Projet Génie Logiciel

Compilateur Decac

Manuel de conception

Équipe GL2

Étudiants : Élina Houdé, Julien Pinto Da Fonseca,

Léa Solo Kwan, Yin Xu

Année 2019/2020

Sommaire

Introduction

Ce manuel de conception s'adresse à tout développeur qui souhaiterait maintenir et/ou faire évoluer le compilateur. Celui-ci est essentiellement composé de la description de l'organisation générale de l'implémentation.

1. Architecture

1.1. Gestion des erreurs

Afin de simplifier la gestion des messages d'erreurs et la maintenabilité du code, il a été décidé de créer une classe dont le rôle est la centralisation des différents messages d'erreur pouvant être retournés à l'utilisateur. De ce fait, lorsqu'une erreur doit être déclenchée au sein du code, le message d'erreur approprié provenant de la classe statique ***ErrorMessages*** est passé en paramètre de l'erreur.

Chemin depuis la racine du projet :

src/main/java/fr/ensimag/deca/ErrorMessages.java

1.2. Codegen

Afin de mener à bien l'implémentation des différentes méthodes entrant dans le cadre de l'étape de génération de code, plusieurs classes personnalisées ont dû être créées. On retrouvera notamment les classes suivantes :

- ***RegisterManager*** : permet de gérer tout ce qui touche aux registres ;
- ***StackManager*** : permet de gérer tout ce qui touche à la pile ;
- ***LabelManager*** : permet de gérer la génération des labels ;
- ***LabelType*** : énumération composée des différents types de label ;
- ***ErrorLabelManager*** : permet de gérer les labels pour les erreurs concernant l'exécution du code. A la fin de la génération du code principal d'un programme, cette classe permet également de générer le code assembleur permettant la gestion de chaque erreur du programme ;
- ***ErrorLabelType*** : énumération composée des différents types de label d'erreur ;
- ***TSTOManager*** : cette classe permet le calcul du nombre maximum d'empilement nécessaire pour l'exécution d'un bloc d'instructions (corps d'une méthode ou corps du main). Ce calcul est effectué lors de la génération du code assembleur.

Chemin depuis la racine du projet :

src/main/java/fr/ensimag/deca/codegen/

1.2.1. RegisterManager

Cette classe comprend les méthodes publiques suivantes :

- ***resetNbMaxRegistersUsed****()* : réinitialise l'attribut *nbMaxRegistersUsed* à 0 ;
- ***getNbMaxRegistersUsed****()* : accesseur de l'attribut *nbMaxRegistersUsed*. Il s'agit d'un nombre entier indiquant le nombre maximum de registres utilisés simultanément depuis le dernier reset (méthode *resetNbMaxRegistersUsed()*);
- ***isTaken****(int regNumber)* : renvoie un booléen pour indiquer si le registre passé en paramètre est utilisé (true) ou libre (false) ;
- ***take****(int regNumber)* : réserve le registre qui est passé en paramètre et, si nécessaire, actualise l'attribut *nbMaxRegistersUsed* ;
- ***free****(int regNumber)* : libère le registre passé en paramètre ;
- ***nextAvailable****()* : renvoie le numéro du premier registre disponible ;
- ***getSize****()* : fonction qui renvoie le nombre total de registre.

De plus, elle comprend la méthode privée suivante :

- ***verifyRegNumber****(int regNumber)* : vérifie que le numéro de registre *regNumber* est bien compris entre 2 et le plus grand numéro de registre possible (donné par l'attribut *size* -1);

1.2.2. LabelManager

Cette classe comprend les méthodes publiques suivantes :

- ***getCurrentLabel****()* : accesseur de l'attribut currentLabel. *currentLabel* correspond au label de fin de la méthode qui est actuellement en train d'être traduite en assembleur. Il est utilisé notamment dans les instructions *return* ;

- ***setCurrentLabel****(Label label)* : mutateur de l'attribut currentLabel. *currentLabel* correspond au label de fin de la méthode qui est actuellement en train d'être traduite en assembleur. Il est utilisé notamment dans les instructions *return* ;
- ***getLabelValue****(LabelType label)* : renvoie l'entier associé au label passé en paramètre ;
- ***incrLabelValue****(LabelType label)* : incrémente de un la valeur associée au label passé en paramètre.

De plus, elle comprend la méthode privée suivante :

- ***verifyLabelType****(LabelType lt)* : méthode permettant de vérifier l'existence dans la map *myLabels* d'un élement lt donné de type *LabelType*.

1.2.3. ErrorLabelManager

Cette classe comprend les méthodes publiques suivantes :

- ***addError****(ErrorLabelType error)* : met à true le booléen associé à l'erreur passée en paramètre pour indiquer qu'elle devra être gérée ;
- ***errorLabelName****(ErrorLabelType error)* : retourne une chaîne de caractère correspondant au nom associé à l'erreur passée en paramètre. Ce nom est notamment utilisé comme label pour les instructions de saut ;
- ***printErrors****(DecacCompiler compiler)* : pour chaque erreur signalée précedemment par la fonction *addError(error)* on demande son affichage grâce à la fonction suivante ;
- ***printError****(DecacCompiler compiler, ErrorLabelType error)* : ajoute le code assembleur pour gérer l'erreur passée en paramètre.
-

De plus, elle comprend la méthode privée suivante :

- ***verifyLabelType****(ErrorLabelType lt)* : méthode permettant de vérifier l'existence dans la map *myLabels* d'un élément lt donné de type *ErrorLabelType*.

1.2.4. StackManager

Cette classe comprend les méthodes publiques suivantes :

- ***getInClass()*** : accesseur de l'attribut *inClass*. Cet attribut indique si les instructions qui sont actuellement en train d'être traduites en assembleur forment le corps d'une méthode (valeur true) ou le corps du main (valeur false) ;
- ***setInClass****(boolean inClass)* : mutateur de l'attribut inClass ;
- ***resetLB()*** : réinitialise l'attribut LB à 1 ;
- ***getGB()*** : renvoie le numéro associé à l'adresse de la prochaine variable globale (adresse de la forme x(GB) avec x un entier) ;
- ***getLB()*** : renvoie le numéro associé à l'adresse de la prochaine variable locale (adresse de la forme x(LB) avec x un entier) ;
- ***incrGB()*** : ajoute 1 au numéro associé à l'adresse de la prochaine variable globale (adresse de la forme x(GB) avec x un entier) ;
- ***incrLB()*** : ajoute 1 au numéro associé à l'adresse de la prochaine variable locale (adresse de la forme x(LB) avec x un entier).

1.2.5. TSTOManager

Cette classe comprend deux attributs permettant le calcul du nombre maximum d'empilement nécessaire pour l'exécution d'un bloc d'instructions : l'attribut current qui représente la taille courante de la pile et l'attribut *max* qui est la taille maximum de la pile.

Cette classe comprend également les méthodes publiques suivantes :

- ***resetCurrentAndMax()*** : réinitialise les attributs *current* et *max* à la valeur 0.
- ***addCurrent****(int i)* : ajoute la valeur *i* à l'attribut *current* et, si nécessaire, actualise l'attribut *max* ;
- ***getMax()*** : accesseur de l'attribut *max*.

2. Spécifications sur le code

Afin de déterminer rapidement le but de chacune des méthodes d'une classe, celles-ci respectent les conventions de nommage suivantes concernant la génération de code assembleur :

- ***codeGenInst****(DecacCompiler compiler)* : méthode réservant les registres nécessaires pour la génération du code assembleur d'une instruction produite grâce à l'appel à la fonction suivante ;
- ***codeGenInst****(DecacCompiler compiler, GPRegister register)* : méthode qui génère le code assembleur correspondant à une instruction et stocke le résultat de l'instruction dans le registre *register* ;
- **codeGenInstArith***(DecacCompiler compiler, GPRegister register1, GPRegister register2)* : méthode similaire à *codeGenInst* mais spécialisé pour les expressions binaires arithmétiques, car celle-ci nécessite l'utilisation de deux registres ;
- ***codeGenPrint*** : méthode qui affiche le résultat d'une expression, après avoir générer, si nécessaire, le code assembleur correspondant à cette instruction avec la méthode *codeGenInst* ;
- ***codeGenCMP****(DecacCompiler compiler, Label label, boolean reverse)* : méthode vérifiant si une condition est respectée lorsque *reverse* vaut false, et si elle n'est pas respecté lorsque *reverse* vaut true, i.e. réalisant la comparaison adéquate entre deux expressions et effectuant un saut conditionnel à un label donné ;
- ***codeGenError****(DecacCompiler compiler)* : méthode générant la condition de saut en cas d'erreur et signalant au ErrorLabelManager qu'il faudra générer le code permettant la gestion de cette erreur à la fin du fichier assembleur ;
- ***codeGenMethodTable****(DecacCompiler compiler)* : méthode qui construit et génère la table des méthodes ;
- ***codeGenMethodAndFields****(DecacCompiler compiler)* : méthode qui génère le code assembleur pour l'initialisation des variables et les méthodes d'une classe ;

- ***codeGenOperandAssign****(DecacCompiler compiler)* : méthode qui génère, si nécessaire, du code assembleur pour permettre l'accès à une variable (ex : champs d'une classe, variable locale, etc.), et qui renvoie l'adresse de type *DAddr* de cette variable.

Concernant les vérifications contextuelles, les méthodes respectent les conventions de nommage suivantes :

- ***verify(List)Class*** : méthode vérifiant contextuellement que la déclaration d'une classe est correcte lors de la passe 1 ;
- ***verify(List)XXXMembers*** : méthodes vérifiant contextuellement que l'élément de type XXX est valide lors de la passe 2. L'élément de type XXX peut représenter une classe, un champ, une méthode ou les paramètres d'une méthode ;
- ***verify(List)XXXBody*** : méthodes vérifiant contextuellement l'élément de type XXX lors de la passe 3. L'élément de type XXX peut représenter une classe, un champ, une méthode ou les paramètres d'une méthode ;
- ***verify(List)Inst*** : méthode qui vérifie contextuellement une instruction ou une liste d'instruction lors de la passe 3 ;
- ***verifyExpr*** : méthode qui vérifie contextuellement une expression lors de la passe 3 ;
- ***verifyYYY*** : méthode qui vérifie vérifie contextuellement un élément de type YYY lors de la passe 3. Cette méthode concerne toutes les éléments pour lesquels un des trois derniers types de méthodes ne peut être utilisé.

3. Algorithmes et structures de données

Afin de correspondre aux spécifications, les attributs suivants ont été ajoutés à la classe *DecacCompiler* :

- *RegisterManager registerManager ;*
- *StackManager stackManager ;*
- *LabelManager labelManager ;*
- *ErrorLabelManager errorLabelManager.*

De plus, afin de générer la classe Object, qui est la classe supérieure de toutes les classes, trois fonctions importantes ont été ajoutées dans la classe Program :

- ***declareObject****(DecacCompiler compiler)* : méthode qui déclare la classe Object et définit la méthode equals ;
- ***codeGenMethodTableObject****(DecacCompiler compiler)* : déclare la classe Object et ses méthodes dans la table des méthodes et génère le code assembleur correspondant ;
- ***codeGenMethodObject****(DecacCompiler compiler)* : génère le code assembleur pour la méthode equals de la classe Object.

4. Erreurs levées lors de la conception

Lors de la conception de nouvelles méthodes, des erreurs de type *DecacFatalError* peuvent survenir : il s'agit d'erreurs signalant une mauvaise utilisation de certaines classes. Ces erreurs ne sont pas sensées survenir lors de l'utilisation du compilateur par un utilisateur. Le code du compilateur doit donc être modifié en conséquence afin de les éviter.

Les erreurs de type *DecacFatalError* sont les suivantes :

- **"Numéro de registre inexistant : ..."**

Cette erreur peut apparaître lors de l'utilisation de la classe *RegisterManager*. Elle signale que vous essayez d'allouer ou de libérer un registre qui n'existe pas ou un registre scratch (registres R0 et R1).

N.B : Les registres scratch n'ont pas besoin d'être réservés.

- **"Un type de label inexistant a été utilisé."**

Cette erreur peut apparaître lors de l'utilisation de la classe *StackManager*. Elle signifie que le type de label que vous essayez d'utiliser n'existe pas. Elle peut survenir lorsque le type de label utilisé est nul ou qu'une mauvaise manipulation a été effectuée dans le code source du compilateur.

- **"Vous ne pouvez pas accéder à la définition de la classe suivante car elle n'est pas déclarée dans l'environnement des types : ..."**

Cette erreur peut apparaître lors de l'utilisation de la méthode *getClassDefinition* de la classe *environmentType*. Elle peut survenir lorsqu'une classe n'a pas été déclarée précédemment grâce à la méthode *declareClass*.

- **"Vous essayez d'accéder à la définition de classe de la variable suivante qui ne possède pas de définition de classe : "**

Cette erreur peut apparaître lors de l'utilisation de la méthode *getClassDefinition* de la classe *environmentType*. Elle survient lorsque vous souhaitez obtenir la définition de classe d'une variable mais que cette variable ne possède pas de définition de type *ClassDefinition*, soit parce que la variable n'est pas de type classe, soit parce que sa définition n'a pas été initialisée.