UNIVERSITE de LORRAINE UNIVERSITE HENRI POINCARE

ESIAL Première année
Projet Structures de données *Année 2011-2012*

Présentation du projet

L'objectif du projet `OGAZ` est de développer un outil capable de générer un analyseur syntaxique descendant d'une grammaire LL(1). C'est l'occasion de mettre en oeuvre les compétences acquises dans les modules Maths Discrètes, Programmation Objet et Techniques et Outils pour la programmation. C'est également un premier pas vers l'apprentissage des techniques de compilation qui seront étudiées en deuxième année dans le module Traduction.

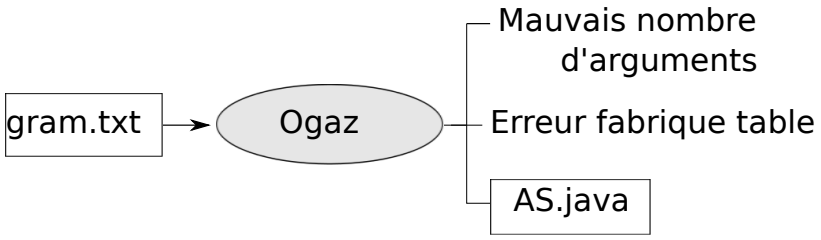
Il existe deux méthodes pour construire un analyseur syntaxique descendant pour une grammaire LL(1) :

1. Par descente récursive : on associe une fonction d'analyse à chaque non-terminal de la grammaire
2. Par automate : l'analyseur s'appuie sur la table de transition d'un automate construit à partir de la grammaire.

Dans la première méthode, l'analyseur est spécifique à la grammaire ; dans le second cas, l'analyseur est le même pour toutes les grammaires ; seule diffère la table de transition d'une grammaire à l'autre. C'est cette seconde technique que nous allons appliquer dans le cadre de ce projet.

Spécification de l'outil `OGAZ`

L'outil OGAZ accepte en entrée la description d'une grammaire et si c'est possible produit une classe Java contenant un analyseur syntaxique descendant de cette grammaire. Cette classe contient une fonction principale d'analyse, capable d'analyser le contenu d'un fichier texte, pour décider si ce texte est ou non conforme à la grammaire.



Grammaires

Pour que l'outil OGAZ puisse reconnaître la description qui lui est fournie en entrée, il s'appuie sur une syntaxe spécifique définie par une grammaire : la grammaire des grammaires (*le commentaire en italique explique la composition d'une grammaire*).

Grammaire	$\longrightarrow$	Règle
	$\longrightarrow$	Règle Grammaire
		Une grammaire contient une ou plusieurs Règles
Règle	$\longrightarrow$	NonTerminal : Alternatives ;
		Une règle commence par un non-terminal suivi d'un double point suivi d'alternatives et d'un point-virgule
Alternatives	$\longrightarrow$	Alternative
		Alternative | Alternatives
		Il peut y avoir une ou plusieurs alternatives dans une règle
Alternative	$\longrightarrow$	
	$\longrightarrow$	SuiteSymboles
		Une alternative est soit le mot vide, soit une suite de terminaux et/ou de non-terminaux
SuiteSymboles	$\longrightarrow$	Symbole
	$\longrightarrow$	Symbole SuiteSymboles
		Une suite de symboles contient au moins un symbole
Symbole	$\longrightarrow$	Terminal
	$\longrightarrow$	NonTerminal
		Un symbole est un Terminal ou un NonTerminal
NonTerminal	$\longrightarrow$	A | B | ... | Z
		Un NonTerminal est une lettre majuscule
Terminal	$\longrightarrow$	a | ... | z | 0 | ... | 9 | + | - | * | / | = | (|)
		Un terminal est soit une lettre minuscule, soit un chiffre, soit un des symbole de l'ensemble $\{+, -, *, /, =, (,)\}$

Dans la suite de caractères décrivant une grammaire, les blancs, fins de Règles et tabulations ne sont pas significatifs. L'axiome de la grammaire est le premier non-terminal cité. L'ordre de description des autres non-terminaux n'a pas d'importance.

Voici un exemple de grammaire acceptée par OGAZ. L'axiome est `A` ; l'ensemble des terminaux est {`a , ( , ) , +`} ; l'ensemble des non-terminaux est { `A , B , C` }. Dans cet exemple, la seconde alternative pour le non-terminal `C` est réduite à une chaîne vide.

```
A  :  a | ( B ) ;
B  :  A C ;
C  :  + A C | ;
```

Organisation du travail

Le travail est à réaliser en binôme.

Le développement de cet outil se fait en plusieurs étapes successives, chacune d'elles donnant lieu au développement d'un produit fini à déposer sur la forge `redmine.esial.uhp-nancy.fr`. Les enseignants se réservent le droit de consulter à tout moment l'état des dépôts pour en véri-

fier la conformité. Toute étape non valide sera à refaire. Vous disposez de onze semaines au maximum (dont deux de vacances) pour mener à bien cette réalisation ; le dépôt final aura lieu à la fin de la semaine 19. Il revient à chaque binôme la tâche difficile de fixer le planning précis du développement, mais il paraît raisonnable de respecter les délais suggérés.

1. Construction manuelle d'un analyseur syntaxique d'une grammaire LL(1)
2. Sur le même principe, construction manuelle de plusieurs analyseurs syntaxiques, en vue de l'automatisation de la construction
3. Automatisation de la construction de l'analyseur
4. Automatisation de la construction de la table de l'analyseur
5. Extension à d'autres types de grammaires

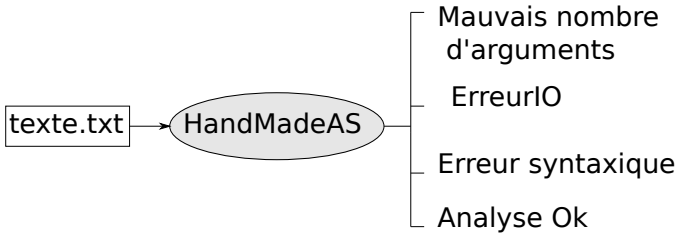
Ce projet ne sera pas évalué uniquement sur la base du dernier dépôt, mais en fonction du respect du planning de développement. Il est donc dommageable de sauter les premières étapes, pour réaliser directement la dernière ou l'avant-dernière.

Etape 1 (Semaine 1 /11)

Cette étape consiste à construire à la main la table de l'analyseur à utiliser pour la grammaire donnée en exemple, puis à programmer et tester l'analyseur syntaxique correspondant. L'analyseur syntaxique doit implémenter l'interface `AnalyseurSyntaxique` définie ci-dessous :

```
interface AnalyseurSyntaxique {
boolean analyse(File f) throws IOException {
}
```

Pour que cette étape soit validée, il faut produire une archive Java qui, lorsqu'elle s'exécute, lit le fichier dont le nom est donné en argument de la commande, l'analyse puis affiche l'un des messages suivants *Mauvais nombre d'arguments* ou *Erreur IO* ou *Analyse syntaxique OK* ou *Erreur syntaxique*.



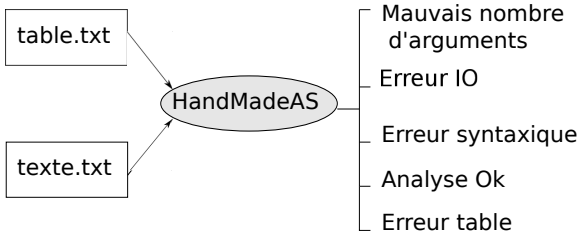

```
% java -jar HandMadeAS.jar
% Mauvais nombre d'arguments
% java -jar HandMadeAS.jar  erty.txt
% Erreur IO
% java -jar HandMadeAS.jar  texte1.txt
% Analyse syntaxique OK
```

% `java -jar HandMadeAS.jar` texte2.txt
% Erreur syntaxique

L'archive doit contenir les fichiers `.java` et `.class`, ainsi que les fichiers de tests `texte1.txt`, `texte2.txt`, *etc.*

Etape 2 (Semaine 1,5/11)

Pour valider cette étape, il faut modifier l'analyseur, de sorte que la table soit lue dans un fichier texte lors de sa création (libre à vous de choisir le format des Règles du fichier). Le nom de ce fichier est le second argument de la Règle de commande. Si le fichier est mal construit, l'exécution de l'archive produit le message *Erreur dans la table.*



% `java -jar HandMadeAS.jar`
% Mauvais nombre d'arguments
% `java -jar HandMadeAS.jar` erty.txt
% Mauvais nombre d'arguments
% `java -jar HandMadeAS.jar` erty.txt table.txt
% Erreur IO
% `java -jar HandMadeAS.jar` erty.txt table.txt
% Erreur table
% `java -jar HandMadeAS.jar` texte1.txt table.txt
% Analyse syntaxique OK
% `java -jar HandMadeAS.jar` texte2.txt table.txt
% Erreur syntaxique

L'archive doit contenir les fichiers `.java` et `.class`, ainsi que les fichiers de tests `texte1.txt`, `texte2.txt`, *etc.* et le fichier `table.txt`.

Etape 3 (Semaine 2,5/11)

Dans cette étape, pas de programmation. Il faut préparer les tests qui seront nécessaires par la suite. L'archive doit être complétée avec différents fichiers de tables et de textes, pour différentes grammaires que vous inventerez.

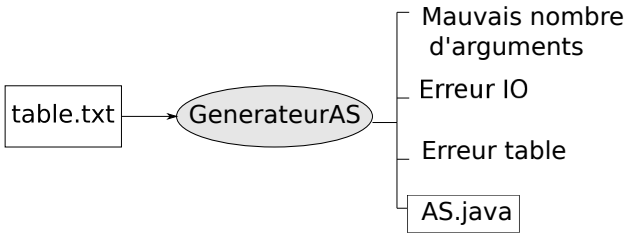
La nomenclature ci-dessous doit être respectée :

- Les grammaires sont rangées dans des fichiers de nom `gram1.txt`, `gram2.txt`, *etc.*
- Les tables correspondantes sont rangées dans des fichiers de nom `table1.txt`, `table2.txt`, *etc.*
- Les fichiers contenant les textes utilisés pour tester la grammaire `gram1.txt` doivent s'appeler `texte1-1.txt`, `texte1-2.txt`, *etc.*
- Plus généralement, les fichiers contenant les textes utilisés pour tester la grammaire `gramk.txt` doivent s'appeler `textek-1.txt`, `textek-2.txt`, *etc.*

Tous les fichiers relatifs à la grammaire `gramk.txt` sont rangés dans le répertoire `grammairek`. Tous ces fichiers et répertoires sont contenus dans l'archive. Ils doivent tous avoir été testés en utilisant l'archive développée à l'étape précédente.

Etape 4 (Semaine 4,5/11)

L'objectif de cette étape est l'automatisation du processus de construction de l'analyseur. A l'issue de cette étape, vous devez déposer une nouvelle archive qui, lorsqu'elle s'exécute, lit la table dans un fichier texte et génère un analyseur conçu comme celui écrit à la main dans les étapes précédentes.



% `java -jar GenerateurAS.jar`
% Mauvais nombre d'arguments
% `java -jar GenerateurAS.jar` table.txt
% Erreur table
% `java -jar GenerateurAS.jar` table.txt
% Génération de AS.java

Etape 5 (Semaine 5/11)

A ce stade du développement, il faut imaginer quelles sont les classes nécessaires à la mémorisation d'une grammaire (probablement `Grammaire`, `Regle`, `Terminal`, `NonTerminal`). Dessinez le diagramme de classes UML. Pour chaque de grammaire étudiée, écrivez (et testez) le code Java qui crée les objets correspondants.

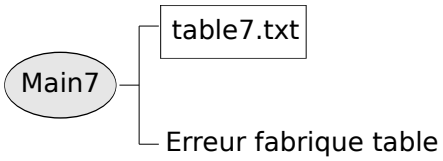
Le répertoire `grammairek` de l'archive doit être complété par les fichiers `Main1.java`, *etc.*, `Maink.java`, contenant respectivement les codes Java qui créent la représentation interne des grammaires `gram1.txt`, etc., `gramk.txt`.

A l'issue de cette étape, vous devez déposer une nouvelle archive incluant les fichiers `Main.java`, ainsi que le diagramme de classes.

Etape 6 (Semaine 6/11)

L'objectif de cette étape est l'automatisation du processus de construction de la table. Il faut compléter les classes développées à l'étape précédente avec les fonctions permettant de construire les premiers, suivants, ... La classe `Grammaire` contient des fonctions de construction et de sauvegarde de la table dans un fichier. Le format de ce fichier doit être rigoureusement identique au format des fichiers `tablek.txt` : cette contrainte permet de réaliser facilement des tests.

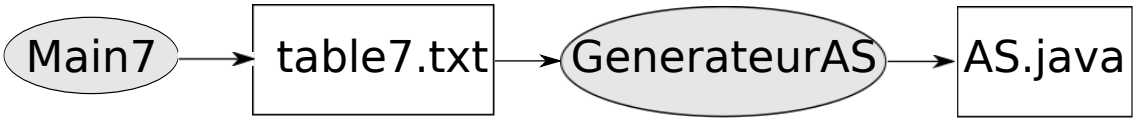
Dans la nouvelle archive à déposer, les classes `Maink` sont exécutables ; elles n'ont besoin d'aucun argument pour produire le fichier contenant la table ou un message d'erreur lorsque ce n'est pas possible.



A l'issue de cette étape, vous devez déposer une nouvelle archive incluant tout.

Etape 7 (Semaine 8/11)

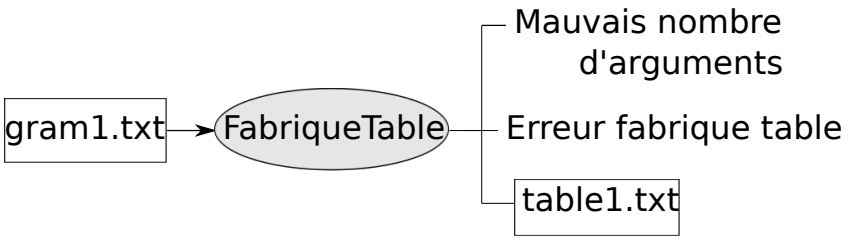
L'objectif de cette étape est la fusion des résultats obtenus lors des étapes 4 et 6. A souRègler que dans ce contexte, aucun message d'erreur n'est plus attendu de la part de `GenerateurAS`.



A l'issue de cette étape, vous devez déposer une nouvelle archive incluant tout.

Etape 8 (Semaine 8,5/11)

L'objectif de cette étape est l'automatisation de la production de la représentation mémoire d'une grammaire, qui se placera en amont de tout le travail effectué jusqu'à maintenant. Il faut écrire un analyseur syntaxique qui lit une grammaire dans un fichier texte et, si la grammaire est correctement écrite (selon la syntaxe définie à la page 1), produit sa représentation intermédiaire. Il reste à construire et sauvegarder la table de cette dernière, comme à l'étape 6.

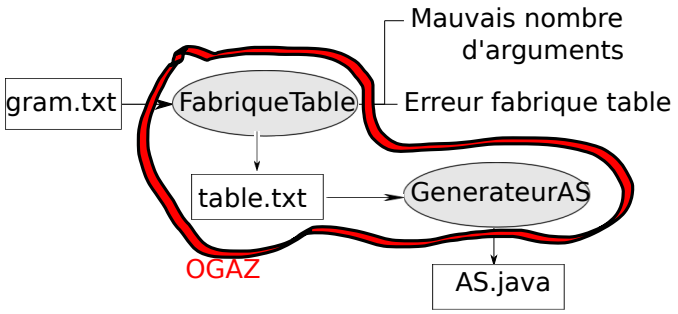


```
% java ogaz.FabriqueTable
% Mauvais nombre d'arguments
% java ogaz.FabriqueTable  gram18.txt
% Erreur fabrique table
% java ogaz.FabriqueTable  gram1.txt
% Génération de table1.txt
```

A l'issue de cette étape, vous devez déposer une nouvelle archive incluant tout.

Etape 9 (Semaine 10/11)

L'objectif de cette ultime étape est la fusion des résultats obtenus lors des étapes 8 et 7.



A l'issue de cette étape, vous devez déposer une nouvelle archive incluant tout.

Etape 10(Semaine 11/11) Cette dernière semaine est l'occasion d'étudier des grammaires de types différents : grammaires LL(1) à une autorisation près, grammaires récursives gauches, *etc.* Cette étude doit donner lieu à la rédaction d'un document précisant l'évolution des classes à prévoir, selon la nature des grammaires étudiées. Si vous disposez de plus de temps, vous pouvez compléter cette étude théorique par sa mise en pratique et fournir un outil qui accepte d'autres types de grammaires.