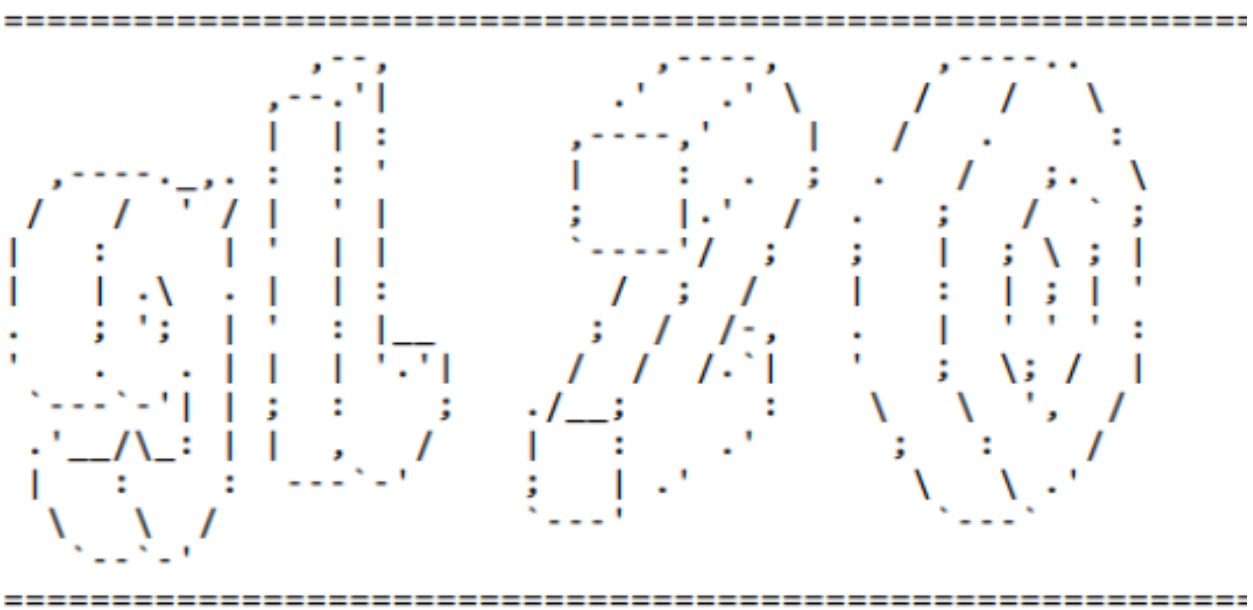
Documentation de Conception du compilateur Deca
- Groupe GL20



Melvin Bonamour
Steve Lening
Mathieu Meunier
Paul Grassi
Emmanuel Loroux

Table des matières

Langage deca sans objet

Analyse syntaxique :

Écriture d'un lexeur et d'un parseur avec ANTLR
ANTLR est un outil très puissant qui permet d'écrire de manière simplifiée les outils nécessaires à l'analyse syntaxique d'un langage. En effet, sans outil de ce genre, pour écrire un lexeur et un parseur il faudrait développer depuis le début un programme de parcours de fichier et de reconnaissance des lexèmes. ANTLR gère cette partie et permet de représenter visuellement les règles reconnues par une structure d'arbre, ce qui facilite grandement le développement. Ainsi, les règles syntaxiques du langage deca implémenté sont décrites dans les fichiers DecaLexer.g4 et DecaParser.g4, qui sont convertis en programme Java par l'outil ANTLR.
Le point d'entrée du parseur est la règle prog (fichier DecaParser.g4) qui parcourt d'abord la liste de déclaration des classes, et ensuite le programme principal. C'est le nœud racine de l'arbre de reconnaissance produit. Pour ajouter des règles, il est donc recommandé de parcourir l'arborescence depuis ce nœud. Il est également possible de rajouter des mots-clés au langage actuellement reconnu dans le fichier DecaLexer.g4.

Il est important dans les règles du parseur de s'assurer que deux conditions sont toujours remplis : le nœud renvoyé n'est pas null, pour pouvoir à l'étape au-dessus en récupérer l'attribut tree; et le nœud renvoyé est placé, ie son attribut Location est rempli, c'est primordial pour les étapes suivantes et pour le bon formatage des erreurs (exception : les listes de nœud peuvent ne pas être placées).
Pour étendre le compilateur aux aspects de deca non traités actuellement (par exemple tableau, gestion de modules, …), il faut ajouter les règles syntaxiques correspondantes dans ces fichiers. Pour plus d'information sur la syntaxe des fichiers .g4 : Documentation ANTLR.

Les pistes de maintenance / amélioration à privilégier sont :

- La distinction des différentes causes à un token **DEFAULT** pour produire des messages d'erreur plus pertinents;
- L'ajout de la reconnaissance de l'opération de **cast** et du test **instanceof** qui sont pour le moment non-reconnus par le compilateur.

Analyse contextuelle sans objet :

Implémentation de la grammaire contextuelle.
Les classes fournies de l'arbre implémentent des méthodes **verifyXYZ()** qui correspondent aux règles de la grammaire contextuelle de deca. Ces méthodes ont deux fonctions :

- comme leur nom le laisse entendre, l'appel à ces méthodes vérifient la syntaxe contextuelle des identifiants, expressions et instructions du langage (instructions placés dans le programme principal, types compatibles, etc). C'est à cette étape que le compilateur lève des erreurs contextuelles le cas échéant.

- De plus, c'est dans ce parcours de l'arbre que le compilateur le décore, c'est-à-dire assigne à chaque expression un type, à chaque variable une définition, etc. C'est aussi ici que des nœuds de conversion peuvent être ajoutés, par exemple **ConvFloat** dans une opération arithmétique entre un entier et un flottant. Pour plus de détails, voire environnement des types.

Pour la partie sans objet, l'arbre n'est parcouru qu'une fois et ce parcours commence dans le fichier Program.java.

Les méthodes **verify** ont été factorisées lorsque c'était possible dans les classes abstraites définissant plusieurs classes filles. C'est le cas par exemple pour les opérateurs arithmétiques (sauf le modulo car son comportement est différent). Par ailleurs, les expressions définissent plusieurs méthodes **verify** pour pouvoir vérifier une expression en tant que telle, en tant que condition (dans une structure while par exemple) ou en tant que type pour les **Identifier**.

Génération de code sans objet :

Implémentation de la génération de code assembleur (pour la génération de bytecode, voire la documentation spécifique à l'extension)
La génération de code se fait encore une fois en parcourant l'arbre des lexèmes produits et décorés par les étapes précédentes. Les instructions de cet arbre implémentent la méthode **codeGenInst()** et les expressions définissent en plus de celle-ci deux méthodes supplémentaires, **codeGenExpr()** et **codeGenCond()** pour des raisons similaires que les différentes méthodes **verify** de la vérification contextuelle. En fonction du contexte dans lequel se trouve l'expression (dans une condition de boucle, dans les paramètres d'un appel à print(), …), le code généré pour cette expression peut varier. Par exemple s'il s'agit d'une condition, il faut générer le code qui évalue la valeur de la condition, puis le code qui branche vers la bonne étiquette en fonction du résultat, étape que n'aurait pas fait un appel depuis une assignation par exemple ou l'on s'attend juste à ce que la valeur soit placée dans le registre courant.

La gestion des registres est faite dans une classe à part, **GenerationCode**, qui fournit un accesseur du premier registre libre et un moyen de signaler que l'on souhaite passer au prochain registre, ainsi qu'une vérification qu'un nouveau registre est bien libre. S'il n'y en a pas, c'est la méthode **codeGen** appelant le test qui doit s'occuper de sauvegarder le registre avant d'en écraser la valeur. L'option **-r** de la ligne de commande du compilateur agit sur ce fichier **GenerationCode** en modifiant la variable du maximum de registres (16 par défaut).
La classe **GenerationCode** permet également de générer les étiquettes qui correspondent aux différentes erreurs d'exécution qu'il est possible de rencontrer. C'est alors plus simple pour le reste de l'étape de génération de se brancher sur l'une de ces étiquettes si la situation provocante est détectée.

Pour générer des étiquettes de branchement pour les structures if / else, il faut s'assurer que chacune est unique pour que le code soit interprété correctement. Ainsi un système

d'indexation est mis en place dans la génération de code de ces structures pour satisfaire cette condition.

Langage deca avec objet

Jusqu'ici des squelettes de code étaient fournis et il s'agissait presque exclusivement de compléter les méthodes. Pour étendre le fonctionnement du compilateur au langage avec objet, il fallait créer de nouvelles classes dans le projet : de nouveaux nœuds de l'arbre et une classe regroupant les particularités de la classe **Object** de deca, à savoir sa déclaration et la méthode equals (déclaration et comportement). Par exemple, au début de la vérification contextuelle, on ajoute avant de commencer la vérification dans l'arbre la définition de la méthode equals dans l'environnement des expressions.
La liste des dépendances des classes implémentées est accessible en annexe : voir diagramme UML.

Environnement des types et des expressions :

L'environnement des types est défini dans la classe **EnvironmentType** et est rempli au début de l'exécution du compilateur avec les types prédéfinis du langage. Ensuite, la première passe de la vérification contextuelle mute cet environnement pour ajouter les objets **ClassType** qui correspondent aux types des classes définies par l'utilisateur du compilateur dans le fichier source.

L'environnement des expressions sert quant à lui à associer les noms de variables à leur définition. Celui-ci est implémenté dans le fichier **EnvironnementExp** et se base sur une instance d'une collection HashMap pour associer les symboles du compilateur (gérés dans **SymbolTable**) à la définition correspondante. Cette collection est performante dans notre cas d'usage car la notion de tri n'est pas nécessaire et l'accès se fait en temps constant. De plus, la classe **EnvironnementExp** est construite comme une liste simplement chaînée pour représenter l'opération d'empilement. Lorsque que la méthode get(Symbol) est appelée, le symbole est d'abord cherché dans l'environnement appelé, puis s'il n'est pas trouvé dans l'environnement parent. Cette notion est primordiale pour la notion de hiérarchie dans le langage deca objet.
Le fonctionnement de la décoration en détail, considérant le code suivant :

```
{
        int x;
        x = 4;
}
```

On considère la déclaration de variable. L'analyse syntaxique reconnaît ici deux **Identifier** ('int' et 'x') puis le point virgule. La vérification contextuelle se décompose alors selon la règle :

$$\textbf{decl\_var} \downarrow env\_types \downarrow env\_exp\_sup \downarrow env\_exp \downarrow class \uparrow \{name \mapsto (\underline{\text{var}}, type)\} \oplus env\_exp \quad (3.17)$$
$$\rightarrow \quad \underline{\texttt{DeclVar}}[$$
$$\textbf{type} \downarrow env\_types \uparrow type$$
$$\underline{\texttt{Identifier}} \uparrow name$$
$$\textbf{initialization} \downarrow env\_types \downarrow env\_exp/env\_exp\_sup \downarrow class \downarrow type$$
$$]$$
$$\textbf{\textit{condition}} \quad type \neq \underline{\text{void}}$$

On vérifie donc que le premier **Identifier** définit bien un type du langage. Pour ce faire, on récupère dans la table de symboles du compilateur le symbole associé au mot clé ‘int’, puis on récupère dans l’environnement des types la définition, qui renvoie ici la définition du type ‘int’. Ensuite, il faut ajouter la variable x à l’environnement d’expression pour associer à ce nom de variable son type. Un symbole est alors créé au niveau du compilateur, puis ce symbole est associé dans l’environnement local à une définition de variable de type ‘int’.
Ainsi, lors de l’analyse de l’assignation (deuxième expression du code exemple), le symbole créé ci-dessus est récupéré et on peut accéder dans l’environnement à la définition de la variable.

Vérification Contextuelle avec objet :

Le principe reste globalement identique ici que lors de la vérification sans la partie objet. La grande différence est qu’on parcourt ici l’arbre un total de trois fois (au lieu de une) pour tenir compte à chaque parcours de la décoration faite à l’étape d’avant. Pour expliquer ce qui implique la nécessité de ces trois passes, prenons l’exemple suivant :

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

class A{
        void a(){
                if (b()){
                        print(“a ok”);
                }
        }
        boolean b(){
                return true;
        }
}
```

Si l’on tente de vérifier d’une traite ce programme, plusieurs problèmes vont se poser :

- l’attribut a dans la classe AContainer est un champ de type non connu à ce moment car la vérification de la classe A n’a pas encore eu lieu;
- de la même manière l’appel à la méthode b() dans la métode a() dans la classe A va provoquer une erreur car celle-ci n’est pas encore connue.

Il y a donc besoin de passer trois fois sur le programme : une première fois pour ajouter à l'environnement des types les classes déclarées; une deuxième qui ajoute aux environnements de ces classes les déclarations de champs et de méthodes (avec leur signature); et enfin la troisième passe qui vérifie et décore le contenu du corps des méthodes, puis du programme principal.

Ainsi dans les classes concernées par la partie objet du langage, on trouve plusieurs méthodes **verify** mais la raison ici est que ces différentes méthodes correspondent à la passe à laquelle elles sont appelées.

Génération de code pour deca avec objet :

La génération de code avec objet est très similaire dans l'implémentation à celle du langage sans objet. Les algorithmes de génération et de gestion de la mémoire (champs, table des méthodes, …) sont ceux décrits dans le poly des spécifications techniques. Il n'y a pas eu ici de choix d'implémentation primordial. La principale différence réside dans l'utilisation au début de la génération de la classe **ObjectInternals**. En effet, la classe objet étant déclarée implicitement, la génération de la méthode equals notamment est faite manuellement avant de générer le code pour le fichier source.

Diagramme UML des classes implémentées (ajoutées à celles fournies)