Projet de compilation
Travail à remettre le xxx mai 2021 (à préciser)

Lionel Clément

version `2021-04-16/10:04:00`

Ce que vous avez à faire est juste une modification de certains fichiers `StreeXXX.java` dans le *package* `fr.ubordeaux.deptinfo.compilation.lea.stree`

Modalité pour rendre le travail

Déposer dans le Moodle du cours, dans *Rendu projet* :
`https://moodle1.u-bordeaux.fr/mod/assign/view.php?id=xxxxxx`
— Un fichier d'archive qui contient tout le code (mais aucune bibliothèque ni aucun fichier compilé)
— Un fichier PDF très court qui contiendra quelques notes à destination du correcteur pour mieux comprendre et évaluer votre dépôt et qui explique le rôle des 4 personnes dans le groupe de projet.

1 Introduction

Le langage dont il est question dans ce projet est toujours Léa, qui a été utilisé pour le mini-projet et très légèrement remanié (on a par exemple retinré la redéfinition des opérateurs).

Le projet consiste à réaliser un compilateur qui teste les types et produit du code intermédiaire tel que décrit dans le chapitre 7 de « Modern Compiler Implementation in Java » donné en référence bibliographique du cours.

Vous pourrez le tester en lançant la commande `ant` qui produira les fichiers
— `data/progr-xxx.output`
— `data/progr-xxx.error`
— `data/XXX-YYY.dot`
à partir des fichiers `data/progr-xxx.lea`.
— Les fichiers `data/progr-xxx.output` contiennent les chaines produites par la méthode `toString()` des objets du code intermédiaire.
— Les fichiers `data/progr-xxx.error` sont les messages d'erreur que le compilateur Léa produit (essentiellement parce qu'il manque du code pour l'instant).
— Les fichiers `XXX_YYY.dot` où `XXX` est une position et `YYY` le nom d'une méthode, sont des fichiers représentant le conde intermédiaire sous forme de graphe. Pour le visualiser, il suffit de produire le fichier `JPG` grâce à la commande `dot -Tjpg -o XXX_YYY.jpg XXX_YYY.dot`
Pour faire ce projet, nous avons réalisé les étapes suivantes :

1. Analyse lexicale
2. Analyse syntaxique
3. Production de la syntaxe abstraite
4. Production du code intermédiaire
5. Production de l'affichage de ce code intermédiaire sous forme de graphe

Mais malheureusement a vérification du typage n'a pas été réalisé, pas plus que les principales productions du code intermédiaire.
Nous avons simplement implémenté la boucle `WHILE` pour montrer l'exemple.

2 Travail à réaliser en groupes de 4 pour le xxx mai 2021 (à préciser en fonction de la date de jury)

1. Découvrir le noyau du projet fourni et le comprendre avec les éléments donnés ici
2. Choisir une structure de contrôle non implémentée
   — boucle avec `for`, `do` ou `foreach`
   — test avec `if` ou `switch`
   — appel de fonction ou de procédure avec passage des paramètres
   — accès à un champ d'un objet (attribut ou méthode)
3. Réaliser l'analyse du typage et produire les messages d'erreur
4. Réaliser l'implémentation de la production du code intermédiaire

3 Éléments fournis

1. Fichier pour la compilation `build.xml`
   Il suffit d'utiliser la commande `ant` pour compiler le tout et pour produire le fichier `data/xxx.output` `data/xxx.error` `data/xxx.dot`
   Ce fichier doit être modifié pour ajouter d'autres exemples.
2. Fichiers exemple `data/input-1.lea`, ... `data/input-3.lea`.
   Éditez ces fichiers et ajoutez-en en fonction de vos ajouts.
3. Classe principale

   `fr.ubordeaux.deptinfo.compilation.lea.Main.java`

   Construit l'analyseur lexical avec le nom du fichier donné en argument de la commande, construit l'analyseur syntaxique, le lance et affiche dans `data/xxx.error` le message correspondant à l'une des exceptions suivantes si elles ont été produites :
   — `IOException` Erreur de lecture du fichier
   — `EnvironmentException` Erreur de manipulation des environnements
   — `TypeException` Erreur de typage
   — `StreeException` Erreur lors de la construction de la syntaxe abstraite
4. Analyseur lexical `Jflex`
   `fr.ubordeaux.deptinfo.compilation.lea.lexer.Lexer.jflex`
5. Grammaire `Bison`
   `fr.ubordeaux.deptinfo.compilation.lea.parser.Parser.y`

   La grammaire est entièrement écrite. Mais il faudra la corriger ou la compléter pour ajouter le typage dans la construction de la syntaxe abstraite.
   La grammaire sert à construire l'arbre de syntaxe (un objet de type `StreeXXX`).
6. Environnements
   Package `fr.ubordeaux.deptinfo.compilation.lea.environment.*`

   Ces classes permettent de conserver l'enregistrement des variables, des types et des constantes.

   `Environment.java` Interface
   `EnvironmentException.java` Exception
   `MapEnvironment.java` Implémentation de `Environment`
   `StackEnvironment.java` Impémentation d'une pile d'environnements

7. Types
   Package `fr.ubordeaux.deptinfo.compilation.lea.type.*`

`Type.java` Interface
`TypeException.java` Exception
`TypeExpression.java` Implémentation de `Type` sous forme d'une arbre binaire
`Tag.java` Étiquettes des types

8. Syntaxe abstraite
   Package `fr.ubordeaux.deptinfo.compilation.lea.stree.*`

   Un grand nombre de classes `StreeXXX.java` étendent la classe abstraite `Stree.java`. Il ne faut pas s'inquiéter du nombre de classes ; elles se ressemblent fortement.

   Un objet de type `Stree` est responsable
   — De la vérification du typage. La méthode `boolean checkType()` doit être redéfinie pour les objets dont on vérifie le typage.
   — De la production de code intermédiaire. La méthode `Stm generateIntermediateCode()` doit être redéfinie pour les objets qui produisent du code intermédiaire.

   Les méthodes virtuelles de cette classe abstraite, c'est-à-dire les méthodes qui n'y sont pas définies mais seulement déclarées doivent être définies dans les classes filles. Pour facilité la visibilité de ces méthodes, nous les avons remplacé les méthodes virtuelles par des méthodes qui provoquent une exception par défaut pour avertir qu'elles n'ont pas été encore implémentée. L'affichage donne ceci :

   ```
   Not yet implemented: checkType() in StreeEQ
   ```

   Ceci signifie que la méthode `checkType()` manque dans `StreeEQ` et qu'il faut donc la définir.

   Ces Méthodes qu'il faut partiellement définir (quand c'est pertinent) sont les suivantes :

   `public Stm getStm()` Donne l'instruction
   `public Exp getExp()` Donne l'expression
   `public Type getType()` Donne le type
   `public ExpList getExpList()` Donne la liste des expressions
   `public Stm generateIntermediateCode()` Produit le code intermédiaire
   `public boolean checkType()` Teste le typage

   Pour vous aider, nous avons implémenté
   — L'affectation (`StreeAFF`)
   — La boucle *while* (`StreeWHILE`)
   — Les variables (`StreeVARIABLE`)
   — Les constantes entières (`StreeINTEGER`)
   — L'addition (`StreePLUS`)

9. Code intermédiaire
   Package `fr.ubordeaux.deptinfo.compilation.lea.intermediate.*`

   Ces classes sont reprises du chapitre 7 du livre *Modern Compiler Implementation in Java* et permettent de construire un code intermédiaire sur la base de deux classes abstraites :
   — `Exp` Les expressions
   — `Stm` Les instructions