Document de conception

I Enrichissement du langage et de la grammaire

Etape A : Analyse lexicale et syntaxique

En premier lieu, il est possible d'élargir le langage, en ajoutant des symboles, des mots ou encore des règles de grammaire.
Pour ajouter un *token* au langage, il faut l'inscrire dans le fichier:

`src/main/antlr4/fr/ensimag/deca/syntax/DecaLexer.g4`

Pour ajouter une règle de grammaire, il faut l'inscrire dans le fichier:

`src/main/antlr4/fr/ensimag/deca/syntax/DecaParser.g4`

(fichier en Antlr4) .

Pour ajouter une exception lançable par le parser, il est recommandé de la placer dans fr.ensimag.deca.syntax et de la lui attribuer `DecaRecognitionException` en mère.

L'ajout de mot et de règle au langage se fait simplement en suivant les directives du cahier des charges. Attention cependant à bien compléter le code java du *parser* (rédigé entre des accolades {}): il faut, si nécessaire, définir l'arbre de retour d'une règle et lui attribuer une localisation dans l'arbre général via la méthode `setLocation`, sans oublier de vérifier les conditions quelconques (du type vérifier que les arbres des règles impliquées par la nouvelle règle ne soient pas nuls).
Ces modifications peuvent entraîner la création de nouvelles classes dans le répertoire `src/main/java/fr/ensimag/deca/tree`.

Etape B: Analyse contextuelle

En second lieu, on peut donc élargir la partie contextuelle.
Les fichiers sources concernés sont placés dans le répertoire `src/main/java/fr/ensimag/deca` et ses dépendances:

- Le package par défaut contient le code concernant le compilateur en lui-même

- Le package `context` contient le code relatif au traitement contextuel: types, définitions, environnements d'expressions et de types, erreur ContextualError spécifique à cette partie
- Le package `tree` contient la description des arbres créés par le *parser*; en particulier la vérification et décoration (méthodes de préfixe "`verify`"), mais aussi l'itération sur les branches (préfixe "`iter`"), la décompilation ("`decompile`") et l'affichage (préfixe "`prettyPrint`").
  Les méthodes `codegen` relatives à l'étape C y sont également implémentées

Les règles ajoutées à l'étape A impliquent souvent la création d'un arbre sur le modèle du package `tree`.
Les types natifs du langage deca sont définis dans l'attribut `envTypesPredef` de la classe `EnvironmentTypes`. Comme en java, on notera que les `String` ne sont pas des types natifs. Ils pourraient donc faire l'objet d'une nouvelle classe deca.

Etape C: Génération de code

La génération de code s'appuie sur les méthodes `codegen` du package `tree`, le package `codegen`, des champs propres à l'objet DecacCompiler ainsi que le package `pseudocode`. Voici quelques consignes pour une maintenance efficace du code. On notera que la plupart des méthodes utilisées prennent en argument un object DecacCompiler, qu'on nommera `compiler` dans le code.

Utilisation des registres

On appellera par la suite registre courant le registre non scratch de plus petit indice non utilisé, de numéro `compiler.getCurrentRegister()` . Le calcul d'une expression peut nécessiter l'utilisation de plusieurs registres. Lorsque la valeur dans un de ces registres ne doit pas être écrasée, il faut appeler la méthode `incrCurrentRegister` du compilateur, et `decrCurrentRegister` lorsqu'on en a plus besoin. Avant d'appeler ces fonctions il faut toutefois s'assurer que le registre courant n'est pas le dernier registre scratch disponible, de numéro `compiler.getCompilerOptions().getRMAX()`. Une assertion dans la fonction `incrCurrentRegister` serait un bon ajout.

Test de débordement de pile

Une alternative à l'utilisation des registres est la sauvegarde en pile, qui peut également survenir dans d'autres contextes. Afin de calculer la place nécessaire

dans la pile, il faut appeler la méthode `incrNbTemp` du compilateur (ou `setNbTemp` si cela est plus pratique), qui va potentiellement augmenter le nombre maximal de temporaires utilisées dans le programme si celui-ci est dépassé. Comme précédemment il faut décrémenter ce nombre quand on restaure les valeurs ou quand on décrémente le sommet de la pile, avec `decrNbTemp` (ou `setNbTemp`).
Comme ces valeurs sont indépendantes entre blocs, elles sont réinitialisées avant la génération de chacun d'eux par la méthode `reinitCounts` du compilateur.

Génération des expressions

La génération de code pour les opérations arithmétiques et de comparaison s'effectue sur le modèle présenté en diapositive 5 de la présentation de la génération de code. C'est la fonction `codeGenInst` qui réalise cette opération. Elle se base sur deux fonctions intermédiaires :

- la fonction dval héritée de la classe `AbstractExpr`, qui doit renvoyer la valeur immédiate dans le cas d'un littéral, ou l'adresse de la valeur pour toute autre expression. Par défaut, celle-ci est le registre courant. Pour une variable ce sera son adresse dans la pile. Il faut noter que pour calculer l'adresse d'un champ de classe il faut générer du code pour calculer l'adresse de l'objet auquel il appartient. En cas d'utilisation d'une sélection il faut donc prendre en compte cette génération de code lors de l'appel de la fonction `dval`.
- la fonction `mnemo` définie dans `codegen/EvalExpr.java` qui génère le code spécifique à chaque instruction. Elle pourra être modifiée comme elle l'a été pour optimiser les opérations de multiplication, division et reste entier par une puissance de 2 en remplaçant les opérations MUL, QUO et REM par des décalages de bits.

Optimisation pour les opérations entières par une puissance de 2

Cette optimisation s'applique lorsque le membre de droite de l'opération est un immédiat entier dont la valeur est une puissance de 2.
Pour les opérations de multiplication et de division l'optimisation est simple : on calcule la puissance de 2 à laquelle correspond le membre droit de l'opération, et on génère autant d'opérations SHL, respectivement SHR, que cette valeur.
Pour le reste entier on copie la valeur du membre de gauche dans le registre R1, on effectue un nombre de décalages à droite égale à la puissance de 2 correspondante, puis autant de décalages à gauche, puis on soustrait la valeur obtenue à la valeur initiale, toujours dans le registre courant. On a ainsi réalisé l'opération

$y = x - x/2^n * 2^n$, avec x l'opérande gauche et "/" la division entière, ce qui est bien l'équation du reste entier.

Booleens

Comme demandé dans le cahier des charges, les expressions booléennes sont codées par des flots de contrôle à travers la fonction `boolCodeGen`. Un appel direct à une opération booléenne via la fonction `codeGenInst` n'a pas de sens avec cette implémentation (cela ne génère pas de code hormis pour les littéraux, puisque la valeur de l'expression n'est pas utilisée). Pour l'assignation d'une valeur booléenne on utilise la fonction `boolInRegister` de `codegen/EvalExpr.java` qui met dans le registre courant la valeur 0 si l'expression en argument est fausse, et 1 sinon. C'est de cette manière que sont traitées les variables booléennes.

Erreurs

Pour ajouter des tests d'erreurs à l'exécution il suffit d'écrire le branchement vers le traitant de cette erreur et d'appeler la méthode `addError` du compilateur avec en argument le label de ce branchement. Cela a pour effet d'ajouter l'erreur à un dictionnaire qui sera utilisé en fin de génération de code pour la création de tous les traitants d'erreurs signalés.

Numérotation des labels

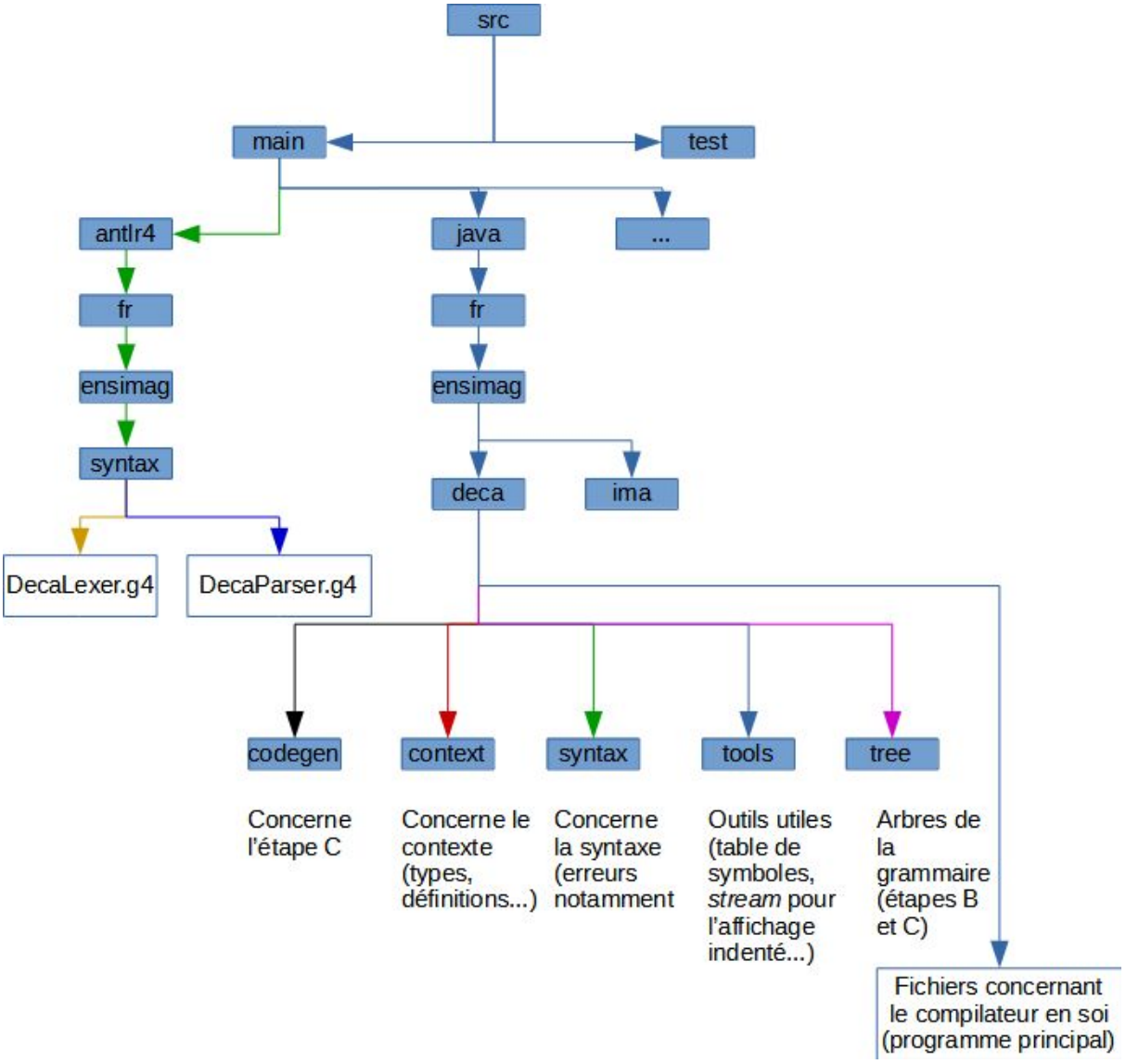
Afin de s'assurer de l'unicité de tous les labels générés, on doit utiliser la méthode `incrNbLabel` du compilateur lorsque l'on en crée un, et ajouter le nombre obtenu par `getNbLabel` à la fin de tout label.

Classes et méthodes

Le code de génération du code relatif au langage objet (tables des méthodes, initialisations des objets) est contenu dans `codegen/ClassCodeGen.java`. Le corps des méthodes est réalisé par les méthodes `codeGenBody` des classes `DeclMethod` et `MethodBody` dans le package `tree`. La gestion des appels de méthodes est réalisée dans la classe `MethodCall` de ce même package.

Résumé schématique de l'architecture pour ces 3 étapes

Code couleur:
vert : étape A - jaune : lexer - bleu : parser - rouge : étape B - noir : étape C - violet : étapes B et C



II Tests

Des programmes en langage deca destinés à tester le bon fonctionnement du compilateur sont écrits dans les sous-répertoires de `src/test/deca` (répartis dans les dossiers codegen, context, lexer et syntax selon la partie de la compilation à tester).
Des programmes java sont disponibles dans les sous-répertoires de `src/test/java/fr/ensimag`, utiles pour tester à la main.
Des automatisations de batteries de test sont disponibles dans `src/test/script`. Dans les dossiers de ce répertoire, on peut trouver notamment la colorisation, et surtout les launchers dans le dossier launchers.
Il est donc possible de rajouter des tests à loisir en suivant cette architecture.

III Options du compilateur

La classe `src/main/java/fr/ensimag/deca/CompilerOptions` gère les options du compilateur qui sont sur la ligne de commande. Le premier traitement d'une nouvelle option se fait donc dans cette classe (la récupération des options).
La classe `src/main/java/fr/ensimag/deca/DecacMain` instancie `CompilerOptions` et `DecacCompiler`. C'est donc dans `DecacMain` ou dans `DecacCompiler` que l'implémentation de l'option se fera (selon le niveau auquel elle agit).
Les erreurs sont représentées par les classes `DecacFatalError` et `CLIException`, mais il est possible d'en implémenter d'autres dans le même répertoire.

IV Extensions du compilateur

Le langage deca embryonnaire souffre de peu de potentiel.

Cette version pour y pallier propose une bibliothèque de fonctions trigonométriques. Cette bibliothèque, dont le code source est dans `src/main/ressources/include/Math.decah`, peut aisément être étoffée d'autres fonctions mathématiques essentielles, telles que le logarithme et l'exponentielle.
Une amélioration des algorithmes pour les fonctions déjà implémentées serait également bienvenue.
Bien entendu, les algorithmes de CORDIC et de CORDIC inverse sont déjà plus que satisfaisants, puisqu'ils constituent même la quasi-totalité des implémentations des fonctions trigonométriques dans tous les langages. Ce qui est souhaitable en revanche est une amélioration de l'algorithme de *range reduction*, qui sert à ramener les nombres dans un certain intervalle (dans notre cas $[-;]$). L'algorithme actuel, quelque peu naïf, se sert de divisions euclidiennes. Les flottants manipulés étant en *single-precision*, il nous a été ardu d'utiliser les algorithmes connus - Payne-Hanek notamment - car ceux-ci sont conçus pour des *double-precision*. S'il est toujours possible de construire un flottant de 64 bits en concaténant deux suites de 32 bits, le problème est loin d'être résolu, étant donné que les algorithme requièrent des fonctionnalités qu'il ne nous était pas permis de développer dans les temps (partie entière, racine carrée...) ; de fait les résultats sont encore erronés pour des valeurs démesurées (se rapporter à la documentation sur l'extension pour de plus amples détails). Toute bonne idée a sa place.

Une autre extension estimable concerne la possibilité de manipuler des structures de données - listes, sets, tables de hachage, tableaux, matrices, etc... Pour les besoins de l'extension mathématique, une implémentation de liste chaînée est visible dans la bibliothèque maths `/src/main/resources/include/Math.decah` , mais bien sûr cette implémentation est spécifique à son cas d'utilisation, comme ce le serait en C.

Concernant le reste des extensions imaginables, rien n'est fourni et tout reste à faire.