Les analyseurs Bottom-up

Pr ZEGOUR DJAMEL EDDINE
Ecole Supérieure d'Informatique (ESI)
www.zegour.uuuq.com
email: d_zegour@esi.dz

Les analyseurs Bottom-up

Comment fonctionnent les analyseurs Bottom-up

Grammaires LR

Génération de la table LR

Compaction de la table LR

Traitement de la sémantique

Traitement des erreurs LR

Comment fonctionne un analyseur Bottom-up

Exemple

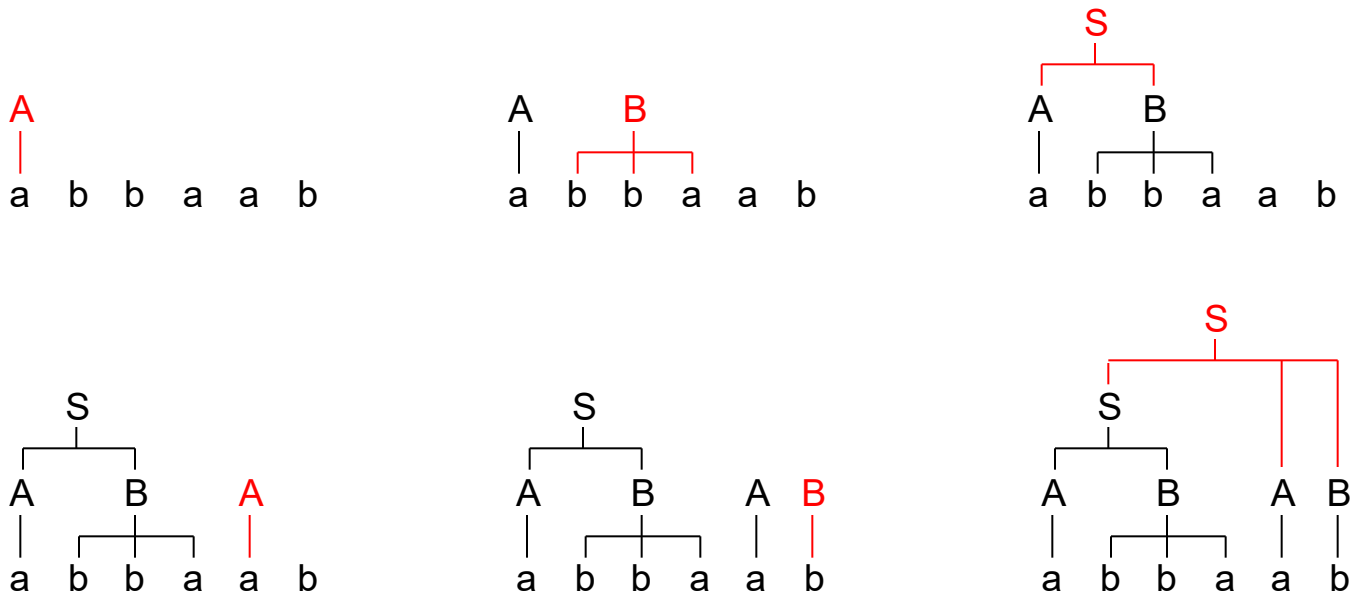
S = A B | S A B.
A = a | a a b.
B = b | b b a.

non LL(1)!

- Ne peut pas être utilisée pour un analyseur top-down
- Sans problème pour un analyseur bottom-up

Entrée: a b b a a b

L'arbre syntaxique est construit de bas en haut (bottom-up)



Utilisation d'une pile pour l'analyse

grammaire

S = A B | S A B.
A = a | a a b.
B = b | b b a.

Entrée

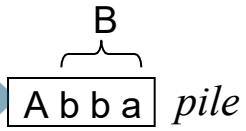
a b b a a b # (# : Marque eof)

Pile	*Entrée*	*Action*
	a b b a a b #	Décaler (cad. Lire la prochaine unité lexicale et l'Empiler)
a	b b a a b #	Réduire *a* to *A*
A	b b a a b #	Décaler
A b	b a a b #	Décaler (ne pas réduire, car ça conduirait vers une impasse!)
A b b	a a b #	Décaler
A b b a	a b #	Réduire *b b a* to *B*
A B	a b #	Réduire *A B* to *S*
S	a b #	Décaler
S a	b #	Réduire *a* to *A*
S A	b #	Décaler
S A b	#	Réduire *b* to *B*
S A B	#	Réduire *S A B* to *S*
S	#	phrase reconnue (la pile contient l'axiome; l'entrée est vide)

Utilisation d'une pile pour l'analyse (cont.)

Que doit connaître l'analyseur ?

- Est-ce qu'en sommet de pile il y a des unités qui peuvent être réduites
- À quel NTS elles devraient être réduites?
- Est-ce que l'analyseur devrait Décaler ou Réduire pour ne pas arriver à une impasse?



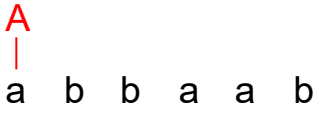
Quatre actions pour l'analyseur

Décaler	lire et empiler la prochaine unité (TS)
Réduire	réduire la fin de la pile à un NTS
accepter	phrase reconnue (seulement si S . #)
erreur	aucune des actions précédentes n'est possible (→ traitement de l'erreur)

Les analyseurs Bottom-up sont donc appelés :

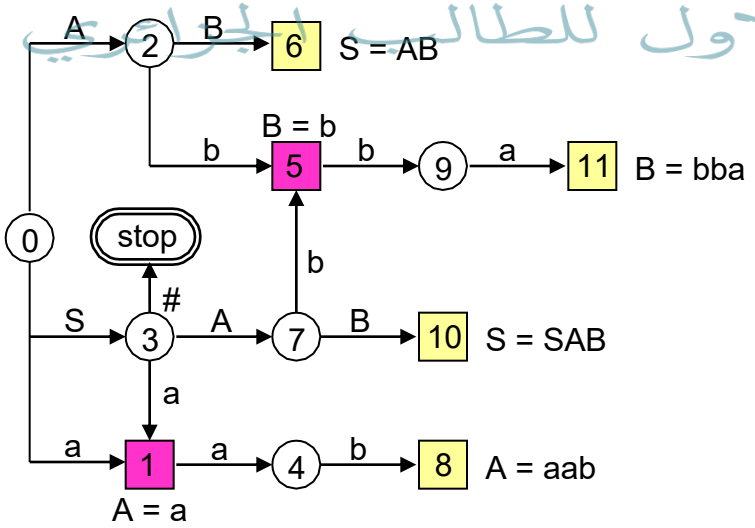
- **Analyseurs Décaler-Réduire**
- **Analyseurs LR**
  elles reconnaissent les phrases de la gauche (**L**eft) vers la droite utilisant les dérivations canoniques droites(**R**ight)

  Dérivation canonique droite = Réduction canonique gauche
  c.a.d la phrase simple la plus à gauche (*handle*) est réduite



L'analyseur comme un PDA(Push-Down Automaton)

Push-down Automaton



état de réduction

état de décalage / réduction

État de décalage

Grammaire

1 S = A B.
2 S = S A B.
3 A = a.
4 A = a a b
5 B = b
6 B = b b a

L'analyseur avec une table d'analyse (LR)

Table de transition (table d'analyse)

	TS			NTS		
	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

s1 ... Avancer vers l'état 1
r3 ... Réduire selon la production 3
- ... Erreur

Grammaire

1 S = A B.
2 S = S A B.
3 A = a.
4 A = a a b
5 B = b
6 B = b b a

Analyse avec des actions Décaler-Réduire

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

1 S = A B.
2 S = S A B.
3 A = a.
4 A = a a b
5 B = b
6 B = b b a

Entrée: a b b a a b #

Pile		*Entrée*	*Action*
0		a b b a a b #	s1
0 1		b b a a b #	r3 (A=a)
0	A	b b a a b #	s2
0 2		b b a a b #	s5
0 2 5		b a a b #	s9
0 2 5 9		a a b #	s11
0 2 5 9 11		a b #	r6 (B=bba)
0 2	B	a b #	s6
0 2 6		a b #	r1 (S=AB)
0	S	a b #	s3
0 3		a b #	s1
0 3 1		b #	r3 (A=a)
0 3	A	b #	s7
0 3 7		b #	s5
0 3 7 5		#	r5 (B=b)
0 3 7	B	#	s10
0 3 7 10		#	r2 (S=SAB)
0	S	#	s3
0 3		#	acc

r3 ... Réduire selon la production 3 (A = a)

- Dépiler 1 état (car le RHS (right-hand side) a une longueur 1)
- Insérer le LHS(left-hand side) (*A*) dans l'entrée
- Réduire est toujours suivie par Décaler avec un NTS

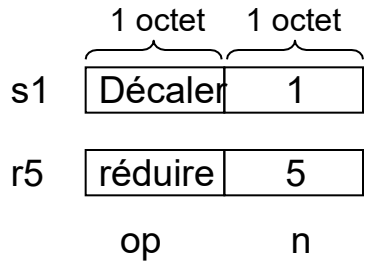
Implémentation d'un analyseur LR

```
void Parse () {
   short[,]   action = { {...}, {...}, ...};   // state transition table
   byte[]     length = { ... };                // production lengths
   byte[]     leftSide = { ... };              // left side NTS of every production
   byte       state;                           // current state
   int        sym;                             // next input symbol
   int        op, n, a;

   ClearStack();
   state = 0; sym = Next();
   for (;;) {
      Push(state);
      a = action[state, sym]; op = a / 256; n = a % 256;
      switch (op) {
         case shift:       // shift n
            state = n; sym = Next(); break;
         case reduce:      // reduce n
            for (int i = 0; i < length[n]; i++) Pop();
            a = action[Top(), leftSide[n]]; n = a % 256;  // shift n
            state = n;
            break;
         case accept: return;
         case error: throw new Exception();  // error handling is still missing
      }
   }
}
```

Analyseur dirigé par une table pour des grammaires quelconques

Entrée de action (2 octets)



Les analyseurs Bottom-up

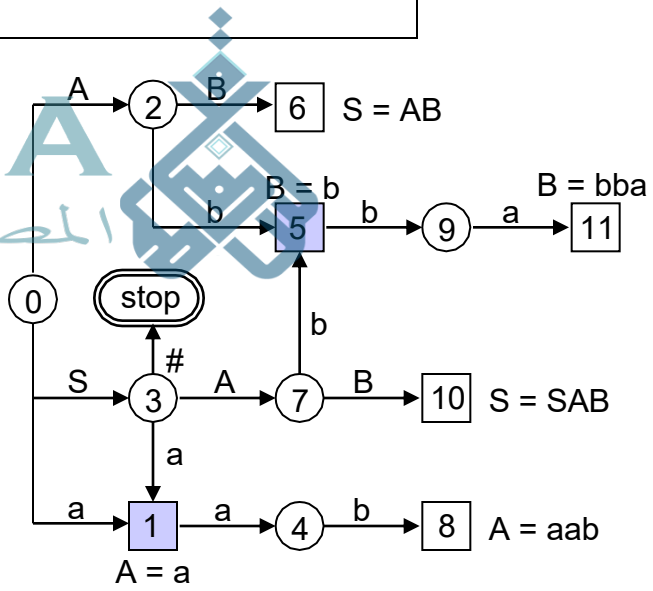
- Comment fonctionnent les analyseurs Bottom-up
- Grammaires LR
- Génération de la table LR
- Compaction de la table LR
- Traitement de la sémantique
- Traitement des erreurs LR

Grammaires LR(0) et LR(1)

LR(0) Analyse de gauche(**L**eft)à droite
Avec des dérivations canoniques droite(**R**ight)
Et **0** unité d'entrée

Une grammaire est LR(0), si

- Il n' y a pas d'état de réduction qui a aussi une action Décaler
- Dans chaque état de réduction il est seulement possible de réduire selon une production unique



Cette grammaire n'est pas LR(0)!

LR(1) Analyse de gauche(**L**eft)à droite
Avec des dérivations canoniques droite(**R**ight)
et **1** unité d'entrée

Une grammaire est LR(1), si dans chaque état une seule unité d'entrée (lookahead) est suffisante pour décider :

- Si on doit faire Décaler ou Réduire
- A quelle production on doit réduire

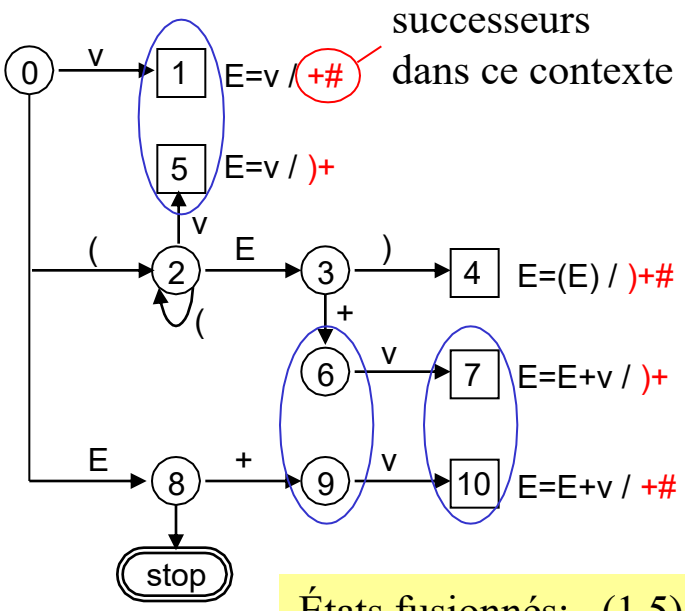
Grammaires LALR(1)

LookAhead LR(1)

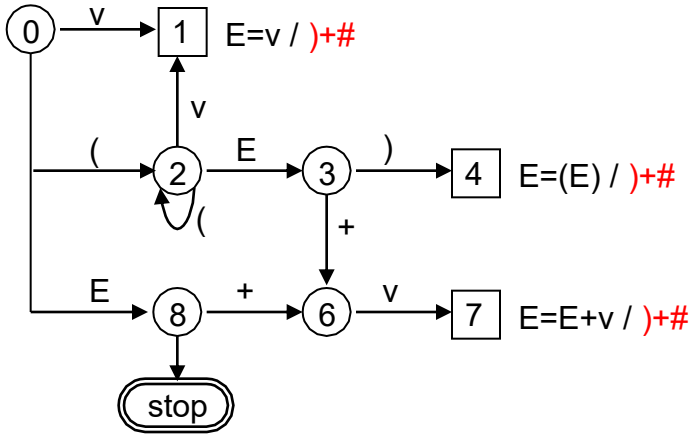
- Sous ensemble des grammaires LR(1)
- Ont des tables plus petites que celle des grammaires LR(1), car les états avec les mêmes actions mais des lookahead différents peuvent être fusionnés.

Exemple E = v | E "+" v | "(" E ")"

LR(1) tables



LALR(1) tables



États fusionnés: (1,5) (6,9) (7,10)

Analyseurs LR et RD (Descente Récursive)

Avantages

- LALR(1) est plus puissante que LL(1)
  - autorise les productions récursives gauches
  - autorise les productions d'un non terminal avec les mêmes symboles début (First)
- Les analyseurs LR sont plus compactes que les analyseurs 'descente récursive' (mais les tables occupent beaucoup d'espace)
- Les analyseurs dirigés par les tables (Table-driven parsers) sont des algorithmes universels qui peuvent être paramétrés avec des tables
- Ils (Table-driven parsers) permettent un meilleur traitement des erreurs.

Inconvénients

- Les tables LALR(1) sont difficiles à construire pour les grandes grammaires (utilisation d'outils)
- Les analyseurs LR sont légèrement plus lents que les analyseurs RD
- Le traitement de la sémantique est plus compliqué pour les analyseurs LR
- Plus difficile de faire les traces dans les analyseurs LR

Les analyseurs LR sont attractifs pour les langages complexes.
Pour les langages simples (Ex. langages de commandes) les analyseurs 'descente récursive' sont meilleurs.

Les analyseurs Bottom-up

Comment fonctionnent les analyseurs Bottom-up

Grammaires LR

Génération de la table LR

Compaction de la table LR

Traitement de la sémantique

Traitement des erreurs LR

Les éléments LR(0)

Exemple

S = a A b
A = c

L'analyseur (dénoté par un point) **se déplace à travers les productions**

S = . a A b
S = a . A b ← Si l'analyseur est devant *A*, il est aussi au début de la production A
A = . c
A = c .
S = a A . b
S = a A b .

Élément LR(0)

Situation de l'analyseur

X = α . β

- β : Partie non traitée de la production
- . : Position de l'analyseur
- α : Partie traitée de la production

Symbole de contrôle

C'est le symbole après le point, Ex.:

A = a . a b symbole de contrôle = a

Élément de décalage	X = α . β	le point n'est pas à la fin de la production
Élément de réduction	X = α .	le point est en fin de la production

Les éléments LR(1)

Élément LR(1)

Situation de l'analyseur

X = α . β / a

- successeur de *X* dans ce contexte
- Partie non traitée de la production
- Position de l'analyseur
- Partie traitée de la production

Symbole de contrôle

C'est le symbole après le point, Ex.:

A = a . a b / c symbole de contrôle = a
A = a a b . / c symbole de contrôle = c

Élément de décalage X = α . β / *a* le point n'est pas à la fin de la production
Élément de réduction X = α . / *a* le point est en fin de la production

État de l'analyseur comme un ensemble d'éléments

État de l'analyseur

Représenté par un ensemble d'éléments sur lequel l'analyseur fonctionne à un moment donné

```
S = A . B c    / #
B = . b        / c
B = . b b a    / c
```

Les analyseurs Top-down travaillent toujours sur une seule production.
Les analyseurs Bottom-up peuvent travailler sur plusieurs *productions en parallèle*

```
X = . a a b          X = a . a b          X = a a . b
X = . a a c  Décaler a  X = a . a c  Décaler a  X = a a . c
```

Seulement à ce point l'analyseur a à décider en faveur de l'une des deux productions (en se basant sur L'unité d'entrée)

Les analyseurs Bottom-up sont donc plus puissants que les analyseurs top-down

Kernel et fermeture d'un état

Kernel

Ce sont les éléments d'un état qui ne commencent pas par un point (sauf dans la production de l'axiome)

S = A . B c / #

Tous les autres éléments d'un état peuvent être dérivés à partir de son kernel.

Fermeture

faire {
  si (un état a un élément de la forme X = α . Y β / T)
    ajouter tous les éléments Y = . ω / First(βs) à l'état tels que
    Y = ω est une production et s dans T
} tant que (de nouveaux éléments sont ajoutés);

X, Y : non terminaux

α, β : chaîne de terminaux et non terminaux éventuellement vides

T : ensemble de terminaux

Exemple

S = A . B c	/ #
B = . b	/ c
B = . b b a	/ c

kernel

fermeture

S = A B c
A = a
A = a a b
B = b
B = b b a

Grammaire

Kernel et fermeture d'un état (Cas LR0)

Kernel

Ce sont les éléments d'un état qui ne commencent pas par un point (sauf dans la production de l'axiome)

S = A . B c

Tous les autres éléments d'un état peuvent être dérivés à partir de son kernel.

Fermeture

faire {
  si (un état a un élément de la forme X = α . Y β)
    ajouter tous les éléments Y = . ω à l'état tels que
    Y = ω est une production
} tant que (de nouveaux éléments sont ajoutés);

X, Y : non terminaux

α, β : chaîne de terminaux et non terminaux éventuellement vides

Exemple

S = A . B c
B = . b
B = . b b a

kernel } fermeture

S = A B c
A = a
A = a a b
B = b
B = b b a

Grammaire

Exemple: calcul de fermeture

Grammaire

0 S' = S #
1 S = A B
2 S = S A B
3 A = a
4 A = a a b
5 B = b
6 B = b b a

Kernel

S' = . S / #

Ajouter toutes les productions S

S' = . S / #
S = . A B / #
S = . S A B / #

Ajouter toutes les productions A

S' = . S #
S = . A B / #
S = . S A B / #
A = . a / b
A = . a a b / b

Ajouter toutes les productions de S (à cause de S = . S A B / #)
Successeurs de S = First(A) = $\{a\}$

S' = . S #
S = . A B / #
S = . S A B / #
A = . a / b
A = . a a b / b
S = . A B / a
S = . S A B / a

→

S' = . S #
S = . A B / #a
S = . S A B / #a
A = . a / b
A = . a a b / b

} fermeture

État successeur

Goto(s, sym)

C'est l'état que nous obtenons à partir de l'état *s* en faisant *décaler sym* (*sym* = symbole de contrôle)

Exemple

état *i*:	S = A . B c	/ #
	B = . b	/ c
	B = . b b a	/ c
Goto(*i*, *b*):	B = b .	/ c
	B = b . b a	/ c
Goto(*i*, *B*):	S = A B . c	/ #

- Contient tous les éléments de *i* qui ont b comme leur symbole de contrôle
- le point a été déplacé à travers b
- ce sont les productions sur lesquelles on travaille à cet instant

Goto(s, sym) est seulement le kernel du nouvel état;
nous devons calculer de nouveau sa fermeture

État successeur

Fonction Goto(I,X)
// I un état = ensemble d'éléments LR
// X un symbole terminal ou non terminal

Soit J l'ensemble des éléments [A → αX . β /a] tels que [A → α . X β /a] est dans I
Retourner Fermeture(J)

Cas LR0

Fonction Goto(I,X)
// I un état = ensemble d'éléments LR
// X un symbole terminal ou non terminal

Soit J l'ensemble des éléments [A → αX . β] tels que [A → α . X β] est dans I
Retourner Fermeture(J)

Génération de la table LR(1)

Étendre la grammaire par une pseudo production *S'* = *S* #
Créer l'état 0 avec le kernel S' = . S # // le seul kernel avec le point au début
Tant que (tous les états ne sont pas visités) {
  s = prochain état non visité;
  calculer la fermeture de s;
  pour (tous les éléments de s) {
    selon le type de l'élément {
      cas : X = S . # : générer accepté;
      cas : X = α . y β / T : créer le nouvel état s1 = Goto(s, y)
                              générer décaler y, s1;
      cas : X = α . / T : générer réduire t , (X = α); pour tout t dans T
      Sinon: générer erreur;
    }
  }
}

LR(1) peut générer des états identiques avec des ensembles T différents

Génération de la table LALR(1)

Étendre la grammaire par une pseudo production *S'* = *S* #
Créer l'état 0 avec le kernel S' = . S # // le seul kernel avec le point au début

```
Tant que (tous les états ne sont pas visités) {
   s = prochain état non visité;
   calculer la fermeture de s;
   pour (tous les éléments de s) {
      selon le type de l'élément {
         cas : X = S . # :          générer accepté;
         cas : X = α . y β / T :    créer le nouvel état s1 = Goto(s, y) s'il n'existe pas encore;
                                    générer décaler y, s1;
         cas : X = α . / T :        générer réduire t , (X = α); pour tout t dans T
         Sinon:                     générer erreur;
      }
   }
}
```

Seuls les kernels sont considérés pour vérifier l'existence des états sans les symboles successeurs, Ex.:

existe		nouveau		
E = v . / #+	+	E = v . /)+	➔	E = v . /)#+

Génération de la table LALR(1)

Grammaire

0 S' = S #
1 S = A B
2 S = S A B
3 A = a
4 A = a a b
5 B = b
6 B = b b a

0 : S' = . S # Créer l'état 0 avec le kernel S'=.S #

Déterminer sa fermeture

S'= . S / #
S = . A B / #a
S = . S A B / #a
A = . a / b
A = . a a b / b

Pour l'élément (S'= . S /#) créer l'état 3 =goto(0, S) =
S' = S . / #
S = S . A B / #a
A = . a / b
A = . a a b / b
Et générer Shift S 3

Pour l'élément (S= . AB /#a) créer l'état 2 =goto(0, A) =
S = A . B / #a
B = . b / #a
B = . b b a / #a
Et générer Shift A 2

Pour l'élément (A= . a /#b) créer l'état 1 =goto(0, a) =
A = a . / b
A = a . a b / b
Et générer Shift a 1

Génération de la table LALR(1)

Grammaire

0	S' = S #
1	S = A B
2	S = S A B
3	A = a
4	A = a a b
5	B = b
6	B = b b a

0	S' = . S #		shift	a	1
	S = . A B	/ #a	shift	A	2
	S = . S A B	/ #a	shift	S	3
	A = . a	/ b			
	A = . a a b	/ b			
1	A = a .	/ b	red	b	3
	A = a . a b	/ b	shift	a	4
2	S = A . B	/ #a	shift	b	5
	B = . b	/ #a	shift	B	6
	B = . b b a	/ #a			
3	S' = S . #		acc	#	
	S = S . A B	/ #a	shift	a	1 (!)
	A = . a	/ b	shift	A	7
	A = . a a b	/ b			
4	A = a a . b	/ b	shift	b	8
5	B = b .	/ #a	red	#,a	5
	B = b . b a	/ #a	shift	b	9
6	S = A B .	/ #a	red	#,a	1
7	S = S A . B	/ #a	shift	b	5
	B = . b	/ #a	shift	B	10
	B = . b b a	/ #a			
8	A = a a b .	/ b	red	b	4
9	B = b b . a	/ #a	shift	a	11
10	S = S A B .	/ #a	red	#,a	2
11	B = b b a .	/ #a	red	#,a	6

Table de l'analyseur

0	S' = . S #		shift	a	1
	S = . A B	/ #a	shift	A	2
	S = . S A B	/ #a	shift	S	3
	A = . a	/ b			
	A = . a a b	/ b			
1	A = a .	/ b	red	b	3
	A = a . a b	/ b	shift	a	4
2	S = A . B	/ #a	shift	b	5
	B = . b	/ #a	shift	B	6
	B = . b b a	/ #a			
3	S' = S . #		acc	#	
	S = S . A B	/ #a	shift	a	1
	A = . a	/ b	shift	A	7
	A = . a a b	/ b			
4	A = a a . b	/ b	shift	b	8
5	B = b .	/ #a	red	#,a	5
	B = b . b a	/ #a	shift	b	9
6	S = A B .	/ #a	red	#,a	1
7	S = S A . B	/ #a	shift	b	5
	B = . b	/ #a	shift	B	10
	B = . b b a	/ #a			
8	A = a a b .	/ b	red	b	4
9	B = b b . a	/ #a	shift	a	11
10	S = S A B .	/ #a	red	#,a	2
11	B = b b a .	/ #a	red	#,a	6

Table de transition des états
(table de l'analyseur)

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

Table de l'analyseur comme une liste

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

Les actions 'erreur' pour les NTS n'apparaissent jamais

- Les actions dans chaque état sont analysées séquentiell.
- La dernière action T dans chaque état est * *error* ou * *ri* (lookahead tokens ne sont pas vérifiés dans les actions *reduire* ; Si un '*reduiré*' est mauvais, ceci est détecté au prochain '*décaler*').
- Les listes sont plus compactes mais plus lentes

que les tables.

	TS actions		*NTS actions*	
0	a	s1	S	s3
	*	error	A	s2
1	a	s4		
	*	r3		
2	b	s5	B	s6
	*	error		
3	a	s1	A	s7
	#	acc		
	*	error		
4	b	s8		
	*	error		
5	b	s9		
	*	r5		
6	*	r1		
7	b	s5	B	s10
	*	error		
8	*	r4		
9	a	s11		
	*	error		
10	*	r2		
11	*	r6		

Génération de la table SLR(1)

Étendre la grammaire par une pseudo production *S'* = *S* #
Créer l'état 0 avec le kernel S' = . S # // le seul kernel avec le point au début
Tant que (tous les états ne sont pas visités) {
  s = prochain état non visité;
  calculer la fermeture de s;
  pour (tous les éléments de s) {
    selon le type de l'élément {
      cas : X = S . # : générer accepté;
      cas : X = α . y β : créer le nouvel état s1 = Goto(s, y)
                          générer décaler y, s1;
      cas : X = α . : générer réduire t (X = α) pour tout t dans Suivant(X);
      Sinon: générer erreur;
    }
  }
}

SLR(1) ne fait de réduction (X = α) que si le prochain symbole est dans Suivant(X)

Tables LALR(1) et Tables LR(1)

LR(1) est légèrement plus puissante que LALR(1), mais sa table est à peu près 10 fois plus grande

Tables LR(1)

- Les états ne sont jamais fusionnés
- La grammaire est LR(1) si aucun état a les conflits suivants:

Conflit décaler-réduire	shift a ... red a ...	L'analyseur ne peut décider avec 1 symbole d'entrée S'il doit faire 'décaler' ou 'réduire'
Conflit réduire-réduire	red a n red a m	L'analyseur ne peut décider avec 1 symbole d'entrée À quelle production il doit réduire

Tables LALR(1)

- Les états peuvent être fusionnés si leurs kernels (sans les symboles successeur) sont identiques

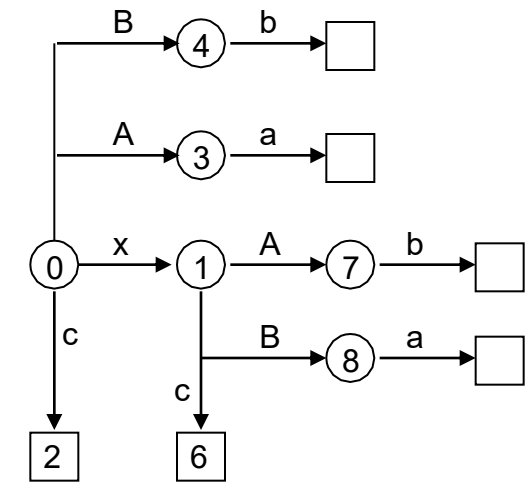
E = v . / #+ + E = v . /)+ ➔ E = v . /)#+

- La grammaire est LALR(1) si il n'y a pas de conflits réduire-réduire après fusion (les conflits décaler-réduire ne peuvent exister)

Exemple: Grammaire LR(1) qui n'est pas LALR(1)

Grammaire

S' = S #
S = x A b
S = x B a
S = A a
S = B b
A = c
B = c



A=c / a
B=c / b

A=c / b
B=c / a

0	S' = . S #		shift	x	1
	S = . x A b	/ #	shift	c	2
	S = . x B a	/ #	shift	A	3
	S = . A a	/ #	shift	B	4
	S = . B b	/ #	shift	S	5
	A = . c	/ a			
	B = . c	/ b			
1	S = x . A b	/ #	shift	c	6
	S = x . B a	/ #	shift	A	7
	A = . c	/ b	shift	B	8
	B = . c	/ a			
2	A = c .	/ a	red	a	(A = c)
	B = c .	/ b	red	b	(B = c)
...	...				
6	A = c .	/ b	red	b	(A = c)
	B = c .	/ a	red	a	(B = c)
...	...				

La fusion des états 2 et 6 conduirait à la situation suivante:

2	A = c .	/ ab	red	a,b	(A = c)
	B = c .	/ ab	red	a,b	(B = c)

Conflit réduire-réduire!

Tables SLR(1) (Simple LR(1))

- l'analyseur SLR(1) est moins puissant que l'analyseur LALR(1)
+ les tables SLR(1) sont plus simples à générer que les tables LALR(1)

Les éléments SLR(1)

Élément décaler	$X = \alpha . \beta$	⟵	Aucun successeur est considéré
Élément réduire	$X = \alpha . / \gamma$	⟵	γ = Suivant(X), cad. Tous les successeurs

Exemple: Grammaire LALR(1), qui n'est pas SLR(1)

Grammaire

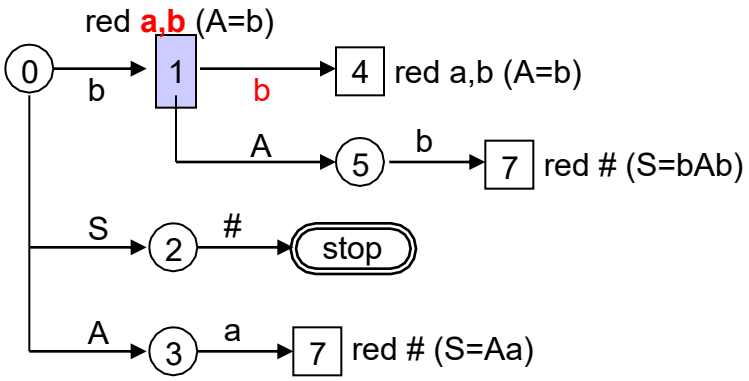
S' = S #
S = b A b
S = A a
A = b

0	S' = . S #	shift	b	1
	S = . b A b	shift	S	2
	S = . A a	shift	A	3
	A = . b			
1	S = b . A b	shift	b	4
	A = b .	red	a,b	(A = b)
	A = . b	shift	A	5

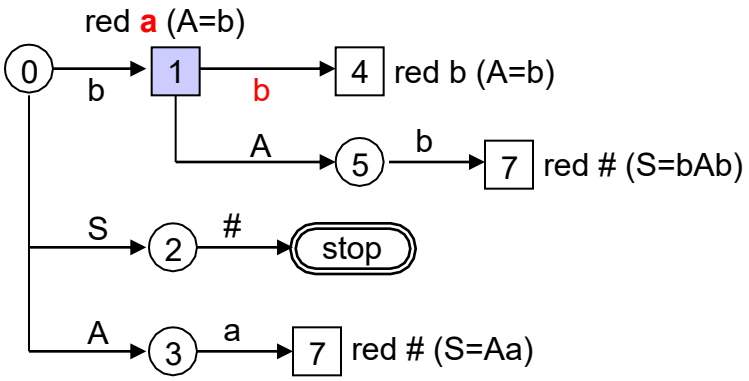
Conflit décaler-réduire, Qui ne peut avoir lieu dans les tables LALR(1)

Suivant(S) = {#}
Suivant(A) = {a, b}

SLR(1)



LALR(1)



Les analyseurs Bottom-up

- Comment fonctionnent les analyseurs Bottom-up
- Grammaires LR
- Génération de la table LR
- Compaction de la table LR
- Traitement de la sémantique
- Traitement des erreurs LR

Taille de la table

Supposition (Ex: C#)

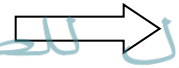
80 symboles terminaux
200 symboles non terminaux
2500 etats ➔ 280 x 2'500 = 700'000 entrées

4 octets / entrée ➔ 700'000 x 4 = 2'800'000 bytes = 2.8 MBytes

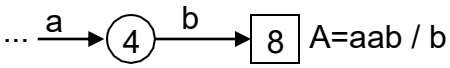
Mais la taille de la table peut être réduite par 90%

Combiner les actions 'décaler' et 'réduire'

	a	b	#
...	...	...	...
4	-	s8	-
...	...	...	...
8	-	r4	-
...	...	...	-



	a	b	#
...	...	...	...
4	-	sr4	-
...	...	...	...

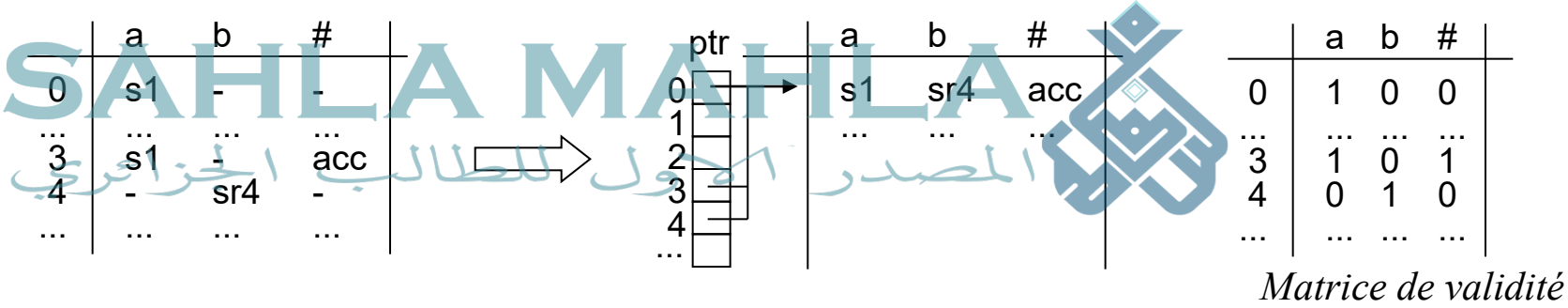


- Si une action 'décaler' conduit vers un état *s* dans lequel il n' y a que des actions '*réduire*' avec la même production , cette action 'décaler' peut être remplacée par '*décaler-réduire*'.
- L'état s peut être éliminé

Modification dans l'algorithme d'analyse

```
...
switch (op) {
   case shift:   // shift n
      state = n; sym = Next(); break;
   case shiftred:  // shiftred n
      for (int i = 0; i < length[n]-1; i++) Pop();
      a = action[Top(), leftSide[n]]; n = a % 256;  // shift n
      state = n;
      break;
   ...
```

Fusion des lignes



Matrice de validité

- Les lignes avec des actions non en conflit peuvent être fusionnées.
- Nécessite un tableau de pointeurs pour adresser la ligne correcte indirectement.
- Nécessite une matrice de bits qui définit les actions valides dans chaque état.

- Les actions sur T et NT devraient être fusionnées indépendamment (plus de chance pour la fusion).
- La même technique peut aussi être utilisée pour la fusion des colonnes
- D'autres compactage sont possibles, mais ça devient de plus en plus cher.

Modifications dans l'algorithme d'analyse

```
BitArray[] valid;
short[] ptr;
...
a = action[ptr[state], sym];
op = a / 256; n = a % 256;
if (!valid[state][sym]) op = error;
switch (op) {
    ...
```

Exemple

Table Originale (6 colonnes x 12 lignes x 2 bytes = **144** bytes)

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

Combiner ‘décaler’ et ‘réduire’ (6 colonnes x 8 lignes x 2 bytes = **96** bytes)

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	sr1
3	s1	-	acc	-	s7	-
4	-	sr4	-	-	-	-
5	r5	s9	r5	-	-	-
7	-	s5	-	-	-	sr2
9	sr6	-	-	-	-	-

Exemple (cont.)

Après avoir combiné les actions décaler et réduire (96 bytes)

	a	b	#	S	A	B
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	sr1
3	s1	-	acc	-	s7	-
4	-	sr4	-	-	-	-
5	r5	s9	r5	-	-	-
7	-	s5	-	-	-	sr2
9	sr6	-	-	-	-	-

Table des pointeurs

(deux tables)

Fusion des lignes (3 colonnes * 6 lignes * 2 bytes
+ 2 * 8 lignes * 2 bytes + 8 * 1 byte = 36 + 32 + 8 = **76** bytes)

T actions

	a	b	#
0,3,4	s1	sr4	acc
1	s4	r3	-
2,7,9	sr6	s5	-
5	r5	s9	r5

NT actions

	S	A	B
0,2	s3	s2	sr1
3,7	-	s7	sr2

matrice

	a	b	#
0	1	0	0
1	1	1	0
2	0	1	0
3	1	0	1
4	0	1	0
5	1	1	1
7	0	1	0
9	1	0	0

Les analyseurs Bottom-up

- Comment fonctionnent les analyseurs Bottom-up
- Grammaires LR
- Génération de la table LR
- Compaction de la table LR
- Traitement de la sémantique
- Traitement des erreurs LR

Actions sémantiques

Sont permises seulement en fin de production (cad au moment des réductions)

Les actions sémantiques au milieu d'une production

```
X = a (. ... .) b.
```

doivent être implémentées comme suit:

```
X = a Y b.
Y = (. ... .).
```
← Production vide; quand c'est réduit l'action sémantique est exécutée

Problème: ceci peut détruire la propriété LALR(1)

```
X = a b c.
X = a (. ... .) b d.
```

⇒

```
X = a b c.
X = a Y b d.
Y = (. ... .).
```

Génération de la table

i	X = a . b c	/ #	shift	b	i+1
	X = a . Y b d	/ #	shift	Y	i+2
	Y = .	/ b	red	b	(Y=)

Conflit décaler-réduire

Raison: l'analyseur ne peut travailler sur les deux productions en parallèle, mais doit décider sur l'une des deux productions (Cad. S'il exécute l'action sémantique ou non).

Attributs

- Chaque symbole a *un* attribut de sortie (éventuellement un objet avec plusieurs champs)
- Quand un symbole est reconnu son attribut est empilé dans la *pile sémantique*

Exemple

Expr✸x = Term✸x.

Expr✸x = Expr✸x "+" Term✸y **(. push(pop() + pop(); .)** // x = x + y;

Term✸x = Factor✸x.

Term✸x = Term✸x "*" Factor✸y **(. push(pop() * pop(); .)** // x = x * y;

Factor✸x = const✸t **(. t = pop(); push(t.val); .)**

Factor✸x = "(" Expr✸x ")".

Modifications dans l'analyseur

Table de productions

	leftSide	length	sem
0	Expr	1	0
1	Expr	3	1
2	Term	1	0
3	Term	3	2
4	Factor	1	3
5	Factor	3	0

Exemple précédent

Évaluateur sémantique

```
void SemAction (int n) {
    switch (n) {
        case 1: Push(Pop() + Pop()); break;
        case 2: Push(Pop() * Pop()); break;
        case 3: Token t = Pop(); Push(t.val); break;
    }
}
```

Les variables apparaissant dans plusieurs actions Sémantiques doivent être déclarées globales (problème avec Les NTS récursifs)

Analyseur

```
...
switch(op) {
    ...
    case reduce:  // red n
        if (sem[n] != 0) SemAction(sem[n]);
        for (int i = 0; i < length[n]; i++) Pop();
        ...
}
```

Les analyseurs Bottom-up

- Comment fonctionnent les analyseurs Bottom-up
- Grammaires LR
- Génération de la table LR
- Compaction de la table LR
- Traitement de la sémantique
- Traitement des erreurs LR

Idée

Situation quand une erreur apparaît

$s_0 \dots s_n \; . \; t_0 \dots t_m \; \#$

pile entrée

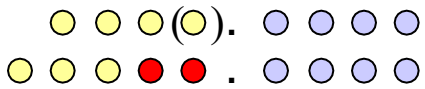
but: synchroniser la pile et l'entrée afin que nous ayons:

$s_0 \dots s_i \; . \; t_j \dots t_m \; \#$

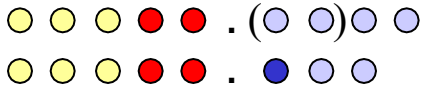
Et t_j est accepté dans l'état s_i

Stratégie

- Empiler ou dépiler des états



- Insérer ou supprimer des unités au début de l'entrée



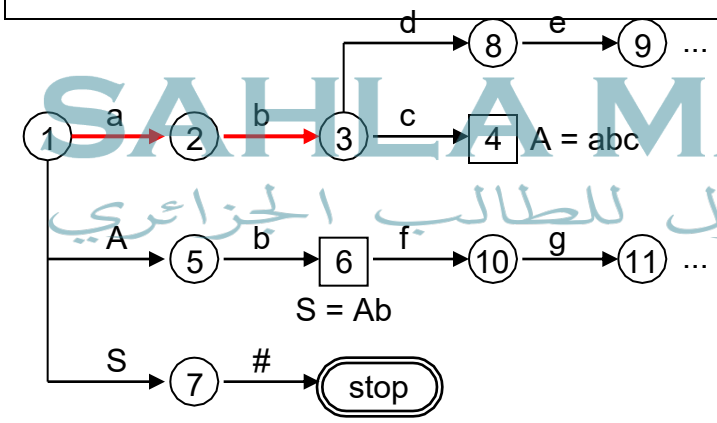
Algorithme

1. **Rechercher un chemin de sortie**
   - Remplacer t_0 .. t_m avec une entrée virtuelle v_0 .. V_k (guides) , qui conduit l'analyseur de l'état d'erreur s_n vers un état final aussi rapidement que possible.
   - Durant l'analyse de v_0 .. v_k collecter toutes les unités qui sont valides dans les états traversés → ancres

2. **Supprimer les unités invalides**
   - Ignorer des unités de l'entrée t_0 .. t_m, jusqu'à ce que une unité t_j apparaît qui est un ancre.

3. **Insérer des unités manquantes**
   - Conduire l'analyseur de l'état s_n avec v_0 .. v_k jusqu'à ce qu'il obtient un état s_i dans lequel t_j est accepté
   - Insérer toutes les unités virtuelles v_0 .. v_k dans l'entrée devant t_j.

Exemple



input: a b b #

1. Rechercher un chemin de sortie

Entrée virtuelle: c b #

ancres:
- (3) c, d
- (5) b
- (6) f
- (7) #

2. Supprimer les unités invalides

Pas de saut car
le prochain symbole d'entrée *b*
est déjà un ancre

3. Insérer les unités manquantes

insérer *c*, pour appliquer [4] et (5)

En (5) l'analyseur peut continuer avec *b*.

Entrée corrigée

a b c b #

Messages d'erreurs

Reporter seulement ce qui a été inséré ou supprimé

- Si les unités *a*, *b*, *c* sont supprimées de l'entrée:

  ligne ... col ... : "a b c" supprimées

- Si les unités *x, y* sont insérées dans l'entrée:

  ligne ... col ... : "x y" insérées

- Si les unités *a*, *b*, *c* sont supprimées et les unités *x*, *y* sont insérées:

  ligne ... col ... : "a b c" remplacées par "x y"

Un autre exemple (avec simulation de l'analyseur)

grammaire

0 S' = S #
1 S = A B
2 S = S A B
3 A = a
4 A = a a b
5 B = b
6 B = b b a

	a	b	#	S	A	B	guide
0	s1	-	-	s3	s2	-	**a**
1	s4	r3	-	-	-	-	**b**
2	-	s5	-	-	-	s6	**b**
3	s1	-	acc	-	s7	-	**#**
4	-	s8	-	-	-	-	**b**
5	r5	s9	r5	-	-	-	**a**
6	r1	-	r1	-	-	-	**a**
7	-	s5	-	-	-	s10	**b**
8	-	r4	-	-	-	-	**b**
9	s11	-	-	-	-	-	**a**
10	r2	-	r2	-	-	-	**a**
11	r6	-	r6	-	-	-	**a**

Entrée erronée

a a a b #

Chaque état a un " symbole guide ". (expliqué par la suite Comment il est déterminé).

Début de l'analyse

pile	*entrée*	*action*
0	a a a b #	s1
0 1	a a b #	s4
0 1 4	a b #	-- erreur!

Rechercher un chemin de sortie et collecter les anc

pile	*guide*	*action*	*ancre*
0 1 4	b	s8	b
0 1 4 8	b	r4	b
0 2	b	s5	b
0 2 5	a	r5	a, b, #
0 2 6	a	r1	a, #
0 3	#	acc	a, #

ancre = {*a*, *b*, #}

Exemple (cont.)

grammaire

0 S' = S #
1 S = A B
2 S = S A B
3 A = a
4 A = a a b
5 B = b
6 B = b b a

	a	b	#	S	A	B	guide
0	s1	-	-	s3	s2	-	**a**
1	s4	r3	-	-	-	-	**b**
2	-	s5	-	-	-	s6	**b**
3	s1	-	acc	-	s7	-	**#**
4	-	s8	-	-	-	-	**b**
5	r5	s9	r5	-	-	-	**a**
6	r1	-	r1	-	-	-	**a**
7	-	s5	-	-	-	s10	**b**
8	-	r4	-	-	-	-	**b**
9	s11	-	-	-	-	-	**a**
10	r2	-	r2	-	-	-	**a**
11	r6	-	r6	-	-	-	**a**

Entrée restante

a b #

ancres

{a, b, #}

Saut à l'entrée

Aucun symbole n'est sauté car le prochain symbole *a*
est déjà dans l'ensemble des ancres {*a*, *b*, #}

Insérer les symboles manquants

Pile	*guide*	*action*	*inséré*	
0 1 4	b	s8	b	← Seul 'décaler' permet l'insertion d'un symbole, pas '*réduire*'
0 1 4 8	b	r4		
0 2	b	s5	b	
0 2 5				

A ce niveau,on peut continuer avec *a*

Exemple (cont.)

grammaire

0 S' = S #
1 S = A B
2 S = S A B
3 A = a
4 A = a a b
5 B = b
6 B = b b a

	a	b	#	S	A	B	guide
0	s1	-	-	s3	s2	-	**a**
1	s4	r3	-	-	-	-	**b**
2	-	s5	-	-	-	s6	**b**
3	s1	-	acc	-	s7	-	**#**
4	-	s8	-	-	-	-	**b**
5	r5	s9	r5	-	-	-	**a**
6	r1	-	r1	-	-	-	**a**
7	-	s5	-	-	-	s10	**b**
8	-	r4	-	-	-	-	**b**
9	s11	-	-	-	-	-	**a**
10	r2	-	r2	-	-	-	**a**
11	r6	-	r6	-	-	-	**a**

Entrée restante

a b #

L'analyse se poursuit

pile	*input*	*action*
0 2 5	a b #	r5
0 2 6	a b #	r1
0 3	a b #	s1
0 3 1	b #	r3
0 3 7	b #	s5
0 3 7 5	#	r5
0 3 7 10	#	r2
0 3	#	acc

Entrée corrigée

a a b b a b #

Message d'erreur

ligne ... col ...: "b b" insérés

Détermination des symboles de guide

1. Ordonner les productions de telle sorte que la première production de chaque NTS est non Récursive et aussi courte que possible

```
S = A B.
S = S A B.   ← La production  récursive devient la seconde
A = a.
A = a a b.   ← La production la plus longue devient la seconde
B = b.
B = b b a.   ← La production la plus longue devient la seconde
```

2. Pendant la construction de la table LALR(1) :
   La première action T générée dans chaque état détermine le symbole de guide pour cet état.

```
0   S' = . S #              shift   a    1   ← Le symbole de guide est a
    S = . A B     / #a      shift   A    2
    S = . S A B   / #a      shift   S    3
    A = . a       / b
    A = . a a b   / b
```

Estimation de cette technique de traitement des erreurs

- Ne ralentit pas l'analyse syntaxique
- Se termine toujours (# est toujours un ancre)
- Ne détecte pas uniquement les erreurs mais les corrige aussi (mais la correction n'est pas toujours ce qu'on veut).
- Génère de bons messages d'erreurs