Decac

Groupe Génie Logiciel n°29

Compilateur pour le langage Deca

Rapport de conception

Janvier 2024

Table des matières

1 Architectures du code

1.1 Package tree

Un ensemble de classes a été initialement fourni pour représenter un arbre abstrait, contenu dans le package ”fr.ensimag.deca.tree”. Ce package inclut notamment la classe de base ”Tree”, commune à tous les nœuds de l'arbre. Les classes associées aux nœuds du langage sans-objet étaient déjà fournies, bien que toutes incomplètes. Par la suite, nous avons développé les classes correspondant aux nœuds du langage objet, telles que :

1. Déclaration d'une classe : AbsractDeclClass, ListDeclClass, DeclClass
    - Utilité : Ces classes représentent respectivement la déclaration abstraite d'une classe, une liste de déclarations de classes, et une déclaration concrète d'une classe.
    - Héritage : AbstractDeclClass hérite de la classe de base Tree, et ListDeclClass peut contenir une liste d'objets de type AbstractDeclClass. DeclClass hérite de AbstractDeclClass
2. Méthodes d'une classe : AbstractDeclMethod, ListDeclMethod, DeclMethod
    - Utilité : Ces classes modélisent respectivement la déclaration abstraite d'une méthode, une liste de déclarations de méthodes, et une déclaration concrète d'une méthode.
    - Héritage : De manière similaire à la déclaration de classes, AbstractDeclMethod hérite de Tree, ListDeclMethod peut contenir une liste d'objets de type AbstractDeclMethod, et DeclMethod hérite de AbstractDeclMethod.
3. Fields d'une classe : AbstractDeclField, ListDeclField, DeclField
    - Utilité : Ces classes représentent respectivement la déclaration abstraite d'un champ (attribut), une liste de déclarations de champs, et une déclaration concrète d'un champ
    - Héritage : AbstractDeclField hérite de Tree, ListDeclField peut contenir une liste d'objets de type AbstractDeclField, et DeclField hérite de AbstractDeclField.
4. Paramètres d'une méthode : AbstractDeclParam, ListDeclParam, DeclParam
    - Utilité : Ces classes modélisent respectivement la déclaration abstraite d'un paramètre de méthode, une liste de déclarations de paramètres, et une déclaration concrète d'un paramètre.
    - Héritage : AbstractDeclParam hérite de Tree, ListDeclParam peut contenir une liste d'objets de type AbstractDeclParam, et DeclParam hérite de AbstractDeclParam.
5. Autres ... : InstanceOf, CallMethod, Cast, New,This, NullLiteral, GetAttribut, Return
    - Utilité : Ces classes représentent différents éléments du langage de programmation, tels que l'opération d'instanciation, l'appel de méthode, la conversion de type, la création d'un objet, l'utilisation de l'expression ”this”, la représentation littérale de la valeur null, l'accès à un attribut, et l'instruction de retour.
    - Héritage : Ces classes héritent de la classe de base Tree, partageant ainsi des fonctionnalités communes.

1.2 Package tools

À l'origine, le package "fr.ensimag.deca.tools" comprenait des classes dédiées à la gestion des erreurs et des symboles. Nous avons enrichi ce package en y ajoutant des classes qui traitent spécifiquement de l'implémentation des Labels, telles que "ClassLabel" et "LabelTable". Ces classes ont pour fonctionnalité la génération de labels indépendants, essentiels pour l'implémentation des boucles, des structures conditionnelles (if), et des méthodes dans le code assembleur.

1.3 Package syntax

Le package "fr.ensimag.deca.syntax" est utilisé pour l'analyse lexicale et syntaxique du langage Deca. Les classes qu'il contient prennent en charge l'ensemble du processus lexical et syntaxique en créant un analyseur syntaxique (parser) et un analyseur lexical (lexer). Il est à noter que tout le code présent ici avait déjà été fourni.

1.4 Package context

Le package "fr.ensimag.deca.contexte" est dédié à la définition des identificateurs dans les expressions, telles que les noms de variables, de méthodes, de classes et de types. Aucune classe n'a été ajoutée de notre part, cependant, plusieurs d'entre elles étaient incomplètes, notamment "EnvironnementType" et "EnvironnementExp". Nous avons enrichi ces classes afin de mettre en œuvre l'utilisation des différents types, de vérifier leur compatibilité, de gérer les conversions (casts), les sous-types, ainsi que d'autres fonctionnalités spécifiques aux identificateurs.

1.5 Package codegen

Le package "fr.ensimag.deca.codegen" ne comprend qu'une seule classe mise en œuvre par notre équipe. Cette classe contient exclusivement des méthodes statiques utilisées pour la gestion des registres et du code assembleur du compilateur.

1.6 Extension

En plus des classes qui nous étaient imposées à implémenter, nous avons ajouté une classe appelée "BinShift" qui effectue l'opération de décalage binaire sur les types float et int, indispensable à notre extension. De plus, un package nommé "fr.ensimag.deca.trigo" a été créé, contenant les classes nécessaires au bon fonctionnement de l'extension trigonométrique que nous avons choisie.

2 Spécifications sur le code

2.1 Partie A

La partie A, qui englobe le lexer et le parser, était fortement guidée, expliquant pourquoi nous n'avons pas fourni d'implémentation spécifique ici. Les seuls ajouts indépendants des explications se concentraient principalement sur la partie objet du parser, qui était relativement concise. Si vous avez des questions spécifiques ou des points que vous aimeriez approfondir, n'hésitez pas à les partager, et je serai ravi de vous aider.

2.2 Partie B

Concernant les stades Hello World et Sans Objet, nous avons utilisé des fonctions prédéfinies, donc nous n'avions pas à implémenter grand-chose de nouveau. Nous avons commencé à véritablement implémenter des fonctions à partir du moment où nous sommes arrivés au langage Essentiel.

Commençons par la classe **declClass**. Elle contient des méthodes permettant d'exécuter les trois passes de l'étape B sur une classe traitée dans un programme Deca.
Pour la passe 1, on a eu besoin du Compiler pour implémenter `verifyClass`, afin de récupérer son EnvironmentType et y déclarer la classe traitée. Puis, pour la passe 2, on a également eu besoin du Compiler, et plus précisément de son EnvironmentType pour cette fois y sauvegarder le nombre de méthodes et d'attributs de la classe mère dans la classe traitée. On appelle ensuite la vérification sur les listes des méthodes et des attributs (`verifyListDeclMethod` et `verifyListDeclField`), où le Compiler est aussi nécessaire. La classe courante étant fournie, cela explique que seul le Compiler soit en paramètre, même si la classe courante est aussi nécessaire pour la vérification. Enfin, concernant la passe 3, les besoins sont identiques à ceux de la passe 2.

Passons maintenant un niveau en dessous, et intéressons-nous à la classe **declField**. Lors du passage de l'étape B dans une classe, et plus particulièrement au niveau des passes 2 et 3, des vérifications sont menées au sein des attributs de la classe. Pour ce qui est de la passe 2, on a le Compiler en paramètre de la méthode de vérification puisqu'on en a besoin pour enregistrer la définition de l'attribut. On a également la classe courante en paramètre qui va permettre de déclarer l'attribut dans cette classe. Pour ce qui est de la passe 3, on retrouve le Compiler et la classe courante en paramètre, car on appelle la vérification sur l'initialisation. Or, cette méthode est déjà fournie, et on a besoin du compiler, de l'environnementExp, de la classe courante et du type de la variable qu'on veut initialiser.

Les attributs ayant été vérifiés, c'est au tour des méthodes, au travers de la classe **declMethod**. Seules les passes 2 et 3 sont concernées, comme pour les attributs. Tout d'abord, en passe 2, on a le Compiler et la ClassDefinition. Le Compiler va nous servir à vérifier le type de la méthode, et à enregistrer la définition de la méthode. On a besoin de la currentClass pour avoir l'environementExp de la classe mère, pour définir si cette méthode est une méthode Override ou non. La currentClass permet d'avoir l'environnementExp, qui nous permet de déclarer la méthode. Elle nous sert aussi à faire d'autres vérifications. Enfin, on n'oublie pas de vérifier la liste des paramètres. Puis en passe 3, on a le Compiler et la currentClass, car on va vérifier le corps de la méthode, et en paramètre de la méthode de vérification du corps de la méthode, on a besoin du compiler, de la classe courante, du type de la méthode, et de l'environnementExp des paramètres de cette méthode.

Passons encore un niveau en dessous. En effet, les méthodes peuvent avoir des paramètres, auquel cas, il est nécessaire d'en vérifier la validité grâce aux méthodes de vérification de la classe **declParam**. En passe 2, on a le compiler, un localEnv, et une classe courante. Le compiler permet de vérifier le type du paramètre et d'enregistrer sa définition et son type. Le localEnv quant à lui permet de déclarer le paramètre. Les passes 1 et 3 ne concernent pas la vérification des paramètres d'une méthode.

Passons maintenant aux appels des attributs et des méthodes. Commençons par la classe **GetAttribut**. On a le compiler, un localEnv, et une currentClass. Ils vont tous servir à vérifier l'expression. Puis, selon si l'expression est précédée d'un This ou non, on utilisera la currentClass ou le Compiler pour accéder à l'environnementType, et ainsi avoir la ClassDefinition qui permettra de trouver le FieldDefinition. Une fois le FieldDefinition, on vérifie sa visibilité, et enregistrer sa définition.

Maintenant parlons de la classe **CallMethod**. Comme précédemment, on a le compiler, un localEnv, et une currentClass. Ils servent à vérifier l'expression. Puis, selon si l'expression est précédée d'un This ou non, on utilisera la currentClass ou le Compiler pour accéder à l'environnementType, et ainsi avoir la ClassDefinition qui permettra de trouver la Definition. Si c'est bien une méthode, on récupérera la MethodDefinition, qu'on enregistrera dans le compiler. Enfin, avec cette MethodDefinition, on va pouvoir récupérer la signature et vérifier les types des paramètres de la méthode.

Ceci conclut les spécifications sur l'étape de vérification contextuelle du compilateur. À présent, passons à la dernière étape : la génération du code assembleur, aussi appelée étape C.

2.3 Partie C

2.3.1 Passe 1 Objet

Après la génération de la table des méthodes (voir partie Algorithme et structure de données), nous appelons la méthode "CodeGenClassPasseOne" pour chaque classe. Cette méthode attribue l'adresse de la classe dans la pile et enregistre, pour chacune de ses méthodes et les méthodes héritées, un emplacement dans la pile ainsi qu'un label. Ces informations seront essentielles lors de la deuxième passe de l'objet.

2.3.2 Génération Variable

Après le calcul des variables les plus utilisées dans le programme (voir partie Algorithme et structure de données), la méthode "codeGenDeclVar" effectuera deux possibilités pour chaque variable. Dans l'ordre, chaque variable se verra attribuer soit une adresse en pile, soit un registre. Ensuite, en fonction de la manière dont elles sont directement initialisées ou non, leur code assembleur sera généré en conséquence.

2.3.3 Génération Main

Dans cette section, l'ensemble du code présent dans le main sera généré à l'aide de la méthode codeGenInst, qui était déjà fournie, mais non complète au début du projet. Ici, chaque classe dans le package "tree" et héritant de AbstractExp devra implémenter sa propre méthode codeGenInst. Ces méthodes seront appelées tout au long du déroulement du programme, notamment pour des opérations

telles que PLUS, MOINS, DIV, THIS, etc. Le programme assembleur sera généré ligne par ligne en fonction de ces appels.

2.3.4 Passe 2 Objet

La passe 2 de l'objet appelé par la méthode codeGenClassPasseTwo apparait après le main, il se sépare en deux parties, la génération de l'initialisation de la classe puis la génération des méthodes de la classe. Nous suivons les instructions écritent dans le poly à la lettre pour générer cette partie là, en utilisant des labels pour sauter directement à ses parties là quand appeler avec callMethod ou New.

2.3.5 Génération Erreurs

Ensuite, le code lié à l'ensemble des erreurs pouvant être appelé pendant le programme, telles que les erreurs de débordement, les erreurs d'entrée/sortie, les erreurs de pile pleine, le déréférencement null, est généré en dernier. Nous utilisons des labels pour effectuer un saut direct vers cette partie du code en cas d'erreur, évitant ainsi une lecture inutile d'une portion du programme. Cette approche contribue à optimiser le traitement des erreurs en ne parcourant pas inutilement les instructions lorsque tout se déroule correctement.

3 Algorithmes et structures de données

3.1 Gestion des Registres

Dans notre approche, la gestion des registres repose sur l'utilisation judicieuse de structures de données appelées Stacks, une fonctionnalité intégrée à JAVA. Cette utilisation des stacks simplifie le processus de stockage et de récupération des valeurs dans les registres.

Lorsqu'une valeur doit être assignée à un registre, ce registre est déplacé de la stack "RegistresLibres" vers la stack "RegistresUtilisés". Cette transition assure une traçabilité claire des registres disponibles et de ceux qui sont actuellement utilisés.

Au moment de récupérer la dernière valeur stockée dans un registre, il suffit de prélever la valeur située au sommet de la stack "RegistresUtilisés". Cette méthode garantit que nous accédons toujours au registre le plus récemment utilisé.

Une fois que la valeur d'un registre a été récupérée, ce même registre retourne à la stack "RegistresLibres", prêt à être réutilisé directement par la suite. Cette rotation efficiente des registres entre les deux stacks assure une utilisation optimale des ressources.

L'utilisation de stacks offre un avantage significatif, car la plupart du temps, seule la dernière valeur stockée dans un registre, celle en haut de la stack, est nécessaire. Cette gestion simplifiée des registres contribue à la clarté du code et à l'efficacité de l'utilisation des ressources, en évitant des manipulations superflues sur des registres non essentiels.

3.2 Gestion des variables

3.2.1 Variable stocké par Registre

Dans notre approche, il est primordial de stocker en registre uniquement les variables les plus fréquemment utilisées. Cela s'explique par le fait que l'opération de récupération de la valeur d'une variable stockée en registre est significativement moins coûteuse que si elle était stockée dans la pile.

Pour déterminer quelles variables méritent d'être stockées en registre, nous utilisons une structure de données intelligente : une HashMap de String, Integer. Pendant la phase B du processus, nous examinons chaque occurrence de chaque variable dans le code. Les variables les plus fréquemment rencontrées sont ensuite sélectionnées pour être stockées dans les registres, en fonction du nombre de registres disponibles, ses mêmes registres sont ensuite retirés de la stack des registresLibre et ne pourront plus être utilisé autrement que par l'appel de la variable.

Cette approche stratégique vise à optimiser les performances en réservant les registres aux variables les plus sollicitées, réduisant ainsi les temps d'accès aux données. L'utilisation de la HashMap facilite la comptabilisation des occurrences, offrant ainsi une méthode efficace pour hiérarchiser les variables à privilégier lors de leur placement en registre. En définitive, cette démarche contribue à l'efficacité générale de l'algorithme de gestion des registres.

3.2.2 Variable stocké en Pile

Les variables moins fréquemment utilisées sont quant à elles stockées dans la pile, empilées les unes au-dessus des autres en suivant leur ordre d'apparition dans le code. Pour accéder à ces variables, nous enregistrons leur adresse, ce qui nous permettra ultérieurement de retrouver et de récupérer efficacement leur valeur dans le code.

Cette approche de gestion des variables optimise l'utilisation des ressources en réservant les registres aux variables les plus sollicitées, tout en assurant une méthode fiable pour stocker et récupérer les valeurs des variables moins fréquemment utilisées à partir de la pile. En combinant cette stratégie avec la méthode décrite précédemment pour les variables les plus utilisées, notre algorithme de gestion des registres vise à garantir un équilibre optimal entre performance et utilisation efficace de la mémoire.

3.3 Classe / Objet

3.3.1 Table des méthodes

Pour générer la table des méthodes des classes, nous avons implémenté un algorithme, Cet algorithme a pour objectif de générer une table des méthodes pour les classes fournies en entrée. Voici une explication détaillée du fonctionnement de l'algorithme, ainsi que de ses avantages et inconvénients potentiels :

1. Initialisation : L'algorithme commence par initialiser une structure de données, en l'occurrence une HashMap nommée tableDesMethodes, qui stockera la table des méthodes des différentes classes.

2. Parcours des classes : Ensuite, l'algorithme parcourt la liste des déclarations de classes (classList).

3. Traitement de la classe courante : Pour chaque classe c dans la liste : Une nouvelle HashMap (table) est créée pour stocker les informations relatives aux méthodes de la classe courante.

    - Une nouvelle HashMap (table) est créée pour stocker les informations relatives aux méthodes de la classe courante.
    - On vérifie si la classe a une classe parente (superclasse). Si oui, on récupère la table des méthodes de la classe parente et on l'ajoute à la table courante.
    - On initialise un index pour suivre l'emplacement des méthodes dans la table courante.
    - On parcourt la liste des déclarations de méthodes de la classe courante (methodeList).
    - Pour chaque méthode, on vérifie si elle est marquée comme étant une redéfinition (isOverride). Si oui, on place la méthode à l'index de sa définition dans la table. Sinon, on place la méthode à un nouvel index calculé en fonction de l'index actuel et de la taille de la table de la classe parente.
    - La table nouvellement créée est associée au nom de la classe dans tableDesMethodes

4. Renvoi de la table des méthodes : Une fois toutes les classes traitées, la fonction renvoie la tableDesMethodes qui contient la table des méthodes de chaque classe.

Avantages de l'algorithme :

- Réutilisation des méthodes de la classe parente : L'algorithme intègre les méthodes de la classe parente, ce qui permet de réutiliser les méthodes déjà définies dans la hiérarchie des classes.
- Gestion des redéfinitions : L'algorithme prend en compte les méthodes redéfinies dans les classes dérivées, assurant ainsi une table des méthodes cohérente.

Défault potentiel de l'algorithme :

- Complexité de l'indexation : L'utilisation d'index pour placer les méthodes dans la table peut conduire à une complexité accrue, en particulier lorsque des méthodes sont redéfinies. L'ajustement des indices peut rendre le code moins intuitif et potentiellement sujet à des erreurs.

En conclusion, cet algorithme offre une approche structurelle pour générer une table des méthodes prenant en compte l'héritage et les redéfinitions. Cependant, il est important de prêter attention à la gestion des indices pour éviter des complications liées à la maintenance du code.

3.3.2 Calcul TSTO, ADDSP

Le calcul s'appuie sur une HashMap qui stocke chaque occurrence de variable, comme expliqué précédemment. Nous calculons également l'état futur de la pile après l'instanciation des classes. Il est important de noter que les variables enregistrées par registre ne doivent pas être prises en compte dans ce calcul. Pour ce qui est du calcul de TSTO, ADDSP à l'intérieur des méthodes, le calcul se base sur le même résonnement, en prenant aussi en compte les paramètres des variables courantes.

3.4 Autres algorithmes

Il existe également d'autres petits algorithmes, chacun ayant sa propre utilité. Bien que nous ne détaillions pas ces algorithmes ici, ils se concentrent sur l'optimisation des registres et le bon déroulement des opérations du compilateur, indépendamment du nombre de registres restants. Ces éléments complémentaires contribuent à assurer l'efficacité globale du compilateur.