Analyse syntaxique

Syntaxe et grammaires

Analyse descendante

Analyse ascendante

Traitement des erreurs

Bison

Syntaxe

Syntaxe

Contraintes sur l'écriture du code dans les langages de programmation

Sémantique

Interprétation du code

Règles de grammaire

Servent à spécifier la syntaxe

symbole --> expression

Dans l'expression on peut avoir deux sortes de symboles :

- ceux du langage final : les symboles **terminaux** (les mêmes qu'en analyse lexicale)
- des symboles intermédiaires, les **variables** ou **non-terminaux**

Exemple

Grammaire pour les expressions arithmétiques simples

E --> **nombre**

E --> **(** *E* **)**

E --> *E* + *E*

E --> *E* **-** *E*

E --> *E* ***** *E*

E --> *E* **/** *E*

Le non-terminal *E* désigne les expressions

Le symbole terminal **nombre** représente les chaînes de caractères qui sont des nombres

Les autres symboles terminaux sont **( ) + - * /**

Définition formelle

Une grammaire est définie par

- un alphabet A de symboles terminaux
- un ensemble V de non-terminaux
- un ensemble fini de règles

$(x, w) \in V \times (A|V)^*$

notées x --> w

- un non-terminal S appelé axiome

Un mot sur A est une suite d'éléments de A

A^* est l'ensemble des mots sur A

Un langage formel est une partie de A^*

Dérivations

Si x --> w est une règle de la grammaire, en **remplaçant *x*** **par *w*** dans un mot on obtient une dérivation :

$$E*(E) \dashrightarrow E*(E+E)$$

Enchaînement de dérivations

Si u_0 --> u_1 ... --> u_n on écrit :

$$u_0 \overset{*}{\dashrightarrow} u_n$$

Langage engendré

On s'intéresse aux dérivations qui vont de S à des mots de A^*

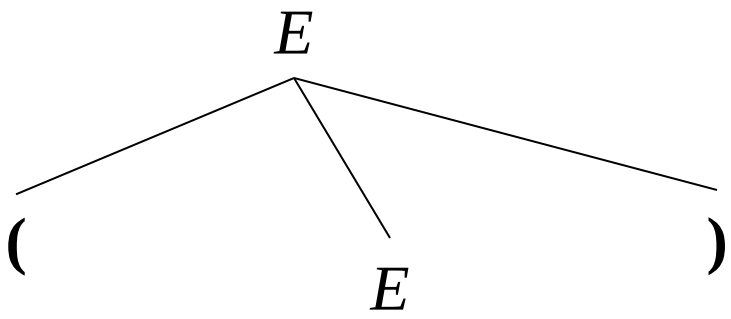
$$L = \{u \in A^* \mid S \overset{*}{\dashrightarrow} u\}$$

Exemple : $E \overset{*}{\dashrightarrow}$ **nombre * (nombre + nombre)**

Arbres

On représente les règles sous forme d'arbres

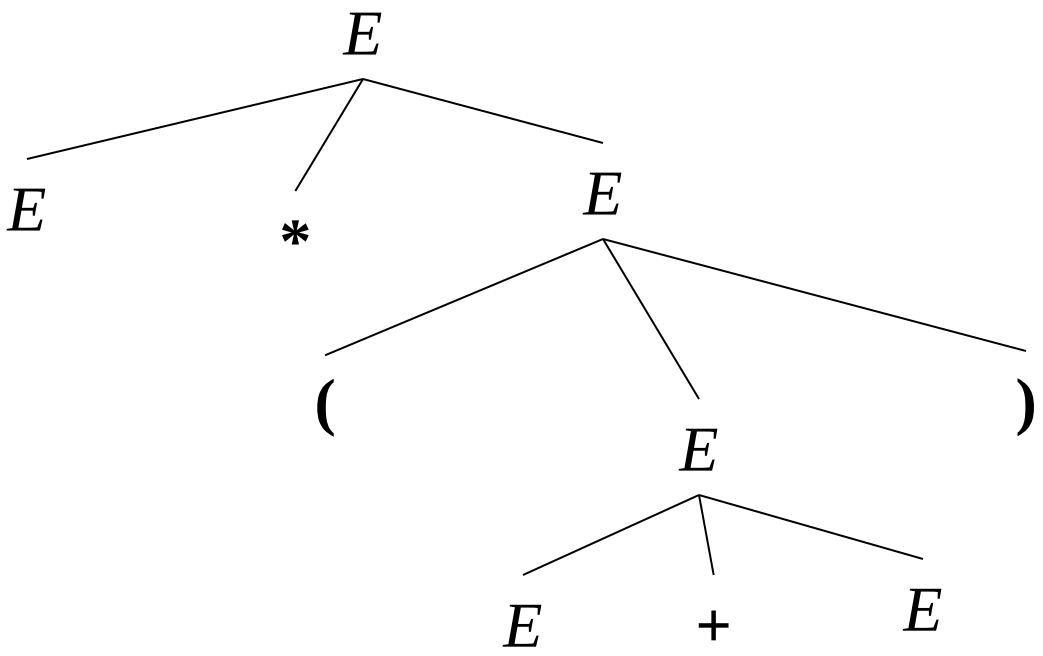
E --> **(** *E* **)**



Arbres

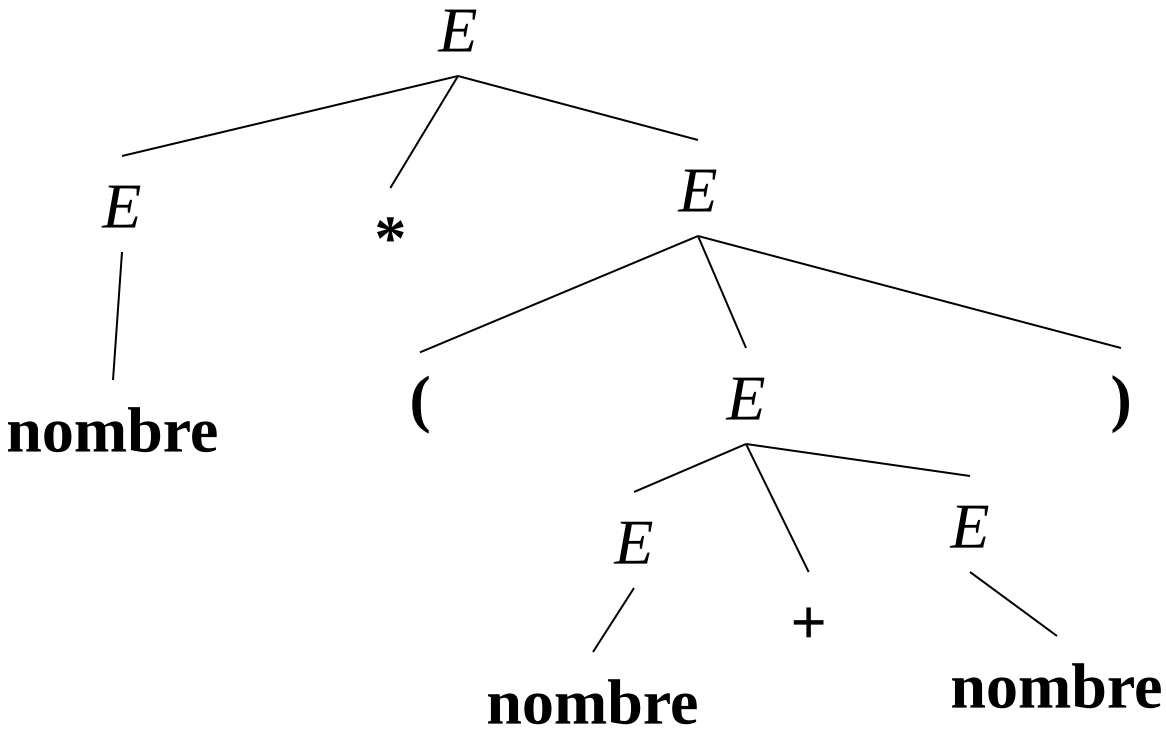
En enchaînant plusieurs dérivations, on a un arbre de hauteur supérieure à 1

E --> $E * E$ --> $E * (E)$ --> $E * (E + E)$



Arbres de dérivation

On s'intéresse aux arbres dont la racine est l'axiome et dont toutes les feuilles sont des symboles terminaux



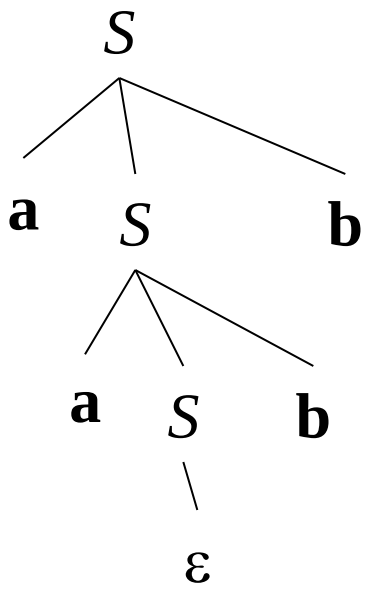
Le langage engendré par la grammaire est l'ensemble des frontières des arbres de dérivation

Arbres de dérivation

S --> **a** *S* **b**

S --> ε

Quels sont tous les arbres de dérivation pour cette grammaire ?



Les arbres de dérivation de hauteur $n > 0$ obtenus en utilisant $n - 1$ fois la première règle et 1 fois la deuxième

La frontière d'un tel arbre est $\mathbf{a}^{n-1}\mathbf{b}^{n-1}$

Langues naturelles

On peut utiliser les grammaires pour décrire la syntaxe des langues naturelles

<phrase> --> <sujet> <verbe>

<phrase> --> <sujet> <verbe> <complement>

<phrase> --> <sujet> <verbe> <complement> <complement>

<sujet> --> <det> <nom>

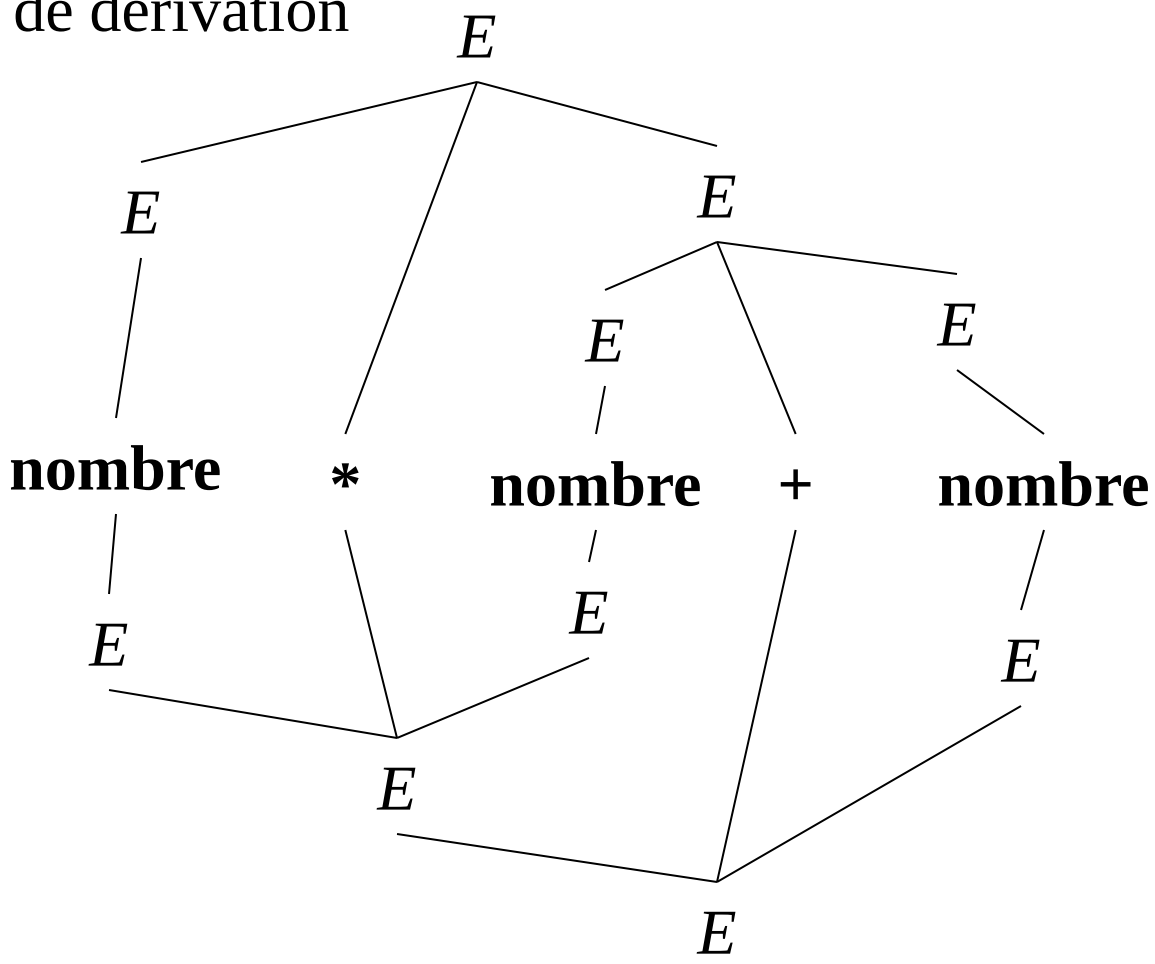
<sujet> --> <det> <adj> <nom>

...

Ce sont les grammaires de Chomsky.

Ambiguïté

Grammaire ambiguë : un même mot possède plusieurs arbres de dérivation



Ambiguïté

Exemple classique de grammaire ambiguë

inst --> **si** *cond* **alors** *inst*

inst --> **si** *cond* **alors** *inst* **sinon** *inst*

inst --> ...

cond --> ...

Exercice : quels sont les arbres pour

si *cond* **alors si** *cond* **alors** *inst* **sinon** *inst*

Conventions de notation

On regroupe plusieurs règles qui ont le même membre gauche

E --> *E* + *E*

E --> **N**

E --> *E* + *E* | **N**

On donne la liste des règles en commençant par l'axiome

inst --> **si** *cond* **alors** *inst*

| **si** *cond* **alors** *inst* **sinon** *inst*

| ...

cond --> ...

Exemples

E --> *E* + *E* | **N**

Grammaire ambiguë

P --> **(** *P* **)** *P* | ε

Grammaire des expressions parenthésées (non ambiguë)

E --> *E E* **+** | **N**

Expressions additives en notation postfixe (non ambiguë)

P --> **(** *P* **)** | ε

Compte des empilements et dépilements (non ambiguë)

Une grammaire non ambiguë pour les expressions

Cette grammaire force les choix suivants :

- associativité à gauche (c'est-à-dire de gauche à droite)
- priorité de * et / sur + et -.

Pour cela, elle utilise trois niveaux : *E*, *T*, *F*, au lieu d'un

- *F* (facteur) : expression qui n'est pas une opération
- *T* (terme) : fait de facteurs avec éventuellement * ou /
- *E* (expression) : fait de termes avec éventuellement + ou -

E --> *E* **+** *T* | *E* **-** *T* | *T*

T --> *T* ***** *F* | *T* **/** *F* | *F*

F --> **(** *E* **)** | **N**

Grammaires équivalentes

Deux grammaires sont équivalentes si elles engendrent le même langage

E --> E **+** T | E **-** T | T

T --> T ***** F | T **/** F | F

F --> **(** E **)** | **N**

E --> $T\ E'$

E' --> **+** $T\ E'$ | **-** $T\ E'$ | ε

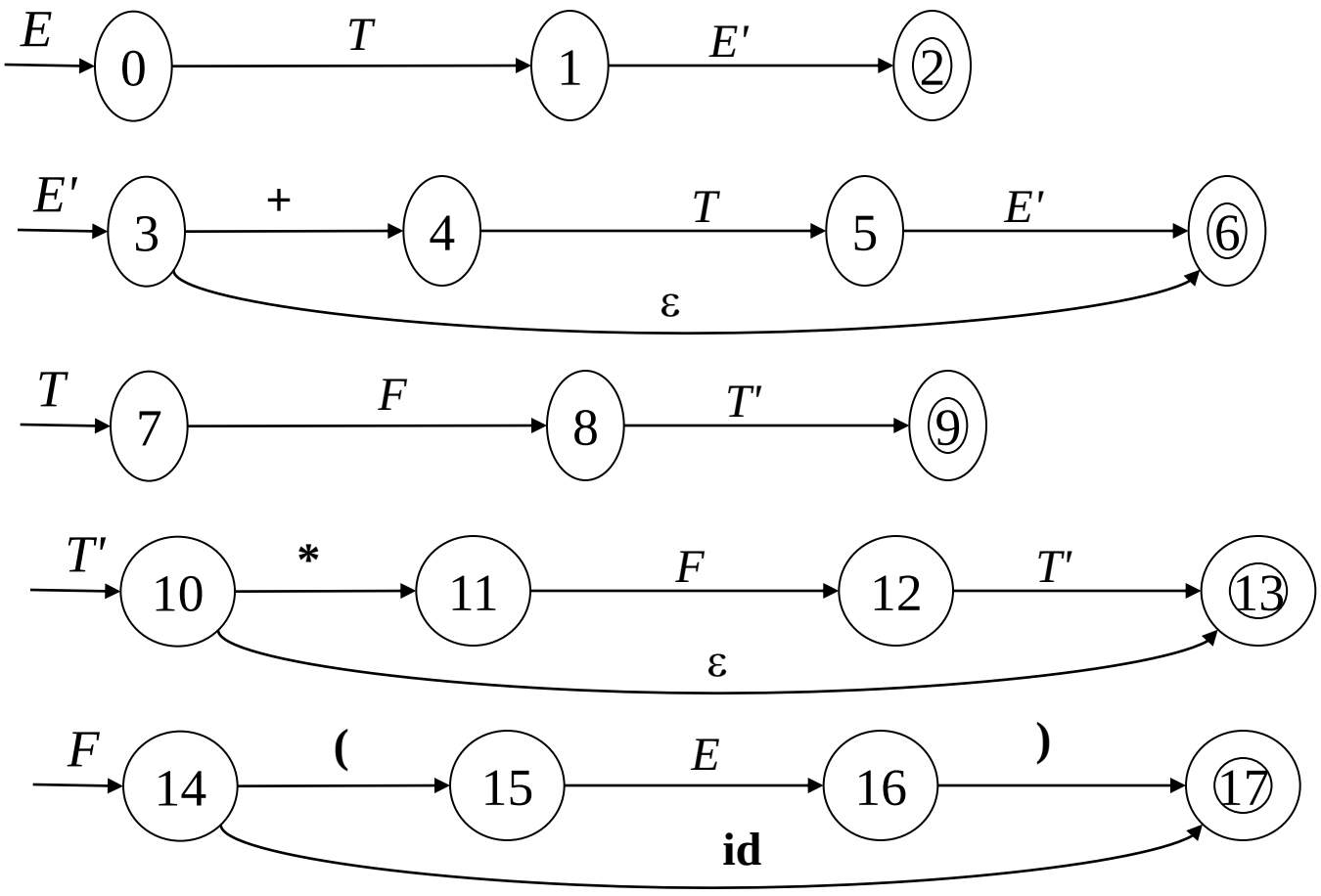
T --> $F\ T'$

T' --> ***** $F\ T'$ | **/** $F\ T'$ | ε

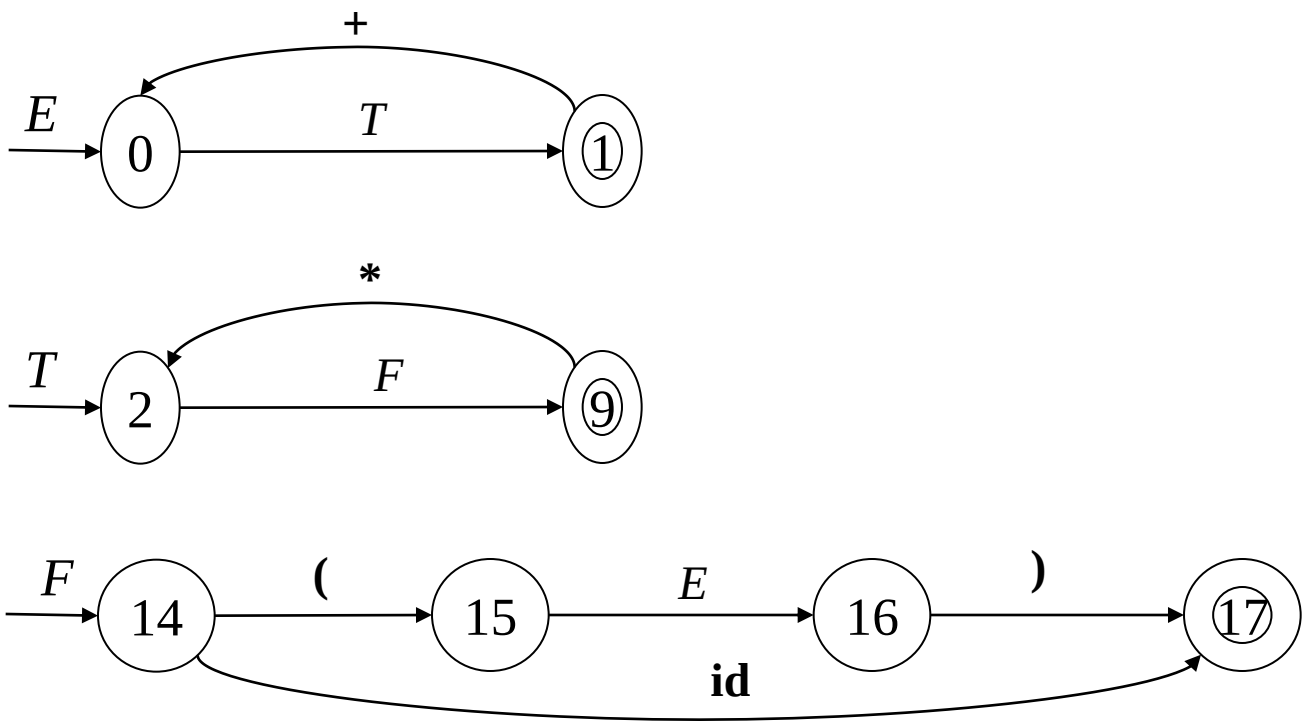
F --> **(** E **)** | **N**

Diagrammes de transitions

Les grammaires peuvent être mises sous la forme de diagrammes de transitions



Diagrammes de transitions



Analyse syntaxique

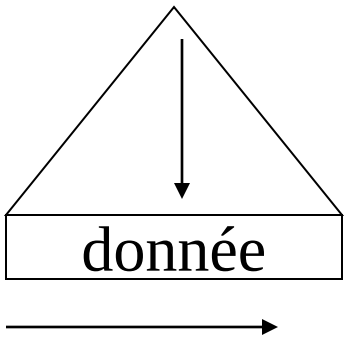
Objectif

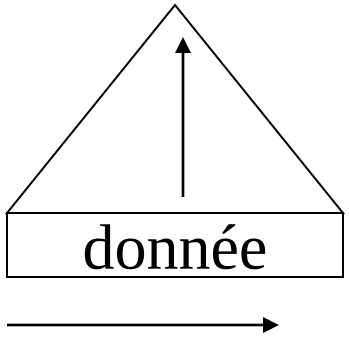
Construire l'arbre de dérivation d'un mot donné

Méthodes

Analyse descendante (descente récursive) : on parcourt le mot en appelant des procédures pour chaque non-terminal

Analyse ascendante : on parcourt le mot en empilant les symboles identifiés





Analyse descendante

Exemple : traduction des expressions arithmétiques dans le mini-compilateur

Cette technique n'est pas applicable à la grammaire suivante :

E --> *E* **+** *T* | *E* **-** *T* | *T*

T --> *T* ***** *F* | *T* **/** *F* | *F*

F --> **(** *E* **)** | **N**

En raison de la **récursivité à gauche,** la fonction expr() s'appellerait elle-même sans consommer de lexèmes et bouclerait

E --> *T E'*

E' --> **+** *T E'* | **-** *T E'* | ε

T --> *F T'*

T' --> ***** *F T'* | **/** *F T'* | ε

F --> **(** *E* **)** | **N**

Cette grammaire est récursive à droite : pas de problème

Suppression de la récursivité à gauche

Lemme d'Arden

$X \dashrightarrow Xu \mid v$ $\qquad$ $X \dashrightarrow vX'$

$X' \dashrightarrow uX' \mid \varepsilon$

Dans les deux cas, $X \overset{*}{\dashrightarrow} vu^*$

Avec plusieurs variables cela peut ne pas suffire

$X \dashrightarrow Xa \mid Yb \mid c$ $\qquad$ $X \dashrightarrow (Yb \mid c)X'$

$Y \dashrightarrow Xd \mid Ye \mid f$ $\qquad$ $X' \dashrightarrow aX' \mid \varepsilon$

$Y \dashrightarrow (Xd \mid f)Y'$

$Y' \dashrightarrow eY' \mid \varepsilon$

Forme matricielle du lemme d'Arden

Prédiction : Premier et Suivant

Pour l'analyse descendante et l'analyse ascendante, on doit prédire quelle règle appliquer en fonction du prochain lexème

Pour cela on définit deux fonctions

Premier() de $(A \mid V)^*$ dans $A \cup \{\varepsilon\}$

$a \in A$: a est dans Premier(u) si on peut dériver à partir de u un mot commençant par a

ε est dans Premier(u) si on peut dériver ε à partir de u

Suivant() de V dans A

$a \in A$: a est dans Suivant(X) si on peut dériver à partir de l'axiome un mot dans lequel X est suivi de a

Prédiction : Premier et Suivant

E --> *E* **+** *T* | *E* **-** *T* | *T*

T --> *T* ***** *F* | *T* / *F* | *F*

F --> **(** *E* **)** | **N**

Premier()

(est dans Premier(*T*) car *T* --> *F* --> **(** *E* **)**

Suivant()

) est dans Suivant(*T*) car *E* --> *T* --> *F* --> **(** *E* **)** --> **(** *E* **+** *T* **)**

Calcul de Premier et Suivant

Algorithme

On construit deux graphes dont les sommets sont des symboles et ε

On a un chemin de X à a (ou ε) ssi a (ou ε) est dans Premier(X) (ou Suivant(X))

Premier()

On a un arc de X vers $Y \in A \cup V$ ssi il existe une règle $X \dashrightarrow uYv$ avec $u \overset{*}{\dashrightarrow} \varepsilon$

On a un arc de X vers ε ssi il existe une règle $X \dashrightarrow \varepsilon$

Suivant()

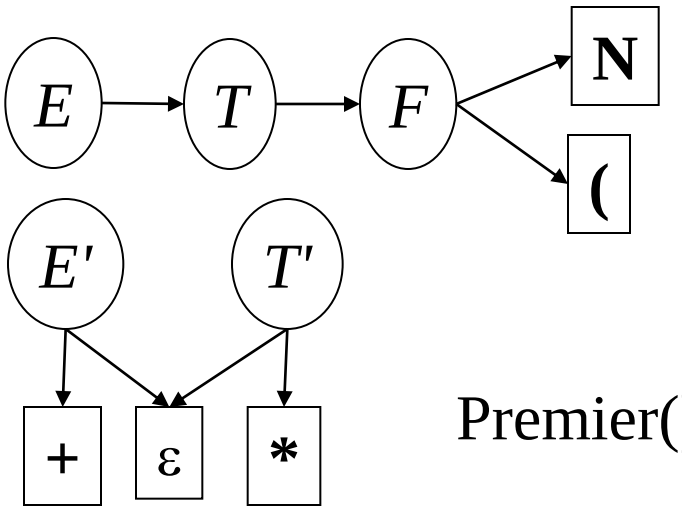
On a un arc de X vers a ssi il existe une règle $Y \dashrightarrow uXv$ avec $a \in$ Premier(v)

On a un arc de X vers Y ssi il existe une règle $Y \dashrightarrow uXv$ avec $v \overset{*}{\dashrightarrow} \varepsilon$

(attention, l'arc remonte la flèche de dérivation, car Suivant(Y)⊆Suivant(X))

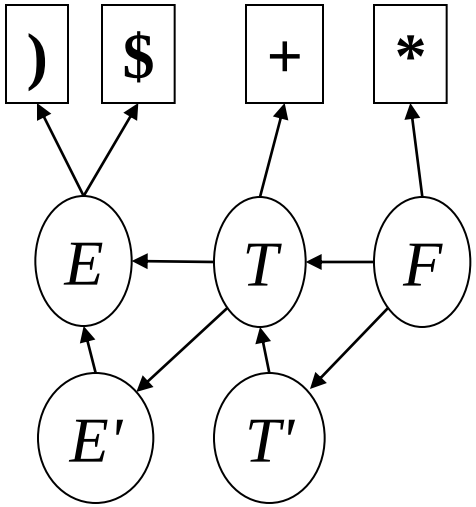
Exemple

S -->	*E* **$**
E -->	*T E'*
E' -->	**+** *T E'* | ε
T -->	*F T'*
T' -->	***** *F T'* | ε
F -->	**(** *E* **)** | **N**



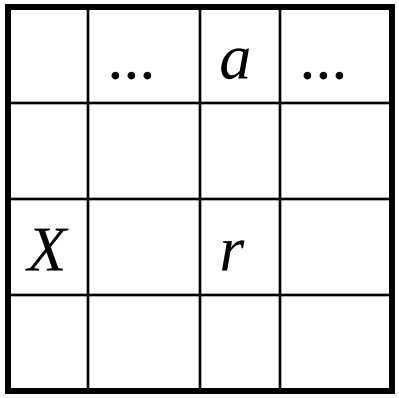
Premier()

Suivant()



Analyse LL

On utilise Premier() et Suivant() pour construire une table d'analyse



La règle *r* est

- soit *X* --> *u* et *a* est dans Premier(*u*)
- soit *X* --> ε et *a* est dans Suivant (*X*)

Il y a un conflit s'il y a deux règles dans la même case du tableau

La grammaire est LL(1) s'il n'y a pas de conflits

Exemple

	+	*	(	)	N	$
E			1		1	
E'	2			3		3
T			4		4	
T'	6	5		6		6
F			7		8	

1	*E* -->	*T E'*
2	*E'* -->	+ *T E'*
3	*E'* -->	ε
4	*T* -->	*F T'*
5	*T'* -->	* *F T'*
6	*T'* -->	ε
7	*F* -->	**(** *E* **)**
8	*F* -->	**N**

Analyse ascendante

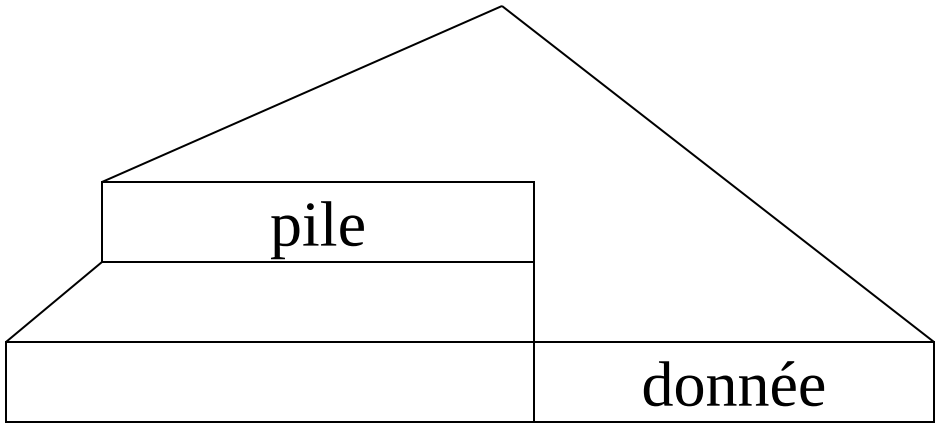
On utilise une pile pour ranger la partie déjà analysée

Chaque symbole dans la pile correspond à un symbole terminal ou non terminal ou à ε

Il y a deux sortes d'actions :

- **empiler** un symbole terminal de la donnée vers la pile

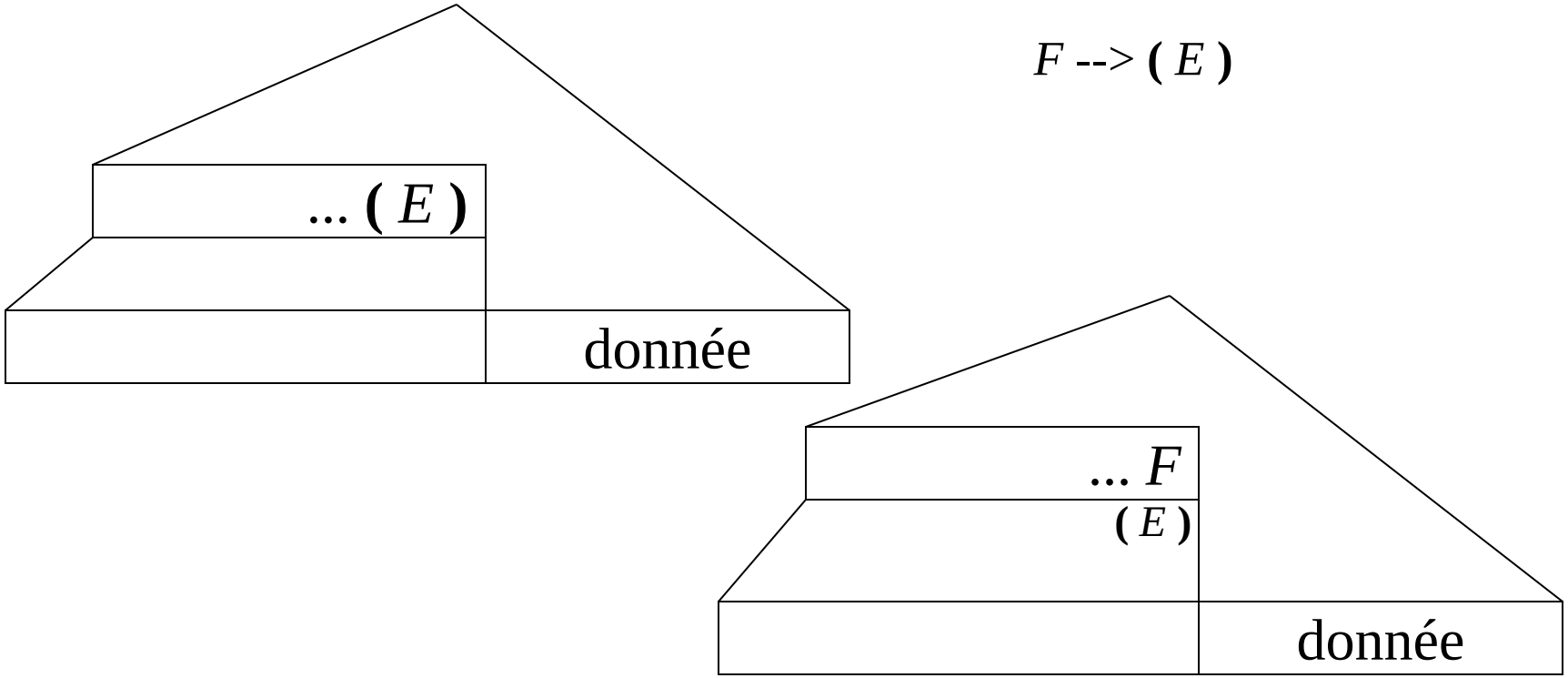
- **réduire** un membre droit de règle en sommet de pile (dépiler le membre droit puis empiler le membre gauche)



Analyse ascendante

Réduire un membre droit de règle en sommet de pile

- dépiler le membre droit
- empiler le membre gauche



Analyseur LR

On construit un analyseur LR utilisant

- des états qui seront les symboles dans la pile
- des actions "empiler" notées **e** *q* qui empilent un état *q*
- des actions "réduire" notées **r** *n* (*n* = numéro de la règle)
- des transitions qui donnent l'état *s* à empiler à la fin d'une réduction

état	terminaux	non-terminaux
	a	*X*
p	**e *q***	
q	**r** *n*	
r		*s*

Analyseur LR

Si l'état en sommet de pile est *q*

et si le prochain terminal est *a*

faire **r** *n* (réduire suivant la règle *n*) :

Si la règle est *X* --> *w*

1. dépiler |*w*| états
2. si on a l'état *r* en sommet de pile, empiler *s*

état	terminaux	non-terminaux
	a	*X*
p	**e *q***	
q	**r** *n*	
r		*s*

Analyse ascendante des expressions arithmétiques

0 S --> E **$**

1 E --> E **+** T

2 E --> T

3 T --> T ***** F

4 T --> F

5 F --> **(** E **)**

6 F --> **N**

L'axiome de la grammaire d'origine était E

On a ajouté une règle supplémentaire

S --> E **$**

pour pouvoir calculer Suivant(E)

Analyse ascendante des expressions arithmétiques

	terminaux						non-terminaux		
état	**N**	+	*	**(**	**)**	**$**	*E*	*T*	*F*
0 (ε)	e5			e4			1	2	3
1 (*E*)		e6				acc			
2 (*T*)		r2	e7		r2	r2			
3 (*F*)		r4	r4		r4	r4			
4 (**(**)	e5			e4			8	2	3
5 (**N**)		r6	r6		r6	r6			
6 (+)	e5			e4				9	3
7 (*)	e5			e4					10
8 (*E*)		e6			e11				
9 (*T*)		r1	e7		r1	r1			
10 (*F*)		r3	r3		r3	r3			
11 (**)**)		r5	r5		r5	r5			

Analyse ascendante des expressions arithmétiques

0 *S* --> *E* **$**

1 *E* --> *E* **+** *T*

2 *E* --> *T*

3 *T* --> *T* ***** *F*

4 *T* --> *F*

5 *F* --> **(** *E* **)**

6 *F* --> **N**

Donnée	Pile		Règle
3*5+4$	0	ε	
*5+4$	0 5	**N**	
*5+4$	0 3	*F*	*F* --> **N**
*5+4$	0 2	*T*	*T* --> *F*
5+4$	0 2 7	*T* *****	
+4$	0 2 7 5	*T* ***** **N**	
+4$	0 2 7 10	*T* ***** *F*	*F* --> **N**
+4$	0 2	*T*	*T* --> *T* ***** *F*

Construction de la table

Les états sont obtenus comme états d'un automate déterministe appelé automate LR(0)

Les états sont des ensembles de règles marquées

$$X \dashrightarrow u.v$$

c'est-à-dire :

> On a déjà u en sommet de pile et on attend d'empiler v pour réduire suivant la règle $X \dashrightarrow uv$ (dépiler uv et empiler X)

Les états de l'automate non déterministe sont les règles marquées

Construction de la table

Les états initiaux sont les *S* --> .*u* (*S* est l'axiome)

Les états finaux sont les *X* --> *u*.

Les transitions sont de deux sortes :

- (empiler) passer de *X* --> *u*.*Yv* à *X* --> *uY*.*v* en lisant *Y*
- (réduire) passer de *X* --> *u*.*Yv* à *Y* --> .*w* par une transition spontanée

Exemple

On obtient pour cette grammaire un automate déterministe avec 12 états

Ci-dessous les états sont représentés sans les *Y* --> .*w*

0 : *S* --> .*E* **$**

1 : *E* --> *E*.**+** *T*

2 : *E* --> *T*.
T --> *T*.***** *F*

3 : *T* --> *F*.

4 : *F* --> **(**.*E* **)**

5 : *F* --> **N**.

6 : *E* --> *E* **+**.*T*

7 : *T* --> *T* *****.*F*

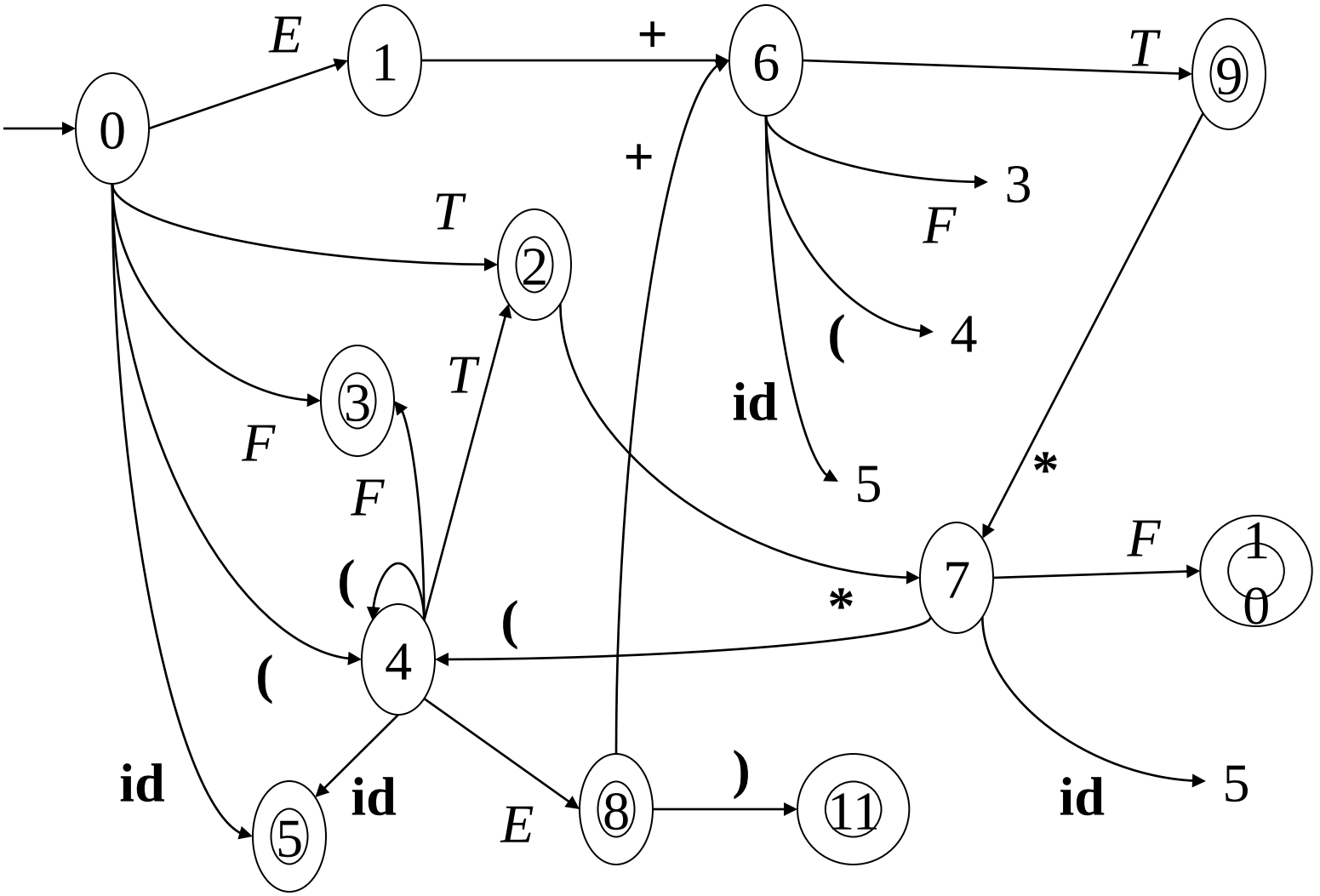
8 : *T* --> *T* ***** *F*.

9 : *E* --> *E* **+** *T*.
T --> *T*.***** *F*

10 : *T* --> *T* ***** *F*.

11 : *F* --> **(** *E* **)**.

Exemple



Construction de la table

Les transitions étiquetées par des symboles terminaux donnent les actions "empiler" : empile l'état but

Les états finaux donnent les actions "réduire" : on réduit suivant la règle qui correspond à *X* --> *u*.

Les transitions étiquetées par des non-terminaux donnent les transitions dans les colonnes des non-terminaux : empiler l'état but

Il y a un conflit quand une case contient deux actions

Exemple : colonne ***** dans les états 2 et 9

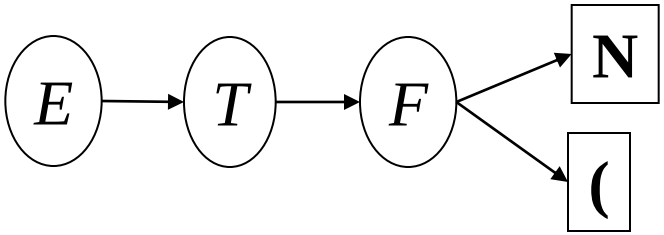
Analyse SLR(1)

C'est la stratégie de solution des conflits la plus simple

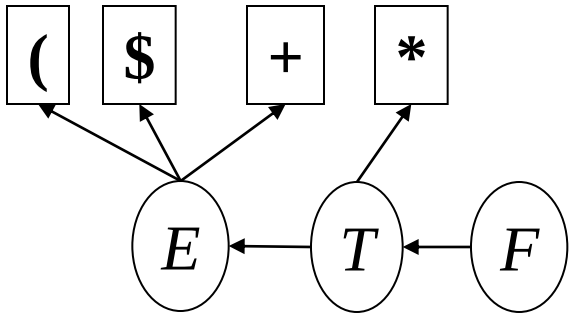
	...	**a**	...
q		?	

On ne place dans le tableau une action de réduction par la règle X --> u que si **a** $\in$ Suivant(X)

Analyse SLR(1)



Premier()



Suivant()

	…	*	…
2		e7	
9		e7	

Analyse SLR(1)

Pour certaines grammaires, la stratégie SLR(1) ne suffit pas à résoudre tous les conflits

Grammaire des affectations en C

S --> *L* = *R*

S --> *R*

L --> * *R*

L --> **id**

R --> *L*

On analyse **id = id**

pile	donnée
	id = id
id	**= id**
L	**= id**

On peut réduire par *R* --> *L* car = est dans Suivant(*R*)

Méthodes plus puissantes

Analyse LR(1)

L'automate est différent (il peut avoir plus d'états)

Analyse LALR(1)

Intermédiaire entre SLR(1) et LR(1)

On fait l'automate LR(1)

On regroupe les états pour obtenir l'automate LR(0)

On utilise l'automate LR(1) pour résoudre les conflits

Exemple LALR(1)

Grammaire des affectations en C

S --> *L* = *R*

S --> *R*

L --> * *R*

L --> **id**

R --> *L*

pile	donnée
	id = id
id	**= id**
L	**= id**

On analyse **id = id**

Dans l'automate LR(1), l'état correspondant à cette situation est :

S --> *L*.= *R*, **$**

R --> *L*. , **$**

où **$** est le symbole terminal attendu lors de la réduction.

Dans ce cas, on ne peut donc pas réduire par *R* --> *L* et on empile =.

Exemple

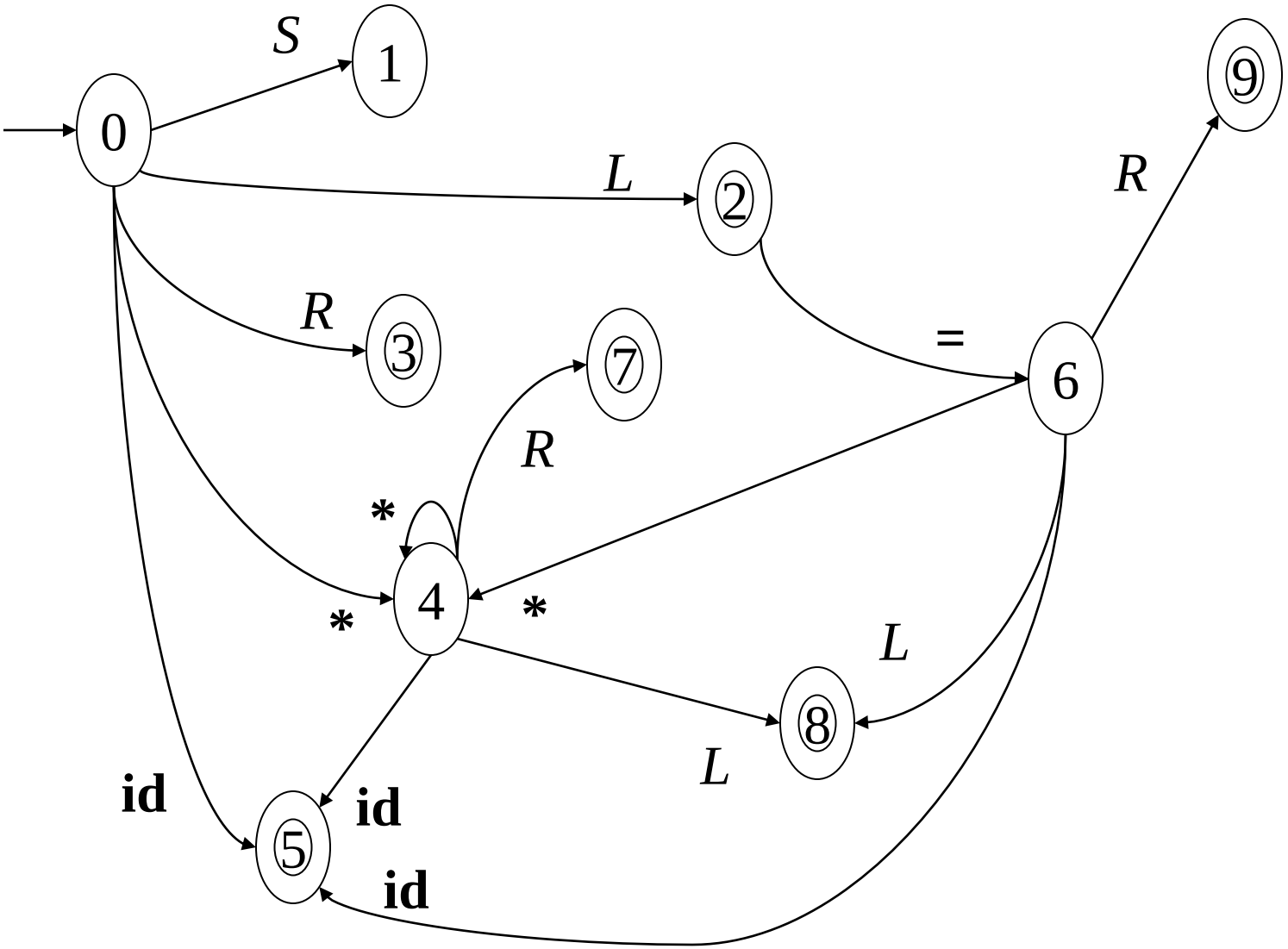


Table d'analyse

	terminaux				non-terminaux		
état	=	*	**id**	**$**	*S*	*L*	*R*
0 (ε)		e4	e5		1	2	3
1 (*S*)				acc			
2 (*L*)	e6			r5			
3 (*R*)				r2			
4 (*)		e4	e5			8	7
5 (**id**)	r4			r4			
6 (=)		e4	e5			8	9
7 (*R*)	r3			r3			
8 (*L*)	r5			r5			
9 (*R*)				r1			

Utilisation de grammaires ambigües

Avec une grammaire ambigüe, on peut résoudre les conflits de l'analyse LR(0) en utilisant des règles d'associativité et de priorité.

E --> *E* **+** *E*

E --> *E* ***** *E*

E --> **(** *E* **)**

E --> **id**

Priorité de ***** sur **+**

Associativité à gauche

Automate LR(0)

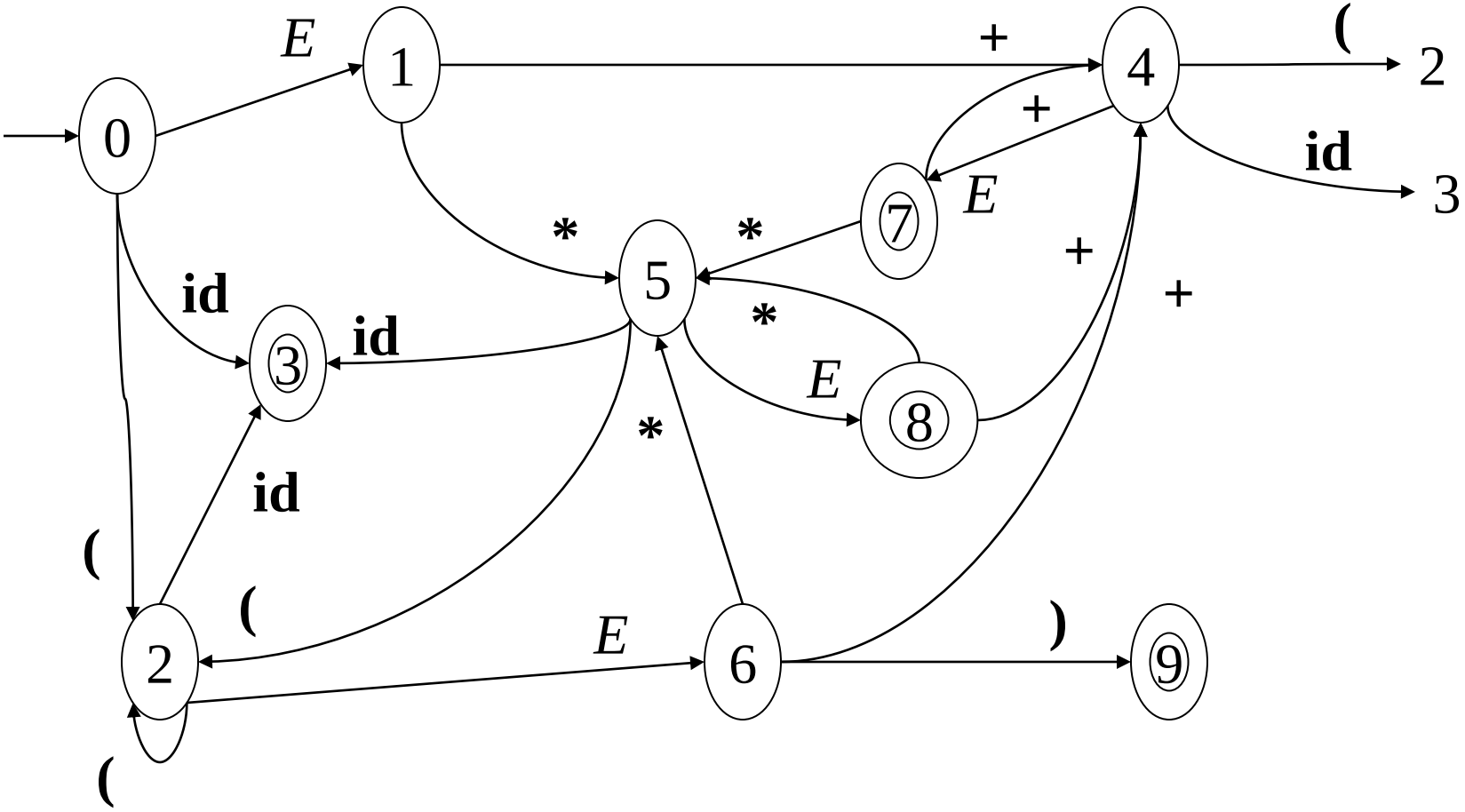


Table d'analyse

	terminaux						n.-t.
état	**id**	+	*	**(**	**)**	**$**	*E*
0 (ε)	e3			e2			1
1 (*E*)		e4	e5			acc	
2 (**(**)	e3			e2			6
3 (**id**)		r4	r4		r4	r4	
4 (+)	e3			e2			7
5 (*****)	e3			e2			8
6 (*E*)		e4	e5		e9		
7 (*E*)		**r1**	**e5**		r1	r1	
8 (*E*)		**r2**	**r2**		r2	r2	
9 (**)**)		r3	r3		r3	r3	

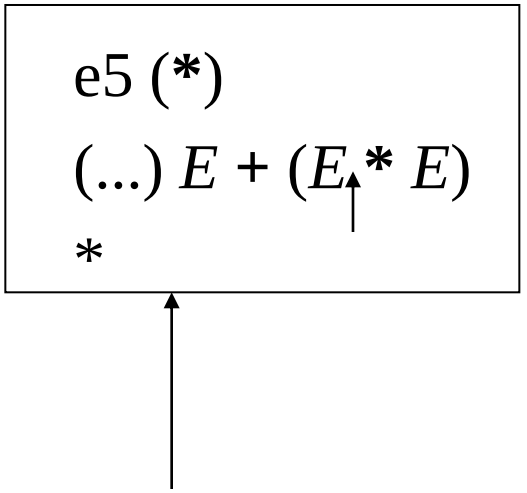
Utilisation de grammaires ambigües

Conflit lié aux priorités

Etat 7 : le sommet de la pile représente (...) *E* + *E*

Prochain symbole terminal : *****

Conflit :	r1 (*E* --> *E* + *E*)	e5 (*****)
	(...) (*E* + *E*) ***** *E*	(...) *E* + (*E* ***** *E*)
Priorité :	+	*



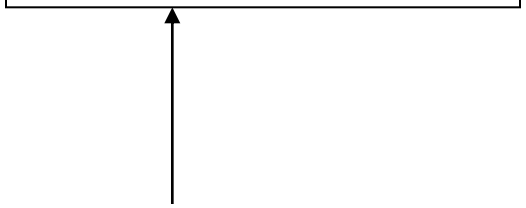
Utilisation de grammaires ambigües

Conflit lié à l'associativité

Etat 7 : le sommet de la pile représente (...) $E + E$

Prochain symbole terminal : +

Conflit :	r1 (E --> $E + E$)	e4 (+)
	(...) ($E + E$) + E	(...) E + (E + E)
Associativité :	à gauche	à droite



Traitement des erreurs

Emission des messages (diagnostic)

On choisit arbitrairement une des hypothèses possibles

Exemple : **e = a + b c ;**

- opérateur manquant (**e = a + b * c ;**)
- identificateur en trop (**e = a + b ;**)
- erreur lexicale (**e = a + bc ;**)

...

Redémarrage

Pour pouvoir compiler la partie après la première erreur

Méthodes de redémarrage

Mode panique

Éliminer les prochains symboles terminaux de la donnée

Mode correction

Modifier les prochains symboles terminaux pour reconstituer ce que serait la donnée sans l'erreur détectée

Le message d'erreur doit être cohérent avec l'hypothèse faite

Règles d'erreur

Ajouter à la grammaire des constructions incorrectes avec leur traitement

La grammaire utilisée par l'analyseur est distincte de celle décrite dans le manuel d'utilisation

Mode panique avec l'analyse LR

X est un non-terminal fixé à l'avance

On suppose que l'erreur est au milieu d'un *X* et on essaie de redémarrer après ce *X*

Dépiler jusqu'à un état *q* qui ait une transition par *X*

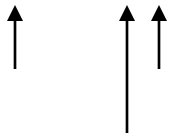
Eliminer des symboles terminaux de la donnée jusqu'à rencontrer un symbole terminal $a \in$ Suivant(*X*)

Reprendre l'analyse à partir de la case (*q*, *X*) dans la table

Exemple

X = *instr*

{ *x* = [; *y* = 0 ;



Mode correction avec l'analyse LR

Ajouter dans les cases vides de la table des appels à des fonctions de traitement d'erreur

Les fonctions émettent un message et effectuent des actions pour redémarrer après l'erreur

Exemple

se1 émettre "opérande manquant" et empiler 3 (**id**)

se2 émettre "parenthèse fermante en trop" et éliminer **)** de la donnée

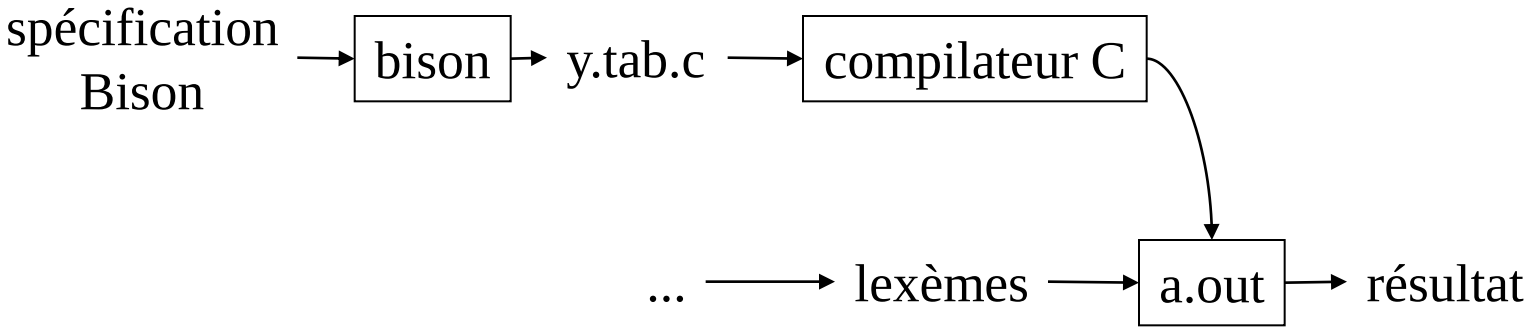
se3 émettre "opérateur manquant" et empiler 4 (+)

se4 émettre "parenthèse fermante manquante" et empiler 9 (**)**)

Exemple

	terminaux						n.-t.
état	**id**	+	*	**(**	**)**	**$**	*E*
0 (ε)	e3	**se1**	**se1**	e2	**se2**	**se1**	1
1 (*E*)	**se3**	e4	e5	**se3**	**se2**	acc	**1**
2 (**(**)	e3	**se1**	**se1**	e2	**se2**	**se1**	6
3 (**id**)	**r4**	r4	r4	**r4**	r4	r4	
4 (+)	e3	**se1**	**se1**	e2	**se2**	**se1**	7
5 (*)	e3	**se1**	**se1**	e2	**se2**	**se1**	8
6 (*E*)	**se3**	e4	e5	**se3**	e9	**se4**	**1**
7 (*E*)	**r1**	r1	e5	**r1**	r1	r1	**1**
8 (*E*)	**r2**	r2	r2	**r2**	r2	r2	
9 (**)**)	**r3**	r3	r3	**r3**	r3	r3	

Utilisation de Bison



4 étapes :

- créer sous éditeur une spécification Bison (grammaire)
- traiter cette spécification par la commande bison
- compiler le programme source C obtenu
- exécuter le programme exécutable obtenu (analyseur LALR(1))

Spécifications Bison

Un programme Bison est fait de trois parties :

déclarations

%%

règles de traduction

%%

fonctions auxiliaires en C

Les règles de traduction sont de la forme

X : $expr_1$ **{** $action_1$ **}**

| $expr_2$ **{** $action_2$ **}**

...

| $expr_n$ **{** $action_n$ **}**

;

où chaque X --> $expr_i$ est une règle et chaque action une suite d'instructions en C.

Exemple

```
%{
/* Calculette */
#include <ctype.h>
%}
%token CHIFFRE
%%
ligne   : expr '\n'    { printf("%d\n", $1) ; }
        ;
expr    : expr '+' terme { $$ = $1 + $3 ; }
        | terme
        ;
terme   : terme '*' fact { $$ = $1 * $3 ; }
        | fact
        ;
fact    : '(' expr ')' { $$ = $2 ; }
        | CHIFFRE
        ;
```

Exemple

```
%%
yylex() {
   int c ;
   c = getchar() ;
   if (isdigit(c)) {
        yylval = c - '0' ;
        return CHIFFRE ; }
   return c ; }
```

Spécifications Bison

Les commentaires /* ... */ ne peuvent être insérés que dans une portion en C :

- dans la partie déclaration, seulement entre **%{** et **%}** ;
- dans la partie règles, seulement dans les actions ;
- dans la partie fonctions auxiliaires, n'importe où.

Dans les règles

X : $expr_1$ **{** $action_1$ **}**

| $expr_2$ **{** $action_2$ **}**

les actions peuvent porter sur les attributs des variables :

$$ est l'attribut de X

$1, $2, ... sont les attributs des symboles de $expr_1$ ou $expr_2$

Utilisation de grammaires ambigües

```
expr    : expr '+' expr { $$ = $1 + $3 ; }
        | expr '*' expr { $$ = $1 * $3 ; }
        | '(' expr ')' { $$ = $2 ; }
        | CHIFFRE
        ;
```

Bison trouve les conflits, les décrit (option -v) et les résout par défaut

Conflit empiler/réduire : en faveur de l'action empiler

Conflit réduire/réduire : en faveur de la règle qui figure en premier dans la spécification Bison

Si cette résolution par défaut ne correspond pas à ce qui est souhaité, on peut faire des déclarations Bison pour résoudre le conflit autrement

Utilisation de grammaires ambigües

```
expr    : expr '+' expr { $$ = $1 + $3 ; }
        | expr '*' expr { $$ = $1 * $3 ; }
        | '(' expr ')' { $$ = $2 ; }
        | CHIFFRE
        ;
```

Associativité

dans la partie déclaration ajouter une déclaration **%left** ou **%right**

```
%{
/* Calculette */
#include <ctype.h>
%}
%token CHIFFRE
%left '+'
%%
```

Utilisation de grammaires ambigües

Conflit empiler/réduire résolu avec les déclarations %left ou %right

- de l'opérateur à empiler
- et du dernier symbole terminal de la règle à réduire

Si déclarés %left, on réduit

Si déclarés %right, on empile

Associativité : cela peut être le même symbole dans les deux cas

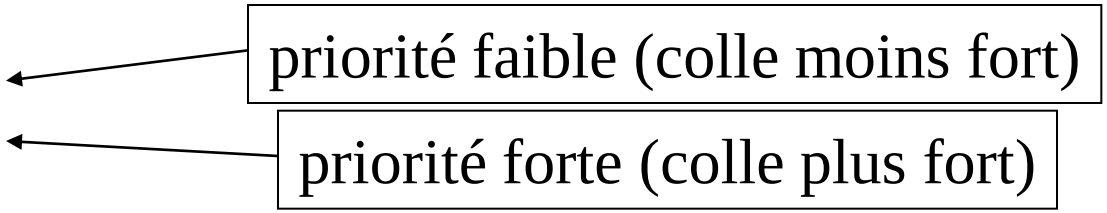
On peut déclarer plusieurs opérateurs dans la même ligne avec %left ou %right :

```
%left '+' '-'
%%
```

Priorités

Placer les déclarations d'associativité **%left** ou **%right** dans l'ordre des priorités croissantes

```
%{
/* Calculette */
#include <ctype.h>
%}
%token CHIFFRE
%left '+' '-'
%left '*' '/'
%%
```



On ne peut pas déclarer les priorités sans les associativités

Priorités

Conflit empiler/réduire résolu avec les déclarations
- de l'opérateur à empiler
- et du dernier symbole terminal de la règle à réduire

Si le plus prioritaire des deux est celui de la règle, on réduit
Si c'est celui à empiler, on empile
(S'ils ont la même priorité, on applique l'associativité)

Priorités

Exceptions

La 5e règle a la priorité de **-** (dernier symbole terminal)

```
expr : expr '+' expr { $$ = $1 + $3 ; }
             | expr '-' expr { $$ = $1 - $3 ; }
             | expr '*' expr { $$ = $1 * $3 ; }
             | expr '/' expr { $$ = $1 / $3 ; }
             | '-' expr { $$ = - $2 ; }
             | '(' expr ')' { $$ = $2 ; }
             | CHIFFRE
             ;
```

On veut au contraire qu'elle ait une priorité plus forte que *****

Priorités

Exceptions

On déclare un symbole terminal supplémentaire UnaryMinus, de priorité plus forte que *****

On ajoute une déclaration **%prec** dans la règle

```
%{
/* Calculette */
#include <ctype.h>
%}
%token CHIFFRE
%left '+' '-'
%left '*' '/'
%left UnaryMinus
%%
     expr    : expr '+' expr { $$ = $1 + $3 ; }
             | (...)
             | '-' expr %prec UnaryMinus { $$ = - $2 ; }
             | (...)
```

Mode panique avec Bison

On ajoute des règles du type

$$X : \text{error } u$$

pour certains non-terminaux X déjà dans la grammaire

Si l'erreur est au milieu d'un X, on essaie de redémarrer après le premier u comme s'il terminait ce X

Dépiler jusqu'à un état q qui ait une transition par X

Eliminer des symboles de la donnée jusqu'au u compris

Reprendre l'analyse à partir de la case (q, X) dans la table

Exemple

instr : error ';' { x = [; y = 0 ;