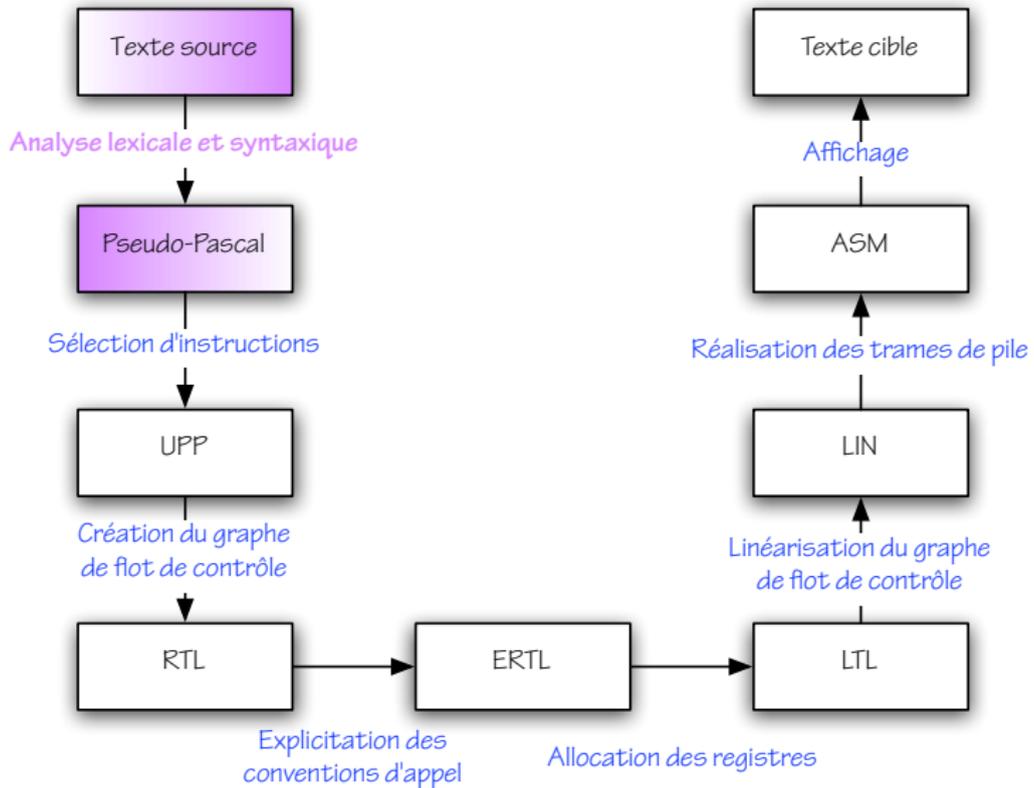


# Compilation (INF 553)

Analyse syntaxique

François Pottier

7 février 2007



Analyses lexicale et syntaxique

Grammaires algébriques

Analyse LL(1)

Analyse LR(1)

L'outil Menhir

Projets

## Analyse lexicale et analyse syntaxique

*L'analyse lexicale* (le «*lexer*» ou «*scanner*») transforme une suite de caractères en une suite d'entités de plus haut niveau appelées *lexèmes* («*tokens*»).

*L'analyse syntaxique* ou *analyse grammaticale* (le «*parser*») transforme une suite de lexèmes en un arbre de syntaxe abstraite.

## Pourquoi séparer ces deux phases ?

Un analyseur lexical est typiquement spécifié à l'aide *d'expressions régulières* (outils: *lex*, *ocamllex*), tandis qu'un analyseur syntaxique est typiquement spécifié à l'aide d'une *grammaire non contextuelle* (« context-free ») appartenant à une *classe* particulière: LL(1) (*JavaCC*), LALR(1) (*yacc*, *bison*, *ocamlyacc*), LR(1) (*Menhir*), etc.

Il existe toutefois des formalismes et outils permettant de combiner ces deux aspects en une seule spécification (*SDF2*). On parle alors de « *scannerless parsing* ».

## Pipelining

Bien que conceptuellement séparées, analyses lexicale et syntaxique sont habituellement « *pipelinées* ». L'analyseur lexical fournit chaque lexème *sur demande* de l'analyseur syntaxique, ce qui évite de construire en mémoire l'intégralité de la suite de lexèmes.

Les deux analyses sont donc exécutées de façon *entremêlée*.

## Note

L'analyse lexicale pour Pseudo-Pascal est sans grande difficulté. *Je n'en parlerai pas* afin de consacrer notre temps à des aspects plus intéressants du compilateur.

Le chapitre 4 du poly de Luc Maranget y est consacré.

Analyses lexicale et syntaxique

Grammaires algébriques

Analyse LL(1)

Analyse LR(1)

L'outil Menhir

Projets

## Grammaires algébriques

Une *grammaire algébrique* ou *non contextuelle* est donnée par un quadruplet  $(\Sigma, V, S, P)$ , où:

- ▶  $\Sigma$  est l'alphabet des *symboles terminaux*, notés  $a, b, \text{etc.}$  Les symboles terminaux sont typiquement les lexèmes produits par l'analyseur syntaxique.
- ▶  $V$  est un ensemble de *symboles non-terminaux*, notés  $A, B, \text{etc.}$
- ▶  $S \in V$  est le *symbole de départ*.
- ▶  $P$  est un ensemble de *productions* de la forme  $A \rightarrow w$ , où  $w$  dénote un *mot* sur l'alphabet  $\Sigma \cup V$ .

## Exemple de grammaire algébrique

Soient:

- ▶  $\Sigma = \{\text{int}, (, ), +, -, *, /\}$ .
- ▶  $V = \{E\}$ .
- ▶  $S = E$ .
- ▶ L'ensemble  $P$  des productions est:

$$E \rightarrow E + E$$

$$E \rightarrow E - E$$

$$E \rightarrow E * E$$

$$E \rightarrow E / E$$

$$E \rightarrow (E)$$

$$E \rightarrow \text{int}$$

## Langage engendré par une grammaire

Une grammaire  $G$  définit un *langage*  $L(G)$  sur l'alphabet  $\Sigma$ , dont les éléments sont les mots *engendrés par dérivation* à partir du symbole de départ  $S$ .

On pose

$$uAv \rightarrow uvw \quad \text{si } A \rightarrow w \in P$$

Le langage  $L(G)$  est alors défini par

$$L(G) = \{w \in \Sigma^* \mid S \rightarrow^* w\}$$

## Exemples de dérivations

Voici trois dérivations menant au mot `int + int * int`.

$$\underline{E} \rightarrow E + \underline{E} \rightarrow \underline{E} + E * E \rightarrow \text{int} + \underline{E} * E \rightarrow \text{int} + \text{int} * \underline{E} \rightarrow \text{int} + \text{int} * \text{int}$$

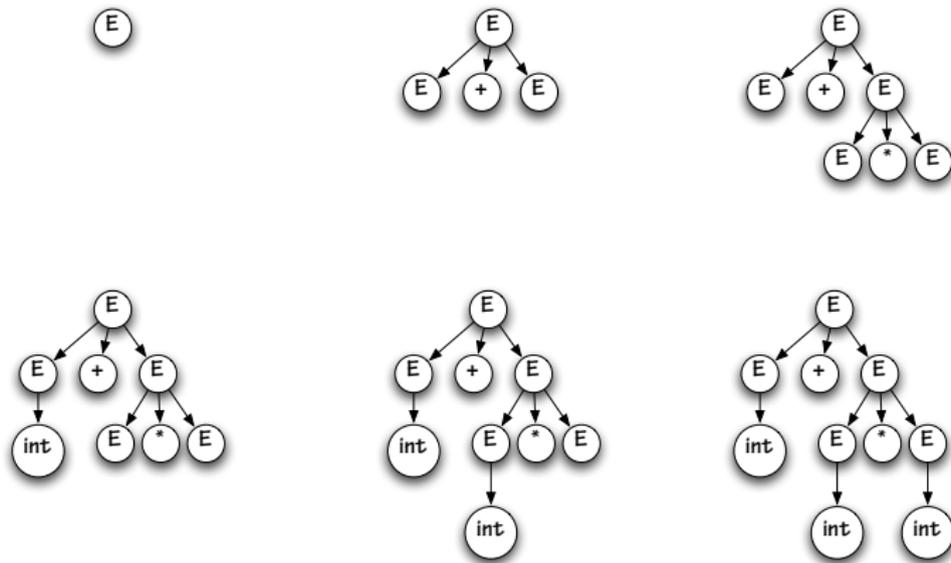
$$\underline{E} \rightarrow \underline{E} * E \rightarrow \underline{E} + E * E \rightarrow \text{int} + \underline{E} * E \rightarrow \text{int} + \text{int} * \underline{E} \rightarrow \text{int} + \text{int} * \text{int}$$

$$\underline{E} \rightarrow \underline{E} + E \rightarrow \text{int} + \underline{E} \rightarrow \text{int} + \underline{E} * E \rightarrow \text{int} + \text{int} * \underline{E} \rightarrow \text{int} + \text{int} * \text{int}$$

Lesquelles sont « équivalentes » et pourquoi ?

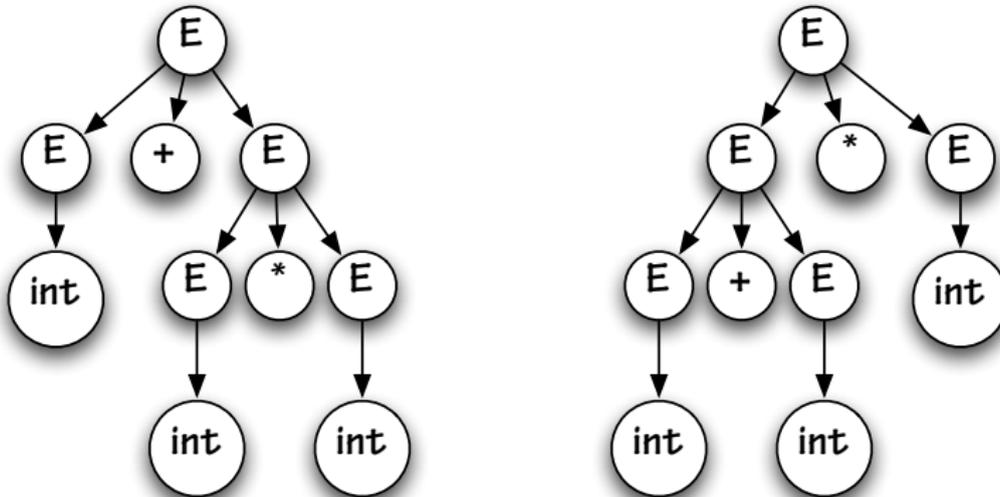
## Arbres de dérivation

On peut préférer parler de dérivation en termes *d'arbres* et non en termes de mots. Par exemple:



## Arbres de dérivation

Il existe *deux* arbres de dérivation pour le mot `int + int * int`:



# Ambiguïté

L'existence de plus d'un arbre de dérivation pour un même mot signifie que la grammaire est *ambiguë*.

Quelles sont, pour cette grammaires des expressions arithmétiques, les sources d'ambiguïté?

## Pourquoi éviter l'ambiguïté

L'ambiguïté est clairement nuisible d'un point de vue *sémantique*. De plus, un analyseur syntaxique déterministe sera plus *efficace*. On souhaite donc en général *interdire* les grammaires ambiguës.

La question de savoir si une grammaire algébrique donnée est ou non ambiguë étant *indécidable*, on s'intéressera souvent à des classes décidables plus restreintes: LL(1), LR(1), etc.

## Comment éviter l'ambiguïté: exemple

Dans le cas de la grammaire des expressions arithmétiques, on peut éviter l'ambiguïté en modifiant la grammaire. On se donne trois non-terminaux  $E, T, F$ , pour *expressions*, *termes* et *facteurs*:

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

Le mot  $\mathbf{int + int * int}$  n'admet qu'un seul arbre de dérivation.

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

## Remarque

Cette nouvelle grammaire est *non ambiguë*. Elle appartient en fait à la classe LR(1).

Le langage qu'elle engendre est *le même* que celui engendré par la grammaire précédente.

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

Analyses lexicale et syntaxique

Grammaires algébriques

Analyse LL(1)

Analyse LR(1)

L'outil Menhir

Projets

# Principe

L'analyse LL(1), dite également *réursive descendante*, consiste à *transcrire* la grammaire de façon quasi littérale en un programme d'analyse syntaxique:

- ▶ chaque non-terminal  $A$  devient une *fonction* de même nom;
- ▶ la fonction  $A$  consulte le *premier* élément du flot de lexèmes puis décide laquelle des productions associées à  $A$  développer;
- ▶ le développement d'une production  $A \rightarrow w$  se fait en reconnaissant successivement tous les éléments du mot  $w$ :
  - ▶ un terminal  $a$  est reconnu par extraction du flot de lexèmes;
  - ▶ un non-terminal  $B$  est reconnu à l'aide d'un *appel récursif* à  $B$ .

## Exemple

Cette approche est-elle applicable à notre grammaire ?

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

Peut-on la transcrire en un analyseur récursif descendant ?

## Nouvelle modification

Pour contourner ce problème, *modifions* encore une fois la grammaire.

Une expression  $E$  commence nécessairement par un terme, suivi d'un reste  $E'$ . Exploitions cela:

$$\begin{array}{ll}
 E \rightarrow T E' & T \rightarrow F T' \\
 E' \rightarrow + T E' & T' \rightarrow * F T' \\
 E' \rightarrow - T E' & T' \rightarrow / F T' \\
 E' \rightarrow \epsilon & T' \rightarrow \epsilon \\
 F \rightarrow \mathbf{int} & F \rightarrow (E)
 \end{array}$$

Peut-on la transcrire en un analyseur récursif descendant?

## Construction systématique d'un analyseur LL(1)

Un analyseur LL(1) doit pouvoir *décider*, étant donné un non-terminal  $A$  à développer et un premier lexème lu  $a$ , quelle production développer.

La règle générale est:

- ▶ Une production  $A \rightarrow w$  peut être développée si  $a \in \text{FIRST}(w)$ .
- ▶ Une production  $A \rightarrow w$  peut être développée si  $w \rightarrow^* \epsilon$  et  $a \in \text{FOLLOW}(A)$ .

La grammaire appartient à la classe **LL(1)** si et seulement si un tel analyseur est *déterministe*, c'est-à-dire si au plus une décision est permise pour tous  $A$  et  $a$ .

## Quelques définitions

L'assertion  $a \in \text{FIRST}(w)$  est vraie si un mot dérivé de  $w$  peut *commencer* par le symbole  $a$ :

$$\text{FIRST}(w) = \{a \in \Sigma \mid \exists v \quad w \rightarrow^* av\}$$

L'assertion  $a \in \text{FOLLOW}(A)$  est vraie si un mot dérivé de  $A$  peut *être suivi* du symbole  $a$ :

$$\text{FOLLOW}(A) = \{a \in \Sigma \mid \exists u, v \quad S \rightarrow^* uAav\}$$

Ces ensembles sont calculables en temps polynomial à l'aide d'un algorithme itératif.

## Retour à notre exemple

La dernière version de notre grammaire *appartient* à la classe LL(1):

$$\begin{array}{ll}
 E \rightarrow T E' & T \rightarrow F T' \\
 E' \rightarrow + T E' & T' \rightarrow * F T' \\
 E' \rightarrow - T E' & T' \rightarrow / F T' \\
 E' \rightarrow \epsilon & T' \rightarrow \epsilon \\
 F \rightarrow \mathbf{int} & F \rightarrow (E)
 \end{array}$$

Les ensembles

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

sont deux à deux *disjoints*, ce qui justifie qu'un seul développement de  $T'$  est possible si un lexème de prévision est connu.

## Pourquoi LL(1)?

En résumé, l'approche LL(1):

- ▶ est conceptuellement *simple*: on pourrait presque transcrire manuellement une grammaire LL(1) en un analyseur.
- ▶ mais nécessite des *transformations* si la grammaire initiale présente des facteurs à gauche ou une récursivité à gauche.

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

Analyses lexicale et syntaxique

Grammaires algébriques

Analyse LL(1)

Analyse LR(1)

L'outil Menhir

Projets

# Principe

L'approche LR est fondée sur les *automates finis à pile*.

On modélise d'abord l'analyse grammaticale par un automate fini à pile *non déterministe*, dont la construction est simple.

Ensuite, on tente de construire un automate *déterministe* équivalent, donc capable d'explorer « en parallèle » différents choix, et ce même si la grammaire présente des facteurs à gauche ou une récursivité à gauche.

## Principe

Si l'automate obtenu est *effectivement* déterministe, alors la grammaire appartient à la classe LR(k), où k est le nombre de lexèmes de prévision utilisés pour la construction de l'automate.

Dans ce qui suit, j'illustre la construction pour  $k=0$ , puis évoque brièvement le cas  $k=1$ .

## Construction LR(0)

Un *automate fini* est un graphe constitué d'un ensemble fini de sommets appelés *états* et d'arêtes étiquetées appelées *transitions*.

Pour la construction LR(0), les *états* sont étiquetés de l'une des deux façons suivantes:

- ▶  $\bullet A$ , « je m'apprête à reconnaître un mot dérivé de  $A$  »;
- ▶  $A \rightarrow u \bullet v$ , « j'ai reconnu un mot dérivé de  $u$ , il me reste à reconnaître un mot dérivé de  $v$  pour pouvoir affirmer avoir reconnu un mot dérivé de  $A$  ».

Les *transitions* sont étiquetées par des *symboles* (terminaux ou non) ou bien par  $\epsilon$ .

## Exemple

Illustrons la construction LR(0) pour cette grammaire simplifiée:

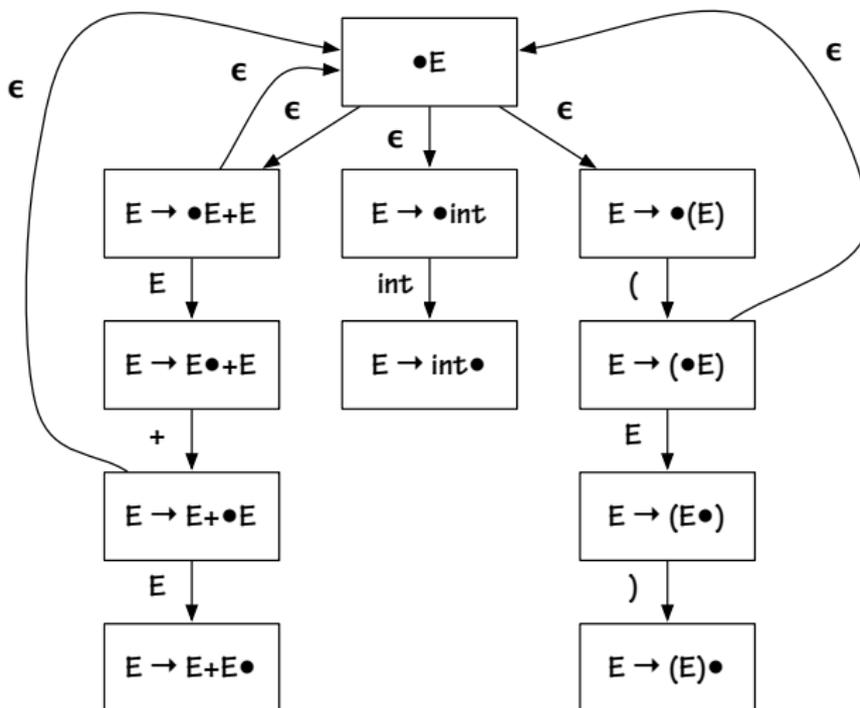
$$E \rightarrow E + E$$

$$E \rightarrow (E)$$

$$E \rightarrow \mathbf{int}$$

Puisque cette grammaire est *ambiguë*, la construction ne mènera *certainement pas* à un automate déterministe... Nous verrons apparaître un *conflit*.

# Automate LR(0) non déterministe



## Interprétation de l'automate

L'automate maintient à tout instant une *pile d'états*, dont le sommet constitue *l'état courant*, ainsi qu'un *flux de lexèmes*.

L'automate peut effectuer des actions de deux natures:

- ▶ *décaler*: si le lexème de tête est  $a$ , l'automate peut retirer  $a$  du flux et empiler un nouvel état  $s'$  accessible à partir de l'état courant  $s$  à travers un chemin étiqueté  $\epsilon^*a$ .
- ▶ *réduire*: si l'état courant  $s$  est étiqueté  $A \rightarrow w\bullet$ , l'automate peut dépiler  $|w|$  éléments, ce qui découvre un état antérieur  $s_0$ , puis empiler un nouvel état  $s'$  accessible à partir de  $s_0$  à travers un chemin étiqueté  $\epsilon^*A$ .

## Exemple d'exécution

Voici comment l'automate peut analyser le mot **int + int + int**:

action	pile	flux
	$(\bullet E)$	int + int + int
décaler <b>int</b>	$(\bullet E) (E \rightarrow \mathbf{int}\bullet)$	+ int + int
réduire	$(\bullet E)$	E + int + int
décaler E	$(\bullet E) (E \rightarrow E \bullet + E)$	+ int + int
décaler <b>+</b>	$(\bullet E) (E \rightarrow E \bullet + E) (E \rightarrow E + \bullet E)$	int + int
décaler <b>int</b>	$(\bullet E) (E \rightarrow E \bullet + E) (E \rightarrow E + \bullet E) (E \rightarrow \mathbf{int}\bullet)$	+ int
réduire	$(\bullet E) (E \rightarrow E \bullet + E) (E \rightarrow E + \bullet E)$	E + int

À ce point, l'automate peut continuer de *deux façons* possibles...  
lesquelles et pourquoi?

## Exemple d'exécution

Première possibilité: *réduire*.

action	pile	flux
	$(\bullet E) (E \rightarrow E \bullet + E) (E \rightarrow E + \bullet E)$	$E + \text{int}$
décaler $E$	$(\bullet E) (E \rightarrow E \bullet + E) (E \rightarrow E + \bullet E) (E \rightarrow E + E \bullet)$	$+ \text{int}$
réduire	$(\bullet E)$	$E + \text{int}$
	...	

Seconde possibilité: *décaler*.

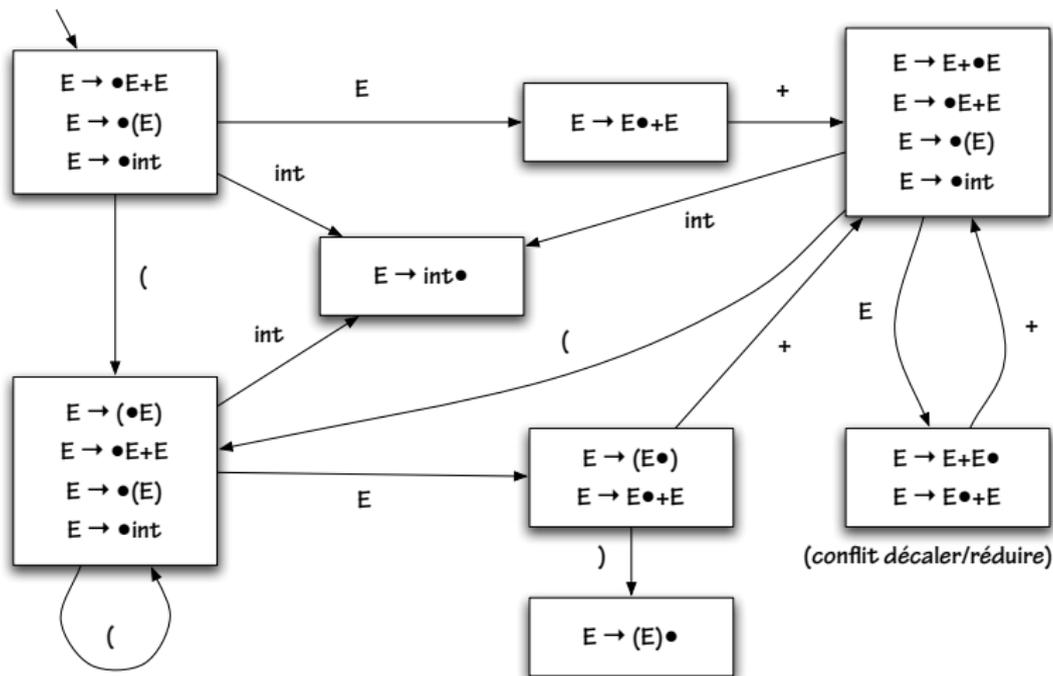
action	pile	flux
	$(\bullet E) (\dots) (E \rightarrow E + \bullet E)$	$E + \text{int}$
décaler $E$	$(\bullet E) (\dots) (E \rightarrow E + \bullet E) (E \rightarrow E \bullet + E)$	$+ \text{int}$
décaler $+$	$(\bullet E) (\dots) (E \rightarrow E + \bullet E) (E \rightarrow E \bullet + E) (E \rightarrow E + \bullet E)$	$\text{int}$
	...	

## Déterminisation

Une fois cet automate obtenu, on tente d'en éliminer le non-déterminisme en construisant un nouvel automate, *sans  $\epsilon$ -transitions*, dont les états correspondent à des *ensembles d'états* de l'automate initial (« powerset construction »).

Le nouvel automate *simule* le précédent: étant donnée une suite de lexèmes fixés, il atteint l'état  $\{s_1, \dots, s_n\}$  si et seulement l'automate original pouvait atteindre l'état  $s_1$  *ou* ... *ou*  $s_n$ .

# Automate LR(0) sans $\epsilon$ -transitions



## Conflits LR(0)

L'automate sans  $\epsilon$ -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.

Les conflits décaler/décaler *n'existent pas* car la « powerset construction » est précisément conçue pour les supprimer.

## De LR(0) à LR(1)

Pour supprimer certains conflits, on peut effectuer une construction légèrement plus complexe, en prenant en compte *un* symbole de prévision: c'est la technique *LR(1)*.

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

$$A \rightarrow u \bullet v [a]$$

à savoir: « j'ai reconnu un mot dérivé de  $u$ , il me reste à reconnaître un mot dérivé de  $v$  *et à vérifier que le lexème suivant est  $a$*  pour pouvoir affirmer avoir reconnu un mot dérivé 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 lexème  $a$  situé en tête de flux* 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*.

## LR(1) en pratique

La classe LR(1) *contient strictement* la classe LL(1).

La construction LR(1) est *coûteuse* en pratique, d'où l'existence d'approximations: SLR(1), LALR(1).

Ces approximations engendrent malheureusement des conflits *«artificiels»* difficiles à expliquer.

**Menhir** effectue une version optimisée, *non approchée*, de la construction LR(1).

## Comment résoudre un conflit LR(1)?

Revenons à notre grammaire simplifiée:

$$E \rightarrow E + E$$

$$E \rightarrow (E)$$

$$E \rightarrow \mathbf{int}$$

Cette grammaire est ambiguë, donc ni LR(0) ni LR(1). On a un conflit *décaler/réduire* après avoir lu  $E + E$  et lorsque le lexème de prévision est  $+$ .

## 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.

## Pourquoi LR(1)?

En résumé, l'approche LR(1):

- ▶ est *puissante* et nécessite en général peu de transformations de la grammaire initiale.
- ▶ mais exige une bonne compréhension de la notion de *conflit*.

Analyses lexicale et syntaxique

Grammaires algébriques

Analyse LL(1)

Analyse LR(1)

L'outil Menhir

Projets

# Menhir

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

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 Objective Caml, appelés *actions sémantiques*, qui indiquent comment construire cette valeur sémantique.

## Exemple de spécification

Voici notre grammaire ambiguë (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 comprendre un conflit

Menhir rapporte les conflits de deux façons:

- ▶ dans le fichier **.automaton**, il *décrit* l'automate 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 dérivation, que voici ».

## Comment comprendre un conflit

Voici une explication proposée par Menhir (page 1/3):

```
** 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 lexème de prévision est  $+$ .

## Comment comprendre un conflit

Menhir explique ensuite pourquoi *décaler* est permis (page 2/3):

```
** In state 6, looking ahead at PLUS, shifting is permitted:
```

```
phrase
```

```
expression EOF
```

```
expression PLUS expression
```

```
expression . PLUS expression
```

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

## Comment comprendre un conflit

Enfin, Menhir explique pourquoi *réduire* est permis (page 3/3):

```
** In state 6, looking ahead at PLUS, reducing production
** expression → expression PLUS expression
** is permitted:
```

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

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

## Comment supprimer un conflit

Pour choisir entre *réduire* et *décaler*, Menhir adopte une convention héritée de *yacc*: 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  $E \rightarrow E + E$  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 avec parcimonie, et préférer le plus souvent *réécrire la grammaire* pour s'approcher autant que possible de la classe LR(1).

Analyses lexicale et syntaxique

Grammaires algébriques

Analyse LL(1)

Analyse LR(1)

L'outil Menhir

Projets

## Sujets proposés

Je vous propose quatre sujets:

- ▶ Ajout de *types enregistrements* au langage Pseudo-Pascal (facile);
- ▶ Production d'*assembleur Intel x86* (difficile);
- ▶ Ajout d'*exceptions* au langage Pseudo-Pascal (difficile);
- ▶ Implémentation puis élimination des *tests aux bornes de tableaux* (difficile).

Plus de détails en ligne prochainement.