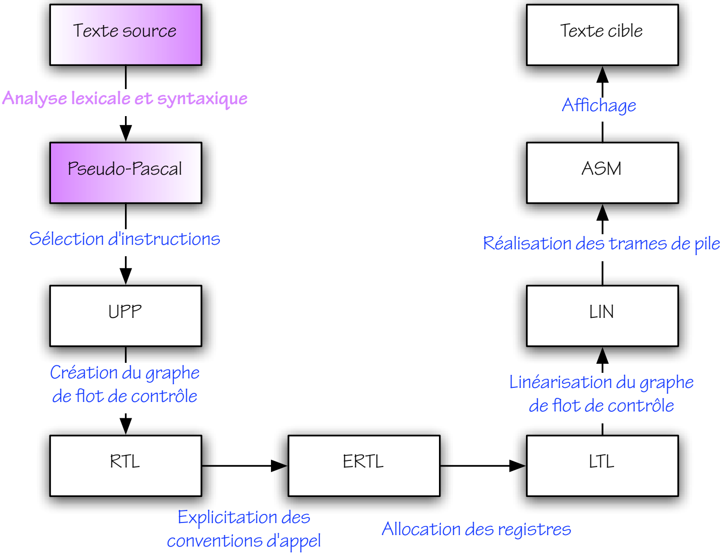
Compilation (INF 564)

Analyse syntaxique

François Pottier

16 décembre 2015


Texte source
Analyse lexicale et syntaxique
Pseudo-Pascal
Sélection d'instructions
UPP
Création du graphe de flot de contrôle
RTL
Explicitation des conventions d'appel
ERTL
Allocation des registres
LTL
Linéarisation du graphe de flot de contrôle
LIN
Réalisation des trames de pile
ASM
Affichage
Texte cible

Analyse lexicale et analyse syntaxique

Un programme se présente d'abord comme une *suite de caractères :*

x 1 ␣ : = ␣ a ␣ * ␣ (x 2 ␣ + ␣ b);

L'analyse lexicale (*"lexing"*) (voir Appel, chapitre 2) la transforme en une suite d'entités de plus haut niveau, les *lexèmes* (*"tokens"*) :

ID("x1") COLONEQ ID("a") TIMES LPAREN
ID("x2") PLUS ID("b") RPAREN SEMICOLON

L'analyse syntaxique (*"parsing"*) doit transformer cette suite de lexèmes en un arbre de syntaxe abstraite.

Une approche déclarative

Écrire un analyseur syntaxique à la main est difficile. Le code est complexe, répétitif. Le risque d'erreur est important.

On préfère écrire une *grammaire* afin de :

- *spécifier* ce que doit faire l'analyseur ;
- *construire* automatiquement l'analyseur à partir de la grammaire.

Grammaires algébriques

Analyse LL(1)

Analyse LR(0)

Quelques mots de l'analyse LR(1)

L'outil Menhir

Exemple de grammaire algébrique

Voici l'exemple classique d'une grammaire d'expressions arithmétiques :

$$E ::= E \mathbf{+} E \mid E \mathbf{-} E \mid E \mathbf{*} E \mid E \mathbf{/} E \mid \mathbf{(}\, E \,\mathbf{)} \mid \mathbf{int}$$

Le symbole E est celui que l'on définit ; les autres sont des lexèmes.

Cette notation est appelée BNF (Backus-Naur Form).

Grammaires algébriques

En général, une *grammaire algébrique* ou *grammaire non contextuelle* est un quadruplet (Σ, V, S, P), où :

- Σ est l'alphabet des *symboles terminaux*, notés a, b, etc. Les symboles terminaux sont typiquement les lexèmes produits par l'analyseur lexical.
- V est un ensemble de *symboles non-terminaux*, notés A, B, etc.
- $S \in V$ est le *symbole de départ*.
- P est un ensemble de *productions* de la forme $A \longrightarrow \beta$, où β est un mot sur l'alphabet $\Sigma \cup V$.

Exemple de grammaire algébrique

Pour la grammaire des expressions arithmétiques,

- $\Sigma = \{\mathbf{int}, \mathbf{(}, \mathbf{)}, \mathbf{+}, \mathbf{-}, \mathbf{*}, \mathbf{/}\}$.
- $V = \{E\}$.
- $S = E$.
- $P = \left\{ \begin{array}{lll lll} E & \rightarrow & E\ \mathbf{+}\ E & E & \rightarrow & E\ \mathbf{-}\ E \\ E & \rightarrow & E\ \mathbf{*}\ E & E & \rightarrow & E\ \mathbf{/}\ E \\ E & \rightarrow & \mathbf{(}\ E\ \mathbf{)} & E & \rightarrow & \mathbf{int} \end{array} \right\}$.

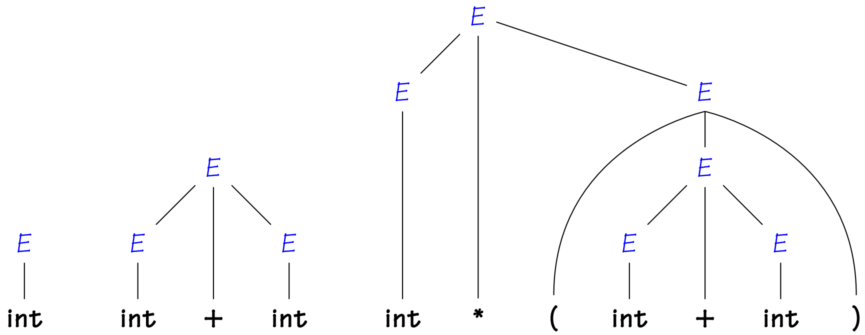
Une grammaire, pour quoi faire ?

Une grammaire définit un *langage*, un ensemble de mots :
par exemple, **int**, ou **int + int**, ou **int * (int + int)**, etc.

De plus, une grammaire attribue à chacun de ces mots
(au moins) une *structure*, sous forme d'un arbre de production...

Arbres de production

Voici trois *arbres de production* correspondant aux mots précédents :



Arbres de production

Un *arbre de production* est formé de deux types de nœuds :

1. un *nœud terminal* est étiqueté par un symbole terminal a et n'a aucun fils ;
2. un *nœud non-terminal* est étiqueté par un symbole non-terminal A et possède des fils dont la séquence des étiquettes forme le mot β, où $A \longrightarrow \beta$ est une production.

La *frange* de l'arbre est le mot formé par les étiquettes des nœuds terminaux.

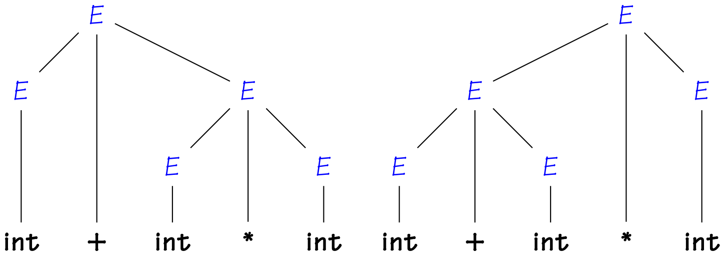
Langage engendré par une grammaire

Le *langage* $L(G)$ engendré par une grammaire G est l'ensemble des franges de tous les arbres de production conformes à G.

Un mot w sur l'alphabet Σ appartient donc à $L(G)$ ss'il existe (au moins) un arbre de production dont w est la frange.

Ambiguïté

Il peut exister plusieurs arbres de production de même frange :



Dans ce cas, la grammaire est *ambiguë*.

Sources d'ambiguité

Quelles sont, pour cette grammaire des expressions arithmétiques, les causes d'ambiguïté ?

Sources d'ambiguité

Quelles sont, pour cette grammaire des expressions arithmétiques, les causes d'ambiguïté ?

- la *priorité* entre deux opérateurs n'a pas été spécifiée ;
- *l'associativité* de chaque opérateur n'a pas été spécifiée.

Éviter l'ambiguïté

L'ambiguïté est, pour nous, *nuisible*. Nous souhaitons donc l'éviter.

De plus, un analyseur syntaxique déterministe sera plus *efficace*.

Malheureusement, déterminer si une grammaire algébrique est ou non ambiguë est un problème *indécidable*.

Éviter l'ambiguïté

On utilise des techniques de construction d'analyseurs déterministes qui réussissent si la grammaire appartient à une certaine *classe* décidable : LL(1), LR(1), *etc.*

L'appartenance à une telle classe implique donc la non-ambiguïté, mais la réciproque est fausse.

Éviter l'ambiguïté

Lorsque l'on est face à une grammaire ambiguë, on cherche une grammaire *non ambiguë* et qui engendre *le même langage*.

Cela demande de l'imagination...

Éviter l'ambiguïté : exemple

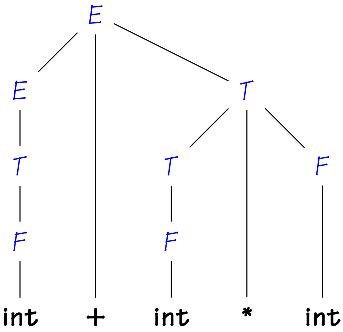
Le langage des expressions arithmétiques peut être décrit par une autre grammaire. On se donne trois non-terminaux E, T, F, pour *expressions*, *termes* et *facteurs* :

$$
\begin{array}{lcl@{\qquad}lcl}
E & \rightarrow & E \; \mathbf{+} \; T & T & \rightarrow & T \; \mathbf{/} \; F \\
E & \rightarrow & E \; \mathbf{-} \; T & T & \rightarrow & F \\
E & \rightarrow & T & F & \rightarrow & \mathbf{(} \; E \; \mathbf{)} \\
T & \rightarrow & T \; \mathbf{*} \; F & F & \rightarrow & \mathbf{int}
\end{array}
$$

Cette décomposition en niveaux, qui permet de refléter des règles de *priorité* et *d'associativité*, est une technique classique.

Éviter l'ambiguïté

Le mot **int + int * int** n'admet plus qu'un seul arbre de production :



Éviter l'ambiguïté

Cette nouvelle grammaire est-elle ambiguë ? *Non.*

> *Ce n'est pas évident. La preuve la plus simple consiste à vérifier qu'elle appartient à la classe LR(1).*

Engendre-t-elle *le même langage* que la grammaire précédente ? *Oui.*

> *À nouveau, cela demande une démonstration.*

L'ambiguïté est donc *une propriété de la grammaire* et non du langage engendré.

Grammaires algébriques

Analyse LL(1)

Analyse LR(0)

Quelques mots de l'analyse LR(1)

L'outil Menhir

Principe

L'analyse descendante est fondée sur un principe simple :

- lire la grammaire comme un *programme* récursif non déterministe.

L'analyse LL(1) y ajoute un second principe :

- résoudre les choix en consultant le *premier* lexème de l'entrée.

Grammaire ≈ programme non déterministe

Une grammaire peut être lue comme un programme *non déterministe* :

```
let rec E () =
  match guess 3 with
  | 0 → E(); consume(+); T()
  | 1 → E(); consume(-); T()
  | 2 → T()

and T () =
  match guess 3 with
  | 0 → T(); consume(*); F()
  | 1 → T(); consume(/); F()
  | 2 → F()

and F () =
  match guess 2 with
  | 0 → consume((); E(); consume())
  | 1 → consume(int)

(* On suppose l'entrée stockée
   dans une variable globale. *)
```

Les fonctions E, T, F, consume renvoient () ou lancent une exception. Elles consomment la partie de l'entrée qu'elles ont reconnue.

Déterminisation LL(1)

Pour rendre ce programme déterministe, la technique LL(1) consiste à remplacer chaque *guess n* par une expression qui *consulte* uniquement le *premier* lexème de l'entrée.

Déterminisation LL(1)

Cette approche est-elle applicable à notre grammaire ?

$$
\begin{array}{l}
\text{let rec } E\ () = \\
\quad \text{match } \mathit{guess}\ 3 \text{ with} \\
\quad \mid 0 \rightarrow E(); \mathit{consume}(\mathbf{+}); T() \\
\quad \mid 1 \rightarrow E(); \mathit{consume}(\mathbf{-}); T() \\
\quad \mid 2 \rightarrow T()
\end{array}
$$

Par quelle expression pourrait-on remplacer le "*guess* 3"... ?

Limitations de la technique LL(1)

Premier problème, la grammaire exhibe des *facteurs à gauche :*

$$\begin{array}{lcl} E & \rightarrow & E \mathbf{+} T \\ E & \rightarrow & E \mathbf{-} T \end{array}$$

Pour déterminer quelle production développer, il faudrait savoir quel lexème, **+** ou **-**, on trouvera *après* avoir lu une expression E.

Consulter le *premier* lexème de l'entrée *ne suffit pas*. Il faudrait dans certains cas lire arbitrairement loin en avant (*"unbounded lookahead"*).

Limitations de la technique LL(1)

Second problème, la grammaire est *récursive à gauche :*

$$E \quad \longrightarrow \quad E + T$$

La fonction E est appelée. Après réflexion, on décide de développer la production ci-dessus. Alors E est appelée récursivement. Le premier lexème n'ayant pas changé, on prend à nouveau la même décision...

Dans ce cas, *l'analyseur ne termine pas.*

Limitations de la technique LL(1)

En bref, une grammaire qui présente :

- soit des facteurs à gauche,
- soit une récursivité à gauche,

ne peut pas être analysée par la technique LL(1).

Nouvelle modification

On pourrait contourner ce problème en *modifiant* encore la grammaire.

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
E & \longrightarrow & T\ E' & T & \longrightarrow & F\ T' \\
E' & \longrightarrow & \mathbf{+}\ T\ E' & T' & \longrightarrow & \mathbf{*}\ F\ T' \\
E' & \longrightarrow & \mathbf{-}\ T\ E' & T' & \longrightarrow & \mathbf{/}\ F\ T' \\
E' & \longrightarrow & \epsilon & T' & \longrightarrow & \epsilon \\
F & \longrightarrow & \mathbf{int} & F & \longrightarrow & \mathbf{(}\ E\ \mathbf{)}
\end{array}
$$

Facteurs à gauche et récursivité à gauche disparaissent.

On *peut* transcrire cette grammaire en un analyseur LL(1).

Pourquoi cette grammaire est-elle LL(1) ?

Les ensembles

$$FIRST(\,\mathbf{*}\ F\ T'\,) = \{\mathbf{*}\} \qquad FIRST(\,\mathbf{/}\ F\ T'\,) = \{\mathbf{/}\} \qquad FOLLOW(T') = \{\mathbf{)}, \mathbf{+}, \mathbf{-}\}$$

sont deux à deux *disjoints*.

Donc, lorsqu'on souhaite consommer T', consulter le prochain symbole de l'entrée suffit à savoir quelle branche choisir.

La situation est analogue en ce qui concerne E'.

Pourquoi LL(1) ?

En résumé, la technique LL(1) :

- est *simple.*
- impose souvent des *transformations* manuelles de la grammaire.

JavaCC est basé sur cette technique (avec des extensions).

Il existe des techniques descendantes plus puissantes : LL(k), LL(*), ...

Grammaires algébriques

Analyse LL(1)

Analyse LR(0)

Quelques mots de l'analyse LR(1)

L'outil Menhir

De LL à LR

La technique LR, due à Knuth (1965), *est fondée* sur le principe suivant, *décrit de* façon très informelle :

- au lieu *de* choisir *entre deux* productions $A \longrightarrow \beta_1$ *et* $A \longrightarrow \beta_2$ *avant* d'avoir lu β_1 ou β_2,
- on va d'abord progresser jusqu'à avoir reconnu β_1 ou β_2, *et* on décidera *a posteriori* quelle production était la bonne.

On *explore* donc plusieurs branches "en parallèle", *et* on *effectue* un choix plus tardif.

Grammaire ≈ automate à pile non déterministe

Plus précisément, l'analyse LR(k) consiste à :

- lire la grammaire comme un *automate non déterministe,*
  - dont *l'état* indique une production partiellement reconnue, et
  - dont *la pile* contient l'historique des états précédents;
- *déterminiser* (si possible) cet automate, à l'aide d'une construction des parties (powerset construction).

Je vais illustrer cela dans le cas $k = 0$.

Construction LR(0)

Les *états* sont étiquetés ainsi :

- soit $\bullet A$, "je m'apprête à reconnaître un mot issu de A";
- soit $A \longrightarrow \beta \bullet \gamma$, "j'ai reconnu un mot issu de β, il me reste à reconnaître un mot issu de γ pour pouvoir affirmer avoir reconnu un mot issu de A".

Les *transitions* sont étiquetées par des *symboles* (terminaux ou non) ou bien par ϵ.

Construction LR(0)

Illustrons la construction pour cette grammaire simplifiée :

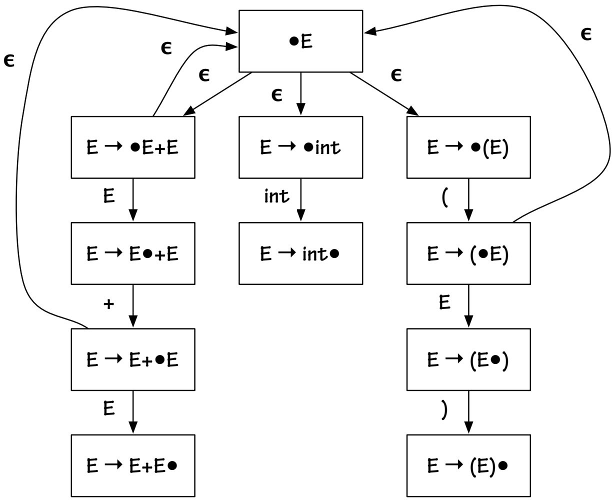
$$E ::= E \mathbf{+} E \mid \mathbf{(}\, E\, \mathbf{)} \mid \mathbf{int}$$

Elle est *récursive à gauche*. Cela ne pose pas de problème.

Les *facteurs à gauche* non plus ne posent pas de problème pour LR.

Cette grammaire est *ambiguë*. La déterminisation de l'automate rencontrera un écueil.

Automate LR(0) non déterministe



Fonctionnement de l'automate

L'automate a une *pile d'états*, dont le sommet est *l'état courant*. De plus, il a une *entrée*, une suite de lexèmes.

Dans un état s, il peut effectuer deux types d'actions :

- *décaler* : si le lexème de tête est a, supprimer a de l'entrée, et empiler un nouvel état s' tel que $s \xrightarrow{\epsilon^\star a} s'$.
- *réduire* : si s est étiqueté $A \rightarrow \beta\bullet$, dépiler $|\,\beta\,|$ éléments, ce qui ramène l'automate à un état antérieur s_0, puis empiler un nouvel état s' tel que $s_0 \xrightarrow{\epsilon^\star A} s'$.

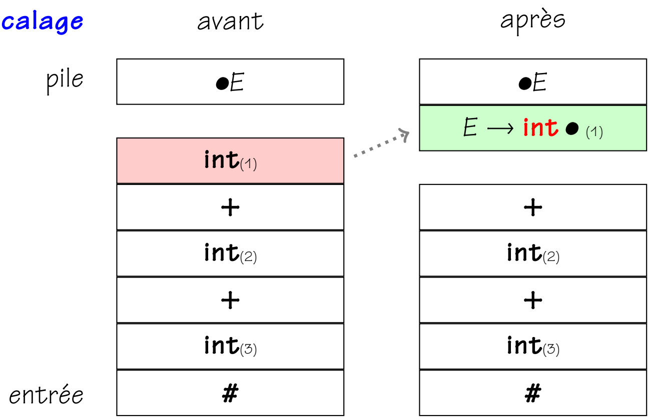
Fonctionnement de l'automate

Voici comment l'automate peut analyser le mot $\mathbf{int}_{(1)} + \mathbf{int}_{(2)} + \mathbf{int}_{(3)}$.

Plusieurs arbres de production correspondent à ce mot. L'automate aura donc *plusieurs* comportements possibles.

Fonctionnement de l'automate

Décalage



int est consommé. Le nouvel état est empilé.

Fonctionnement de l'automate

Réduction	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow$ **int** $\bullet$ $_{(1)}$	
		$E_{(1)}$
	+	+
	int$_{(2)}$	**int**$_{(2)}$
	+	+
	int$_{(3)}$	**int**$_{(3)}$
entrée	#	#

Le membre droit est dépilé, le membre gauche ajouté à l'entrée.

Fonctionnement de l'automate

Décalage	avant	après
pile	$\bullet E$	$\bullet E$
		$E \longrightarrow E \bullet + E_{(1)}$
	$E_{(1)}$	
	$+$	$+$
	int$_{(2)}$	**int**$_{(2)}$
	$+$	$+$
	int$_{(3)}$	**int**$_{(3)}$
entrée	#	#

La réduction comprend le décalage d'un non-terminal (*goto*).

Fonctionnement de l'automate

Décalage

	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet + E_{(1)}$	$E \longrightarrow E \bullet + E_{(1)}$
		$E \longrightarrow E + \bullet E$
	$+$	
	int(2)	**int**(2)
	$+$	$+$
	int(3)	**int**(3)
entrée	#	#

$+$ est consommé. Le nouvel état est empilé.

Fonctionnement de l'automate

Décalage	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet + E_{(1)}$	$E \longrightarrow E \bullet + E_{(1)}$
	$E \longrightarrow E + \bullet E$	$E \longrightarrow E + \bullet E$
		$E \longrightarrow \textbf{int} \bullet {}_{(2)}$
	$\textbf{int}_{(2)}$	
	+	+
	$\textbf{int}_{(3)}$	$\textbf{int}_{(3)}$
entrée	#	#

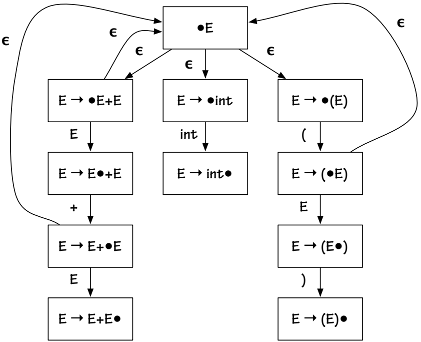
int est consommé. Le nouvel état est empilé.

Fonctionnement de l'automate

Réduction	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet + E_{(1)}$	$E \longrightarrow E \bullet + E_{(1)}$
	$E \longrightarrow E + \bullet E$	$E \longrightarrow E + \bullet E$
	$E \longrightarrow$ **int** $\bullet$ $_{(2)}$	
		$E_{(2)}$
	+	+
	int$_{(3)}$	**int**$_{(3)}$
entrée	#	#

int est réduit en E. Ensuite, *deux possibilités* apparaissent...

Pourquoi deux possibilités apparaissent



Il y a *deux* chemins étiquetés $\epsilon^\star E$ issus de l'état $E \longrightarrow E\mathbf{+}\bullet E$.

Première possibilité

Décalage	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet +E_{(1)}$	$E \longrightarrow E \bullet +E_{(1)}$
	$E \longrightarrow E+\bullet E$	$E \longrightarrow E+\bullet E$
		$E \longrightarrow E+E\bullet_{(2)}$
	$E_{(2)}$	
	$+$	$+$
	int$_{(3)}$	**int**$_{(3)}$
entrée	#	#

Le premier conduit à l'état $E \longrightarrow E{+}E\bullet$, où on peut *réduire*...

Première possibilité

Réduction	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet + E_{(1)}$	
	$E \longrightarrow E + \bullet E$	
	$E \longrightarrow E + E \bullet {}_{(2)}$	$E_{(1+2)}$
	$+$	$+$
	int$_{(3)}$	**int**$_{(3)}$
entrée	#	#

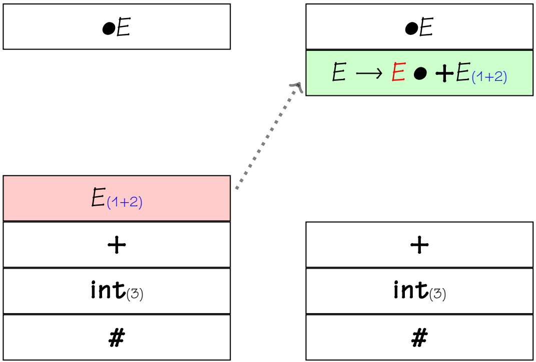
Le membre droit $E + E$ est *réduit* en le membre gauche E.

Première possibilité

Décalage avant après

pile

entrée



Ensuite, pas le choix : il faut décaler E…

Première possibilité

Décalage	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet + E_{(1+2)}$	$E \longrightarrow E \bullet + E_{(1+2)}$
		$E \longrightarrow E + \bullet E$
	$+$	
	int$_{(3)}$	**int**$_{(3)}$
entrée	#	#

...puis décaler +...

Première possibilité

Décalage	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet + E_{(1+2)}$	$E \longrightarrow E \bullet + E_{(1+2)}$
	$E \longrightarrow E + \bullet E$	$E \longrightarrow E + \bullet E$
		$E \longrightarrow$ **int** $\bullet_{(3)}$
	int$_{(3)}$	
entrée	**#**	**#**

...puis décaler **int**...

Première possibilité

Réduction	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet + E_{(1+2)}$	$E \longrightarrow E \bullet + E_{(1+2)}$
	$E \longrightarrow E + \bullet E$	$E \longrightarrow E + \bullet E$
	$E \longrightarrow \mathbf{int} \bullet_{(3)}$	
		$E_{(3)}$
entrée	#	#

...puis réduire la production $E \longrightarrow \mathbf{int}$.

Première possibilité

Décalage	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet + E_{(1+2)}$	$E \longrightarrow E \bullet + E_{(1+2)}$
	$E \longrightarrow E + \bullet E$	$E \longrightarrow E + \bullet E$
		$E \longrightarrow E + E \bullet {}_{(3)}$
	$E_{(3)}$	
entrée	#	#

Encore deux possibilités ; je n'illustre ici que la *"bonne"*.

Première possibilité

Réduction	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet + E_{(1+2)}$	
	$E \longrightarrow E + \bullet E$	
	$E \longrightarrow E + E \bullet {}_{(3)}$	
		$E_{((1+2)+3)}$
entrée	#	#

Une dernière réduction, et l'automate termine avec succès.

Seconde possibilité

Décalage	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet + E_{(1)}$	$E \longrightarrow E \bullet + E_{(1)}$
	$E \longrightarrow E + \bullet E$	$E \longrightarrow E + \bullet E$
		$E \longrightarrow E \bullet + E_{(2)}$
	$E_{(2)}$	
	$+$	$+$
	$\mathbf{int}_{(3)}$	$\mathbf{int}_{(3)}$
entrée	#	#

Le second chemin mène à $E \longrightarrow E \bullet + E$, où on peut *décaler*...

Seconde possibilité

Décalage	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet +E_{(1)}$	$E \longrightarrow E \bullet +E_{(1)}$
	$E \longrightarrow E+ \bullet E$	$E \longrightarrow E+ \bullet E$
	$E \longrightarrow E \bullet +E_{(2)}$	$E \longrightarrow E \bullet +E_{(2)}$
		$E \longrightarrow E+ \bullet E$
	+	
	int$_{(3)}$	**int**$_{(3)}$
entrée	**#**	**#**

Ensuite, pas le choix : il faut décaler **+**...

Seconde possibilité

Décalage	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet + E_{(1)}$	$E \longrightarrow E \bullet + E_{(1)}$
	$E \longrightarrow E + \bullet E$	$E \longrightarrow E + \bullet E$
	$E \longrightarrow E \bullet + E_{(2)}$	$E \longrightarrow E \bullet + E_{(2)}$
	$E \longrightarrow E + \bullet E$	$E \longrightarrow E + \bullet E$
		$E \longrightarrow \mathbf{int} \bullet_{(3)}$
	$\mathbf{int}_{(3)}$	
entrée	#	#

...puis décaler **int**...

Seconde possibilité

Réduction	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet + E_{(1)}$	$E \longrightarrow E \bullet + E_{(1)}$
	$E \longrightarrow E + \bullet E$	$E \longrightarrow E + \bullet E$
	$E \longrightarrow E \bullet + E_{(2)}$	$E \longrightarrow E \bullet + E_{(2)}$
	$E \longrightarrow E + \bullet E$	$E \longrightarrow E + \bullet E$
	$E \longrightarrow \textbf{int} \bullet {}_{(3)}$	$E_{(3)}$
entrée	#	#

...puis réduire **int** en E.

Seconde possibilité

Décalage	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet + E_{(1)}$	$E \longrightarrow E \bullet + E_{(1)}$
	$E \longrightarrow E + \bullet E$	$E \longrightarrow E + \bullet E$
	$E \longrightarrow E \bullet + E_{(2)}$	$E \longrightarrow E \bullet + E_{(2)}$
	$E \longrightarrow E + \bullet E$	$E \longrightarrow E + \bullet E$
		$E \longrightarrow E + E \bullet_{(3)}$
	$E_{(3)}$	
entrée	#	#

Ici, le "bon" choix est d'aller en $E \longrightarrow E+E\bullet$, où on peut *réduire*...

Seconde possibilité

Réduction	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet + E_{(1)}$	$E \longrightarrow E \bullet + E_{(1)}$
	$E \longrightarrow E + \bullet E$	$E \longrightarrow E + \bullet E$
	$E \longrightarrow E \bullet + E_{(2)}$	
	$E \longrightarrow E + \bullet E$	
	$E \longrightarrow E + E \bullet_{(3)}$	$E_{(2+3)}$
entrée	#	#

$E + E$ est réduit en E.

Seconde possibilité

Décalage	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet + E_{(1)}$	$E \longrightarrow E \bullet + E_{(1)}$
	$E \longrightarrow E + \bullet E$	$E \longrightarrow E + \bullet E$
		$E \longrightarrow E + E \bullet\ _{(2+3)}$
	$E_{(2+3)}$	
entrée	#	#

Ici, le “bon” choix est à nouveau d'aller en $E \longrightarrow E + E \bullet$.

Seconde possibilité

Réduction	avant	après
pile	$\bullet E$	$\bullet E$
	$E \longrightarrow E \bullet + E_{(1)}$	
	$E \longrightarrow E + \bullet E$	
	$E \longrightarrow E + E \bullet \ {\scriptstyle (2+3)}$	
		$E_{(1+(2+3))}$
entrée	#	#

À nouveau, $E + E$ est réduit en E, puis l'automate termine.

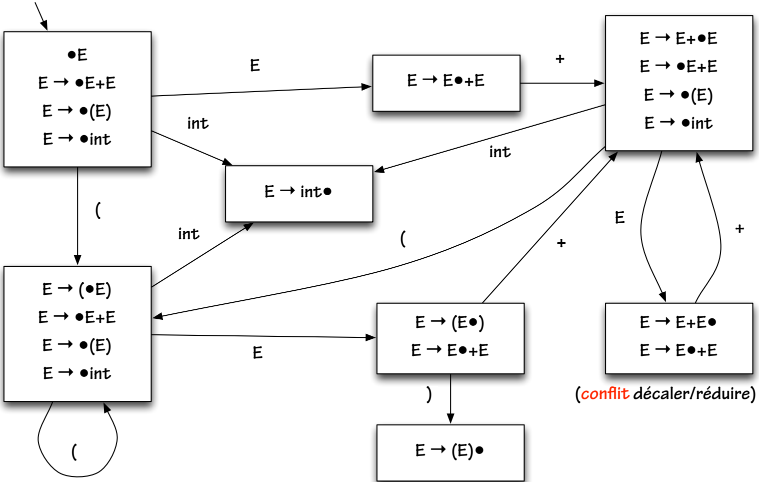
Déterminisation

Une fois cet automate obtenu, on construit un nouvel automate, *sans ϵ-transitions*, dont les états sont des *ensembles d'états* de l'automate initial (powerset construction).

Ce nouvel automate est *bisimilaire* au précédent :

- s'il peut atteindre l'état $\{s_1, \ldots, s_n\}$, alors l'automate original pouvait atteindre n'importe quel état $s \in \{s_1, \ldots, s_n\}$;
- si l'automate original pouvait atteindre s, alors le nouvel automate peut atteindre un état $\{s_1, \ldots, s_n\}$ qui contient s.

Automate LR(0) sans ϵ-transitions



Automate LR(0) sans ϵ-transitions

Ce nouvel automate peut être déterministe ou non.

- s'il l'est, alors la grammaire est certainement *non ambiguë* ; on dit alors qu'elle appartient à la classe LR(0).

S'il ne l'est pas, c'est qu'il y a des *conflits*, ou sources d'ambiguïté résiduelles...

Conflits LR(0)

L'automate sans ϵ-transitions peut exhiber deux sortes de conflits :

- *décaler/réduire* : en un certain état s, l'automate hésite entre interpréter ce qui a déjà été lu ou bien continuer à lire.
- *réduire/réduire* : en un certain état s, l'automate hésite entre deux interprétations de ce qui a déjà été lu.

Nous avons observé un exemple de conflit décaler/réduire.

Il n'y a pas de conflits décaler/décaler : la "powerset construction" les supprime. C'est l'idée de Knuth : *décalons, nous choisirons plus tard*.

Grammaires algébriques

Analyse LL(1)

Analyse LR(0)

Quelques mots de l'analyse LR(1)

L'outil Menhir

De LR(0) à LR(1)

La construction LR(1) s'appuie sur des items de la forme

$$A \longrightarrow \beta \bullet \gamma \; [a]$$

à savoir : "j'ai reconnu un mot issu de β, il me reste à reconnaître un mot issu de γ *et à vérifier que le lexème suivant est a* pour pouvoir affirmer avoir reconnu un mot issu de A".

L'automate LR(1) aura (beaucoup) plus d'états que l'automate LR(0).

Conflits LR(1)

Un automate LR(1) consulte l'état courant s *et le prochain lexème a* pour décider s'il doit décaler ou réduire.

Un conflit décaler/réduire ou réduire/réduire ne peut donc se produire que si deux actions sont possibles pour un même s *et un même a.*

La grammaire appartient à la classe LR(1) si et seulement si l'automate obtenu ne présente *aucun conflit.*

Conflits LR(1)

Notre grammaire simplifiée :

$$E ::= E \mathbf{+} E \mid \mathbf{(}\, E \,\mathbf{)} \mid \mathbf{int}$$

n'appartient à la classe LR(k) pour aucun k, puisqu'elle est ambiguë.

Pour cette grammaire, l'automate LR(1) exhibera donc également un conflit.

Pouvons-nous préciser quel conflit, sans construire l'automate ?

Conflits LR(1)

Lorsque l'automate LR(1) a reconnu $E + E$, il atteint un état qui :

- contient des items de la forme $E \rightarrow E \bullet + E$ [_] ;
- contient l'item $E \rightarrow E + E \bullet$ [+].

Dans cet état, si le prochain symbole d'entrée est +, alors *décaler* et *réduire* sont permis.

Comment savons-nous qu'un tel état existe ?

Conflits LR(1)

Lorsque l'automate LR(1) a reconnu $E + E$, il atteint un état qui :

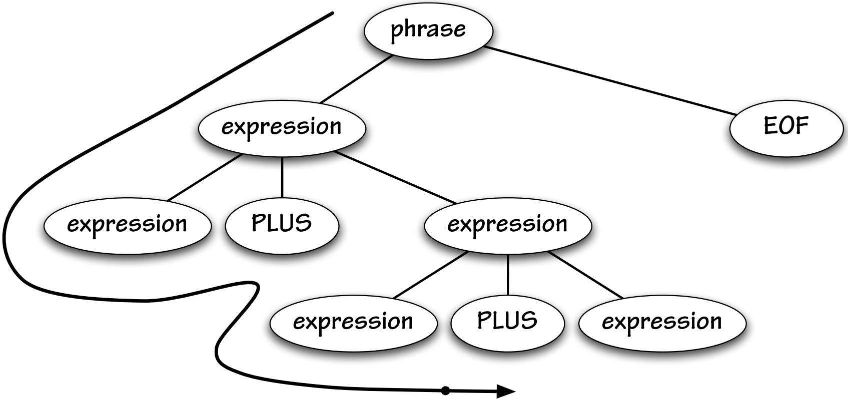
- contient des items de la forme $E \longrightarrow E \bullet + E$ [_] ;
- contient l'item $E \longrightarrow E + E \bullet$ [+].

Dans cet état, si le prochain symbole d'entrée est **+**, alors *décaler* et *réduire* sont permis.

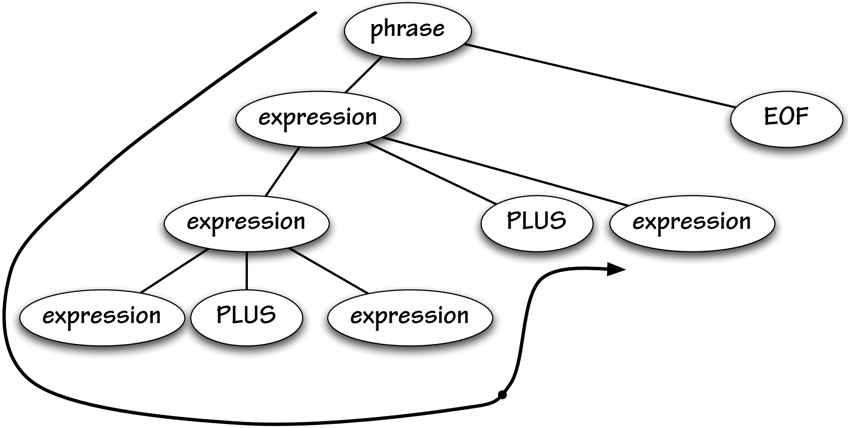
Comment savons-nous qu'un tel état existe ?

Parce qu'il existe *deux arbres de production partiels* qui expliquent la présence de ces items dans cet état...

Pourquoi décaler est permis



Pourquoi réduire est permis



Une autre caractérisation de la classe LR(1)

Ces deux arbres partiels ont un même *préfixe de frange :* $E + E \bullet +$.

Leur existence suffit à prouver que la grammaire n'est pas LR(1).

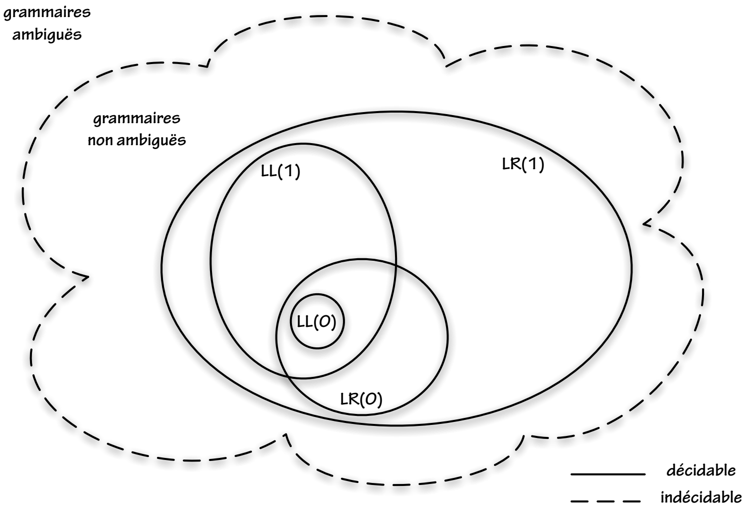
On peut en fait *définir* la classe LR(1) en termes d'arbres de production partiels, *sans parler d'automates.*

Pourquoi LR(1) ?

En résumé, l'approche LR(1) :

- est *puissante*, strictement plus puissante que LL(1), et nécessite en général peu de transformations de la grammaire initiale ;
- mais exige de comprendre la notion de *conflit*.

Quelques classes de grammaires



Grammaires algébriques

Analyse LL(1)

Analyse LR(0)

Quelques mots de l'analyse LR(1)

L'outil Menhir

Menhir

Menhir est un *générateur d'analyseurs syntaxiques* : il transforme une spécification de grammaire en un analyseur écrit en OCaml.

Menhir utilise une version optimisée de la construction LR(1), suivant un algorithme dû à Pager (1977).

Valeurs sémantiques

Un analyseur ne se contente pas d'indiquer si la suite de lexèmes appartient ou non à la grammaire : il produit une *valeur sémantique*, en général un arbre de syntaxe abstraite.

La spécification doit donc contenir des fragments de code OCaml, appelés *actions sémantiques*, qui indiquent comment construire cette valeur sémantique.

Valeurs sémantiques

Tout symbole, terminal ou non-terminal, a une valeur sémantique, dont le type est choisi par le programmeur.

Pour un terminal, on doit déclarer ce type. On écrit :

```
%token<int> INT
```

Pour un non-terminal, on peut déclarer ce type ; sinon, il est inféré.

```
%type<int> expression
```

Valeurs sémantiques

Une production :

$$A \longrightarrow BC$$

s'écrira, pour Menhir, sous la forme :

```
A:
  b = B
  c = C
    { ... (* expression ocaml qui utilise b et c *) ... }
```

Le code indique comment calculer la valeur sémantique associée à A à partir de celles associées à B et C.

Exemple de spécification

Voici notre minuscule grammaire (fichier **.mly**) :

```
%token PLUS LPAR RPAR EOF
%token <int> INT
%start <int> phrase
%%
expression:
  e1 = expression; PLUS; e2 = expression { e1 + e2 }
| LPAR; e = expression; RPAR             { e }
| i = INT                                { i }

phrase:
  e = expression; EOF                    { e }
```

Comment résoudre un conflit LR(1) ?

Cette minuscule grammaire est ambiguë, donc ni LR(0) ni LR(1). On a un conflit *décaler/réduire* après avoir lu $E + E$ et lorsque le premier symbole de l'entrée est $+$.

Comment comprendre un conflit

Menhir décrit les conflits de deux façons :

- dans le fichier **.automaton**, il *décrit* l'automate obtenu et indique quels états présentent un conflit ;
- dans le fichier **.conflicts**, il *explique* les conflits : “pour telle séquence de symboles, on peut construire *deux* arbres de production partiels, que voici”.

Comment comprendre un conflit

Voici une explication proposée par Menhir (page 1/4) :

```
** Conflict (shift/reduce) in state 6.
** Token involved: PLUS
** This state is reached from phrase after reading:

expression PLUS expression
```

Comme prévu, le conflit se produit après avoir lu un début de *phrase* de la forme $E + E$ et lorsque le premier symbole de l'entrée est $+$.

Comment comprendre un conflit

Voici une explication proposée par Menhir (page 2/4) :

```
** The derivations below have the following common factor:
** The hole (?) is where they begin to differ.

phrase
expression EOF
(?)
```

Menhir s'apprête à exhiber deux arbres de production possibles, et commence par en présenter la partie commune.

Comment comprendre un conflit

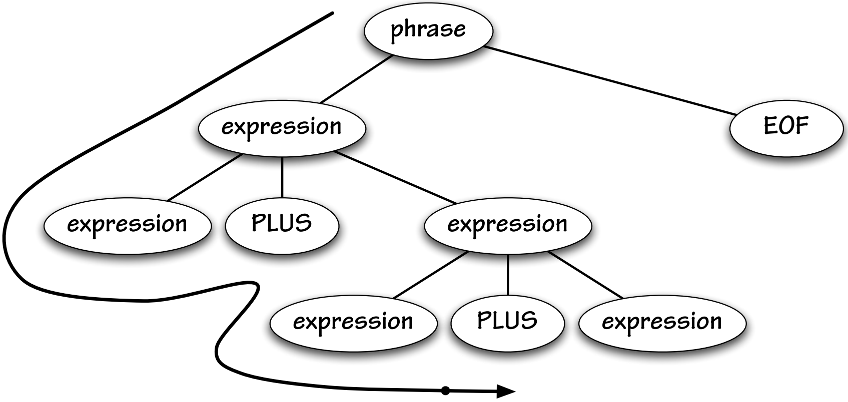
Menhir explique ensuite pourquoi *décaler* est permis (page 3/4) :

```
** In state 6, looking ahead at PLUS, shifting is permitted
** because of the following sub-derivation:

expression PLUS expression
                expression . PLUS expression
```

Ceci doit être lu comme un *arbre de production* dont la *frange* commence par $E + E +$.

Pourquoi décaler est permis



Comment comprendre un conflit

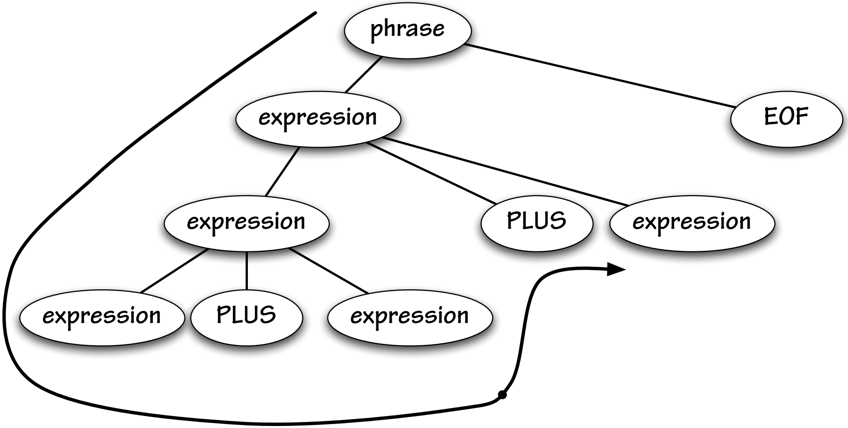
Enfin, Menhir explique pourquoi *réduire* est permis (page 4/4) :

```
** In state 6, looking ahead at PLUS, reducing production
** expression -> expression PLUS expression
** is permitted because of the following sub-derivation:

expression PLUS expression // lookahead token appears
expression PLUS expression .
```

Ceci constitue un second arbre de production dont la frange commence *également* par $E + E +$.

Pourquoi réduire est permis



Comment résoudre un conflit LR(1) ?

On peut résoudre ce conflit de deux façons :

- Réécrire la grammaire pour utiliser deux non-terminaux E et T ; technique exposée précédemment.
- Sans modifier la grammaire, indiquer *manuellement* si l'automate doit préférer *réduire* ou *décaler*. L'un de ces choix rend l'opérateur **+** associatif à gauche, l'autre le rend associatif à droite. Pourquoi ?

La seconde solution *sort* du cadre strict des grammaires algébriques, pour gagner un peu de confort.

Comment supprimer un conflit

Pour choisir entre *réduire* et *décaler*, Menhir adopte une convention héritée de *yacc* (1970) : il compare la *priorité* de la production à réduire avec celle du lexème à décaler.

La *priorité d'une production* est, par défaut, la priorité du lexème situé le plus à droite dans son membre droit.

La *priorité d'un lexème* lui est attribuée par l'utilisateur à l'aide de déclarations explicites.

Comment supprimer un conflit

Si la priorité de la *production* est supérieure, l'analyseur préfère *réduire*.

Si la priorité du *lexème* est supérieure, l'analyseur préfère *décaler*.

Si tous deux sont situés au *même* niveau de priorité, l'analyseur préfère *réduire* si ce niveau a été déclaré associatif à *gauche* et *décaler* si ce niveau a été déclaré associatif à *droite*.

Si l'une de ces deux priorités est *indéfinie*, aucun choix n'est effectué et le conflit est signalé.

Exemple de déclaration d'associativité

Voici donc comment éviter notre conflit. On ajoute la ligne suivante au fichier **.mly** :

```
%left PLUS
```

La production $E \longrightarrow E\ \mathbf{+}\ E$ et le lexème **+** sont au même niveau, que nous déclarons associatif à gauche. L'automate préfère donc *réduire*.

Exemple de déclaration de priorité

Pour une grammaire un peu moins simpliste, on aurait pu avoir besoin de plusieurs niveaux :

```
%left MINUS PLUS
%left TIMES SLASH
```

On déclare ici *deux* niveaux de priorité distincts, un par ligne. Par convention, les niveaux sont déclarés par ordre de priorité *croissante*. Deux lexèmes sont associés à chacun de ces niveaux.

Prudence !

Les déclarations de priorité et d'associativité constituent un mécanisme délicat, *difficile à maîtriser*, en dehors de quelques cas simples comme le précédent.

En pratique, il faut l'utiliser peu ou pas du tout, et préférer *réécrire la grammaire* pour qu'elle appartienne à la classe LR(1).

Conclusion

L'analyse syntaxique fournit un bel exemple de *code* — l'analyseur — produit automatiquement à partir d'une *spécification* — la grammaire.

De plus, dans le cas de LL(1) et LR(1), si la construction réussit, alors la grammaire est garantie *non ambiguë* — un théorème gratuit !