DOCUMENTATION DE CONCEPTION

EQUIPE 20

GROUPE 4

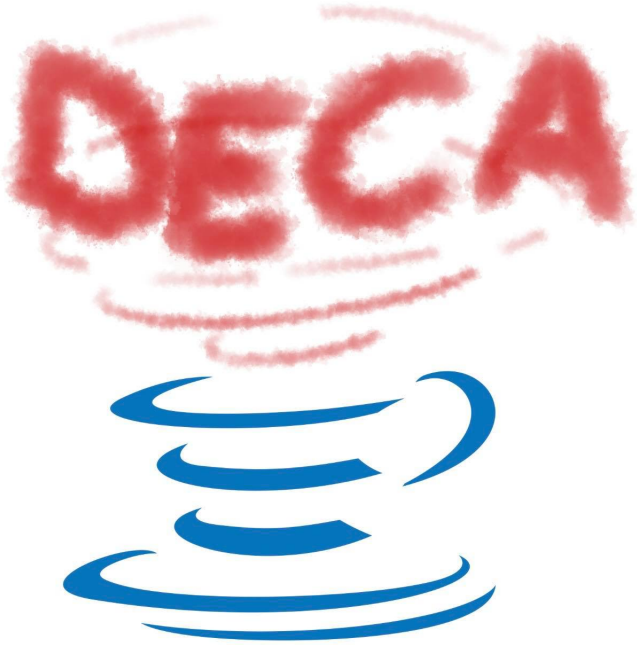

Table des matières

SECTION 1

Le compilateur Decac

1.1 Gestion des paramètres

Lorsque la commande `decac` est lancée, c'est la classe `DecacMain` qui est appelée. Celle-ci appelle `DecacOptions` pour la gestion des options passées dans le terminal.

1.2 Ajout de fonctionnalités

Pour ajouter une nouvelle option au compilateur, il suffit donc d'intégrer la lettre relative dans la méthode de parsing de la classe `DecacOptions` et de lever le `flag` correspondant. Attention, pour une option qui attend des arguments supplémentaires, il ne faut pas oublier d'incrémenter l'indice `i` (qui parcours les arguments de la ligne de commande) du nombre de paramètres supplémentaires traités. Par exemple, pour ajouter l'option `opt`, il faudra ajouter un attribut `optSet` ainsi que son accesseur. Le traitement relatif à cette commande devra être effectué dans la classe `DecacCompiler` ou `DecacMain` en récupérant le flag en question et en vérifiant si il a été levé.

1.3 Détails supplémentaires

Une petite documentation du compilateur est affichée à chaque fois qu'une entrée n'est pas en accord avec les options disponibles. Cette documentation est enregistrée au format txt dans le même répertoire que les classes associées au compilateur. Celle-ci peut alors être modifiée pour ajouter les informations nécessaires à l'utilisation de nouvelles options.
La bannière affichée via l'option -b est aussi disponible et modifiable au même endroit que la documentation du compilateur.

SECTION 2

Analyse lexicale et syntaxique

2.1 Lexer

Le lexer de ce projet est implémenté en ANTLR, dans le fichier `DecaParser.g4`. Son implémentation suit "à la lettre" la lexicographie du langage Deca, donc nous n'avons pas pris beaucoup d'initiative. Cependant, nous avons tout de même fait le choix de remonter les erreurs lexicales directement dans ce fichier.
Pour l'ajout de nouvelle fonctionnalité, le programmeur doit prendre soin de conserver l'ordre de déclaration de chaque token.

2.2 Parser

Le parser de ce projet est implémenté dans le fichier `DecaParser.g4`. Il s'agit là du parcours d'un arbre dont les noeuds représentent les variables et les instances du fichier parsé. Le parser correspond à la partie syntaxique du projet et permet par la suite l'affichage de l'arbre sans décorations. Il récupère par ailleurs les tokens analysés par la partie lexicale, c'est-à-dire le lexer. Voici un exemple de code appelant le parser sur un fichier à partir des tokens du lexer :

```
DecaLexer lex = AbstractDecaLexer.createLexerFromArgs(args);
CommonTokenStream tokens = new CommonTokenStream(lex)
DecaParser parser = new DecaParser(tokens);
```

Le parcours de l'arbre est appelé avec la méthode `parseProgramm()`. Chaque branche est alors parcouru de la façon suivante :

— Chaque noeud peut avoir une ou plusieurs expression, séparées par le symbole

`'|'`, il peut s'agir d'une feuille (par exemple `INT`) ou d'un noeud avec un ou plusieurs fils

— Dans le cas d'une feuille, on appelle alors le constructeur associé (ici `IntLiteral`)
— Dans le cas d'un noeud, on vérifie que ses fils ne soient pas *null*, puis on construit l'arbre du noeud en fonction de celui de ses fils
— Dans les deux cas, il est important d'appeler la méthode setLocation, cela permettra par la suite de créer les définitions ou de situer les erreurs

Certains choix ont été faits pour permettre d'affecter un type à plusieurs variables à l'aide d'un unique identifier, par exemple :

```
int i = 1, j = 2;
```

Ces choix ont été faits pour la déclaration de champs et de variables. Les méthodes DeclVarSet et DeclFieldSet vont alors passer en argument une liste qui sera modifiée et enrichie de nouveaux éléments par la suite, ainsi que d'autres éléments comme le type ou la visibilité.

SECTION 3

Analyse contextuelle

3.1 Architecture générale de la partie contextuelle

L'architecture de la partie contextuelle reprend l'architecture du fichier DecaParser.g4, dont les noeuds se traduisent ici par des classes. En effet, chaque classe possède le nom explicite d'un noeud et un constructeur qui est appelé depuis le parser lors du parcours de l'arbre. Cette partie se base sur la vérification contextuelle d'un programme, cette vérification est donc faite par les méthodes `verify*` de chaque classe, directement définies ou héritées de classes abstraites. Ainsi, pour un grand nombre d'opérateurs par exemple, les méthodes de vérification contextuelle se trouvent dans la classe `AbstractOpArith`.

3.2 Classes `EnvironmentExp` et `EnvironmentType`

Ces deux classes sont primordiales pour le reste de l'analyse contextuelle. `EnvironmentExp` est représenté par un HashMap permettant d'associer à un symbole la définition d'une expression. Cette définition sera affichée en décoration de l'arbre lors de la déclaration ou de l'appel d'un identificateur. Il est à noter pour la partie avec objet qu'il peut y avoir plusieurs `EnvironmentExp` associés non seulement à Main, mais à des classes ou des méthodes définies dans le programme. Cette classe possède des méthodes pour obtenir une définition à partir d'un symbole, ajouter un symbole et sa définition, obtenir l'environnement parent de l'environnement courant dans le cas des classes ou obtenir le dictionnaire en entier avec ses clés et ses valeurs. Il est finalement possible d'effectuer les opérations d'empilement et d'union disjointe sur deux environnements en appelant

les méthodes du même nom.
Pour ce qui est de la classe `EnvironmentType`, elle est représentée par un HashMap permettant d'associer à un symbole la définition de son type. Cette définition sera affichée en décoration de l'arbre lors d'une opération ou d'une instance. Cette classe est commune au `Main` et à toutes les classes définies dans le programme. On y trouve les définitions des types `Int`, `Float`, `Boolean`, `String` et `Void`, ainsi que de la classe `Object`, mère de toutes les classes, et de la méthode `equals`. Tout comme la classe `EnvironmentExp`, on peut ajouter un symbole et sa `typeDefinition`, ou obtenir le type ou la définition du type d'un symbol. Il n'y a cependant pas la possibilité d'empiler ou de prendre l'union disjointe entre deux `EnvironmentType`.

3.3 Méthodes `verify`

L'analyse contextuelle passe ensuite par ces méthodes. Il s'agit tout d'abord de vérifier les conditions précisées par l'analyse plus théorique de la grammaire, et de renvoyer une `ContextualError` dans le cas où une condition est violée. Dans le cas contraire, on assigne un type à l'opérateur ou à l'instance analysée, ou on déclare un nouveau paramètre dans le cas des déclarations pour pouvoir vérifier son type et sa définition lorsqu'il sera appelé plus tard. Il est a noter qu'un `Identifier` peut représenter un type et non une variable, par conséquent, on ne peut pas appeler certaines méthodes pour ces `Identifier`. En particulier, pour l'`Identifier` d'une classe, on récupère la définition de la classe par l'intermédiaire des deux étapes suivantes :

```
ClassType classType = (ClassType) type.getType();
ClassDefinition def = classType.getDefinition();
```

Pour les classes, il est important de rappeler que chaque classe doit avoir son propre environnement contenant les champs et les méthodes, éventuellement héritées de la classe mère. Il faut alors vérifier que la classe mère est une classe, que les méthodes et les champs hérités sont bien des méthodes et des champs de même signature et de même type. Dans le code de notre projet, ces étapes de vérification sont factorisées le plus haut possible afin d'éviter leur réécriture. Par exemple, il est déjà vérifié dans la classe `DeclClass` que l'environnement de la classe mère est bien l'environnement parent de celui de la classe courante, il n'est donc pas utile de le revérifier dans les classes parcourues aux passes suivantes, l'erreur sera levée dès la première passe dans `DeclClass`.
Enfin, pour chaque méthode, un environnement d'expression est créé, auquel on ajoute les paramètres parcourus et déclarés dans `DeclParam`, puis on empile l'environnement de la classe. Par exemple, cela permet d'éviter les confusions entre un paramètre `x` de la méthode et l'expression `this.x`, `x` étant un champ de la classe. Pour repérer les champs et les méthodes, on leur associe un index. Pour les méthodes, cet index dépend du maximum des index de cette classe et de la classe mère, si la méthode n'est pas héritée. Ainsi, dans une `ClassDefinition`, `indexMethods` représente l'index de la prochaine méthode que l'on définira dans la classe, n'étant pas héritée. Tout comme l'incrément du nombre de champs et de méthodes d'une classe, on peut l'incrémenter à l'aide de la méthode `getIndexMethod`. Il faut également faire attention à la méthode sameType dé-

clarée dans les définitions, partie contextuelle : celle-ci renvoie le bon résultat s'il s'agit de types prédéfinis dans `EnvironmentType`, mais ne compare que si deux type sont des `ClassType`, peu importe s'ils sont de même classe.

SECTION 4

Génération de code assembleur

4.1 Architecture générale de la partie de génération de code

L'architecture de la partie génération de code s'appuie naturellement sur celle de l'étape d'analyse contextuelle. Le programme principal appelle donc la génération de code sur la racine de l'arbre, et la génération de code se propage jusqu'aux racines de l'arbre.
D'un autre côté, deux classes ont été ajoutées dans le package `codegen` pour aider à la génération de code dans les classes du package `tree`. Ces deux classes sont `Data` et `Labels`. La première a pour but d'aider à la gestion de la mémoire physique, c'est à dire des registres, de la pile et du tas. La seconde est une classe pratique pour gérer les étiquettes écrites en assembleurs, notamment pour les étiquettes relatives aux messages d'erreur.

4.2 Les méthodes relatives à la génération de code assembleur

Dans le package `tree`, chaque classe non abstraite est susceptible d'être appelée pour une partie de la génération du code assembleur, mais chacune peut avoir plusieurs façon d'être appelée.
Par exemple, le calcul d'une expression booléenne n'est pas réalisé de la même manière si le résultat doit être stocké dans une variable ou si cette expression est une condition d'une structure de contrôle. Ainsi, une classe comme `And` qui doit réaliser le "Et" booléen possède une méthode `codeGenInst` pour la première situation, et une méthode `codeBoolean` qui précise à quelle étiquette se rendre dans le cadre d'une structure de contrôle.

Attention, des exceptions `UnsupportedOperationException` sont encore volontairement présentes dans la classe `AbstractExpr`. En effet, certaines de ces méthodes n'ont pas vocation à être réutilisé ou réécrite par certaines sous-classes, et plutôt que de laisser des méthodes vides, il est préférable de laisser cette exception pour pouvoir débuger plus facilement l'ajout de nouvelles fonctionnalités.

4.2.1 Les opérations arithmétiques

Pour les opérations arithmétique binaires et unaires, l'idée était de déléguer le maximum du travail aux classes `AbstractBinaryExpr` et `AbstractUnaryExpr`, toute deux s'appuyant sur la classe `Data` que l'on précisera ci-dessous (4.3). L'ensemble de la gestion des registres et de la pile était préparé lorsque que l'on tombe sur une feuille de l'architecture représentant une opérations arithmétiques, prenons par exemple la classe `Multiply`. Il ne s'agit donc plus que de gérer les erreurs possibles, et donc spécifiques à chaque opérations, puis d'ajouter l'instruction correspondante.
Dans un soucis d'optimisation de l'utilisation des registres, après chaque opération, nous indiquons que le dernier registre utilisé est celui qui reçoit la dernière opération. Nous ne retenons donc pas dans des registres des résultats intermédiaires qui ne seront plus utiles dans la suite.

4.2.2 Les opérations booléennes et structures de contrôles

Pour les opérations booléennes réalisées dans le cadre des conditions des structures de contrôle, nous utilisons une modélisation par flots de contrôle. Cela est très adapté aux structures de contrôle, car plutôt que de calculer le résultat booléen de la condition puis de le comparer à 0 ou 1, pour savoir si un branchement est à faire ou non, on effectue directement ces branchement sans passer par le stockage de 0 ou de 1. Ainsi, aucun registre n'est nécessaire pour effectuer ces opérations. Cette implémentation est gérée par la méthode `codeBoolean` qui est héritée de la classe abstraite `AbstractExpr`. Toutes les classes filles de `AbstractExpr` n'ont pas vocation à être traitées comme des booléens, mais il était impossible de placer cette fonction plus haut dans l'architecture car la classe `Identifier` devait contenir cette méthode.
D'un autre côté, pour stocker le résultat d'un booléen dans une variable, nous avons pris la convention suivante : un booléen est représenté en mémoire par un entier, il vaut `false` si l'entier vaut 0 et `true` si l'entier vaut 1.

4.2.3 Les objets

Pour la partie relative à la programmation orientée objet, la majorité du code est écrite dans la classe `DeclClass`. Il est assez facile de retracer le déroulement de cette dernière grâce aux noms de méthodes très explicites et aux nombreux commentaires présents. Dans cette partie, il est important de comprendre que l'écriture du code permettant l'initialisation d'un objet ainsi que les codes de ses différentes méthodes doivent être indépendant du reste du code générer, notamment du point de vu de la gestion de la mémoire. Pour se faire, nous avons décidé de créer deux environnements distincts

concernant la gestion de la mémoire. Un premier consacré au programme principale, qui est instancié au début de la génération du code, et un second, temporaire, qui est réinstancié à chaque fois qu'on débute l'écriture du code assembleur d'une méthode. Cela permet notamment de calculer plus facilement les `TSTO` et de manipuler avec plus de facilité les différents blocs de notre programme assembleur.
De la même manière que pour le code de la partie sans objet, une multitude de méthodes du type `codeGen...` ont été créées pour gérer les différentes fonctionnalités. Par exemple, pour le cas des sélections, une méthode `codeGenSelect` est présente à différents niveaux de l'arbre pour gérer les différents cas de sélection. En effet, un identifiant par exemple peut tout à fait être appelé pour une sélection lorsque celui-ci identifie un objet. Il en est de même pour le codage des appels de méthodes.
Pour l'instruction `instanceof`, il est utile de remarqué qu'une méthode privée `isInstanceOf` est écrite, et que l'instruction `Cast` utilise une méthode privée similaire. Cette méthode utilisée pour générer le code assembleur vérifiant si un objet est une instance d'une certaine classe devrait être factorisé dans la classe `AbstractExpr`, soyez donc vigilant si vous désirez modifier cette méthode, et modifiez aussi sa copie si cela est nécessaire.

4.3 Structure de la classe `Data`

La classe `Data` a été notre point d'appui pour l'étape de génération de code, elle regroupe tous les algorithmes de gestion des registres, de la pile et du tas. Un objet de la classe est associé à notre programme principal, de la classe `IMAProgram`, et ensuite, à chaque programme subsidiaire nous lui associons aussi une nouvelle `Data`. Nous pouvons ainsi gérer les instructions comme `TSTO` à la fin de chaque utilisation d'une `Data`.
Cette classe a été modifiée, étoffée, corrigée tout le long du projet pour essayer de représenter au mieux notre conception des nouvelles fonctionnalités. Nous estimons que son implémentation est maintenant la plus épurée et intuitive possible. Cependant, s'il vous venait à utiliser ou modifier cette classe, quelques points sont indispensables :

- Toute les méthodes `codeGen...` prennent un objet `DecacCompiler` en argument, et cet objet peut être définit pour écrire dans le programme principale ou subsidiaire. Il est important de savoir retracer l'exécution du code pour savoir dans quel programme nous écrivons.
- Une méthode `restoreData` nous permet de nettoyer une mémoire, elle est à utiliser avec précaution pour ne pas perdre des informations que nous ne souhaitions pas oublier, et il faut faire attention à appeler cette méthode sur la `Data` du programme principal ou subsidiaire.

4.3.1 Génération de la table des méthodes

La génération de la table des méthodes s'effectue de deux manières en parallèle. Premièrement, grâce à une table de hachage, les clés sont les index de chaque méthode et les valeurs sont les labels associés. Celle-ci est réutilisé plus tard dans le code java. Deuxièmement, on ajoute en assembleur cette implémentation. À priori, cette portion de code ne sera pas amené à changer.