Projet Génie Logiciel

Documentation : Manuel Utilisateur

Gr2 - Gl10 - 2023 - 2024

Membres du groupe:

CHRIF M'HAMED Mohamedou
DIAB Dana
KONE Madou
MOHAMED AHMED Mohamed Lemine

Le Sommaire

1. Description du compilateur :

Le compilateur DecaC garantit la compilation d'un fichier ayant comme extension .deca en un fichier assembleur, prêt à être exécuté avec la machine abstraite IMA pour afficher le résultat attendu. Le fichier résultant, d'extension .ass, est généré dans le même répertoire que le fichier source. L'utilisation du compilateur n'est pas limitée à un seul fichier, et plusieurs options peuvent être exploitées lors de la compilation de plusieurs fichiers sources, que ce soit de manière parallèle ou consécutive. Les détails sur ces options seront explicités dans la suite de la documentation. Lors de la compilation, le compilateur traverse trois étapes distinctes.

La première étape, que nous nommerons A, correspond à la phase d'analyse syntaxique. Dans cette étape, un analyseur syntaxique est mis en œuvre pour le langage Deca. Cet analyseur construit l'arbre abstrait primitif du programme, offrant une représentation structurée et hiérarchique de la syntaxe du code source. Cette étape joue un rôle crucial dans la compréhension et la représentation du programme.

Pour ce qui est de la deuxième étape, que nous nommerons B, elle comporte deux aspects principaux. D'une part, cette étape consiste à vérifier la syntaxe contextuelle spécifique à Deca. D'autre part, elle vise à enrichir l'arbre abstrait en y ajoutant des informations contextuelles, également appelées décorations. Ces décorations comprennent les définitions des identificateurs et les types d'expressions. En effectuant ces enrichissements, le compilateur renforce la compréhension du contexte du programme, ce qui est essentiel pour l’étape ultérieure du processus de compilation.

Pour ce qui est de la troisième et dernière étape, que nous nommerons C, elle revêt une importance cruciale dans le processus de compilation. Cette étape est dédiée à la génération du code en assembleur à partir d'un code Deca qui a été validé sur le plan syntaxique et contextuel. L'objectif est de produire un code assembleur qui respecte les règles et les spécifications du langage Deca. Cette phase de génération de code constitue la conclusion du processus de compilation, préparant le programme pour son exécution ultérieure sur la machine abstraite cible.

Des erreurs éventuelles peuvent survenir à différentes étapes de la compilation, notamment à l'étape B. Nous détaillerons ces erreurs dans la suite de la documentation.

2. Commandes et options :

Les options disponibles et fonctionnelles dans notre compilateur Decac sont les suivantes :

- **-b** : Affiche une bannière indiquant les noms des membres de l'équipe.
- **-p** : Construit l'arbre correspondant au fichier source à compiler puis décompile l'arbre de façon à obtenir un programme Deca syntaxiquement correct.

- **-v** : Vérifie contextuellement le fichier source et ne produit aucune sortie en l'absence d'erreur.
- **-n** : Supprime les tests à l'exécution relatifs à la division entière, au reste de la division par 0, ainsi qu'aux débordements arithmétiques. En activant cette option, ces vérifications ne seront pas effectuées lors de l'exécution du programme compilé. Il est important de noter que l'utilisation de cette option peut entraîner un comportement indéfini si les conditions exclues par les tests ne sont pas garanties par le code source.
- **-r X** : Limite le nombre de registres à X, avec X compris entre 4 (inclus) et 16 (inclus). Les valeurs acceptées sont {0, ..., X-1}.
- **-d** : Active le mode de débogage. Le niveau des traces est défini en fonction du nombre de **-d** spécifiés, sachant qu'il existe 4 niveaux.
- **-P** : Permet la compilation en parallèle de plusieurs fichiers sources.

Les options **-p** et **-v** sont incompatibles. Leur utilisation simultanée générera une erreur. La commande Decac sans options et sans fichiers sources permet d'afficher les options disponibles.

L'option **-b** doit être utilisée seule, sans aucune autre option ni fichier source.

3. Messages d'erreurs :

Lors de la compilation d'un fichier source, plusieurs erreurs peuvent survenir à différents moments du processus. Voici quelques exemples de messages d'erreurs non exhaustifs liés aux différentes étapes de compilation :

A. Les erreurs lexicales et syntaxiques potentielles à la première étape 'A' de la compilation indiquent le chemin du fichier source, le numéro de la ligne et la colonne de l'erreur et qui sont les suivantes :

A.1 <u>Les erreurs lexicales</u> : Les erreurs lexicales consistent à l'utilisation d'un mot indéfini dans le programme lexer ou une mal utilisation d'une définition d'un token :

1. Utilisation d'un mot inconnu par lexer : Erreur:token recognition at 'l'endroit de l'erreur'.
    - Chaîne sans " à la fin : exemple { "hello world ! }
    - Utilisation de " non protégé par \ : { " bonjour " tout le monde"}
    - Utilisation de \ non protégé par \ : { " bonjour \ tout le monde"}

A.2 <u>Les erreurs syntaxiques</u> : qui consistent à l'utilisation erronée d'une phrase ou d'une ligne de commande ou d'un chemin inconnu par notre langage.

1. Erreur missing character :
    - Par exemple sans '}' : { "hello" ;
    - Oublier de mettre un ';' : { int x }

2. Une fonction qui prend en paramètre une condition non nulle, mais à laquelle on ne fournit rien ou un token non autorisé et elle attend la liste qu'on peut voir sur la sortie: on affiche mismatched input 'le token avant' et on liste la liste des tokens possibles .

    - Par exemple : if( ){..}
    - Ou pour while : while () ;
    - Void get() asm("hello";);

3. Mal utilisation d'un nœud : mismatched input 'nom du nœud' expecting {une liste des choix possibles}.

    - Else sans if qui précède.
    - Else qui prend une condition : else(..)
    - Void get() asm (3);

4. Un chemin inconnu par le langage : non viable alternative at input 'le token où il y a l'erreur '.
    - Déclaration d'une variable après un instruction : int x,y; x =3 ; int z;
    - Utilisation de deux mains dans le même programme : {...} {...}
    - Utilisation de main avant la définition d'une classe : {..} classe A {..}
    - Définition d'une classe dans main : {.. classe A{} }
    - Utilisation de main dans une classe : class A {..{..}..}
5. Mal utilisation d'une règle :
    - Ne pas respecter la condition d'affectation qui oblige la valeur de gauche à être un lvalue : {.. 2 = 3+4; }. L'erreur : left-hand side of assignment is not an lvalue.
    - Ne pas mettre l'un des opérandes ou l'utilisation d'un non autorisé d'une opération : x = y op ; ou x = y*+;
    - Mal définir une classe : A {..} sans le mot class
    - Mal définir un champ : x; sans type
    - Mal définir une méthode : getF(){..} sans type de retour.

B. La liste complètes des erreurs contextuelles se trouve sur la branche master du gitlab dans gl10/docs/etapeB/docs/Erreurs.pdf

Une liste non exhaustive des erreurs contextuelles potentielles à la seconde étape 'B' de la compilation sont les suivantes :

1. Message : *Identificateur non déclaré*

Message d'erreur pour l'utilisation d'une variable non définie. Par exemple :

```
{
        int b = 0 ;
        b = b + a ;
}
```

2. Message : *Identificateur de type non déclaré*

Message d'erreur pour l'utilisation d'un type non défini. Par exemple :

```
{
        intt a = 2 ;
}
```

3. Message : *Le type doit être différent de void*

Message d'erreur pour la déclaration d'un variable de type void . Par exemple :

```
{
        void x ;
}
```

4. Message : *Variable déjà déclarée*

Message d'erreur pour la duplication de la déclaration d'une variable. Par exemple :

```
{
        int x = 2 ;
        int x = 1 ;
}
```

5. Message : *Type1 ne peut pas être affecté à Type2*

Message d'erreur lors d'une affectation incompatible. Par exemple :

```
{
        boolean bool = true ;
        int x = bool ;
}
```

6. Message : *L'opération 'op' n'est pas possible entre 'type1' et 'type2'*
   Avec op : + , - , * , / , % , && , || , ! , > , >= , < , <= , == , !=

Message d'erreur lors d'une opération non définie. Par exemple :

```
{
        int x ;
        x = 1 + true ;
}
```

7. Message : *le type attendu int , float , string*

Message d'erreur si on passe un type autre que (int, float, string) à print(ln)(x). Par exemple :

```
{
  print(true);
}
```

8. Message : *Le type attendu est boolean*

Message d'erreur si on passe un type non boolean comme une condition à if ou while. Par exemple :

```
{
  if(1){ }
```

```
}
```

9. Message : *l'opération 'op' devant type n'est pas valide*, avec op : UnaryMinus (-) , Not(!)

Message d'erreur si on applique une opération unaire non valide. Par exemple :

```
{
  ! 1 ;
}
```

10. Message : *la classe A est déjà déclarée*

En cas de duplication de la déclaration d'une classe. Par exemple :

```
class A {          }

class A {          }
```

11. Message : *classe C n'est pas définie*

Message d'erreur pour l'utilisation d'une classe non définie. Par exemple :

```
class D extends C {          }

class C {}
```

12. Message : *une redéfinition inacceptable du x en tant que méthode*

Message d'erreur si on utilise le même symbole pour un champ du super classe pour définir une méthode dans le sous classe. Par exemple :

```
class A {
          int x ;
}
class B extends A {
           int x() {}
}
```

13. Message : *une redéfinition inacceptable du get en tant que champ*

Message d'erreur si on utilise le même symbole pour une méthode de super classe pour définir un champ dans le sous classe. Par exemple :

```
class A {
  int get(){        }
}

class B extends A {
  int get ;
}
```

14. Message : *une méthode redéfinie doit avoir la même signature que la méthode héritée*

Message d'erreur si on redéfinit une méthode de la super classe en changeant la signature. Par exemple :

```
class A {
   int add(int a , int b){      }
}

class B extends A {
   int add(int a) {   }
}
```

15. Message : *type1 n'est pas un sous type du type2*

Message d'erreur si on redéfinit une méthode de la super classe en changeant le type de retour vers un type qui n'est pas un sous type de la type de retour du super classe. Par exemple :

```
class A {
   int get() {    }
}

class B extends A {
   boolean get() {     }
}
```

16. Message : *nature attendu : champ , paramètre , variable*

Message d'erreur si on affecte à un symbole qui n'est de nature : champ, paramètre, variable. Par exemple :

```
class A {
   int x;
   int get() {
      get = 1 ;
   }
}

class B {
   A a ;
   void inti() {
      a.get = 0 ;
   }
}
```

17. Message : *casting entre 'B' et 'A' n'est pas possible*

Message d'erreur pour une opération de casting non valide. Par exemple :

```
class A { }

class B { }
```

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

18. Message : *l'opération instanceOf n'est pas possible entre 'type1' et 'type2'*

Message pour une utilisation non valide du instanceof. Par exemple :

```
{
   true instanceof Object;
}
```

19. Message : *'this' ne peut pas être appelé en dehors d'une classe*

Message d'erreur pour l'utilisation du 'this' dans main. Par exemple :

```
class A {
   int x ;
}

{
   int a = this.x;
}
```

20. Message : *x est protégé*

Message d'erreur si on accède à un champ protège. Par exemple :

```
class A {
   protected int x;
}

class X {
   void m() {
      A a = new A();
      println(a.x) ;
   }
}
```

C. Les erreurs potentielles à la dernière étape 'C' de la compilation sont les suivantes :

- Division par zéro : par exemple, dans les expressions telles que x/0 ou x/(4-4).
- Reste de la division par zéro : par exemple, dans les expressions x%0 ou x%(4-4).
- Débordement arithmétique : cela se produit lorsque le résultat d'un calcul dépasse la capacité de représentation.
- Excès de variables temporaires : cette erreur survient lorsque l'évaluation d'une expression implique un nombre excessif de variables temporaires.

- Débordement de la pile : cette situation se produit lorsque la pile mémoire atteint sa capacité maximale.
- Débordement du tas : cette situation se produit lorsque le tas atteint sa capacité maximale.
- Déréférencement null : cette situation se produit lorsqu'un programme tente d'accéder ou de manipuler la valeur d'un pointeur qui est actuellement défini comme nul

4. Limitations :

Partie A: À ce stade, nous considérons que le code est complet, et nous n'avons pas identifié de bugs. Cependant, il pourrait être bénéfique d'envisager la factorisation du code dans le parseur, en utilisant ou en appelant des fonctions utilitaires, telles que getText(), depuis le lexer. Cette approche pourrait améliorer la lisibilité et la maintenabilité du code, en réduisant la redondance et en favorisant une structure plus modulaire.

Partie B : Étant cruciale pour la dernière étape C, nous avons consacré une attention particulière à l'architecture de cette section, en veillant à établir une base solide. Nous avons mis en place une structure robuste tout en effectuant une batterie exhaustive de tests. Cette approche vise à garantir la fiabilité et la stabilité de la partie B, assurant ainsi une transition en douceur vers la phase finale de la compilation, l'étape C.

Partie C: La partie C de notre compilateur expose les faiblesses les plus significatives de notre implémentation. Ces vulnérabilités sont cruciales à identifier et à documenter pour une compréhension claire des limites actuelles de notre compilateur.

- *TSTO :* Cette instruction n'a pas été correctement gérée pour les blocs. En effet, on avait la possibilité de le mettre au tout début du programme, mais pour les blocs, on y arrivait pas. Par exemple, si on voulait le mettre au début d'un bloc de code d'une méthode, on n'avait accès qu'au début et la fin de la liste chaînée des instructions.
- *ADDSP :* Il s'agit du même cas que le TSTO
- *Cast :* Tout type de cast n n'a pas été implémenté sauf la conversion automatique entre float et int.
- *Instanceof :* Cette instruction bug car son implémentation n'est pas finie. Il restait à faire une boucle pour parcourir les bases des tables de méthodes. Le nombre de tours de boucle nécessitait l'ajout de nouvelles variables dans la génération de la table des méthodes.
- *MethodAsmBody :* Nous n'avons pas géré le cas où les fichiers sources contiennent des méthodes dont le corps est écrit en assembleur.
- Le corps de la méthode equals de Object a été codé et peut être appelé comme attendu, cependant, la méthode renvoie toujours 1 (True). C'est dû au fait

qu’on ne savait pas s’il fallait déréférencer les adresses de this et other avant de comparer.

- La génération de code pour une instruction faisant appelle à null (comme une condition par exemple) n’a pas été implémentée.
- Les champs de type float et Class, ne sont pas correctement initialisés.
  Cela est dû au fait qu’un champ non initialisé rentre dans la classe NoInitialization et dans cette classe on a aucun moyen de rentrer dans les classes, float et ClassType
- Les selections marchent sauf pour le cas où on fait (this.champ).
- L’option -n ne supprime pas certains tests à l'exécution. Comme le fait qu’une méthode peut se terminer sans passer par return.