Conception

Ce document de conception de notre compilateur Deca décrit l'organisation générale de l'implémentation. Il complète la Javadoc qui comporte davantage d'information sur les classes et les méthodes de notre compilateur. Pour générer ce document il suffit d'exécuter la commande :

```
mvn javadoc:javadoc
```

La Javadoc sera alors accessible dans `target/site/apidocs/index.html`.

Sommaire

- 1. Architecture
  - 1.1. Gestionnaires de génération de code
    - 1.1.1. ErrorCatcher
    - 1.1.2. LabelManager
    - 1.1.3. StackManager
  - 1.2. Noeuds de l'arbre abstrait du deca objet
- 2. Spécification sur le code du compilateur
  - 2.1. Génération de l'instruction TSTO
  - 2.2. Génération des blocs d'initialisation et des blocs de méthode
- 3. Algorithmes et structures de données
  - 3.1. Vérification contextuelle des noeuds
  - 3.2. Génération de code du deca sans objet
    - 3.2.1 Déclaration de variable
    - 3.2.2. Initialisation
    - 3.2.3. Affectation
    - 3.2.4. Comparaisons
    - 3.2.5. Évaluation des expressions arithmétiques
    - 3.2.6. Évaluation des expressions booléennes
    - 3.2.7. Branchements IfThenElse
    - 3.2.8. Boucle While
  - 3.3. Génération de code du deca objet
    - 3.3.1 Table des méthodes
    - 3.3.2. Bloc d'initialisation d'objet
    - 3.3.3. Initialisation des champs
    - 3.3.4. Allocation d'objet
    - 3.3.5. Sélection de champs et de méthodes
    - 3.3.6. Bloc de méthode
    - 3.3.7. Return
    - 3.3.8. Appel de méthode
    - 3.3.9. Conversion de type et instanceof

1. Architecture

1.1. Gestionnaires de génération de code

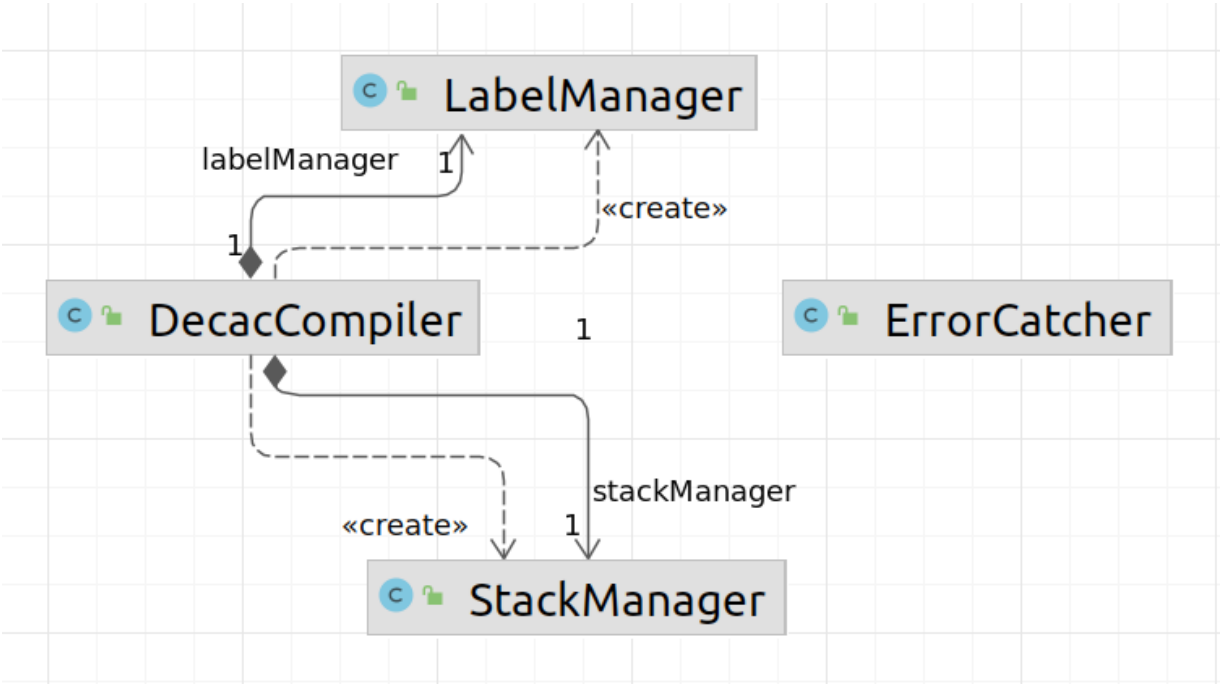

Diagramme de dépendances du package `fr.ensimag.deca.codegen`

Le package `fr.ensimag.deca.codegen` contient quatre classes gestionnaires qui facilitent l'étape de génération de code.

Ces gestionnaires sont en attributs (public final) dans la classe `DecacCompiler` (sauf pour `ErrorCatcher`).

1.1.1. ErrorCatcher

Il s'agit d'une classe non instanciable donc les attributs et les méthodes sont statiques. Elle facilite la gestion des erreurs d'exécution en :

- contenant le nom des `Label` d'erreur
- créant les `Label` d'erreur via le `LabelManager`
- générant le code assembleur gérant les erreurs (affichage d'un message d'erreur et arrêt du programme)

Ce dernier point ce traduit par une méthode qui est appelée par `DecacCompiler` après la génération de code du programme et qui génère le code gérant toutes les erreurs d'exécution.

Voir `/docs/Manuel-Utilisateur.pdf` pour la liste des erreurs d'exécution.

1.1.2. LabelManager

Il s'agit d'un dictionnaire associant le nom d'une étiquette (`String`) à un objet `Label`. Elle gère la création et la récupération de label pour des sauts de différents types. Pour cela on indexe plusieurs type de `Label` sous la forme :

- branchements : `"if_{i}"`, `"else_{i}"`, `"end_if_{i}"`
- boucles : `"while_{i}"`, `"end_while{i}"`
- évaluation de conditions : `"end_cond_{i}"`
- évaluation de `instanceof` : `"instanceof_trueBranch_{i}"`, `"instanceof_end_{i}"`
- fin de méthode : `end_method_{i}`

La concentration de ces opérations dans `LabelManager` permet d'éviter les doublons de `Label` et de les catégoriser.

1.1.3. StackManager

Cette classe gère l'initialisation et l'usage de la pile dans les blocs d'instructions (main, bloc d'initialisation d'objet ou bloc de méthode). Elle contient des compteurs avec les rôles suivants :

- `gbOffsetCounter` : incrémenter le pointeur GB lors de la génération de la table des méthodes
- `lbOffsetCounter` : incrémenter le pointeur LB lors de la génération de code des déclarations au sein des blocs
- `savedRegisterCounter` : compter le nombre de registres à sauvegarder en début de méthode
- `varCounter` : compter le nombre de variables déclarées dans un bloc ainsi que le nombre de temporaires nécessaires à l'évaluation d'expressions arithmétiques
- `maxMethodCallParamNb` : déterminer le nombre maximal de paramètres des méthodes appelées dans un bloc. Il est mis à jour à chaque ajout d'un `BSR` si le nombre de paramètres dépasse celui d'une méthode précédemment appelé dans le bloc

À la génération d'un nouveau bloc d'instructions, la méthode `resetStackCounters()` permet de réinitialiser les compteurs. En effet, les blocs sont générés à la suite et ne partagent pas donc par les informations associées aux compteurs.

Lors de la génération d'un bloc, on ajoute les objets représentant les instructions `TSTO` et `ADDSP` au programme du `DecacCompiler` en gardant leur référence. On met à jour les compteurs au fil des instructions ajoutées. Puis on met à jour l'attribut représentant l'argument des instructions `TSTO` et `ADDSP` via leur référence.

Les compteurs `savedRegisterCounter`, `varCounter`, `maxMethodCallParamNb` permettent de connaître la valeur de l'argument *d* de l'instruction `TSTO` à ajouter en début de chaque bloc. Le compteur `varCounter` permet de connaître la valeur de l'argument *d* de l'instruction `ADDSP` à ajouter en début de chaque bloc.

1.2. Noeuds de l'arbre abstrait du deca objet

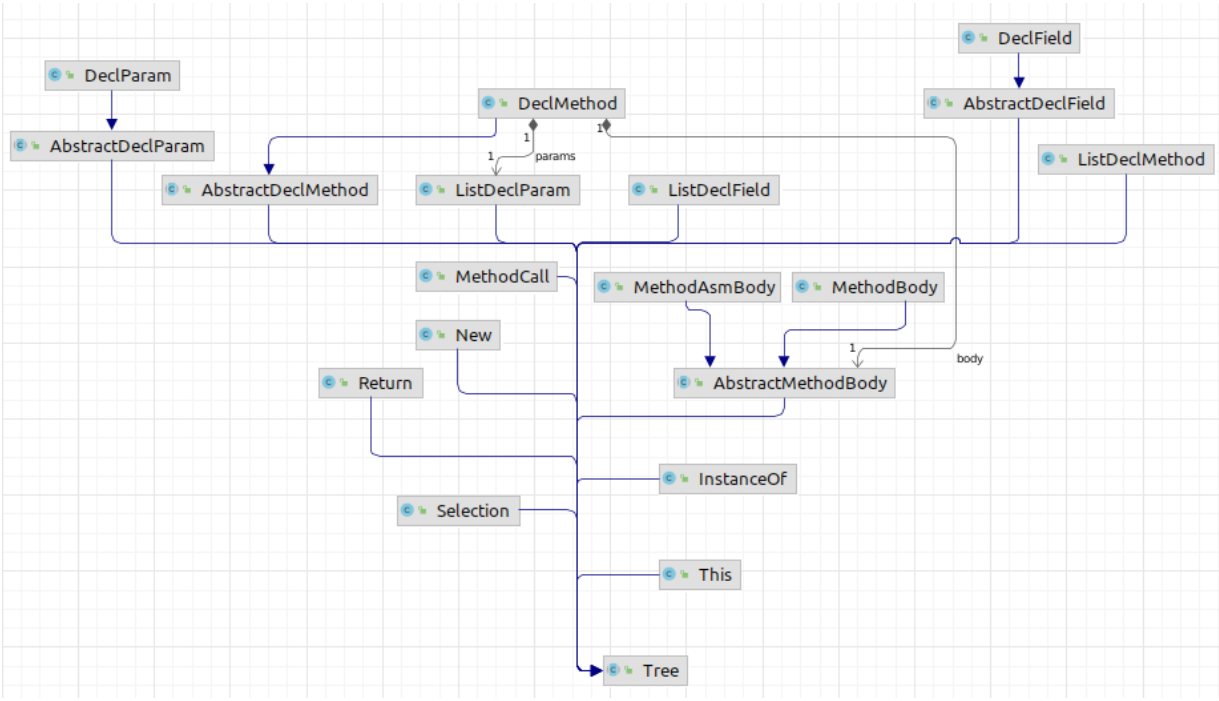

Diagramme de dépendances du package fr.ensimag.deca.tree avec seulement les classes supplémentaires

Ces classes sont celles que nous avons rajouté pour représenter les noeuds de l'arbre abstrait syntaxique pour la portion objet du langage deca. Elles implémentent de la même manière que les classes déjà existantes les méthodes de vérification contextuelle (une par passe et par classe), de génération de code, de décompilation, d'affichage de noeud et d'itération de noeud.

Voir la Javadoc pour des informations plus détaillées sur ces méthodes.

2. Spécification sur le code du compilateur

2.1. Génération de l'instruction TSTO

Le calcul de l'argument du `TSTO` donne un résultat supérieur à ce qui est réellement nécessaire pour réaliser les instructions d'un bloc. Cela provient du fait qu'au lieu de déterminer le maximum de temporaires nécessaires à l'évaluation d'expression arithmétique, on incrémente seulement le nombre de temporaire utilisé.

Concrètement, si une expression `e1` nécessite 5 temporaires et une expression `e2` nécessite 2 temporaires dans un même bloc, le `TSTO` aura un argument trop grand de 2.

Cela a pour conséquence que des programmes ayant beaucoup de temporaires puissent ne pas s'exécuter.

2.2. Génération des blocs d'initialisation et des blocs de méthode

On sauvegarde systématiquement tous les registres de `R2` à `R{MAX}` (2 < `MAX` < 16) au début de chaque bloc d'initialisation d'objet et des blocs méthodes pour des raisons de sûreté.

Il aurait été possible de déterminer les registres non-scratch utilisés dans chaque bloc pour ne sauvegarder que ceux-là mais ce n'était pas un point critique de notre compilateur car l'objectif principal n'était pas sa performance.

2.3. Implémentation de la classe Object

On définit `Object` dans l'environnement prédéfini `EnvironmentType` en tant que `ClassType`avec le `Symbol` "Object" associé.

On définit également au même endroit la méthode `equals(Object)` de la classe `Òbject` avec sa signature et on l'ajoute à l'environnement de cette classe.

En le faisant dans `EnvironmentType`, on se garantit l'existence d'`Object` lors de toutes les phases de la compilation. Cela permet notamment d'affecter par défaut `Object` en tant que super classe aux classes sans `extends` lors du parsing.

3. Algorithmes et structures de données

La génération de code d'une expression requiert un numéro de registre en paramètre pour stocker le résultat de l'expression. Elle se traduit par la méthode `codeGenExpr` et est implémentée par les noeuds héritant de `AbstractExpr`.

La génération de code d'une condition requiert un boolean indiquant si elle est positive ou négative et un Label vers lequel sauter si la condition est fausse. Elle se traduit par la méthode `codeGenCondition` qui est implémentée par les expressions pouvant renvoyer un booléen.

Pour faciliter la génération de code, on utilise les structures de données définies dans le package `fr.ensimag.deca.codegen`. Elles sont plus détaillées dans la section 1.1. Gestionnaires de génération de code.

3.1. Vérification contextuelle des noeuds

Chaque noeud de l'arbre abstrait contient des méthodes de vérification contextuelle pour vérifier les règles de [SyntaxeContextuelle]. Il y a autant de méthode par noeud que de passes de vérification nécessaire pour le noeud. Ainsi, si on prend l'exemple de `DeclClass`, la méthode `verifyClass()` implémente la passe 1, `verifyClassMembers()` implémente la passe 2 et `verifyClassBody()` implémente la passe 3.

Ces méthode de vérification appellent aussi les méthodes de vérification des noeuds fils. Cela évite d'implémenter plusieurs un même type de vérification.

De manière générale, on crée ou on met à jour les définitions (`FieldDefinition`, `MethodDefinition`, `ParamDefinition`, `VariableDefinition`) et les type (type prédéfini dans `EnvironmentType` ou un `ClassType`) associés aux noeuds lorsque les règles contextuelles sont respectées et on déclenche une erreur `ContextualError` si ce n'est pas le cas (voir /docs/Manuel-Utilisateur.pdf pour plus d'informations sur ces erreurs).

3.2. Génération de code du deca sans objet

Cette section décrit les différents algorithmes de génération de code du deca sans objet.

3.2.1 Déclaration de variable

`codeGenDeclVar:`

1. `codeGenInitialization:` Si la déclaration de variable comporte une initialisation, générer le code de l'initialisation (voir 3.1.2. Initialisation)
2. Associer la variable à une adresse de la pile de la forme *k*(LB)
3. Incrémenter *k* pour les déclarations suivantes via le `StackManager`

Cet algorithme est implémenté dans `DeclVar`.

3.2.2. Initialisation

`codeGenInitialization:`

1. `codeGenExpr:` Générer le code de l'initialisation avec l'algorithme de génération d'expressions sur l'expression avec laquelle on fait l'initialisation

Cet algorithme est implémenté dans `Initialisation`.

3.2.3. Affectation

`codeGenInst`:

1. `codeGenExpr:` Générer le code de l'opérande de droite et placer la valeur de cette expression dans `R2`
2. `codeGenLeftValue:` Générer le code pour récupérer l'adresse de l'opérande de gauche dans `R3`
3. Affecter `R2`, `R3`

Cet algorithme est implémenté dans `Assign`.

3.2.4. Comparaisons

Évaluation en tant que condition : On considère `l` le Label associé au code exécuté dans le cas où la comparaison est fausse `codeGenCondition:`

1. `codeGenExpr:` Générer le code de l'expression de gauche dans le registre R2
2. `codeGenExpr:` Générer le code de l'expression de droite dans le registre R3
3. Comparer R3 et R2
4. Si l'expression est un `Not`, on inverse seulement la valeur de vérité de la condition. Sinon, si l'expression est négative, faire un saut vers `l` si la comparaison est vraie (le choix du saut est fait dans les classes héritants de `AbstractOpCmp`)
5. Sinon, faire un saut vers `l` si la comparaison est fausse

Cet algorithme est implémenté dans `AbstractOpCmp`, `Not`, `Equals`, `NotEquals`, `Greater`, `GreaterOrEqual`, `Lower`, `LowerOrEqual`.

Évaluation en tant qu'expression : `codeGenExpr:` Voir 3.1.6. Évaluation des expressions booléennes

3.2.5. Évaluation des expressions arithmétiques

`codeGenExpr:` On distingue deux cas pour la génération de code des expressions arithmétiques :

- Si le registre R*x* dans lequel le résultat doit être est le dernier registre disponible

1. Incrémenter le nombre de temporaires nécessaires via le `StackManager`
2. `codeGenExp`: Générer le code de l'opérande de gauche et placer la valeur de l'expression dans R*x*
3. Sauvegarder dans la pile la valeur de R*x*
4. `codeGenExp`: Générer le code de l'opérande de droite et placer la valeur de l'expression dans R*x*
5. Sauvegarder dans R1 la valeur de R*x* (opérande de droite)
6. Restaurer la dernière sauvegarde de la pile dans R*x* (opérande de gauche)

- Sinon

1. `codeGenExp`: Générer le code de l'opérande de gauche et placer la valeur de l'expression dans R*x*
2. `codeGenExp`: Générer le code de l'opérande de droite et placer la valeur de l'expression dans R*x+1*

Après l'un de ces deux groupes d'instructions, on génère l'instruction pour l'opération arithmétique que l'on souhaite au sein des classes héritant de `AbstractOpArith`.

Dans le cas ou l'un des deux opérandes est un float, on ajoute une vérification de dépassement de flottant (overflow).

Pour le cas des des multiplications et des divisions, il y a également un risque dépassement de flottant par valeur inférieure (underflow). Pour cela ajoute des instructions pour vérifier si l'un des deux opérandes vaut zéro (seulement celui de gauche pour la division). Si ce n'est pas le cas et que le résultat vaut zéro, alors on déclenche une erreur `UD_ERROR`.

Cet algorithme est implémenté dans `AbstractOpArith`, `Divide`, `Minus`, `Modulo`, `Multiply`, `Plus`.

3.2.6. Évaluation des expressions booléennes

`codeGenExpr:` Il s'agit du même algorithme que pour un branchement IfThenElse. On évalue l'expression comme une condition via `codeGenCondition:` puis on considère l'affectation de 1 dans le registre de résultat comme étant le code du bloc "then" et l'affectation de 0 dans le registre de résultat comme étant le code du bloc "else.

Cet algorithme est implémenté dans `AbstractBinaryExpr`.

3.2.7. Branchements IfThenElse

`codeGenInst:`

1. Créer les Label `"if", "else", "end_if"` avec le `LabelManager`

2. `codeGenCondition:` Générer le code de la condition à évaluer en tant qu'expression positive et passer en paramètre le Label `"else"` qui servira au saut dans le cas où la condition est évaluée à fausse
3. `codeGenInst:` Générer le code des instructions du bloc "then"
4. Ajouter une instruction pour sauter le bloc "else" avec le Label `"end_if"`
5. Ajouter le Label `"else"` dans le programme
6. `codeGenInst:` Générer le code des instructions du bloc "else"
7. Ajouter le Label `"end_if"` dans le programme pour terminer l'instruction `IfThenElse`

À noter, le `else if` se traduit par l'ajout d'un noeud `IfThenElse` dans la liste d'instruction du bloc "else" du `IfThenElse` courant lors du parsing mais cela ne change pas l'algorithme du `IfThenElse` car il s'agit d'une instruction.

Cet algorithme est implémenté dans `IfThenElse`.

3.2.8. Boucle While

`codeGenInst:`

1. Créer les Label `"while", "end_while"` avec le `LabelManager`
2. `codeGenCondition:` Générer le code de la condition à évaluer en tant qu'expression positive et passer en paramètre le Label `"end_while"` qui servira au saut dans le cas où la condition est évaluée à fausse
3. `codeGenInst:` Générer le code des instructions du corps de la boucle
4. Ajouter une instruction pour reboucler vers la condition avec le Label `"while"`
5. Ajouter le Label `"end_while"` dans le programme pour terminer l'instruction `While`

Cet algorithme est implémenté dans `While`.

3.3. Génération de code du deca objet

Cette section décrit les différents algorithmes de génération de code du deca objet.

3.3.1 Table des méthodes

Il est nécessaire de respecter les points suivants :

- les adresses des cases de la table de méthodes sont sous la forme *k*(GB)
- il faut donc incrémenter *k* pour chaque ajout d'entrée dans la table via le `StackManager`
- les Label des méthodes sont sous la forme `code.<className>.<methodName>`
- il faut stocker l'adresse de chaque classe dans la table dans sa `ClassDefinition`

`codeGenListDeclClass:`

1. `codeGenDeclClassObject:` Affecter `null` à la première case de la table (super classe de Object qui est nulle). Il s'agit du début du chaînage des classes
2. `codeGenDeclClassObject:` Affecter le Label de la méthode `equals` de Object dans la deuxième case.
3. `codeGenDeclClass:` Pour chaque déclaration de classe :
    - Charger dans la table le pointeur vers la super classe
    - `codeGenListDeclMethod:` Charger dans la table les Label de toutes les méthodes de la super classe qui ne sont pas redéfinies et toutes les méthodes de la classe

Concernant le dernier point, on recherche les méthodes par index dans l'environnement de la classe (objet `EnvironmentExp`). Cet environnement contient aussi l'environnement de la super classe qui lui-même contient l'environnement de sa super classe, etc. Donc on recherchera aussi les méthodes dans la super classe si elles n'ont pas été redéfinies.

Cet algorithme est implémenté à travers les classes `ListDeclClass`, `DeclClass`, `ListDeclMethod` et `EnvironmentExp`.

3.3.2. Bloc d'initialisation d'objet

Les labels des blocs d'initialisation d'objet sont sous la forme `init.`.

`codeGenClassInit`:

1. Tester que la pile ne déborde pas via un `TSTO` et déclencher une erreur si c'est le cas
2. Allouer de la place pour les variables locales via un `ADDSP`
3. Sauvegarder les registres `R2` à `R{MAX}` (2 < `MAX` < 16)
4. Appeler `init.` si la classe hérite d'une autre classe (voir 3.2.6. Appel de méthode)
5. `codeGenListDeclField:` Générer le code de la déclaration des champs
6. Restaurer les registres `R2` à `R{MAX}` (2 < `MAX` < 16)
7. Ajouter l'instruction `RTS` pour quitter le bloc d'initialisation

Cet algorithme est implémenté à dans la classe `DeclClass`.

3.3.3. Initialisation des champs

`codeGenDeclField:`

1. `codeGenInitialization:` Générer le code pour affecter la valeur d'initialisation dans le champ. On passe le type du champ en paramètre pour faire une initialisation par défaut s'il n'y a pas de valeur spécifiée.
2. Sauvegarder dans le tas la valeur du champ à l'index du champ précédemment déterminée lors de la vérification contextuelle.

Cet algorithme est implémenté à travers les classes `DeclField`, `Initialization` et `NoInitializationn``.

3.3.4. Allocation d'objet

`codeGenExp:`

1. Allouer de la mémoire dans le tas pour un objet *n* champs (champs du super inclus) dans le tas via un `NEW` et tester qu'il n'y a pas de débordement de tas
2. Stocker l'adresse de la classe présente dans la table des méthodes dans la première case de l'espace du tas alloué
3. Appeler `init.<className>` pour initialiser les champs de l'objet (voir 3.2.6. Appel de méthode)

Cet algorithme est implémenté dans la classe `New`.

3.3.5. Sélection de champs et de méthodes

Selection implicite

Par défault, chaque identificateur présent dans une méthode est une **Selection** implicite via l'expression **This**.

Exemple :

```
class A {
    int x;
    void method1(){
        method2();    // this.method2()
        x = 2;        // this.x = 2
    }
    void method2(){}
}
```

Ce n'est plus le cas dans un bloc d'une methode quand il y a une variable ou un paramètre avec le même nom. Dans ce cas-là, si on veut accéder au champ de la classe, il faut une référence **This**.

```
class A {
    int x = 1;
    void method(int x){
        this.x = x + 1;     // this.x: champ de la classe
                            // x: paramètre
```

```
    }
    void method2(){
        int x;
        this.x = x + 1;     // this.x: champ de la classe
                            // x: paramètre
    }
}
```

Sélection d'un champ ou d'une méthode

Le but de deux étapes ici est de récupérer l'indice ***index*** de champs/méthodes avec l'adresse ***addrObjet*** de l'objet dans la pile :

Au sein du main	Au sein d'une méthode (ou du constructeur)
Chercher la classe (type dynamique) de l'objet dans l'environnement des types et extraire son environnement	Extraire l'environnement de la classe courante (type statique)
Chercher le Symbol associé au champs/méthode dans l'environnement de la definition de la classe et récupérer son indice par rapport au type statique. **Pour des méthodes** : La recherche va prioritiser la méthode dans la classe courante	Chercher le Symbol associé au champs/méthode dans l'environnement de la definition de la classe et récupérer son indice par rapport la classe courante. **Pour des champs** : La recherche va prioritiser des variables locales et des paramètres **Pour des méthodes** : La recherche va prioritiser la méthode dans la classe courante

Après avoir récupéré l'indice ***index*** de champ/méthod et l'adresse ***addrObjet*** de l'objet dans la pile :

Lors d'une lecture/écriture d'un champ

L'accès au champ se fait à partir de l'emplacement de la classe dans la pile avec un decalage de son indice, à l'adresse : `addrObjet + index`

Lors d'un appel de méthode

1. Récupérer l'adresse de la classe dans la table des méthodes qui est rangée dans la première case de l'emplacement du tas alloué à l'objet :

   ```
   addrMethod <- VAL[addrObjet]              // VAL[x]: lecture de la pile à
   l'adresse x
   ```

2. Appel de méthode en utilisant l'indice de la méthode. L'étiquette de la méthode est rangé à l'adresse `addrMethod + index`

3.3.6. Bloc de méthode

Les Label des méthodes sont sous la forme `code.<className>.<methodName>`.

`codeGenMethodBody:`

1. Créer un Label `"end_method"` via le `LabelManager`
2. Tester que la pile ne déborde pas via un `TSTO` et déclencher une erreur si c'est le cas
3. Allouer de la place pour les variables locales via un `ADDSP`
4. Sauvegarder les registres `R2` à `R{MAX}` (2 < `MAX` < 16)
5. `codeGenListDeclVariable:` Générer le code des déclarations de variables locales
6. `codeGenListInst:` Générer le code des instructions de la méthode. Ce bloc devrait contenir une instruction `return` si la méthode n'est pas de type `void`
7. Ajouter une instruction qui déclenche une erreur `RET_ERROR`. Ce code ne devrait s'exécuter que s'il n'y a pas eu de `return`
8. Ajouter un label `"end_method"` au programme (voir 3.2.6. Return)
9. Restaurer les registres `R2` à `R{MAX}` (2 < `MAX` < 16)
10. Ajouter l'instruction `RTS` pour quitter le bloc de méthode

Cet algorithme est implémenté dans la classe `MethodBody`.

3.3.7. Return

`codeGenInst:`

1. `codeGenExp:` Générer le code de l'expression retournée
2. Charger le registre contenant le résultat de l'expression dans `R0`
3. Sauter vers le Label `"end_method"` créé par le `codeGenMethodBody`.

Cet algorithme est implémenté dans la classe `Return`.

3.3.8. Appel de méthode

`codeGenExp:`

1. Allouer de la place dans la pile pour les passages paramètres (l'implicite qui est l'objet duquel on appelle la méthode et les explicites qui forment la signature) via un `ADDSP`
2. Empiler les paramètres de la méthode dans 0(SP), -1(SP), ..., -*n*(SP) avec *n = nombre de paramètres explicites*
3. Vérifier que le paramètre implicite n'est pas null (déréférencement de null) et déclencher une erreur `NULL_ERROR` si c'est le cas
4. Récupérer l'adresse de la méthode dans la table des méthodes via le pointeur se trouvant dans la première case de l'espace du tas alloué à l'objet et l'index de la méthode
5. Appeler la méthode via un `BSR` vers l'adresse de la méthode
6. Dépiler les paramètres de la méthode
7. Charger le contenue de R0 (valeur de retour) dans le registre spécifié pour l'évaluation de l'expression

Cet algorithme est implémenté dans la classe `MethodCall`.

3.3.9. Conversion de type et instanceof

Conversion implicite

1. Opération entre float et int Les deux lignes en-dessous sont équivalentes

```
int x = 5
float y = x + 5;
```

```
int x = 5
float y = (float)(x) + 5;
```

2. Affectation avec des objets Lors d'une affectation d'un nouvel objet de type B dans une variable de type A avec B un sous-type de A, nous avoons un **cast implicite**. Les deux lignes en-dessous sont équivalentes:

```
A a = new B();
A a = (A)(new B());
```

Instruction instanceof et le cast explicit

L'idée: Pour ces deux instructions, il nous faudrait verifier que le type dynamique de l'expression que l'on souhaite cast (vérifer pour le cas de instanceof) correspond au type par lequel on veut cast (vérifier). Lors de la vérification contextuelle, nous n'avons pas d'information sur le type dynamique de l'objet. C'est pourquoi, pendant l'exécution, nous devons comparer l'adresse du tableau des méthodes du type dynamique de l'objet avec celle de la classe attendue.

Parcours de la table des méthodes pour vérifier que la classe C1 de l'expression *a* de type dynamique C2 est sous-type de l'autre

"Adresse de la classe" sous-entends "son adresse dans la table des méthodes"

Entrée : l'adresse de ***a*** (nous ne connaissons pas son type dynamique), l'adresse de **C1**

1. Load l'adresse de l'objet **A** dans **Rx**
2. Load dans **Rx** l'adresse de la classe **C2** qui est à l'adresse contenue dans **Rx**
3. Load l'adresse de la classe **C1** dans **Ry**
4. Si **Rx** est null, la vérification est évaluée à fausse (c'est la superclasse de **Object**)
5. Si **Rx** == **Ry**, la vérification est évaluée à vraie
6. Load le contenu à l'adresse présente dans **Rx** (adresse de sa superclasse) dans **Rx**
7. Retourner à l'étape 4

Algorithme pour *x* instance of C

1. Si l'expression *x* est de type **statique Null**, nous renvoyons la valeur **true** (car null est un sous-type de toutes les classes)
2. Si l'expression *x* est de type **dynamique Null**, nous renvoyons la valeur **true** (car null est un sous-type de toutes les classes)
3. Nous appliquons la vérification ci-dessus pour l'expression ***x*** et la classe **C** pour vérifier le sous-typage
4. Si la vérification est valide, (***x*** instance of **C**), nous renvoyons la valeur **true** sinon **false**

Algorithme pour le cast (C)(*x*)

Cas 1 : Cast int -> float ou float -> int

Nous convertissons l'expression en utilisant les instructions **FLOAT** ou **INT**

Cas 2 : Cast pour des objets

1. Si l'expression *x* est de type dynamique **Null**, nous renvoyons l'erreur de `CAST_ERROR`
2. Nous appliquons la vérification ci-dessus pour l'expression ***x*** et la classe **C** pour vérifier la compatibilité du cast.
3. Nous renvoyons comme valeur l'adresse de l'expression