# Compilation

## Analyse ascendante

### SI4 — 2018-2019

Erick Gallesio

# Introduction

**Principe de l'analyse:**

> A partir de la phrase à analyser, on va essayer de remonter à l'axiome par réductions successives.

A chaque étape, on essaie donc de reconnaître une partie droite de règle et de la remplacer par le non terminal qui la produit (càd partie gauche de la règle).

```
E → aABe                    (r1)
A → Abc | b                 (r2, r3)
B → d                       (r4)
```

Analyse de la phrase `abbcde`.

| entrée analyseur | action |
|---|---|
| a**b**bcde | **b** apparaît en partie droite de r3 ⇒ réduire en **A** |
| a**Abc**de | **Abc** apparaît dans r2 ⇒ réduire en **A** |
| aA**d**e | **d** apparaît dans r4 ⇒ réduire en **B** |
| **aABe** | **aABe** apparaît dans r1 ⇒ réduire en **S** |
| **S** | ⇒ SUCCÈS |

On a donc: **S → aABe → aAde → aAbcde → abbcde**

# Exemple d'analyse ascendante

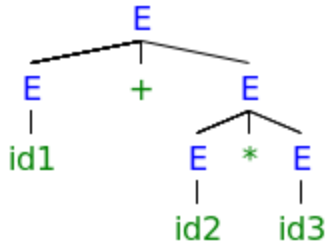
**Remarque:**

Si on réussit à remonter jusqu'à l'axiome, on a construit (à l'envers) la dérivation la plus à droite.

```
E → E + E | E * E | (E) | id
```

Analyse de la phrase: $\mathbf{id_1 + id_2 * id_3}$

| **entrée analyseur** | **handle** | **règle de réduction** |
|---|---|---|
| $\mathbf{id_1}$ + $id_2$ * $id_3$ | $id_1$ | E → id |
| E + $\mathbf{id_2}$ * $id_3$ | $id_2$ | E → id |
| E + E * $\mathbf{id_3}$ | $id_3$ | E → id |
| E + **E * E** | E * E | E → E * E |
| **E + E** | E + E | E → E + E |
| **E** | | SUCCÈS |




On a donc: $\mathbf{E \rightarrow E + E \rightarrow E + E * E \rightarrow E + E + id_3 \rightarrow E + id_2 * id_3 \rightarrow id_1 + id_2 * id_3}$

# Analyseur shift-reduce: Principe

Pour analyser un programme on va avoir un analyseur qui possède:

- une pile qui contient des symboles de la grammaire
- un buffer qui contient le mot *m* à analyser

On utilise le caractère `$` :

- pour marquer le fond de la pile
- pour marquer la fin du mot à analyser

L'analyseur travaille en

- décalant (**shift**) 0 ou plusieurs symboles de l'entrée vers la pile;
- réduisant (**reduce**) une *handle* β lors quelle se trouve en sommet de la pile.

Lorsqu'on on a `$S` dans la pile et que l'entrée est réduite à `$` , le mot *m* est accepté.

# Analyseur shift-reduce: Exemple

```
E → E + E | E * E | (E) | id
```

Analyse de la phrase: $\mathbf{id_1 + id_2 * id_3}$

| | Pile | Entree | Action |
|---|---|---|---|
| 1 | $ | $id_1 + id_2 * id_3$ | shift |
| 2 | $$id_1$ | $+ id_2 * id_3$ | reduce (E→id) |
| 3 | $E | $+ id_2 * id_3$ | shift |
| 4 | $E+ | $id_2 * id_3$ | shift |
| 5 | $E+$id_2$ | $* id_3$ | reduce (E→id) |
| 6 | $E+E | $* id_3$ | shift |
| 7 | $E+E* | $id_3$ | shift |
| 8 | $E+E*$id_3$ | $ | reduce (E→id) |
| 9 | $E+E*E | $ | reduce (E→E*E) |
| 10 | $E+E | $ | reduce (E→E+E) |
| 11 | $E | $ | SUCCES |

**Remarque:**

- à l'étape 6 on a choisi *shift* plutôt que *reduce* (avec E→E+E)
- on dit ici que l'on a un conflit *shift*/*reduce*

# Conflits dans un analyseur shift-reduce (1 / 2)

Une grammaire algébrique peut ne pas être analysable avec un analyseur *shift-reduce*.

On se trouve dans ce cas lorsqu'on ne peut pas décider entre

- décaler ou réduire (**conflit shift/reduce**)
- plusieurs réductions possibles (**conflit reduce/reduce**)

**Conflit shift-reduce:**

On se trouve dans ce cas avec:

```
instr → if ( expr ) instr
      | if ( expr ) instr else instr
      | ...
```

On peut se retrouver avec:

| PILE | ENTREE | ACTION |
|---|---|---|
| … | … | |
| $… if ( expr ) instr | else … $ | *shift* ou *reduce*?? |

- Notre grammaire est ambiguë; elle n'est donc pas LR(1).

# Conflits dans un analyseur shift-reduce (2 / 2)

| PILE | ENTREE | ACTION |
|---|---|---|
| … | … | |
| $… if ( expr) instr | else … $ | *shift* ou *reduce*?? |

En cas de **if** emboîtés:

```
if (expr1)
    if (expr2)
       instr1;
    else
       instr2;
```

- Le choix de la réduction (*reduce*) revient à «fermer» le **if** courant et donc à associer le **else** au **if** le plus externe.
- Le choix du décalage (*shift*) revient à continuer la construction et donc différer la réduction au moment où on aura analysé le **if** interne ⇒ association du **else** au **if** interne (choix habituel).
- Toutefois, comme on l'a dit, ce type d'ambiguïté n'est pas un problème en pratique.

### Conflit reduce/reduce:

Les conflits *reduce*/*reduce* sont plus rares et dénotent en général un problème dans la définition de la grammaire (voir TD).

# Analyseur LR(k)

Un analyseur LR(k):

**L:**

pour *Left to Right Scanning* car on parcourt le texte de la gauche vers la droite;

**R:**

pour *Rightmost derivations* car on reconstruit (à l'envers) les dérivations droites;

**k:**

la valeur *k* est la taille du lookahead nécessaire avant de décider comment traiter les symboles déjà vus.

Un analyseur LR est **déterministe**

- permet l'analyse sans retour arrière
- le temps d'analyse est proportionnel à la taille de l'entrée

En général, la grammaire d'un langage de programmation

- se prête bien à l'analyse LR.
- demande peu de modification pour être traitée (*vs analyse LL*)

# Analyseur LR: Problématique

```
E → T  | E + T
T → id | (E)
```

Analyse de l'expression `(id + id)$`:

| PILE | ENTREE | ACTION |
|---|---|---|
| … | … | |
| $(E+ | id)$ | shift |
| $(E+id | )$ | reduce (T → id) |
| $(E+T | )$ | reduce (E → E + T). ⚠ |
| … | … | |

⚠ : Ici, on a choisi de réduire en `E → E + T` plutôt que `E → T` car

- `(E + E` n'est pas un *préfixe viable* (ce choix conduirait à une analyse qui échoue).

**Problématique:**

> Déterminer la «handle» à réduire ne dépend pas seulement des symboles en sommet de pile, mais de **toute** la séquence qui est dans la pile. Il faut donc encoder le **contexte** d'analyse.

Dans l'exemple précédent, on est à la recherche de ')'.

# Analyseur LR: États

On modifie un peu l'analyseur shift-reduce précédent:

- au lieu de décaler des symboles dans la pile, on y place des états.
- les états encodent le contexte gauche courant
- étant donnés l'état et la fenêtre courants, on saura s'il faut
  - *réduire (reduce)*
  - *empiler (shift) un nouvel état dans la pile*

Un analyseur LR utilise deux tables:

**table d'actions:**

Une case de la table `Act[e, a]` indique ce qu'il faut faire quand on voit le terminal `a` alors qu'on est dans l'état `e`.

**table de sauts:**

Une case de la table `Goto[e, X]` indique l'état à empiler après réduction de `X`, alors qu'on est dans l'état `e`.

# Analyseur LR: Principe

- On démarre avec l'état $s_0$ sur la pile
- Soit $s_i$, l'état courant et a le lexème courant

**Algorithme:**

- si Act[$s_i$, a] = shift $s_j$, on empile l'état $s_j$ et on avance sur l'entrée.
- si Act[$s_i$, a] = reduce $X \rightarrow X_1 \ldots X_n$
  - *enlever n etats de la pile*
  - *remplacer l'état $s_t$ maintenant au sommet de la pile par Goto[$s_t$, X]*
  - *ne pas avancer*
- si Act[$s_i$, a] = accept, on a gagné!
- si Act[$s_i$, a] est vide, on a une erreur de syntaxe.

# Analyseur LR: Exemple (1 / 2)

```
E → E + T | T                    (r1, r2)
T → T * F | F                    (r3, r4)
F → ( E ) | id                   (r5, r6)
```

**Table combinée:**

| Etat courant | id | + | * | ( | ) | $ | E | T | F |
|---|---|---|---|---|---|---|---|---|---|
| 0 | s5 | | | s4 | | | 1 | 2 | 3 |
| 1 | | s6 | | | | accept | | | |
| 2 | | r2 | s7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | s5 | | | s4 | | | 8 | 2 | 3 |
| 5 | | r6 | r6 | | r6 | r6 | | | |
| 6 | s5 | | | s4 | | | | 9 | 3 |
| 7 | s5 | | | s4 | | | | | 10 |
| 8 | | s6 | | | s11 | | | | |
| 9 | | r1 | s7 | | r1 | r1 | | | |
| 10 | | r3 | r3 | | r3 | r3 | | | |
| 11 | | r5 | r5 | | r5 | r5 | | | |

- La table des actions est a gauche (terminaux)
- La table des sauts est à droite (non terminaux)

# Analyseur LR: Exemple (2 / 2)

| Etat | id | + | ( | ) | $ | E | T |
|---|---|---|---|---|---|---|---|
| 0 | s4 | | s3 | | | 1 | 2 |
| 1 | | s5 | | | accept | | |
| 2 | r2 | r2 | r2 | r2 | r2 | | |
| 3 | s4 | | s3 | | | 6 | 2 |
| 4 | r4 | r4 | r4 | r4 | r4 | | |
| 5 | s4 | | s3 | | | | 8 |
| 6 | | s5 | | s7 | | | |
| 7 | r3 | r3 | r3 | r3 | r3 | | |
| 8 | r1 | r1 | r1 | r1 | r1 | | |

**Une version simplfiée de ETF**

```
E → E + T | T      (r1, r2)
T → ( E ) | id     (r3, r4)
```

| Pile | Entrée | Action |
|---|---|---|
| $0 | id+id$ | s4 |
| $0 id4 | +id$ | r4 et passer dans l'état 2 |
| $0 T2 | +id$ | r2 et passer dans l'état 1 |
| $0 E1 | +id$ | s5 |
| $0 E1 +5 | id$ | s4 |
| $0 E1 +5 id4 | $ | r4 et passer dans l'état 8 |
| $0 E1 +5 T8 | $ | r1 et passer dans l'état 1 |
| $0 E1 | $ | accept |

# Construction de la table d'analyse (1 / 8)

### Notion d'item LR(0)

Un *item* LR(0), ou *configuration*, est une production de la grammaire avec un point dans la partie droite. Par exemple, la règle `T → T * F` a quatre items possibles:

```
T → • T * F
T → T • * F
T → T * • F
T → T * F •
```

Intuitivement:

- ce qui est à gauche du point (•) est ce qui a déjà été vu et placé dans la pile
- ce qui est à droite correspond à ce qui est susceptible d'être lu.

### Note:

- Si le point est au milieu d'une production, `T → T * • F`,
  - *on est en cours de reconnaissance d'une «handle» ⇒*
  - *on attend un terminal ∈ PREMIER(F)*
- Si le point est à fin d'une règle `T → T + F •`
  - *on a reconnu une partie droite complète ⇒*
  - *on est en présence d'une handle. On peut donc réduire.*

# Construction de la table d'analyse (2 / 8)

### Notion de fermeture

Lorsqu'on a `T → T * • F`,

- on vient de reconnaître `*` et on attend une dérivation de `F`.
- Comme `F` peut donner `id` ou `( E )`, ces trois productions correspondent au même état de l'analyseur.

```
T → T * • F
F → • id
F → • ( E )
```

L'ajout de configurations équivalentes à un ensemble de configurations est appelé **fermeture**.

Pour calculer l'ensemble configurations pour la configuration de départ `A → •α`

1. placer `A → •α` dans l'ensemble.
2. si α commence par un terminal, ne rien faire
3. si α est de la forme Bρ, ajouter toutes les productions de B avec un '•' en tête
4. continuer jusqu'à ce que l'on ne puisse plus ajouter de règle.

# Construction de la table d'analyse (3 / 8)

### Notion de successeur

Intuitivement, c'est la fonction qui permet de dire dans quel état on doit passer lorsqu'on a reconnu un symbole.

Le calcul est simple: pour chaque configuration *c* de l'ensemble *C* de configurations, on déplace le • à droite afin d'obtenir un nouvel ensemble *C'*. Construire la fermeture sur *C'*

Considérons l'item `E → E • + T`, son successeur sur `+` est l'item `E → E + • T` que l'on place dans un ensemble vide *C'*. Après fermeture, cet ensemble devient:

```
E → E + • T
T → • T * F
T → • T
```

Pour démarrer la construction de la table on va **augmenter la grammaire** en ajoutant la règle de production S' → S (où S est l'axiome)

On commence donc par mettre S' → • S dans la configuration initiale et on construit ensuite l'ensemble des configurations.

# Construction de la table d'analyse (4 / 8)

### Grammaire ETF simplifiée:

```
E' → E                    r0
E  → E + T                r1
E  → T                    r2
T  → ( E )                r3
T  → id                   r4
```

**Ensemble de configuration initial**: fermeture sur `E' → • E`

```
I0: E' → • E
    E  → • E + T
    E  → • T
    T  → • ( E )
    T  → • id
```

On peut ajouter les successeurs de chaque item:

```
I0: E' → • E            I1
    E  → • E + T        I1
    E  → • T            I2
    T  → • ( E )        I3
    T  → • id           I4
```

# Construction de la table d'analyse (5 / 8)

```
E' → E                    r0
E  → E + T                r1
E  → T                    r2
T  → ( E )                r3
T  → id                   r4
```

⇒

```
I0: E' → • E              I1
    E  → • E + T          I1
    E  → • T              I2
    T  → • ( E )          I3
    T  → • id             I4
```

```
I1: E' → E •              accept
    E  → E • + T          I5
```

```
I2: E → T •               reduce r2
```

```
I3: T → ( • E )           I6
    E → • E + T           I6
    E → • T               I2
    T → • ( E )           I3
    T → • id              I4
```

```
I4: T → id •              reduce r4
```

```
I5: E → E + • T           I8
    T → • ( E )           I3
    T → • id              I4
```

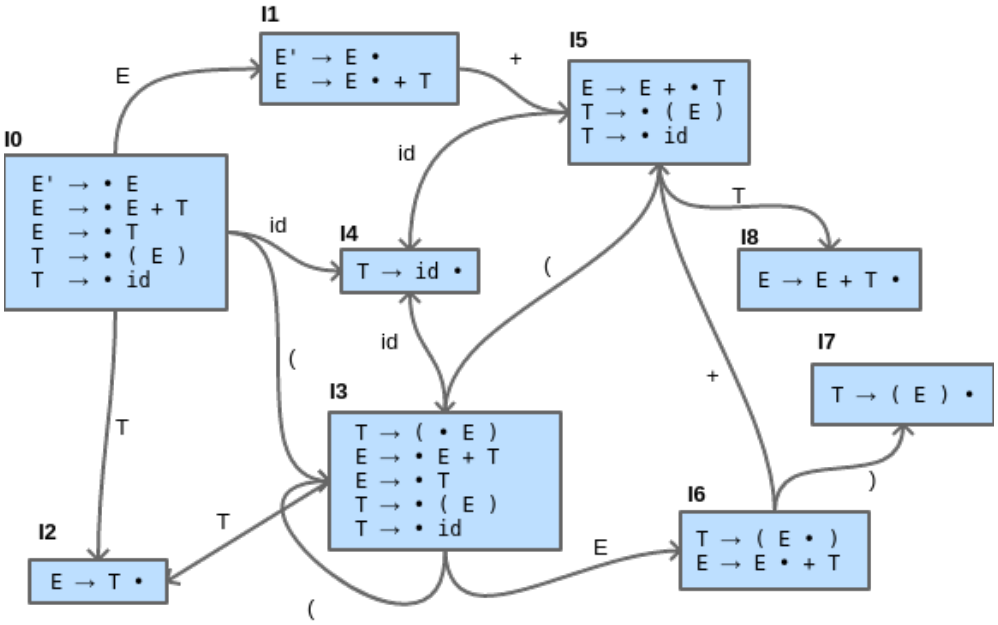
```
I6: T → ( E • )           I7
    E → E • + T           I5
```

```
I7: T → ( E ) •           reduce r3
```

```
I8: E → E + T •           reduce r1
```

# Construction de la table d'analyse (6 / 8)

On a donc le **diagramme de transition** suivant:



# Construction de la table d'analyse (7 / 8)

### Construction de la table LR(0)

1. Construire C = $\{I_0\ I_1\ \ldots\ I_n\}$ l'ensemble des ensembles de configuration.
2. Table des actions:
    1. *si $S' \rightarrow S\bullet \in I_i$ alors Act[i,a] ← accept*
    2. *si $A \rightarrow \alpha\bullet \in I_i$ alors Act[i,a] ← reduce($A \rightarrow \alpha$) pour toutes les entrées.*
    3. *si $A \rightarrow \alpha\bullet a\beta \in I_i$ et le sucesseur du terminal a est $I_j$ alors mettre Act[i,a] ← shift(j)*
3. Table des sauts:
    - *pour tous les non terminaux et pour lesquels le successeur est $I_j$ alors mettre Goto[i,a] ← j*

### Rappel:

Les cases vides sont des erreurs.

# Construction de la table d'analyse (8 / 8)

En utilisant la méthode de construction précédente.

```
I0: E' → • E          I1
    E  → • E + T      I1
    E  → • T          I2
    T  → • ( E )      I3
    T  → • id         I4
```

```
I1: E' → E •          accept
    E  → E • + T      I5
```

```
I2: E → T •           reduce r2
```

```
I3: T → ( • E )       I6
    E → • E + T       I6
    E → • T           I2
    T → • ( E )       I3
    T → • id          I4
```

```
I4: T → id •          reduce r4
```

```
I5: E → E + • T       I8
    T → • ( E )       I3
    T → • id          I4
```

```
I6: T → ( E • )       I7
    E → E • + T       I5
```

```
I7: T → ( E ) •       reduce r3
```

```
I8: E → E + T •       reduce r1
```

| Etat | id | + | ( | ) | $ | E | T |
|---|---|---|---|---|---|---|---|
| 0 | s4 | | s3 | | | 1 | 2 |
| 1 | | s5 | | | accept | | |
| 2 | r2 | r2 | r2 | r2 | r2 | | |
| 3 | s4 | | s3 | | | 6 | 2 |
| 4 | r4 | r4 | r4 | r4 | r4 | | |
| 5 | s4 | | s3 | | | | 8 |
| 6 | | s5 | | s7 | | | |
| 7 | r3 | r3 | r3 | r3 | r3 | | |
| 8 | r1 | r1 | r1 | r1 | r1 | | |

# Grammaires LR(0)

La méthode utilisée ici construit la table sans utiliser le *lookahead*.

Les grammaires reconnues sont les grammaires LR(0)

- zéro ici indique qu'on n'utilise pas de symbole d'avance
- l'analyseur décide des actions à prendre (réductions) en fonction de ce qui est déjà dans la pile, pas de la prochaine entrée.

Par conséquent, un ensemble de configuration LR(0)

- ne peut pas contenir à la fois *shift* et *reduce*
- ne peut contenir qu'un *reduce* au plus.

Peu de grammaires satisfont les contraintes LR(0) (les ε-productions par exemple posent problème).

Les grammaires LR(0) constituent donc la famille la plus faible des grammaires LR

# Analyseur SLR(1): Principe

L'analyse LR(0) n'est souvent pas suffisante à cause de l'absence de lookahead.

⇒ On veut prendre en compte le (ou les) lexème(s) qui arrive(nt) pour orienter l'analyse.

```
E' → E
E  → E + T | T
T  → (E) | id | id[E]
```

Lors de la construction des ensembles d'items on a

```
I0: E' → • E              I1
    E  → • E + T          I1
    E  → • T              I2
    T  → • ( E )          I3
    T  → • id             I4
    T  → • id [ E ]       I4

....

I4: T  → id •
    T  → id • [ E ]
```

- Analyse LR(0) ⇒ conflit entre *shift* et *reduce*
- Analyse SLR(1) ⇒ si on a un `[` , alors *shift* sinon *reduce*

# Analyseur SLR(1): Construction de la table

La construction de la table est quasi identique à la construction LR(0).

Pour l'analyse SLR(1) (*Simple LR(1)*), on ne fait qu'une seule **petite modification** en 2.1:

1. Construire $C = \{I_0\ I_1\ \ldots\ I_n\}$ l'ensemble des ensembles de configuration.
2. Table des actions:
    1. *si* $S \rightarrow S\bullet \in I_i$ *alors* $Act[i,a] \leftarrow accept$
    2. *si* $A \rightarrow \alpha\bullet \in I_i$ *alors* $Act[i,a] \leftarrow reduce(A \rightarrow \alpha)$ *pour les entrées où* $\boldsymbol{a \in SUIVANT(A)}$
    3. *si* $A \rightarrow \alpha\bullet a\beta \in I_i$ *et le successeur du terminal a est* $I_j$ *alors mettre* $Act[i,a] \leftarrow shift(j)$
3. Table des sauts:
    - *pour tous les non terminaux et pour lesquels le successeur est* $I_j$ *alors mettre* $Goto[i,a] \leftarrow j$

- On ne met plus *reduce* dans toutes les cases systématiquement (une ligne de la table peut contenir des *shifts* et des *reduces*)
- On prend donc en compte le lexème qui est en entrée.
- On a une analyse plus puissante que l'analyse LR(0).

**Note:**

Toutes les grammaires LR(0) sont SLR(1) et l'analyse SLR(1) est plus puissante que l'analyse LR(0).

# Autres méthodes d'analyse ascendante

### Analyse LR(1):

L'analyse LR(1) permet de reconnaître des formes qui seraient rejetées par SLR.

- SLR regarde la «handle» qui est en sommet de pile, mais on peut avoir parfois besoin de plus de contexte:
  - *par exemple dans* `*v = 3`, *veut ont* `(*v) = 3` *ou* `*(v = 3)` *?*

Au prix d'une modification de l'algorithme de fermeture et des successeurs, on peut construire une analyse LR(1).

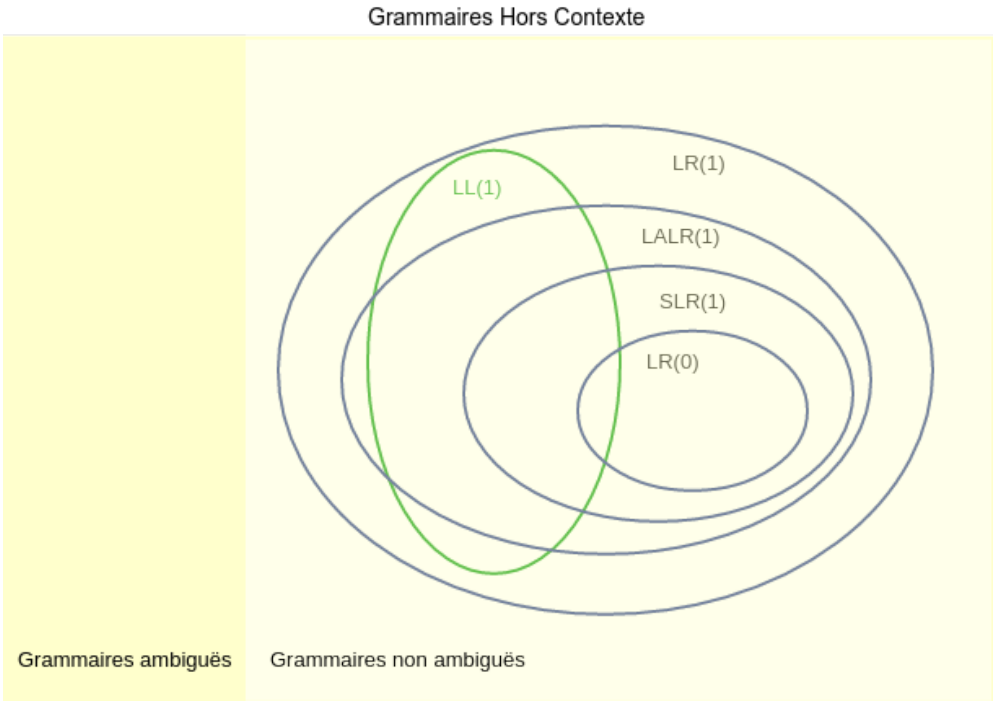
### Analyse LALR(1):

Un analyseur LR(1) nécessite **beaucoup** d'états, ce qui peut être impraticable.

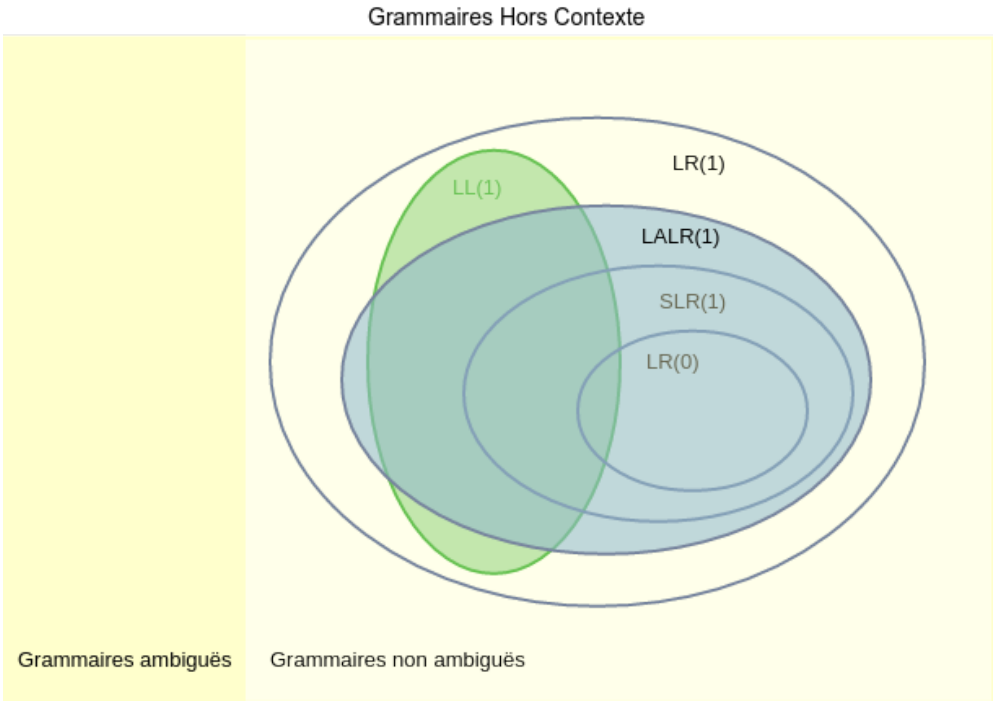
L'analyse LALR(1) (*Lookahead LR*), permet de fusionner certains de ces états et d'obtenir un analyseur avec autant d'états qu'un analyseur LR(0):

- La fusion peut engendrer des conflits *reduce*/*reduce* ⇒ moins général que LR(1)
- Mais moins de conflits que dans SLR(1).
- *Bison*/*Yacc* produit des analyseurs LALR(1), par défaut.

# Classe des grammaires (1 / 2)



# Classe des grammaires (2 / 2)




Dans la pratique, les analyseurs LL(1) et LALR(1):

- sont les plus utilisés;
- couvrent une bonne partie des grammaires de langages de programmation

# Bilan sur des méthodes ascendantes

Les analyseurs de type LR sont intéressants:

- Ils peuvent reconnaître à peu près toutes les grammaires des langages de programmation.
- Ils sont très efficaces pour implémenter une analyse *shift-reduce*
- La classe des grammaires LR est un sur-ensemble des grammaires LL.
- Ils permettent une bonne récupération des erreurs.

Par contre, il est très difficile de construire un analyseur à la main et on passera en général par un *générateur d'analyseurs*.

---