Documentation de conception

Équipe 17

Anaïs Hadj-Azzem - Lucas Grellier - Théo Cachet
- Clément Vanhemelryck - César Dumas

Table des matières

1 États du compilateur

1.1 Vérification syntaxique

Le compilateur reconnaît les programmes déca selon la spécification qui nous a été fournie.
Il lève une erreur lorsque le flottant ou entier est trop grand, ou lorsque le flottant entré est trop proche de zéro.
En revanche, nous ne rattrapons pas les erreurs en cas d'*overflow*, par exemple lorsqu'il y a une expression arithmétique avec trop de parenthèses. Une amélioration possible de cette partie serait donc de rattraper ces erreurs afin de gérer les débordement de pile.

1.2 Vérification contextuelle

Nous considérons que le compilateur effectue une analyse contextuelle correcte. Nous avons effectué une base de test que nous pensons suffisamment complète et l'analyse contextuelle passe sur l'ensemble de ces tests. Sauf certains bugs que nous n'aurions pas décelé, nous pensons que cette partie est correctement traitée. De même, l'enrichissement et la décoration d'arbre, la mise à jour des différents attributs utilisés pour la partie C (par exemple, l'indexage pour la création de la table des méthodes, les labels...) sont effectués.

Toutefois, le code implémentant cette partie aurait pu être optimisé et peut-être mieux factorisé: Nous n'avons pas systématiquement eu recours à l'utilisation de classes abstraites, ce qui peut entraîner des contraintes supplémentaires si l'on veut modifier ou faire évoluer notre compilateur, en ajoutant des règles à la grammaire par exemple.

1.3 Génération de code

Partie sans objet

Il est possible de déclarer plusieurs variables, sans initialisation, ou avec initialisation (opérations, entiers, flottants ou booléens). Il est aussi possible d'effectuer tous types d'instructions : assignation, boucle, comparaison, affichage, opérations arithmétiques etc...

Partie avec objet

Il est possible de générer le code de programme simple.
En effet, plusieurs classes peuvent être déclarés, et peuvent hériter d'autres : les attributs et méthodes peuvent être redéfinis. Quelque soit le cas, la création de la table des méthodes ainsi que l'initialisation des attributs s'effectuent correctement. Dans le programme principal ou dans une méthode, il est possible de faire des appels de fonctions avec des paramètres qui peuvent être :

- des entiers, flottants, ou booléens
- des variables globales ou locales
- des paramètres de la fonction dans laquelle elle est appelée
- des opérations arithmétiques

Il est également possible, au sein d'une méthode, de retourner un nouvel élément d'une autre classe.

Cependant, nous ne couvrons pas l'ensemble des cas possibles. En effet, certains cas provoquent des erreurs inattendues à l'exécution. Le code assembleur dans sa structure semble être le bon, mais la gestion des registres provoque des erreurs. Par exemple, il est possible que lors d'un appel de fonction, le registre utilisé précédemment pour stocker une valeur, soit utilisé, écrasant ainsi la valeur précédente, pourtant nécessaire à la suite du programme. Une voie d'amélioration serait de revoir les appels aux instructions PUSH et POP qui sont faits dans le code.
De plus, le cas d'une division ou d'un modulo par un appel de fonction ne vérifie pas que le retour de cette fonction ne vaut pas zéro.
Pour finir, les *casts* entre littéraux sont possibles, mais le cas des *casts* de classe n'a pas été traité.

2 Implémentation et architecture

2.1 Vérification contextuelle

La vérification contextuelle démarre à l'appel de la méthode *verifyProgram* dans la classe ***DecacCompiler***. Celle-ci appelle à sont tour les différentes méthodes *verify* des attributs de l'instance *Program* (ici, sur la *ListDeclClass* et le *Main*). De cette manière, les appels sur ces méthodes se déroulent sur l'arbre abstrait du programme: Chaque règle de la grammaire est traduite dans ces méthodes, qui sont implémentées dans toutes les classes de `fr.ensimag.deca.tree`, ou, si ce n'est pas le cas, dans une classe abstraite parente.

Par analogie, les attributs hérités sont traduits comme les arguments des méthodes *verify*, et les attributs synthétisés, lorsqu'ils sont utilisés dans la règle, sont traduits comme les éléments renvoyés par ces méthodes (en général, ces attributs sont soient des types, soit des signatures). De cette manière, les règles de la grammaire soumises à des conditions donnent lieu, lorsque ces dernières ne sont effectivement pas respectées, à la levée d'erreurs contextuelles, par le biais de la classe *ContextualError* du package `fr.ensimag.deca.context`. Les différentes passes de la vérification donnent lieu, pour certaines classes (les classes concernant la partie objet notamment : *DeclClass*, *DeclMethod*, *DeclFields*...) à deux, voire 3 méthodes *verify* (*verifyMembers* ou *verifyBody*, par exemple) correspondant à la règle de la grammaire attribuée de la passe correspondante. C'est pendant ce parcours de l'arbre que la décoration et l'enrichissement sont effectués.

Gestion des environnements : Le premier choix d'implémentation auquel nous avons été confrontés est le choix de structure de données pour la gestion de l'environnement des identificateurs d'expressions et l'environnement des types. Ainsi, dans le package `fr.ensimag.deca.context`, nous avons créé une classe ***EnvironmentType***, semblable à la classe ***EnvironmentExp*** déjà fournie. Dans ces deux classes, nous avons choisi de représenter les dits environnements avec la structure *HashMap*: Cette structure de donnée est bien adaptée pour garantir l'unicité des éléments ainsi que l'ajout et l'accès rapide à ces derniers via leur clés. Les spécifications de ces classes sont données en figure 1. Le *HashMap* est alors déclaré dans le constructeur de ces classes.

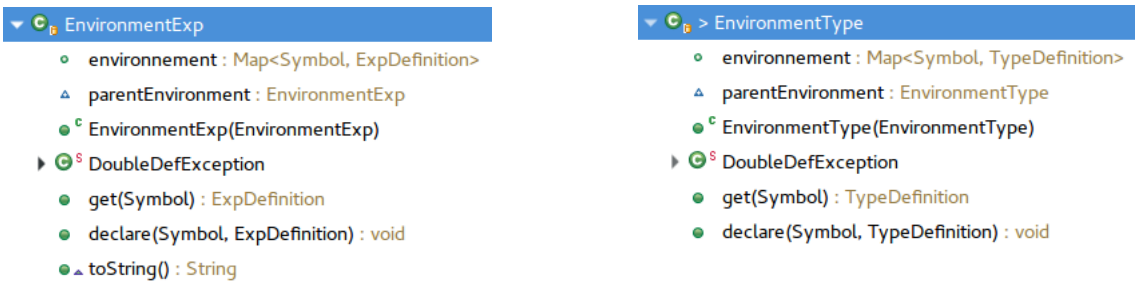

Figure 1: Squelettes des classes ***EnvironmentExp*** et ***EnvironmentType***

Ajout des non-terminaux non-fournis : Certains éléments de la grammaire ont été omis dans les sources de base, nous avons donc du ajouter diverses classes au package `fr.ensimag.deca.tree`. La figure 2 précise les classes qui ont été rajoutées dans la hiérarchie de la classe racine *Tree*. On y voit l'ajout de certains terminaux intervenant dans la règle des non-terminaux **inst** et **expr** : *Cast*, *Instanceof*, *Return*, *New*, *This*, etc. De même, on voit l'apparition de classes traduisant les non-terminaux permettant la déclaration de classe : *DeclMethod*, *DeclFieldSet*, *DecFields*...

Implémentation des méthodes utiles : La partie [**SyntaxeContextuelle**] du polycopié introduit les différentes grammaires attribuées en définissant des notions propres au langage deca, telles que les règles de sous-typage, la compatibilité pour la conversion, etc. Pour optimiser et factoriser au mieux notre code, nous avons donc été amenés à implémenter des méthodes statiques telles que *subtype* dans la classe ***AbstractExpr***, *correctCast* dans la classe ***Cast***, etc.

Égalité des signatures : Lors de la redéfinition d'une méthode dans une classe fille, la règle 2.7 de la grammaire attribuée à la passe 2 impose à la méthode d'avoir la même signature que la méthode parente. Ainsi il a fallu implémenter une méthode *equals* pour les éléments de cette classe.

2.2 Génération de code

- **La gestion des registres**

  Pour gérer les registres, c'est à dire pour savoir quel registre utiliser et ainsi éviter les problèmes de mémoire, nous avons créer une classe ***RegisterManager***. Celle-ci contient différents attributs :

  - *registerCourant*, qui nous indique le numéro du premier registre libre. Ce numéro peut être incrémenté lors d'une instruction LOAD par exemple, et décrémenté lors d'une instruction STORE à l'aide des fonctions *incrementCompteurCourant()* et *decrementRegisterCourant()*
  - *registerMax*, utilisé lorsque l'option -r du compilateur est activée. Cette variable, par défaut, vaut 15 (nombre de registre maximum que nous pouvons utiliser). Elle est utile notamment dans les fonctions *verify(int nbPlaceNecessaire)* et *getMaxAtteint()* qui sont utilisées lors de l'activation de -r.
  - *varGlobale*, utilisé pour connaître le nombre de variables globales déclarées dans un programme principal, et donc utile à l'ajout des instructions ADDSP en début de bloc.

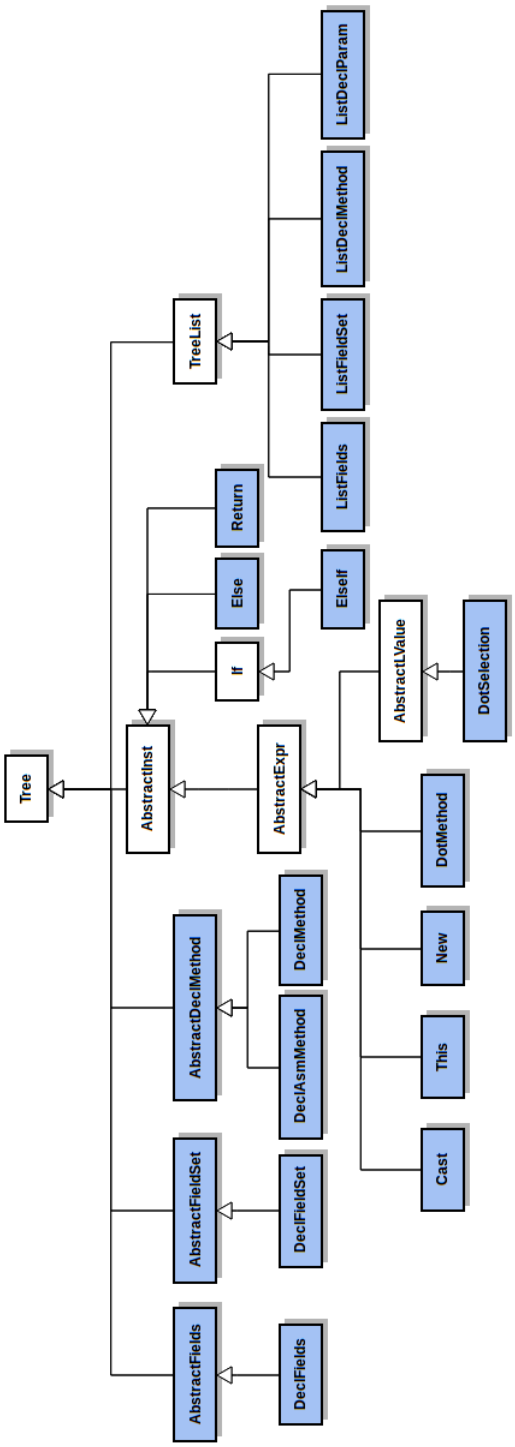

Figure 2: Les classes ajoutées

- *maxTempo* permet de savoir combien de registres temporaires on utilise au maximum dans le programme principal. Cela est nécessaire à l'ajout de l'instruction TSTO en début de bloc.
- *varLocal* correspond au nombre de variables locales d'un méthode, nécessaire pour déclarer le TSTO et le ADDSP en début de méthode.

Un objet de type *RegisterManager* est déclaré en tant qu'attribut dans le classe ***CompilerOptions***, lui même attribut de la classe ***DecacCompiler***. On utilise donc ces deux classes pour accéder aux valeurs des registres évoqués précédemment.
RegisterManager nous a permis de régler nos problèmes de mémoire pour la partie sans objet. En revanche, pour la partie objet, il aurait été possible d'effectuer une fonction dans la classe ***RegisterManager***, indiquant si l'on devait PUSH ou POP le registre courant. Par manque de temps, ce type de fonction n'a pas été implémenté. Ainsi, lorsqu'un programme effectue un grand nombre d'appels de fonctions, certains registres clés sont écrasés, déclenchant alors des erreurs à l'exécution, tel que des opérations ou comparaisons avec des registres indéfinis.

- **La classe *DecacCompiler***

Certains attributs ont été ajoutés à la classe ***DecacCompiler***.
Lors de la déclaration de plusieurs booléens à la suite, qui nécessite des opérations booléennes (ex : *boolean b = true ou false*), différentes étiquettes sont nécessaires. Ainsi, on introduit un compteur d'étiquettes *iLabel* dédié à cet effet. De même pour les déclarations de méthodes : on utilise le compteur *nombreMethod*. Le même principe est appliqué dans les classes ***While*** et ***IfThenElse***, mais cette fois-ci, les attributs sont à l'intérieur des classes.

On remarque aussi, dans ***DecacCompiler***, la présence d'autres attributs : *compteur*, qui représente un compteur pour les variables globales (3(GB), 4(GB) etc...), *compteurLB*, pour les déclarations de variables locales aux méthodes (1(LB), 2(LB) etc..), et *compteurParam* pour empiler les paramètres lors d'un appel de fonction (-3(LB), -4(LB) etc...). Ces trois attributs ainsi que leurs accesseurs et mutateurs, auraient du être déclaré dans la classe ***RegisterManager*** pour plus de cohérence. Il est possible de corriger rapidement cet oubli, en remplaçant les appels *compiler.get...()* par *compiler.getCompilerOptions().getRegisterManager().get...()*, par exemple.

Pour finir, lors de la génération de code pour la partie avec objet, et plus particulièrement pour la génération de code des méthodes, il est nécessaire d'effectuer des instructions ADDSP x en début de code par exemple. La variable x étant connue uniquement à la fin de la génération de code, nous devons utiliser la méthode *addFirst* sur un objet de type IMAPProgram pour que ces instructions soient placées au bon endroit dans le fichier assembleur. Pour remédier à ce problème, nous avons créé une liste chaînée d'objet IMAPProgram. À chaque fois que nous générons du code pour une méthode, nous créons un nouveau IMAPProgram, que nous ajoutons à la fin de la liste chaînée. On ajoute alors nos instructions, toujours dans le dernier IMAPProgram de cette liste, de la manière suivante : *compiler.getLinkedList().getLast().addInstruction(__)*.

- **Particularité de l'utilisation des registres**

Nous avons choisi, dans certains cas, de ne pas faire appel au registre courant, mais d'utiliser toujours le même registre pour effectuer une suite d'instructions. Ces cas sont les suivants :

- Lors de la création d'un objet, le registre utilisé avant et après l'appel à l'initialisation des paramètres (BSR init.) est R2.
- Lors d'un appel de méthode, les instructions LOAD, CMP, BEQ, LOAD, BSR utilisent le registre R2.
- Pour la création de la table des méthodes, on utilise R0 pour stocker les labels puis les placer au bon endroit dans la pile.
- Le retour d'une fonction est toujours stocké dans R0.

- **Table des étiquettes et des méthodes**

  Comme le conseillait le sujet, nous avons créé une table des étiquettes. Pour cela, nous avons ajouté un attribut *ArrayList¡Label¿ tableEtiquette* dans la classe ***ClassDefinition***. C'est à ce moment que nous prenons en compte l'héritage ou la redéfinition des méthodes. Une fois l'*ArrayList* correctement construit, la génération de code pour la table des méthodes est beaucoup plus simple : il suffit de parcourir les éléments de l'*ArrayList* généré.

- **Généralité sur le code**

  Dans l'ensemble, les fonctions que nous avons implémenté dans les classes mères sont ensuite redéfinies dans les classes filles, car spécifiques à chacune d'entre elles. Par exemple, une initialisation ne s'effectue pas de la même manière pour un entier, un identifier ou un booléen.
  Dans la majorité du temps nous distinguons la génération de code dans le cas d'une méthode et d'un programme principal, car les registres utilisés sont différents. Une voie d'amélioration aurait été de passer les registres en paramètres des fonctions.
  Cependant, nous avons tenté de mettre à profit l'abstraction de certaines classes pour éviter le code redondant. C'est le cas par exemple dans la classe ***AbstractOpArith***.