amU Faculté des sciences Aix Marseille Université

Année 2024-25

Domaine Sciences et Technologies
Licence mathématiques-informatique
Compilation : TP 3
Codes UE : SIN6U64L
TP 3 : analyseur SLR

1 Introduction

Dans ce TP, on va implémenter un analyseur SLR qui permet de calculer la dérivation d'un mot à partir d'une grammaire.

1.1 Fork du projet

Comme pour le premier TP, vous allez devoir forker un dépôt git pour ensuite le cloner en local. Référez-vous aux consignes du premier TP.

Le dépôt à forker se trouve à l'adresse suivante :

https://etulab.univ-amu.fr/alaboure/slr-parser-template

1.2 Exécution du projet

Pour compiler et exécuter votre programme, il faut passer par l'onglet *gradle* à droite.

- pour les tests, il faut cliquer deux fois sur `SLRParser` -> `Tasks` -> `verification` -> `test`.
- pour l'exécution du code, il faut cliquer deux fois sur `SLRParser` -> `Tasks` -> `application` -> `run` pour exécuter la méthode `main` présente dans la classe `slr.Main`.

2 Exécution d'un analyseur SLR

Le but de la première partie du TP est d'implémenter l'exécution d'un analyseur SLR. On n'implémentera pas pour le moment l'initialisation de la table à partir de la grammaire.

2.1 Code déjà présent

Le dépôt contient déjà du code réparti en deux *packages* :

- le *package* `slr.grammar` contient :
  - une interface `Symbol` pour les symboles (terminaux ou non-terminaux) d'une grammaire ;
  - une classe `AbstractSymbol` qui sert classe de base pour les classes de symboles terminaux et non-terminaux ;
  - une classe `TerminalSymbol` pour les symboles terminaux d'une grammaire ;
  - une classe `NonTerminalSymbol` pour les symboles non-terminaux d'une grammaire ;
  - une classe `Rule` pour les règles de réécriture d'une grammaire ;
  - une classe `Grammar` pour les grammaires formelles.
- le *package* `slr.parser` contient :

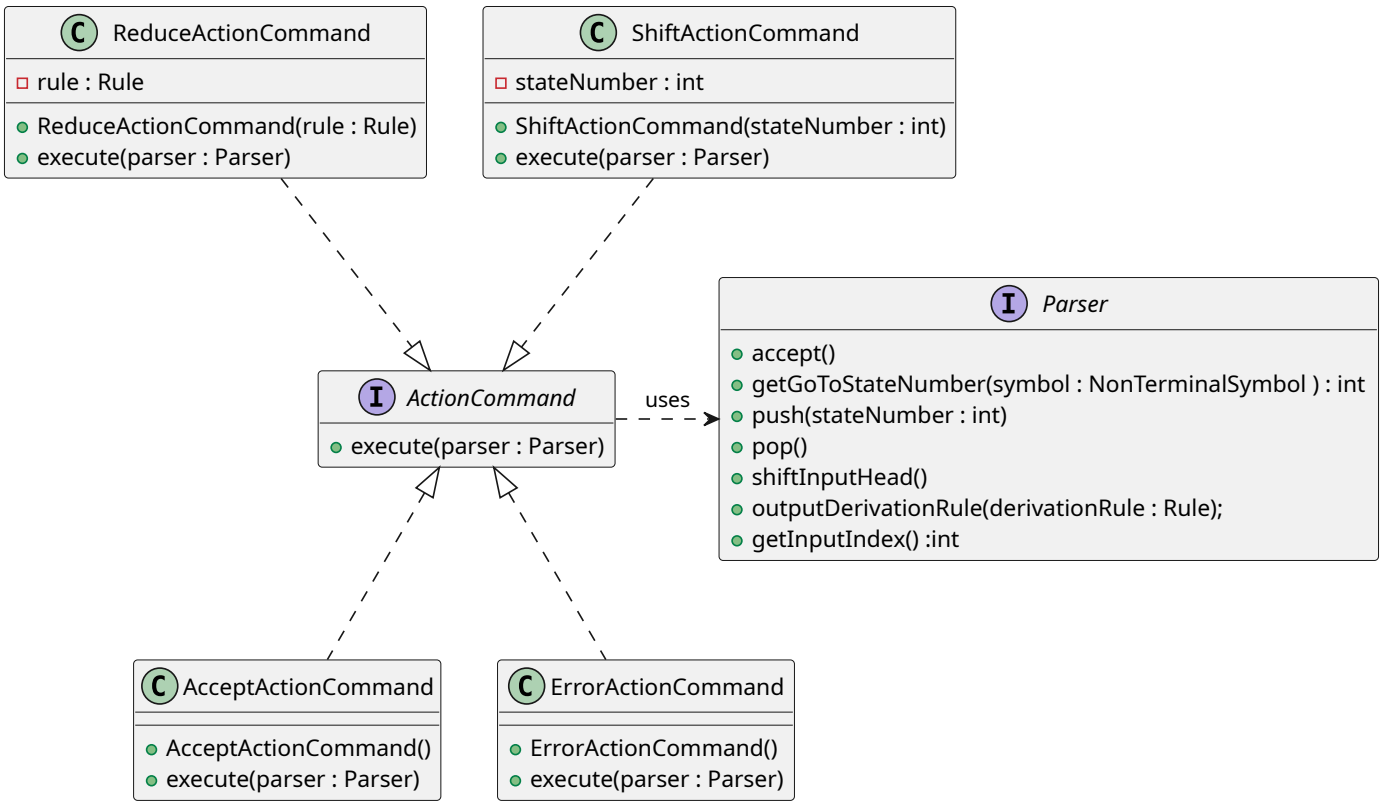
— une interface `ActionCommand` pour les actions (décalage, réduction, erreur ou acceptation) d'une table d'analyse ;
— une classe `AnalysisTable` **à compléter** pour les tables d'analyse d'analyseur SLR qui contient une table d'action et une table de *goto* ;
— une classe `InputBuffer` qui permet de gérer la lecture de l'entrée par l'analyseur en gérant la position de la tête de lecture et en générant des symboles terminaux ;
— une interface `ActionCommand` qui devra être implémentée par les classes correspondant aux quatre actions possibles présente dans la table d'analyse : décalage, réduction, erreur ou acceptation
— une interface `Parser` qui liste les méthodes qui seront utilisé par les classes implémentant `ActionCommand` pour agir sur le `SLRParser` ;
— une classe `SLRParser` **à compléter** pour l'analyseur SLR ;
— une classe `ParsingException` qui correspond à l'exception à lever lorsque l'entrée analysée par un `SLRParser` n'est pas reconnue par l'analyseur.

2.2 Ajout classes correspondant aux actions de la table d'analyse

Votre première tâche est de créer quatre classes qui implémenteront chacune l'interface `ActionCommand` et qui correspondront aux quatre actions possibles d'une table d'analyse SLR : décalage, réduction, erreur ou acceptation.

— Classe `ShiftActionCommand` qui correspond à un décalage :
  — cette classe stockera un numéro d'état j ;
  — la méthode `execute` devra faire les actions suivantes sur le `parser` :
    1. empiler j sur la pile ;
    2. avancer la tête de lecture.
— Classe `ReduceActionCommand` qui correspond à une réduction :
  — cette classe stockera une règle de réécriture $A \to \beta$;
  — la méthode `execute` devra faire les actions suivantes sur le `parser` :
    1. ajouter la règle à la sortie du `parser` ;
    2. dépiler $|\beta|$ symboles de la pile ;
    3. empiler l'état correspondant dans la table `goto` correspondant au symbole A.
— Classe `AcceptActionCommand` qui correspond à une acceptation de l'entrée :
  — la méthode `execute` devra faire accepter le mot au `parser`.
— Classe `ErrorActionCommand` qui correspond à une erreur :
  — la méthode `execute` devra lever une `ParsingException` construite avec la position de la tête de lecture du `parser`.

Le diagramme de classes suivant vous donne l'organisation de code souhaitée.



2.3 Ajout de code dans la classe `AnalysisTable`

Cette tâche consiste à compléter le code de la classe `AnalysisTable` correspondant à une table d'analyse SLR. Lisez bien la *javadoc* de la classe pour compléter son code. On rappelle que le rôle d'une table d'analyse en SLR est de stocker pour chaque état possible de l'automate, l'action à réaliser pour chaque symbole terminal ainsi que l'état auquel aller pour chaque symbole non-terminal.

2.4 Ajout de code dans la classe `Parser`

Cette tâche consiste à compléter le code de la classe `SLRParser` correspondant à un analyseur SLR. Lisez bien la *javadoc* de la classe pour compléter son code. La principale méthode de la classe est la méthode `parse` qui consiste en deux phases :

1. **Initialisation de l'analyseur :**
    1. Création du `BufferInput` ;
    2. Initialisation de la pile ;
    3. Initialisation des autres données membres de l'analyseur.
2. **Analyse** : répéter les deux suivantes tant qu'une action `accept` n'a pas été faite :
    1. Afficher l'état de la pile et des règles de dérivations ;
    2. Récupérer l'état courant de l'automate (tête de la pile) ;
    3. Récupérer l'action à effectuer à partir de la table d'analyse ;
    4. Exécuter l'action récupérée.

2.5 Test de votre *parser*

Pour tester le fonctionnement de votre `SLRParser`, on vous donne le code d'une méthode `main` de la classe `SLR.Main` qui exécute un *parser* sur une entrée. La grammaire est la suivante :

1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to a$

La table d'analyse qui est initialisé dans le code est la suivante :

état	action						goto		
	a	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6				acc			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Le code de la méthode `main` exécute le `parser` avec la table d'analyse ci-dessus. Vous devriez obtenir l'affichage suivant :

```
Stack : [0]
Derivation rules : []
Stack : [0, 5]
Derivation rules : []
Stack : [0, 3]
Derivation rules : [F→a]
Stack : [0, 2]
Derivation rules : [T→F, F→a]
Stack : [0, 2, 7]
Derivation rules : [T→F, F→a]
Stack : [0, 2, 7, 5]
```

```
Derivation rules : [T→F, F→a]
Stack : [0, 2, 7, 10]
Derivation rules : [F→a, T→F, F→a]
Stack : [0, 2]
Derivation rules : [T→T*F, F→a, T→F, F→a]
Stack : [0, 1]
Derivation rules : [E→T, T→T*F, F→a, T→F, F→a]
Stack : [0, 1, 6]
Derivation rules : [E→T, T→T*F, F→a, T→F, F→a]
Stack : [0, 1, 6, 5]
Derivation rules : [E→T, T→T*F, F→a, T→F, F→a]
Stack : [0, 1, 6, 3]
Derivation rules : [F→a, E→T, T→T*F, F→a, T→F, F→a]
Stack : [0, 1, 6, 9]
Derivation rules : [T→F, F→a, E→T, T→T*F, F→a, T→F, F→a]
Stack : [0, 1]
Derivation rules : [E→E+T, T→F, F→a, E→T, T→T*F, F→a, T→F, F→a]
```

3 Construction de la table d'analyse

Le but de cette partie du TP est de construire la table d'analyse à partir de la grammaire. On rappelle les étapes de cette construction :

1. Augmentation de la grammaire.
2. Construction des ensembles d'articles de départ (FERMETURE).
3. Construction de la fonction de transition (ALLER_A).
4. Construction des SUIVANT() pour la grammaire.

3.1 Augmentation de la grammaire

Pour cette étape, on crée une grammaire augmentée G' en rajoutant un nouvel axiome S dont la seule production est $S \rightarrow P$ vers l'ancien axiome P de la grammaire originale G.

Tâche : Créer une classe `GrammarAugmenter` contenant une unique méthode `Grammar augment(Grammar grammar)` qui retourne la grammaire augmentée de la grammaire passée en argument.

3.2 Notions d'articles et de fermeture

Un *article* est une production avec un marqueur spécial $\bullet$ en partie droite de la production. Ce marqueur indique la partie du manche qui a déjà été observée sur la pile.

Tâche : Créer une classe `Item` permettant de représenter des articles.

La fonction FERMETURE(I) prend un ensemble d'articles et, pour tout article $A \rightarrow \alpha \bullet B\gamma$ dans FERMETURE(I), ajoute $B \rightarrow \bullet\beta$ pour toute règle $B \rightarrow \beta$ de la grammaire.

Tâche : Créer une classe `ClosureComputer` contenant une unique méthode `Set<Item> closure(Item item, Grammar grammar)` qui retourne la fermeture d'un article pour une grammaire.

3.3 Construction des ensembles d'articles de départ (FERMETURE)

Pour cette étape, on doit construire l'ensemble d'articles initial FERMETURE($\{S \rightarrow \bullet P\}$).

Tâche : Créer une classe `LRAutomatonBuilder` qui contiendra pour le moment une seule méthode `Set<Item> initialState(Grammar grammar)` qui calcule l'ensemble d'articles initial FERMETURE($\{S \rightarrow \bullet P\}$) d'une grammaire.

3.4 Fonction GOTO(I, X)

À partir d'un ensemble d'articles I, et d'un symbole X terminal ou non terminal de la grammaire, la fonction GOTO(I, X) est la FERMETURE de l'ensemble de tous les articles $A \rightarrow \alpha X \bullet \beta$ tels que $A \rightarrow \alpha \bullet X\beta$ est dans I. C'est-à-dire, la fonction GOTO(I, X) fait avancer le marqueur $\bullet$ pour tous les articles de I dont le prochain symbole après le marqueur est X.

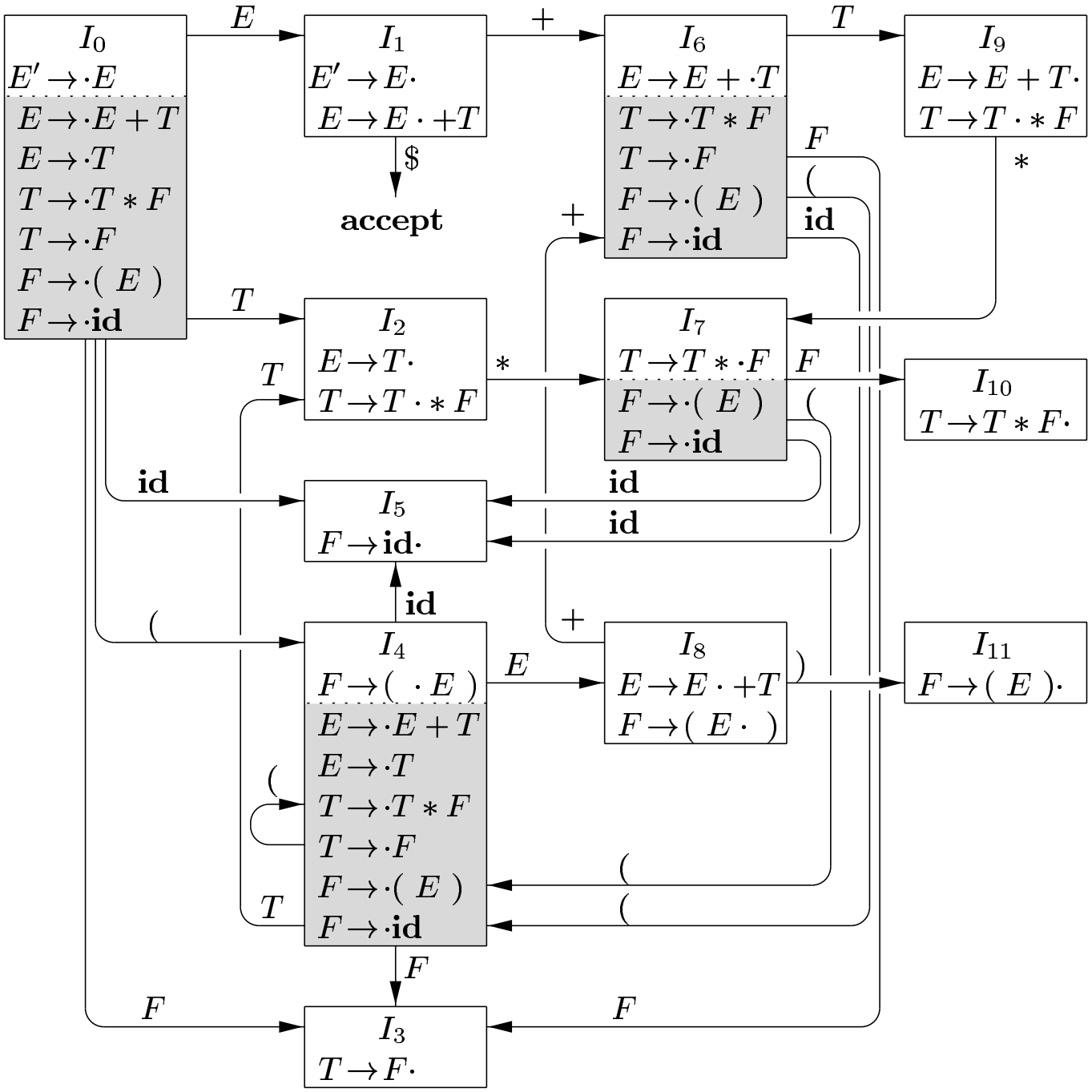
Tâche : Rajouter dans la classe `LRAutomatonBuilder` une méthode `Set<Item> goTo(Set<Item> items, Symbol symbol, Grammar grammar)` qui calcule l'ensemble d'articles GOTO(I, X) d'une grammaire pour I et X l'ensemble d'articles et symbole spécifié.

3.5 Automate $LR(0)$

Le but de cette étape est de construire l'automate $LR(0)$ de la grammaire de la façon suivante :

- Les états sont les ensembles d'articles I calculés précédemment
- Les transitions sont données par les valeurs de GOTO(I, X)
- L'état initial est FERMETURE($\{S \rightarrow \bullet P\}$
- Une transition étiquetée \$ depuis l'état FERMETURE($\{S \rightarrow P \bullet\}$) mène à l'état `accept`

L'automate $LR(0)$ de la grammaire utilisée en exemple est le suivant (avec id correspondant à a).



Tâche : Créez une classe `LRAutomaton` permettant de représenter un automate $LR(0)$.

Tâche : Rajouter dans la classe `LRAutomatonBuilder` une méthode `LRAutomaton buildAutomaton(Grammar grammar)` qui calcule l'automate $LR(0)$ de la grammaire spécifiée.

3.6 PREMIER(X)/SUIVANT(X)

Pour chaque symbole non terminal X de la grammaire, on a besoin de calculer PREMIER(X) et SUIVANT(X).

Pour calculer PREMIER(X), on applique les règles suivantes jusqu'à ce qu'aucun symbole ne puisse être ajouté à PREMIER(X).

1. Si $X \to \varepsilon$ est une production de la grammaire, on ajoute ε à PREMIER(X).
2. Si $X \to Y_1 \dots Y_k \in P$, mettre a dans PREMIER(X) s'il existe i tel que a est dans PREMIER(Y_i) et que ε est dans tous les PREMIER(Y_1)$\dots$PREMIER(Y_{i-1}). Si $\varepsilon \in$ PREMIER(Y_j)$\forall j, 1 \leq j \leq k$, alors on ajoute ε à PREMIER(X) [1].

Tâche : Créez une classe `EmptySymbol` implémentant `Symbol` et correspondant au mot vide.

Tâche : Créez une classe `GrammarOperation` contenant une méthode `Set<Symbol> first(NonTerminalSymbol symbol,Grammar grammar)` qui calcule PREMIER(X) avec X le symbole spécifié.

Pour calculer SUIVANT(X), on applique les règles suivantes jusqu'à ce qu'aucun symbole ne puisse être ajouté à SUIVANT(X).

1. Mettre \$ dans SUIVANT(S).
2. si $A \to \alpha X \beta$, le contenu de PREMIER(β), excepté ε, est ajouté à SUIVANT(X).
3. s'il existe une règle $A \to \alpha X$ ou une règle $A \to \alpha X \beta$ telle que $\varepsilon \in$ PREMIER(β) (c'est-à -dire $\beta \overset{*}{\Rightarrow} \varepsilon$), les éléments de SUIVANT(A) sont ajoutés à SUIVANT(X).

Tâche : Créez une classe `GrammarOperation` contenant une méthode `Set<Symbol> follow(NonTerminalSymbol symbol,Grammar grammar)` qui calcule SUIVANT(X) avec X le symbole spécifié.

3.7 Construction de la table SLR

Pour cette dernière étape, on doit calculer la table d'analyse SLR pour la grammaire comme suit.

1. Construire $C = \{I_0, I_1, \dots, I_n\}$ la collection d'ensemble d'articles $LR(0)$ pour G'
2. L'état i est construit à partir de I_i. Les actions d'analyse syntaxique pour l'état i sont déterminées comme suit :

   1. Si $A \to \alpha \bullet a\beta$ est dans I_i et si GOTO(I_i, a) $= I_j$, alors ACTION$[i, a] = dj$. Dans ce cas, a doit être un terminal.
   2. Si $A \to \alpha\bullet$ est dans I_i, alors ACTION$[i, a] = rj$ où j est le numéro de la règle $A \to \alpha$ pour tout $a \in$ SUIVANT(A), à l'exception de S'
   3. Si $S' \to S\bullet$ est dans I_i, alors ACTION$[i, \$] = acc$

   Si un conflit entre différentes actions résulte de ces règles, la grammaire n'est pas SLR. L'algorithme ne permet pas dans ce cas la construction d'un analyseur syntaxique.
3. Les transitions de transfert GOTO$[i, A]$ pour l'état i sont construites pour tout non-terminal A comme suit : si GOTO(I_i, A) $= I_j$ alors GOTO$[i, A] = j$

[1] PREMIER(a) $= \{a\}$ pour a terminal.

4. Toutes les entrées non remplies par les règles 2 et 3 sont positionnées à `err` (cellules vides)
5. L'état initial est celui construit à partir de l'ensemble d'items contenant $S' \to \bullet S$

Tâche : Créez une classe `SLRAnalysisTableBuilder` contenant une unique méthode `AnalysisTable computeAnalysisTable(Grammar grammar)` qui retourne la table d'analyse SLR correspondant à la grammaire spécifiée ou levant une exception si la grammaire n'est pas $SLR(1)$ (c'est-à-dire que la table contient un conflit).