Analyse Syntaxique

CS410 - Langages et Compilation

Julien Henry
Catherine Oriat

Grenoble-INP Esisar

2012-2013

Summary

Rappel : Analyse Sytaxique

- En *entrée* : une suite de mots renvoyée par l'analyseur lexical.
- En *sortie* : un arbre abstrait représentant le programme.

Objectif : Reconnaître que la suite de mot appartient au langage, et construire en même temps l'abre abstrait du programme.

Problème de la reconnaissance

La principale difficulté de l'analyse syntaxique est de faire le lien entre le programme (la suite de mots) et la grammaire hors contexte du langage.

Quelles règles de dérivation de la grammaire ont été appliquées pour obtenir cette suite de mots ?

Phrase reconnue

Une phrase m est reconnue si il existe un arbre de dérivation syntaxique :

- Les noeuds internes représentent des symboles non-terminaux.
- Les noeuds externes représentent des symboles terminaux.
- La racine de l'arbre est S.
- Dans un parcours infixe, les feuilles forment le mot m.
- Si un noeud interne étiqueté X possède les sous arbres de racine $X_1, \cdots X_n$, alors $X \rightarrow X_1 \cdots X_n \in R$.

Solution générale

Le problème de la reconnaissance est décidable pour les langages hors contexte : c'est à dire que pour tout langage hors contexte $L(G)$, il existe un algorithme qui détermine si $x \in L(G)$.
Algorithme :

- On part de l'axiome.
- On construit l'arbre des possibilités en utilisant à chaque fois l'ensemble des règles de grammaire possibles.

Si on trouve la chaîne à reconnaître $\rightarrow$ succès. Sinon, il faut un critère d'arrêt pour décider de l'échec.

Solution générale

On manipule des couples (m, α), avec :

- $m \in (V_T \cup V_N)^*$, un modèle de phrase.
- $\alpha \in V_T^*$, la phrase que l'on veut reconnaître.

On veut déterminer si $m \rightarrow^* \alpha$.

Initialement, on part de (S, α), avec α la chaîne à reconnaître. On peut appliquer les opérations suivantes :

- **Effacement** : $(xm, x\alpha) \longrightarrow (m, \alpha)$.
- **Expansion** : $(Am, \alpha) \longrightarrow (\beta m, \alpha)$ pour toutes les règles de la forme $A \rightarrow \beta$.

Exemple

Soit la grammaire G suivante :

$$\begin{aligned} S &\rightarrow AB \\ B &\rightarrow a \\ A &\rightarrow aA \\ A &\rightarrow Bb \end{aligned}$$

Le mot $aba \in L(G)$?

Autre solution

On utilise des grammaires qui appartiennent à des sous classes des langages hors contexte :

- Grammaires LL : grammaires dont on peut faire une analyse *descendante* déterministe
- Grammaires LR : grammaires dont on peut faire une analyse *ascendante* déterministe

Ici, déterministe veut dire : il existe un algorithme de complexité linéaire par rapport au nombre de mots.

Analyse descendante

Déjà vu en parlant des grammaires LL(1).

Exemple :
Soit la grammaire G suivante :

$$\begin{aligned} S &\rightarrow AB \\ B &\rightarrow a \\ A &\rightarrow aA \\ A &\rightarrow Bb \end{aligned}$$

Le mot $aba \in L(G)$?

Summary

Analyse ascendante

L'analyse ascendante est la méthode d'analyse utilisée dans les "vrais" outils.
Principe :

- On part de la chaîne à reconnaître.
- On essaie de reconnaître des parties droites de règles jusqu'à obtenir l'axiome.

Analyse ascendante

- L'analyse construit l'arbre de dérivation syntaxique en partant des feuilles (on part de la chaîne à reconnaître)
- On garde en mémoire dans une **pile** une liste de non-terminaux et de terminaux correspondant à la portion d'arbre à reconstruire.
- Deux opérations :
  - lecture / shift : on fait passer le terminal du mot à lire sur la pile.
  - réduction / reduce : on reconnaît sur la partie droite $x_1 \cdots x_n$ d'une règle $X \to x_1 \cdots x_n$ et on la remplace par X.
- Le mot est reconnu si on termine avec le symbole S sur la pile et le mot vide à lire.

Exemple d'analyse ascendante

Soit la grammaire :

$$
\begin{array}{lcl}
S & \rightarrow & E \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & id
\end{array}
$$

On fait une analyse ascendante de la phrase $id + id * id$.

Action	Pile	Phrase
	ε	id + id*id
shift	id	+ id * id
reduce	E	+ id * id
shift	$E+$	id * id
shift	$E + id$	* id
reduce	$E + E$	* id
shift	$E + E*$	id
shift	$E + E * id$	ε
reduce	$E + E * E$	ε
reduce	$E + E$	ε
reduce	E	ε
reduce	S	ε

Conflits

Lors de l'analyse, on peut avoir deux types de conflits :

- Conflits **reduce/reduce** : plusieurs réductions sont possibles.
  Exemple :
  $S \rightarrow fAcd|faBce, A \rightarrow ab, B \rightarrow b$ avec fab en pile, réduire ab à A ou b à B ?
- Conflits **shift/reduce** : on peut choisir entre une lecture ou une réduction.
  Exemple :
  $S \rightarrow fAbc|fBce, A \rightarrow a, B \rightarrow ab$ avec fa en pile, réduire a à A ou lire b pour ensuite réduire ab à B ?

Grammaires LR(0)

Définition: (Grammaire LR(0))

Une grammaire est LR(0) si elle n'a aucun conflit reduce/reduce ou shift/reduce.

LR(0) : décider de l'action

Pour décider si on doit lire ou réduire :

- On utilise un automate.
- Chaque état de la pile est un état de l'automate : un état correspond aux parties droites de règles qui peuvent être reconnues.
- En fonction de l'état de l'automate dans lequel on est et du symbole à lire :
  - si lecture : le nouvel état dépend de l'état courant et du caractère lu.
  - si réduction par $X \rightarrow x_1 x_2 \cdots x_n$: le nouvel état dépend de l'état depuis lequel on a lu x_1.

Exemple

On considère la grammaire suivante :

$$
\begin{array}{lcl}
S & \rightarrow & E\# \\
E & \rightarrow & E + T \\
E & \rightarrow & T \\
T & \rightarrow & id \\
T & \rightarrow & (E)
\end{array}
$$

Automate LR(0)

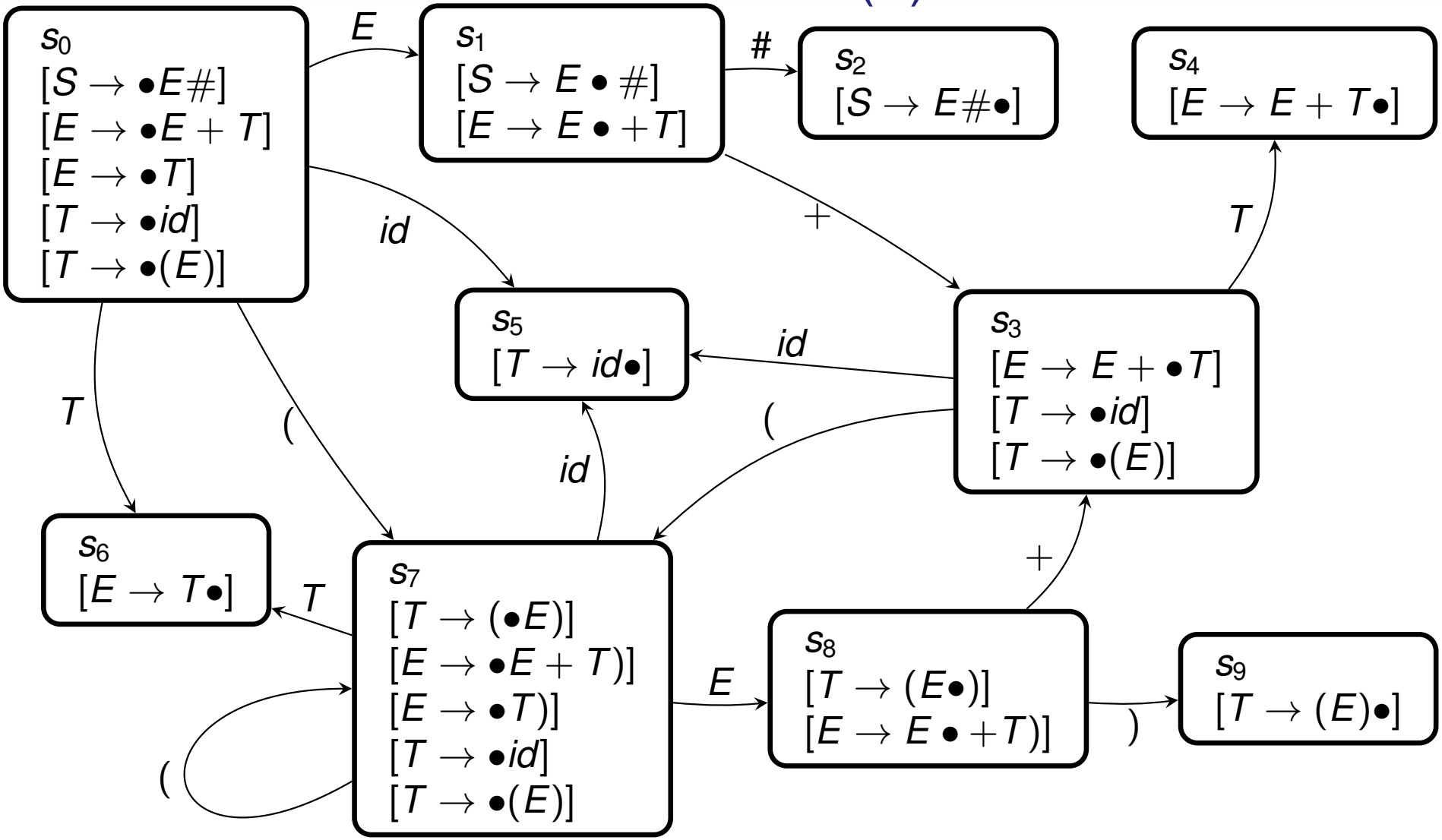


Table de l'automate LR(0)

Etats	Action	table de saut						
		id	+	(	)	#	E	T
s_0	shift	s_5		s_7			s_1	s_6
s_1	shift		s_3			s_2		
s_2	reduce $[S \rightarrow E\#]$							
s_3	shift	s_5		s_7				s_4
s_4	reduce $[E \rightarrow E + T]$							
s_5	reduce $[T \rightarrow id]$							
s_6	reduce $[E \rightarrow T]$							
s_7	shift	s_5		s_7			s_8	s_6
s_8	shift		s_3		s_9			
s_9	reduce $[T \rightarrow (E)]$							

Exemple

Action	Pile	Phrase
	s_0	id + (id + id) #
shift	$s_0.id.s_5$	+ (id + id) #
reduce	$s_0.T.s_6$	+ (id + id) #
reduce	$s_0.E.s_1$	+ (id + id) #
shift	$s_0.E.s_1.+.s_3$	(id + id) #
shift	$s_0.E.s_1.+.s_3.(.s_7$	id + id) #
shift	$s_0.E.s_1.+.s_3.(.s_7.id.s_5$	+ id) #
reduce	$s_0.E.s_1.+.s_3.(.s_7.T.s_6$	+ id) #
reduce	$s_0.E.s_1.+.s_3.(.s_7.E.s_8$	+ id) #
shift	$s_0.E.s_1.+.s_3.(.s_7.E.s_8.+.s_3$	id) #
shift	$s_0.E.s_1.+.s_3.(.s_7.E.s_8.+.s_3.id.s_5$	) #
reduce	$s_0.E.s_1.+.s_3.(.s_7.E.s_8.+.s_3.T.s_4$	) #
reduce	$s_0.E.s_1.+.s_3.(.s_7.E.s_8$	) #
shift	$s_0.E.s_1.+.s_3.(.s_7.E.s_8.).s_9$	#
reduce	$s_0.E.s_1.+.s_3.Ts_4$	#
reduce	$s_0.E.s_1$	#
shift	$s_0.E.s_1.\#.s_2$	
reduce	ACCEPT	

Conflits LR(0)

Si un état contient des items décidant d'actions différentes, on a un conflit LR(0) :

- conflit shift/reduce : si l'état de l'automate possède à la fois un item $[A \rightarrow \alpha\bullet]$ et un item $[B \rightarrow \beta \bullet a\gamma]$.
- conflit reduce/reduce : si l'état de l'automate possède à la fois un item $[A \rightarrow \alpha\bullet]$ et un item $[B \rightarrow \beta\bullet]$ ($A \neq B$ ou $\alpha \neq \beta$).

L'outil Cup

On comprend mieux les messages d'erreurs de Cup, notamment en cas de grammaire ambigüe :

```
[java] Warning : *** Shift/Reduce conflict found
[java] in state #60
[java] between expr ::= expr MOINS expr (*)
[java] and expr ::= expr (*) MOINS expr
[java] under symbol MOINS
[java] Resolved in favor of shifting.
```

Analyse SLR(1)

Prenons maintenant la grammaire

$$\begin{array}{lcl} E & \rightarrow & E + T | T \\ T & \rightarrow & T * F | F \\ F & \rightarrow & i \end{array}$$

- il est facile de voir que les items $[E \rightarrow \bullet T]$ et $[T \rightarrow \bullet T * F]$ de s_0, après transition sur T, donneront un état contenant $[E \rightarrow T \bullet]$ et $[T \rightarrow T \bullet * F]$: conflit décalage-réduction !
- on peut voir que les suivants de E sont $+$ et $\#$, mais pas $*$
- on peut donc décider le décalage sur $*$ et la réduction sur les suivants de E
- on obtient un automate SLR(1), soit simple LR(1).
  - $[A \rightarrow \alpha \bullet]$ décide une réduction uniquement sur les suivants de A
  - $[A \rightarrow \alpha \bullet a\beta]$ décide un décalage uniquement sur a

Automate SLR(1)

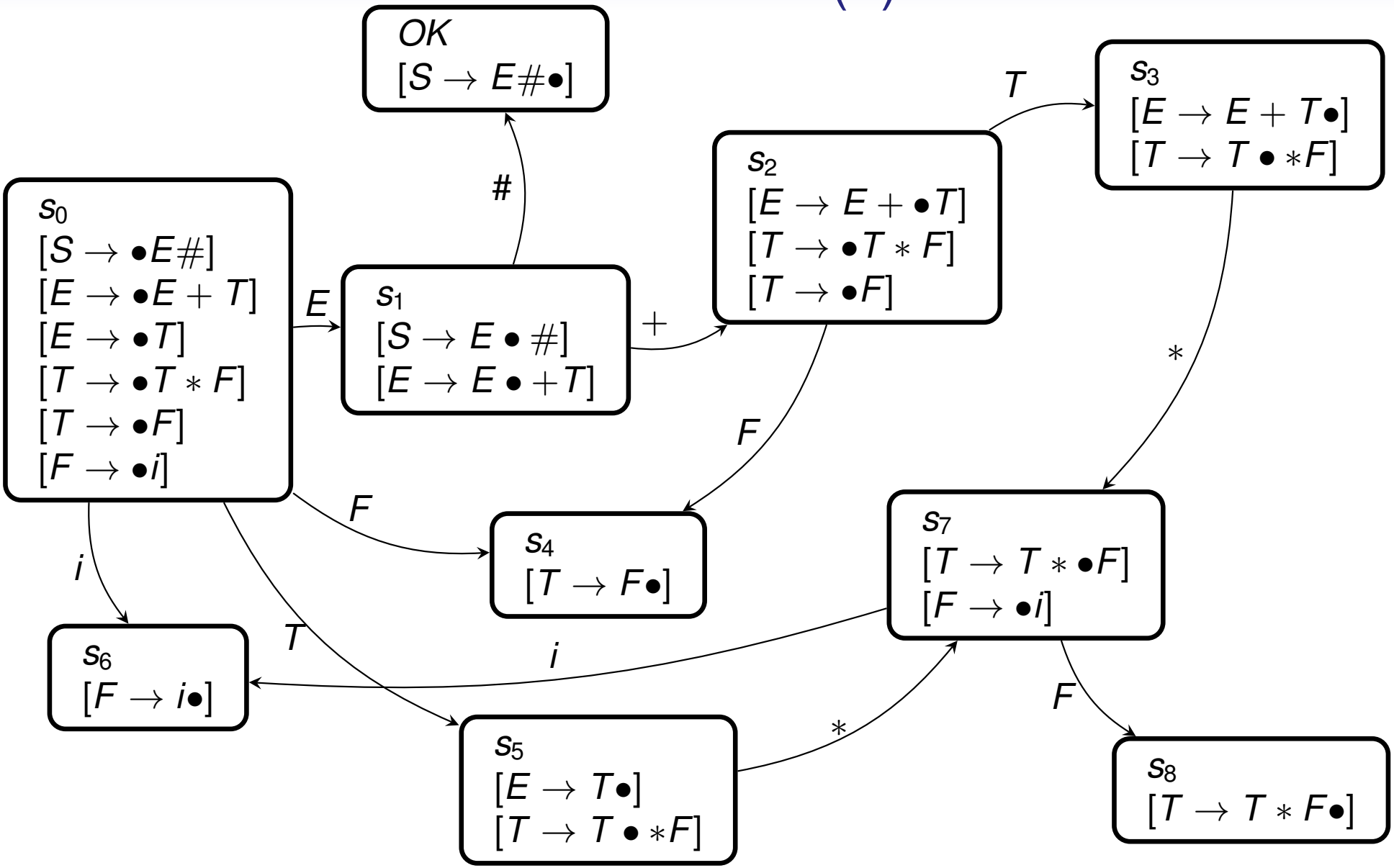
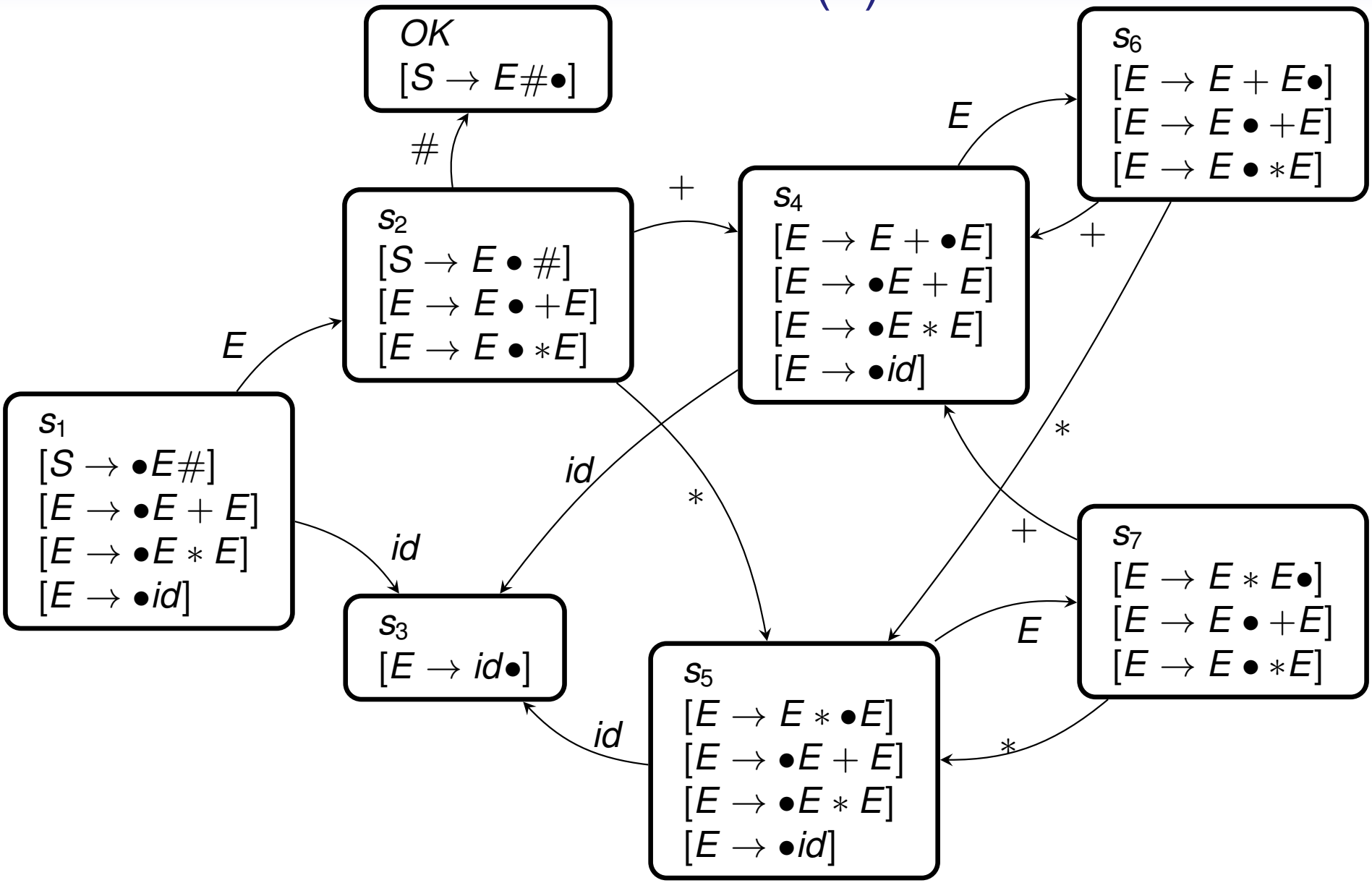


Table SLR(1)

Etats	table des actions						
	i	+	*	#	F	E	T
s_0	s_6				s_4	s_1	s_5
s_1		s_2		OK			
s_2					s_4		s_3
s_3		$[E \to E + T]$	s_7	$[E \to E + T]$			
s_4	$[T \to F]$	$[T \to F]$	$[T \to F]$	$[T \to F]$			
s_5		$[E \to T]$	s_7	$[E \to T]$			
s_6	$[F \to i]$	$[F \to i]$	$[F \to i]$	$[F \to i]$			
s_7	s_6				s_8		
s_8	$[T \to T * F]$	$[T \to T * F]$	$[T \to T * F]$	$[T \to T * F]$			
OK	$[S \to E\#]$	$[S \to E\#]$	$[S \to E\#]$	$[S \to E\#]$			

Automate SLR(1) ?



Règles de précédences

Parfois, on peut des règles de précédence pour savoir lequel a la priorité parmi un *shift* ou un *reduce*.

Exemple

Action	Pile	Phrase
	s_1	id + id*id #
shift	$s_1.id.s_3$	+ id * id #
reduce	$s_1.E.s_2$	+ id * id #
shift	$s_1.E.s_2.+.s_4$	id * id #
shift	$s_1.E.s_2.+.s_4.id.s_3$	* id #
reduce	$s_1.E.s_2.+.s_4.E.s_6$	* id #
shift	$s_1.E.s_2.+.s_4.E.s_6.*.s_5$	id #
shift	$s_1.E.s_2.+.s_4.E.s_6.*.s_5.id.s_3$	#
reduce	$s_1.E.s_2.+.s_4.E.s_6.*.s_5.E.s_7$	#
reduce	$s_1.E.s_2.+.s_4.E.s_5$	#
reduce	$s_1.E.s_2$	#
shift	$s_1.E.s_2.\#$	ε
reduce	S	ε

L'outil Cup

En cas de problème, Cup peut choisir automatiquement les règles de précédence.

```
[java] Warning : *** Shift/Reduce conflict found
[java] in state #60
[java] between expr ::= expr MOINS expr (*)
[java] and expr ::= expr (*) MOINS expr
[java] under symbol MOINS
[java] Resolved in favor of shifting.
```

Analyse descendante / Analyse ascendante

Avantages et inconvénients des deux techniques d'analyse syntaxique :

- L'analyse ascendante
  - est plus adaptée à l'analyse des langages de programmation.
  - permet de reconnaître de façon déterministe une plus grande classe de langages.
- L'analyse descendante
  - est utilisée lorsqu'on écrit un analyseur à la main (plus simple).
  - est bien adaptée au rattrapage d'erreurs.

Summary

Analyseur syntaxique LL(1)

On peut facilement écrire un analyseur syntaxique pour une grammaire LL(1).

- On écrit une méthode par symbole non-terminal
- Ces méthodes font une action différente selon le premier lexème lu, et s'appellent mutuellement.
- L'analyse syntaxique débute en appelant la méthode associée au symbole initial S.

Type Lexeme

```
public enum Code_lex {Begin_lex, Affect, Num, ...}

public class Lexeme {
        // code du lexeme
        Code_lex code;
        // chaine correspondant a l'unite lexicale
        String chaine;
        // numeros de ligne et colonne du lexeme
        int num_ligne, num_colonne;
}
```

Méthode correspondant à un non-terminal A

Si la grammaire du langage a les règles $A \rightarrow \alpha_1|\alpha_2|\cdots|\alpha_n$, avec les $\alpha_i \in (V_T \cup V_N)^*$:

```
Lexeme LC; //lexeme courant
public void RecA() {
   switch (LC.code) {
      case <éléments de Directeur(A → α1)>: analyser(α1);
      case <éléments de Directeur(A → α2)>: analyser(α2);
      ...
      case <éléments de Directeur(A → αn)>: analyser(αn);
      default : Erreur de syntaxe;
   }
}
```

Grammaire LL(1) $\Rightarrow$ les éléments directeurs sont disjoints, donc le switch est correct.

Fonction *analyser*

On code la fonction *analyser* de la façon suivante :

```
analyser(ε) ≡ ;

analyser(X1X2...Xn) ≡ analyser(X1); ...;analyser(Xn);

analyser(X ∈ VT) ≡
          if (LC.code == X)
            Lex_suiv;
          else Erreur de syntaxe;

analyser(X ∈ VN) ≡ RecX();
```

Enfin, l'analyse syntaxique se lance en appelant `RecS`.

Exemple

On considère la grammaire des expressions arithmétiques :

$$\begin{array}{lcl} exp & \rightarrow & terme|exp + exp|exp - exp \\ terme & \rightarrow & terme * terme|Facteur \\ Facteur & \rightarrow & idf|num|(exp) \end{array}$$

Dans un premier temps, on transforme la grammaire pour qu'elle soit LL(1) :

$$\begin{array}{lcl} exp & \rightarrow & terme\ exp\_suite \\ exp\_suite & \rightarrow & \varepsilon| + exp| - exp \\ terme & \rightarrow & Facteur\ terme\_suite \\ terme\_suite & \rightarrow & \varepsilon| * terme \\ Facteur & \rightarrow & idf|num|(exp) \end{array}$$

Exemple : exp

exp $\rightarrow$ *terme exp_suite*

```
Lexeme LC; //lexeme courant
public void Rec_exp() {
        Rec_terme();
        Rec_exp_suite();
}
```

Exemple : exp_suite

$exp\_suite \rightarrow \varepsilon | + exp | - exp$

```
public void Rec_exp_suite() {
  switch (LC.code) {
    case PLUS_TOKEN :
      Lex_suiv();
      Rec_exp();
      break;
    case MOINS_TOKEN :
      Lex_suiv();
      Rec_exp();
      break;
    case EOF_TOKEN :
    case PARDROITE_TOKEN :
      break;
    default :
      Erreur_syntaxe();
      break;
  }
}
```

Exemple : terme

terme $\rightarrow$ *Facteur terme_suite*

```
public void Rec_terme() {
        Rec_Facteur();
        Rec_terme_suite();
}
```

Exemple : terme_suite

terme_suite $\rightarrow \varepsilon | * \textit{terme}$

```
public void Rec_terme_suite() {
  switch (LC.code) {
    case PARDROITE_TOKEN :
    case EOF_TOKEN :
    case PLUS_TOKEN :
    case MOINS_TOKEN :
      break;
    case MULT_TOKEN :
      Lex_suiv();
      Rec_terme();
      break;
    default :
      Erreur_syntaxe("MULT_TOKEN attendu");
      break;
  }
}
```

Exemple : Facteur

Facteur $\rightarrow$ *idf*|*num*|(*exp*)

```
public void Rec_Facteur() {
  switch (LC.code) {
    case IDF_TOKEN :
    case NUM_TOKEN :
      Lex_suiv();
      break;
    case PARGAUCHE_TOKEN:
      Lex_suiv();
      Rec_exp();
      if (LC.code == PARDROITE_TOKEN)
        Lex_suiv();
      else
        Erreur_syntaxe("Erreur : manque PARDROITE_TOKEN");
      break;
      default: Erreur_syntaxe("Erreur");
}}
```

Calcul d'attributs

Si, au lieu de travailler sur une grammaire LL(1), on travaille sur une grammaire LL(1) attribuée, il faut calculer les attributs pendant l'analyse.
Les attributs de la grammaires deviennent des paramètres des méthodes `Rec_`* :

- attributs *hérités* : deviennent des paramètres entrants de la méthode.
- attributs *synthétisés* : deviennent des paramètres sortants de la méthode.

Exemple

On veut calculer la valeur des expressions dans la grammaire.

$$
\begin{array}{lcl}
exp \uparrow^{val} & \rightarrow & terme \uparrow^{val_1} \ \ exp\_suite \downarrow_{val_1} \uparrow^{val} \\
exp\_suite \downarrow_{val} \uparrow^{val'} & \rightarrow & \varepsilon \mid + \ exp \uparrow^{val_1} \mid - \ exp \uparrow^{val_1} \\
terme \uparrow^{val} & \rightarrow & Facteur \uparrow^{val_1} \ \ terme\_suite \downarrow_{val_1} \uparrow^{val} \\
terme\_suite \downarrow_{val} \uparrow^{val'} & \rightarrow & \varepsilon \mid * \ terme \uparrow^{val_1} \\
Facteur \uparrow^{val} & \rightarrow & idf \mid num \mid (exp \uparrow^{val_1})
\end{array}
$$

Exemple : exp

$exp \uparrow^{val} \rightarrow terme \uparrow^{val_1} \; exp\_suite \downarrow_{val_1} \uparrow^{val}$

```
Lexeme LC; //lexeme courant
public void Rec_exp(Integer val) {
        Integer val1;
        Rec_terme(val1);
        Rec_exp_suite(val1,val);
}
```

Exemple : exp_suite

$exp\_suite \downarrow_{val} \uparrow^{val'} \rightarrow \varepsilon \mid + exp \uparrow^{val_1} \mid - exp \uparrow^{val_1}$

```
public void Rec_exp_suite(Integer val, Integer val2) {
  Integer val1;
  switch (LC.code) {
    case PLUS_TOKEN :
      Lex_suiv(); Rec_exp(val1);
      val2 = val + val1;
      break;
    case MOINS_TOKEN :
      Lex_suiv(); Rec_exp(val1);
      val2 = val - val1;
      break;
    case EOF_TOKEN :
    case PARDROITE_TOKEN :
      val2 = val;
      break;
    default : Erreur_syntaxe();
}}
```

Exemple : terme

$terme \uparrow^{val} \rightarrow Facteur \uparrow^{val_1} \; terme\_suite \downarrow_{val_1} \uparrow^{val}$

```
public void Rec_terme(Integer val) {
        Integer val1;
        Rec_Facteur(val1);
        Rec_terme_suite(val1,val);
}
```

Exemple : terme_suite

$terme\_suite \downarrow_{val} \uparrow^{val'} \rightarrow \varepsilon \mid * \; terme \uparrow^{val_1}$

```
public void Rec_terme_suite(Integer val,Integer val2) {
  Integer val1;
  switch (LC.code) {
    case PARDROITE_TOKEN :
    case EOF_TOKEN :
    case PLUS_TOKEN :
    case MOINS_TOKEN :
      val2 = val;
      break;
    case MULT_TOKEN :
      Lex_suiv();
      Rec_terme(val1);
      val2 = val * val1;
      break;
    default : Erreur_syntaxe("MULT_TOKEN attendu");
  }
}
```

Exemple : Facteur

$Facteur \uparrow^{val} \rightarrow idf | num | (exp \uparrow^{val_1})$

```
public void Rec_Facteur(Integer val) {
  switch (LC.code) {
    case IDF_TOKEN :
    case NUM_TOKEN :
      Lex_suiv();
          val = LC.valeur;
      break;
    case PARGAUCHE_TOKEN:
      Lex_suiv();
      Rec_exp(val);
      if (LC.code == PARDROITE_TOKEN)
        Lex_suiv();
      else
        Erreur_syntaxe("Erreur : manque PARDROITE_TOKEN");
      break;
      default: Erreur_syntaxe("Erreur");
}}
```

Construction d'un arbre abstrait

La construction d'un arbre abstrait se fait grâce à une grammaire attribuée.

- Les attributs sont de type `Noeud`, où `Noeud` représente un noeud de l'arbre.
- Les attributs `Noeud` sont synthétisés : on fait une construction de l'arbre depuis les feuilles.

Exemple

La grammaire des expressions arithmétiques peut être attribuée pour générer un arbre :

$$
\begin{array}{lcl}
exp \uparrow^{a} & \rightarrow & terme \uparrow^{a} \\
exp \uparrow^{Plus(a1,a2)} & \rightarrow & exp \uparrow^{a1} + exp \uparrow^{a2} \\
exp \uparrow^{Moins(a1,a2)} & \rightarrow & exp \uparrow^{a1} - exp \uparrow^{a2} \\
terme \uparrow^{Mult(a1,a2)} & \rightarrow & terme \uparrow^{a1} * terme \uparrow^{a2} \\
terme \uparrow^{a} & \rightarrow & Facteur \uparrow^{a} \\
Facteur \uparrow^{Idf(val)} & \rightarrow & idf \uparrow^{val} \\
Facteur \uparrow^{Num(val)} & \rightarrow & num \uparrow^{val} \\
Facteur \uparrow^{a} & \rightarrow & exp \uparrow^{a}
\end{array}
$$