PROC – Compilation

Christophe Mouilleron

Organisation du cours

Planning :

- 1 demi-journée de Cours + TP — `flex`
- 1 demi-journée de Cours + TD — analyse LR
- 1 demi-journée de TP (+ projet ?) — `bison`
- 1 demi-journée de projet

Organisation du cours

Planning :

- 1 demi-journée de Cours + TP — `flex`
- 1 demi-journée de Cours + TD — analyse LR
- 1 demi-journée de TP (+ projet ?) — `bison`
- 1 demi-journée de projet

Evaluation = Projet :

- rendu(s) intermédiaire(s) en TP
- rendu final sur exam pour le **vendredi 20 juin** à 23h59

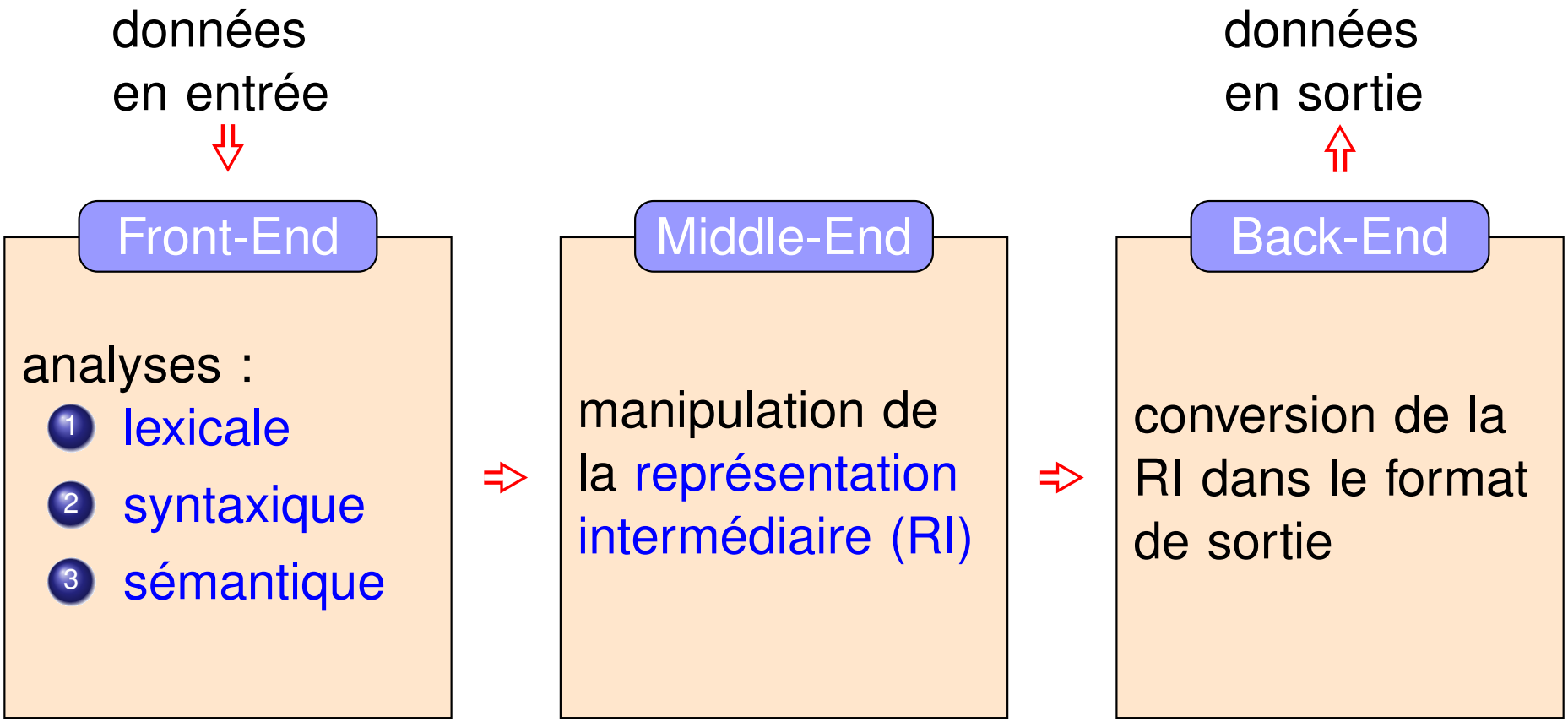
Plan

1. Compilation : Vue d'ensemble
2. Analyse lexicale
   - Reconnaissance des lexèmes
   - Passage à la pratique avec `flex`
3. Analyse syntaxique
   - Utilisation de `bison`
   - Dérivations et arbres syntaxiques
   - Analyse ascendante
   - Analyse de type LR(0)
   - Résolution de conflits

Plan

1. Compilation : Vue d'ensemble
2. Analyse lexicale
   - Reconnaissance des lexèmes
   - Passage à la pratique avec `flex`
3. Analyse syntaxique
   - Utilisation de `bison`
   - Dérivations et arbres syntaxiques
   - Analyse ascendante
   - Analyse de type LR(0)
   - Résolution de conflits

Principales étapes du processus de compilation



Exemple 1 : fichier C → exécutable

Front-end = production d'un AST (abstract syntax tree) cohérent

- analyse lexicale → découpage en mots
  - constantes `3.14 "foo"`
  - symboles `+ {} ;`
  - mots clés `if void`
  - variables / noms de fonctions `main`
- analyse syntaxique → construction de l'AST
  - extraction de la structure à partir de la suite de mots
  - échec si syntaxe du C non respectée
- analyse sémantique → vérifications
  - cohérence des types
  - existence des variables

Exemple 1 : fichier C → exécutable (suite)

Middle-end = optimisations diverses

- élimination de code mort
- propagation de constantes
- déroulement de boucles
- ...

Back-end = production du fichier binaire

- allocation de registres
- sélection d'instructions

Exemple 2 : conversion d'un graphe de `XML` vers `DOT`

Entrée = fichier `.xml` représentant le graphe

Front-end = extraction des sommets et des arêtes

- découpage en mots (balises, contenus)
- analyse de la suite de mots pour obtenir les éléments du graphe

Middle-end = ajout de contraintes de placement

Back-end = production du fichier au format `DOT`

Sortie = fichier `.dot`

Programme

Remarques :

- Front-end toujours suivant un même schéma général
- Middle-end complètement dépendant du problème
- Back-end facile dans les cas courants (conversion de format)

$\rightsquigarrow$ On va s'intéresser au Front-end :

- analyse lexicale et utilisation de `flex`
- analyse syntaxique avec `bison`

Plan

1. Compilation : Vue d'ensemble
2. Analyse lexicale
   - Reconnaissance des lexèmes
   - Passage à la pratique avec `flex`
3. Analyse syntaxique
   - Utilisation de `bison`
   - Dérivations et arbres syntaxiques
   - Analyse ascendante
   - Analyse de type LR(0)
   - Résolution de conflits

Plan

1. Compilation : Vue d'ensemble
2. Analyse lexicale
   - Reconnaissance des lexèmes
   - Passage à la pratique avec `flex`
3. Analyse syntaxique
   - Utilisation de `bison`
   - Dérivations et arbres syntaxiques
   - Analyse ascendante
   - Analyse de type LR(0)
   - Résolution de conflits

Objectif de l'analyse lexicale

Problème

Découper une suite de caractères en entrée pour former une suite de mots appelés lexèmes (ou *tokens*).

lexème = unité lexicale
$\rightarrow$ entité assez simple à décrire
$\rightarrow$ caractérisation via une expression rationnelle

$\Rightarrow$ Reconnaissance d'un lexème faisable à l'aide d'un automate fini

Rappels du cours de LASF

Expression rationnelle sur un alphabet A
= ensemble Rat(A) défini inductivement :

- $\varepsilon \in \mathrm{Rat}(A)$ — mot vide
- si $a \in A$, alors $a \in \mathrm{Rat}(A)$ — lettre
- si $e_1, e_2 \in \mathrm{Rat}(A)$, alors $e_1 \cdot e_2 \in \mathrm{Rat}(A)$ — concaténation
- si $e_1, e_2 \in \mathrm{Rat}(A)$, alors $e_1 + e_2 \in \mathrm{Rat}(A)$ — union
- si $e \in \mathrm{Rat}(A)$, alors $e^* \in \mathrm{Rat}(A)$ — étoile

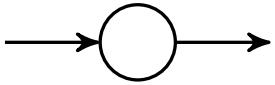
Procédé pour une reconnaissance efficace :

expression rationnelle
$\rightarrow$ automate fini via l'algorithme de Glushkov
$\rightarrow$ déterminisation puis minimisation

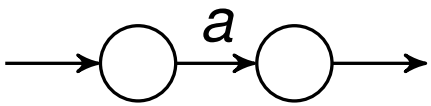
Expr. rationnelle → automate – Version compilation

Construction inductive de l'automate :

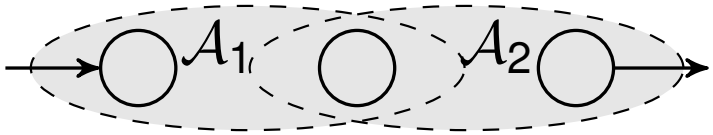
ε :



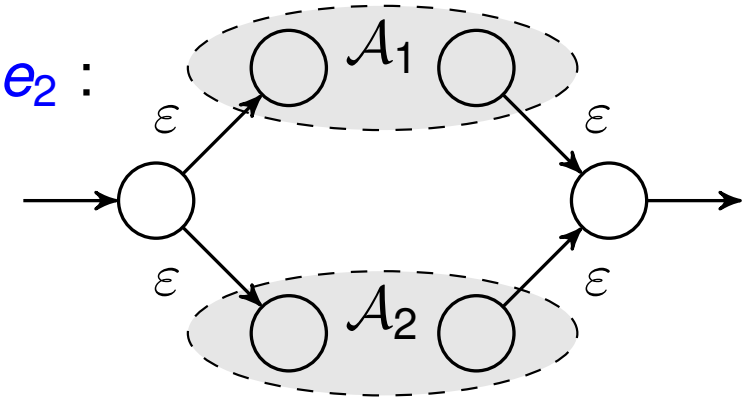
a :



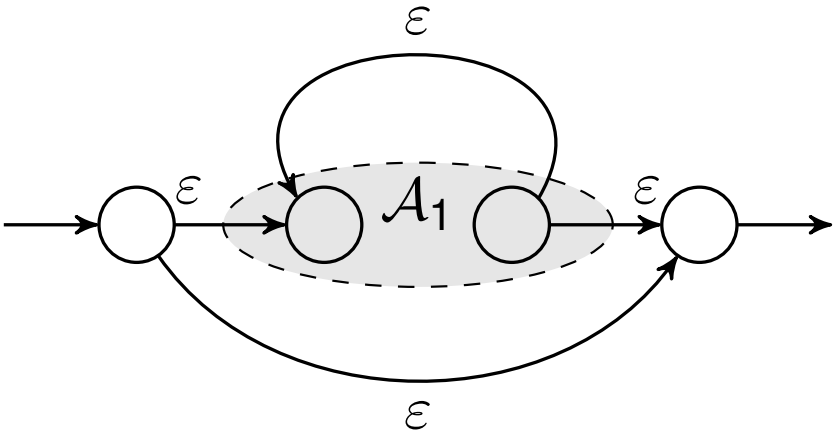
$e_1 \cdot e_2$:



$e_1 + e_2$:



e_1^* :



Comparaison des deux approches

Glushkov :

- ✓ faisable à la main
- ✓ nombre d'états limité
- ✗ algorithme un peu complexe à implanter

Version compilation :

- ✓ facile à coder
- ✗ produit beaucoup plus d'états
- ✗ introduit des transitions vides qu'il faudra éliminer

$\rightsquigarrow$ Version compilation plus adaptée dans le cas d'un outil informatique

Plan

1. Compilation : Vue d'ensemble
2. Analyse lexicale
   - Reconnaissance des lexèmes
   - Passage à la pratique avec `flex`
3. Analyse syntaxique
   - Utilisation de `bison`
   - Dérivations et arbres syntaxiques
   - Analyse ascendante
   - Analyse de type LR(0)
   - Résolution de conflits

L'outil `flex`

`flex` = `F`ast `LEX`ical analyzer

Prend en entrée un fichier avec :

- des expression rationnelles définissant les lexèmes
- des règles = traitements à effectuer lorsqu'un lexème est reconnu

Produit en sortie un fichier `C` :

- code reposant sur un unique automate fini
- traitement à effectuer déterminé par l'état d'acceptation

→ lancement de l'analyse grâce à la fonction `int yylex()`

Format d'entrée

Fichier `.l` ou `.lex`

```
%{
   #include <stdio.h>
   ...
%}

name1     regex1
name2     regex2
...
%option noyywrap
%%

name1   { return ...; }
name2   { return ...; }
...

%%

int f() {
   ... yylex() ...
}
```

entête en C

définition des différents lexèmes

options

règles = lexème + code C

code C

Exemple : calculatrice en notation polonaise inversée

⇝ voir `calc_rpn.l`

Compilation : `flex -dTv calc_rpn.l`

- -d à l'exécution, affiche les règles appliquées — mode debug
- -v à la compilation, affiche un compte-rendu
- -T à la compilation, affiche en plus les automates utilisés

⇝ Production de `lex.yy.c`

Automate non-déterministe utilisé par `flex`

Automate décrit par :

- l'état de départ
- les transitions depuis chaque état
  - au plus 2 successeurs
  - 2 successeurs $\Rightarrow$ 2 transitions ε
- pour chaque état, 0 ou 1 règle acceptée

Automate non-déterministe utilisé par `flex`

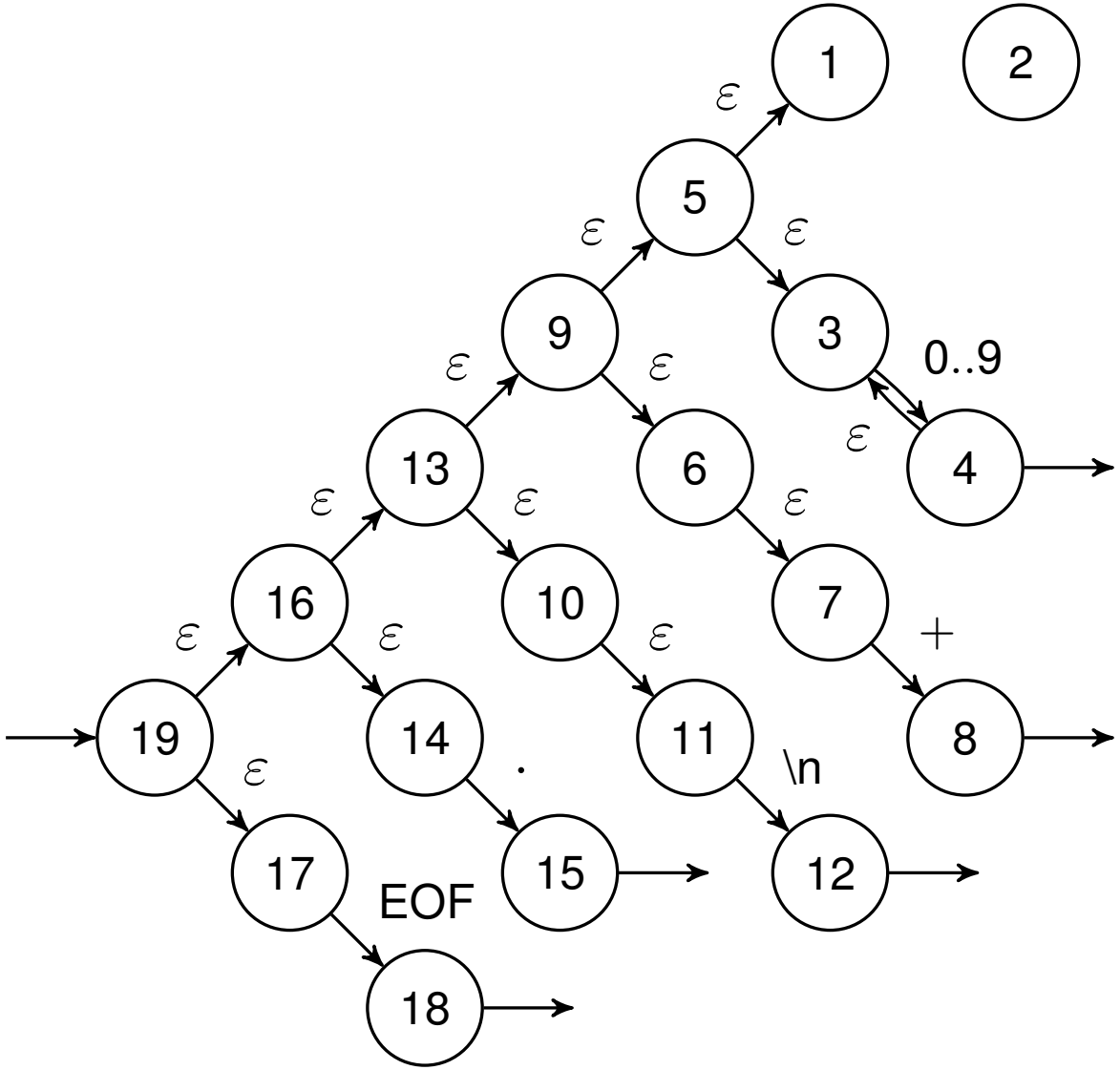
Automate décrit par :

- l'état de départ
- les transitions depuis chaque état
  - au plus 2 successeurs
  - 2 successeurs $\Rightarrow$ 2 transitions ε
- pour chaque état, 0 ou 1 règle acceptée

Convention pour les lettres :

- 257 = transition ε
- x = transition avec le caractère de code ASCII x
- -x = transition spéciale
  - x^{e} plage de caractères ([...]) dans la définition des lexèmes
  - joker ($\cdot$) ou fin de fichier (EOF)

Automate non-déterministe utilisé pour `calc_rpn.l`



Automate déterministe utilisé par `flex`

Automate décrit par :

- une correspondance ASCII $\rightarrow$ entier alphabet
- un état initial + pour chaque état, 0 ou 1 règle acceptée

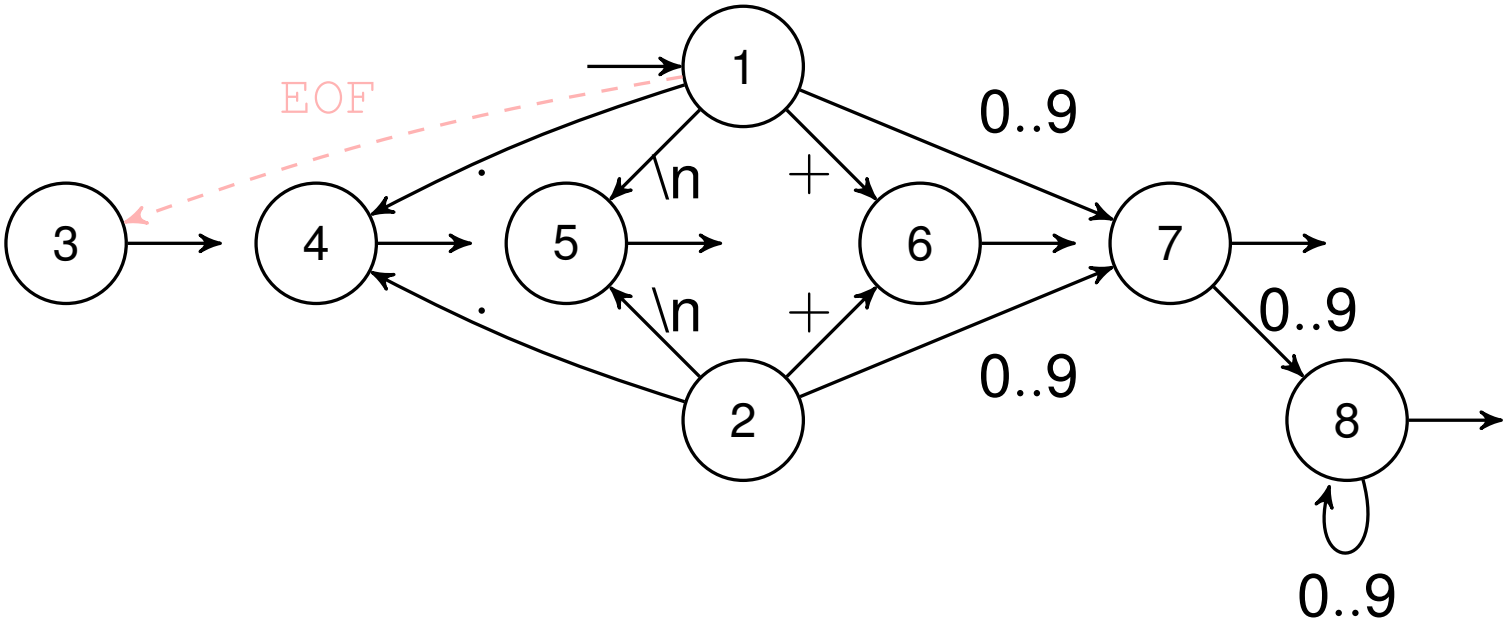
Sur l'exemple :

Automate déterministe utilisé par `flex`

Automate décrit par :

- une correspondance ASCII → entier alphabet
- un état initial + pour chaque état, 0 ou 1 règle acceptée

Sur l'exemple :



Remarques sur `flex`

Quelques règles supplémentaires :

- lecture d'un maximum de caractères pour former un lexème
  - ⇝ `"123"` → entier 123 (et pas 1 puis 2 puis 3)
  - ⇝ justifie l'état 8 dans l'exemple
- en cas d'égalité, priorité au premier lexème défini
  - ⇝ au plus une règle acceptée par état

Automate de `flex` pas forcément minimal cf exemple, état 2
→ compromis temps de calcul / taille du résultat

rappel : l'automate déterministe peut avoir jusqu'à 2^n états

Plan

1. Compilation : Vue d'ensemble
2. Analyse lexicale
   - Reconnaissance des lexèmes
   - Passage à la pratique avec `flex`
3. Analyse syntaxique
   - Utilisation de `bison`
   - Dérivations et arbres syntaxiques
   - Analyse ascendante
   - Analyse de type LR(0)
   - Résolution de conflits

Plan

1. Compilation : Vue d'ensemble
2. Analyse lexicale
   - Reconnaissance des lexèmes
   - Passage à la pratique avec `flex`
3. Analyse syntaxique
   - Utilisation de `bison`
   - Dérivations et arbres syntaxiques
   - Analyse ascendante
   - Analyse de type LR(0)
   - Résolution de conflits

L'outil `bison`

`bison` = amélioration de `yacc` (Yet Another Compiler Compiler)

Prend en entrée un fichier `.y` avec des :

- déclarations de lexèmes
- déclarations de types associés symboles non-terminaux
- règles de grammaire + traitements associés

Produit en sortie :

- un fichier `.tab.h` à inclure dans le `.l` — export des déclarations
- un fichier `.tab.c` — code du parseur

→ lancement de l'analyse grâce à la fonction `int yyparse()`

Format du fichier .y

```
%{
   #include <stdio.h>
   ...
%}

%union{ int val; ...; }

%token EOL
%token  <val>  NUM

%type  <val>  E
...
%start S

%%

...
E:
     NUM          { $$=$1; }
   | E PLUS E     { $$=$1+$3; }
   ;
...

%%

int f() { ... yyparse() ... }
```

entête en C

type de retour des traitements

déclaration des lexèmes (%token) avec ou sans valeur associée

association d'une valeur à un symbole non-terminal

ensemble de règles + code C associé

code C

Format du fichier .y

```
%{
   #include <stdio.h>
   ...
%}

%union{ int val; ...; }

%token EOL
%token <val> NUM

%type <val> E
...
%start S

%%

...
E:
     NUM          { $$=$1; }
   | E PLUS E     { $$=$1+$3; }
   ;
...

%%

int f() { ... yyparse() ... }
```

entête en C

type de retour des traitements

déclaration des lexèmes (%token) avec ou sans valeur associée

association d'une valeur à un symbole non-terminal

ensemble de règles + code C associé

code C

Exemple : calculatrice en notation polonaise inversée

⇝ voir `calc_rpn_bison.y` et `calc_rpn_bison.l`

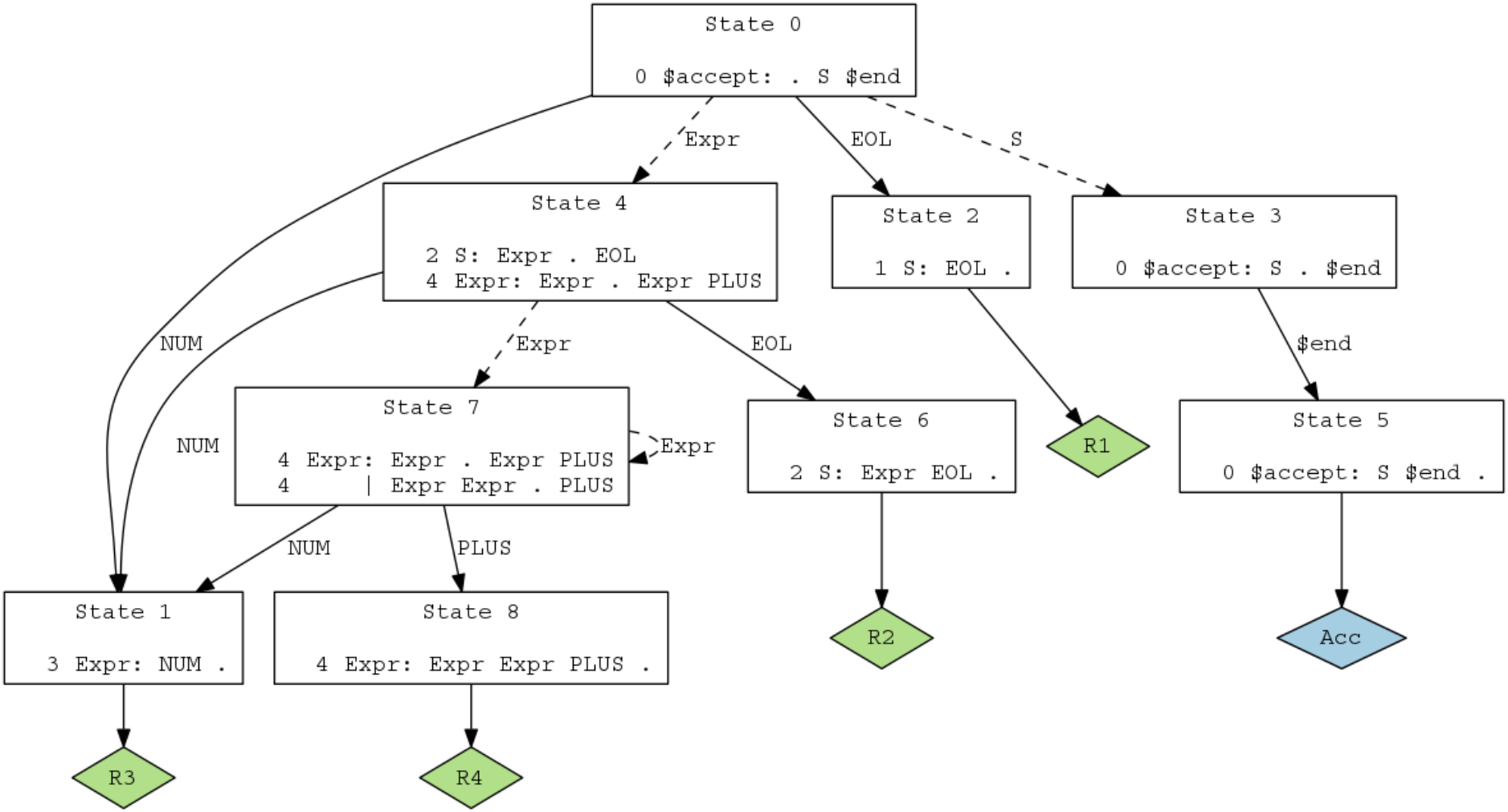
Compilation du `.y` : `bison -d -g -v calc_rpn_bison.y`

- -d pour avoir un `.tab.h` — déclarations
- -g produit l'automate associé au parseur — fichier `.gv` ou `.dot`
- -v mode verbeux
- -t mode debug + mettre `yydebug = 1;` avant `yyparse();`

Ensuite :

- `dot -Tpng calc_rpn_bison.gv -o out.png`
- appeler `flex`
- compiler les différents `.c`
- ajouter `%option noyywrap` au `.l` — ou linker avec l'option `-ll`

Analyseur utilisé par `bison`



Plan

1. Compilation : Vue d'ensemble
2. Analyse lexicale
   - Reconnaissance des lexèmes
   - Passage à la pratique avec `flex`
3. Analyse syntaxique
   - Utilisation de `bison`
   - Dérivations et arbres syntaxiques
   - Analyse ascendante
   - Analyse de type LR(0)
   - Résolution de conflits

Stratégies de dérivation

Considérons la grammaire $\mathcal{G}$: $S \to S + S \mid S \times S \mid 0 \mid 1 \mid \ldots \mid 9$

Génération de $2 + 3$?

Plusieurs stratégies possibles, dont :

- dérivation à gauche d'abord Leftmost

$$\underline{S} \to \underline{S} + S \to 2 + \underline{S} \to 2 + 3$$

Stratégies de dérivation

Considérons la grammaire $\mathcal{G}$: $S \to S + S \mid S \times S \mid 0 \mid 1 \mid \ldots \mid 9$

Génération de $2 + 3$?

Plusieurs stratégies possibles, dont :

- dérivation à gauche d'abord — Leftmost

$$\underline{S} \to \underline{S} + S \to 2 + \underline{S} \to 2 + 3$$

- dérivation à droite d'abord — Rightmost

$$\underline{S} \to S + \underline{S} \to \underline{S} + 3 \to 2 + 3$$

Dérivation et arbre syntaxique

Considérons la dérivation

$$\begin{array}{lllllll} & \underline{S} & & \rightarrow & \underline{S} + S & \rightarrow & 2 + \underline{S} \\ \rightarrow & 2 + \underline{S} \times S & & \rightarrow & 2 + 3 \times \underline{S} & \rightarrow & 2 + 3 \times 5 \end{array}$$

On peut lui associer un arbre syntaxique de la façon suivante :

$\underline{S}$

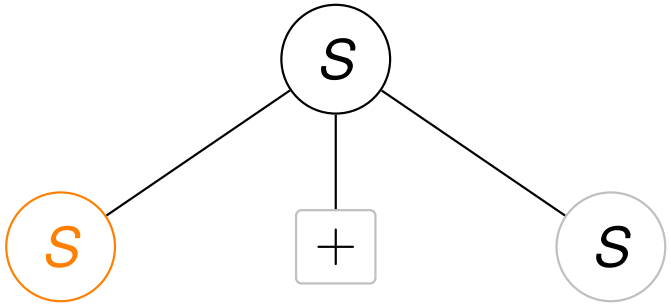
Dérivation et arbre syntaxique

Considérons la dérivation

$$\begin{array}{lllllll} & \underline{S} & & \rightarrow & \underline{S} + S & \rightarrow & 2 + \underline{S} \\ \rightarrow & 2 + \underline{S} \times S & & \rightarrow & 2 + 3 \times \underline{S} & \rightarrow & 2 + 3 \times 5 \end{array}$$

On peut lui associer un arbre syntaxique de la façon suivante :

$$\underline{S} \quad \rightarrow \quad \underline{S} + S$$



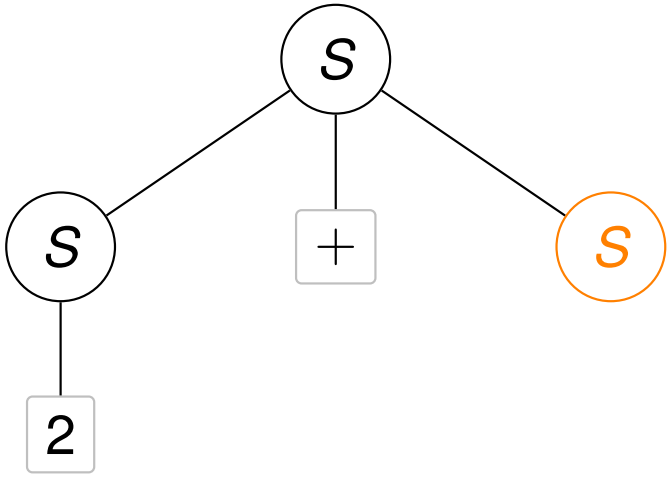
Dérivation et arbre syntaxique

Considérons la dérivation

$$\begin{array}{lllllll} & \underline{S} & & \rightarrow & \underline{S} + S & \rightarrow & 2 + \underline{S} \\ \rightarrow & 2 + \underline{S} \times S & & \rightarrow & 2 + 3 \times \underline{S} & \rightarrow & 2 + 3 \times 5 \end{array}$$

On peut lui associer un arbre syntaxique de la façon suivante :

$$\begin{array}{lll} \underline{S} & \rightarrow & \underline{S} + S \\ & \rightarrow & 2 + \underline{S} \end{array}$$



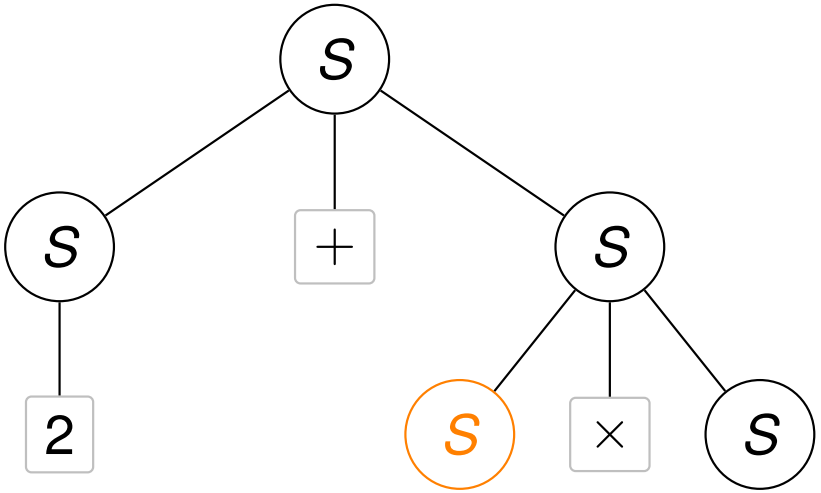
Dérivation et arbre syntaxique

Considérons la dérivation

$$\begin{array}{lllllll} & \underline{S} & & \rightarrow & \underline{S} + S & \rightarrow & 2 + \underline{S} \\ \rightarrow & 2 + \underline{S} \times S & & \rightarrow & 2 + 3 \times \underline{S} & \rightarrow & 2 + 3 \times 5 \end{array}$$

On peut lui associer un arbre syntaxique de la façon suivante :

$$\begin{array}{lll} \underline{S} & \rightarrow & \underline{S} + S \\ & \rightarrow & 2 + \underline{S} \\ & \rightarrow & 2 + \underline{S} \times S \end{array}$$



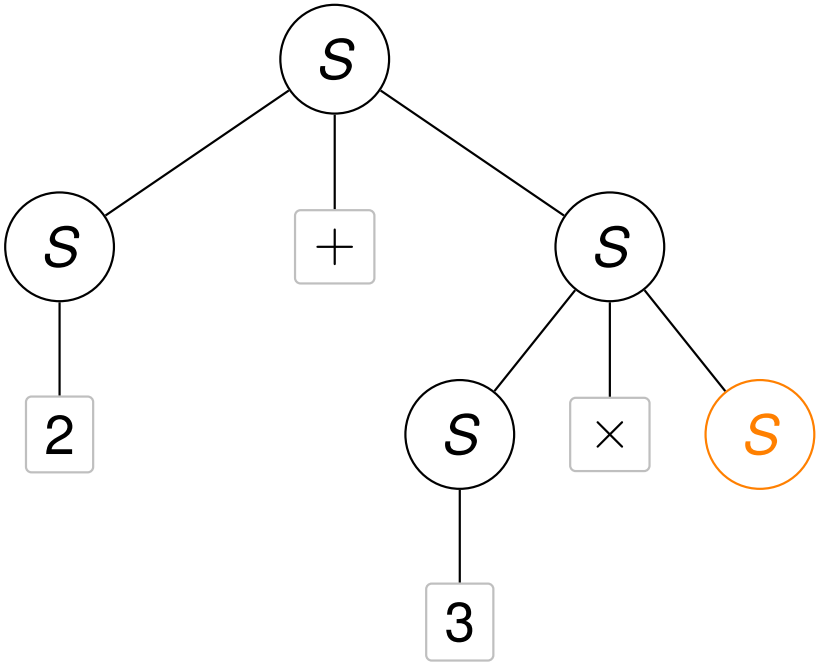
Dérivation et arbre syntaxique

Considérons la dérivation

$$\begin{array}{lllllll} & \underline{S} & & \rightarrow & \underline{S} + S & \rightarrow & 2 + \underline{S} \\ \rightarrow & 2 + \underline{S} \times S & & \rightarrow & 2 + 3 \times \underline{S} & \rightarrow & 2 + 3 \times 5 \end{array}$$

On peut lui associer un arbre syntaxique de la façon suivante :

$$\begin{array}{lll} \underline{S} & \rightarrow & \underline{S} + S \\ & \rightarrow & 2 + \underline{S} \\ & \rightarrow & 2 + \underline{S} \times S \\ & \rightarrow & 2 + 3 \times \underline{S} \end{array}$$



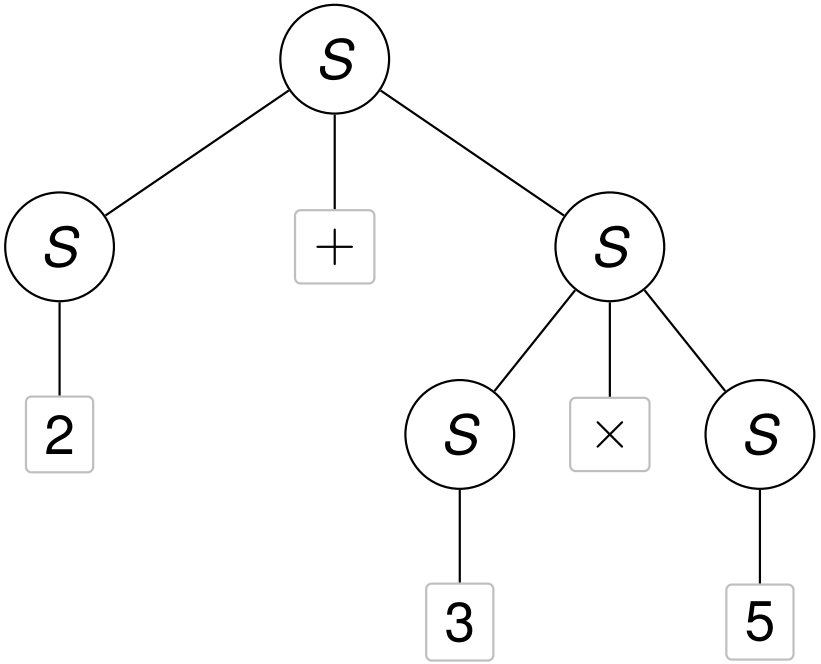
Dérivation et arbre syntaxique

Considérons la dérivation

$$\begin{array}{lllllll} & \underline{S} & \rightarrow & \underline{S} + S & \rightarrow & 2 + \underline{S} \\ \rightarrow & 2 + \underline{S} \times S & \rightarrow & 2 + 3 \times \underline{S} & \rightarrow & 2 + 3 \times 5 \end{array}$$

On peut lui associer un arbre syntaxique de la façon suivante :

$$\begin{array}{lll} \underline{S} & \rightarrow & \underline{S} + S \\ & \rightarrow & 2 + \underline{S} \\ & \rightarrow & 2 + \underline{S} \times S \\ & \rightarrow & 2 + 3 \times \underline{S} \\ & \rightarrow & 2 + 3 \times 5 \end{array}$$



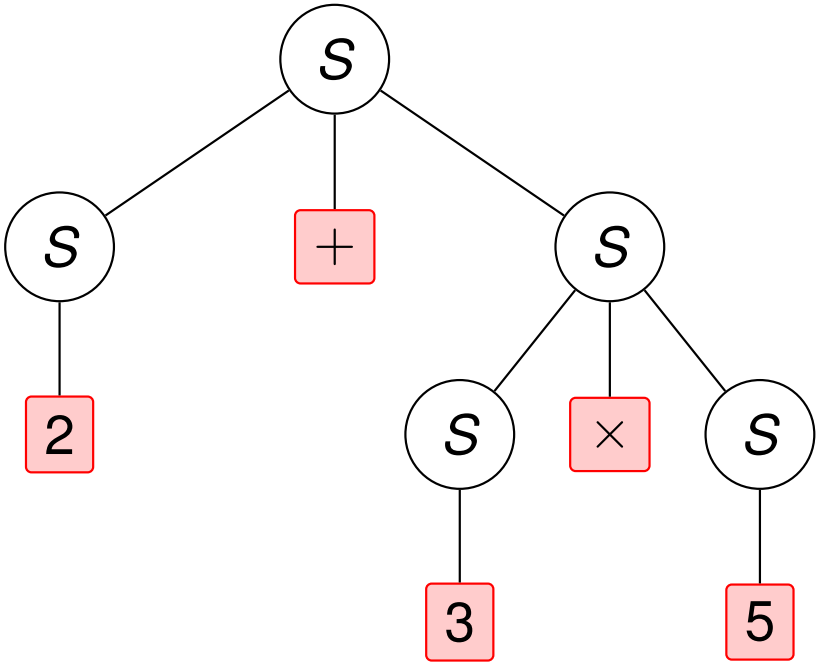
Dérivation et arbre syntaxique

Considérons la dérivation

$$\begin{array}{lllllll} & \underline{S} & \rightarrow & \underline{S} + S & \rightarrow & 2 + \underline{S} \\ \rightarrow & 2 + \underline{S} \times S & \rightarrow & 2 + 3 \times \underline{S} & \rightarrow & 2 + 3 \times 5 \end{array}$$

On peut lui associer un arbre syntaxique de la façon suivante :

$$\begin{array}{lll} \underline{S} & \rightarrow & \underline{S} + S \\ & \rightarrow & 2 + \underline{S} \\ & \rightarrow & 2 + \underline{S} \times S \\ & \rightarrow & 2 + 3 \times \underline{S} \\ & \rightarrow & 2 + 3 \times 5 \end{array}$$



Problème lié aux dérivations

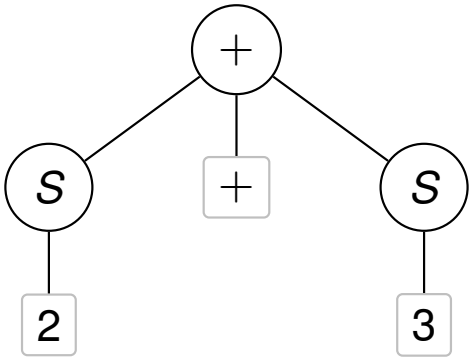
1 dérivation $\rightsquigarrow$ 1 arbre

Problème lié aux dérivations

1 dérivation $\rightsquigarrow$ 1 arbre

Leftmost

$$\underline{S} \rightarrow \underline{S} + S \rightarrow 2 + \underline{S} \rightarrow 2 + 3$$



Rightmost

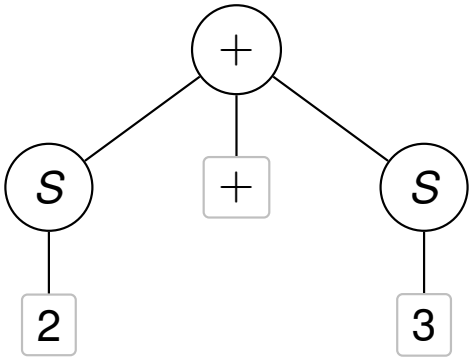
$$\underline{S} \rightarrow S + \underline{S} \rightarrow \underline{S} + 3 \rightarrow 2 + 3$$

Problème lié aux dérivations

1 dérivation $\rightsquigarrow$ 1 arbre

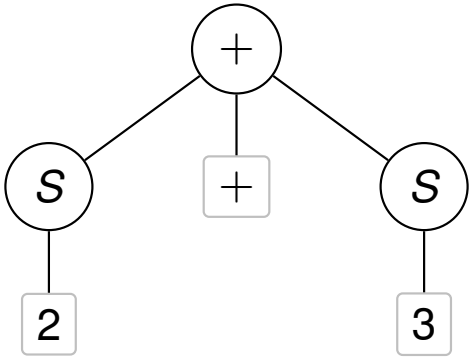
Leftmost

$$\underline{S} \rightarrow \underline{S} + S \rightarrow 2 + \underline{S} \rightarrow 2 + 3$$



Rightmost

$$\underline{S} \rightarrow S + \underline{S} \rightarrow \underline{S} + 3 \rightarrow 2 + 3$$

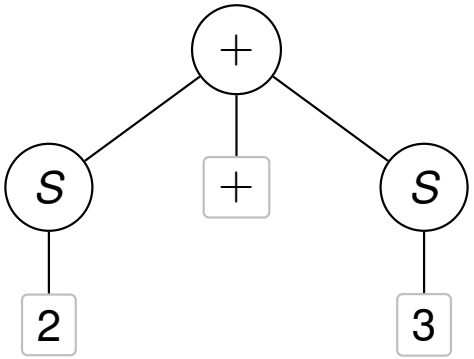


Problème lié aux dérivations

1 dérivation $\rightsquigarrow$ 1 arbre

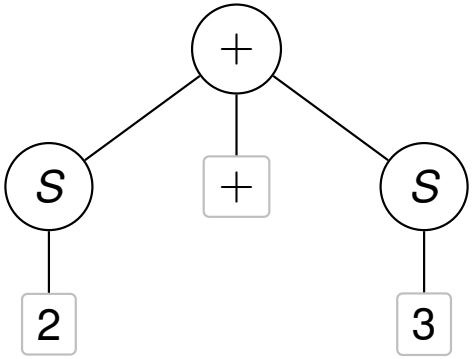
Leftmost

$$\underline{S} \rightarrow \underline{S} + S \rightarrow 2 + \underline{S} \rightarrow 2 + 3$$



Rightmost

$$\underline{S} \rightarrow S + \underline{S} \rightarrow \underline{S} + 3 \rightarrow 2 + 3$$



Choix de la stratégie :

- utile pour limiter l'espace de recherche d'une dérivation
- aucun impact sur l'arbre syntaxique donc pas de pb.

Problème lié aux dérivations (2)

Rappel de la grammaire : $S \rightarrow S + S \mid S \times S \mid 0 \mid 1 \mid \ldots \mid 9$

Génération de $2 + 3 \times 5$?

Problème lié aux dérivations (2)

Rappel de la grammaire : $S \rightarrow S + S \mid S \times S \mid 0 \mid 1 \mid \ldots \mid 9$

Génération de $2 + 3 \times 5$?

Plusieurs dérivations possibles à stratégie fixée :

- Leftmost avec $+$ en premier

$$S \rightarrow \underline{S} + S \rightarrow 2 + \underline{S} \rightarrow 2 + \underline{S} \times S \rightarrow 2 + 3 \times \underline{S} \rightarrow 2 + 3 \times 5$$

- Leftmost avec $\times$ en premier

$$S \rightarrow \underline{S} \times S \rightarrow \underline{S} + S \times S \rightarrow 2 + \underline{S} \times S \rightarrow 2 + 3 \times \underline{S} \rightarrow 2 + 3 \times 5$$

Problème lié aux dérivations (3)

$+$ en premier

$S \rightarrow \underline{S} + S \rightarrow 2 + \underline{S} \rightarrow 2 + \underline{S} \times S \rightarrow$
$2 + 3 \times \underline{S} \rightarrow 2 + 3 \times 5$

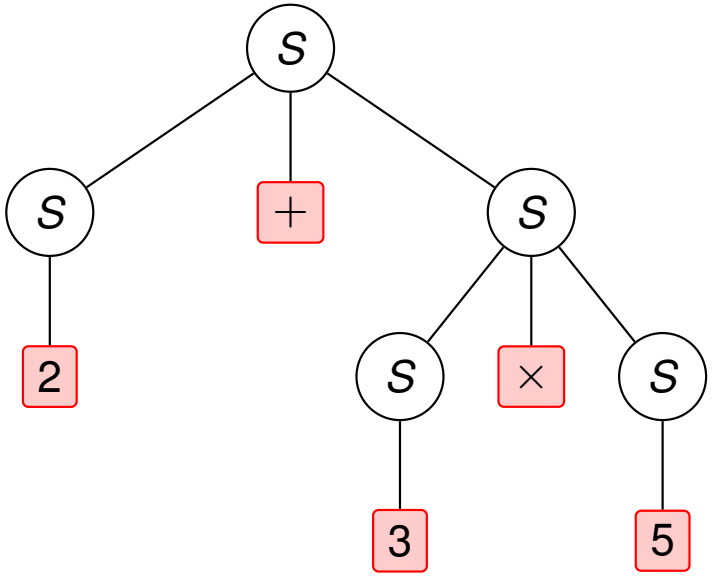
$\times$ en premier

$S \rightarrow \underline{S} \times S \rightarrow \underline{S} + S \times S \rightarrow$
$2 + \underline{S} \times S \rightarrow 2 + 3 \times \underline{S} \rightarrow 2 + 3 \times 5$

Problème lié aux dérivations (3)

$+$ en premier

$S \rightarrow \underline{S} + S \rightarrow 2 + \underline{S} \rightarrow 2 + \underline{S} \times S \rightarrow$
$2 + 3 \times \underline{S} \rightarrow 2 + 3 \times 5$



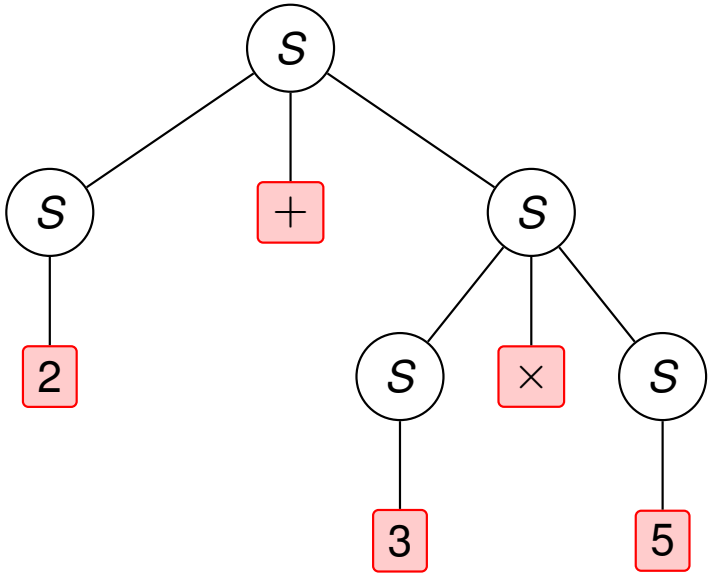
$\times$ en premier

$S \rightarrow \underline{S} \times S \rightarrow \underline{S} + S \times S \rightarrow$
$2 + \underline{S} \times S \rightarrow 2 + 3 \times \underline{S} \rightarrow 2 + 3 \times 5$

Problème lié aux dérivations (3)

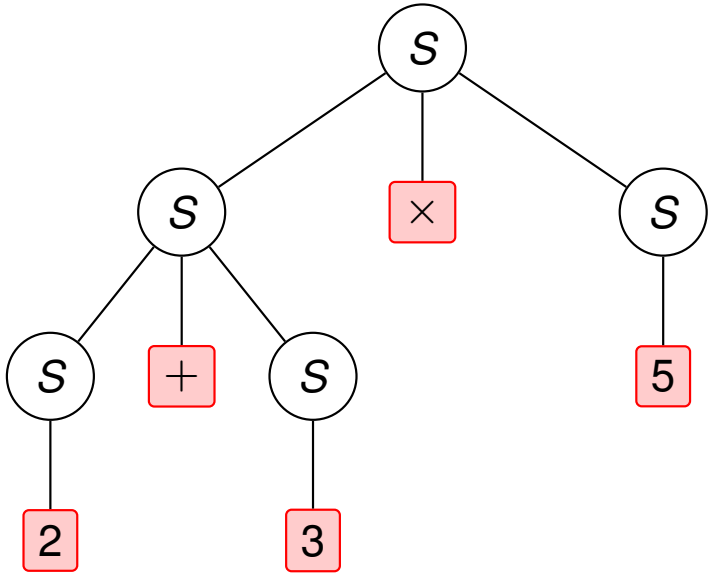
$+$ en premier

$S \to \underline{S} + S \to 2 + \underline{S} \to 2 + \underline{S} \times S \to$
$2 + 3 \times \underline{S} \to 2 + 3 \times 5$



$\times$ en premier

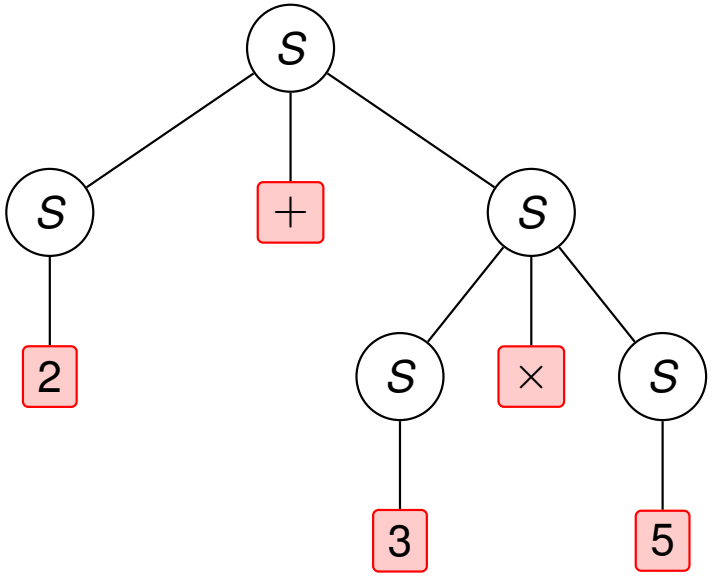
$S \to \underline{S} \times S \to \underline{S} + S \times S \to$
$2 + \underline{S} \times S \to 2 + 3 \times \underline{S} \to 2 + 3 \times 5$



Problème lié aux dérivations (3)

$+$ en premier

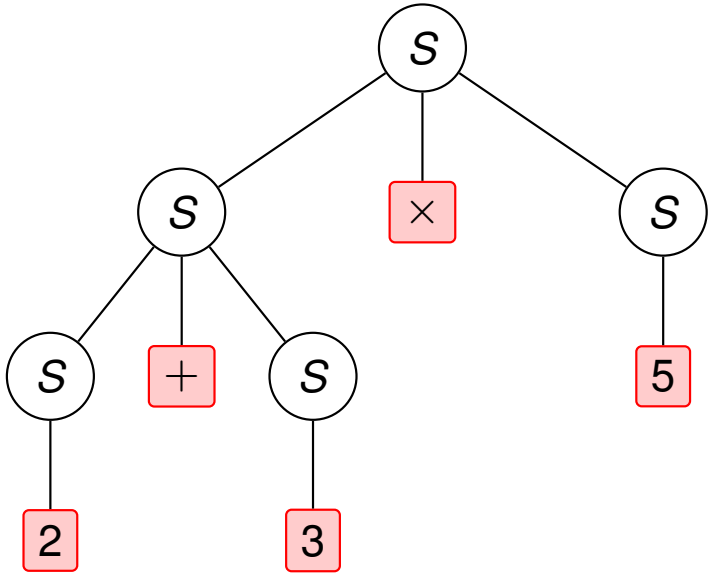
$S \to \underline{S} + S \to 2 + \underline{S} \to 2 + \underline{S} \times S \to$
$2 + 3 \times \underline{S} \to 2 + 3 \times 5$



Résultat du calcul = 17

$\times$ en premier

$S \to \underline{S} \times S \to \underline{S} + S \times S \to$
$2 + \underline{S} \times S \to 2 + 3 \times \underline{S} \to 2 + 3 \times 5$

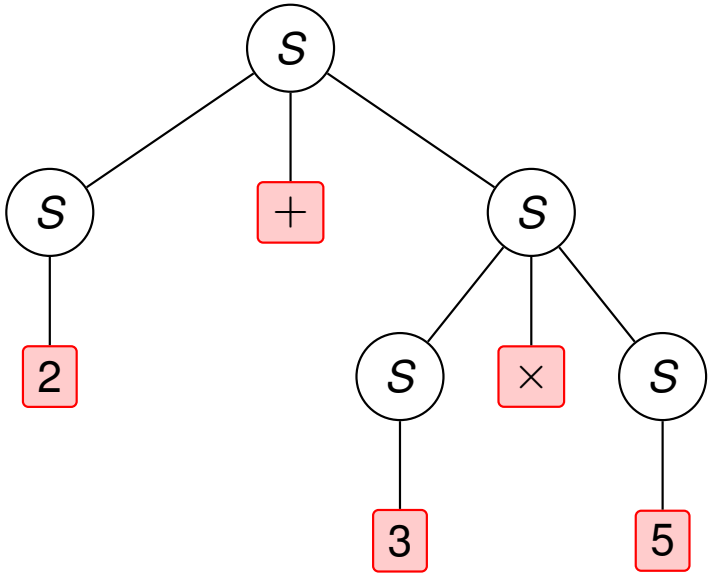


Résultat du calcul = 25

Problème lié aux dérivations (3)

$+$ en premier

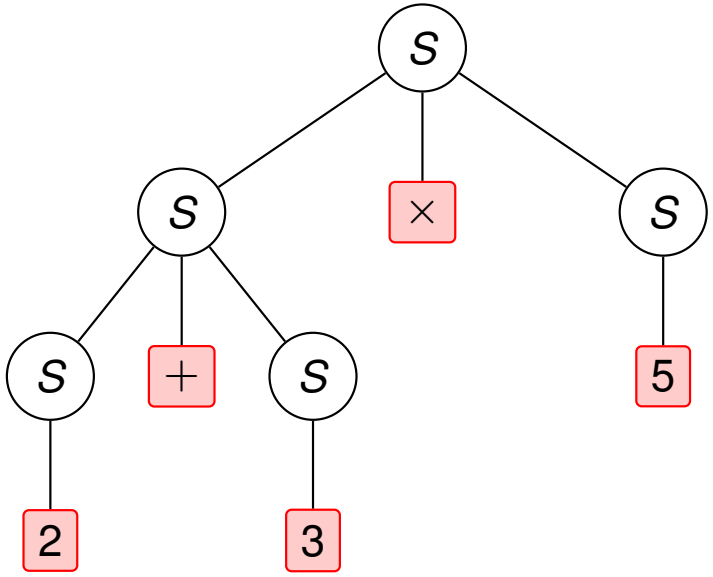
$S \to \underline{S} + S \to 2 + \underline{S} \to 2 + \underline{S} \times S \to$
$2 + 3 \times \underline{S} \to 2 + 3 \times 5$



Résultat du calcul = 17

$\times$ en premier

$S \to \underline{S} \times S \to \underline{S} + S \times S \to$
$2 + \underline{S} \times S \to 2 + 3 \times \underline{S} \to 2 + 3 \times 5$



Résultat du calcul = 25

$\rightsquigarrow$ résultat dépendant de la dérivation

Grammaires ambiguës

Définition

On dit qu'une grammaire $\mathcal{G}$ est ambiguë s'il existe un mot $w \in L(G)$ admettant plusieurs dérivations le plus à gauche possible différentes.

Grammaires ambiguës

Définition

On dit qu'une grammaire $\mathcal{G}$ est ambiguë s'il existe un mot $w \in L(G)$ admettant plusieurs dérivations le plus à gauche possible différentes.

- w admet plusieurs arbres syntaxiques différents
- On peut parfois transformer une grammaire ambiguë en une grammaire non-ambiguë équivalente :

$$S \rightarrow S + S \mid S \times S \mid C(n) \qquad \rightsquigarrow$$

Grammaires ambiguës

Définition

On dit qu'une grammaire $\mathcal{G}$ est ambiguë s'il existe un mot $w \in L(G)$ admettant plusieurs dérivations le plus à gauche possible différentes.

- w admet plusieurs arbres syntaxiques différents
- On peut parfois transformer une grammaire ambiguë en une grammaire non-ambiguë équivalente :

$$S \to S + S \mid S \times S \mid C(n) \quad \rightsquigarrow \quad \begin{array}{l} S \to S + T \mid T \\ T \to T \times C(n) \mid C(n) \end{array}$$

Grammaires ambiguës

Définition

On dit qu'une grammaire $\mathcal{G}$ est ambiguë s'il existe un mot $w \in L(G)$ admettant plusieurs dérivations le plus à gauche possible différentes.

- w admet plusieurs arbres syntaxiques différents
- On peut parfois transformer une grammaire ambiguë en une grammaire non-ambiguë équivalente :

$$S \to S + S \mid S \times S \mid C(n) \quad \rightsquigarrow \quad \begin{array}{l} S \to S + T \mid T \\ T \to T \times C(n) \mid C(n) \end{array}$$

- Il existe des langages intrinsèquement ambigus :
  $L(G) = L$ intrinsèquement ambigu $\Rightarrow G$ ambiguë

Plan

1. Compilation : Vue d'ensemble
2. Analyse lexicale
   - Reconnaissance des lexèmes
   - Passage à la pratique avec `flex`
3. Analyse syntaxique
   - Utilisation de `bison`
   - Dérivations et arbres syntaxiques
   - Analyse ascendante
   - Analyse de type LR(0)
   - Résolution de conflits

Complément au cours de LASF

Grammaire pour les sommes en notation polonaise inversée :

$$S \rightarrow \texttt{eol} \mid E\ \texttt{eol}$$
$$E \rightarrow \texttt{num} \mid E\ E\ \texttt{+}$$

où

- `num` est un lexème représentant un entier
- `eol` représente le symbole de fin de ligne

Reconnaissance de ces sommes ?

- par analyse descendante — pas terrible ici
- par analyse ascendante

Complément au cours de LASF (2)

Reconnaissance de `1 2 + eol` par analyse ascendante

départ	• `1 2 + eol`
→ lecture de 1	`1` • `2 + eol`
→ 1 forcément produit via `E` → `num`	`E` • `2 + eol`

Complément au cours de LASF (2)

Reconnaissance de `1 2 + eol` par analyse ascendante

	départ	• 1 2 + eol
→	lecture de 1	1 • 2 + eol
→	1 forcément produit via `E → num`	E • 2 + eol
→	impossible de réduire `E`, donc lecture de 2	E 2 • + eol
→	2 forcément produit via `E → num`	E E • + eol

Complément au cours de LASF (2)

Reconnaissance de `1 2 + eol` par analyse ascendante

	départ	• 1 2 + eol
→	lecture de 1	1 • 2 + eol
→	1 forcément produit via `E` → `num`	E • 2 + eol
→	impossible de réduire `E`, donc lecture de 2	E 2 • + eol
→	2 forcément produit via `E` → `num`	E E • + eol
→	impossible de réduire `E E`, donc lecture de +	E E + • eol

Complément au cours de LASF (2)

Reconnaissance de `1 2 + eol` par analyse ascendante

	départ	• 1 2 + eol
→	lecture de 1	1 • 2 + eol
→	1 forcément produit via `E → num`	E • 2 + eol
→	impossible de réduire `E`, donc lecture de 2	E 2 • + eol
→	2 forcément produit via `E → num`	E E • + eol
→	impossible de réduire `E E`, donc lecture de +	E E + • eol
→	réduction possible via `E → E E +`	E • eol

Complément au cours de LASF (2)

Reconnaissance de `1 2 + eol` par analyse ascendante

	départ	• 1 2 + eol
→	lecture de 1	1 • 2 + eol
→	1 forcément produit via `E → num`	E • 2 + eol
→	impossible de réduire `E`, donc lecture de 2	E 2 • + eol
→	2 forcément produit via `E → num`	E E • + eol
→	impossible de réduire `E E`, donc lecture de +	E E + • eol
→	réduction possible via `E → E E +`	E • eol
→	impossible de réduire, donc lecture de `eol`	E eol •
→	réduction possible via `S → E eol`	S •
→	mot reconnu	

Complément au cours de LASF (3)

Exemple de séquence non reconnue : `1 + eol`

	départ	• 1 + eol
→	lecture de 1	1 • + eol
→	1 forcément produit via `E` → `num`	E • + eol
→	impossible de réduire, donc lecture de +	E + • eol
→	impossible de réduire, donc lecture de `eol`	E + eol •
→	impossible de réduire	E + eol •
→	échec	

Séquence non reconnue car :

- résultat différent de `S` •
- aucune autre réduction possible

Shift et Reduce

Seulement deux types d'étapes dans l'analyse ascendante :

Shift =

- lecture (consommation) d'un lexème
- déplacement du marqueur • d'un cran vers la droite

Shift et Reduce

Seulement deux types d'étapes dans l'analyse ascendante :

Shift =

- lecture (consommation) d'un lexème
- déplacement du marqueur • d'un cran vers la droite

Reduce =

- réduction d'un morceau précédent immédiatement •
  - $\rightsquigarrow$ application d'une règle à l'envers
- pas de lecture, donc pas de déplacement de •

Plan

1. Compilation : Vue d'ensemble
2. Analyse lexicale
   - Reconnaissance des lexèmes
   - Passage à la pratique avec `flex`
3. Analyse syntaxique
   - Utilisation de `bison`
   - Dérivations et arbres syntaxiques
   - Analyse ascendante
   - Analyse de type LR(0)
   - Résolution de conflits

Vers une automatisation de l'analyse ascendante

Idée :

- construire un automate comme pour les regex
- utiliser une pile pour stocker ce qui précède •
  - $\rightarrow$ nécessaire pour les réductions

Vers une automatisation de l'analyse ascendante

Idée :

- construire un automate comme pour les regex
- utiliser une pile pour stocker ce qui précède •

  $\rightarrow$ nécessaire pour les réductions

États de l'automate = ensemble de règles avec marqueur •

ex : $\mathtt{E} \rightarrow \mathtt{E} \bullet \mathtt{E} \; +$

$\rightsquigarrow$ On vise à faire $\mathtt{E}$ à partir du $\mathtt{E}$ déjà lu et d'un futur $\mathtt{E} \; +$.

Vers une automatisation de l'analyse ascendante

Idée :

- construire un automate comme pour les regex
- utiliser une pile pour stocker ce qui précède •

  $\rightarrow$ nécessaire pour les réductions

États de l'automate = ensemble de règles avec marqueur •

ex : `E` $\rightarrow$ `E` • `E` `+`

$\rightsquigarrow$ On vise à faire `E` à partir du `E` déjà lu et d'un futur `E` `+`.

État initial = `Z` $\rightarrow$ • `S` `$` convention

$\rightsquigarrow$ On vise à lire exactement un `S` d'ici la fin de de la séquence (`$`).

Construction de l'analyseur LR(0)

Partir de l'état initial et répéter tant qu'il a des nouveaux états :

1. Compléter les nouveaux états =

   Si N <u>non-terminal</u> suit un • :

   - ajouter les règles N → ... on essaie de lire un *N*
   - mettre • juste après →

   Ex : S → • E eol ⇝ ajout de $\begin{cases} \texttt{E} \rightarrow \bullet\ \texttt{num} \\ \texttt{E} \rightarrow \bullet\ \texttt{E}\ \texttt{E}\ + \end{cases}$

Construction de l'analyseur LR(0)

Partir de l'état initial et répéter tant qu'il a des nouveaux états :

1. Compléter les nouveaux états =

   Si $\mathtt{N}$ <u>non-terminal</u> suit un $\bullet$:

   - ajouter les règles $\mathtt{N} \rightarrow \ldots$ — on essaie de lire un N
   - mettre $\bullet$ juste après $\rightarrow$

   Ex : $\mathtt{S} \rightarrow \bullet\ \mathtt{E}\ \mathtt{eol} \quad \rightsquigarrow \quad$ ajout de $\begin{cases} \mathtt{E} \rightarrow \bullet\ \mathtt{num} \\ \mathtt{E} \rightarrow \bullet\ \mathtt{E}\ \mathtt{E}\ + \end{cases}$

2. Ajouter les transitions sortant des nouveaux états

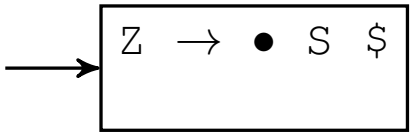
   Pour une transition avec x :

   - ne garder que les règles contenant $\bullet\ x$ — consommation du x
   - déplacer $\bullet$ après x

   Ex : Avec $x = \mathtt{E}$, $\mathtt{S} \rightarrow \bullet\ \mathtt{E}\ \mathtt{eol} \xrightarrow{\mathtt{E}} \mathtt{S} \rightarrow \mathtt{E}\ \bullet\ \mathtt{eol}$

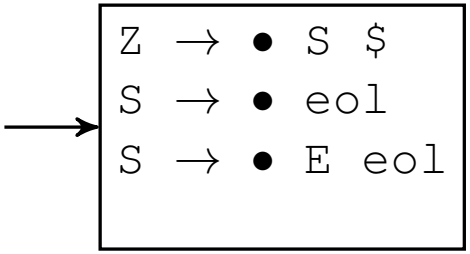
Construction de l'analyseur LR(0) – Exemple

Illustration sur les sommes en notation polonaise inversée



Construction de l'analyseur LR(0) – Exemple

Illustration sur les sommes en notation polonaise inversée



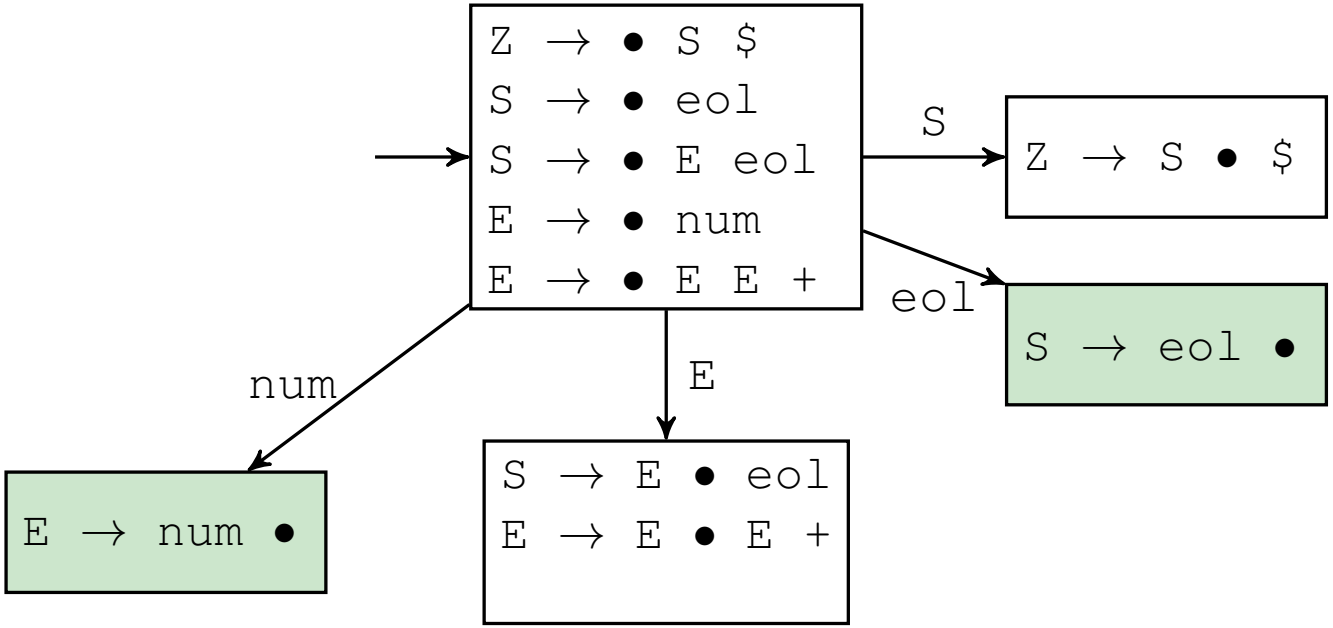
Construction de l'analyseur LR(0) – Exemple

Illustration sur les sommes en notation polonaise inversée

$$\longrightarrow \boxed{\begin{array}{l} \texttt{Z} \rightarrow \bullet\ \texttt{S}\ \texttt{\$} \\ \texttt{S} \rightarrow \bullet\ \texttt{eol} \\ \texttt{S} \rightarrow \bullet\ \texttt{E}\ \texttt{eol} \\ \texttt{E} \rightarrow \bullet\ \texttt{num} \\ \texttt{E} \rightarrow \bullet\ \texttt{E}\ \texttt{E}\ \texttt{+} \end{array}}$$

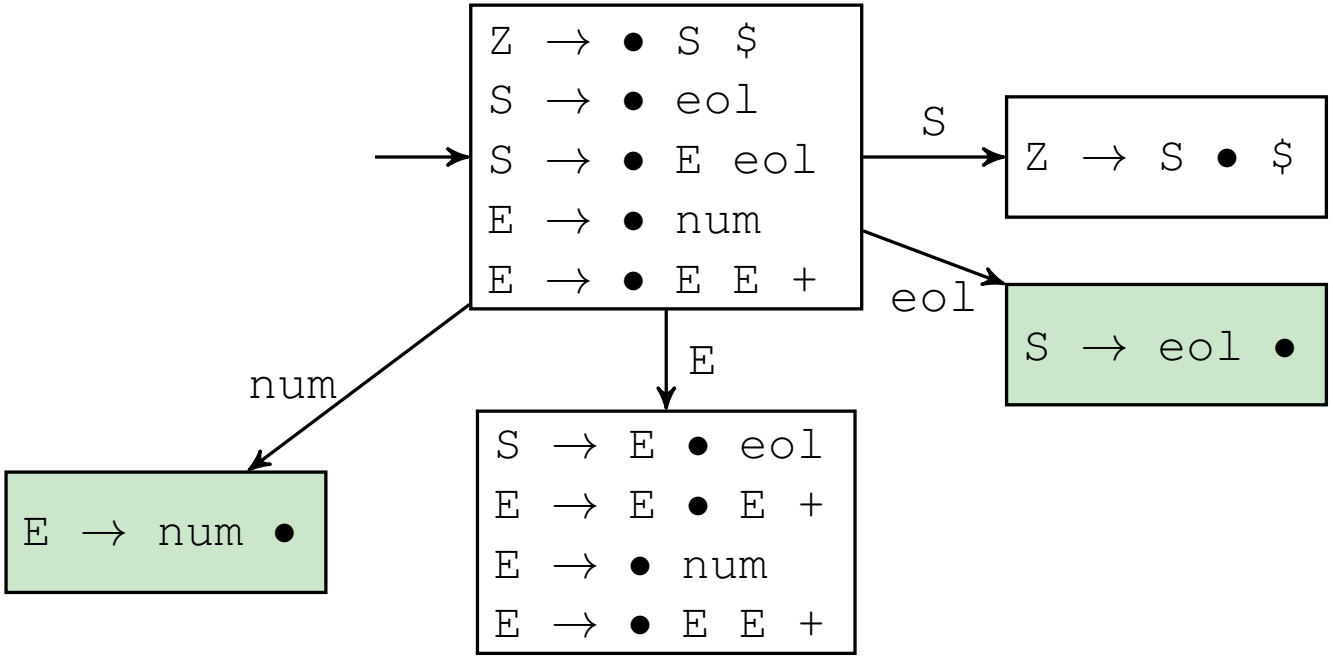
Construction de l'analyseur LR(0) – Exemple

Illustration sur les sommes en notation polonaise inversée



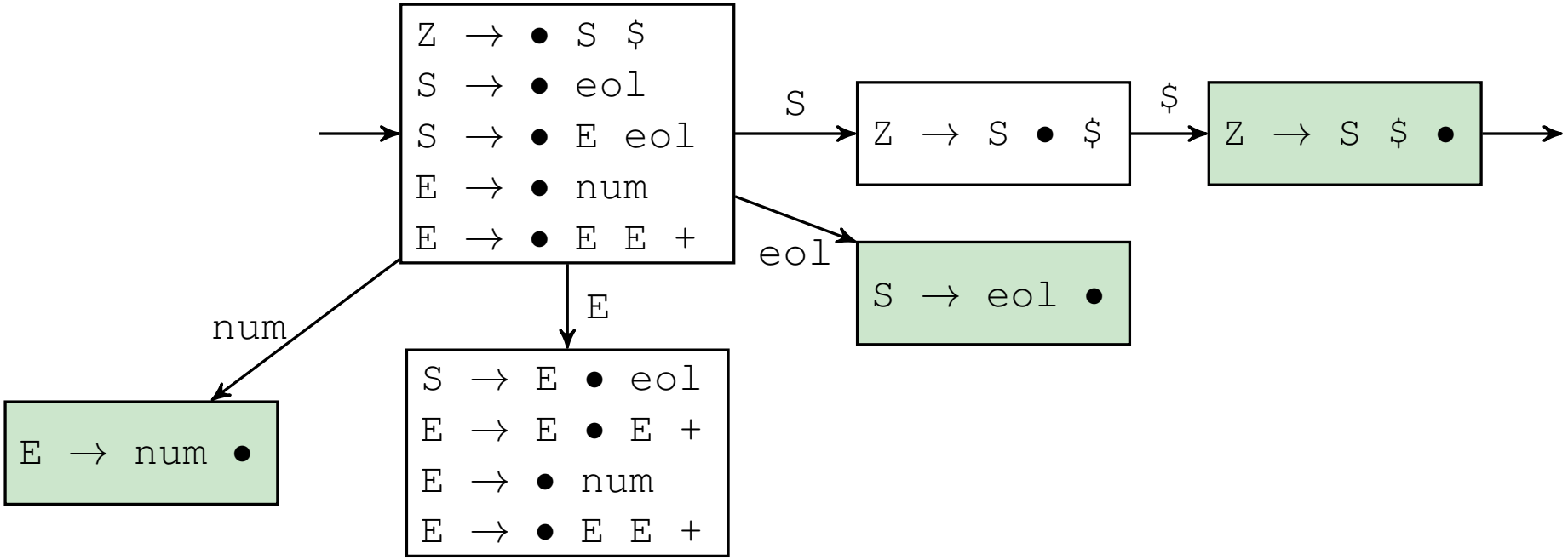
Construction de l'analyseur LR(0) – Exemple

Illustration sur les sommes en notation polonaise inversée



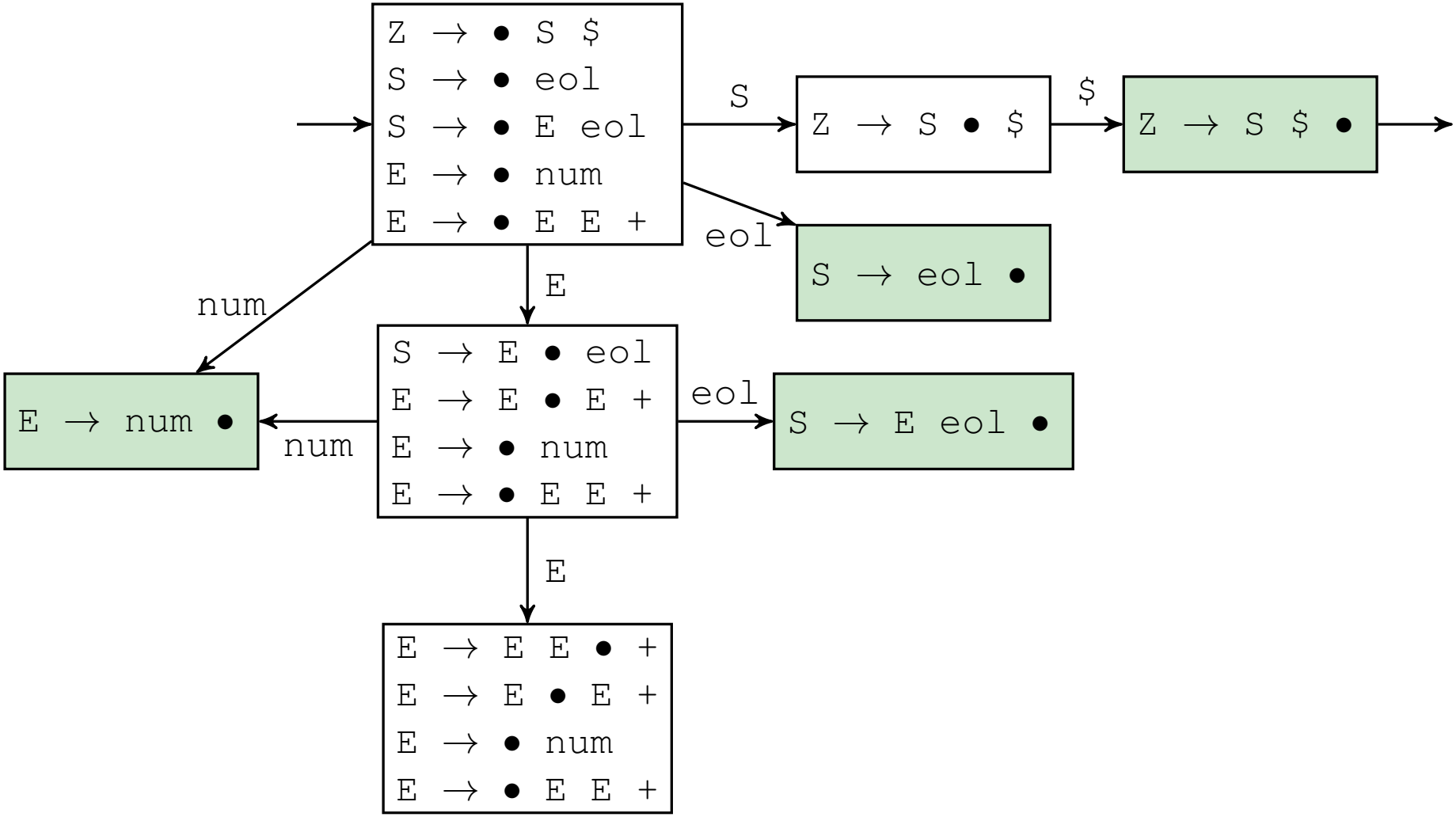
Construction de l'analyseur LR(0) – Exemple

Illustration sur les sommes en notation polonaise inversée



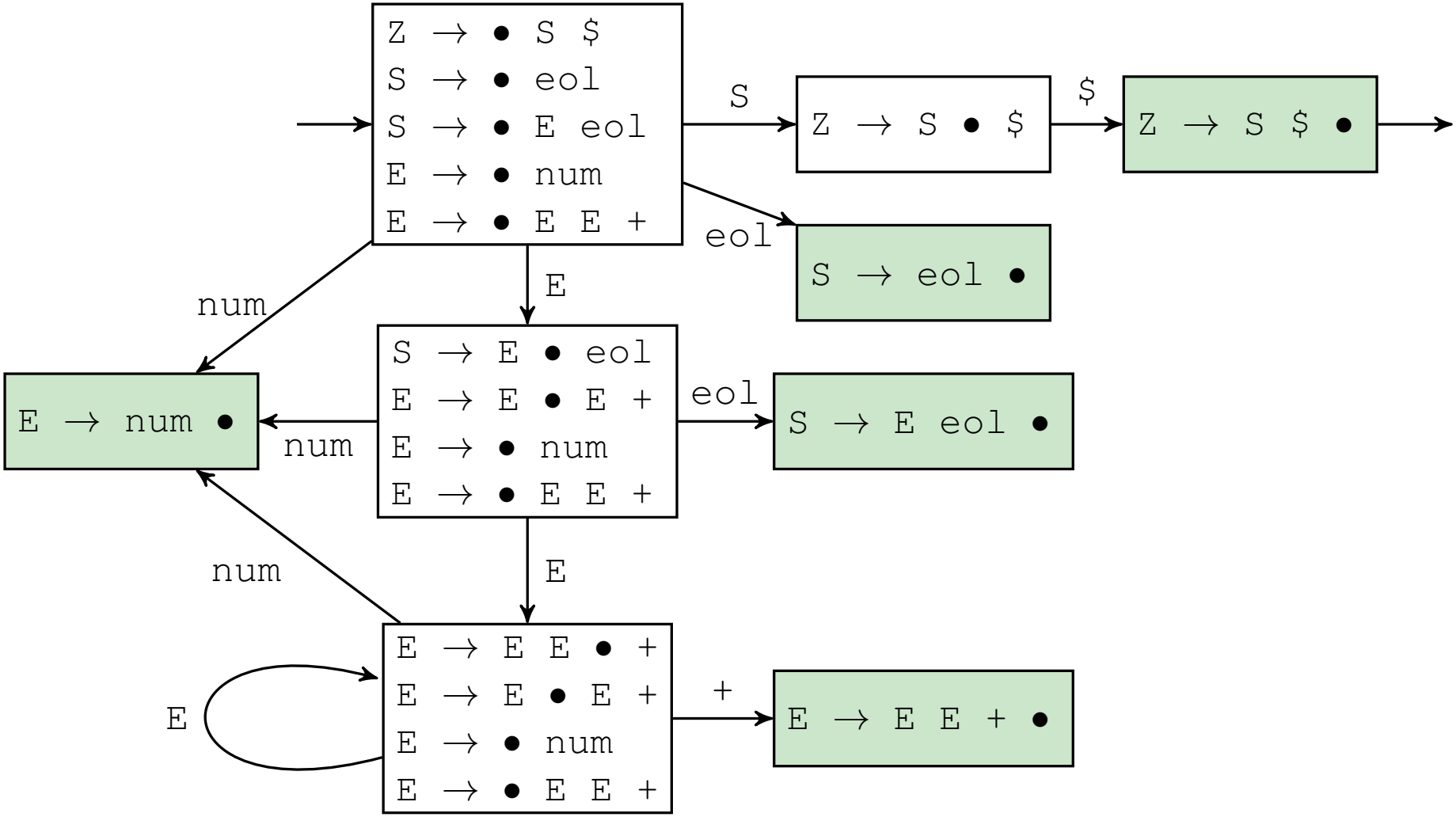
Construction de l'analyseur LR(0) – Exemple

Illustration sur les sommes en notation polonaise inversée



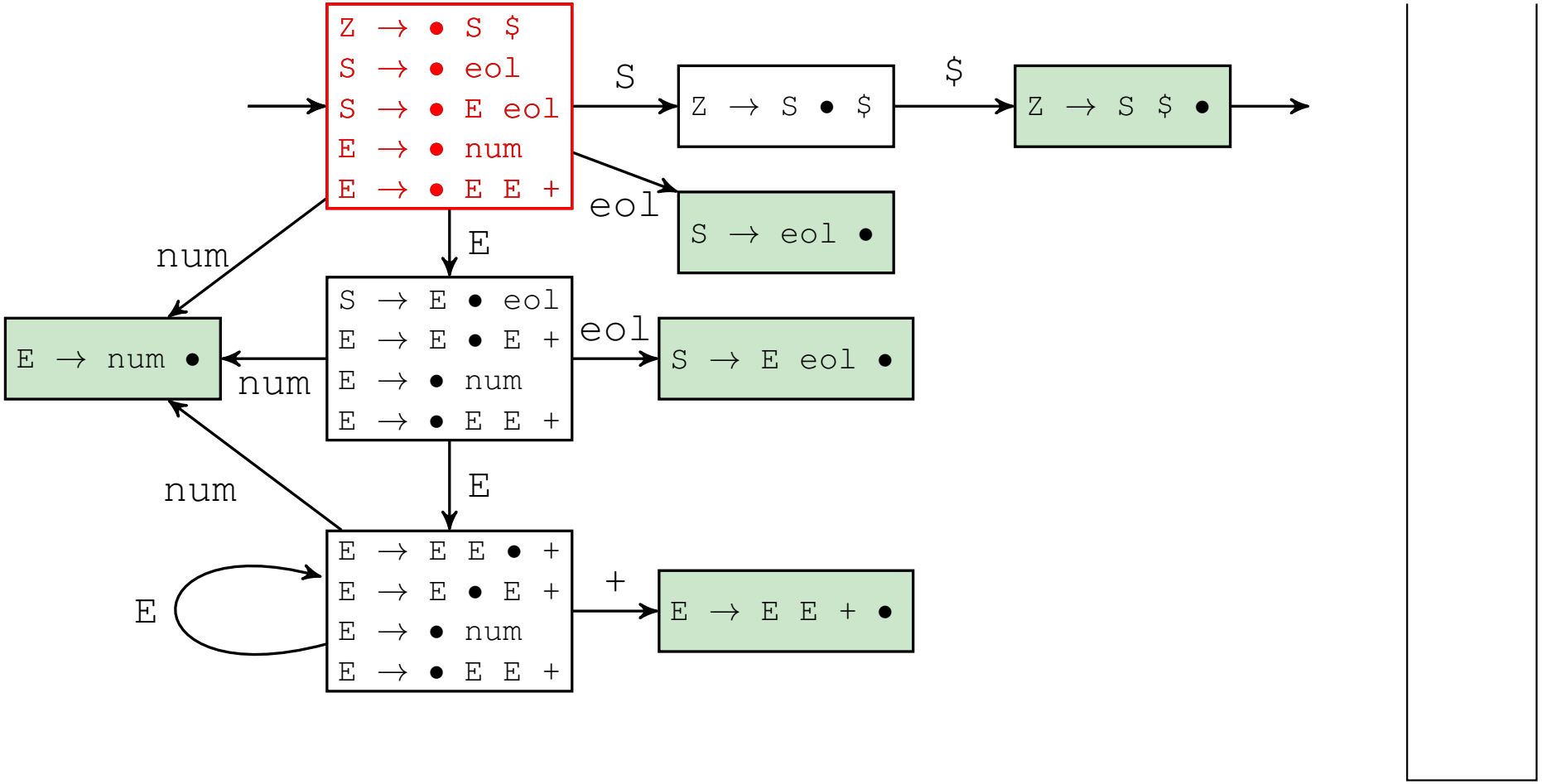
Construction de l'analyseur LR(0) – Exemple

Illustration sur les sommes en notation polonaise inversée



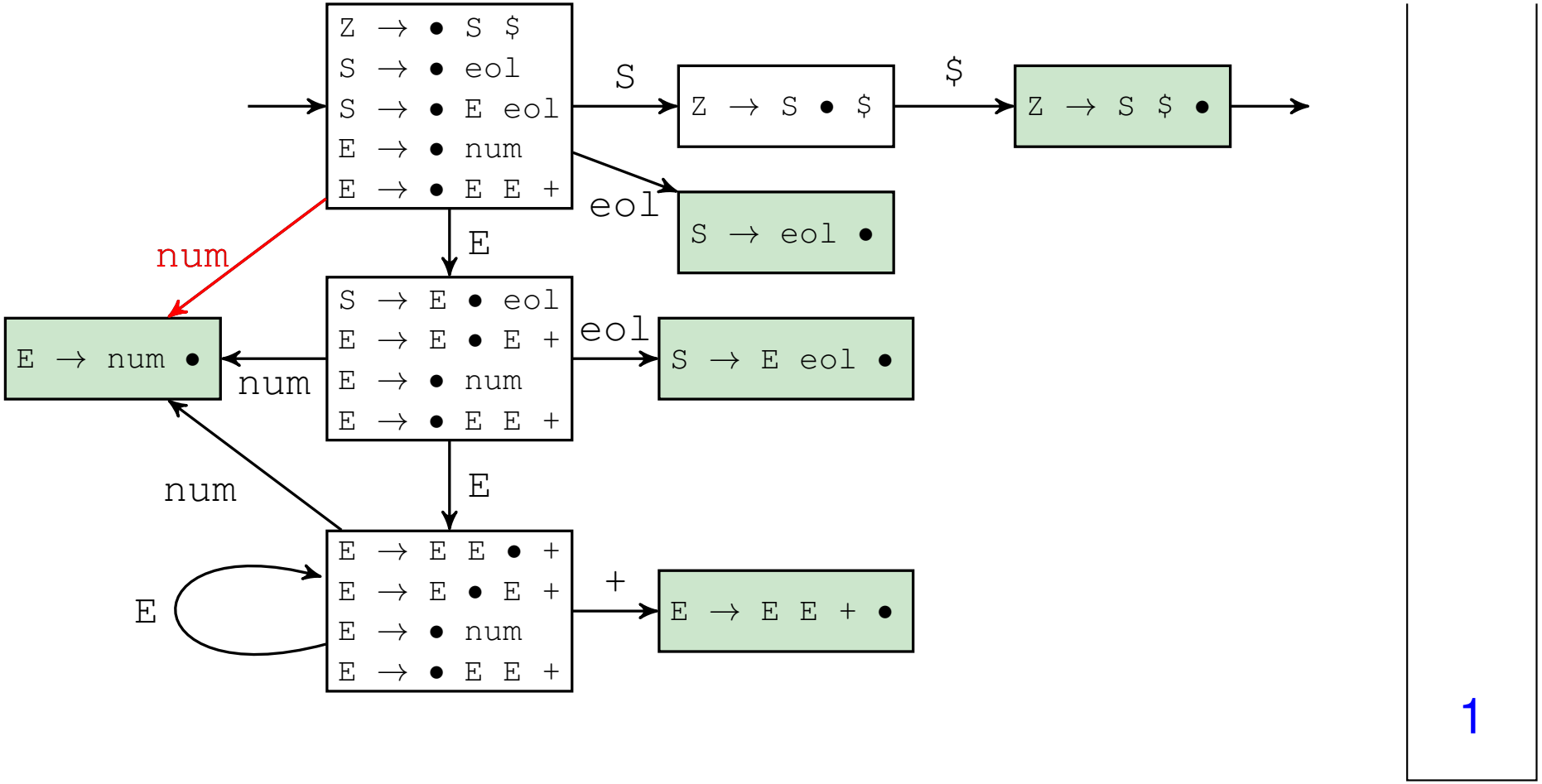
Exemple de reconnaissance

Départ : • 1 2 + eol $ pile vide



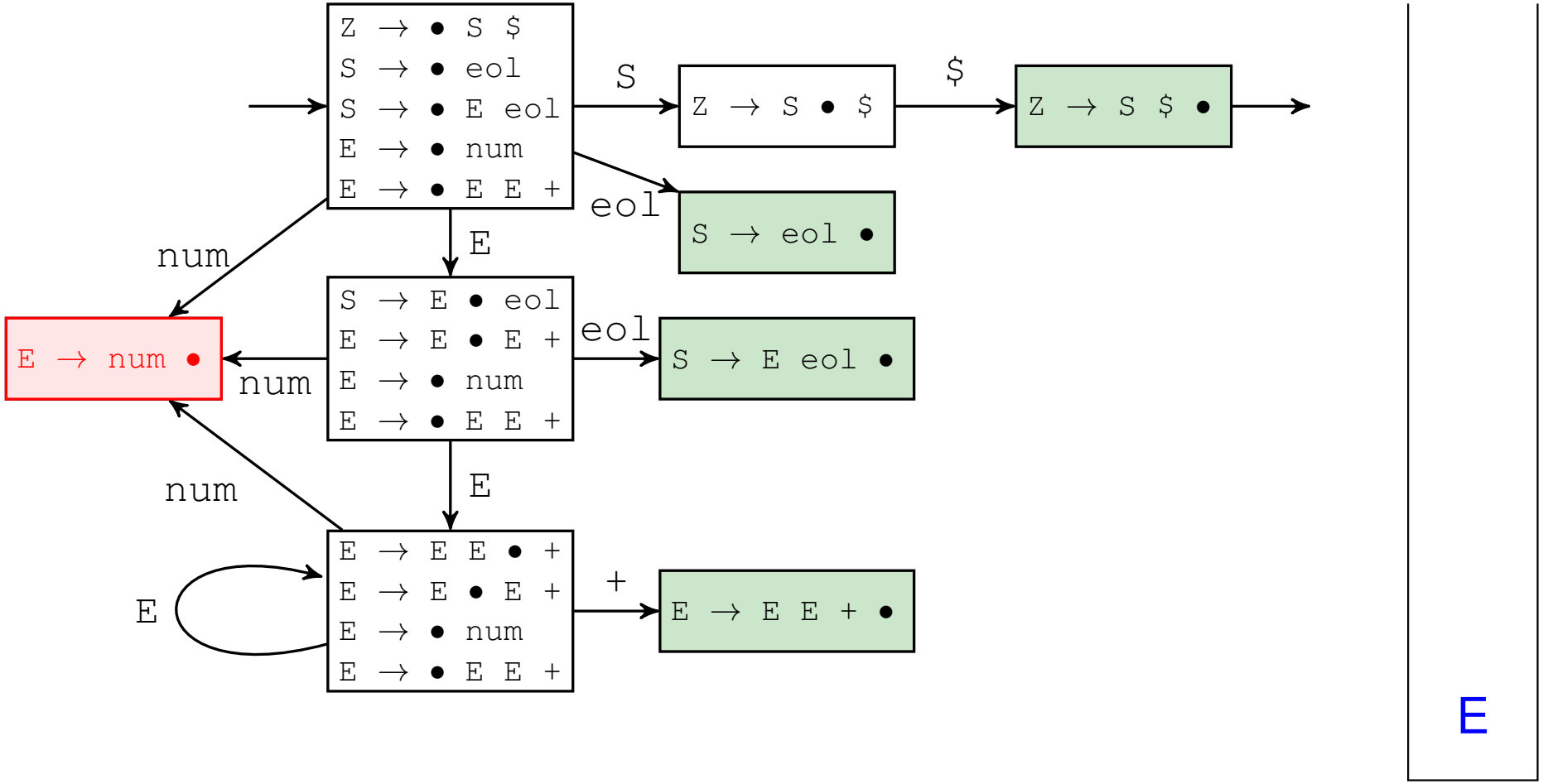
Exemple de reconnaissance

SHIFT : `1 • 2 + eol $` ajout de 1 sur la pile



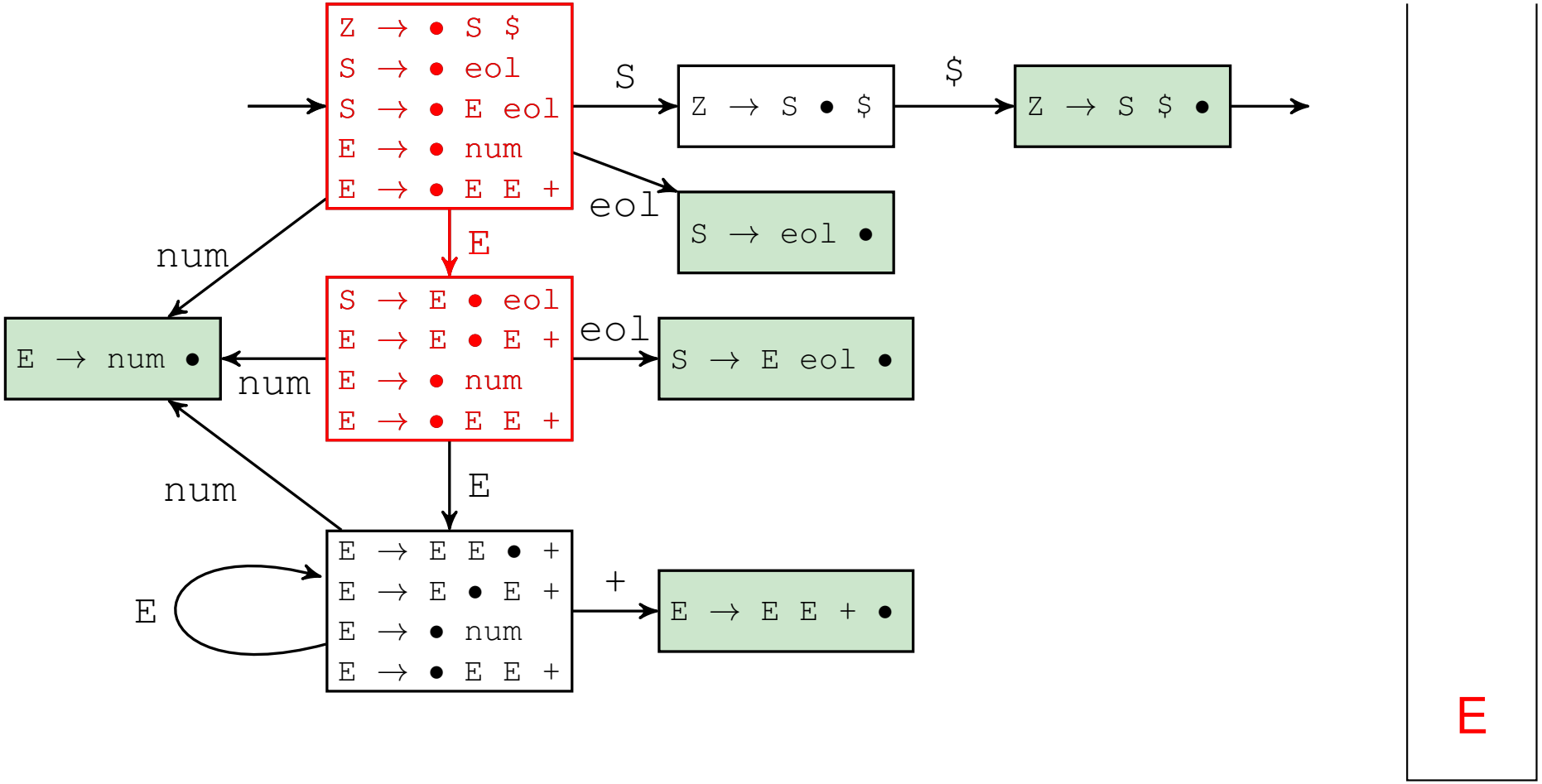
Exemple de reconnaissance

REDUCE : `E • 2 + eol $` 1 en haut de pile devient `E`



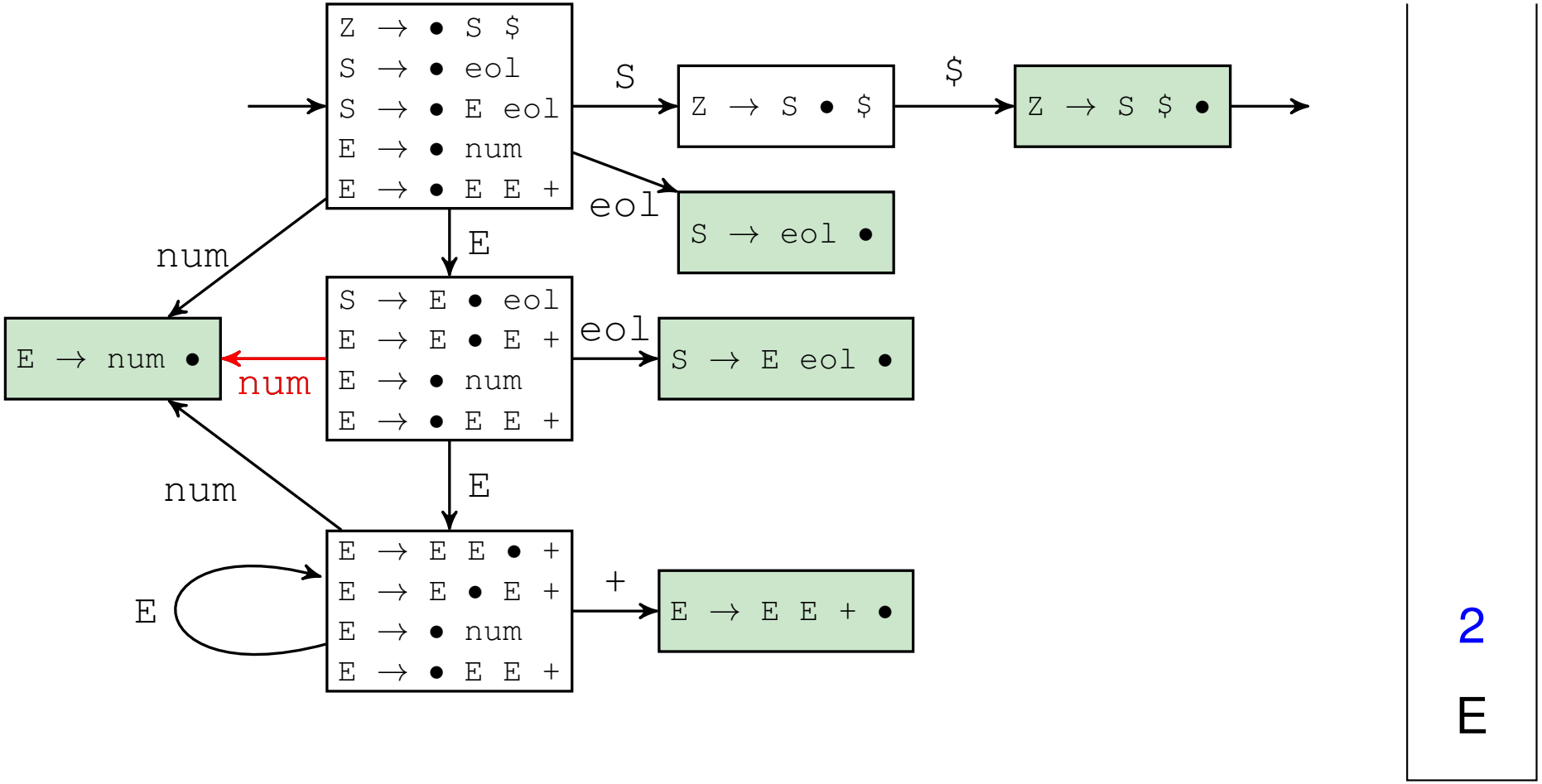
Exemple de reconnaissance

Départ : `E • 2 + eol $` placement en fonction de la pile



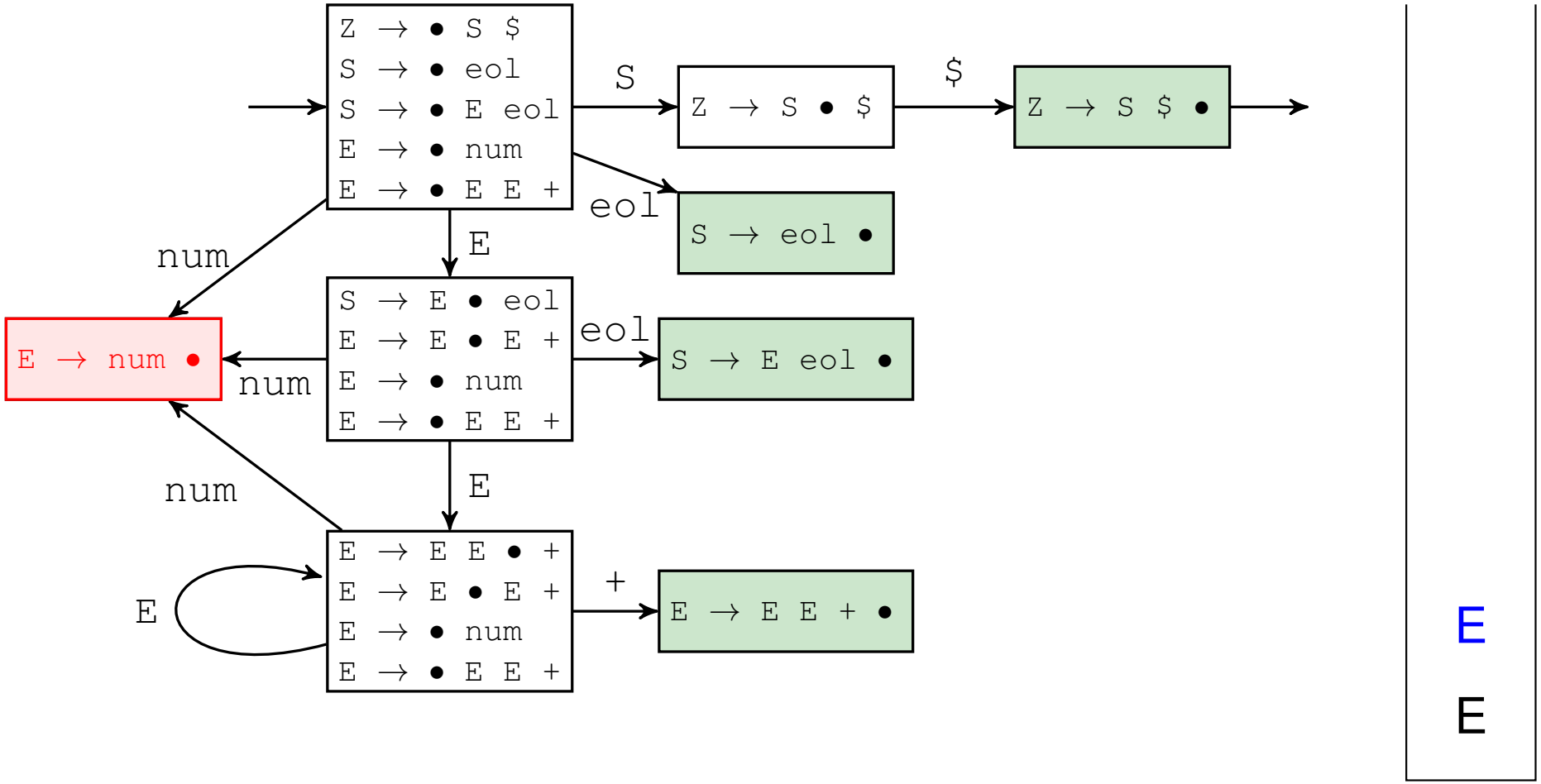
Exemple de reconnaissance

SHIFT : E 2 • + eol $ ajout de 1 sur la pile



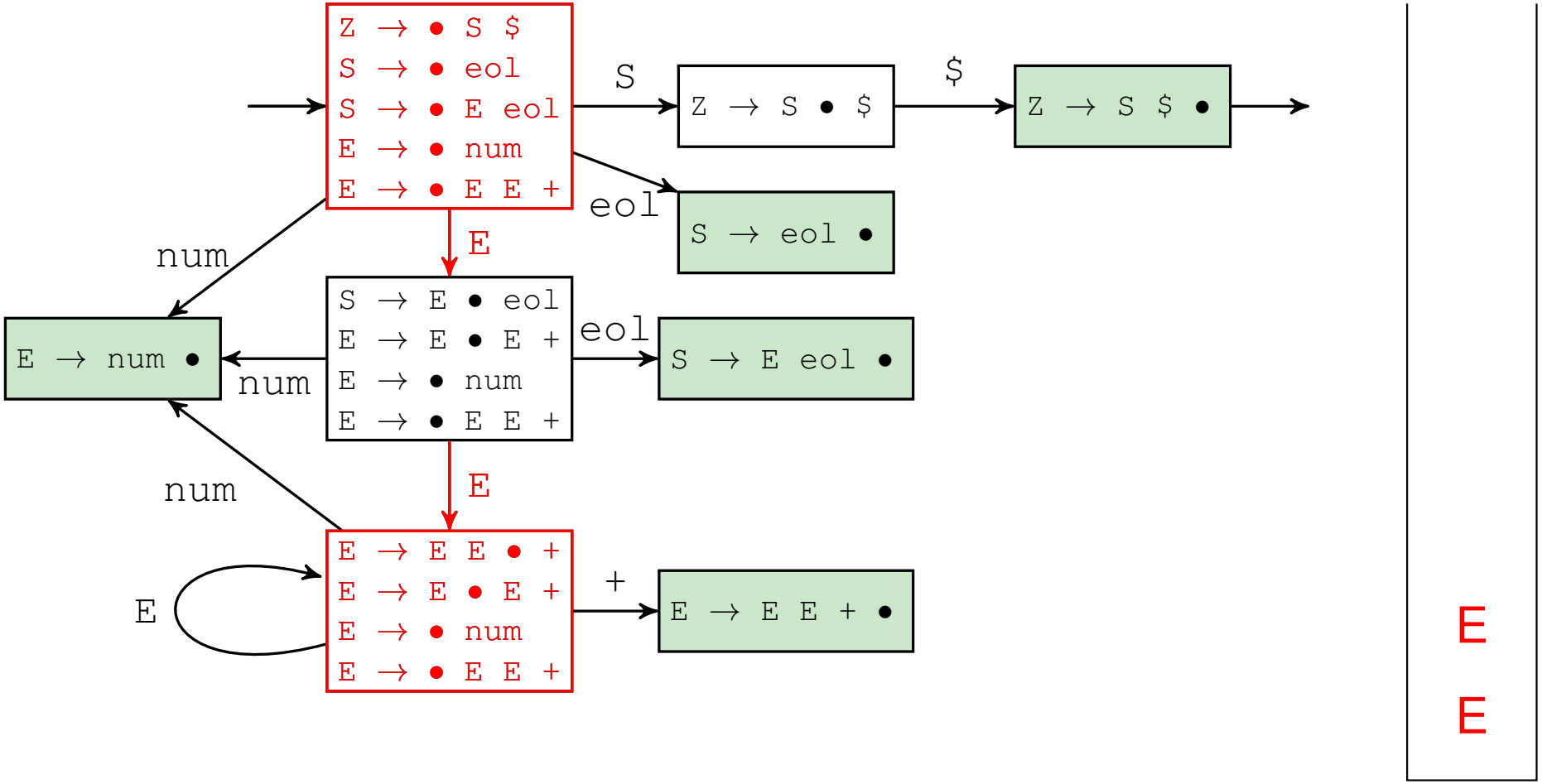
Exemple de reconnaissance

REDUCE : E E • eol $ 2 en haut de pile devient E



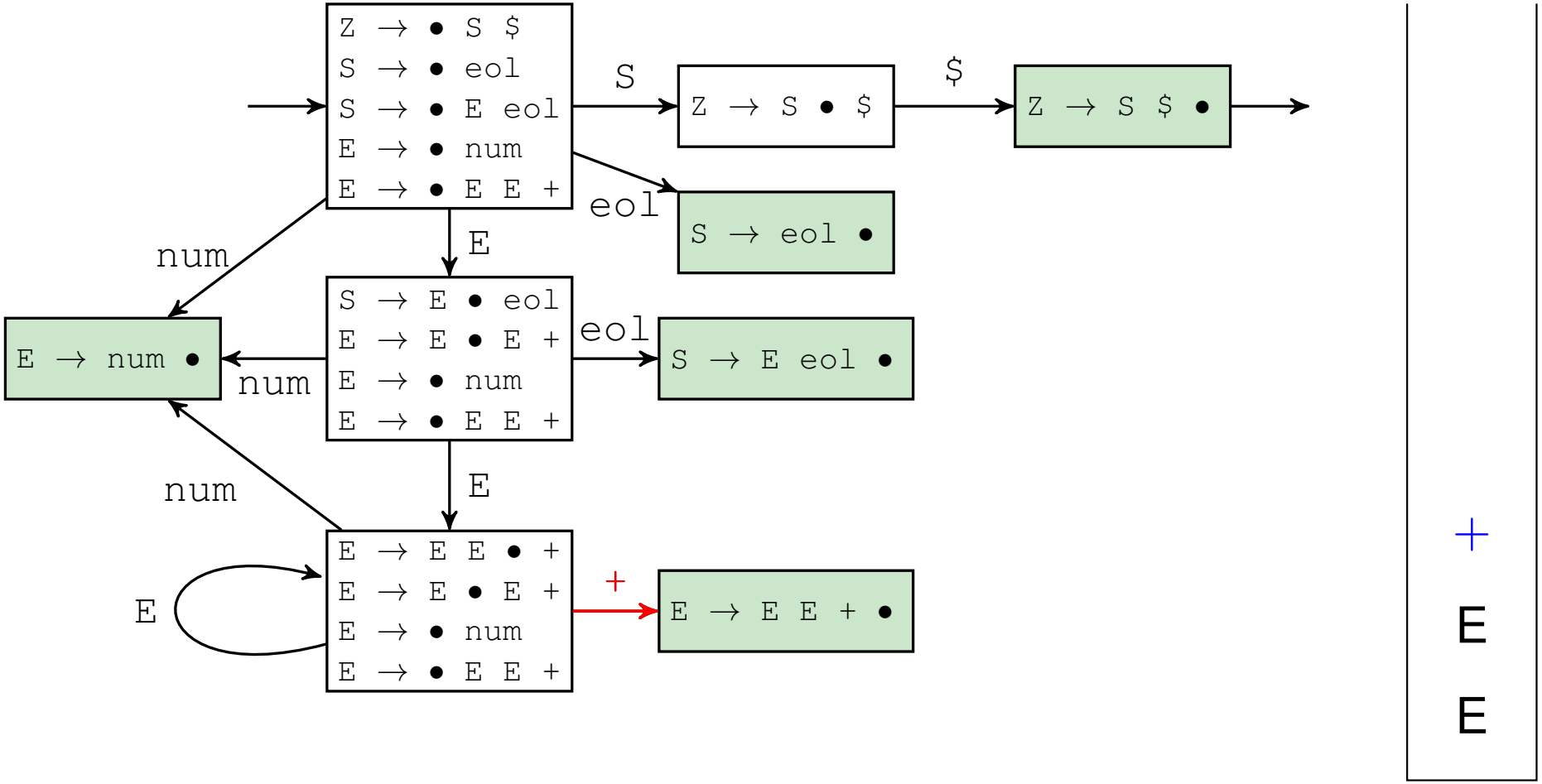
Exemple de reconnaissance

Départ : E E • + eol $ placement en fonction de la pile



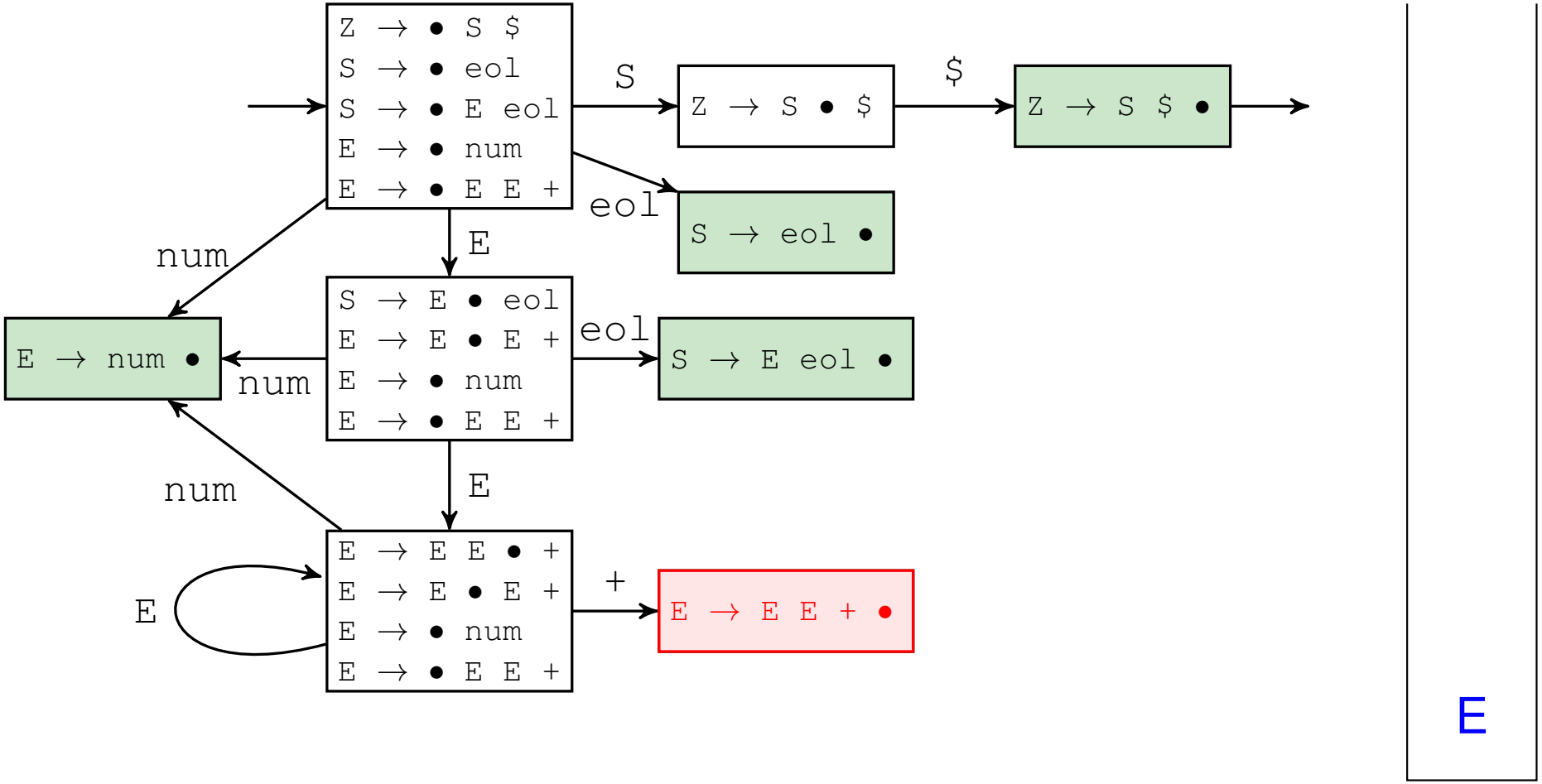
Exemple de reconnaissance

SHIFT : E E + • eol $ ajout de + sur la pile



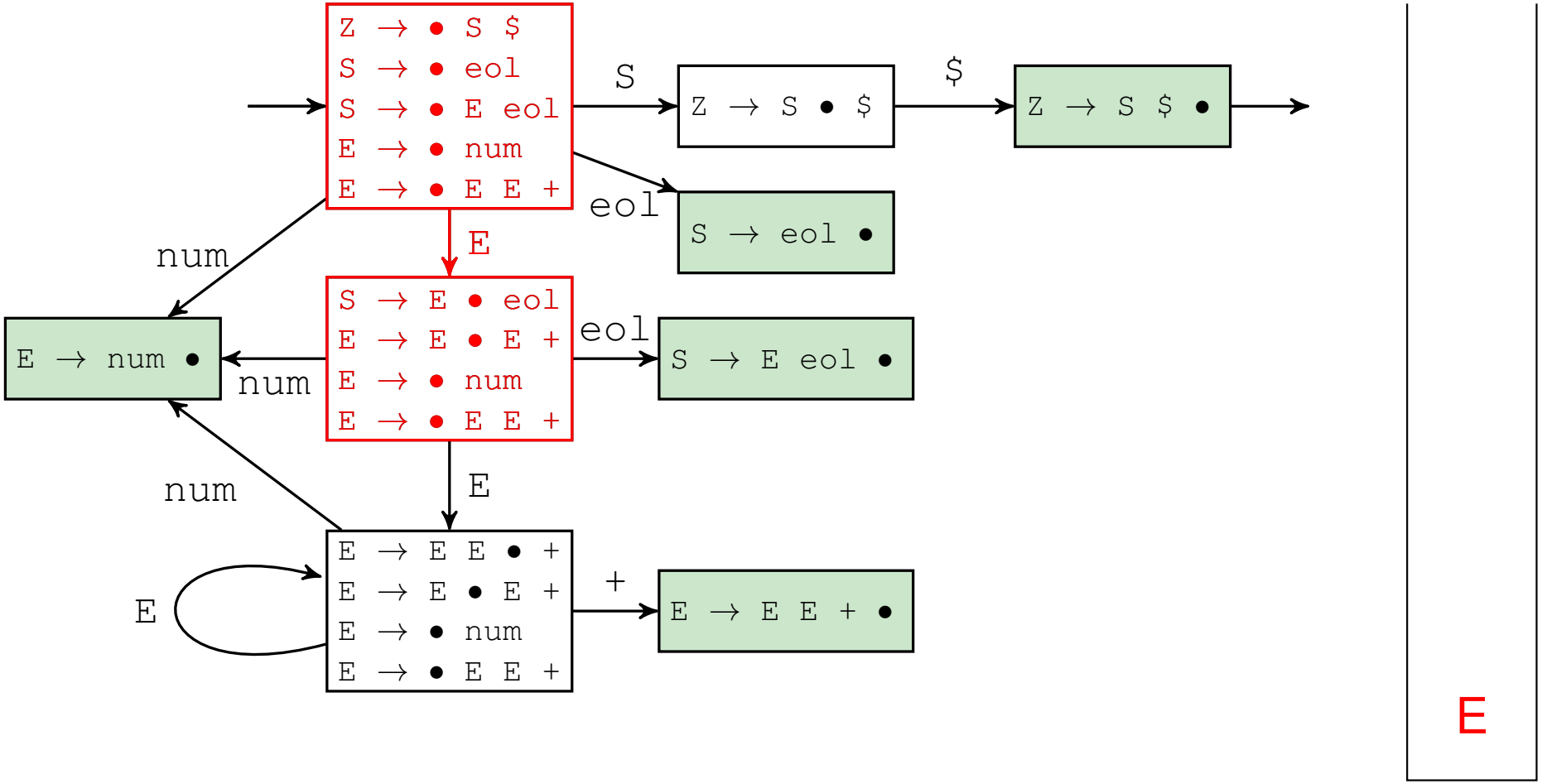
Exemple de reconnaissance

REDUCE : `E • eol $` contenu de la pile remplacé par `E`



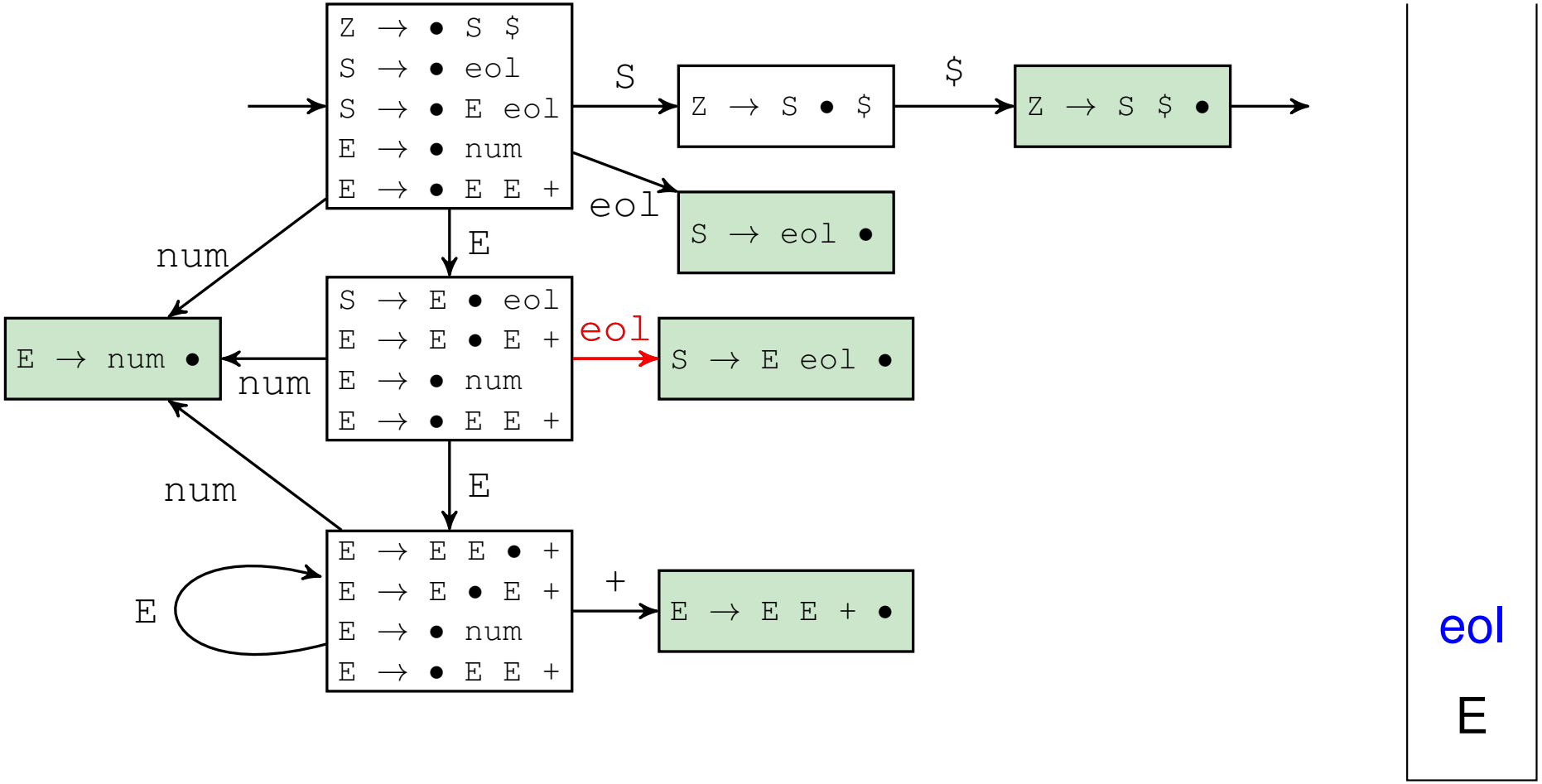
Exemple de reconnaissance

Départ : E • eol $ placement en fonction de la pile



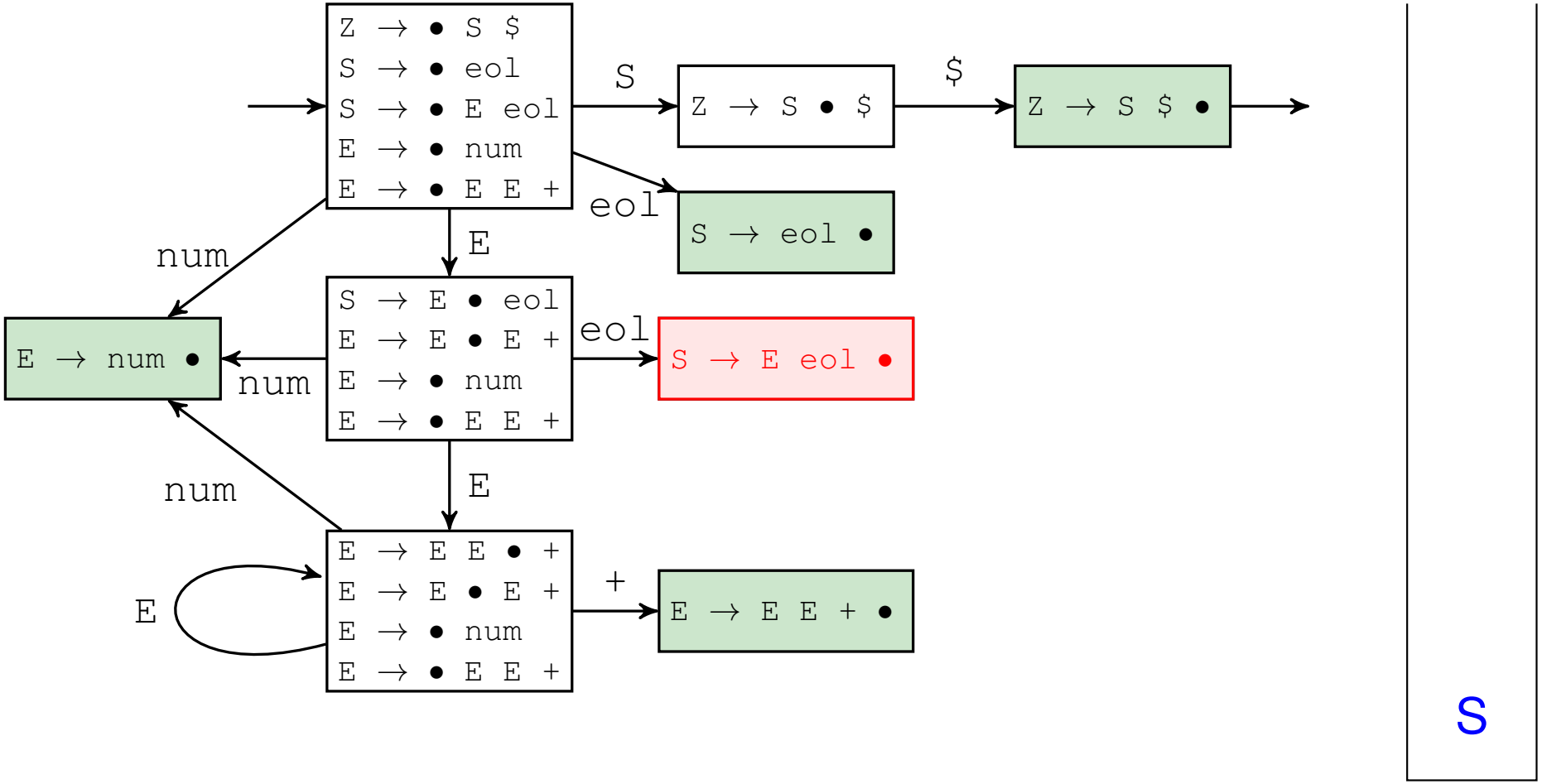
Exemple de reconnaissance

SHIFT : E eol • $ ajout de + sur la pile



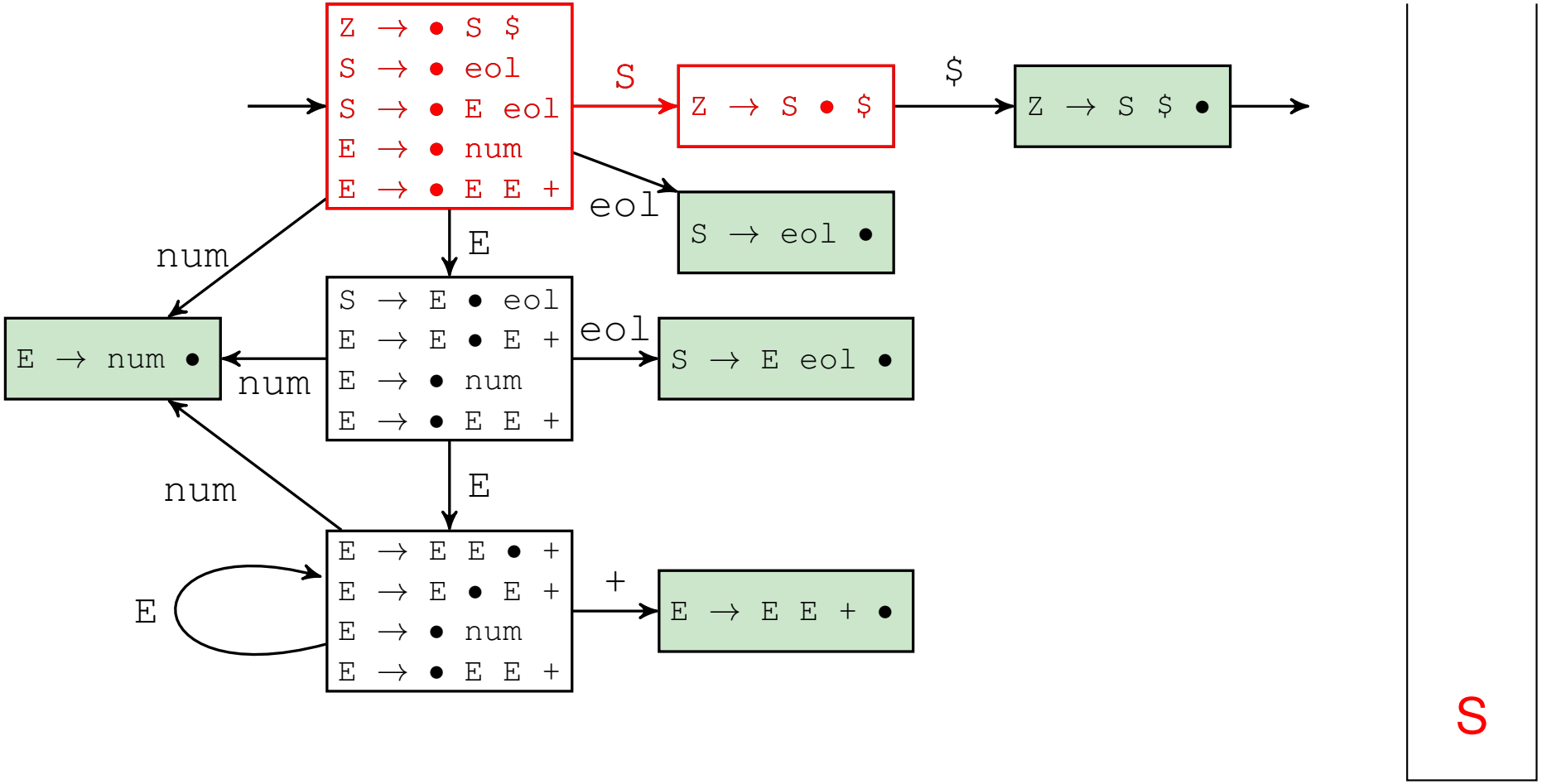
Exemple de reconnaissance

REDUCE : S • $ contenu de la pile remplacé par S



Exemple de reconnaissance

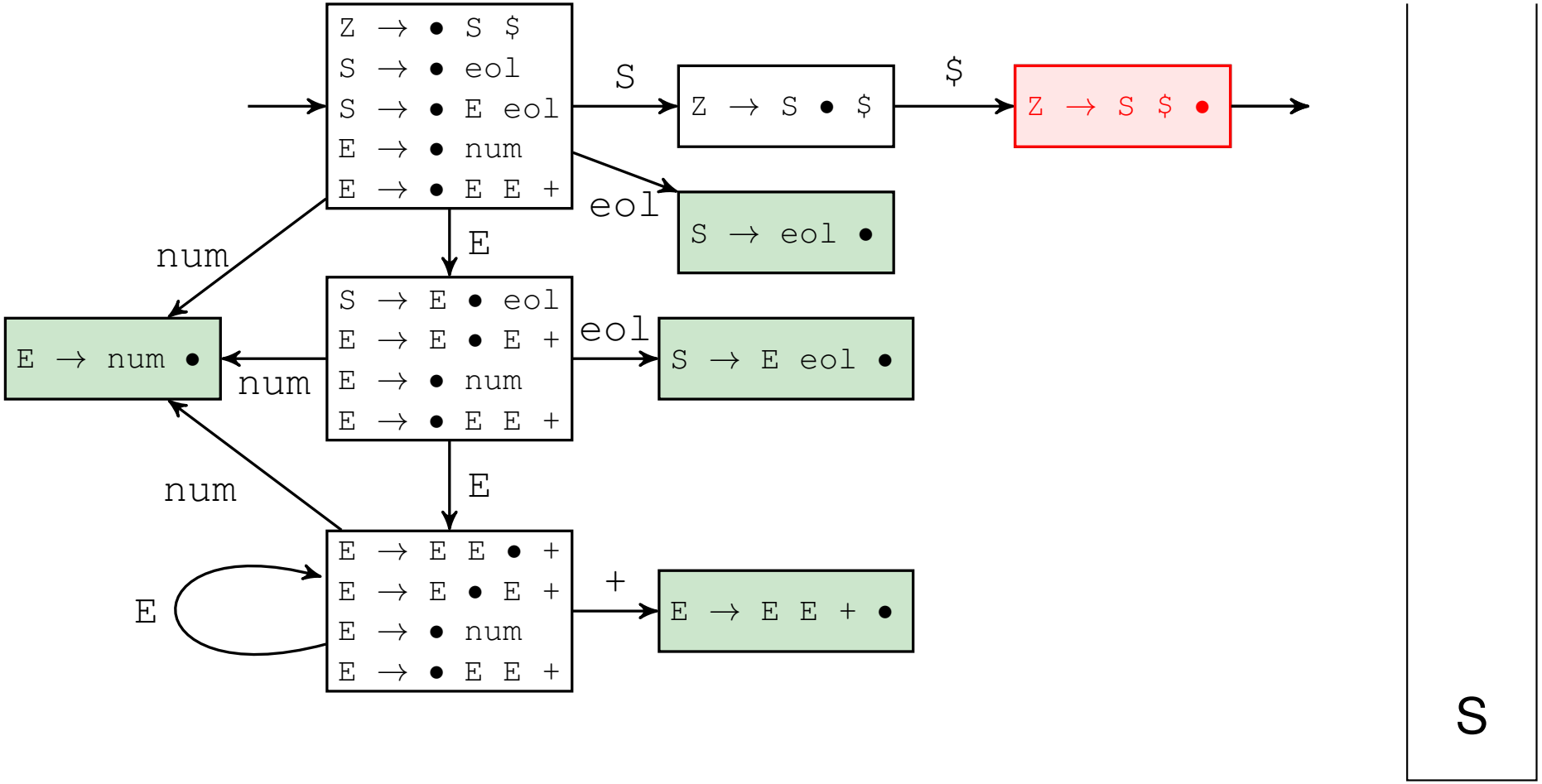
Départ : S • $ placement en fonction de la pile



Exemple de reconnaissance

SHIFT : S $ •

mot accepté



Bilan sur la reconnaissance par analyseur LR(0)

Analyse mécanique

Répétition en boucle de :

- ajout de lexèmes sur la pile jusqu'à arriver à un état de réduction
- application de la réduction $\rightarrow$ modification du haut de la pile
- lecture de la pile pour se replacer

Définition

On dit qu'une grammaire $\mathcal{G}$ est LR(0) lorsque les mots qu'elle engendre sont exactement ceux reconnus par l'analyseur LR(0) construit à partir de $\mathcal{G}$.

Conflits shift/reduce et reduce/reduce

L'analyse LR(0) peut échouer si on arrive à un état avec :

Conflits shift/reduce et reduce/reduce

L'analyse LR(0) peut échouer si on arrive à un état avec :

- une règle de réduction + une règle type shift

$$\begin{array}{l} \texttt{E} \rightarrow \texttt{num} \bullet \\ \texttt{E} \rightarrow \texttt{num} \bullet \texttt{+ E} \\ \ldots \end{array}$$

  $\rightarrow$ conflit shift/reduce
  - shift ou réduction ?

plutôt shift si possible

Conflits shift/reduce et reduce/reduce

L'analyse LR(0) peut échouer si on arrive à un état avec :

- une règle de réduction + une règle type shift

```
E → num •
E → num • + E
...
```

  → conflit shift/reduce
  - shift ou réduction ?

plutôt shift si possible

- deux règles de réductions

```
E → num •
E → var •
...
```

  → conflit reduce/reduce = **DÉMON**
  - choix impossible !

⇒ grammaire LR(0) = aucun conflit

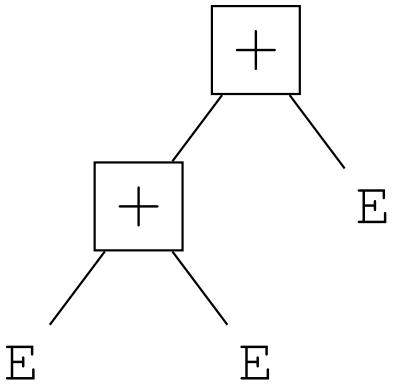
Plan

1. Compilation : Vue d'ensemble
2. Analyse lexicale
   - Reconnaissance des lexèmes
   - Passage à la pratique avec `flex`
3. Analyse syntaxique
   - Utilisation de `bison`
   - Dérivations et arbres syntaxiques
   - Analyse ascendante
   - Analyse de type LR(0)
   - Résolution de conflits

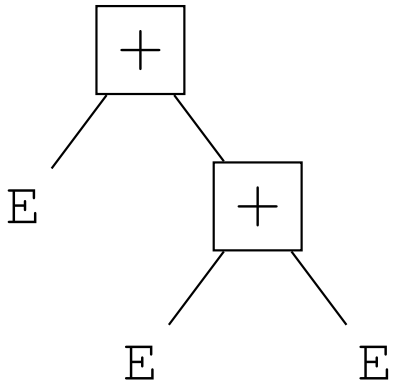
Déclaration l'associativité d'un opérateur

Cas typique : $\mathtt{E} \rightarrow \mathtt{E} + \mathtt{E}$

Analyse de $\mathtt{E} + \mathtt{E} + \mathtt{E}$ =



ou



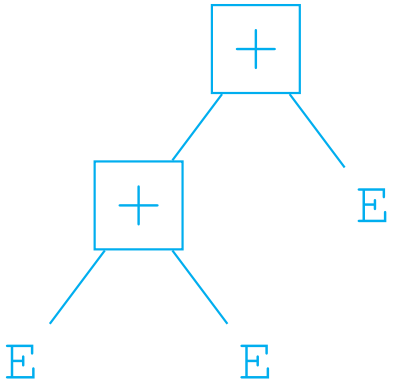
$\rightarrow$ conflit shift/reduce

$$\boxed{\begin{array}{cccccc} \mathtt{E} & \rightarrow & \mathtt{E} & + & \mathtt{E} & \bullet \\ \mathtt{E} & \rightarrow & \mathtt{E} & \bullet & + & \mathtt{E} \end{array}}$$

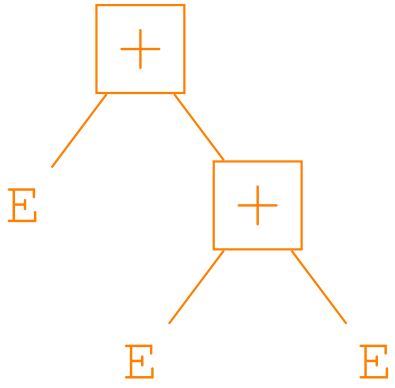
Déclaration l'associativité d'un opérateur

Cas typique : E → E + E

Analyse de E + E + E =



ou



→ conflit shift/reduce

E → E + E •
E → E • + E

Solution : préciser l'associativité

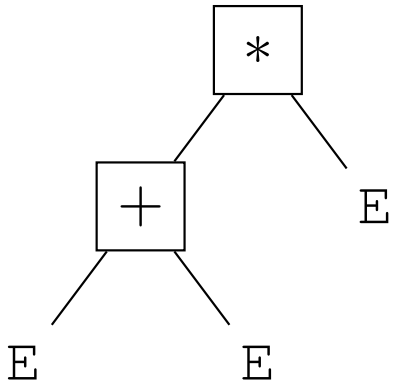
`%left PLUS` ou `%right PLUS`

Réduction — Shift

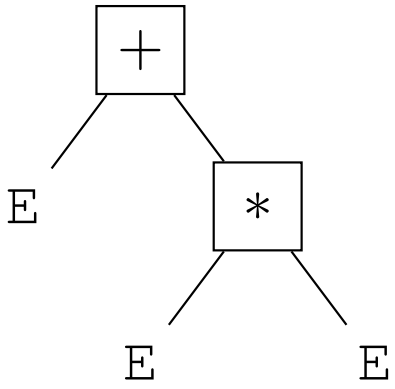
Imposer des priorités

Cas typique : $\mathtt{E} \rightarrow \mathtt{E} + \mathtt{E} \mid \mathtt{E} * \mathtt{E}$

Analyse de $\mathtt{E} + \mathtt{E} * \mathtt{E}$ =



ou



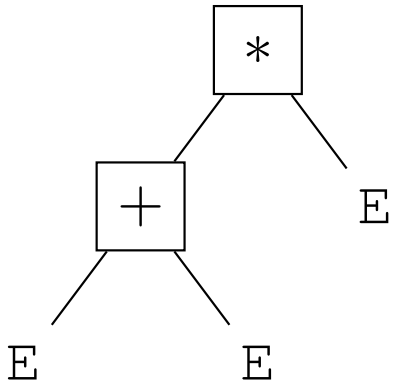
$\rightarrow$ conflit shift/reduce

$$\boxed{\begin{array}{cccccc} \mathtt{E} & \rightarrow & \mathtt{E} & + & \mathtt{E} & \bullet \\ \mathtt{E} & \rightarrow & \mathtt{E} & \bullet & * & \mathtt{E} \end{array}}$$

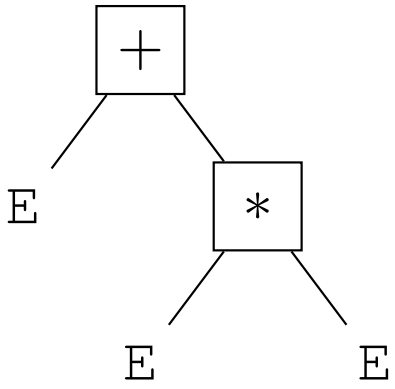
Imposer des priorités

Cas typique : `E → E + E | E * E`

Analyse de `E + E * E` =



ou



→ conflit shift/reduce

E	→	E	+	E	●
E	→	E	●	*	E

Solution : ici, fixer la priorité

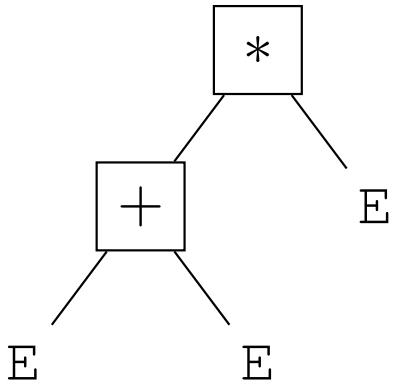
```
%left PLUS
%left TIMES
```

→ `TIMES` prioritaire

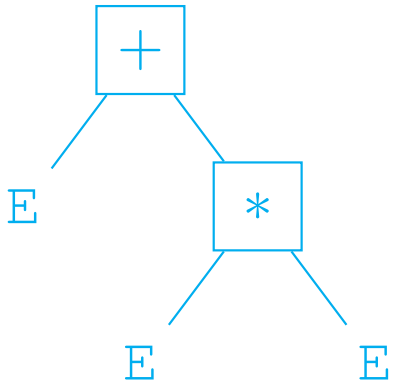
Imposer des priorités

Cas typique : E → E + E | E * E

Analyse de E + E * E =



ou



→ conflit shift/reduce

E	→	E	+	E	●
E	→	E	●	*	E

Solution : ici, fixer la priorité

```
%left PLUS
%left TIMES
```

→ TIMES prioritaire → Shift

Jouer avec un coup d'avance

Idée :

- raisonner sur le prochain caractère à venir
- éliminer les *shift* et *reduce* absurde

$\rightarrow$ moins de possibilités $\Rightarrow$ moins de conflits

Rappels : $N = \{\text{non-terminaux}\}$ A = { terminaux }

$$\begin{aligned}\text{FIRST}(X) &= \{x,\ X \rightarrow^* xw \text{ avec } x \in A \text{ et } w \in A^*\} \\ &= \{\text{lexème débutant une séquence générée à partir de } X\}\end{aligned}$$

$$\begin{aligned}\text{FOLLOW}(X) &= \{x,\ S \rightarrow^* uXxv \text{ avec } x \in A \text{ et } v \in A^*\ \} \\ &= \{\text{lexème pouvant suivre } X\}\end{aligned}$$

Anaylseur SLR(1)

Analyseur SLR(1) = Analyseur LR(0) + résolution de certains conflits :

- $\boxed{\begin{array}{l} \texttt{X} \rightarrow \texttt{Y} \bullet \\ \texttt{X} \rightarrow \texttt{U} \bullet \texttt{v W} \end{array}}$

  $\rightarrow$ `reduce` si prochain caractère à lire $\neq v$ et $\in$ FOLLOW(X)

  $\rightarrow$ `shift` si prochain caractère à lire $= v \notin$ FOLLOW(X)

  $\Rightarrow$ pas de vrai conflit si $v \notin$ FOLLOW(X)

Anaylseur SLR(1)

Analyseur SLR(1) = Analyseur LR(0) + résolution de certains conflits :

- $\boxed{\begin{array}{l} \texttt{X} \rightarrow \texttt{Y} \bullet \\ \texttt{X} \rightarrow \texttt{U} \bullet \texttt{v W} \end{array}}$

  $\rightarrow$ `reduce` si prochain caractère à lire $\neq v$ et $\in$ FOLLOW(X)
  $\rightarrow$ `shift` si prochain caractère à lire $= v \notin$ FOLLOW(X)

  $\Rightarrow$ pas de vrai conflit si $v \notin$ FOLLOW(X)

- $\boxed{\begin{array}{l} \texttt{X} \rightarrow \texttt{Y} \bullet \\ \texttt{U} \rightarrow \texttt{V} \bullet \end{array}}$

  $\rightarrow$ choix du `reduce` possible si FOLLOW(X) $\cap$ FOLLOW(U) $= \emptyset$

Stratégie `bison` : analyseur LALR(1)

Idée de l'analyseur LR(1) :

- construction similaire à LR(0)
- utilise des règles avec marqueur + caractères pouvant suivre •

→ automate plus fin, mais plus gros que LR(0)

Analyseur LALR(1) :

- optimisation en espace via fusion des états avec le même ensemble de règles avec marqueur

→ compromis entre LR(0) et LR(1)

En pratique :

- LR(1) > LALR(1) > SLR(1) > LR(0)
- la majorité du temps, SLR(1) suffit

SLR(1) $\simeq$ LR(1)