Documentation de validation: Projet génie logiciel

Auteurs :
Mehdi El Idrissi El Fatmi
Khadija El Adnani
Alaa Jennine
Badr El Khoundafi
Yassine El Kouri

22/01/2025

Contents

Introduction

Dans le cadre du projet Génie Logiciel, la validation constitue une étape essentielle pour garantir que le compilateur Deca respecte les spécifications et fonctionne correctement. Cette documentation présente les méthodologies, les outils et les processus adoptés pour évaluer la conformité des différentes étapes du développement.

L'objectif est de fournir une description complète des critères de validation, des scénarios de test et des résultats obtenus, avec une attention particulière portée aux tests unitaires et fonctionnels. Ces éléments permettent d'assurer la robustesse et la fiabilité du produit final tout en respectant les contraintes du projet.

Cette démarche s'inscrit dans une volonté de livrer un compilateur de haute qualité, répondant aux exigences fonctionnelles et techniques fixées dans le cahier des charges.

1 Descriptif des tests

Dans un premier temps, des tests unitaires ont été réalisés pour valider chaque composant du compilateur de manière isolée. Ces tests ont permis de vérifier que chaque partie implémentée (analyse lexicale, syntaxique, contextuelle, etc.) fonctionnait correctement selon les spécifications. Une fois ces composants validés individuellement, des tests d'intégration ont été mis en place. Ces derniers ont intercepté l'ensemble du processus, depuis l'entrée d'un programme source jusqu'à la génération du code final, afin de s'assurer de la cohérence et du bon fonctionnement global du compilateur. Cette approche progressive a permis d'identifier rapidement les erreurs spécifiques tout en garantissant une validation complète de l'ensemble du système.

Dans un premier temps, nous avons concentré nos efforts sur des tests liés à la partie sans objet, en repoussant les tests pour la partie orientée objet à une phase ultérieure. Les tests de chaque étape ont été soigneusement organisés dans des dossiers dédiés, en veillant à ne pas réutiliser les tests d'une étape dans une autre. Ainsi, chaque partie a bénéficié d'un ensemble de tests spécifique. Nous avons également décidé que les tests de la partie B incluraient automatiquement l'exécution de la partie A, et de même pour la partie C, qui inclut les deux étapes précédentes. Cette approche progressive nous a permis de maximiser la couverture de code tout en identifiant des erreurs potentielles que les tests isolés des étapes antérieures auraient pu manquer.

2 Tests des options

Bien qu'un script automatique ait été envisagé pour tester les options du compilateur Decac, la contrainte de temps n'a pas permis sa mise en œuvre. Les tests ont donc été réalisés manuellement pour s'assurer du bon fonctionnement de chaque option.

L'option `-b` a été vérifiée pour confirmer qu'elle affiche correctement les informations sur le projet et quitte immédiatement, conformément à la spécification. L'option `-p` a été testée avec des fichiers sources variés (valides, avec erreurs syntaxiques ou cas limites), confirmant qu'elle effectue uniquement une analyse lexicosyntaxique sans générer de fichier assembleur. L'option `-v`, utilisée pour l'analyse contextuelle, a été validée avec des programmes valides et invalides. Elle signale précisément les erreurs contextuelles sans générer d'arbre décoré ni de fichier assembleur. Il est également à noter que le script

des tests de la partie B repose sur l'utilisation de l'option `-v`, confirmant ainsi son bon fonctionnement.

Pour `-r`, des tests ont confirmé que le compilateur accepte un nombre de registres compris entre 4 et 16, générant des erreurs appropriées pour des valeurs hors limites. L'option `-d` a été vérifiée avec différents niveaux de débogage, s'assurant que des informations détaillées sur le processus de compilation sont affichées à chaque niveau. Pour l'option `-n`, des cas contenant des erreurs d'exécution potentielles ont été utilisés, confirmant que ces erreurs ne sont pas signalées lorsqu'elle est activée. Enfin, pour `-P`, des tests ont démontré que la compilation parallèle fonctionne correctement avec des fichiers indépendants.

3 Tests de l'étape A : Analyse lexicale et syntaxique

L'étape A du compilateur Decac se concentre sur la transformation du programme source en une représentation lexicale et syntaxique correcte. Pour valider cette étape, des tests spécifiques ont été conçus afin de couvrir une large gamme de scénarios, des cas simples aux cas complexes.

3.1 Objectifs des tests

Les tests de l'étape A ont pour objectifs :

- De vérifier que les lexèmes sont correctement identifiés et classifiés selon les règles définies dans la spécification lexicale.
- De s'assurer que les programmes syntaxiquement valides sont correctement traduits en arbres abstraits primitifs.
- De détecter et signaler les erreurs lexicales et syntaxiques de manière claire et précise.
- De vérifier que la fonction **decompile** permet de réafficher le programme source correctement.

3.2 Approche de test

Pour cette étape, les tests ont été organisés en deux catégories principales :

1. **Tests valides** : Des programmes Deca valides, incluant des cas simples (exemple : `println("Hello, world");`) et des programmes comportant des constructions syntaxiques plus complexes.
2. **Tests invalides** : Des programmes volontairement erronés, visant à tester la gestion des erreurs comme des identificateurs non valides, des symboles inattendus ou des parenthèses non appariées.

3.3 Exemples de tests

```
{
    boolean var = 0 > 1;
    if (var){
```

```
        println("vrai");
    }
    else {
        println("faux");
    }
}
```

Le résultat de l'analyse syntaxique pour ce programme est illustré par l'arbre suivant :

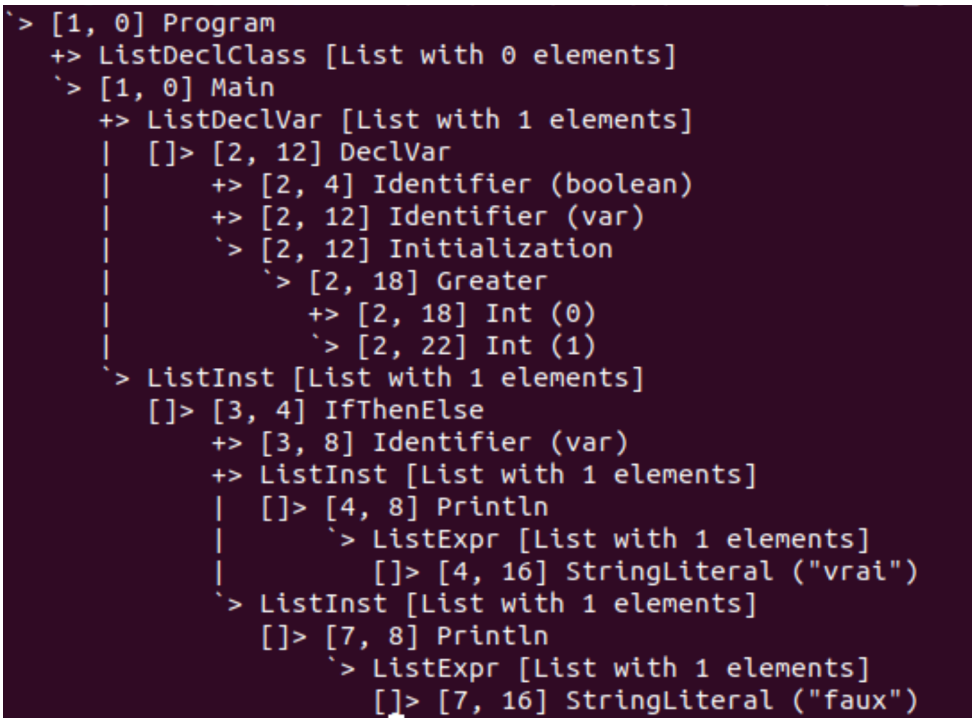

Figure 1: Arbre syntaxique généré pour le programme valide

Programme avec une erreur syntaxique

Un programme incorrect où une erreur syntaxique est présente dans la condition `if` :

```
{
    if {
        a = b;
    }
}
```

Résultat attendu : une erreur est signalée avec un message précis indiquant une mauvaise structure syntaxique ou un identificateur invalide dans la condition du `if`.

```
src/test/deca/syntax/invalid/provided/perso/if_Sans_cond.deca:2:6: mismatched input '{' expecting '('
```

Figure 2: Message d'erreur généré pour une mauvaise syntaxe dans la condition `if`

3.4 Organisation des tests

Les tests ont été classés dans des répertoires distincts en fonction de leur objectif :

- `gl02/src/test/deca/syntax/valid/provided/perso` : Contient des programmes syntaxiquemenet valides.
- `gl02/src/test/deca/syntax/invalid/provided/perso` : Contient des programmes avec des erreurs syntaxiques.

3.5 Résultats des tests

Les tests ont permis de valider le fonctionnement de l'analyse lexicale et syntaxique dans la plupart des scénarios. Les erreurs sont signalées conformément au format spécifié :

```
<nom_fichier>:<ligne>:<colonne>: description de l'erreur
```

Cette approche a assuré une couverture complète des cas d'utilisation, tout en permettant une détection rapide et précise des anomalies.

4 Tests de l'étape B : Analyse contextuelle

L'analyse contextuelle vérifie que les programmes validés par l'analyse syntaxique respectent les règles sémantiques définies dans le langage Deca. Cette étape est cruciale pour garantir la cohérence et la validité des programmes avant leur compilation finale. Les tests de cette étape ont été conçus pour détecter des erreurs liées aux déclarations, aux types, et à l'utilisation des variables et fonctions.

4.1 Objectifs des tests

Les tests de l'analyse contextuelle ont pour but :

- De valider que les variables et fonctions utilisées dans le programme ont été correctement déclarées.
- De vérifier la cohérence des types dans les expressions et les affectations.
- D'identifier des erreurs liées aux structures interdites, comme la redéclaration d'identificateurs dans le même environnement.
- De s'assurer que les règles spécifiques du langage (par exemple, les contraintes sur les paramètres de fonctions ou les conversions implicites) sont respectées.

4.2 Approche de test

Les tests pour cette étape ont été organisés en deux catégories principales :

1. **Tests valides** : Programmes conformes aux règles contextuelles, permettant de vérifier que l'analyse est correctement effectuée sans générer d'erreurs.
2. **Tests invalides** : Programmes volontairement erronés, utilisés pour détecter des violations des règles contextuelles, comme des conflits de type ou des références à des identificateurs non déclarés.

4.3 Exemples de tests

Programme valide :

Ce programme définit une classe simple `A`, représentant une structure contenant un unique attribut entier `a`.

```
class A {
    int a;
}
```

Résultat attendu : aucune erreur signalée, et le programme est validé par l'analyse contextuelle.

Programme avec une erreur contextuelle : Manipulation de types et calculs conditionnels

Ce programme contient une erreur liée à l'utilisation incorrecte de types lors d'une opération arithmétique. En effet, la variable `isEnabled` de type `boolean` est utilisée dans une expression arithmétique, ce qui est interdit.

```
{
    int num = 10;
    float multiplier = 3.5;
    boolean isEnabled = false;
    float total;

    if (num > 5 && !isEnabled) {
        print("Les conditions initiales sont valides.");
    }

    total = num * multiplier + isEnabled;

    print("Valeur finale calculée : ", total);

    if (total > 30) {
        print("Le total dépasse 30.");
    } else {
        print("Le total est inférieur ou égal à 30.");
    }
}
```

Erreur attendue : Une erreur contextuelle doit être signalée sur la ligne suivante :

```
total = num * multiplier + isEnabled;
```

4.4 Résultat des tests

Les tests réalisés ont permis de confirmer que l'analyse contextuelle détecte correctement les erreurs liées aux déclarations, aux types, et à l'utilisation des identificateurs. Les

messages d'erreur générés sont précis et permettent une localisation rapide des anomalies dans le code.

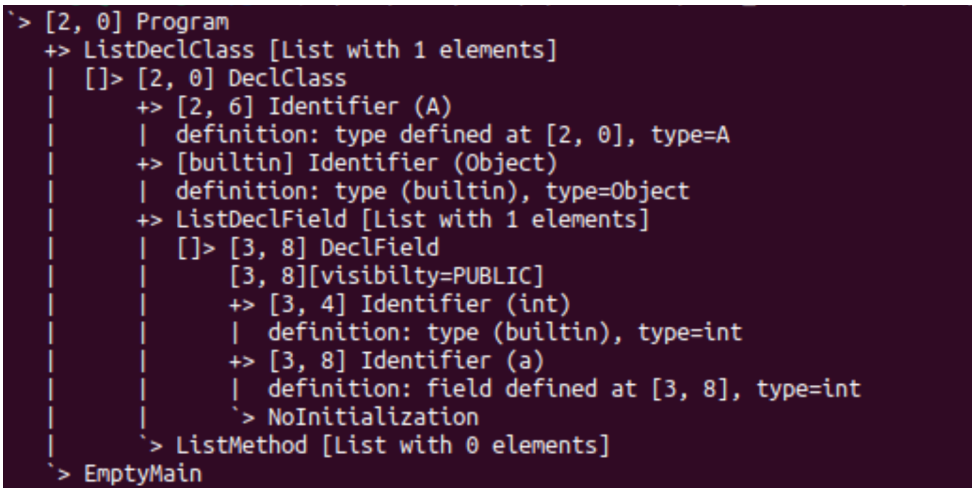

Figure 3: Arbre abstrait résultat du programme valide

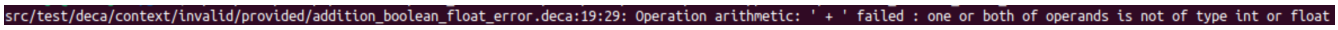

Figure 4: Message d'erreur résultat du programme invalide

4.5 Script de tests pour la partie B

Pour automatiser les tests de la partie B, un script a été généré afin de tester efficacement des programmes valides et invalides. Ce script permet de vérifier que l'analyse contextuelle produit les résultats attendus, en comparant les sorties avec les comportements spécifiés. Le script est situé dans le répertoire suivant :

`gl02/src/test/script/tests_context_perso.sh`

Ce fichier contient les commandes nécessaires pour exécuter les tests de l'analyse contextuelle.

Tests valides

Pour les tests valides, le script exécute la commande suivante pour chaque fichier source `.deca` contenant un programme syntaxiquement et contextuellement correct :

```
decac -v fichier_valide.deca
```

Résultat attendu : La commande `decac -v` ne produit aucune sortie, indiquant que le programme est valide du point de vue contextuel. Tout message généré par la commande est considéré comme une anomalie.

Tests invalides

Pour les tests invalides, le script exécute également la commande `decac -v`, mais spécifie la ligne exacte où l'erreur doit être détectée. Chaque fichier de test contient un programme volontairement erroné (par exemple, une variable non déclarée ou un conflit de type). Le script compare ensuite la sortie du compilateur avec les erreurs attendues.

Exemple de commande exécutée dans le script :

```
decac -v fichier_invalide.deca
```

Exemple de message d'erreur attendu pour un test invalide :

```
fichier_invalide.deca:5:12: Erreur contextuelle : variable non déclarée.
```

Résultats

Ce script, situé dans `gl02/src/test/script/tests_context_perso.sh`, a permis de valider efficacement le comportement de l'analyse contextuelle en testant un large éventail de cas. Les tests valides n'ont produit aucune sortie, confirmant leur conformité, tandis que les tests invalides ont généré des messages d'erreur localisés précisément, comme attendu.

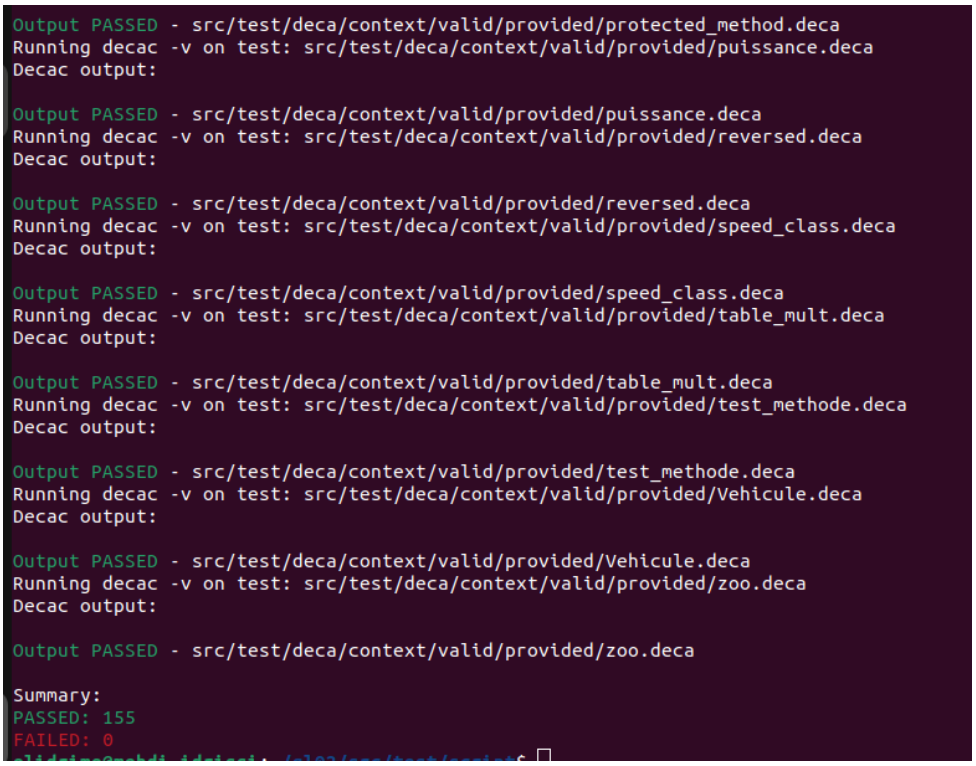

Figure 5: Résultat du script confirmant la réussite des tests

5 Tests de l'´etape C : Génération de code

5.1 Tests de génération de code pour sans objet

Pour valider la génération de code produite par notre compilateur, plusieurs tests ont été effectués, allant de cas simples à des scénarios plus complexes. Voici un résumé des tests réalisés et leur utilité.

5.1.1 Tests simples

Le premier test consiste à vérifier la génération de code pour des instructions de base, telles que l'initialisation de variables, des opérations arithmétiques et l'affichage de résultats. Voici un exemple de code source utilisé :

Listing 1: Test simple d'opérations arithmétiques

```
int x = 1;
int y = 2;
println(x - y);
```

Après validation du comportement pour ce type de code, un second test a été réalisé en introduisant des expressions arithmétiques plus complexes, nécessitant une gestion efficace des registres et de la pile. Le code suivant illustre ce cas :

Listing 2: Test avec une expression complexe

```
int x = 1;
int y = 2;
int z = 3;
println(x * x * x * x * z * y * x * x - y * z * 2);
```

Ce test a permis de vérifier que la pile est correctement utilisée pour stocker les valeurs intermédiaires, surtout lorsque le nombre de registres disponibles est limité (par exemple, à 4 registres).

5.1.2 Test de gestion de la pile avec des instructions conditionnelles

Pour tester la gestion de la pile dans des cas impliquant des conditions et des branchements, nous avons utilisé un code source avec des variables conditionnelles (`boolean`) et des instructions `if`. Voici un exemple :

Listing 3: Test avec gestion de la pile et des conditions

```
int a = 5;
int b = 10;
boolean c;
if (a > 2 && b < 20) {
    c = true;
    println("a marche");
}
```

Ce test vise à vérifier que les conditions sont correctement traduites en instructions de branchement, et que la pile est utilisée de manière cohérente pour évaluer les expressions logiques.

5.1.3 Tests avec des boucles et des structures complexes

Pour valider la gestion des boucles, des conditions imbriquées et des branchements multiples, des scénarios plus complexes ont été testés. Un exemple typique est le suivant :

Listing 4: Test avec boucles et conditions imbriquées

```
// Initialisation
```

```
boolean estVrai = true;
int a = 7;
int b = 8;
int x = 1;

// Condition simple
if (estVrai) {
    println("Vrai");
} else {
    println("Faux");
}

// Boucle avec condition
while (a > 3 && b <= 10) {
    a = a - 1;
    b = b + 1;
}
println(a);

// Boucle avec branchements multiples
while (x > 0) {
    if (false) {
        println("je suis l  ");
    } else {
        println("je ne suis plus l  ");
    }
    x = x - 1;
}

// R sultat attendu :
// Vrai
// 4
// je ne suis plus l
```

5.1.4 Analyse des résultats

Ces tests ont permis de vérifier plusieurs aspects de la génération de code, notamment :

- La gestion efficace des registres et de la pile pour les calculs complexes.
- La traduction correcte des conditions en instructions de branchement.
- La gestion des boucles imbriquées et des structures de contrôle.
- La conformité du comportement du code généré avec les résultats attendus.

5.2 Tests avec des classes et des classes héritées

Pour tester la génération de code orienté objet, nous avons commencé par des exemples simples de classes, puis nous avons introduit l'héritage avec des `class extends`, avant de

les relier au `main`. Voici quelques exemples concrets.

5.2.1 Exemple 1 : Classe simple avec une méthode d'accès

Le premier exemple illustre une classe simple `A` avec un attribut protégé `x` et une méthode `getX()` permettant d'accéder à sa valeur.

Listing 5: Classe `A` avec un attribut et une méthode d'accès

```
// Definition de la classe A
class A {
   protected int x;
   int getX() {
      return x;
   }
}

// Programme principal
{
    A a = new A();
    println("a.getX() =", a.getX());
}

// Sortie : a.getX() = 0
```

Ce test permet de vérifier que le compilateur génère correctement du code pour l'initialisation par défaut de l'attribut `x` (à `0`) et l'appel de la méthode `getX()`.

5.2.2 Exemple 2 : Classe avec une méthode modifiant un attribut

Dans cet exemple, la classe `A` contient un attribut `z` et une méthode `m()` qui modifie la valeur de cet attribut. Ce test permet de valider que les méthodes modifiant l'état interne d'une classe fonctionnent correctement.

Listing 6: Classe `A` avec une méthode modifiant un attribut

```
// Definition de la classe A
class A {
    int z = 6;
    void m(int a) {
        z = z + 1;
    }
}

// Programme principal
{
    A a = new A();
    a.m(1);
    println("a.z =", a.z);
}

// Sortie : a.z = 7
```

Ce test vérifie la gestion des méthodes avec des paramètres et l'effet des modifications d'attributs sur l'état de l'objet.

5.2.3 Exemple 3 : Classe héritée

Dans ce test, nous introduisons une classe `B` qui hérite de la classe `A`. Cela permet de tester la génération de code pour l'héritage et l'accès aux membres de la classe parente.

Listing 7: Classe héritée avec surcharge de méthode

```
// Classe de base
class A {
    protected int x;
    int getX() {
        return x;
    }
}

// Classe h rit e
class B extends A {
    void setX(int value) {
        x = value;
    }
}
```

Ce test permet de valider :

- L'accès aux attributs protégés de la classe parente.
- La gestion des méthodes héritées (`getX()`) et surchargées (`setX()`).

5.2.4 Conclusion

Ces tests montrent que le compilateur génère correctement du code pour :

- Les classes simples avec des attributs et des méthodes.
- Les modifications d'attributs via des méthodes.
- L'héritage, avec une gestion cohérente des méthodes et attributs des classes parentes et dérivées.

Ils constituent une base solide pour tester des scénarios plus complexes avec des classes et des objets.

5.3 Exemple de génération de code pour `Point2D` et `Point3D`

Cet exemple illustre la génération de code pour deux classes :

- `Point2D`, qui représente un point dans un plan avec des coordonnées `x` (abscisse) et `y` (ordonnée).
- `Point3D`, qui hérite de `Point2D` et ajoute une coordonnée `z` (hauteur).

Les méthodes `diag(int a)` permettent de déplacer les points en diagonale.

Listing 8: Classes `Point2D` et `Point3D` en Java

```
// Classe Point2D
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;
    }
}

// Classe Point3D
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;
    }
}
```

5.3.1 Code généré

Le code généré pour cet exemple inclut la gestion des tables des méthodes, l'initialisation des objets, et les implémentations des méthodes `diag` pour `Point2D` et `Point3D`.

Listing 9: Code généré pour `Point2D` et `Point3D`

```
; But du programme principal
Tables.DES.Methodes:
    LOAD #null, R0
    STORE R0, 1(GB)
    LOAD code.Object.equals, R0
    STORE R0, 2(GB)
; Construction de la table des methodes de Point2D
    LEA 1(GB), R0
    STORE R0, 3(GB)
    LOAD code.Object.equals, R0
    STORE R0, 4(GB)
    LOAD code.Point2D.diag, R0
    STORE R0, 5(GB)
; Construction de la table des methodes de Point3D
    LEA 3(GB), R0
    STORE R0, 6(GB)
```

```
    LOAD code.Object.equals, R0
    STORE R0, 7(GB)
    LOAD code.Point3D.diag, R0
    STORE R0, 8(GB)
; Programme principal
    WSTR "Il manque le main"
    HALT

initPoint2D:
    LOAD #0, R0
    LOAD -2(LB), R1
    STORE R0, 1(R1)
    LOAD #0, R0
    LOAD -2(LB), R1
    STORE R0, 2(R1)
    RTS

code.Point2D.diag:
    PUSH R2
    PUSH R3
    LOAD -2(LB), R2
    LOAD -2(LB), R3
    LOAD 1(R3), R3
    ADD -3(LB), R3
    STORE R3, 1(R2)
    LOAD -2(LB), R2
    LOAD -2(LB), R3
    LOAD 2(R3), R3
    ADD -3(LB), R3
    STORE R3, 2(R2)
    POP R3
    POP R2
    RTS

initPoint3D:
    LOAD -2(LB), R0
    PUSH R0
    BSR initPoint2D
    SUBSP #1
    LOAD #0, R0
    LOAD -2(LB), R1
    STORE R0, 3(R1)
    RTS

code.Point3D.diag:
    PUSH R2
    PUSH R3
    LOAD -2(LB), R2
```

```
    LOAD -2(LB), R3
    LOAD 1(R3), R3
    ADD -4(LB), R3
    STORE R3, 1(R2)
    LOAD -2(LB), R2
    LOAD -2(LB), R3
    LOAD 2(R3), R3
    ADD -4(LB), R3
    STORE R3, 2(R2)
    LOAD -2(LB), R2
    LOAD -2(LB), R3
    LOAD 3(R3), R3
    ADD -4(LB), R3
    STORE R3, 3(R2)
    POP R3
    POP R2
    RTS

code.Object.equals:
    HALT

; Fin du programme principal

stack_overflow_error:
    WSTR "Error: Stack Overflow"
    WNL
    ERROR

dereferencement_null:
    WSTR "Error: Null pointer dereference"
    WNL
    ERROR
```

5.3.2 Description du code généré

- **Tables des méthodes** : Les tables des méthodes sont construites pour les classes `Point2D` et `Point3D`, en associant chaque méthode à son adresse.
- **Initialisation des objets** : Les procédures `initPoint2D` et `initPoint3D` initialisent les champs des objets respectifs.
- **Méthodes** `diag` : Les méthodes `code.Point2D.diag` et `code.Point3D.diag` implémentent le déplacement en diagonale pour les instances des classes correspondantes.

Ce code vérifie le fonctionnement des tables de méthodes, l'héritage et les redéfinitions de méthode.

5.4 Test avec le script `test_codegen.sh`

Afin de tester la génération de code pour différents programmes, on a écrit un script shell `test_codegen.sh`. Ce script permet d'exécuter une série de tests automatisés pour vérifier le bon fonctionnement de la génération de code pour divers programmes. Chaque test correspond à un programme spécifique, et le script compile, génère le code et effectue les vérifications nécessaires pour valider les résultats.

Le script inclut des tests simples, comme ceux que nous avons précédemment abordés, ainsi que des programmes plus complexes. Il exécute chaque programme, puis vérifie la sortie attendue, en comparant les résultats obtenus avec les résultats prévus. Ce processus permet de s'assurer que la génération de code fonctionne correctement dans une variété de scénarios.

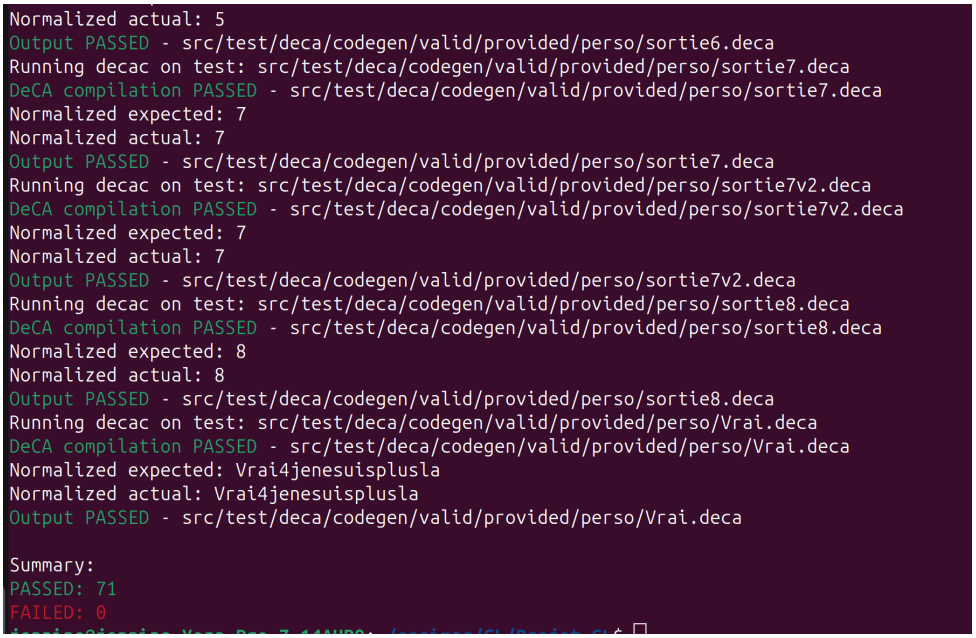

Figure 6: Résultat du script confirmant la réussite des tests

6 Conclusion

En conclusion, ce document a permis de structurer et de présenter les différentes étapes de validation du compilateur Deca. Les tests réalisés ont joué un rôle essentiel dans la détection et la correction des erreurs, garantissant ainsi la robustesse et la fiabilité des fonctionnalités implémentées.