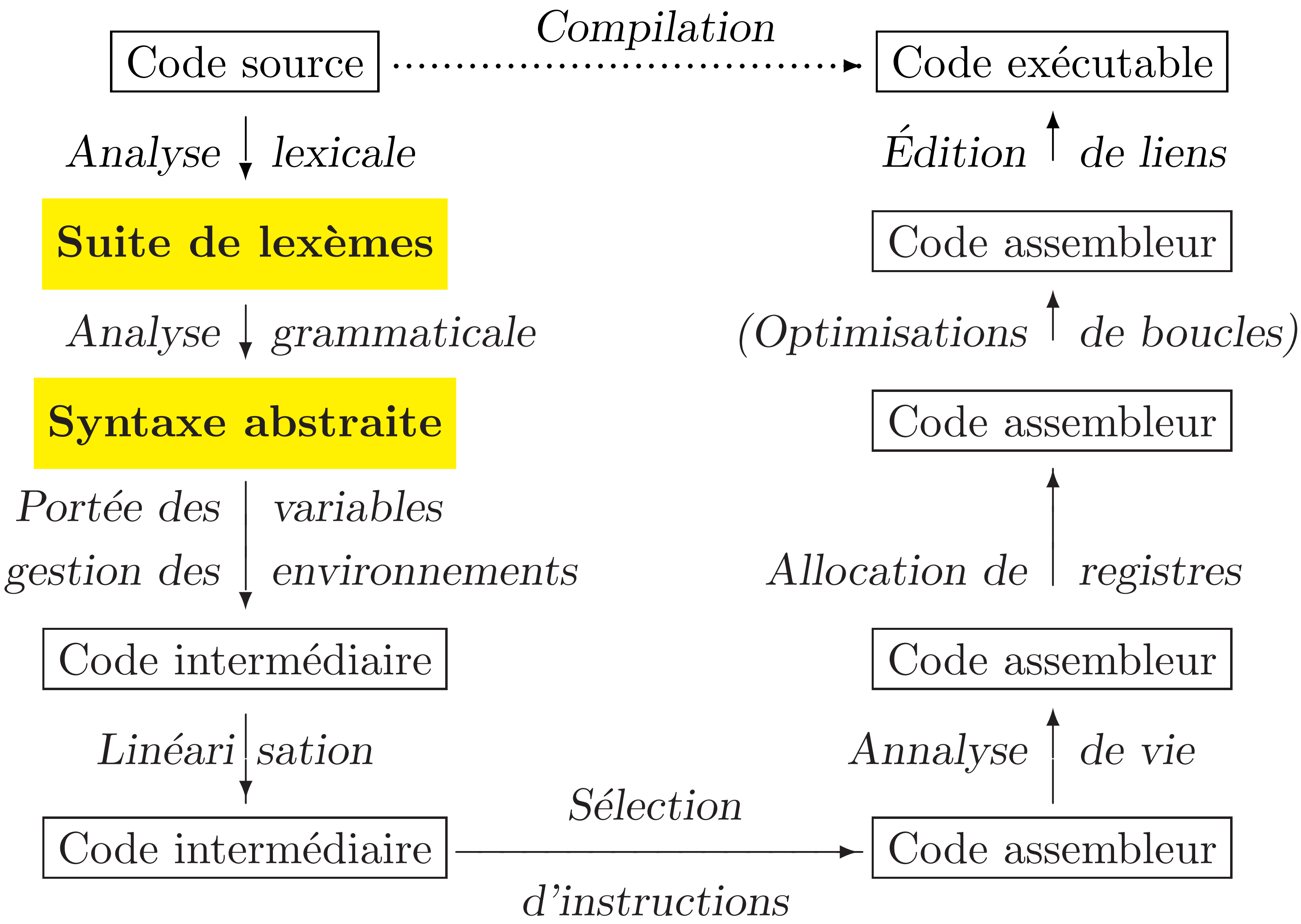
Grammaires algébriques
Analyse grammaticale

Luc Maranget
http://pauillac.inria.fr/~maranget

2005


Code source
Compilation
Code exécutable
Analyse lexicale
Édition de liens
Suite de lexèmes
Code assembleur
Analyse grammaticale
(Optimisations de boucles)
Syntaxe abstraite
Code assembleur
Portée des variables
gestion des environnements
Allocation de registres
Code intermédiaire
Code assembleur
Linéarisation
Annalyse de vie
Code intermédiaire
Sélection d'instructions
Code assembleur

Grammaires algébriques

Une grammaire algébrique (*context-free*) est un quadruplet (Σ, V, S, P) avec :

- Σ est l'alphabet des terminaux a, b etc. (ou lexèmes, ou *tokens*),
- V est l'alphabet des non-terminaux A, B, etc. (disjoint de Σ),
- $S \in V$ est le symbole de départ,
- P est un ensemble de productions $A \to \alpha$.

Notations

- Γ, Δ, etc. designe un caractère de $\Sigma \cup V$.
- α, β, etc. désigne un mot de $(\Sigma \cup V)^*$ (sauf ϵ, mot vide).
- $A \to \alpha_1 \mid \ldots \alpha_n$ désigne $A \to \alpha_1, \ldots, A \to \alpha_n$

Langages algébriques

Une grammaire G définit le langage $L(G)$ sur Σ.

Les mots de $L(G)$ sont engendrés à partir de S et en appliquant les productions jusqu'à plus soif (en *dérivant* S).

- Application d'une production au mot $\alpha A\beta$.

$$\alpha A\beta \Rightarrow \alpha\gamma\beta, \quad \text{avec } A \rightarrow \gamma \in P.$$

- Si $S \stackrel{*}{\Rightarrow} w$ (fermeture reflexive et transitive de $\Rightarrow$) avec $w \in \Sigma^*$, alors $w \in L(G)$.

Exemple $\Sigma = \{\mathbf{int}, (,), \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}$, $V = \{E\}$, départ en E et

$$E \rightarrow E \texttt{ + } E \qquad E \rightarrow E \texttt{ - } E \qquad E \rightarrow E \texttt{ * } E \qquad E \rightarrow E \texttt{ / } E$$

$$E \rightarrow \texttt{(}E\texttt{)} \qquad E \rightarrow \mathbf{int}$$

Example (suite)

Trois dérivations de la même expression arithmétique **1 + 2 ∗ 3** (**1**, **2** et **3** sont des entiers **int**).

$$\underline{E} \Rightarrow E + \underline{E} \Rightarrow \underline{E} + E * E \Rightarrow \mathbf{1} + \underline{E} * E \Rightarrow \mathbf{1} + \mathbf{2} * \underline{E} \Rightarrow \mathbf{1} + \mathbf{2} * \mathbf{3}$$

$$\underline{E} \Rightarrow \underline{E} * E \Rightarrow \underline{E} + E * E \Rightarrow \mathbf{1} + \underline{E} * E \Rightarrow \mathbf{1} + \mathbf{2} * \underline{E} \Rightarrow \mathbf{1} + \mathbf{2} * \mathbf{3}$$

$$\underline{E} \Rightarrow \underline{E} + E \Rightarrow \mathbf{1} + \underline{E} \Rightarrow \mathbf{1} + \underline{E} * E \Rightarrow \mathbf{1} + \mathbf{2} * \underline{E} \Rightarrow \mathbf{1} + \mathbf{2} * \mathbf{3}$$

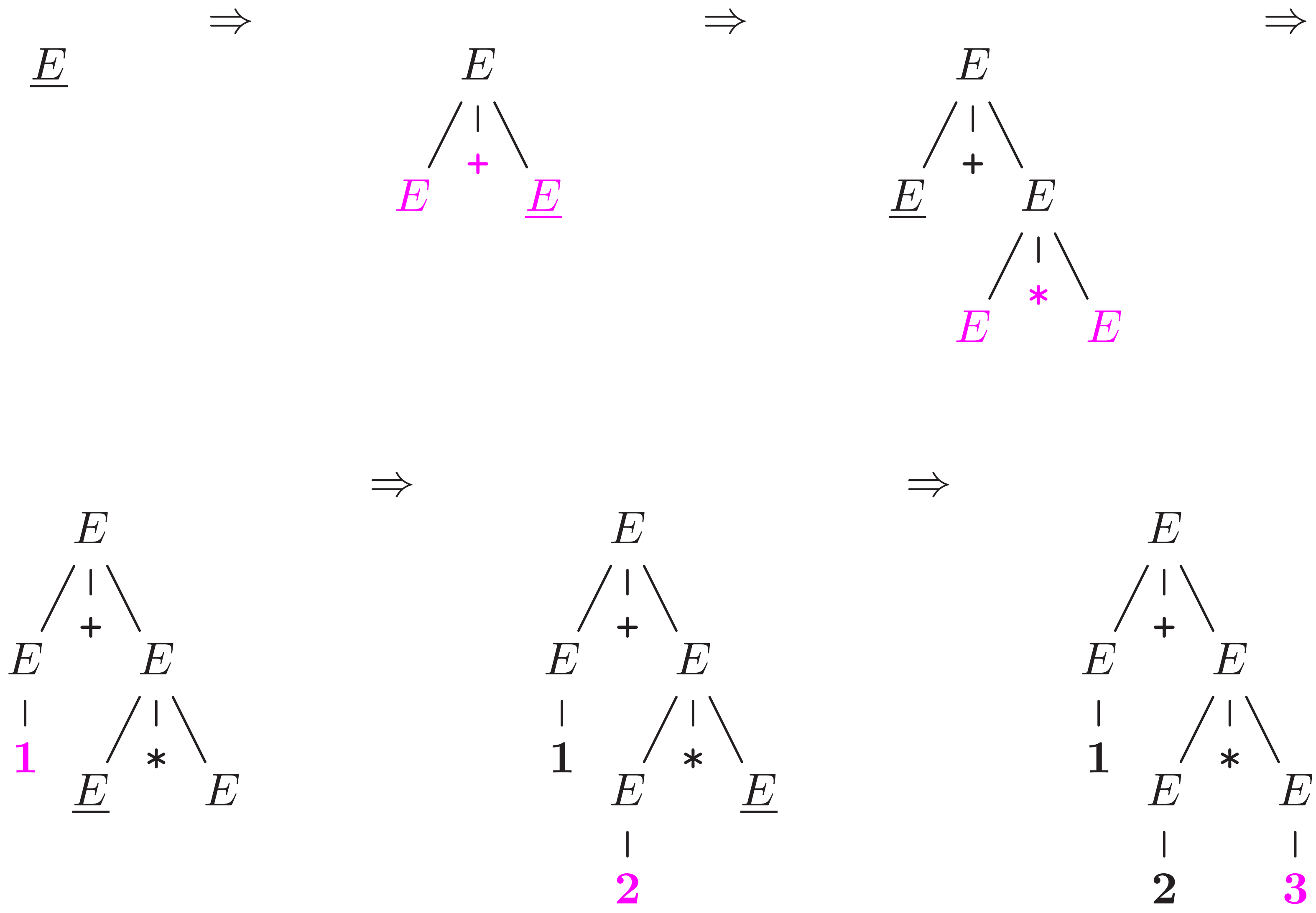
« *Sens* » des dérivations :

$$\underline{7} \Rightarrow 1 + \underline{6} \Rightarrow 1 + 2 * 3 \qquad \underline{9} \Rightarrow \underline{3} * 3 \Rightarrow 1 + 2 * 3$$

$$\underline{7} \Rightarrow 1 + \underline{6} \Rightarrow 1 + 2 * 3$$

1 et 3 ont le même sens, 1 et 2 ont des sens distincts.

Un sens grammatical, arbres de dérivation

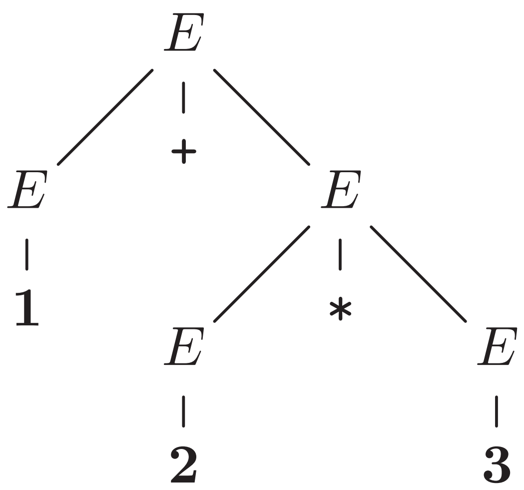


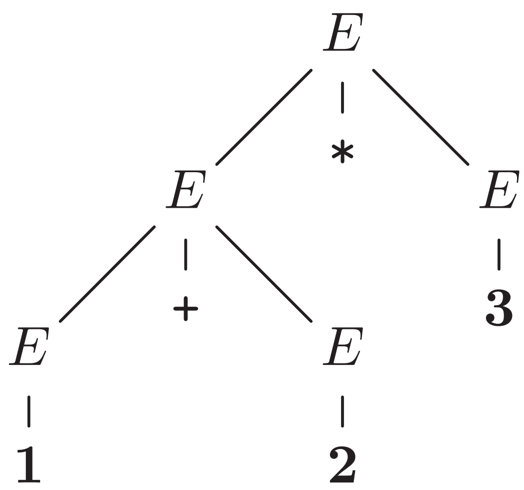
Seul l'arbre final compte

Le mot **1 + 2 ∗ 3** est dans $L(G)$

$$E \stackrel{*}{\Rightarrow} \mathbf{1} + \mathbf{2} * \mathbf{3}$$

Avec deux sens distincts :





L'arbre de dérivation ressemble à un AST (en plus riche).

De même, seules les dérivations gauches (ou droites) comptent.

Pour la compilation...

L'ambiguïté est nuisible.

- Le sens de la syntaxe concrète n'est plus clair.
- Les analyseurs sont déterministes de toute façon.

Il faut G' équivalente à G — $L(G) = L(G')$, et non-ambigüe — un seul arbre de dérivation (une seule dérivation gauche, droite).

Cf. la grammaire usuelle des expression arithmétiques.

$$E \longrightarrow E \texttt{+} T \qquad E \longrightarrow E \texttt{-} T \qquad E \longrightarrow T$$

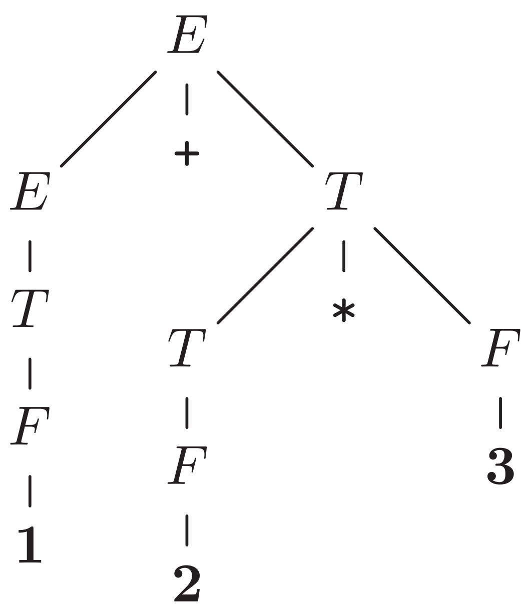
$$T \longrightarrow T \texttt{*} F \qquad T \longrightarrow T \texttt{/} F \qquad T \longrightarrow F$$

$$F \longrightarrow \texttt{(}E\texttt{)} \qquad F \longrightarrow \textbf{int}$$

(priorités, arbres qui penchent à gauche.)

Écrire un analyseur

Un premier essai avec G' :



Non-terminaux $\sim$ appels de fonctions, terminaux $\sim$ consommation de lexèmes dans un flux. Lire la grammaire comme la définition de fonctions récursives.

Ici ça doit rater, pourquoi ?

Recommençons

$$E \longrightarrow T \texttt{+} E \qquad E \longrightarrow T \texttt{-} E \qquad E \longrightarrow T$$

$$T \longrightarrow F \texttt{*} T \qquad T \longrightarrow F \texttt{/} T \qquad T \longrightarrow F$$

$$F \longrightarrow (E) \qquad F \longrightarrow \textbf{int}$$

Et là ça fonctionne.

```
let rec expr flux =
  term flux ;
  begin match look flux with (* Regarder le lexème de tête *)
  | (ADD|SUB) -> eat flux (* le manger *); expr flux
  | _         -> ()
  end
```

```
and term flux =
  factor flux ;
  begin match look flux with
  | (MUL|DIV) -> eat flux ; term flux
  | _          -> ()
  end

and factor flux = match look flux with
| INT _ -> eat flux
| LPAR  ->
   eat flux ; expr flux ; is RPAR flux (* vérifier et manger RPAR *)
| _ -> raise Error
```

Un vraie question : pourquoi ça marche ? Comment systématiser l'approche ?

Analyse LL(1)

L*eft-to-right parse*, **L***eftmost derivation*, *(***1***-token lookahead)*.

Une compilation des grammaires vers les analyseurs, avec pour cible :

1. Une fonction (qui prend un flux) par non-terminal.
2. Les fonctions commencent par `match look flux with...`
3. Les actions du filtrage sont des séquences traduites des membres droits :
   - $A \mapsto$ `A flux ;`
   - **token** $\mapsto$ `is TOKEN;`
4. Les filtrages se terminent obligatoirement par la clause `| _ -> raise Error`.

Autrement dit, on se décide seulement au début de l'appel et on examine tous les lexèmes possibles.

Première étape

Éliminer les cycles $A \to A$ qui partent tous seuls.

Éliminer la récursion gauche Transformer

$$A \to A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \beta_m$$

en

$$A \to \beta_1 A_0 \mid \ldots \beta_m A_0 \qquad\qquad A_0 \to \alpha_1 A_0 \mid \ldots \mid \alpha_n A_0 \mid \epsilon$$

Généralisable aux cycles et à la récursion gauche indirects :
$A \overset{*}{\Rightarrow} A\alpha$.

Élimination de la récursion gauche

Partons de :

$$S \longrightarrow E\ \mathbf{eof}$$

$$E \longrightarrow E\ \mathtt{+}\ E \qquad E \longrightarrow E\ \mathtt{-}\ E \qquad E \longrightarrow T$$

$$T \longrightarrow T\ \mathtt{*}\ T \qquad T \longrightarrow T\ \mathtt{/}\ T \qquad T \longrightarrow F$$

$$F \longrightarrow \mathbf{(}E\mathbf{)} \qquad F \longrightarrow \mathbf{int}$$

On obtient :

$$E \longrightarrow TE_0 \qquad E_0 \longrightarrow \mathtt{+}\ TE_0\ \mid \mathtt{-}\ TE_0\ \mid\ \epsilon$$

$$T \longrightarrow FT_0 \qquad T_0 \longrightarrow \mathtt{*}\ FT_0\ \mid \mathtt{/}\ FT_0\ \mid\ \epsilon$$

$$F \longrightarrow \mathbf{(}E\mathbf{)}\ \mid\ \mathbf{int}$$

Grammaire non-ambigüe, inutile de le vérifier (ça viendra tout seul par la suite).

Se decider entre les membres droits

Définissons FIRST, de $(\Sigma \cup V)^*$ dans $\Sigma \cup \{\epsilon\}$.

$$\mathrm{FIRST}(\alpha) = \{a \in \Sigma \mid \alpha \overset{*}{\Rightarrow} a\beta\} \quad (\cup\{\epsilon\} \text{ si } \alpha \overset{*}{\Rightarrow} \epsilon)$$

Définissons FOLLOW, de V dans Σ.

$$\mathrm{FOLLOW}(A) = \{a \in \Sigma \mid S \overset{*}{\Rightarrow} \alpha A a \beta\}$$

On se décide entre deux productions non-vides $A \to \alpha \mid \beta$ si $\mathrm{FIRST}(\alpha) \cap \mathrm{FIRST}(\beta) = \emptyset$.

On peut appliquer la production vide $A \to \epsilon$ si le lexème examiné est dans $\mathrm{FOLLOW}(A)$.

Calcul de FIRST et FOLLOW

$$\text{FIRST}(\epsilon) = \{\epsilon\} \qquad\qquad \text{FIRST}(a) = \{a\}$$

$$\text{FIRST}(A) = \text{FIRST}(\alpha_1) \cup \ldots \cup \text{FIRST}(\alpha_n),\ \text{quand } A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$$

$$\text{FIRST}(\Gamma\alpha) = \text{FIRST}(\Gamma),\ \text{si } \epsilon \notin \text{FIRST}(\Gamma)$$

$$\text{FIRST}(\Gamma\alpha) = (\text{FIRST}(\Gamma) \setminus \{\epsilon\}) \cup \text{FIRST}(\alpha),\ \text{si } \epsilon \in \text{FIRST}(\Gamma)$$

Production	**Contrainte**
$A \rightarrow \alpha B \beta$	$(\text{FIRST}(\beta) \setminus \{\epsilon\}) \subseteq \text{FOLLOW}(B)$
$A \rightarrow \alpha B$	$\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$
$A \rightarrow \alpha B \beta$, avec $\epsilon \in \text{FIRST}(\beta)$	$\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$

Calcul de FIRST

$$E \rightarrow TE_0 \qquad\qquad E_0 \rightarrow \texttt{+}\, TE_0 \;\mid\; \texttt{-}\, TE_0 \;\mid\; \epsilon$$

$$T \rightarrow FT_0 \qquad\qquad T_0 \rightarrow \texttt{*}\, FT_0 \;\mid\; \texttt{/}\, FT_0 \;\mid\; \epsilon$$

$$F \rightarrow \mathbf{(}E\mathbf{)} \;\mid\; \mathbf{int}$$

En itérant à partir de $\mathrm{FIRST}^0(\Gamma) = \emptyset$.

$$\mathrm{F}^1(F) = \mathrm{F}^0(\mathbf{(}E\mathbf{)}) \cup \mathrm{F}^0(\mathbf{int}) \cup \mathrm{F}^0(F) = \{\mathbf{(}, \mathbf{int}\}$$

$$\mathrm{F}^1(T_0) = \mathrm{F}^0(\texttt{*}\, FT_0) \cup \mathrm{F}^0(\texttt{/}\, FT_0) \cup \mathrm{F}^0(\epsilon) \cup \mathrm{F}^0(T_0) = \{\texttt{*}, \texttt{/}, \epsilon\}$$

$$\mathrm{F}^1(E_0) = \mathrm{F}^0(\texttt{+}\, TE_0) \cup \mathrm{F}^0(\texttt{-}\, TE_0) \cup \mathrm{F}^0(\epsilon) \cup \mathrm{F}(E_0) = \{\texttt{+}, \texttt{-}, \epsilon\}$$

$$\mathrm{F}^1(T) = \mathrm{F}^0(FT_0) \cup \mathrm{F}^0(T) = \emptyset$$

$$\mathrm{F}^1(E) = \mathrm{F}^0(TE_0) \cup \mathrm{F}^0(E) = \emptyset$$

$$\mathrm{F}^2(F) = \{\textbf{(}, \textbf{int}\} \qquad \mathrm{F}^2(T_0) = \{\texttt{*}, \texttt{/}, \epsilon\} \qquad \mathrm{F}^2(E_0) = \{\texttt{+}, \texttt{-}, \epsilon\}$$

(changent pas)

$$\mathrm{F}^2(T) = \mathrm{F}^1(FT_0) \cup \mathrm{F}^1(T) = \{\textbf{(}, \textbf{int}\} \qquad (\epsilon \notin \mathrm{F}^1(F))$$

$$\mathrm{F}^2(E) = \mathrm{F}^1(TE_0) \cup \mathrm{F}^1(E) = \emptyset$$

$$\mathrm{F}^3(F) = \{\textbf{(}, \textbf{int}\} \qquad \mathrm{F}^3(T_0) = \{\texttt{*}, \texttt{/}, \epsilon\} \qquad \mathrm{F}^3(E_0) = \{\texttt{+}, \texttt{-}, \epsilon\}$$

$$\mathrm{F}^3(T) = \{\textbf{(}, \textbf{int}\} \qquad \mathrm{F}^3(E) = \{\textbf{(}, \textbf{int}\}$$

Un calcul plus intelligent

$$\vdots$$

$$\mathrm{F}^{n+1}(T) = \mathrm{F}^{n+1}(FT_0) \cup \mathrm{F}^{n}(T) = \mathrm{F}^{n+1}(F) \cup \mathrm{F}^{n}(T) \qquad (\epsilon \not\in \mathrm{F}^{n}(F))$$

$$\mathrm{F}^{n+1}(E) = \mathrm{F}^{n+1}(T)$$

Résultat

	FIRST	FOLLOW
S	(, **int**	
E	(, **int**	), **eof**
E_0	ϵ, +, -	), **eof**
T	(, **int**	), +, -, **eof**
T_0	ϵ, *, /	), +, -, **eof**
F	(, **int**	), +, -, *, /, **eof**

On peut maintenant remplir une table d'analyse prédictive :

- Pour toutes les $A \rightarrow \alpha$, mettre α dans les case de la la colonne a pour tous les a de FIRST(α).
- En outre, si ϵ est dans FIRST(α), mettre α dans les cases de la colonne a pour tous les a de FOLLOW(A).

Table d'analyse prédictive

	int	(	)	+	-	*	/	**eof**
S	E **eof**	E **eof**						
E	TE_0	TE_0						
E_0			ϵ	$+\,TE_0$	$-\,TE_0$			ϵ
T	FT_0	FT_0						
T_0			ϵ	ϵ	ϵ	$*\,FT_0$	$/\,FT_0$	ϵ
F	**int**	(E)						

- Case vide $\mapsto$ une erreur à l'analyse.
- Pas de doublons dans les cases, la grammaire est LL(1).

Analyseur LL(1)

```
let rec start flux = match look flux with
| INT _|LPAR -> expr flux ; is EOF flux
| _          -> raise Error

and expr flux = match look flux with
| INT _|LPAR -> term flux ; expr0 flux
| _          -> raise Error

and expr0 flux =  match look flux with
| ADD      -> is ADD flux ; term flux ; expr0 flux
| SUB      -> is SUB flux ; term flux ; expr0 flux
| RPAR|EOF -> ()
| _        -> raise Error
```

```
and term flux = match look flux with
| INT _|LPAR -> facteur flux ; term0 flux
| _          -> raise Error

and term0 flux =  match look flux with
| MUL                -> is  MUL flux ; facteur flux ; term0 flux
| DIV                -> is  DIV flux ; facteur flux  ;  term0 flux
| RPAR|EOF|ADD|SUB  -> ()
| _                  -> raise Error

and factor flux = match look flux with
| INT i  -> is  (INT i)
| LPAR   -> is LPAR ; expr flux ; is RPAR flux
| _      -> raise Error
```

Une grammaire non-ambigüe et non-LL(1)

Grammaire :

$$S \longrightarrow E\ \textbf{eof} \qquad E \longrightarrow T + E \qquad E \longrightarrow T \qquad T \longrightarrow F * T \qquad T \longrightarrow F$$

$$F \longrightarrow (E) \qquad\qquad F \longrightarrow \textbf{int}$$

On a la table :

	int	(	)	+	*	**eof**
E	$T, T + E$	$T, T + E$				
T	$F, F * T$	$F, F * T$				
F	**int**	(E)				

Solution : factoriser

Remplacer $E \to T \mid T + E$ par $E \to TE_0$ et $E_0 \to \epsilon \mid + E$.

$$S \to E \ \mathbf{eof} \qquad E \to TE_0 \qquad E_0 \to \epsilon \qquad E_0 \to + E$$

$$T \to FT_0 \qquad T_0 \to \epsilon \qquad T_0 \to * T \qquad F \to \mathbf{int} \qquad F \to (E)$$

	int	**(**	**)**	**+**	*****	**eof**
E	$T + E_0$	$T + E_0$				
E_0			ϵ	$+ E$		ϵ
T	$F * T_0$	$F * T_0$				
T_0			ϵ		$* T$	ϵ
F	**int**	(E)				

Ou alors se donner plus de pouvoir (cf. analyseur écrit à la main).

LL, le bilan

LL(k) : examiner k lexèmes d'avance (tables potentiellement énormes).

Pour :

- Simple à comprendre.
- Guide pour écrire les analyseurs à la main.
- On peut concevoir les langages de programmation avec des grammaires LL (cf Pascal).

Contre (pour la production automatique d'analyseurs).

- Il y a plus puissant (LR).
- Transformations de la grammaire nécessaires.

Automate *shift-reduce*

Une nouvelle cible :

- Ils ont une pile de symboles de $\Sigma \cup V$.
- Ils lisent un flux de lexèmes.

Les actions possibles :

- *shift*, *i.e.* consommer et empiler un lexème, ou
- *reduce*, *i.e.* réduire une règle. Cela revient à appliquer une règle de production sur le sommet (partie droite) de la pile.

Les automates LR cherchent à déterminiser le processus.

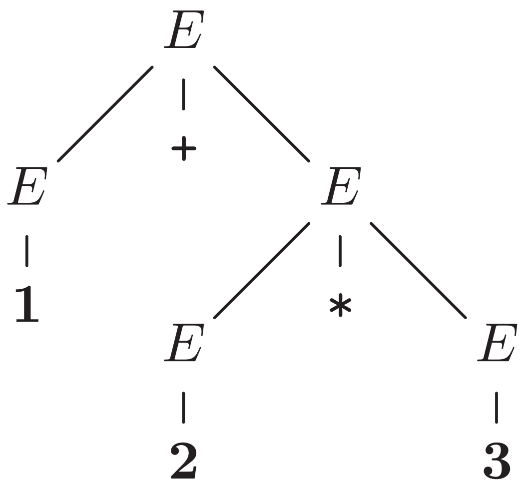
Exemples d'éxecution de l'automate *shift-reduce*

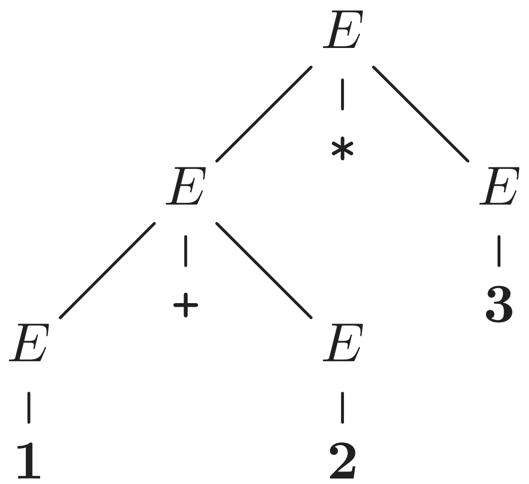
Dans la grammaire :

$$E \rightarrow E \texttt{+} E \qquad E \rightarrow E \texttt{-} E \qquad E \rightarrow E \ast E \qquad E \rightarrow E \,/\, E$$

$$E \rightarrow (E) \qquad\qquad E \rightarrow \textbf{int}$$

Trouver les deux dérivations :





Bizarrement, l'automate à pile construit une dérivation droite.

La première

Pile	Flux	Action	Production
	$\mathbf{1} + \mathbf{2} * \mathbf{3}$	*shift*	
$\mathbf{1}$	$+ \mathbf{2} * \mathbf{3}$	*reduce*	$E \longrightarrow \textbf{int}$
E	$+ \mathbf{2} * \mathbf{3}$	*shift*	
$E +$	$\mathbf{2} * \mathbf{3}$	*shift*	
$E + \mathbf{2}$	$* \mathbf{3}$	*reduce*	$E \longrightarrow \textbf{int}$
$E + E$	$* \mathbf{3}$	*shift*	
$E + E *$	$\mathbf{3}$	*shift*	
$E + E * \mathbf{3}$		*reduce*	$E \longrightarrow \textbf{int}$
$E + E * E$		*reduce*	$E \longrightarrow E * E$
$E + E$		*reduce*	$E \longrightarrow E + E$
E			

La seconde

Pile	Flux	Action	Production
	$\mathbf{1} + \mathbf{2} * \mathbf{3}$	*shift*	
$\mathbf{1}$	$+ \mathbf{2} * \mathbf{3}$	*reduce*	$E \longrightarrow \textbf{int}$
E	$+ \mathbf{2} * \mathbf{3}$	*shift*	
$E +$	$\mathbf{2} * \mathbf{3}$	*shift*	
$E + \mathbf{2}$	$* \mathbf{3}$	*reduce*	$E \longrightarrow \textbf{int}$
$E + E$	$* \mathbf{3}$	*reduce*	$E \longrightarrow E + E$
E	$* \mathbf{3}$	*shift*	
$E *$	$\mathbf{3}$	*shift*	
$E * \mathbf{3}$		*reduce*	$E \longrightarrow \textbf{int}$
$E * E$		*reduce*	$E \longrightarrow E * E$
E			

Comment déterminiser (*shift* ou *reduce* ?)

Avec la grammaire usuelle (non-ambigüe).

L'automate devrait pouvoir se décider :

- À l'aide du sommet de la pile,
- En fonction du lexème en attente.

Le contrôle devrait pouvoir être confié à un automate fini.

Sur l'exemple (grammaire usuelle des expressions arithmétiques).

- Comment choisir $E \overset{*}{\Rightarrow} \mathbf{1}$, $T \overset{*}{\Rightarrow} \mathbf{2}$, $F \overset{*}{\Rightarrow} \mathbf{3}$?
- Comment choisir *shift*, à l'étape critique signalée.

Pile	Flux	Action	Production
	$\mathbf{1} + \mathbf{2} * \mathbf{3}$	*shift*	
$\mathbf{1}$	$+ \mathbf{2} * \mathbf{3}$	*reduce*	$F \longrightarrow \mathbf{int}$
F	$+ \mathbf{2} * \mathbf{3}$	*reduce*	$T \longrightarrow F$
T	$+ \mathbf{2} * \mathbf{3}$	*reduce*	$E \longrightarrow F$
E	$+ \mathbf{2} * \mathbf{3}$	*shift*	
$E +$	$\mathbf{2} * \mathbf{3}$	*shift*	
$E + \mathbf{2}$	$* \mathbf{3}$	*reduce*	$F \longrightarrow \mathbf{int}$
$E + F$	$* \mathbf{3}$	*reduce*	$T \longrightarrow F$
$E + T$	$* \mathbf{3}$	*shift*	
$E + T *$	$\mathbf{3}$	*shift*	
$E + T * \mathbf{3}$		*reduce*	$F \longrightarrow \mathbf{int}$
$E + T * F$		*reduce*	$T \longrightarrow T * F$
$E + T$		*reduce*	$E \longrightarrow E + T$
E			

Écrire un analyseur LR à la main

$$S \rightarrow E\ \mathbf{eof} \qquad\qquad E \rightarrow E\ \mathbf{+}\ E \qquad\qquad E \rightarrow \mathbf{int}$$

```
type symbol = E | T of token
```

```
let rec auto stack flux =
  match look flux, stack with
    ...
and shift stack flux =
   let tok = look flux in
   eat flux ; auto (T tok::stack) flux
```

Un automate possible

```
let rec auto stack flux = match look flux, stack with
(* Accepter l'entrée *)
| EOF, [E] -> ()
| ADD, [E] -> shift stack flux
(* Réduction de E → E + E *)
| (ADD|EOF), E::T ADD::E::rem -> auto (E::rem) flux
(* Réduction de E → int *)
| (ADD|EOF), T (INT _)::rem -> auto (E::rem) flux
(* Shift de int *)
| INT _, (T ADD::E::_|[]) -> shift stack flux
(* N'importe quoi d'autre est une erreur *)
| _ -> raise Error
```

Un autre automate possible

```
let rec auto stack flux = match look flux, stack with
(* Accepter l'entrée *)
| EOF, [E] -> ()
(* Toujours shifter + *)
| ADD, E::_ -> shift stack flux
(* Réduction de E → E + E *)
| EOF, E::T ADD::E::rem -> auto (E::rem) flux
(* Réduction de E → int *)
| (ADD|EOF), T (INT _)::rem -> auto (E::rem) flux
(* Shift de int *)
| INT _, ([]|T ADD::_) -> shift stack flux
| _ -> raise Error
```

Automate LR(1)

- Il y a en fait quatre actions, *shift*, *reduce*$(A \to \alpha)$ (la production réduite est indiquée), *accept* (succès) et *error* (échec).
- Pile (Δ_i, s_i) d'états et de symboles de $\Sigma \cup V$ (sauf le fond de pile s_0, état initial)
- L'automate est déterminé par deux fonctions :
  - $\mathsf{goto}(s, \Delta)$ vers les états,
  - et $\mathsf{action}(s, a)$ vers les actions.

Avec une pile $s_0, (\Delta_1), \ldots (\Delta_m, s_m)$, l'automate (de contrôle) consulte le lexème a en attente et selon la valeur de $\mathsf{action}(s_m, a)$ effectue :

shift Le lexème est consommé et $(a, \mathsf{goto}(s_m, a))$ est empilé.

reduce$(A \to \alpha)$ L'automate dépile l eléments, où l est la longueur de α et $(A, \mathsf{goto}(s_{m-l}, A))$ est empilé.

accept* ou *error L'automate signale le succès ou un échec.

Automate de contrôle

La fonction goto représente les transitions de l'automate de contrôle, un de ses états I est un ensemble de configurations C. une configuration C est une paire composée,

1. d'une production pointée $A \rightarrow \alpha \bullet \beta$, où $A \rightarrow \alpha\beta$ est une production,
2. et du prochain lexème possible a après $\alpha\beta$.

Une configuration C décrit un état courant de l'automate *shift-reduce* sous-jacent, avec α en sommet de pile et un flux dont le début dérive de βa.

Calcul des états et des transitions

La fermeture d'un ensemble de configurations est le plus petit ensemble I contenant I et satisfaisant :

$$((A \rightarrow \alpha \bullet B\beta, a) \in I \ \wedge \ B \rightarrow \gamma \in G \ \wedge \ b \in \mathrm{FIRST}(\beta a)) \Longrightarrow (B \rightarrow\bullet \gamma, b) \in I$$

Les transitions sont :

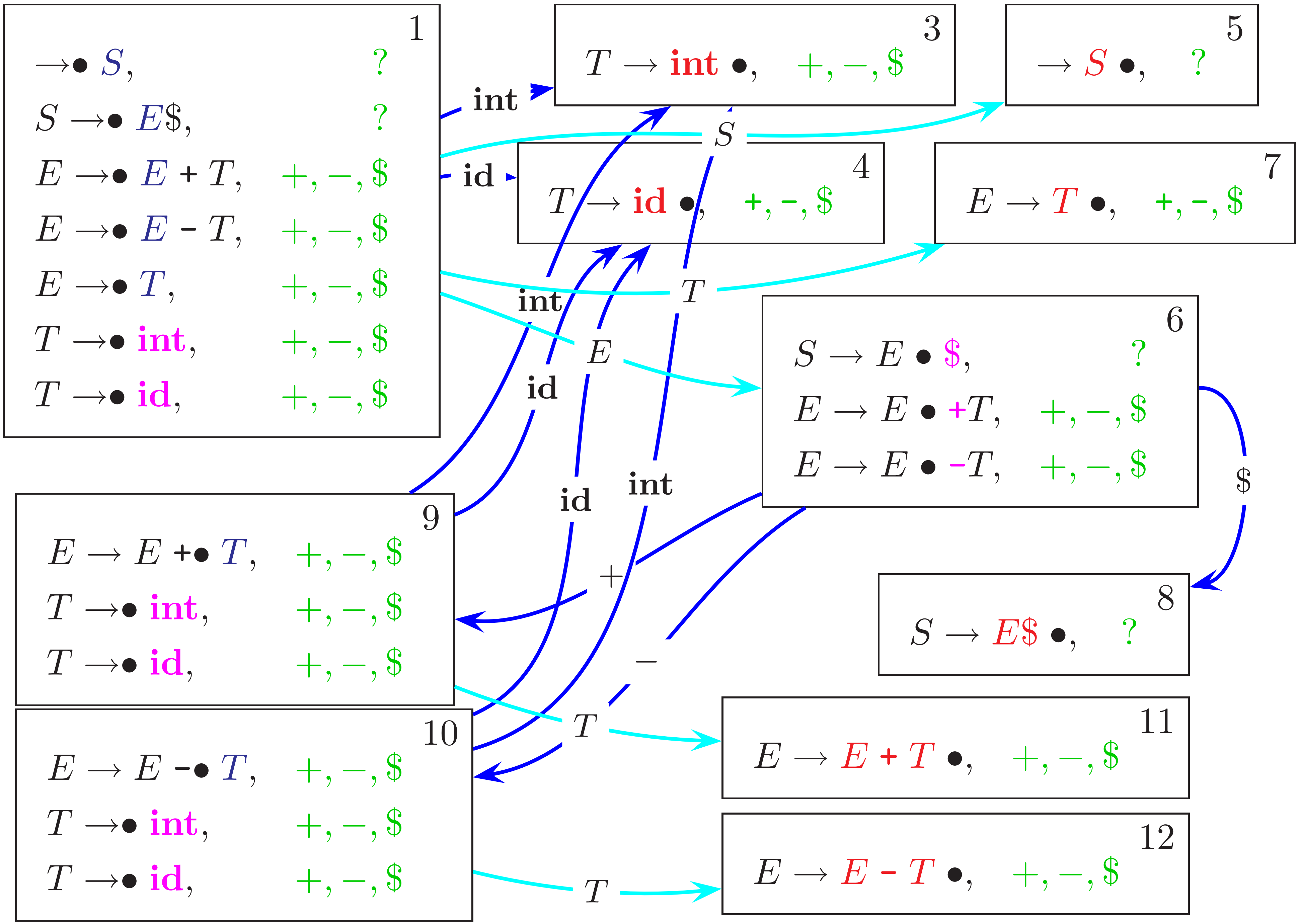
$$\mathsf{goto}(I, \Delta) = \mathrm{fermeture}(\{(A \rightarrow \alpha\Delta \bullet \beta, a) \mid (A \rightarrow \alpha \bullet \Delta\beta, a) \in I\})$$

Le calcul est celui de la déterminisation des NFA, à partir de la configuration initiale « $(\rightarrow\bullet S, \Sigma)$ ».

Avec la grammaire :

$$\rightarrow S \qquad S \rightarrow E\ \$ \qquad E \rightarrow E \mathbin{\textsf{+}} T \qquad E \rightarrow E \mathbin{\textsf{-}} T \qquad E \rightarrow T$$

$$T \rightarrow \mathbf{id} \qquad\qquad T \rightarrow \mathbf{int}$$


1
→• S, ?
S →• E$, ?
E →• E + T, +, −, $
E →• E - T, +, −, $
E →• T, +, −, $
T →• int, +, −, $
T →• id, +, −, $
3
T → int •, +, −, $
5
→ S •, ?
4
T → id •, +, −, $
7
E → T •, +, −, $
6
S → E • $, ?
E → E • +T, +, −, $
E → E • -T, +, −, $
8
S → E$ •, ?
9
E → E +• T, +, −, $
T →• int, +, −, $
T →• id, +, −, $
10
E → E -• T, +, −, $
T →• int, +, −, $
T →• id, +, −, $
11
E → E + T •, +, −, $
12
E → E - T •, +, −, $
int
id
S
T
E
$
+
−
int
id
id
int
T
T

Calcul des tables LR(1)

1. Si $(A \to \alpha \bullet a\beta, b) \in I$ et $\mathsf{goto}(I, a) = J$, alors $\mathsf{action}(I, a) = \mathit{shift}$ (on peut représenter simultanément $\mathsf{goto}(I, a)$ en écrivant $\mathit{shift}(J)$).,
2. Si $(A \to \alpha \bullet, a) \in I$, alors $\mathsf{action}(I, a) = \mathit{reduce}(A \to \alpha)$. Sauf, si $A \to \alpha$ est l'axiome $\to S$, auquel cas $\mathsf{action}(I, a) = \mathit{accept}$.

Si $\mathsf{action}(I, a)$ n'est pas défini par ces deux règles, alors on a $\mathsf{action}(I, a) = \mathit{error}$.

En cas de conflit, G n'est pas LR(1).

reduce/reduce Lorsque $(A \to \alpha \bullet, a) \in I$ et $(B \to \beta \bullet, a) \in I$.
Super grave.

shift/reduce Lorsque $(A \to \alpha \bullet a\gamma, b) \in I$ et $(B \to \beta \bullet, a) \in I$,
parfois moins grave (priorités).

LALR(1) et bilan

- LR(1) généralisé en LR(k) (se décider sur k lexèmes).
- LALR(1) (`yacc`, `bison` etc.), « simplification » de LR(1) : ignorer les a dans $(A \to \alpha \bullet \beta, a)$ et fusionner les états. Les tables sont plus petites (moins d'états).
- Bilan

$$\mathrm{LL}(k) \subset \mathrm{LL}(k+1) \qquad \mathrm{LR}(k) \subset \mathrm{LR}(k+1) \qquad \mathrm{LL}(k) \subset \mathrm{LR}(k)$$

$$\mathrm{LALR}(1) \subset \mathrm{LR}(1)$$

Toute les grammaires cités sont non-ambigües.

En pratique : `ocamlyacc` (`yacc` adapté à Caml) traite les grammaires LALR(1).

ocamlyacc

Source *Nom*.mly, objet *Nom*.ml et *Nom*.mli.

```
%{
open Ast
%}
/* Déclaration des lexèmes */
%token LPAR RPAR
%token ADD SUB MUL DIV
%token <int> INT
%token EOF
/* Point d'entrée */
%start expr
%type <Ast.t> expr
%%
```

Deux dernières sections.

```
/* productions, avec actions */
expr:
  expr1 EOF {$1}
;

expr1:
  expr1 ADD expr1 {Binop (Add,$1, $3)}
| expr1 SUB expr1 {Binop (Sub,$1, $3)}
| expr1 MUL expr1 {Binop (Mul,$1, $3)}
| expr1 DIV expr1 {Binop (Div,$1, $3)}
| SUB expr1       {Binop (Sub, Int 0, $2)}
| INT             {Int $1}
| LPAR expr1 RPAR {$2}
;
```

Compilation de l'exemple

```
# ocamlyacc -v arith.mly
20 shift/reduce conflicts
```

Un analyseur est quand même produit.

Savoir (rare), aller voir dans `arith,output` (option `-v`).

```
16: shift/reduce conflict (shift 10, reduce 2) on ADD
 ...
16: shift/reduce conflict (shift 13, reduce 2) on DIV
state 16
expr1 : expr1 . ADD expr1  (2)
expr1 : expr1 ADD expr1 .  (2)
 ...
expr1 : expr1 . DIV expr1  (5)
```

E + E • op E vu comme E + (E op E) (*shift* gagne par défaut).

Solution par priorités

- On donne des (niveaux de) priorités aux lexèmes. La priorité d'une production est celle de son dernier lexème. En cas de conflit, la plus forte priorité gagne.
- Pour les cas où la priorité est la même. Il y a une information d'associativité, gauche (*reduce* gagne), droite (*shift* gagne), aucune (*error* gagne).

Les priorités sont déclarées dans la deuxième section.

```
/* Des moins prioritaires aux plus prioritaires */
%left ADD SUB
%left MUL DIV
```

En avons nous fini ?

Non ! et le moins unaire ? (pourtant `ocamlyacc` ne signale plus de conflits).

État de réduction du moins unaire.

```
state 9
expr1 : expr1 . ADD expr1  (2)
expr1 : expr1 . SUB expr1  (3)
expr1 : expr1 . MUL expr1  (4)
expr1 : expr1 . DIV expr1  (5)
expr1 : SUB expr1 .  (6)

MUL  shift 12
DIV  shift 13
.  reduce 6
```

On veut `-` $E \bullet$ `op` E compris comme (`-` E) `op` E dans tous les cas.

Solution

Définir une priorité, l'assigner à la règle du moins unaire.

```
%left ADD SUB
%left MUL DIV
/* left  sans importance ici, car pas de lexème de cette priorité */
%left UMINUS
    ...
| SUB expr1 %prec UMINUS {Binexp (Sub, Int 0, $2)}
```

Les Projets

Trois projets « classiques » :

- Recibler vers Pentium/IA-32
- Construire un système à bytecode.
- Ajouter le passage par variable.

`http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/compil/projet/`

D'autres projets sont possibles...

x

Votre projet !