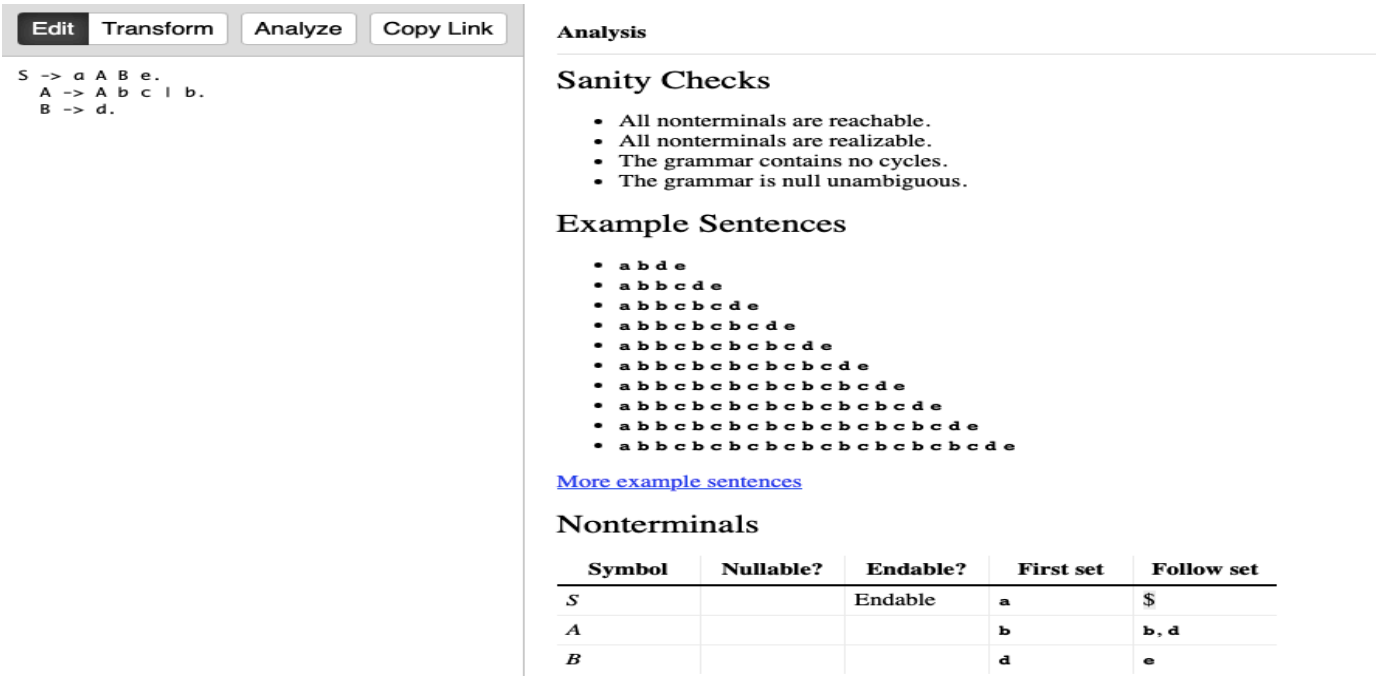
TP 10 --- L3 -- Langage et Compilation

Plan du TP :

I. Grammophone



Edit | Transform | Analyze | Copy Link

```
S -> a A B e.
  A -> A b c | b.
  B -> d.
```

Analysis

Sanity Checks

- All nonterminals are reachable.
- All nonterminals are realizable.
- The grammar contains no cycles.
- The grammar is null unambiguous.

Example Sentences

- abde
- abbcde
- abbcbcde
- abbcbcbcde
- abbcbcbcbcde
- abbcbcbcbcbcde
- abbcbcbcbcbcbcde
- abbcbcbcbcbcbcbcde
- abbcbcbcbcbcbcbcbcde
- abbcbcbcbcbcbcbcbcbcde

More example sentences

Nonterminals

Symbol	Nullable?	Endable?	First set	Follow set
S		Endable	a	$
A			b	b, d
B			d	e

1. Signification des Expressions :

- **All nonterminals are reachable :**
  Tous les non-terminaux sont atteignables : *Chaque non-terminal peut être atteint à partir de l'axiome de départ.*

- **All nonterminals are realizable :**
  Tous les non-terminaux sont réalisables : Chaque non-terminal peut produire une chaîne de terminaux.

- **The grammar contains no cycles** :
  La grammaire ne contient pas de cycles : Il n'y a pas de règles qui permettent de revenir au même non-terminal indéfiniment

- **The grammar is null unambiguous :**
  La grammaire est sans ambiguïté nulle : Il n'y a pas de production ambiguë qui pourrait générer une chaîne vide.

2. Traduction des expressions

- **first set :** ensemble des premiers

- **follow set :** ensemble des suivants
- **nullable :** annulable
- **endable :** terminable

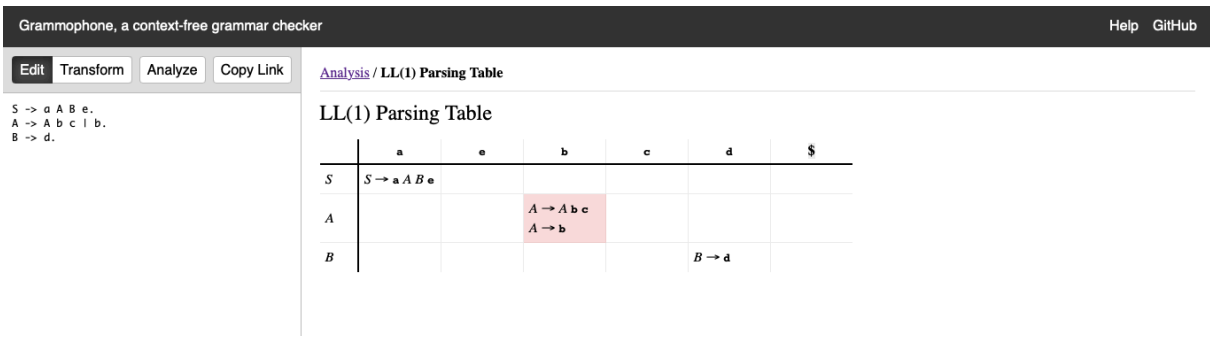
II. Grammaires et analyse LL

Manipulation de grammaire. Rendre une grammaire LL(1)

Considérons la grammaire suivante qui n'est pas LL(1).

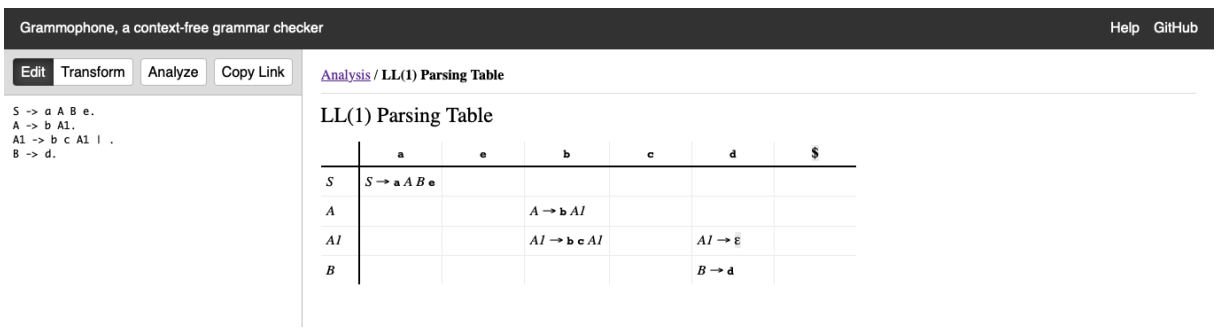
```
S -> a A B e.
A -> A b c | b.
B -> d.
```

Résultat d'analyse



Conflit : Il y a un conflit de prédiction dans la table LL(1) pour le non-terminal A car A peut commencer par b dans les deux productions.

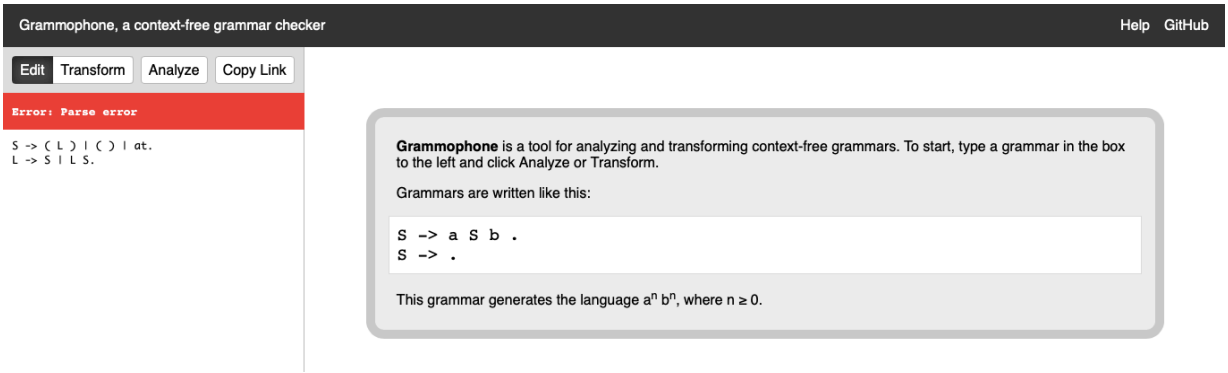
Transformation :



Grammaire initiale :

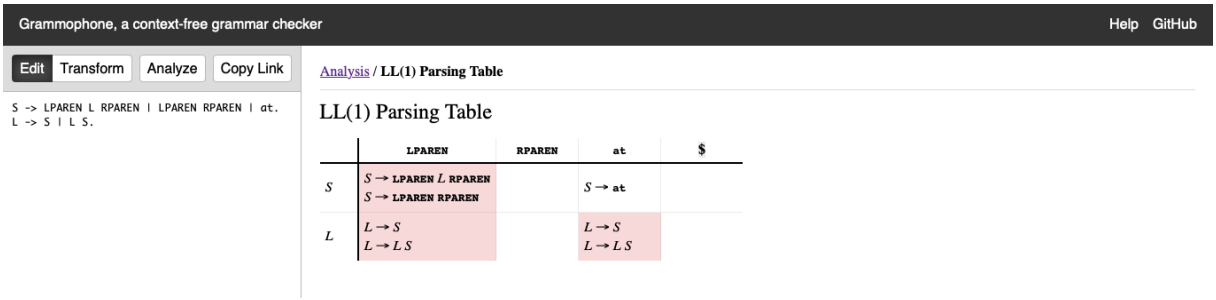
```
S -> ( L ) | ( ) | at.
L -> S | L S.
```

Résultat d'analyse



Conflit : on remarque que nous avons une erreur. Cette erreur est dû à un problème d'ambiguïté car le grammophone ne reconnais pas certain caractère nous allons lever d'ambiguïté et relancer l'analyse.

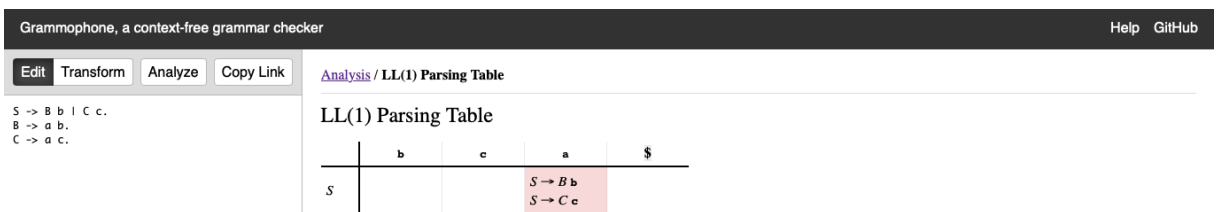
Transformation :



Grammaire initiale :

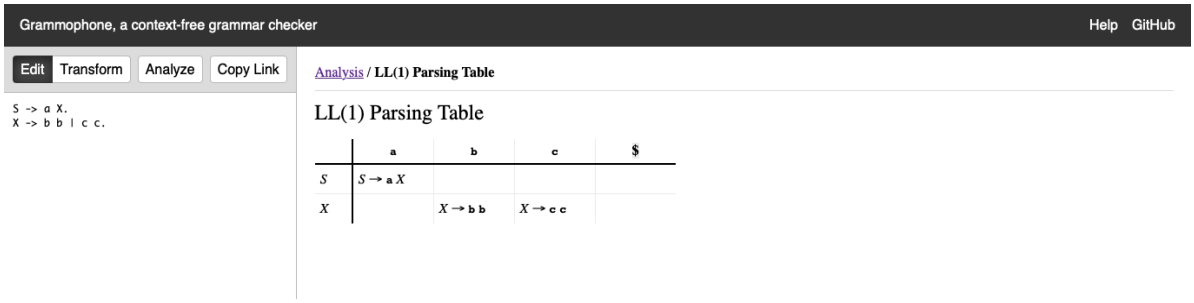
```
S -> B b | C c.
B -> a b.
C -> a c.
```

Résultat d'analyse :



Conflit : Les productions B -> a b et C -> a c partagent un préfixe commun (a), ce qui rend la grammaire non LL(1). L'analyseur ne peut pas décider entre B et C en ne regardant que le premier symbole (a).

Transformation :

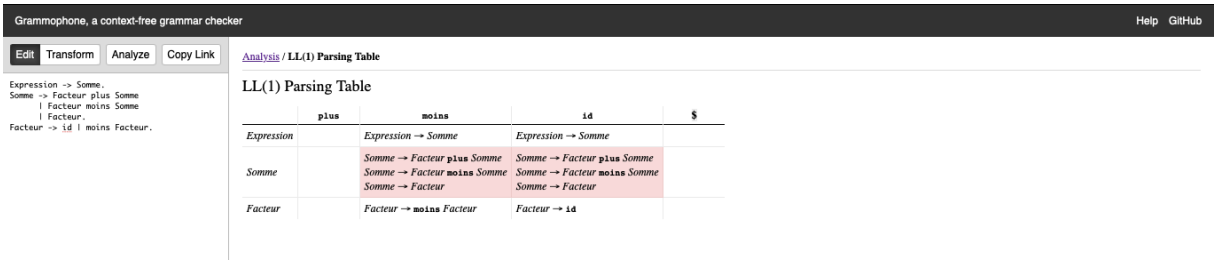


grammaires reconnaissant des expressions arithmétiques.

1. Grammaire initiale :

```
Expression -> Somme.
Somme -> Facteur plus Somme
      |  Facteur moins Somme
      |  Facteur.
Facteur -> id | moins Facteur.
```

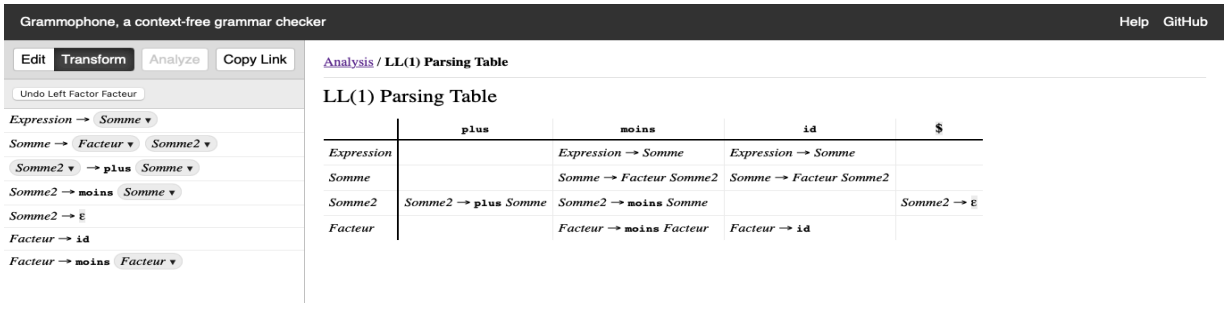
Résultat d'analyse



Conflit :

Cette grammaire est récursive à droite et elle n'a pas de récursivité gauche.

Transformation :

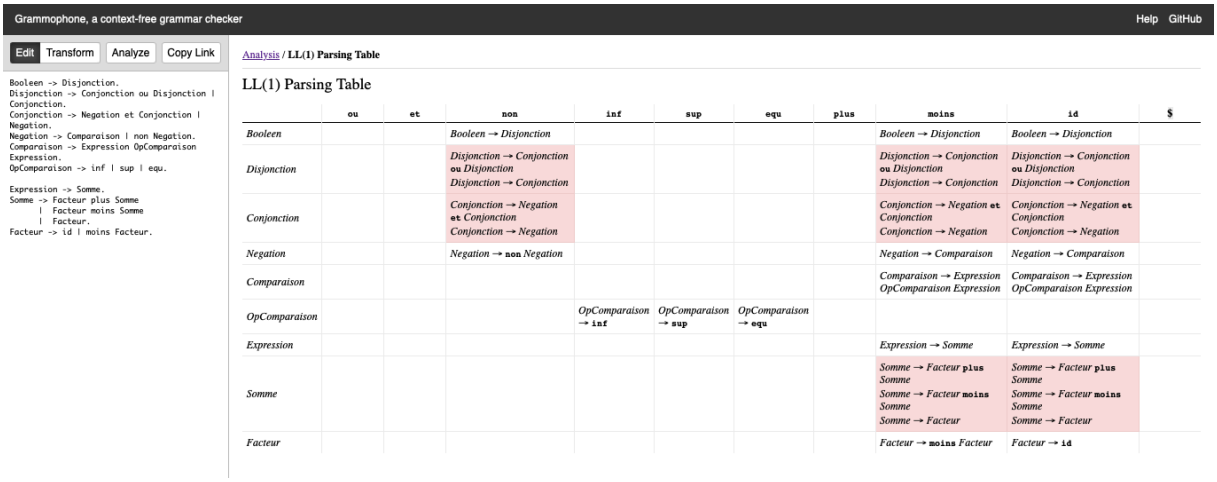


2. Même question en ajoutant des expressions booléennes :

```
Booleen -> Disjonction.
Disjonction -> Conjonction ou Disjonction | Conjonction.
Conjonction -> Negation et Conjonction | Negation.
Negation -> Comparaison | non Negation.
Comparaison -> Expression OpComparaison Expression.
OpComparaison -> inf | sup | equ.

Expression -> Somme.
Somme -> Facteur plus Somme
      |  Facteur moins Somme
      |  Facteur.
Facteur -> id | moins Facteur.
```

Résultat d'analyse



Grammophone, a context-free grammar checker — Help GitHub

Edit | Transform | Analyze | Copy Link

```
Booleen -> Disjonction.
Disjonction -> Conjonction ou Disjonction |
Conjonction.
Conjonction -> Negation et Conjonction |
Negation.
Negation -> Comparaison | non Negation.
Comparaison -> Expression OpComparaison
Expression.
OpComparaison -> inf | sup | equ.

Expression -> Somme.
Somme -> Facteur plus Somme
      |  Facteur moins Somme
      |  Facteur.
Facteur -> id | moins Facteur.
```

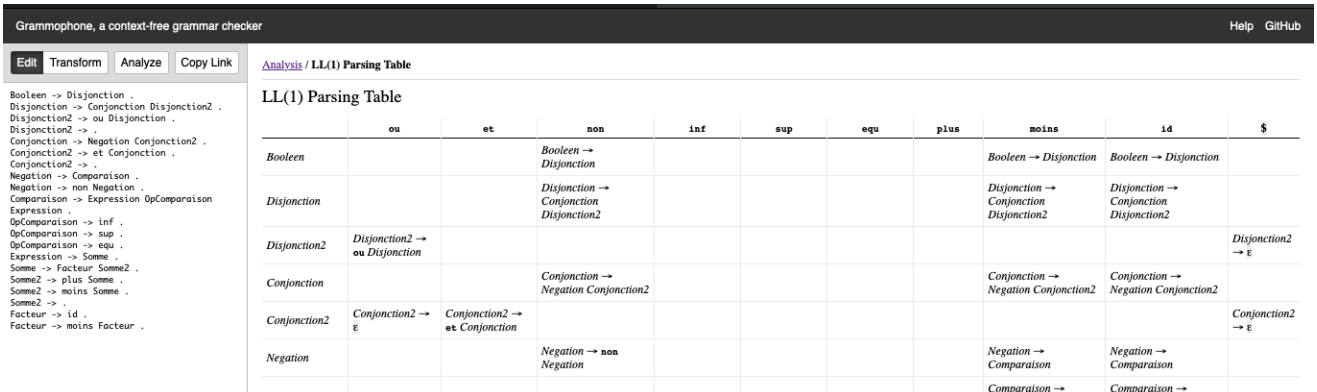
Analysis / **LL(1) Parsing Table**

LL(1) Parsing Table

	ou	et	non	inf	sup	equ	plus	moins	id	$
Booleen			*Booleen* → *Disjonction*					*Booleen* → *Disjonction*	*Booleen* → *Disjonction*	
Disjonction			*Disjonction* → *Conjonction* **ou** *Disjonction* *Disjonction* → *Conjonction*					*Disjonction* → *Conjonction* **ou** *Disjonction* *Disjonction* → *Conjonction*	*Disjonction* → *Conjonction* **ou** *Disjonction* *Disjonction* → *Conjonction*	
Conjonction			*Conjonction* → *Negation* **et** *Conjonction* *Conjonction* → *Negation*					*Conjonction* → *Negation* **et** *Conjonction* *Conjonction* → *Negation*	*Conjonction* → *Negation* **et** *Conjonction* *Conjonction* → *Negation*	
Negation			*Negation* → **non** *Negation*					*Negation* → *Comparaison*	*Negation* → *Comparaison*	
Comparaison								*Comparaison* → *Expression* *OpComparaison Expression*	*Comparaison* → *Expression* *OpComparaison Expression*	
OpComparaison				*OpComparaison* → **inf**	*OpComparaison* → **sup**	*OpComparaison* → **equ**				
Expression								*Expression* → *Somme*	*Expression* → *Somme*	
Somme								*Somme* → *Facteur* **plus** *Somme* *Somme* → *Facteur* **moins** *Somme* *Somme* → *Facteur*	*Somme* → *Facteur* **plus** *Somme* *Somme* → *Facteur* **moins** *Somme* *Somme* → *Facteur*	
Facteur								*Facteur* → **moins** *Facteur*	*Facteur* → **id**	

Conflit : Les règles Disjonction -> Conjonction ou Disjonction et Conjonction -> Negation et Conjonction présentent une récursivité gauche, ce qui n'est pas compatible avec une analyse LL(1).

Transformation :



Grammophone, a context-free grammar checker — Help GitHub

Edit | Transform | Analyze | Copy Link

```
Booleen -> Disjonction .
Disjonction -> Conjonction Disjonction2 .
Disjonction2 -> ou Disjonction .
Disjonction2 -> .
Conjonction -> Negation Conjonction2 .
Conjonction2 -> et Conjonction .
Conjonction2 -> .
Negation -> Comparaison .
Negation -> non Negation .
Comparaison -> Expression OpComparaison
Expression .
OpComparaison -> inf .
OpComparaison -> sup .
OpComparaison -> equ .
Expression -> Somme .
Somme -> Facteur Somme2 .
Somme2 -> plus Somme .
Somme2 -> moins Somme .
Somme2 -> .
Facteur -> id .
Facteur -> moins Facteur .
```

Analysis / **LL(1) Parsing Table**

LL(1) Parsing Table

	ou	et	non	inf	sup	equ	plus	moins	id	$
Booleen			*Booleen* → *Disjonction*					*Booleen* → *Disjonction*	*Booleen* → *Disjonction*	
Disjonction			*Disjonction* → *Conjonction* *Disjonction2*					*Disjonction* → *Conjonction* *Disjonction2*	*Disjonction* → *Conjonction* *Disjonction2*	
Disjonction2	*Disjonction2* → **ou** *Disjonction*									*Disjonction2* → ε
Conjonction			*Conjonction* → *Negation* *Conjonction2*					*Conjonction* → *Negation* *Conjonction2*	*Conjonction* → *Negation* *Conjonction2*	
Conjonction2	*Conjonction2* → ε	*Conjonction2* → **et** *Conjonction*								*Conjonction2* → ε
Negation			*Negation* → **non** *Negation*					*Negation* → *Comparaison*	*Negation* → *Comparaison*	
Comparaison								*Comparaison* →	*Comparaison* →	

3. Peut-on encore obtenir une grammaire équivalente LL(1) en ajoutant la règle :

```
Facteur -> ( Expression ) .
```

Non nous ne pouvons pas obtenir une grammaire équivalente LL(1)

4. Ajoutez enfin les expressions booléennes parenthésées :

```
Negation -> ( Booleen ).
```

Le Problème viens de …

III. Grammaire et Analyse LR

1. Premièr exemple

```
S -> a A B e.
A -> A b c | b.
B -> d.
```

Est-elle SLR(1) ?

Pour être **SLR(1)**, une grammaire ne doit pas avoir de **conflit shift/reduce** ou **reduce/reduce** dans la table d'analyse SLR(1).

Analyse de la table SLR(1) :

- **Le problème ici vient de la règle** `A -> A b c | b.`
- L'ensemble **FOLLOW(A) contient `b`**, ce qui crée un **conflit shift/reduce** car `b` est à la fois un début de production et une réduction possible.

Conclusion : **Cette grammaire n'est pas SLR(1)** à cause de ce conflit.

2. Analyse LR

Soit la grammaire suivante :

```
S -> a S b | a b.
```

NON, cette grammaire **n'est pas LL(1)** car il y a un conflit dans les ensembles FIRST et FOLLOW. Par Contre elle est SLR(1) car il n'y a pas conflit de type **shift/reduce** dans la table SLR

Utiliser la table d'analyse SLR(1) pour réaliser l'analyse des entrées : ab et abb.

Analyse des entrées dans la table SLR(1) :

- **Entrée : `ab`**
  1. `a` → shift
  2. `b` → reduce avec `S -> a b`
  3. Accepté
- **Entrée : `abb`**
  1. `a` → shift
  2. `b` → shift
  3. `b` → réduit avec `S -> a S b`, mais ça crée un problème car `S` attend un `b` supplémentaire pour être complet.
  4. **Problème de parsing → Rejeté**

Conclusion : la grammaire est bien SLR(1), mais elle n'est pas LL(1).

3. Conflit *shift/reduce*

Soit la grammaire suivante :

```
Instruction -> si condition alors Instruction
      |   si condition alors Instruction  sinon Instruction
      |   expression
      .
```

- **Observons les conflits dans la table d'analyse LL**

⇒ Dans **si condition alors Instruction sinon Instruction**, l'analyseur ne sait pas si **sinon** appartient à la première ou la deuxième instruction.

⇒ Il peut soit **shifter sinon** en supposant qu'il appartient à l'instruction la plus proche (priorité shift), soit **réduire**Instruction -> si condition alors Instruction trop tôt (priorité reduce).

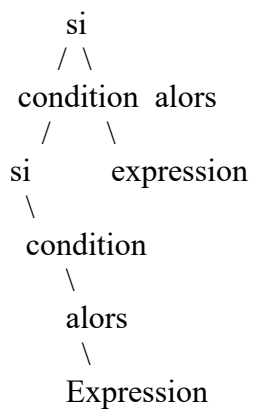
Solution

Utiliser une règle de priorité pour `sinon` :

- On force l'analyse à toujours **associer le `sinon` au `si` le plus proche**.
- On peut aussi **introduire des parenthèses pour lever l'ambiguïté** dans la syntaxe du langage.

Arbre d'analyse pour :

```
si condition alors si condition alors expression sinon expression
```



4. Conflit *reduce/reduce*

Soit la grammaire suivante :

```
S ->    V assign E
  |     id.
V ->    id.
E ->    V
  |     num.
```

Observez les conflits dans les tables d'analyse LL et SLR. Comment les résoudre ?

Observons les conflits :

- Lorsque l'analyseur lit un `id`, il ne sait pas s'il doit :
    1. Réduire immédiatement avec `S -> id`.
    2. Continuer pour voir s'il y a `assign` pour réduire avec `S -> V assign E`.
- Ce conflit se produit car `id` peut être soit un `V`, soit un `S` directement.

Solution

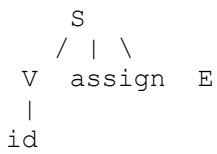
- **Imposer une priorité de réduction** :
    - On force l'analyseur à ne pas réduire immédiatement `S -> id` tant qu'il y a une chance de voir `assign`.
- **Utiliser une approche plus forte comme LR(1)** :
    - En regardant plus loin (`id assign`), l'analyse devient déterministe.

Arbre d'analyse :

- Pour **id** :

S
|
Id

- Pour **id assign num** :



IV. GRAMMAIRE LL(*) et ANTLR

1. Grammaire LL(2)

Reprenons la grammaire LL(2) précédente :

```
S -> a S b | a b.
```

Cela donne en syntaxe ANTLR :

```
grammar ll2;

start : s EOF;

s : 'a' s 'b'
  | 'a' 'b'
  ;
```

La grammaire `ll2.g` est-elle autorisée par ANTLR ?

Reponse :

- **NON** pour LL(1) car elle nécessite LL(2) (regarder deux symboles en avant).
- **OUI** pour LL(*) car ANTLR utilise une analyse plus puissante qui permet des prédictions sur plusieurs tokens.

Soit la grammaire suivante :

```
A ->    C
  |     B.
C ->    a C a
  |     b.
B ->    a B
  |     c.
```

Est-elle LL(1) ?

- **NON**, car `FIRST(A) = { a, b, c }` et `FOLLOW(A)` crée un conflit entre `B` et `C`.

Est-elle SLR(1) ?

- **NON**, car elle a des conflits shift/reduce dans la table d'analyse LR(0).

Est-elle LL(k) ?

- **OUI pour LL(2)**, car avec deux symboles d'anticipation, on peut distinguer `C` de `B`.

Est-elle LL(*) ?

- **OUI**, ANTLR peut la gérer grâce à sa puissance de lookahead illimité.