Projet Génie Logiciel

Gabriel Soria	**Meriem Lyna Safar Remali**	**Julian Gomez**
Camille Kasprzak	**Charles Breton**	**Arthur Lenne**
Alex Soubeyrand	**Maël Veyrat**	

Documentation de validation

Elaboration des tests

Pour s'assurer du bon fonctionnement du compilateur, il convient de faire une batterie de tests explorant un maximum de possibilités pour connaître les limites et les défauts du compilateur, ce qui permet d'apporter les corrections nécesssaires. Ces tests sont écrits en langage déca et les différents étapes de la compilation ainsi que l'exécution sont testées.

Les premiers tests passés par le compilateur sont des tests très basiques et sont dits fonctionnels. L'idée est de vérifier que le compilateur reconnait et accepte les bases contextuelles et syntaxiques du langage et qu'il sait compiler et exécuter des programmes basiques. Ces tests doivent toujours être validés et sont par conséquents repassés à l'implémentation de chaque nouvelle fonctionnalité du compilateur. Pour cela nous utilisons l'intégration continue sur Gitlab. A chaque nouvelle mise à jour du dépôt, les tests sont relancés et doivent être validés. S'ils ne le sont pas, la mise à jour est refusée.

Les tests syntaxiques permettent de valider que le compilateur reconnaît un programme dont la syntaxe est valide. Peu importe si celui-ci passera ou non l'étape de la compilation. Exemples de tests fonctionnels syntaxiques :

```
//test 0
/* test de programme vide */
```

✓

```
// test 1

{}
```

✓

```
// test 2

{;}
```

✓

```
// test 3
{
  int x = 5;
  println(-(5*7+83*8) );
}
```

✓

Tous les tests ci-dessus doivent être acceptés par l'analyseur syntaxique. Le reste des test syntaxiques se trouvent dans le répertoire /src/test/deca/syntax .
Exemples de tests fonctionnels contextuelles invalides :

```
{
println(x);
}
```

✗

Ce programme demande à afficher une variable non déclarée. La syntaxe est bonne mais l'analyseur contextuel doit refuser ce programme et nous indiquer que la variable x n'est pas déclarée.

```
{
int x ;
int y ;
int z ;
int y ;
z=1 ;

}
```

✗

Ici la variable y est déclarée deux fois et l'analyseur contextuelle ne doit pas accepter ce programme et signaler la double déclaration.

Le reste des tests contextuels se trouvent dans le répertoire /src/test/deca/context.

Exemples de tests de génération de codes valides :

```
{
print("hello ");
println(" world!");
}
```

Résultat :

```
hello world !
```

✓ La compilation fonctionne et l'affichage est bien celui attendu.

```
{
int x ;
x=1 ;
println("1=",x);
}
```

Résultat :

```
1=1
```

✓

La déclaration et l'affectation de la variable fonctionnent correctement.

Le reste des tests de génération de code et de compilation sont contenus dans le répertoire /src/test/deca/codegen .

Nous devions maintenant créer des tests pouvant vérifier des fonctionnalités plus complexes. Il y a eu différentes démarches dans l'élaboration de ces tests. Certains ont été écrits lors de l'implémentation d'une fonctionnalité, afin de vérifier son fonctionnement basique. Ces tests ne sont pas nécessairement répertoriés et la plupart ont disparus car beaucoup ne sont pas nécessaires à la compréhension du compilateur.

D'autres tests ont été construits suite à l'importation d'une librairie. Les tests construits à partir des fonctionnalités apportées par la librairie nous ont permis de voir ce que le compilateur ne supportait pas et ainsi, d'améliorer celui-ci.

Enfin, certains tests ont été élaborés en aveugle en se basant sur nos connaissances des autres langages de programmation.

Organisation des tests

Les tests de lexique sont lancés grace au script `basic_lex.sh`, seul certains tests invalides ont été ajoutés aux tests déjà présents. Les tests valides, quant à eux, sont directement mis dans le répertoire `deca/codegen/valide/sansObjets`. De même, certains tests invalides de context ont été ajoutés et sont exécutés grâce au script `basic-context.sh`. Les tests invalides permettent de vérifier le bon fonctionnement du gestionnaire d'erreurs.

Concernant les tests de fonctionnalitées, le répertoire `deca/codegen/valide/sansObjets` permet de valider les opérations arithmétiques et booléennes, de lecture et d'écriture, ainsi que les boucles conditionnelles. Le script `renduInter.sh`, compile et interprête les tests du répertoire `sansObjets` (valid et invalid), ainsi que les options (-p, -P, -v, -r). Ce script permet donc de valider le rendu intermédiaire.

La démarche pour déterminer si une fonctionnalitées est bien valide est de multiplier les tests (white box, black box). Ces tests sont implémentés de manière à couvrir tous les cas possibles (ou du moins le maximum de cas).

Gestion des risques et gestion des rendus

Pour gérer les risques, nous avons implementé comme demandé un système d'integration continue:

L'intégration continue (CI) désigne la pratique qui consiste à automatiser l'intégration des changements de code réalisés par plusieurs contributeurs dans un seul et même projet de développement. Il s'agit d'une bonne pratique DevOps, permettant aux développeurs de logiciels de merger (fusionner) fréquemment des changements de code dans un dépôt central où les builds et les tests s'exécutent ensuite.

Dans notre cas, lors d'un push ou d'un merge sur les branches **master** et **dev** les tests de compilation et les tests fonctionnels (de non regression) se lancent. Sur les autres branches, seulement un test de compilation est lancé.

Cela nous permet d'éviter d'avoir une version du compilateur non fonctionnelle sur la branche master mais aussi de voir si du code essentiel au fonctionnement du compilateur n'est pas modifié ou ecrasé par les nouvelles fonctionnalités des developpeurs.

Nos tests fonctionnels lancés par la CI sont décrit dans les scripts :

- basic-context.sh
- basic-decac.sh
- basic-gencode.sh
- basic-lex.sh
- basic-synt.sh
- common-tests.sh
- renduInitial.sh
- renduInter.sh

Concernant la gestion des rendus, l'implémentation d'un système de deploiement continue (CD) ne semble pas nécessaire pour le moment. La version fonctionnelle du compilateur se situe sur la branche **master**.

Remarque : il existe deux scripts en plus, mais ils sont seulement utiles aux développeurs. Le script clean.sh permet de supprimer tous les fichiers temporaires qui ne doivent surtout pas être push sur GIT. Pour finir, le script testAll.sh permet au développeur de lancer tous les tests cités au-dessus en simultanés. Cela permet de simuler le comportement du système d'intégration continue.

Création de nouveaux tests

Les tests se trouvent dans le répertoire /src/test/deca. Il y a 3 étapes pour compiler un programme deca. Ces trois étapes sont la vérification de la syntaxe du programme (/src/test/deca/sytntax), la vérification du contexte (/src/test/deca/context), et la génération de code (/src/test/deca/codegen). On trouve dans chacun de ces répertoires deux répertoires : valid et invalid. Certains tests sont créés pour vérifier le bon fonctionnement d'une fonctionnalité. D'autres sont là pour vérifier que le compilateur enverra un message d'erreur sur une mauvaise manipulation de l'utilisateur.

Intégration du code de plusieurs développeurs

Pour chaque nouvelle fonctionnalité, nous créons une nouvelle branche avec un nom qui résume ce que l'on fait sur cette branche. Nous fonctionnons avec un système de merge request. Nous avons une règle qui demande à ce que les merge requests soient validées par 2 développeurs ainsi que par le responsable qualité code avant de pouvoir les merger. Si c'est le responsable qualité code qui a fait la merge request, il faut que 3 développeurs aient validé la merge request. Cette procédure est appliqué en temps normal, lorsqu'il n'y a pas d'échéance proche. En période de rendu, il est nécessaire de merger toutes les branches annexes sur la branche principale master. Une fois les fonctionnalités terminées et les branches prêtes à être mergées, nous rassemblons les membres concernés par chacune des branches et effectuons les merges une à une jusqu'à ce que tout soit sur la branche master. Les derniers conflits et bugs sont résolus et nous pouvons passer aux fonctionnalités suivantes.

Tests de Decompile:

En ce qui concerne les tests de décompilation, nous avons choisi de comparer les arbres de synthèse plutôt que les fichiers de code source .deca. En effet, lors du processus de décompilation, il était fréquent d'observer de nombreuses différences au niveau des fichiers .deca qui n'avaient pas réellement d'impact sur les arbres de synthèse. Par exemple, des différences dans le parenthésage implicite et explicite pouvaient être observées. Cependant, ces différences rendaient la création de tests automatisés difficile, en particulier lors de la comparaison avec le fichier de vérification .expected en utilisant la commande « diff » sur Bash. Afin de simplifier cette comparaison, les informations de localisation ont été supprimées des arbres de synthèse attendus. Nous avons convenu que deux arbres de synthèse contenant les mêmes nœuds dans le même ordre sont considérés comme équivalents et on ignore d'autres différences peu influentes tel que le nombre d'espaces entre les mots.

Exemple d'arbre de synthese généré pour les tests:

```
`>  Program
   +> ListDeclClass [List with 0 elements]
   `>  Main
      +> ListDeclVar [List with 0 elements]
```

```
   `> ListInst [List with 1 elements]
      []>  IfThenElse
          +>  GreaterOrEqual
          |  +>  Int (1)
          |  `>  Int (0)
          +> ListInst [List with 1 elements]
          |  []>  Println
          |      `> ListExpr [List with 1 elements]
          |         []>  StringLiteral (ok)
          `> ListInst [List with 1 elements]
             []>  Println
                 `> ListExpr [List with 1 elements]
                    []>  StringLiteral (erreur)
```

Arbre de synthèse normal:

```
`> [10, 0] Program
   +> ListDeclClass [List with 0 elements]
   `> [10, 0] Main
      +> ListDeclVar [List with 0 elements]
      `> ListInst [List with 1 elements]
         []> [11, 3] IfThenElse
             +> [11, 7] GreaterOrEqual
             |  +> [11, 7] Int (1)
             |  `> [11, 12] Int (0)
             +> ListInst [List with 1 elements]
             |  []> [12, 6] Println
             |      `> ListExpr [List with 1 elements]
             |         []> [12, 14] StringLiteral (ok)
             `> ListInst [List with 1 elements]
                []> [14, 6] Println
                    `> ListExpr [List with 1 elements]
                       []> [14, 14] StringLiteral (erreur)
```