Documentation Conception

gl58

January 25, 2017

1 Implémentation de l'analyse lexicale et syntaxique

1.1 Les classes supplémentaires par rapport à la documentation actuelle

Le schéma en page suivante représente l'arbre de relation des différentes classes du package tree de ce projet.
Les relations d'extension sont représentés par des liens, avec, quand le sens de la relation d'extension est ambigu, un rectangle au bout du lien, du côté de la superclasse. Ainsi, AbstractIdentifier étend AbstractLValue.

En bleu sont représentés les classes qui existaient déjà partiellement dans la notice précédente. En jaune sont les classes que nous avons récemment créées.

- AbstractDeclMethod, DeclMethod, AbstractDeclParam, DeclParam, AbstractDeclField, DeclField, ont dû être créées pour implémenter le concept de classe.
  Ces derniers héritent de leur abstract associés dans 3 des cas, ou de Tree directement dans le cas des abstracts. Les règles les concernant sont déjà décrites dans la documentation précédente.

- ListDeclMethod,ListDeclParam et ListDeclField ont été par conséquent créées, pour conserver une certaine homogénéité dans l'organisation de l'arbre et de l'analyseur syntaxique. Ils héritent donc de TreeList.
  Une astuce de code similaire à celle utilisée pour ListDeclVar a été utilisée pour ListDeclField, c'est-à-dire utiliser du passage de paramètre descendant vers la règle decl_field, pour fournir les paramètres de visibilité et la liste à remplir.

- L'implémentation des méthodes a nécessité de créer le noeud-classe Return.

- This, Null, New et InstanceOf ont été créés comme conséquence de l'apparition du concept de classe.

- L'accès aux paramètres et les appels aux méthodes ont une construction particulière :
  CallMethod est un noeud représentant un appel à une méthode indépendamment de ce qui le précède, c'est-à-dire qu'il existe que ce soit pour un appel dans une méthode d'une classe qui appelle une autre des méthodes de cette classe (ou superclasse), ou un appel de méthodes spécifiant l'instance de classe à laquelle il est appliqué.
  Dans le second cas, l'appel x.f(?) est représenté par une construction DotMethod ayant pour attribut un noeud CallMethod.
  Le noeud Dot sert à accéder aux paramètres et hérite d'AbstractLValue.

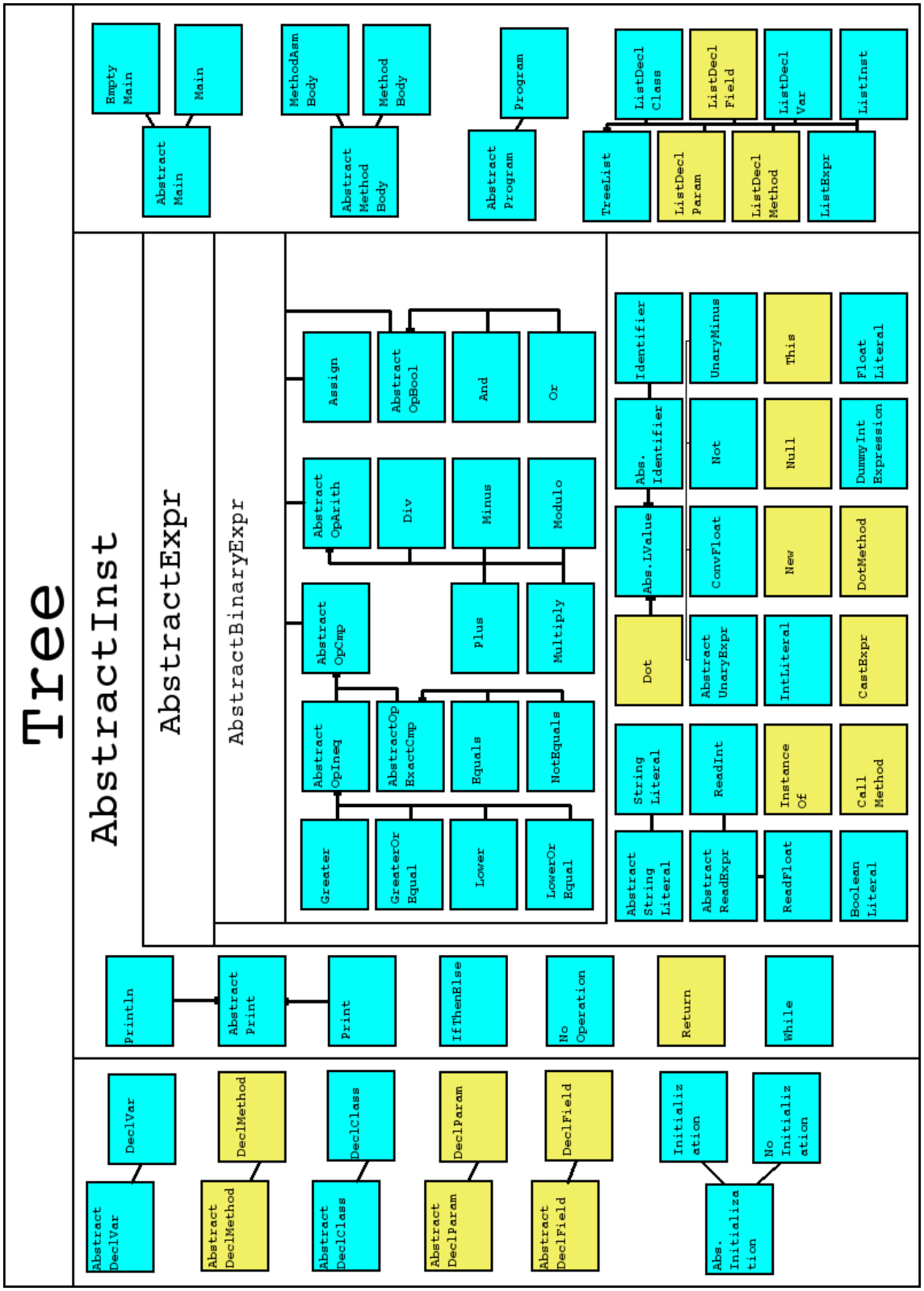

Figure 1: Organisation du package tree.

1.2 Information utile : la gestion des strings

Le lexème STRING, en sortie d'analyseur, contient les guillemets au début et à la fin de la string, qui ont servis à l'identifier.
Ce problème est géré dans le parser, lors de l'analyse de StringLiteral, à l'aide de la ligne :
`$tree=new StringLiteral( $str.text.substring(1,$str.text.length()-1) );`

2 Implémentation de la vérification contextuelle

2.1 `EnvironmentExp`

`EnvironmentExp` est une structure de données permettant de stocker des symboles et les définitions leur correspondant.

On retrouve cette class dans `ClassDefinition`, ou elle contient les symboles correspondant aux champs et aux méthodes de classe (ainsi que leur `FieldDefinition` et `MethodDefinition`).
Lors de la vérification contextuelle, on crée aussi de nouveaux environnements, notamment lorsqu'on entre dans le main du programme ou le corps d'une méthode. Cette environnement créé servira de référence pour vérifier si un symbole existe dans le contexte où on se situe (il faut donc, bien entendu, adapter l'`EnvironmentExp` en fonction de l'emplacement dans le programme).

De plus, pour l'implémentation de la grammaire avec objets, il a fallu modifier la méthode `get` de cette classe, pour qu'elle est itère sur les environnements parents de l'`EnvironmentExp`. En effet il est parfois nécessaire de regarder l'environnement de super-classes, pour gérer les champs et méthodes hérités par exemple.

2.2 L'environnement types

L'environnement types, présent dans la classe `DecacCompiler` sous la forme du champ `envTypes` permet de garder en mémoire le nom et la définition des types prédéfinis en langage Deca et les classes déclarées par un programme deca.

On a décidé d'implémenter `envTypes` comme une `HashMap` et non pas un `EnvironmentExp` car `EnvironmentExp` comme il etait defini demande des `ExpDefinition` (incompatible avec les `TypeDefinition` et `ClassDefinition` dans l'environnement types) et car l'environnement types n'hérite d'aucun environnement parent, comme la plupart des `EnvironnementExp` (l'environnement d'`Object` contenant la méthode `equals` exclu).

L'environnement des types et l'environnement de la classe `Object` sont initialisés par la méthode `initSymbolsAndEnvTypes`, qui elle même est appelée par `doLexingAndParsing` qui construit l'arbre du programme.

2.3 La vérification contextuelle

1. Grammaire sans objets

   L'analyse contextuelle de l'arbre donné par le parser se fait avec les méthodes `verifyXyz` des classes de `src/main/java/fr/ensimag/deca/tree/`. Chacune de ces méthodes appel les méthodes `verifyXyz` de ces arbres fils (`verifyMain` qui appel `verifyListDeclVar` et `verifyListInt` par exemple) et vérifie que les règles de grammaire correspondant à ce noeud de l'arbre sont bien respectées. Dans le cas échéant, une `ContextualError` est levée.

   Pour la grammaire sans objets, on a pas besoin de ce préoccuper des passes, car il n'y a pas de déclaration de classe a gérer.

2. Grammaire avec objets

   Pour la grammaire avec objets du langage Deca, il faut ajouter a certains noeuds de l'arbre (notamment les arbres liées a la déclaration de classes, comme `DeclMethod`, `DeclField` et

DeclParam) quelques méthodes verify pour gérer les 3 passes de vérifications á faire.

- Passe 1
  - verifyListClass
    Cette méthode appelle le verifyClass de chacun des éléments de ListClass et, une fois ces itérations terminées, appelle verifyListClassMembers de cette liste pour commencer la passe 2.
  - verifyClass
    Cette méthode de DeclClass vérifie que la super classe est déjà déclarée, vérifie que la classe déclarée ne le soit pas déjà et en l'absence d'erreurs ajoute la classe a l'environnement types du compilateur.
- Passe 2
  - verifyListClassMembers
    Cette méthode appelle le verifyClassMembers de chacun des éléments de ListClass et, une fois ces itérations terminées, appelle verifyListClassBody de cette liste pour commencer la passe 3.
  - verifyClassMembers
    Cette methode de DeclClass appelle les methodes verifyListField et verifyListMethod qui appellent respectivement le méthodes verifyDeclField et verifyDeclMethod de leurs éléments.
  - verifyDeclField et verifyDeclMethod
    Ces méthodes vérifient si les champs et méthodes d'une classe écrasent les champs ou méthodes de ces super-classes, initialisent les définitions de ces champs et méthodes et les ajoutent a l'environnement members de la ClassDefinition.
    verifyDeclMethod appelle aussi la méthode verfyDeclParam des paramètres de la méthode, qui vérifie leur type et les ajoute a la signature de la méthode.
- Passe 3
  - verifyListClassBody
    Cette méthode appelle verifyClassBody de chacun des éléments de ListClass.
  - verifyClassBody Cette méthode de DeclClass appelle les méthodes verifyListInit et verifyListBody qui appellent respectivement les méthodes verifyFieldInit et verifyMethodBody de leurs éléments.
  - verifyFieldInit et verifyMethodBody
    Ces méthodes vérifient contextuellement l'initialisation des champs et le corps des méthodes, en prenant comme environnement local l'environnement de la classe (et les paramètres de la méthode dans le cas de verifyMethodBody).

  Pour le reste de la passe 3, la vérification contextuelle est à peu près identique a celle de la grammaire sans objets. Quelques expressions ont ete ajoutees (comme CastExpr et InstanceOf) et certaines conditions ont été ajoutées aux verify de classes existantes pour prendre en compte les classes.

  La seul modification délicate entre la grammaire avec objets et la grammaire sans objets est le paramètre localEnv dans les méthodes verifyXyz. Dans le Main, celui-ci doit être un nouvel environnement, héritant de l'environnement de la classe Object et contenant les variables définies dans le main. Dans un MethodBody, localEnv sera un environnement contenant les paramètres de la méthodes et les variables définies dans le corps, et héritant de l'environnement de la classe.
  Ces nuances sont délicates car il faut faire en sorte que la vérification des expressions se fasse correctement quel que soit l'environnement local et si ces environnements sont mal gerés, des symboles qui devraient être reconnues ne le seront pas et des erreurs inattendues surgiront.

2.4 La décoration de l'arbre

Dans les noeuds de l'arbre qui descendent de AbstractExpr, il faut ajouter un type et dans le cas des Identifier une définition, pour qu'ils apparaissent lorsque prettyPrint est appelé.

Pour obtenir ces informations, les méthodes `verifyExpr` (ou `verifyType`) vont chercher dans l'`EnvironmentExp` local (resp. l'`envTypes` de `DecacCompiler`) la définition correspondant au symbole de l'identifier et le type donné dans cette définition.

Les expressions ne correspondant pas à des feuilles de l'arbre auront un type qui dépend soit du type de ses opérandes (par exemple les appels de méthodes) soit de l'expression en elle-même (par exemples les comparaisons qui seront toujours des booléens).

2.5 L'enrichissement de l'arbre

L'enrichissement de l'arbre consiste à convertir les entiers en flottants à l'aide du noeud `ConvFloat` lorsque cela est nécessaire.

Cette manipulation est faite dans la méthode `verifyRValue` de la classe `AbstractExpr`. En temps normal, si le paramètre `expectedType` de cette méthode et le type de l'expression sont différents, la methode renvoie la `ContextualError : expected different type`. Mais, dans le cas ou le type de l'expression est un `IntType` et le type attendu est un `FloatType`, le noeud `AbstractExpr` en question sera remplacé par un noeud `ConvFloat` ayant pour opérande cette expression.

2.6 Limitations de notre implémentation

Une des limites de notre analyse contextuelle est que le programme va lever une erreur (`method already defined` ou `field already defined`) si un champ et une méthode ont le même nom (il n'est d'ailleurs pas clair si ce cas doit être possible en Deca ou pas, mais il l'est en Java).
Ceci est du aux structures d'`EnvironmentExp` et de `ClassDefinition`. `ClassDefinition` contient un seul `EnvironmentExp` nommé `members` qui regroupe les champs et les méthodes. Or, comme un `EnvironmentExp` ne peut avoir qu'une seule fois le même symbole, il y aura une `DoubleDefException` si on tente d'ajouter une méthode a `members` alors qu'un champ dans `members` a le même nom (car il ne peut y avoir qu'un seul symbole par `String` dans une même table de symboles).

Une solution possible serait de créer les symboles a partir de `String` qui ne correspondent pas a leur nom, mais une combinaison de leur nom et de leur définition (`"a_field"` et `"a_method"` par exemple). Mais cette solution n'est pas concevable dans notre structure actuel. En effet, le compilateur reprend exactement la même table de symbole que celle qu'utilise le parser. Or le parser ne pourrait pas faire la différence entre une méthode et un champ lorsqu'il définit un symbole.

La seule autre solution possible serait de changer la structure de `ClassDefinition` en séparant les environnements des champs et des méthodes. L'inconvénient de cette solution est que la structure résultante serait plus compliquée à utiliser lorsqu'on faudra vérifier si un `Identifier` appartient à l'environnement local, car cela augmenterait le nombre de cas à traiter.

3 Implémentation de la génération du code

Pour la partie de gestion du code nous avons fait le choix de respecter le squelette de code fourni. C'est à dire que nous avons fait le choix d'appliquer des fonction génératrice de code en parcourant l'arbre fourni par la partie B. Pour cela nous avons implémenté des méthodes `codeGen`.

3.1 Principes de fonctionnement général

3.1.1 Principe de fonctionnement de la mémoire

Nous avons à notre disposition n registres avec $4 \leq n \leq 16$. Ainsi nous avons décidé pour optimiser l'exécution du programme d'utiliser un maximum de registres lors de l'exécution du programme. Pour cela nous disposons dans la classe `DecacCompiler` d'un ensemble de méthodes d'allocation de mémoire. Elle permettent de gérer l'allocation de la mémoire libre à instant T. Nous avons décider de garder en réserve les registres $R0$, $R1$ et $R2$ servant respectivement de registre temporaire, de registre d'affichage et registre temporaire, et de registre temporaire pour les classes. Les autres

registres peuvent être alloué pour représenter une variable ou une valeur dans un calcul. Par commodité nous libérons toute la mémoire après chaque ligne de calcul.

3.1.2 Principe de fonctionnement de `codeGen`

Nous avons choisis de modifier la structure de la méthdode `codeGen`. Cette méthode maintenant retourne une classe `DVal`. Cette classe retourné comporte la valeur retourné par le calcul effectué lors de l'exécution des commandes généré par `codeGen`. Ainsi cette valeur retourné par `codeGen` peut prendre 2 aspects. Elle est soit un registre qui contient le résultat du calcul précédent. Soit une adresse de la pile contenant le résultat.

3.1.3 Fonctions assembleur

Nous avons décider de factoriser un maximum le code des fonctions du même type. Ainsi nous avons décidé de créer les fonctions de génération de code pour chaque type de fonction assembleur. Chacune de ces fonctions prend comme paramètre un constructeur de classe du package `pseudocode.instruction`. Ainsi nous avons une seul fonction pour générer tout un ensemble de fonction assembleur. Ces classes génératrices sont dans le package `deca.codegen` dans les classes du type `codeGen...`

3.2 Le code sans Objets

3.2.1 Les fonctions d'assignation et d'affichage

Il existe deux type de fonction de calcul noeuds d'origine:

- Fonctions d'affichage
- Fonctions d'assignation

Le principe de génération du code est la même. Il faut pour chacune des fonction générer le code du calcul et le mettre dans le registre ou la variable correspondante. Chancune des fonctions utilise les fonctions de génération de code des fonctions feuille d'elle même dans l'abre. Elle recupère alors le résultat de ces calcul dans les plages mémoire désigné par les `DVal` de retour des fonctions `codeGen`. Il suffit donc de placer le résultat dans la variable pour assigner le résultat ou de placer le résultat dans $R1$ pour pouvoir l'afficher. Pour une question d'optimisation de la mémoire utilisé lors de l'affichage chaque classe pouvant mener à un affichage possède une méthode `codeGen` et une méthode `codeGenPrint` se chargeant respectivement de la génération du code et de l'affichage

3.2.2 Le calcul

Comme expliqué précédement, pour effectuer la génération d'un code d'une fonction binaire ou unaire dont on placera le résultat dans un registre ou dans la pile: il suffit de générer le code des deux sous branches de notre noeud et d'appeller les méthodes de génération dans le package `deca.codegen`. Cette façon de calculer le résultat d'une opération est la même que le calcul ce fasse sur un `int`, un `float` ou un `boolean`. Ce qui pose un problème pour les booléens évaluer pour des conditions de boucles ou des conditions.

3.2.3 Les boucles et les conditions

Pour la gestion des boucles nous nous sommes eurté à deux problèmes:

- Évaluation des opérations booléennes
- Gestion des B*cc* et forme des boucles et conditions

Pour l'évaluation des opérations booléennes nous avons choisi la même architecture que pour la généraion du code d'affichage. Ainsi pour optimiser l'éxécution des conditions booléennes, chaque opération suséptible de générer comme sortie un booléen possède deux façon de générer du code. La méthode `codeGen` expliqué plus haut et la méthode `codeGenCond` qui prend comme paramètre un label et un booléen. Cet dernière choisi de sauter au label si le résultat de l'opération généré est égal au booléen passé en paramètre.

Pour la forme des conditions nous avons choisi la forme classique d'un `if` en assembleur. C'est à dire une structure en un seul saut quelques soit l'issue de l'évaluation de la condition. Si la condition est réussi, le saut n'est pas effectué, puis l'on sautera la partie `else`. Si la condition est raté on sautera directement à la partie `else`. L'avantage de cette solution est qu'elle est très lisible (quand on relie le code assembleur) et une des plus efficace.
Pour la forme des boucles nous avons choisi de transformer nos `while` en `do while` afin que le code généré soit le plus efficace possible. Ainsi la première action de chaque boucle est de sauter à la fin et d'évaluer la condition. Si la condition est respecter alors on saute sur le label de début des instructions. Sinon on sort de la boucle en allant à l'instruction suivante.

3.3 Les Classes

L'objectif de l'implémentation des classes été de ne pas avoir à trop modifier le code sans objets. Ainsi nous avons opté pour un implémentation très simple où chaque instance de classe est traité de la même façon qu'une variable ordinaire. Ainsi nous avons pus conserver notre code de la partie précédente presque inchangé. De gros problèmes ce sont surtout posé lors de l'implémentation des méthodes.

- **Calcul de la table des méthodes:**
  Pour des raisons de commodité nous effectuons ces calculs dans le package `deca.context` dans la classe `ClassDefinition`. En effet chaque définition de classe possède un liens vers sa `super`. Ainsi chaque définition possède une méthode appelé `write`. Cette méthode appelle la création de la table des méthodes de la `super` si ce n'est pas fait, crée la table des méthodes de la classe si ce n'est pas fait puis renvoie l'adresse de la classe dans cette table des méthodes. Ainsi si l'on souhaite récupérer l'adresse de notre classe dans la table des méthode il faut utiliser la méthode `write`.

- **Création de la méthode d'initialisation:**
  La méthode d'initialisation se comporte comme la déclaration des variables. En effet elle passe juste en revue tout les champs d'une méthodes et vérifie si ils ont une valeur initiale donné. Si c'est le cas alors elle initialise le champs. Sinon elle ne les initialise pas. Nous précisons que la fonction d'initialisation n'est pas dans la table des méthodes car en `deca` les extension de classes n'héritent pas de la méthode d'initialisation.

- **Les Méthodes:**
  Pour pouvoir compiler les méthodes plus simplement j'ai décidé de modifier le code fourni dans le package `ima.pseudocode.instructions`. Les modifications apporté sont des amélioration permettant de modifier les valeurs d'une instruction à tout moment dans le programme pourvus que l'on ai gardé un pointeur vers cette instance. Par exemple il est désormais possible de modifier la valeur du `TSTO` après avoir écrit la méthode. J'ai aussi rajouté un moyen de rajouter un flux de commande sur un flag de la liste (le flag doit cependant être unique en action). Cette amélioration permet de faire la sauvegarde des registres uniquement si ils sont utilisé dans la méthode et ainsi d'économiser du temps d'exécution. Cependant il existe un bug d'origine inconnu qui fait sauvegarder plus de registre qu'utilisé. Les soupçons ce portent autour de la façon dont le compilateur est reset à chaque création de méthodes.
  Le principe de compilation d'une méthodes est pour le reste assez ressemblant à la méthode de compilation du main.

- **Les Paramètres,Variables et Champs de même nom:**
  Pour gérer les paramètres,variables et champs de même nom nous avons décider que nous vérifierons si la variable/paramètre est dans la table des variables. Sinon ce champs est considéré comme appartenant à la classe. Ainsi il est possible d'utiliser un champs sans mentionner le mot clef `this` si cela n'est pas ambiguë.

3.4 Limitation du compilateur

À ce jour nous connaissons deux grosse limitation du compilateur qui freine ses capacité.

- **Plusieurs opération liée enchainé:**
  Si l'on effectue plusieurs opération successive qui ont un rapport entre elle, le code n'est pas

optimisé du tout. En effet :
`x=0`
`x=x+1`
le code compilé donne:

```
1:  LOAD #0 R3
2:  STORE R3 1(GB)
3:  LOAD 1(GB) R3
4:  LOAD #1 R4
5:  ADD R4 R3
6:  STORE R3 1(GB)
```

On remarque alors immédiatement que les lignes 2 et 3 ne servent à rien. Il faudrait cependant repasser après la génération du code pour pouvoir remarquer de tel chose. Ce que nous n'avons pas eu le temps d'implémenter.

- **Gestion du déréférencement:**
  Notre compilateur ne gère pas le déréférencement de classe. Ainsi le code suivant provoque un heap overflow.
  `while(true) a=new A();` Et nous n'avons pas d'idée simple permettant de corriger cette faiblesse de notre code à part la mise à disposition d'une fonction de libération. Ce que nous n'avons pas eu le temps d'implémenter.