ENSIMAG

Projet Génie Logiciel

Groupe 13

Validation

Réalisé par :

Benjelloun Otman
Birée Thomas
Boulouz Amine
El Goumiri Rida
Loginova Aleksandra

Année Universitaire : 2024/2025

Table des matières

1 Introduction

La validation est une étape importante et cruciale dans le développement de tout logiciel. Cela est particulièrement vrai lorsqu'il s'agit du développement d'un compilateur, où la fiabilité et la robustesse sont des préoccupations majeures à garder à l'esprit tout au long du processus. Cette phase de validation vise à garantir que le compilateur implémente fidèlement les fonctionnalités attendues, notamment la traduction correcte du code source en code cible. Cette tâche, qui représente une part importante de l'effort de développement, permet d'accroître la confiance dans la fiabilité et la robustesse du compilateur.

La phase de validation vise à identifier et corriger les défauts potentiels du code tout en assurant que le compilateur respecte les spécifications fonctionnelles et non fonctionnelles définies. Elle couvre l'ensemble des étapes du projet, qu'il s'agisse du fonctionnement isolé de chaque phase ou du comportement global du compilateur.

2 Descritptif des tests

2.1 Etape A : Analyse Syntaxique

Les tests pour cette étape étaient répartis en deux catégories principales, à savoir "***syntax/valid*** "et "***syntax/invalid***". Chacune de ces catégories est subdivisée en plusieurs sous-catégories, chacune testant une fonctionnalité spécifique du compilateur ou une "règle" définie dans la documentation du projet. Cependant, certains tests se trouvent directement dans la catégorie "*valid*" (ou "*invalid*") ; il s'agit de tests visant à vérifier une certaine combinaison de règles élémentaires que l'on souhaite évaluer.

Les objectifs des tests varient selon leur emplacement dans la hiérarchie des tests, mais se regroupent généralement en deux types complémentaires. Les premiers tests, centrés sur les fonctionnalités de base, visent à confirmer le bon fonctionnement de l'analyse syntaxique pour chaque règle définie dans la documentation. Ces tests assurent également que les modifications ultérieures n'ont pas compromis les fonctionnalités précédemment validées. Les seconds tests, orientés vers les combinaisons de règles élémentaires, évaluent la robustesse du compilateur face à des configurations complexes. Ces stress tests permettent de vérifier que la validation syntaxique reste fiable, même dans des situations impliquant des interactions entre plusieurs blocs de code.

2.2 Etape B : Vérification contextuelle

En ce qui concerne la structure des tests et leurs types, celle adoptée dans la partie B est identique à celle de la partie A. Cette structure ne prend pas en compte la subdivision de la vérification contextuelle en trois passes, mais se concentre plutôt sur la subdivision du langage **DECA** en termes de règles contextuelles. Cette phase joue le rôle d'un pont entre l'analyse syntaxique et la génération de code. Ainsi, les tests visent à vérifier qu'un code **DECA** , bien que syntaxiquement correct, respecte également les règles contextuelles du langage. Pour ce faire, les tests visent, par exemple, la cohérence des types dans les expressions, la validité des appels de méthodes, la portée des variables, etc. Cela permet de détecter des erreurs subtiles qui auraient pu passer inaperçues dans l'analyse syntaxique. Ces tests renforcent la fiabilité du compilateur en vérifiant non seulement la structure du code, mais aussi sa logique et sa sémantique.

2.3 Etape C : Génération de code

La structure adoptée reste la même dans cette partie aussi.

La génération de code est la dernière phase de la compilation. À ce stade, toute l'analyse du code **DECA** a été réalisée. Les tests, qu'ils soient élémentaires ou plus généraux, ont pour objectif principal de vérifier que le code assembleur généré respecte bien les spécifications. Si cette génération est correcte, le code est exécuté et une validation du résultat est effectuée pour s'assurer de son bon fonctionnement.

Les tests visent à soumettre le compilateur à différentes combinaisons de blocs **DECA** , en incluant des scénarios où des problèmes peuvent apparaître à cause des limitations techniques de l'architecture cible. Un script **DECA** peut être correct d'un point de vue syntaxique, contextuel et même en termes de génération d'assembleur, mais il peut dépasser les capacités de l'architecture visée. Par exemple, un script de calcul peut être bien conçu, mais un débordement dû à des valeurs trop grandes peut empêcher l'exécution du programme, même si le code est parfait du point de vue du langage.

3 Les scripts de test

Le principe des scripts de test est le même pour toutes les parties. Pour chaque partie, le script approprié exécute le "launcher" correspondant sur tous les tests présents dans le répertoire dédié. Les sorties de ces tests sont ensuite enregistrées dans un sous-répertoire nommé ***"test_out"***.

Ensuite, le script compare les résultats obtenus avec les résultats attendus, stockés dans le répertoire ***"verif"***. Il affiche alors un message du type **PASSED** (ce qui nous fait plaisir :D) ou **FAILED** (ce qu'on n'aime pas beaucoup), accompagné d'un message détaillé indiquant le résultat attendu et celui réellement obtenu. Pour les tests "invalid", le message affiché est plutôt du type **Error correctly caught** ou **Mismatch in expected error**.

En général, si c'est **vert**, tout va bien. Si c'est **rouge**... pas vraiment.

3.1 Partie A

Le script utilisé est **"syntax.sh"** :

```
#!/bin/bash
cd "$(dirname "$0")"/../../.. || exit 1

PATH=./src/test/script/launchers:"$PATH"

RED='\e[31m'
GREEN='\e[32m'
YELLOW='\e[33m'
BLUE='\e[34m'
NC='\e[0m'

# create test_out diretory (and all subdirectories) if it does not
   ↪ already
# exist
mkdir -p test_out/test/deca/syntax/valid/provided
mkdir -p test_out/test/deca/syntax/valid/parallel_compile
```

```
mkdir -p test_out/test/deca/syntax/valid/print
mkdir -p test_out/test/deca/syntax/valid/class
mkdir -p test_out/test/deca/syntax/invalid/provided
mkdir -p test_out/test/deca/syntax/invalid/class

valid_test_nb=0
valid_err_nb=0

syntax_test_valid () {
    echo -e "Syntax test for ${BLUE}VALID${NC} source files"

    for deca_source in $(find src/test/deca/syntax/valid -type f -
       ↪name "*.deca")
    do
        passed=true
        ((valid_test_nb=valid_test_nb+1))

        # information on test execution
        comparison_possible=false

        # output files
        synt_lis="${deca_source%.deca}"-synt.lis
        synt_lis="${synt_lis/src/test_out}"

        synt_res="${deca_source%.deca}"-synt.res
        synt_res="${synt_res/src/test_out}"

        # comparison file
        expected_tree="${deca_source%.deca}"-verif.lis
        expected_tree="${expected_tree/src/verif}"

        test_synt "$deca_source" | grep -v "DEBUG" 1> "$synt_lis"
           ↪2> "$synt_res"

        # unexpected syntax errors
        if [ -s "$synt_res" ]; then
            echo -e "${RED}Unexpected syntax error in file:${NC} $
               ↪{deca_source}"
            echo -e "${RED}test_synt output:${NC}"
            cat "${synt_res}"
            echo -e "\n"
            passed=false
            ((valid_err_nb=valid_err_nb+1))

        # Comparison of the generated syntax tree with the
           ↪expected one
        elif [ -f "$expected_tree" ]; then
            comparison_possible=true
            if ! diff -q "$synt_lis" "$expected_tree" > /dev/null;
               ↪ then
                echo -e "${RED}Mismatch in syntax tree for file:${
```

```
                    ↪NC} ${deca_source}"
                echo -e "Expected:"
                cat "$expected_tree"
                echo -e "Got:"
                cat "$synt_lis"
                echo -e "\n"
                passed=false
                ((valid_err_nb=valid_err_nb+1))
            fi
        # else
        #   echo -e "${YELLOW}Missing expected syntax tree file
            ↪for:${NC} ${deca_source}"
        fi

        # print individual test result
        if [ "$passed" = true ]; then
            echo -e -n "${GREEN}PASSED: ${NC}"
        else
            echo -e -n "${RED}FAILED: ${NC}"
        fi

        echo -e -n "${deca_source} "
        echo -e -n "[Syntax tree comparison: "
        if [ "$comparison_possible" = true ]; then
            echo -e  "${GREEN}YES${NC}]"
        else
            echo -e  "${YELLOW}NO${NC}]"
        fi

    done

    if ((valid_err_nb > 0)) then
        echo -e "\n${RED}FAILED${NC} Syntax test for ${BLUE}VALID$
            ↪{NC} source files"
        echo -e "${RED}${valid_err_nb}${NC}/${valid_test_nb} tests
            ↪ failed. \n"
    else
        echo -e "${GREEN}PASSED${NC} Syntax test for ${BLUE}VALID$
            ↪{NC} source files \n"
        echo -e "All ${GREEN}${valid_test_nb}${NC} tests passed. \
            ↪n"
    fi
}

invalid_test_nb=0
invalid_err_nb=0

syntax_test_invalid () {
    echo -e "Syntax test for ${BLUE}INVALID${NC} source files"

    for deca_source in $(find src/test/deca/syntax/invalid -type f
```

```
        ↪ -name "*.deca")
    do
        passed=true
        ((invalid_test_nb=invalid_test_nb+1))

        # output file
        synt_res="${deca_source%.deca}"-synt.res
        synt_res="${synt_res/src/test_out}"

        # comparison file
        expected_error="${deca_source%.deca}"-verif.res
        expected_error="${expected_error/src/verif}"

        test_synt "$deca_source" 1> /dev/null 2> "$synt_res"

        if [ -f "$expected_error" ]; then
            if ! diff -q "$synt_res" "$expected_error" > /dev/null
               ↪; then
                echo -e "${RED}Mismatch in expected syntax error
                   ↪for file:${NC} ${deca_source}"
                echo -e "Expected:"
                cat "$expected_error"
                echo -e "Got:"
                cat "$synt_res"
                echo -e "\n"
                passed=false
                ((invalid_err_nb=invalid_err_nb+1))
            else
                echo -e "${GREEN}Syntax error correctly caught in
                   ↪file:${NC} ${deca_source}"
            fi
        else
            echo -e "${RED}Missing expected error file for:${NC} $
               ↪{deca_source}"
            passed=false
            ((invalid_err_nb=invalid_err_nb+1))
        fi
    done

    if ((invalid_err_nb > 0)) then
        echo -e "\n${RED}FAILED${NC} Syntax test for ${BLUE}
           ↪INVALID${NC} source files"
        echo -e "${RED}${invalid_err_nb}${NC}/${invalid_test_nb}
           ↪tests failed. \n"
    else
        echo -e "${GREEN}PASSED${NC} Syntax test for ${BLUE}
           ↪INVALID${NC} source files \n"
        echo -e "All ${GREEN}${invalid_test_nb}${NC} tests passed.
           ↪ \n"
    fi
}
```

```
syntax_test_valid
syntax_test_invalid
```

3.2 Partie B

Le script utilisé est **"context.sh"** :

```
#!/bin/bash
cd "$(dirname "$0")"/../../.. || exit 1

PATH=./src/test/script/launchers:"$PATH"

RED='\e[31m'
GREEN='\e[32m'
YELLOW='\e[33m'
BLUE='\e[34m'
NC='\e[0m'

# create test_out diretory (and all subdirectories) if it does not
   ↪ already
# exist
mkdir -p test_out/test/deca/context/invalid/assign
mkdir -p test_out/test/deca/context/invalid/initialization
mkdir -p test_out/test/deca/context/invalid/provided
mkdir -p test_out/test/deca/context/invalid/object
mkdir -p test_out/test/deca/context/valid/assign
mkdir -p test_out/test/deca/context/valid/initialization
mkdir -p test_out/test/deca/context/valid/provided
mkdir -p test_out/test/deca/context/valid/class
mkdir -p test_out/test/deca/context/valid/object

valid_test_nb=0
valid_err_nb=0

context_test_valid () {
    echo -e "Context test for ${BLUE}VALID${NC} source files"

    for deca_source in $(find src/test/deca/context/valid -type f
       ↪-name "*.deca")
    do
        passed=true
        ((valid_test_nb=valid_test_nb+1))

        # information on test execution
        comparison_possible=false

        # output files
        cont_lis="${deca_source%.deca}"-cont.lis
        cont_lis="${cont_lis/src/test_out}"
```

```
cont_res="${deca_source%.deca}"-cont.res
cont_res="${cont_res/src/test_out}"

# comparison file
expected_tree="${deca_source%.deca}"-verif.lis
expected_tree="${expected_tree/src/verif}"

test_context "$deca_source" | grep -v "DEBUG" 1> "
   ↪$cont_lis" 2> "$cont_res"

# unexpected context errors
if [ -s "$cont_res" ]; then
    echo -e "${RED}Unexpected context error in file:${NC}
       ↪${deca_source}"
    echo -e "${RED}test_context output:${NC}"
    cat "${cont_res}"
    echo -e "\n"
    passed=false
    ((valid_err_nb=valid_err_nb+1))

# Comparison of the generated context tree with the
   ↪expected one
elif [ -f "$expected_tree" ]; then
    comparison_possible=true
    if ! diff -q "$cont_lis" "$expected_tree" > /dev/null;
       ↪ then
        echo -e "${RED}Mismatch in context tree for file:$
           ↪{NC} ${deca_source}"
        echo -e "Expected:"
        cat "$expected_tree"
        echo -e "Got:"
        cat "$cont_lis"
        echo -e "\n"
        passed=false
        ((valid_err_nb=valid_err_nb+1))
    fi
# else
#   echo -e "${YELLOW}Missing expected context tree file
   ↪for:${NC} ${deca_source}"
fi

# print individual test result
if [ "$passed" = true ]; then
    echo -e -n "${GREEN}PASSED: ${NC}"
else
    echo -e -n "${RED}FAILED: ${NC}"
fi

echo -e -n "${deca_source} "
echo -e -n "[Context tree comparison: "
if [ "$comparison_possible" = true ]; then
```

```
            echo -e  "${GREEN}YES${NC}]"
        else
            echo -e  "${YELLOW}NO${NC}]"
        fi

    done

    if ((valid_err_nb > 0))
    then
        echo -e "\n${RED}FAILED${NC} Context test for ${BLUE}
           ↪VALID${NC} source files"
        echo -e "${RED}${valid_err_nb}${NC}/${valid_test_nb} tests
           ↪ failed. \n"
    else
        echo -e "${GREEN}PASSED${NC} Context test for ${BLUE}
           ↪VALID${NC} source files \n"
        echo -e "All ${GREEN}${valid_test_nb}${NC} tests passed. \
           ↪n"
    fi
}

invalid_test_nb=0
invalid_err_nb=0

context_test_invalid () {
    echo -e "Context test for ${BLUE}INVALID${NC} source files"

    for deca_source in $(find src/test/deca/context/invalid -type
       ↪f -name "*.deca")
    do
        passed=true
        ((invalid_test_nb=invalid_test_nb+1))

        # output file
        cont_res="${deca_source%.deca}"-cont.res
        cont_res="${cont_res/src/test_out}"

        # comparison file
        expected_error="${deca_source%.deca}"-verif.res
        expected_error="${expected_error/src/verif}"

        test_context "$deca_source" 1> /dev/null 2> "$cont_res"

        if [ -f "$expected_error" ]; then
            if ! diff -q "$cont_res" "$expected_error" > /dev/null
               ↪; then
                echo -e "${RED}Mismatch in expected context error
                   ↪for file:${NC} ${deca_source}"
                echo -e "Expected:"
                cat "$expected_error"
                echo -e "Got:"
```

```
                cat "$cont_res"
                echo -e "\n"
                passed=false
                ((invalid_err_nb=invalid_err_nb+1))
            else
                echo -e "${GREEN}Context error correctly caught in
                   ↪ file:${NC} ${deca_source}"
            fi
        else
            echo -e "${RED}Missing expected error file for:${NC} $
               ↪{deca_source}"
            passed=false
            ((invalid_err_nb=invalid_err_nb+1))
        fi
    done

    if ((invalid_err_nb > 0))
    then
        echo -e "\n${RED}FAILED${NC} Context test for ${BLUE}
           ↪INVALID${NC} source files"
        echo -e "${RED}${invalid_err_nb}${NC}/${invalid_test_nb}
           ↪tests failed. \n"
    else
        echo -e "${GREEN}PASSED${NC} Context test for ${BLUE}
           ↪INVALID${NC} source files \n"
        echo -e "All ${GREEN}${invalid_test_nb}${NC} tests passed.
           ↪ \n"
    fi
}

context_test_valid
context_test_invalid
```

3.3 Partie C

Le script utilisé est "codegen.sh" :

```
#!/bin/bash

cd "$(dirname "$0")"/../../.. || exit 1

PATH=./src/test/script/launchers:"$PATH"

RED='\e[31m'
GREEN='\e[32m'
YELLOW='\e[33m'
BLUE='\e[34m'
NC='\e[0m'

# create test_out diretory (and all subdirectories) if it does not
   ↪ already
```

```
# exist
mkdir -p test_out/test/deca/codegen/valid
mkdir -p test_out/test/deca/codegen/valid/arithmetique
mkdir -p test_out/test/deca/codegen/valid/assign
mkdir -p test_out/test/deca/codegen/valid/bool
mkdir -p test_out/test/deca/codegen/valid/if
mkdir -p test_out/test/deca/codegen/valid/print
mkdir -p test_out/test/deca/codegen/valid/provided
mkdir -p test_out/test/deca/codegen/valid/unary-minus
mkdir -p test_out/test/deca/codegen/valid/value
mkdir -p test_out/test/deca/codegen/valid/while
mkdir -p test_out/test/deca/codegen/valid/unary-minus
mkdir -p test_out/test/deca/codegen/perf/provided
mkdir -p test_out/test/deca/codegen/valid/assign
mkdir -p test_out/test/deca/codegen/valid/class
mkdir -p test_out/test/deca/codegen/valid/cast
mkdir -p test_out/test/deca/codegen/valid/instanceof

mkdir -p test_out/test/deca/codegen/invalid
mkdir -p test_out/test/deca/codegen/invalid/if
mkdir -p test_out/test/deca/codegen/invalid/arithmetiques
mkdir -p test_out/test/deca/codegen/invalid/logiques
mkdir -p test_out/test/deca/codegen/invalid/loops

valid_test_nb=0
valid_err_nb=0

codegen_test_valid () {
    echo -e "IMA codegen test for ${BLUE}VALID${NC} source files"

    for deca_source in $(find src/test/deca/codegen/valid -type f
       ↪-name "*.deca")
    do
        passed=true
        ((valid_test_nb=valid_test_nb+1))

        # information on test execution
        comparison_possible=false
        execution_verification=false

        # output files
        codegen_out="${deca_source%.deca}".ass

        codegen_res="${deca_source%.deca}"-codegen.res
        codegen_res="${codegen_res/src/test_out}"

        ima_res="${deca_source%.deca}".out
        ima_res="${ima_res/src/test_out}"

        # comparison file
        expected_ass="${deca_source%.deca}"-verif.ass
```

```
expected_ass="${expected_ass/src/verif}"

# execution verification file
expected_out="${deca_source%.deca}"-verif.out
expected_out="${expected_out/src/verif}"

decac "$deca_source" 2> "$codegen_res"

# unexpected syntax errors
if [ -s "$codegen_res" ]; then
    echo -e "${RED}Unexpected compilation error in file:${
       ↪NC} ${deca_source}"
    echo -e "${RED}decac output:${NC}"
    cat "${codegen_res}"
    echo -e "\n"
    passed=false
    ((valid_err_nb=valid_err_nb+1))

# Comparison of the generated IMA assembly code with the
   ↪expected one
elif [ -f "$expected_ass" ]; then
    comparison_possible=true
    if ! diff -q "$codegen_out" "$expected_ass" > /dev/
       ↪null; then
        echo -e "${RED}Mismatch in IMA assembly code for
           ↪file:${NC} ${deca_source}"
        echo -e "Expected:"
        cat "$expected_ass"
        echo -e "Got:"
        cat "$codegen_out"
        echo -e "\n"
        passed=false
        ((valid_err_nb=valid_err_nb+1))
    fi
# else
#   echo -e "${YELLOW}Missing expected IMA assembly code
   ↪file for:${NC} ${deca_source}"
fi

# Verification of IMA virtual machine execution
ima "$codegen_out" > "$ima_res"
if [ -f "$expected_out" ] && [ "$passed" = true ]; then
    execution_verification=true
    if ! diff -q "$ima_res" "$expected_out" > /dev/null;
       ↪then
        echo -e "${RED}Mismatch in IMA VM exeuction output
           ↪ for file:${NC} ${deca_source}"
        echo -e "Expected:"
        cat "$expected_out"
        echo -e "Got:"
        cat "$ima_res"
```

```
                echo -e "\n"
                passed=false
                ((valid_err_nb=valid_err_nb+1))
            fi
        # else
        #     echo -e "${YELLOW}Missing expected IMA VM output
           ↪file for:${NC} ${deca_source}"
        fi

        # print individual test result
        if [ "$passed" = true ]; then
            echo -e -n "${GREEN}PASSED: ${NC}"
        else
            echo -e -n "${RED}FAILED: ${NC}"
        fi

        echo -e -n "${deca_source} "
        echo -e -n "[IMA assembly comparison: "
        if [ "$comparison_possible" = true ]; then
            echo -e -n "${GREEN}YES${NC}]"
        else
            echo -e -n "${YELLOW}NO${NC}]"
        fi

        echo -e -n " [IMA VM output comparison: "
        if [ "$execution_verification" = true ]; then
            echo -e "${GREEN}YES${NC}]"
        else
            echo -e "${YELLOW}NO${NC}]"
        fi

    done

    if ((valid_err_nb > 0)) then
        echo -e "\n${RED}FAILED${NC} IMA codegen test for ${BLUE}
           ↪VALID${NC} source files"
        echo -e "${RED}${valid_err_nb}${NC}/${valid_test_nb} tests
           ↪ failed. \n"
    else
        echo -e "${GREEN}PASSED${NC} IMA codegen test for ${BLUE}
           ↪VALID${NC} source files \n"
        echo -e "All ${GREEN}${valid_test_nb}${NC} tests passed. \
           ↪n"
    fi
}

invalid_test_nb=0
invalid_err_nb=0

codegen_test_invalid () {
    echo -e "Codegen test for ${BLUE}INVALID${NC} source files"
```

```
for deca_source in $(find src/test/deca/codegen/invalid -type
   ↪f -name "*.deca")
do
    passed=false
    ((invalid_codegen_nb=invalid_codegen_nb+1))

    # Output file
    codegen_out="${deca_source%.deca}".ass

    codegen_res="${deca_source%.deca}-codegen.res"
    codegen_res="${codegen_res/src/test_out}"

    ima_out="${deca_source%.deca}".out
    ima_out="${expected_out/src/test_out}"

    # Expected error file
    expected_codegen_error="${deca_source%.deca}-verif.res"
    expected_codegen_error="${expected_error/src/verif}"

    expected_ima_out="${deca_source%.deca}"-verif.out
    expected_ima_out="${expected_out/src/verif}"
    # Run code generation and redirect error output
    decac "$deca_source" 2> "$codegen_res"

    if [ -f "$expected_codegen_error" ]; then
        if ! diff <(sed 's|^src/||' "$expected_codegen_error")
           ↪ <(sed 's|^.*/src/|src/|' "$codegen_res"); then
        # if ! grep -qF "$(cat "$expected_codegen_error" > /
           ↪dev/null)" "$codegen_res"; then
            echo -e "${RED}Mismatch in expected codegen error
               ↪for file:${NC} ${deca_source}"
            echo -e "Expected:"
            cat "$expected_codegen_error"
            echo -e "Got:"
            cat "$codegen_res"
            echo -e "\n"
            passed=false
            ((invalid_codegen_err_nb=invalid_codegen_err_nb+1)
               ↪)
        else
            echo -e "${GREEN}Codegen error correctly caught in
               ↪ file:${NC} ${deca_source}"
        fi
    else
        ima "$codegen_out" > "$ima_out"
        if [ -f "$expected_ima_out" ]; then
            if ! diff -q "$ima_out" "$expected_ima_out" > /dev
               ↪/null; then
                echo -e "${RED}Mismatch in expected ima error
                   ↪for file:${NC} ${deca_source}"
```

```
                    echo -e "Expected:"
                    cat "$expected_ima_out"
                    echo -e "Got:"
                    cat "$ima_out"
                    echo -e "\n"
                    passed=false
                    ((invalid_codegen_err_nb=
                       ↪invalid_codegen_err_nb+1))
                else
                    echo -e "${GREEN}Codegen error correctly
                       ↪caught in file:${NC} ${deca_source}"
                fi
            else
                echo -e "${RED}Missing expected error file for:${
                   ↪NC} ${deca_source}"
                passed=false
                ((invalid_codegen_err_nb=invalid_codegen_err_nb+1)
                   ↪)
            fi
        fi
    done

    if ((invalid_codegen_err_nb > 0))
    then
        echo -e "\n${RED}FAILED${NC} Codegen test for ${BLUE}
           ↪INVALID${NC} source files"
        echo -e "${RED}${invalid_codegen_err_nb}${NC}/${
           ↪invalid_codegen_nb} tests failed.\n"
    else
        echo -e "${GREEN}PASSED${NC} Codegen test for ${BLUE}
           ↪INVALID${NC} source files\n"
        echo -e "All ${GREEN}${invalid_codegen_nb}${NC} tests
           ↪passed.\n"
    fi
}

# Hi, if you are reading this, you are wondering: why? SO ARE WE.
# WHY MUST WE LIVE IN SUCH A WRETCHED WORLD? WHY MUST YOU BE LIKE
   ↪THIS, BASH??????
# (coridalement)
rm "./verif/test/deca/codegen/valid/not-verif.out"
echo yeeees > "./verif/test/deca/codegen/valid/not-verif.out"
codegen_test_valid

# just in case...
rm "./verif/test/deca/codegen/valid/not-verif.out"
echo yeeees > "./verif/test/deca/codegen/valid/not-verif.out"

codegen_test_invalid
# uhm... apparently it needs to be here too?
rm "./verif/test/deca/codegen/valid/not-verif.out"
```

```
echo yeeees > "./verif/test/deca/codegen/valid/not-verif.out"

#Well turns out it modifies a random test every time, WHYYYY BASH
   ↪WHYYYYY?? T-T
```

Remarque : pour des raisons que nous ignorons toujours, le script codegen.sh modifie systématiquement, à la fin de son exécution, un fichier de vérification pour un test valide choisi aléatoirement. Ce test apparaît ensuite comme FAILED, même s'il est en réalité valide. :(

3.4 Test général

Le test de tout le compilateur se fait en lançant : ***mvn test***, ou ***mvn clean compile test*** pour recompiler de nouveau. Concretement cela lance dans l'ordre `"syntax.sh"`, `"context.sh"` et `"codegen.sh"`.

3.5 Ajouter de nouveaux tests

L'ajout d'un test se découpe en trois étapes :

— **Étape 1 : création du test `.deca`**

Une fois le fichier `.deca` de vos rêves créé (celui qui fera battre votre cœur de développeur), il est temps de le placer dans son emplacement approprié dans la hiérarchie du projet.

Tous les tests se trouvent dans le répertoire ***"src/test/deca/<répertoire-convenable>"***, le répertoire convenable étant défini par la partie concernée par le test, son type (valide ou invalide) et, si nécessaire, sa sous-catégorie.

Par exemple, un test valide intitulé `"super-test-if.deca"` vérifiant le contexte d'une boucle `if` sera placé dans : ***"src/test/deca/context/valid/if"***.

— **Étape 2 : le fichier de vérification**

Nos scripts de test se basent sur une comparaison entre le résultat obtenu et un résultat attendu défini préalablement. Ce résultat attendu doit être créé par vous. Tous les fichiers de vérification se trouvent dans le répertoire : ***"verif/test/deca/<répertoire-convenable>"***.

La hiérarchie du répertoire ***"verif"*** doit être **IDENTIQUE** à celle du répertoire des tests, à l'exception des extensions des fichiers.

En reprenant l'exemple du test précédent, `"super-test-if.deca"`, il faut créer un fichier `"super-test-if-verif.lis"` dans : ***"verif/test/deca/context/valid"***.Ce fichier doit contenir l'arbre enrichi attendu.

Les extensions des fichiers de vérification sont les suivantes :

- `".lis"` : pour les arbres,
- `".res"` : pour les erreurs attendues dans le cas d'un test invalide,
- `".out"` : pour les tests valides concernant la partie C (génération de code), contenant la sortie attendue après compilation et exécution du code compilé.

— **Étape 3 : mise à jour du script de test**

Pour tout nouveau répertoire ajouté aux tests, il faut l'indiquer dans le script de test approprié en ajoutant une ligne du type :

```
mkdir -p test_out/test/deca/<reste-du-chemin>
```

En reprenant notre exemple, si l'on souhaite créer un répertoire pour des tests valides de boucles `if`, il faut ajouter dans le script la ligne suivante :

```
mkdir -p test_out/test/deca/context/valid/if
```

Cette commande crée la hiérarchie du répertoire ***"test_ out"***, où sont stockées les sorties des tests pour ensuite les comparer aux résultats attendus. Tout répertoire de tests n'ayant pas de sous-répertoire approprié dans ***"test_ out"*** sera ignoré par les scripts de test.

4 Résultats Jacoco

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		72%		55%	497	697	543	2,049	248	368	3	48
fr.ensimag.deca.tree		74%		65%	266	745	550	2,104	112	467	6	89
fr.ensimag.arm.pseudocode		0%		0%	99	99	207	207	80	80	26	26
fr.ensimag.deca		80%		81%	25	107	60	340	8	56	1	5
fr.ensimag.deca.context		79%		60%	60	175	62	298	33	127	1	22
fr.ensimag.ima.pseudocode		80%		64%	31	107	41	227	19	86	2	26
fr.ensimag.ima.pseudocode.instructions		67%		n/a	21	62	36	111	21	62	16	54
fr.ensimag.deca.codegen		80%		80%	8	28	11	54	7	23	0	2
fr.ensimag.deca.tools		90%		87%	2	17	4	42	1	13	0	3
Total	6,698 of 24,206	72%	573 of 1,446	60%	1,009	2,037	1,514	5,432	529	1,282	55	275

Nous avons une couverture totale de 72%, ce qui est en soi assez satisfaisant, bien que légèrement inférieur à notre objectif de 80%(qui a été quand même atteint pour `fr.ensimag.deca`). Cependant, cette couverture a été négativement impactée par certaines parties du code qui ne sont pas visitées, simplement parce que le compilateur n'a pas besoin de prendre ces chemins. Par exemple, nous sommes obligés de "Override" certaines méthodes dans les sous-classes d'une classe abstraite, comme certains getters ou setters que nous n'utilisons pas nécessairement, mais que nous devons définir pour que Java soit content.

Ainsi, dans certaines parties du code, les principes de polymorphisme ou d'abstraction, bien qu'ils soient bénéfiques pour la structure globale, entraînent l'existence de code inutile ou non couvert. Cela aurait pu être un problème à résoudre, mais toute tentative de correction risquerait de rendre la structure beaucoup plus complexe que nécessaire et plus difficile à gérer d'un point de vue développeur.

```
@Override
public ExpDefinition getExpDefinition() {
    try {
        return (ExpDefinition) definition;
    } catch (ClassCastException e) {
        throw new DecacInternalError(
                "Identifier "
                        + getName()
                        + " is not a Exp identifier, you can't call getExpDefinition on it");
    }
}
```

En outre, l'approche de "programmation défensive" adoptée nous conduit parfois à vérifier plusieurs fois certaines sections du code. Cela entraîne une duplication des vérifications, ce qui fait que les blocs "catch" des vérifications ultérieures, à l'exception

de la toute première, ne sont jamais atteints ou exécutés lors des tests effectués sur le compilateur.

```
public ParamDefinition getParamDefinition() {
    try {
        return (ParamDefinition) definition;
    } catch (ClassCastException e) {
        throw new DecacInternalError(
                "Identifier "
                        + getName()
                        + " is not a Param identifier, you can't call getParamDefinition on it");
    }
}
```

```
if (this.name == null) {
    throw new ContextualError("The identifier's name cannot be null.", getLocation());
}
```

5 Gestion des risques & rendus

5.1 Test

Étant des êtres humains, nous ne sommes pas parfaits. Bien que notre compilateur semble répondre aux spécifications demandées, il est impossible (au moins pour nous) de garantir qu'aucun cas complexe n'a été omis. La conception d'un compilateur est un projet exigeant, nécessitant une vérification rigoureuse.

Ainsi, le principal risque que nous identifions pour notre compilateur réside dans la possibilité que certains tests, malgré leur exhaustivité apparente, ne parviennent pas à détecter des erreurs subtiles ou des comportements imprévus. De telles lacunes pourraient entraîner des dysfonctionnements dans des cas d'usage spécifiques, compromettant la fiabilité de la compilation.

6 Autres méthodes de validation

Hormis la stratégie de test adoptée, nous nous sommes principalement focalisés sur une revue de code. L'idée était que la logique appliquée puisse être validée, même par des membres n'ayant pas forcément participé au développement de la partie en question, mais qui comprennent le principe général de fonctionnement.

Cela nous a permis, dans une certaine mesure, de tester la logique adoptée tout en bénéficiant d'une vision extérieure. Cette méthode s'est avérée très utile pour détecter des erreurs lors du débogage. En effet, lorsque l'on développe une partie spécifique d'un projet, on peut parfois être trop investi, trop concentré sur cette section, au point de tomber dans ce que l'on appelle une "vision en tunnel". Cela peut nous empêcher de remarquer certains problèmes, alors qu'un regard neuf et détaché est souvent capable de les détecter beaucoup plus facilement.