

Documentation de conception

Version 1.0 (release)

Sommaire

Sommaire	2
Introduction	3
A qui s'adresse ce document	3
Motivation	3
Organisation générale	4
Interprète de machine abstraite	4
Planning	5
Dossier source	5
Architecture du logiciel	5
Spécification du code	7
Compilation parallèle	7
Connaissance de DecacCompiler	7
Factorisation du code	7
Développer le compilateur	8
Implémentation de la fonctionnalité	8
Tester la fonctionnalité	8

Introduction

A qui s'adresse ce document

Ce document s'adresse à une personne souhaitant maintenir ou faire évoluer le compilateur Decac. Il décrit l'organisation générale de l'implémentation du logiciel et la justifie.

Il n'est pas ici question de rentrer en détail dans l'explication du code, ni de donner une liste des méthodes pour chaque classe (veuillez consulter la documentation utilisateur et la documentation de l'extension de la bibliothèque standard).

Motivation

L'architecture logicielle est une grande partie de la conception. De la même manière que dans le domaine du génie civil, l'architecture logicielle permet au projet d'être robuste dans le temps mais aussi plus facilement manipulable. Elle permet au logiciel de répondre aux besoins et de résister aux modifications qu'il est susceptible de subir, quel que soit le nombre de personnes participant au projet (on dit *scalable*). Une mauvaise architecture est susceptible de se faire ressentir en temps de développement. Finalement, on peut retenir qu'une bonne architecture va de paire avec un logiciel de qualité.

Organisation générale

L'architecture logicielle représente le logiciel avec beaucoup de recul, sans rentrer dans les détails : c'est une vision de haut niveau. N'importe qui doit être capable de comprendre ce que fait le logiciel, de quoi est-il composé et comment interagissent ses sous-parties. C'est l'objectif de cette partie.


La racine du projet est constituée de :


projet_gl/

├─  docker

├─  docs

├─  global

├─  planning


├─  pom.xml

├─  README.md

├─  src

└─  target

Interprète de machine abstraite

Le dossier  global contient tout ce qui est nécessaire à l'exécution de l'interpréteur de la machine abstraite ima. Il contient notamment un script nommé ima, qui lance l'interpréteur. L'interpréteur, nommé ima-x86_64-Linux, est compilé pour une machine fonctionnant sous Linux avec un processeur de la famille x86.

On peut lancer l'exécution de l'interpréteur sur un fichier codé en un langage qui ressemble à de l'assembleur. Si vous souhaitez vous familiariser avec ce langage pour écrire un fichier assembleur et tester l'interpréteur, vous devrez vous référer à la partie génération de code (voir Cahier des charges, page 209).


Le fichier que vous écrivez doit respecter le style de nommage de fichier de votre système et son extension doit être .ass. Vous pouvez par exemple créer un fichier test.ass dont le contenu est :

```
WSTR "Bonjour monde !"
```




```
HALT
```

puis l'exécuter avec ``ima test.ass``. Cela affichera la chaîne Bonjour monde ! dans votre terminal. Pour plus d'informations, référez vous à la page 227 du Cahier des charges.


Planning


Le dossier  `planning` contient des fichiers d'exemples d'un planning utilisé en gestion de projet pour la création d'un compilateur et qui peuvent être utilisés dans un futur projet réaliser en mode agile.


Dossier source


Le dossier  `src` contient les fichiers sources du projet. Ce dossier est divisé en deux parties : un dossier  `main` qui contient les fichiers qui font fonctionner le compilateur et un dossier  `test` qui contient les fichiers qui sont nécessaire à la vérification du bon fonctionnement du compilateur.

Architecture du logiciel

La liste des classes et leurs dépendances. On trouvera plusieurs sous dossier utiles dans  `src/main` :

 `antlr/fr/ensimag/deca/syntax/` pour tout ce qui est lié au lexer et parser.

 `java/fr/ensimag/deca` avec la classe représentant le compilateur et ses classes dépendantes.

 `java/fr/ensimag/deca/tree/` pour tout ce qui est lié à la construction et à la manipulation des arbres générés par le parser. En particulier on trouvera les fonctions `verify` et `codeGen` qui réalise le parcours de l'arbre pour les décorations lors de la vérification contextuelle et pour la génération du code assembleur.

📁 `java/fr/ensimag/deca/context/` pour tout ce qui concerne les définitions des différents types les types eux-même, ainsi que les définitions des classes pour deca, des variables, puis des tableaux implémenté dans notre extension.

📁 `java/fr/ensimag/ima/pseudocode/` pour tout ce qui concerne la manipulation du code assembleur, avec en particulier le sous-dossier 📁 `instructions` contenant les classes représentant les instructions machines

Pour la structure des classes, on trouve une classe mère `Tree` pour l'ensemble des noeuds de l'arbres, puis ensuite on trouve deux catégories de classes : les classe abstraite `AbsctractNom` qui représente les non-terminaux de la grammaire, et les autres classes qui représente les terminaux de la grammaire et donc les noeuds concret de l'arbre généré.

Ensuite on trouve deux classes mère pour les types et définition associé : `Type` et `TypeDefinition` qui représente respectivement un type abstrait dont découle tout les types natif de deca et la définition associée dont découle les définitions associées à chacun des types. Ces objets servent à décorer l'arbre avec par exemple pour tous les nœuds dérivant de `AbstractExpr` représentant les expressions, leur type associées permettant la vérification du typage de ces expressions, ou encore le lieux de définition des variables et des classes.

Pour la partie génération de code, on retrouve la liste des instructions machine dérivant toutes de la classe `Instruction`. Ainsi que les classes dérivant de `Register` pour la manipulation des registres.

Par ailleurs, on notera que pour partager des variables semblables à des variables globale, on les déclare dans la classe `DecacCompiler` comme par exemple l'environnement des types.

Pour une meilleure compréhension de l'architecture du logiciel, vous pouvez consulter le diagramme UML qui fournit des détails supplémentaires sur les différentes parties du logiciel et sur les interactions entre elles.

Spécification du code

Compilation parallèle

Concernant les parties du code qui sont spécifiques à notre implémentation, il y a en première partie le choix de l'implémentation du parallélisme. Pour cela nous avons créé un objet de type `Executors` dans lequel nous avons utilisé la méthode `newFixedThreadPool` qui nous permet de créer autant de thread que de processeur possible puis nous créons une `ArrayList` de type `Future<Boolean>` de taille le nombre de fichier à compiler puis dans une boucle. Enfin pour chaque fichier on crée un objet `Future<Boolean>` que l'on soumet à notre `Executors` avec la fonction de compilation puis nous ajoutons cet objet à notre `ArrayList`. On crée un itérateur pour la liste de future et on appelle `get` sur chaque fur afin de vérifier que la compilation c'est fini.


Connaissance de DecacCompiler

Il est généralement préférable de minimiser l'utilisation de variables globales. Les variables globales peuvent également causer des problèmes de concurrence dans les applications multithread. Nous avons choisi de placer certaines variables dans la classe `DecacCompiler` puisque celle-ci est "omnisciente". Par exemple, une instance du compilateur a connaissance de l'environnement des types (`EnvironmentType`) et du programme assembleur (`IMAProgram`).

Factorisation du code

On a utilisé des classes abstraites intermédiaires pour factoriser le code le plus possible, par exemple pour la décoration des opérations binaires, nous avons une classe spécifique aux opérations arithmétiques et une pour les opérations booléennes. En particulier, le parcours de l'arbre se fait de sorte à éviter les tests sur les sous-nœuds dans les classes mères. Cependant, il reste une partie des fonctions qui ne sont pas factorisables. De plus, l'utilisation de types abstraits permet une meilleure flexibilité dans le contenu possible des sous-nœuds et ainsi une meilleure lisibilité pour la relecture.


Développer le compilateur






Si l'on souhaite rajouter des fonctionnalités en plus dans le langage, il faudra d'abord prendre en compte la dépendance entre chaque composants grâce au diagramme UML (nommé ULMtree.png) situé dans  src/doc. Ce document précise aussi la liste des composants déjà présents dans le projet. Pour compiler et exécuter ce projet il faut d'abord avoir une machine ima (cf Documentation utilisateur) puis exécuter la commande ``mvn compile`` , puis ``mvn test-compile`` si l'on souhaite tester la code ce qui va lancer la totalité de nos test (85% deca 15% JUnit). Pour plus d'informations sur l'utilisation de Maven, veuillez vous référer à la page 119 du Cahier des charges.

Implémentation de la fonctionnalité

Pour ajouter une nouvelle fonctionnalité au langage, il faut d'abord réfléchir à l'implémentation de l'extension sur la grammaire déjà présentée (cf Document Extension) puis ajouter au `lexer.g4` les nouveaux caractères (chaîne de caractère) que l'on souhaite détecter. Enfin il faut modifier le parser pour coller à la nouvelle grammaire et si on ajoute de nouveaux terminaux il faut créer une nouvelle classe `.java` dans le package `Tree` avec les différents champs que l'on souhaite utiliser puis il faut implanter les différentes fonctions qui découlent de `Tree` (`decompile`, `prettyprint...`) puis réaliser l'étape B afin de décorer la partie de l'arbre qui va changer avec l'extension et ajouter les nouvelles instructions sur le code machine en modifiant la fonction `codeGen`.

Tester la fonctionnalité

Une autre étape importante est le test des différentes parties ajoutées pour cela il faut créer un dossier avec le nom de l'extension dans  `src/test/deca` puis compléter l'architecture :

```
 src/test/deca/  
├──  codegen/  
├──  context/  
├──  extension/  
└──  syntax/
```


puis il faut créer des fichiers `.deca` avec un nom explicite et le placer dans un sous-répertoire : 📁 `valid` ou 📁 `invalid` suivant si le résultat doit être une erreur ou pas.

Enfin il faut créer un script comme les `complete-[NOM-DU-TEST]-extension.sh` puis l'ajouter dans le `pom.xml` comme ici avec le 🛠️ `complete-parser.sh` :

```
<execution>
  <id>complete-parser</id>
  <configuration>
    <executable>./src/test/script/complete-parser.sh</executable>
    <configuration>
      <phase>test</phase>
      <goals>
        <goal>exec</goal>
      </goals>
    </configuration>
  </execution>
```

Il faut après lancer ``mvn test`` afin de vérifier que les nouveaux tests couvrent bien toutes les nouvelles fonctions.

On s'est servi des scripts 🛠️ `basix-lex.sh` et 🛠️ `basic-synt.sh` pour écrire de meilleurs scripts. Un mainteneur du compilateur peut, lui aussi, s'inspirer de nos scripts.