GL38

Manuel Utilisateur du compilateur Deca

Antoine Turkie, Augustin Chenevois, Aurélien Delory, Mihaja Razafimahefa, Sarah Ramaharobandro

Grenoble INP - Ensimag · Projet GL 2023-2024

Description du compilateur

Présentation des fonctionnalités

Nous avons globalement réalisé une grande partie du cahier des charges en termes de fonctionnalités. Après avoir implémenté “HelloWorld”, nous avons réalisé le langage sans objet qui implémente principalement un main avec des instructions sans classe. Puis nous avons implémenté le langage avec objet lors de la dernière semaine de projet.

Voici les fonctionnalités que nous avons réalisés :

Langage sans objet :
- Print des littéraux (Int, Float, String)
- Déclaration des types
- Affectation et gestion des expressions
- Opérations arithmétiques (Addition , Soustraction, Multiplication, Division entière, Division flottante, Modulo, Moins Unaire)
- Opération booléennes (==, !=, <, <=, >, >=, &&, ||, !)
- Comparaison des types
- Structure de contrôles (If/Else/While)
- Gestion de ConvFloat
- Gestion des commentaires
- Include
- Gestions Entrée-Sortie (ReadInt/ReadFloat)
- Cast (sans classes)
- Gestion d’erreurs (overflow, division par 0)

Langage avec objet :
- Déclaration des classes
- Gestion des extends
- Déclaration des fields
- Visibilité (Protected, Public(rien))
- Déclaration des méthodes
- Déclaration de paramètres
- Appel de méthodes
- Selection de fields
- Gestion de return
- Gestion de new
- Gestion des casts
- Gestion de Object (classe et méthode equals)
- Gestion de instanceof
- Gestion de this
- Gestion d’erreurs(déréférencement null, pas de return)

Toutes les fonctionnalités ont été établies en réalisant les étapes A (lexer et parser), B (vérifications contextuelles et décoration de l’arbre) et C (génération de code) ainsi qu’une multitude de tests pour votre confort.

Commandes et options

Nous avons réalisé toutes les options du cahier des charges :

`decac -p` permet d'arrêter decac après l'étape de construction de l'arbre abstrait et affiche la décompilation de ce dernier.
`decac -b` affiche notre magnifique bannière d'équipe
`decac -v` arrête decac après l'étape de vérification et produit une sortie uniquement en cas d'erreurs.
`decac -n` permet de supprimer les tests à l'exécution des règles 11.1 et 11.3.
`decac -r X` permet de choisir le nombre de registres disponibles entre 4 et 16
`decac -d` permet d'activer différents niveaux de débug
`decac -P` permet de lancer la compilation en parallèle de plusieurs fichiers deca

Nous avons rajouté une option pour activer notre extension : `decac -optim`.

Lancement des tests

Pour tester toutes nos fonctionnalités nous avons réalisé des scripts permettant de lancer tous nos tests lexicaux, syntaxiques, contextuels, codegen.
En utilisant le cycle de vie de Maven, `mvn test` lance ces scripts de test et vérifie que decac renvoie bien les sorties attendues en plus des tests de base.

Fonctionnalités non-implémentées

Nous espérons avoir réalisé la quasi-totalité des fonctionnalités. Cependant nous avons oublié d'implémenter l'overflow des flottants, dû à un oubli de notre part.

Messages d'erreurs

Erreurs syntaxiques et lexicographiques

Pour les erreurs lexicales, les erreurs sont du type : "token recognition error at" puisqu'il reconnaît pas le token.
Pour les erreurs syntaxiques, les erreurs sont du type :

- "no viable alternative at input" quand il ne trouve pas de règles de grammaire qui permettent de construire cette syntaxe.
- "Missing .. at …" quand il manque un token nécessaire à la grammaire
- "extraneous input … expecting …" quand il s'attend à un ensemble de token mais il en reçoit un en trop.
- "mismatched input … expecting …" quand il ne reconnaît pas le bon token (mais un autre)
- " left-hand side of assignment is not an lvalue" quand l'opérande de gauche n'est ni une sélection ni un identificateur.

Erreurs de syntaxes contextuelles

Pour la vérification contextuelle, nous disposons de plusieurs messages d'erreurs respectant les règles de l'étape B :

- "La règle 0.1 n'est pas respectée" : L'identifier n'appartenait pas à l'environnement des expressions. Par exemple, si l'identifier n'a pas été initialisé.
- "Le type ne respecte pas la règle 0.2" : Le type n'appartenait pas à l'environnement des types. Par exemple, si on utilise un type dont la classe n'existe pas et qui ne correspond pas aux types prédéfinis.
- "La superclass n'est pas dans l'environnement. La règle (1.3) n'est pas respectée." : Par exemple, si on hérite d'une classe non déclarée.
- "La superclass n'est pas un identificateur de classe. La règle (1.3) n'est pas respectée." : Par exemple, si on essaie d'hériter de int.
- "Il y a double définition de classe. La règle (1.3) n'est pas respectée" : Par exemple, si on déclare une classe déjà déclarée.
- "La règle (2.3) n'est pas respectée... Pas de Super Classe" : La super classe de la classe courante n'appartenait pas à l'environnement des types. Par exemple, si la superclasse qu'on essaie d'hériter n'est pas déclarée.
- "La classe utilisée n'a pas été déclarée, la règle (2.3) n'est pas respectée." : Par exemple, si la superclasse qu'on essaie d'hériter n'est pas déclarée.
- "L'identificateur de la super classe ne définit pas une classe, la règle (2.3) n'est pas respectée." : Par exemple, si on hérite d'un type qui n'est pas une classe.
- "Des attributs et des méthodes ont le même nom. La règle (2.3) n'est pas respectée." : Par exemple, si on déclare un attribut et une méthode ayant le même nom.
- "La règle (2.3) n'est pas respectée car la super classe n'est pas définie correctement." : La superclasse est null ou n'est pas une classe.
- "Double définition d'un champ : la règle (2.4) n'est pas respectée" : Le nom du champ existe déjà. Par exemple, si on déclare deux champs de même nom mais de type différents.
- "Field de type void. Cela ne respecte pas la regle 2.5". Par exemple, si on déclare un champ de type void.
- "Cette variable apparait dans la classe super mais n'est pas un field. Cela ne respecte pas la regle 2.5". Par exemple, si on déclare un champ de même nom qu'une méthode héritée de la superclasse.
- "Des méthodes ont été déclarées avec le même nom, la règle (2.6) n'est pas respectée" : Le nom de la méthode existe déjà. Par exemple, si on déclare un deux champs de même nom mais de type de retour différent.
- "L'identificateur de la méthode à déclarée existe dans la super classe mais ne définit pas une méthode : la règle (2.7) n'est pas respectée." : Par exemple, si on définit une méthode qui correspond à un champ de la superclasse.
- "La signature n'est pas la même et la règle (2.7) n'est pas respectée." : Par exemple, si on redéfinit une méthode mais qui n'a pas le même nombre de paramètres que la superclasse.
- "La redéfinition est mauvaise (problème de sous-type) et la règle (2.7) n'est pas respectée." : Par exemple, si le type de retour de la méthode redéfini n'est pas un sous-type du type de retour de la méthode de la superclasse.
- "Le type du paramètre est void : la règle (2.9) n'est pas respectée" : Par exemple, si on met en paramètre d'une méthode une variable de type void.

- “Paramètre déclaré deux fois : la règle (3.12) n'est pas respectée.” : Par exemple, si on définit deux paramètres de méthodes ayant le même nom.
- “Double définition d'un paramètre : la règle (3.13) n'est pas rescpetée.” : Le nom du paramètre qu’on déclare existe déjà. Par exemple, si on définit deux paramètres de même nom.
- “Une variable de type void est invalide : la règle (3.17) n'est pas respectée” : Par exemple, si on déclare un champ de type void.
- “Le symbole existe déjà : la règle (3.17) n'est pas respectée” : On déclare une variable dont le nom existe déjà.
- “Le type retourné par la fonction est nulle : la règle (3.24) n'est pas respectée.” : Par exemple, si on déclare une méthode qui ne renvoie rien mais qu’on ajoute tout de même l’instruction return.
- “La règle 3.28 n'est pas respectée : le type n'est pas compatible pour l'affectation” : Par exemple, on déclare une variable de type int mais on l’initialise à false.
- “Le paramètre de l'instruction n'est pas de type boolean : règle (3.29)” : Par exemple, la condition d’une structure de contrôle n’est pas de type booléenne.
- “Le type d'un parametre de print ne respecte pas la règle 3.30” : Il s’agit en réalité de la règle 3.31 mais nous avons mis 3.30. Par exemple, si on essaie d’afficher un objet ou un booléen.
- “Le type ne respecte pas la règle 3.33” : Par exemple, dans une opération ET, l’opérande à droite est de type entier.
- “Les opérandes d'un opérateur boolean doivent être de type boolean. Regle 3.33” : Par exemple, dans une opération ET, l’opérande à droite est de type entier.
- “Au moins un des opérandes de la comparaison n'est pas de type int ou float. Regle 3.33”. Par exemple, dans une opération <, l’opérande à droite est de type booléen.
- “Les types des opérandes de la comparaison ne sont pas compatibles. Regle 3.33” : Dans les comparaisons, si on essaie de comparer des types différents, hors conversion des int et float. Par exemple, si on compare une variable de type A avec une variable de type entier.
- “Le type ne respecte pas la règle 3.37” : Le type d’une opération unaire n’est pas respecté. Par exemple, si on fait l’opération Moins d’un booléen.
- “Le cast n'est pas valide et la règle 3.39 n'est pas respectée” : Si l’on souhaite faire un cast de deux classes mais aucun n’est sous-type de l’autre.
- “La regle 3.40 n'a pas ete respectee. Voir type_instance_of.” : Si l’on souhaite vérifier une instanceof et que la variable n’est pas une instance d’une classe. Par exemple, `i instanceOf A` avec i de type entier, ou `a instanceof int` avec a de type Objet.
- “La règle 3.42 n'a pas été respectée, le type n'est pas un type de classe.” : Par exemple, si l’on instancie avec une variable avec new mais le type instancié n’est pas une classe.
- “La regle 3.43 n'est pas respectee. CurrentClass est null” : Par exemple, si on utilise `this` dans Main
- “Cela ne respecte pas 3.65 et 3.66. La classe n'existe pas, cela n'est pas sensé se produire.” : Répétion d’une gestion d’erreur mais qui ne devrait pas s’afficher.
- “Cela ne respecte pas 3.65 et 3.66. Le type de l'expression n'est pas une classe.” Par exemple, dans une sélection où l’expression de gauche n’est pas une instance de classe.

- “Appel d'un champ protected dans le main : la règle 3.66 n'est pas respectée.” : Par exemple, si on appelle un champ de visibilité protected d’une classe dans le programme principal.
- “Cela ne respecte pas 3.66. Le type de l'expression n'est pas un sous-type du type de la classe actuelle.” : Par exemple, dans une sélection, on essaie d’accéder à un champ protégé en utilisant une instance de classe qui n’est pas une sous-classe de la classe où existe l’instance.
- Cela ne respecte pas 3.66. Le type de la classe actuelle n'est pas un sous-type de la class du field protected : même type d’erreur que ci-dessus.
- “La règle 3.71 n'est pas respectée, le type de l'expression n'est pas une classe” : Par exemple, si on accède une méthode à partir d’une variable qui n’est pas une instance de classe.
- “La règle 3.71 n'est pas respectée, le nombre de paramètres ne correspond pas avec la signature.” : Par exemple, il n’y a pas le même nombre de paramètres lors de l’appel d’une méthode.
- “La règle 3.71 n'est pas respectée, Un des paramètres n'a pas le bon type” : Erreur de signature de méthode lors de son appel.

Erreurs à l’exécution

Les erreurs relevées par ima lors de la génération de code sont constituées de :

- “La pile pleine” s’il y a débordement de la pile lors de l’exécution du programme
- “Le tas est plein” s’il y a débordement du tas avec les allocations de mémoire effectuées par le programme
- “Erreur de division par 0” s’il y a une expression qui utilise cette opération
- “Dépassement ou mauvais format” si l’entrée d’un readInt/readFloat est trop long ou n’est pas du type demandé
- “Une méthode de type de retour non void n'a rien retourné” lorsqu’il n’y a pas d’instruction return dans une méthode de type de retour non void
- “Déréférencement de null” lorsque le paramètre implicite qui sélectionne un champ ou appelle une méthode est null
- “Cast impossible” quand un cast est impossible

Présentation de l’extension

Nous avons choisi l’extension OPTIM qui implémente des techniques d’optimisation classiques utilisées par de vrais compilateurs. Elle ne nécessite aucune configuration préalable du fichier source et est activée lorsque l’option `-optim` est spécifiée. Le fichier résultante du compilateur reste au format .ass et est exécutable par la machine abstraite. Pour voir la différence de performance entre le code machine produit par `decac` sans l’option `-optim` et avec l’option, ima propose l’option `-s` pour mesurer le temps d’exécution propre de chaque fichier .ass.

Notre extension supporte toutes les fonctionnalités que nous avons développées, vérifiées par nos tests de non-régression. Elle ne comporte aucune limitation connue selon ces tests.