/syntax/invalid/SansObjet`.

Nous trouvons alors plusieurs scripts nommés en général par `exec-tests-sous-langage.sh` et `regression-tests-sous-langage.sh`, et de simples `exec-tests.sh` ainsi que `regression-tests.sh`. Les `exec-tests` servent à générer les fichiers de résultats dans `src/test/tmp` tout en validant le succès (pour `valid`) ou l'échec (pour `invalid`) pour un certain niveau de granularité. S'il n'y a pas de sous-langage précisé, c'est que le script concerne tous les sous-langages à la fois. Pour ce qui des fichiers `regression-tests`, ils permettent de comparer les fichiers temporaires ainsi générés à leur équivalent enregistrés dans `src/test/results/étape/validité/sous-langage`.

Pour plus de facilité un script `allin-sous-langage.sh` permet de lancer les deux scripts précédents à la fois.

Enfin, on met à la racine de chaque répertoire d'étape ou de validité un script qui peut lancer tous les scripts des sous-répertoires ainsi décrits. `mvn verify` lance alors le `allin.sh` situé le plus haut dans l'arborescence pour pouvoir générer tous les fichiers et les comparer à leurs résultats sauvegardés.

Sur la Figure 10, on peut observer les résultats en cas de réussite du `mvn verify` :

En cas d'échec, on verrait apparaître le nom des fichiers problématiques en rouge.

Pour plus de facilité d'utilisation, nous avons également rédigé un script `Launch-test.sh` et `Regression-test.sh` prenant pour argument un launcher (`lex`, `synt` ou `context`) puis le type de test (`valid` ou `invalid`) et un sous-langage (`HelloWorld`, `SansObjet`, `Objet`) pour utiliser les launchers fournis sur le répertoire qui nous intéresse et identifier tous les tests qui passent. (cf. Figure 11)

```

[INFO] --- exec-maven-plugin:3.0.0:exec (all-tests) @ Deca ---
Starting valid synt tests for HelloWorld
[OK] : DoubleComment-00C
[OK] : HelloWorld
[OK] : Inst4-EmptyListExpr
[OK] : ListInst-0Inst
[OK] : ListInst-1Inst
[OK] : ListInst-5Inst
[OK] : Main1-EmptyProgram
[OK] : Main2-EmptyBlock
[OK] : Println-DblEscp
[OK] : Println-FourEscp
[OK] : Println-Multiple
[OK] : Println-OneEscpQuote
[OK] : Println-TrplEscpQuote
Starting regression verification on valid synt tests for HelloWorld
[NO-REGRESSION] : DoubleComment-00C
[NO-REGRESSION] : HelloWorld
[NO-REGRESSION] : Inst4-EmptyListExpr
[NO-REGRESSION] : ListInst-0Inst
[NO-REGRESSION] : ListInst-1Inst
[NO-REGRESSION] : ListInst-5Inst
[NO-REGRESSION] : Main1-EmptyProgram
[NO-REGRESSION] : Main2-EmptyBlock

```

FIGURE 10 – Affichage des scripts : OK pour les tests valides réussis, NO-REGRESSION lors de la comparaion avec les résultats précédemment sauvegardés

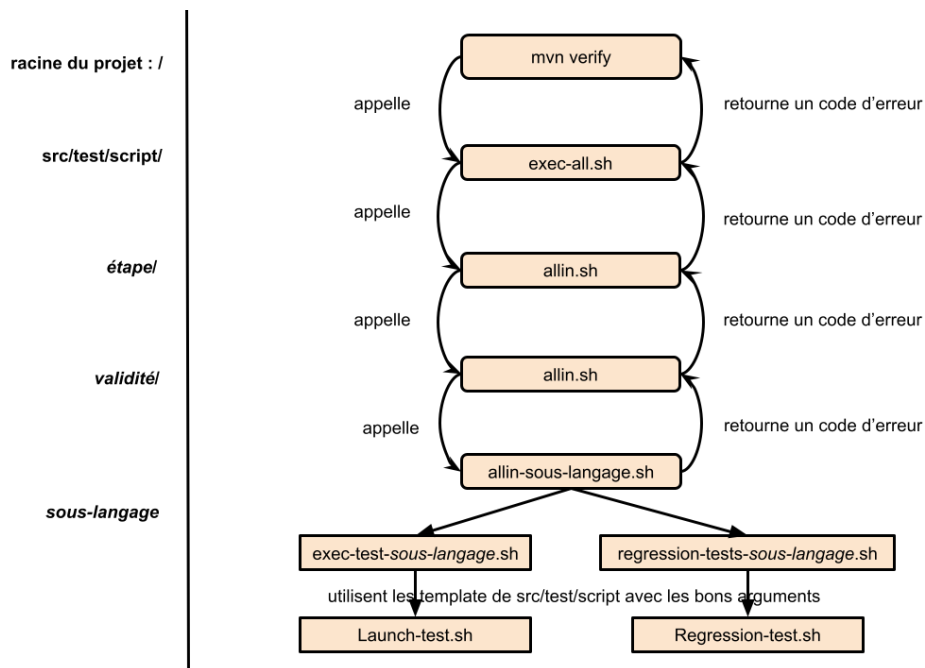


FIGURE 11 – Schéma représentant le fonctionnement des scripts dans l’arborescence