

École Polytechnique

# INF564 – Compilation

Jean-Christophe Filliâtre

analyse lexicale et syntaxique

l'objectif de l'analyse syntaxique est de reconnaître les phrases appartenant à la syntaxe du langage

son entrée est la syntaxe concrète, c'est-à-dire une suite de caractères, et sa sortie est la syntaxe abstraite

on découpe ce travail en deux étapes

- l'**analyse lexicale**, qui découpe le texte source en « mots » appelés lexèmes (*tokens*)
- l'**analyse syntaxique** proprement dite, qui reconnaît les suites de mots légales

source = suite de caractères

```
fun x -> (* ma fonction *)
x+1
```

↓  
**analyse lexicale**  
 ↓

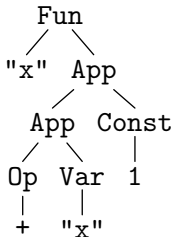
suite de lexèmes

fun	x	->	x	+	1
-----	---	----	---	---	---

⋮

⋮  
 ↓  
**analyse syntaxique**  
 ↓

syntaxe abstraite



---

## analyse lexicale

les blancs (espace, retour chariot, tabulation, etc.) jouent un rôle dans l'analyse lexicale ; ils permettent notamment de séparer deux lexèmes

ainsi `funx` est compris comme un seul lexème (l'identificateur `funx`) et `fun x` est compris comme deux lexèmes (le mot clé `fun` et l'identificateur `x`)

de nombreux blancs sont néanmoins inutiles (comme dans `x + 1`) et simplement ignorés

les blancs n'apparaissent pas dans le flot de lexèmes renvoyé

les conventions diffèrent selon les langages,  
et certains des caractères « blancs » peuvent être significatifs

exemples :

- les tabulations pour `make`
- retours chariot et espaces de début de ligne en Python ou en Haskell (l'indentation détermine la structure des blocs)

les commentaires jouent le rôle de blancs

```
fun(* et hop *)x -> x + (* j'ajoute un *) 1
```

ici le commentaire `(* et hop *)` joue le rôle d'un blanc significatif (sépare deux lexèmes) et le commentaire `(* j'ajoute un *)` celui d'un blanc inutile

note : les commentaires sont parfois exploités par certains outils (javadoc, ocamlloc, etc.), qui les traitent alors différemment dans leur propre analyse lexicale

```
val length : 'a list -> int  
(** Return the length (number of elements) of ...
```

pour réaliser l'analyse lexicale, on va utiliser

- des **expressions régulières** pour décrire les lexèmes
- des **automates finis** pour les reconnaître

on exploite notamment la capacité à construire automatiquement un automate fini déterministe reconnaissant le langage décrit par une expression régulière



---

## expressions régulières

on se donne un alphabet  $A$

$r ::=$	$\emptyset$	langage vide
	$\epsilon$	mot vide
	$a$	caractère $a \in A$
	$rr$	concaténation
	$r r$	alternative
	$r^*$	étoile

conventions : l'étoile a la priorité la plus forte, puis la concaténation, puis enfin l'alternative

le **langage** défini par l'expression régulière  $r$  est l'ensemble de mots  $L(r)$  défini par

$$L(\emptyset) = \emptyset$$

$$L(\epsilon) = \{\epsilon\}$$

$$L(a) = \{a\}$$

$$L(r_1 r_2) = \{w_1 w_2 \mid w_1 \in L(r_1) \wedge w_2 \in L(r_2)\}$$

$$L(r_1 \mid r_2) = L(r_1) \cup L(r_2)$$

$$L(r^*) = \bigcup_{n \geq 0} L(r^n) \quad \text{où } r^0 = \epsilon, \quad r^{n+1} = r r^n$$

sur l'alphabet  $\{a, b\}$

- mots de trois lettres

$$(a|b)(a|b)(a|b)$$

- mots se terminant par un  $a$

$$(a|b) \star a$$

- mots alternant  $a$  et  $b$

$$(b|\epsilon)(ab) \star (a|\epsilon)$$

constantes entières décimales, éventuellement précédées de zéros

$$(0|1|2|3|4|5|6|7|8|9) (0|1|2|3|4|5|6|7|8|9)^*$$

identificateurs composés de lettres, de chiffres et du souligné, et commençant par une lettre

$$(a|b|\dots|z|A|B|\dots|Z)(a|b|\dots|z|A|B|\dots|Z|_|0|1|\dots|9)^*$$

constantes flottantes (3.14 2. 1e-12 6.02e23 etc.)

$$d d^* (.d^* | (\epsilon | .d^*)) (e|E) (\epsilon | + | -) d d^*$$

avec  $d = 0|1|\dots|9$

les commentaires de la forme  $( * \dots * )$ , **non imbriqués**, peuvent également être définis de cette manière

$$\boxed{(} \boxed{*} \boxed{(} \boxed{*} r_1 \mid r_2 \boxed{)} \boxed{*} \boxed{*} \boxed{*} \boxed{)}$$

où  $r_1$  = tous les caractères sauf  $*$  et  $)$

et  $r_2$  = tous les caractères sauf  $*$



les expressions régulières ne sont pas assez expressives pour définir les commentaires **imbriqués** (le langage des mots bien parenthésés n'est pas régulier)

on expliquera plus loin comment contourner ce problème

---

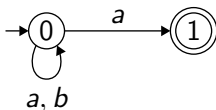
## automates finis

## Définition

Un automate fini sur un alphabet  $A$  est un quadruplet  $(Q, T, I, F)$  où

- $Q$  est un ensemble fini d'états
- $T \subseteq Q \times A \times Q$  un ensemble de transitions
- $I \subseteq Q$  un ensemble d'états initiaux
- $F \subseteq Q$  un ensemble d'états terminaux

exemple :  $Q = \{0, 1\}$ ,  $T = \{(0, a, 0), (0, b, 0), (0, a, 1)\}$ ,  $I = \{0\}$ ,  $F = \{1\}$



un mot  $a_1 a_2 \dots a_n \in A^*$  est **reconnu** par un automate  $(Q, T, I, F)$  ssi

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \cdots s_{n-1} \xrightarrow{a_n} s_n$$

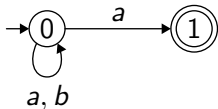
avec  $s_0 \in I$ ,  $(s_{i-1}, a_i, s_i) \in T$  pour tout  $i$ , et  $s_n \in F$

le **langage** défini par un automate est l'ensemble des mots reconnus

## Théorème (de Kleene)

*Les expressions régulières et les automates finis définissent les mêmes langages.*

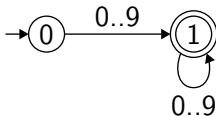
$(a|b)^* a$



expression régulière

$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$

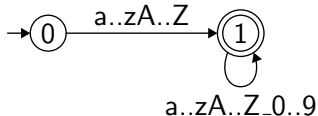
automate



expression régulière

$$(a|b|\dots|z|A|B|\dots|Z)(a|b|\dots|z|A|B|\dots|Z|_|0|1|\dots|9)^*$$

automate

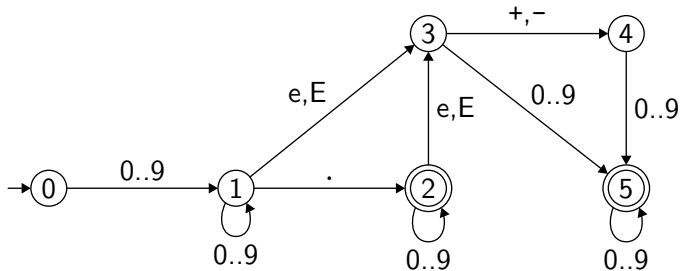


expression régulière

$$d d^* (.d^* | (\epsilon | .d^*)(e|E) (\epsilon | + | -) d d^*)$$

où  $d = 0|1|\dots|9$

automate





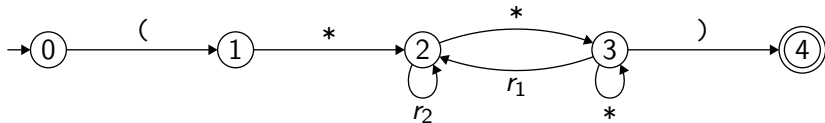
expression régulière

$$\boxed{(} \boxed{*} \boxed{(} \boxed{*} \boxed{*} r_1 \mid r_2 \boxed{)} \boxed{*} \boxed{*} \boxed{*} \boxed{)}$$

où  $r_1$  = tous les caractères sauf \* et )

et  $r_2$  = tous les caractères sauf \*

automate fini



---

## analyseur lexical

un **analyseur lexical** est un automate fini pour la « réunion » de toutes les expressions régulières définissant les lexèmes

le fonctionnement de l'analyseur lexical, cependant, est différent de la simple reconnaissance d'un mot par un automate, car

- il faut décomposer un mot (le source) en une **suite** de mots reconnus
- il peut y avoir des **ambiguïtés**
- il faut construire les lexèmes (les états finaux contiennent des **actions**)

le mot `funx` est reconnu par l'expression régulière des identificateurs, mais contient un préfixe reconnu par une autre expression régulière (`fun`)

⇒ on fait le choix de reconnaître le lexème le plus long possible

le mot `fun` est reconnu par l'expression régulière du mot clé `fun` mais aussi par celle des identificateurs

⇒ on classe les lexèmes par ordre de priorité

avec les trois expressions régulières

$a, ab, bc$

un analyseur lexical va **échouer** sur l'entrée

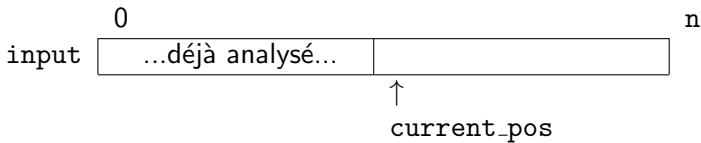
$abc$

( $ab$  est reconnu, comme plus long, puis échec sur  $c$ )

pourtant le mot  $abc$  appartient au langage  $a|ab|bc$

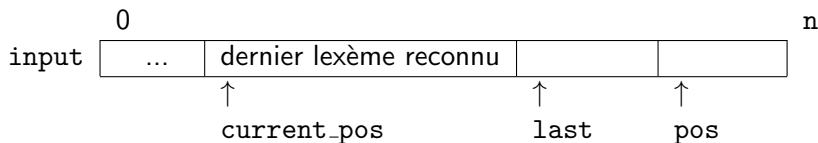
les lexèmes sont produits un par un, à la demande (de l'analyseur syntaxique)

l'analyseur lexical mémorise donc la position où l'analyse de l'entrée devra reprendre



lorsqu'un nouveau lexème est demandé, on démarre dans l'état initial de l'automate, à partir de `current_pos`

tant qu'une transition est possible, on l'emprunte, tout en mémorisant le dernière lexème reconnu (dernier état final rencontré)

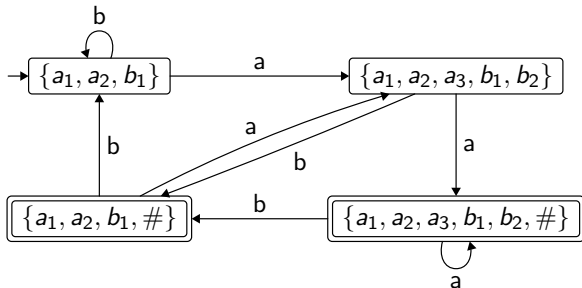


lorsqu'une transition n'est plus possible, de deux choses l'une :

- si un lexème a été reconnu, on le renvoie et `current_pos` prend la valeur de `last`
- sinon, c'est un échec de l'analyse lexicale

on peut construire l'automate fini correspondant à une expression régulière en passant par l'intermédiaire d'un automate non déterministe (Thompson, 1968)

mais on peut aussi construire directement un automate déterministe (Berry, Sethi, 1986) ; pour  $(a|b) \star a(a|b)$  on obtient



voir le poly page 48



---

## outils

en pratique, on dispose d'outils qui construisent les analyseurs lexicaux à partir de leur description par des expressions régulières et des actions

c'est la grande famille de `lex` : `lex`, `flex`, `jflex`, `ocamllex`, etc.

on présente ici `jflex` (pour Java) et `ocamllex` (pour OCaml)

pour illustrer ces outils, écrivons un analyseur lexical minimal pour un langage d'expressions arithmétiques avec

- des constantes entières
- des parenthèses
- une soustraction

---

## **l'outil jflex**

un fichier jflex porte le suffixe .flex et a la forme suivante

```
... préambule ...
%{
... code Java arbitraire
}%
%%
<YYINITIAL> {
    expression régulière { action }
    ...
    expression régulière { action }
}
```

où chaque action est un code Java  
(qui le plus souvent renvoie un lexème)

on écrit un fichier `Lexer.flex` pour nos expressions arithmétiques

```
import static sym.*;    /* importe les lexèmes */

%%

%class Lexer           /* notre classe s'appellera Lexer */
%unicode               /* les caractères sont unicode */
%cup                  /* analyse syntaxique avec cup */
%line                 /* activer le décompte des lignes */
%column              /* et celui des colonnes */
%yylexthrow Exception /* on peut lever Exception */

%{
    /* pas besoin de préambule Java ici */
%}
```

...

...

```

WhiteSpace      = [ \t\r\n]+      /* raccourcis */
Integer         = [:digit:]+
%%
<YYINITIAL> {
    "-"         { return new Symbol(MINUS, yyline, yycolumn); }
    "("         { return new Symbol(LPAR, yyline, yycolumn); }
    ")"         { return new Symbol(RPAR, yyline, yycolumn); }
    {Integer}
                { return new Symbol(INT, yyline, yycolumn,
                                     Integer.parseInt(yytext())); }
    {WhiteSpace}
                { /* ignore */ }
    .           { throw new Exception (String.format (
                "Line %d, column %d: illegal character: '%s'\n",
                yyline, yycolumn, yytext())); }
}

```

- la nature des lexèmes est libre ;  
on utilise ici la classe `Symbol` qui vient avec `cup` (voir plus loin)
- `MINUS`, `LPAR`, `RPAR` et `INT` sont des entiers (la nature des lexèmes) ici produits par l'outil `cup` et importés depuis `sym.java`
- les variables `yyline` et `yycolumn` sont mises à jour automatiquement
- `yytext()` renvoie la chaîne reconnue par l'expression régulière



on compile le fichier `Lexer.flex` avec `jflex`

```
jflex Lexer.flex
```

on obtient un fichier `Lexer.java` contenant notamment

- un constructeur

```
Lexer(java.io.Reader)
```

- une méthode

```
Symbol next_token()
```

<code>.</code>	n'importe quel caractère
<code>a</code>	le caractère 'a'
<code>"foobar"</code>	la chaîne "foobar" (en particulier $\epsilon = ""$ )
<code>[caractères]</code>	ensemble de caractères (par ex. [a-zA-Z])
<code>[^caractères]</code>	complémentaire (par ex. [^"])
<code>[:ident:]</code>	ensemble prédéfini de caractères (par ex. [:digit:])
<code>{ident}</code>	expression régulière définie plus haut
<code><math>r_1 \mid r_2</math></code>	l'alternative
<code><math>r_1 r_2</math></code>	la concaténation
<code><math>r^*</math></code>	l'étoile
<code><math>r^+</math></code>	une ou plusieurs répétitions de $r$ ( $\stackrel{\text{def}}{=} r r^*$ )
<code><math>r^?</math></code>	une ou zéro occurrence de $r$ ( $\stackrel{\text{def}}{=} \epsilon \mid r$ )
<code>( r )</code>	parenthésage

---

## **l'outil ocamllex**

un fichier ocamllex porte le suffixe `.mll` et a la forme suivante

```
{
  ... code OCaml arbitraire ...
}
rule nom = parse
| expression régulière { action }
| expression régulière { action }
| ...
{
  ... code OCaml arbitraire ...
}
```

où chaque action est un code OCaml

```
let white_space = [' ' '\t' '\n']+
let integer     = ['0'-'9']+
rule next_token = parse
  | white_space
    { next_token lexbuf }
  | integer as s
    { INT (int_of_string s) }
  | '-'
    { MINUS }
  | '('
    { LPAR }
  | ')'
    { RPAR }
  | eof
    { EOF }
  | _ as c
    { failwith ("illegal character" ^ String.make 1 c) }
```

- on suppose ici le type suivant pour les lexèmes

```
type token =  
| INT of int  
| MINUS  
| LPAR  
| RPAR  
| EOF
```

il sera construit par l'analyseur syntaxique

- contrairement à `jflex`
  - on rappelle explicitement `next_token` quand on ignore les blancs
  - on ne manipule pas explicitement les lignes et colonnes

on compile le fichier `lexer.mll` avec `ocamllex`

```
ocamllex lexer.mll
```

ce qui produit un fichier OCaml `lexer.ml` qui définit une fonction

```
val next_token: Lexing.lexbuf -> token
```

(on construit son argument avec la fonction `Lexing.from_channel`)

# les expressions régulières d'ocamllex

<code>-</code>	n'importe quel caractère
<code>'a'</code>	le caractère 'a'
<code>"foobar"</code>	la chaîne "foobar" (en particulier $\epsilon = ""$ )
<code>[caractères]</code>	ensemble de caractères (par ex. <code>['a'-'z' 'A'-'Z']</code> )
<code>[^caractères]</code>	complémentaire (par ex. <code>[^ '"]</code> )
<i>ident</i>	expression régulière définie plus haut
<code>r<sub>1</sub>   r<sub>2</sub></code>	l'alternative
<code>r<sub>1</sub> r<sub>2</sub></code>	la concaténation
<code>r *</code>	l'étoile
<code>r +</code>	une ou plusieurs répétitions de $r$ ( $\stackrel{\text{def}}{=} r r^*$ )
<code>r ?</code>	une ou zéro occurrence de $r$ ( $\stackrel{\text{def}}{=} \epsilon   r$ )
<code>( r )</code>	parenthésage
<code>eof</code>	la fin de l'entrée



pour plus de détails, voir les documentations de `jflex` et `ocamllex`, accessibles depuis

- la page du cours
- le TD 3

- les **expressions régulières** sont à la base de l'analyse lexicale
- le travail est grandement **automatisé** par des outils tels que `jflex` ou `ocamllex`
- `jflex/ocamllex` est **plus expressif** que les expressions régulières car on peut écrire du code arbitraire dans les actions et rappeler l'analyseur lexical récursivement sur une condition  
⇒ permet notamment de reconnaître des commentaires **imbriqués**

---

## analyse syntaxique

suite de lexèmes

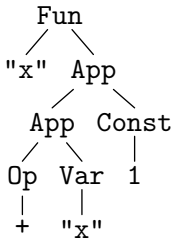
fun	x	->	(	x	+	1	)
-----	---	----	---	---	---	---	---



analyse syntaxique



syntaxe abstraite



en particulier, l'analyse syntaxique doit détecter les erreurs de syntaxe et

- les localiser précisément
- les identifier (le plus souvent seulement « erreur de syntaxe » mais aussi « parenthèse non fermée », etc.)
- voire, reprendre l'analyse pour découvrir de nouvelles erreurs

pour l'analyse syntaxique, on va utiliser

- une **grammaire non contextuelle** pour décrire la syntaxe
- un **automate à pile** pour la reconnaître

c'est l'analogie des expressions régulières / automates finis utilisés dans l'analyse lexicale

## Définition

*Une grammaire non contextuelle (ou hors contexte) est un quadruplet  $(N, T, S, R)$  où*

- *$N$  est un ensemble fini de **symboles non terminaux***
- *$T$  est un ensemble fini de **symboles terminaux***
- *$S \in N$  est le symbole de départ (dit **axiome**)*
- *$R \subseteq N \times (N \cup T)^*$  est un ensemble fini de **règles de production***

## exemple : expressions arithmétiques

$N = \{E\}$ ,  $T = \{+, *, (, ), \text{int}\}$ ,  $S = E$ ,  
et  $R = \{(E, E+E), (E, E*E), (E, (E)), (E, \text{int})\}$

en pratique on note les règles sous la forme

$$\begin{array}{l} E \rightarrow E + E \\ \quad | \quad E * E \\ \quad | \quad ( E ) \\ \quad | \quad \text{int} \end{array}$$

les terminaux de la grammaire seront les lexèmes produits par l'analyse lexicale

int désigne ici le lexème correspondant à une constante entière  
(*i.e.* sa nature, pas sa valeur)



## Définition

Un mot  $u \in (N \cup T)^*$  se **dérive** en un mot  $v \in (N \cup T)^*$ , et on note  $u \rightarrow v$ , s'il existe une décomposition

$$u = u_1 X u_2$$

avec  $X \in N$ ,  $X \rightarrow \beta \in R$  et

$$v = u_1 \beta u_2$$

exemple :

$$\underbrace{E *}_{u_1} \left( \underbrace{E}_X \right) \underbrace{)}_{u_2} \rightarrow E * \left( \underbrace{E + E}_{\beta} \right)$$

une suite  $w_1 \rightarrow w_2 \rightarrow \dots \rightarrow w_n$  est appelée une dérivation

on parle de **dérivation gauche** (resp. **droite**) si le non terminal réduit est systématiquement le plus à gauche *i.e.*  $u_1 \in T^*$  (resp. le plus à droite *i.e.*  $u_2 \in T^*$ )

on note  $\rightarrow^*$  la clôture réflexive transitive de  $\rightarrow$

$E \rightarrow E * E$   
 $\rightarrow \text{int} * E$   
 $\rightarrow \text{int} * ( E )$   
 $\rightarrow \text{int} * ( E + E )$   
 $\rightarrow \text{int} * ( \text{int} + E )$   
 $\rightarrow \text{int} * ( \text{int} + \text{int} )$

## Définition

Le **langage** défini par une grammaire non contextuelle  $G = (N, T, S, R)$  est l'ensemble des mots de  $T^*$  dérivés de l'axiome, i.e.

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

dans notre exemple

$$\text{int} * (\text{int} + \text{int}) \in L(G)$$

## Définition

À toute dérivation  $S \rightarrow^* w$ , on peut associer un **arbre de dérivation**, dont les nœuds sont étiquetés ainsi

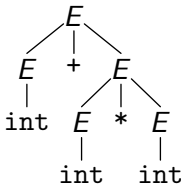
- la racine est  $S$
- les feuilles forment le mot  $w$  dans l'ordre infixe
- tout nœud interne  $X$  est un non terminal dont les fils sont étiquetés par  $\beta \in (N \cup T)^*$  avec  $X \rightarrow \beta$  une règle de la dérivation

attention : ce n'est pas la même chose que l'arbre de syntaxe abstraite

la dérivation gauche

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

donne l'arbre de dérivation



mais la dérivation droite

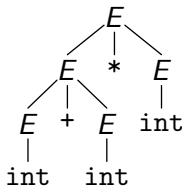
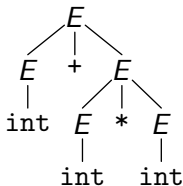
$$E \rightarrow E + E \rightarrow E + E * E \rightarrow E + E * \text{int} \rightarrow E + \text{int} * \text{int} \rightarrow \text{int} + \text{int} * \text{int}$$

également

## Définition

Une grammaire est dite **ambiguë** si un mot au moins admet plusieurs arbres de dérivation

exemple : le mot `int + int * int` admet les deux arbres de dérivations



et notre grammaire est donc ambiguë

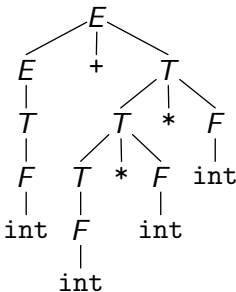
pour ce langage-là, il est néanmoins possible de proposer une autre grammaire, non ambiguë, qui définit le même langage

$$\begin{array}{l} E \rightarrow E + T \\ \quad | T \\ T \rightarrow T * F \\ \quad | F \\ F \rightarrow ( E ) \\ \quad | \text{int} \end{array}$$

cette nouvelle grammaire traduit la priorité de la multiplication sur l'addition, et le choix d'une associativité à gauche pour ces deux opérations



ainsi, le mot `int + int * int * int` n'a plus qu'un seul arbre de dérivation, à savoir



correspondant à la dérivation gauche

$$\begin{aligned}
 E &\rightarrow E + T \rightarrow T + T \rightarrow F + T \rightarrow \text{int} + T \rightarrow \text{int} + T * F \\
 &\rightarrow \text{int} + T * F * F \rightarrow \text{int} + F * F * F \rightarrow \text{int} + \text{int} * F * F \\
 &\rightarrow \text{int} + \text{int} * \text{int} * F \rightarrow \text{int} + \text{int} * \text{int} * \text{int}
 \end{aligned}$$

déterminer si une grammaire est ou non ambiguë n'est **pas décidable**

(rappel : décidable veut dire qu'on peut écrire un programme qui, pour toute entrée, termine et répond oui ou non)

on va utiliser des **critères décidables suffisants** pour garantir qu'une grammaire est non ambiguë, et pour lesquels on sait en outre décider l'appartenance au langage efficacement (avec un automate à pile déterministe)

les classes de grammaires définies par ces critères s'appellent LR(0), SLR(1), LALR(1), LR(1), LL(1), etc.

avant de commencer, on a besoin de quelques définitions...

### Définition (NULL)

Soit  $\alpha \in (T \cup N)^*$ .  $\text{NULL}(\alpha)$  est vrai si et seulement si on peut dériver  $\epsilon$  à partir de  $\alpha$  i.e.  $\alpha \rightarrow^* \epsilon$ .

### Définition (FIRST)

Soit  $\alpha \in (T \cup N)^*$ .  $\text{FIRST}(\alpha)$  est l'ensemble de tous les premiers terminaux des mots dérivés de  $\alpha$ , i.e.  $\{a \in T \mid \exists w. \alpha \rightarrow^* aw\}$ .

### Définition (FOLLOW)

Soit  $X \in N$ .  $\text{FOLLOW}(X)$  est l'ensemble de tous les terminaux qui peuvent apparaître après  $X$  dans une dérivation, i.e.  $\{a \in T \mid \exists u, w. S \rightarrow^* uXaw\}$ .

pour calculer  $\text{NULL}(\alpha)$  il suffit de déterminer  $\text{NULL}(X)$  pour  $X \in N$

$\text{NULL}(X)$  est vrai si et seulement si

- il existe une production  $X \rightarrow \epsilon$ ,
- ou il existe une production  $X \rightarrow Y_1 \dots Y_m$  où  $\text{NULL}(Y_i)$  pour tout  $i$

problème : il s'agit d'un ensemble d'équations mutuellement récursives

dit autrement, si  $N = \{X_1, \dots, X_n\}$  et si  $\vec{V} = (\text{NULL}(X_1), \dots, \text{NULL}(X_n))$ , on cherche la **plus petite solution** d'une équation de la forme

$$\vec{V} = F(\vec{V})$$

## deux exemples de telles équations



## Théorème (existence d'un plus petit point fixe (Tarski))

Soit  $A$  un ensemble fini muni d'une relation d'ordre  $\leq$  et d'un plus petit élément  $\varepsilon$ . Toute fonction  $f : A \rightarrow A$  croissante, i.e. telle que  $\forall x, y. x \leq y \Rightarrow f(x) \leq f(y)$ , admet un plus petit point fixe.

preuve : comme  $\varepsilon$  est le plus petit élément, on a  $\varepsilon \leq f(\varepsilon)$   
 $f$  étant croissante, on a donc  $f^k(\varepsilon) \leq f^{k+1}(\varepsilon)$  pour tout  $k$   
 $A$  étant fini, il existe donc un plus petit  $k_0$  tel que  $f^{k_0}(\varepsilon) = f^{k_0+1}(\varepsilon)$   
 $a_0 = f^{k_0}(\varepsilon)$  est donc un point fixe de  $f$

soit  $b$  un autre point fixe de  $f$   
on a  $\varepsilon \leq b$  et donc  $f^k(\varepsilon) \leq f^k(b)$  pour tout  $k$   
en particulier  $a_0 = f^{k_0}(\varepsilon) \leq f^{k_0}(b) = b$   
 $a_0$  est donc le plus petit point fixe de  $f$

□

le théorème de Tarski donne des conditions suffisantes mais pas nécessaires



dans le cas du calcul de NULL, on a

$A = \text{BOOL} \times \cdots \times \text{BOOL}$  avec  $\text{BOOL} = \{\text{false}, \text{true}\}$

on peut munir  $\text{BOOL}$  de l'ordre  $\text{false} \leq \text{true}$  et  $A$  de l'ordre point à point

$$(x_1, \dots, x_n) \leq (y_1, \dots, y_n) \quad \text{si et seulement si} \quad \forall i. x_i \leq y_i$$

le théorème s'applique alors en prenant

$$\varepsilon = (\text{false}, \dots, \text{false})$$

car la fonction calculant  $\text{NULL}(X)$  à partir des  $\text{NULL}(X_i)$  est croissante

pour calculer les  $\text{NULL}(X_i)$ , on part donc de

$$\text{NULL}(X_1) = \text{false}, \dots, \text{NULL}(X_n) = \text{false}$$

et on applique les équations jusqu'à obtention du point fixe *i.e.* jusqu'à ce que la valeur des  $\text{NULL}(X_i)$  ne soit plus modifiée

$$\begin{array}{l}
 E \rightarrow T E' \\
 E' \rightarrow + T E' \\
 \quad | \epsilon \\
 T \rightarrow F T' \\
 T' \rightarrow * F T' \\
 \quad | \epsilon \\
 F \rightarrow ( E ) \\
 \quad | \text{int}
 \end{array}$$

$E$	$E'$	$T$	$T'$	$F$
false	false	false	false	false
false	true	false	true	false
false	true	false	true	false

pourquoi cherche-t-on le **plus petit** point fixe ?

- ⇒ par récurrence sur le nombre d'étapes du calcul précédent, on montre que si  $\text{NULL}(X) = \text{true}$  alors  $X \rightarrow^* \epsilon$
- ⇐ par récurrence sur le nombre d'étapes de la dérivation  $X \rightarrow^* \epsilon$ , on montre que  $\text{NULL}(X) = \text{true}$  par le calcul précédent

de même, les équations définissant FIRST sont mutuellement récursives

$$\text{FIRST}(X) = \bigcup_{X \rightarrow \beta} \text{FIRST}(\beta)$$

et

$$\text{FIRST}(\epsilon) = \emptyset$$

$$\text{FIRST}(a\beta) = \{a\}$$

$$\text{FIRST}(X\beta) = \text{FIRST}(X), \quad \text{si } \neg \text{NULL}(X)$$

$$\text{FIRST}(X\beta) = \text{FIRST}(X) \cup \text{FIRST}(\beta), \quad \text{si } \text{NULL}(X)$$

de même, on procède par calcul de point fixe sur le produit cartésien

$A = \mathcal{P}(T) \times \cdots \times \mathcal{P}(T)$  muni, point à point, de l'ordre  $\subseteq$  et avec

$\epsilon = (\emptyset, \dots, \emptyset)$

$$\begin{aligned}
 E &\rightarrow T E' \\
 E' &\rightarrow + T E' \\
 &\quad | \epsilon \\
 T &\rightarrow F T' \\
 T' &\rightarrow * F T' \\
 &\quad | \epsilon \\
 F &\rightarrow ( E ) \\
 &\quad | \text{int}
 \end{aligned}$$

NULL

$E$	$E'$	$T$	$T'$	$F$
false	true	false	true	false

FIRST

$E$	$E'$	$T$	$T'$	$F$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\emptyset$	$\{+\}$	$\emptyset$	$\{*\}$	$\{(, \text{int})\}$
$\emptyset$	$\{+\}$	$\{(, \text{int})\}$	$\{*\}$	$\{(, \text{int})\}$
$\{(, \text{int})\}$	$\{+\}$	$\{(, \text{int})\}$	$\{*\}$	$\{(, \text{int})\}$
$\{(, \text{int})\}$	$\{+\}$	$\{(, \text{int})\}$	$\{*\}$	$\{(, \text{int})\}$

là encore, les équations définissant FOLLOW sont mutuellement récursives

$$\text{FOLLOW}(X) = \bigcup_{Y \rightarrow \alpha X \beta} \text{FIRST}(\beta) \cup \bigcup_{Y \rightarrow \alpha X \beta, \text{NULL}(\beta)} \text{FOLLOW}(Y)$$

on procède par calcul de point fixe, sur le même domaine que pour FIRST

on introduit un symbole spécial # dans les suivants du symbole de départ (ce que l'on peut faire directement, ou en ajoutant une règle  $S' \rightarrow S\#$ )

NULL

$E$	$E'$	$T$	$T'$	$F$
false	true	false	true	false

$E \rightarrow T E'$   
 $E' \rightarrow + T E'$   
 $\quad \mid \epsilon$   
 $T \rightarrow F T'$   
 $T' \rightarrow * F T'$   
 $\quad \mid \epsilon$   
 $F \rightarrow ( E )$   
 $\quad \mid \text{int}$

FIRST

$E$	$E'$	$T$	$T'$	$F$
{(, int}	{+}	{(, int}	{*}	{(, int}

FOLLOW

$E$	$E'$	$T$	$T'$	$F$
{#}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
{#, )}	{#}	{+, #}	$\emptyset$	{*}
{#, )}	{#, )}	{+, #, )}	{+, #}	{*, +, #}
{#, )}	{#, )}	{+, #, )}	{+, #, )}	{*, +, #, )}
{#, )}	{#, )}	{+, #, )}	{+, #, )}	{*, +, #, )}



---

## analyse ascendante

l'idée consiste à lire l'entrée de gauche à droite, en cherchant à reconnaître des membres droits de productions pour construire l'arbre de dérivation de bas en haut (*bottom-up parsing*)

l'analyse manipule une pile qui est un mot de  $(T \cup N)^*$

à chaque instant, deux actions sont possibles

- opération de **lecture** (*shift* en anglais) : on lit un terminal de l'entrée et on l'empile
- opération de **réduction** (*reduce* en anglais) : on reconnaît en sommet de pile le membre droit  $\beta$  d'une production  $X \rightarrow \beta$ , et on remplace  $\beta$  par  $X$  en sommet de pile

dans l'état initial, la pile est vide

lorsqu'il n'y a plus d'action possible, l'entrée est reconnue si elle a été entièrement lue et si la pile est réduite à  $S$

	pile	entrée	action
	$\epsilon$	int+int*int	lecture
	int	+int*int	réduction $F \rightarrow \text{int}$
	$F$	+int*int	réduction $T \rightarrow F$
	$T$	+int*int	réduction $E \rightarrow T$
$E \rightarrow E + T$	$E$	+int*int	lecture
$T$	$E+$	int*int	lecture
$T \rightarrow T * F$	$E+\text{int}$	*int	réduction $F \rightarrow \text{int}$
$F$	$E+F$	*int	réduction $T \rightarrow F$
$F \rightarrow ( E )$	$E+T$	*int	lecture
int	$E+T*$	int	lecture
	$E+T*\text{int}$		réduction $F \rightarrow \text{int}$
	$E+T*F$		réduction $T \rightarrow T*F$
	$E+T$		réduction $E \rightarrow E+T$
	$E$		succès

comment prendre la décision lecture / réduction ?

en se servant d'un automate fini et en examinant les  $k$  premiers caractères de l'entrée ; c'est l'analyse LR( $k$ )

(LR signifie « **L**eft to right scanning, **R**ightmost derivation »)

en pratique  $k = 1$

*i.e.* on examine uniquement le premier caractère de l'entrée

la pile est de la forme

$$s_0 x_1 s_1 \dots x_n s_n$$

où  $s_i$  est un état de l'automate et  $x_i \in T \cup N$  comme auparavant

soit  $a$  le premier caractère de l'entrée ; on regarde la transition de l'automate pour l'état  $s_n$  et l'entrée  $a$

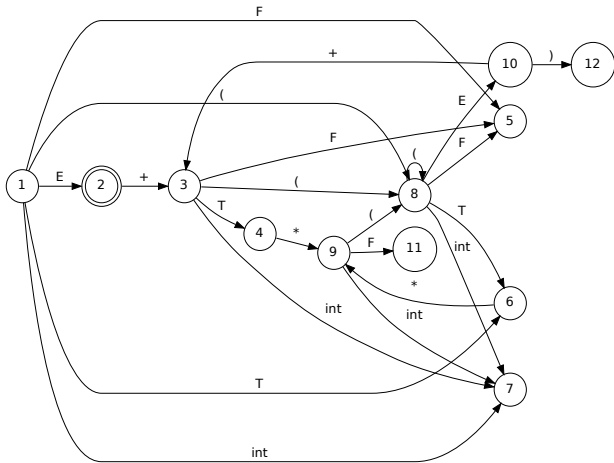
- si c'est un succès ou un échec, on s'arrête
- si c'est une lecture, alors on empile  $a$  et l'état résultat de la transition
- si c'est une réduction  $X \rightarrow \alpha$ , avec  $\alpha$  de longueur  $p$ , alors on doit trouver  $\alpha$  en sommet de pile

$$s_0 x_1 s_1 \dots x_{n-p} s_{n-p} \mid \alpha_1 s_{n-p+1} \dots \alpha_p s_n$$

on dépile alors  $\alpha$  et on empile  $Xs$ , où  $s$  est l'état résultat de la transition  $s_{n-p} \xrightarrow{X} s$ , i.e.

$$s_0 x_1 s_1 \dots x_{n-p} s_{n-p} X s$$

dans l'exemple plus haut, on s'est servi de cet automate

$$\begin{array}{l}
 E \rightarrow E + T \\
 \quad | \quad T \\
 T \rightarrow T * F \\
 \quad | \quad F \\
 F \rightarrow ( E ) \\
 \quad | \quad \text{int}
 \end{array}$$


---

## construction de l'automate



fixons pour l'instant  $k = 0$

on commence par construire un automate **asynchrone**

c'est-à-dire contenant des transitions spontanées  
appelées  **$\epsilon$ -transitions** et notées  $s_1 \xrightarrow{\epsilon} s_2$

les **états** sont des *items* de la forme

$$[X \rightarrow \alpha \bullet \beta]$$

où  $X \rightarrow \alpha\beta$  est une production de la grammaire ; l'intuition est « je cherche à reconnaître  $X$ , j'ai déjà lu  $\alpha$  et je dois encore lire  $\beta$  »

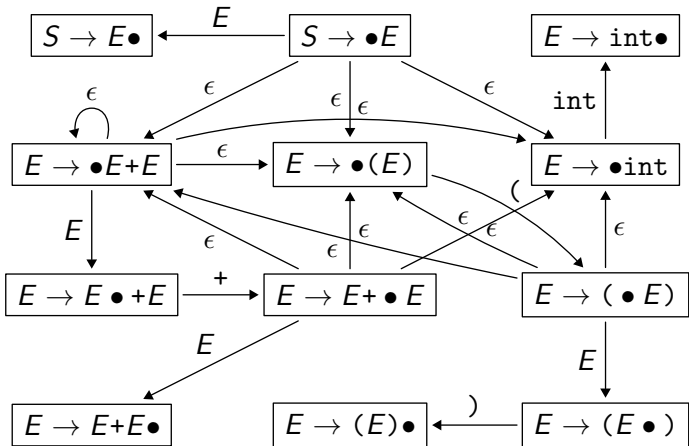
les **transitions** sont étiquetées par  $T \cup N$  et sont les suivantes

$$[Y \rightarrow \alpha \bullet a\beta] \xrightarrow{a} [Y \rightarrow \alpha a \bullet \beta]$$

$$[Y \rightarrow \alpha \bullet X\beta] \xrightarrow{X} [Y \rightarrow \alpha X \bullet \beta]$$

$$[Y \rightarrow \alpha \bullet X\beta] \xrightarrow{\epsilon} [X \rightarrow \bullet \gamma] \quad \text{pour toute production } X \rightarrow \gamma$$

$S \rightarrow E$   
 $E \rightarrow E+E$   
 $E \rightarrow (E)$   
 $E \rightarrow \text{int}$



**déterminisons** l'automate LR(0)

pour cela, on regroupe les états reliés par des  $\epsilon$ -transitions

les états de l'automate déterministe sont donc des ensembles d'*items*, tel que

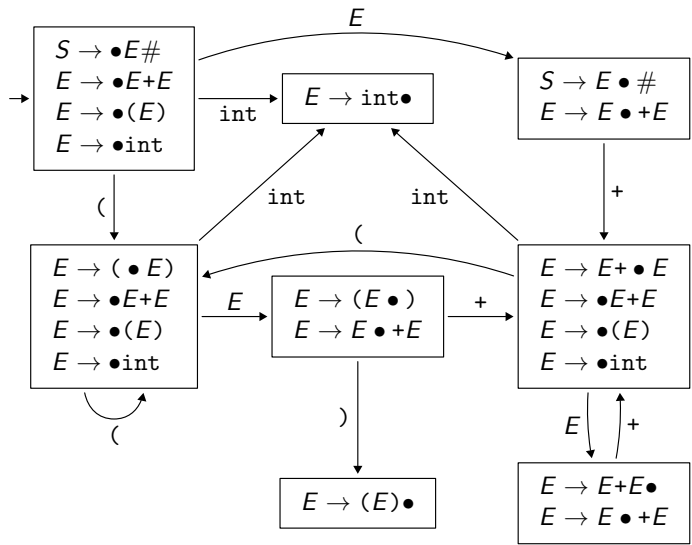
$$\begin{array}{l} E \rightarrow E+ \bullet E \\ E \rightarrow \bullet E+E \\ E \rightarrow \bullet (E) \\ E \rightarrow \bullet \text{int} \end{array}$$

chaque état  $s$  est **saturé** par la propriété

si  $Y \rightarrow \alpha \bullet X \beta \in s$   
 et si  $X \rightarrow \gamma$  est une production  
 alors  $X \rightarrow \bullet \gamma \in s$

l'état initial est celui contenant  $S \rightarrow \bullet E$

$S \rightarrow E$   
 $E \rightarrow E+E$   
 $E \rightarrow (E)$   
 $E \rightarrow int$



en pratique, on ne travaille pas directement sur l'automate mais sur deux tables

- une table d'**actions** ayant pour lignes les états et pour colonnes les terminaux; la case  $\text{action}(s, a)$  indique
  - shift  $s'$  pour une lecture et un nouvel état  $s'$
  - reduce  $X \rightarrow \alpha$  pour une réduction
  - un succès
  - un échec
- une table de **déplacements** ayant pour lignes les états et pour colonnes les non terminaux; la case  $\text{goto}(s, X)$  indique l'état résultat d'une réduction de  $X$

on construit ainsi la table action

- $\text{action}(s, \#) = \text{succès}$  si  $[S \rightarrow E \bullet \#] \in s$
- $\text{action}(s, a) = \text{shift } s'$  si on a une transition  $s \xrightarrow{a} s'$
- $\text{action}(s, a) = \text{reduce } X \rightarrow \beta$  si  $[X \rightarrow \beta \bullet] \in s$ , pour tout  $a$
- échec dans tous les autres cas

on construit ainsi la table goto

- $\text{goto}(s, X) = s'$  si et seulement si on a une transition  $s \xrightarrow{X} s'$



sur notre exemple, la table est la suivante :

	<i>action</i>					<i>goto</i>
état	(	)	+	int	#	<i>E</i>
1	shift 4			shift 2		3
2	reduce $E \rightarrow \text{int}$					
3			shift 6		succès	
4	shift 4			shift 2		5
5		shift 7	shift 6			
6	shift 4			shift 2		8
7	reduce $E \rightarrow (E)$					
8			shift 6			
	reduce $E \rightarrow E+E$					

la table LR(0) peut contenir deux sortes de conflits

- un conflit **lecture/réduction** (*shift/reduce*), si dans un état  $s$  on peut effectuer une lecture mais aussi une réduction
- un conflit **réduction/réduction** (*reduce/reduce*), si dans un état  $s$  deux réductions différentes sont possibles

### Définition (classe LR(0))

*Une grammaire est dite LR(0) si la table ainsi construite ne contient pas de conflit.*

on a un conflit lecture/réduction dans l'état 8

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

il illustre précisément l'ambiguïté de la grammaire sur un mot tel que `int+int+int`

on peut résoudre le conflit de deux façons

- si on favorise la **lecture**, on traduira une associativité à droite
- si on favorise la **réduction**, on traduira une associativité à gauche

privilégions la réduction et illustrons sur un exemple

	(	)	+	int	#	$E$
1	s4			s2		3
2	reduce $E \rightarrow \text{int}$					
3			s6		ok	
4	s4			s2		5
5		s7	s6			
6	s4			s2		8
7	reduce $E \rightarrow (E)$					
8	reduce $E \rightarrow E+E$					

pile	entrée	action
1	int+int+int	s2
1 int 2	+int+int	$E \rightarrow \text{int}, g3$
1 $E$ 3	+int+int	s6
1 $E$ 3 + 6	int+int	s2
1 $E$ 3 + 6 int 2	+int	$E \rightarrow \text{int}, g8$
1 $E$ 3 + 6 $E$ 8	+int	$E \rightarrow E+E, g3$
1 $E$ 3	+int	s6
1 $E$ 3 + 6	int	s2
1 $E$ 3 + 6 int 2	#	$E \rightarrow \text{int}, g8$
1 $E$ 3 + 6 $E$ 8	#	$E \rightarrow E+E, g3$
1 $E$ 3	#	succès

la construction LR(0) engendre très facilement des conflits  
on va donc chercher à limiter les réductions

une idée très simple consiste à poser  $\text{action}(s, a) = \text{reduce } X \rightarrow \beta$  si et seulement si

$$[X \rightarrow \beta \bullet] \in s \quad \text{et} \quad a \in \text{FOLLOW}(X)$$

### Définition (classe SLR(1))

*Une grammaire est dite SLR(1) si la table ainsi construite ne contient pas de conflit.*

(SLR signifie *Simple LR*)

la grammaire

$$\begin{aligned} S &\rightarrow E\# \\ E &\rightarrow E + T \\ &\quad | T \\ T &\rightarrow T * F \\ &\quad | F \\ F &\rightarrow ( E ) \\ &\quad | \text{int} \end{aligned}$$

est SLR(1)

exercice : le vérifier (l'automate contient 12 états)

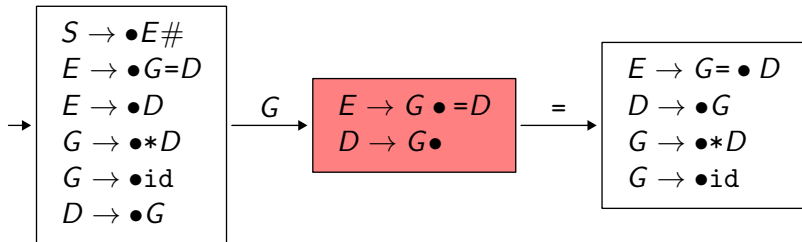
# limites de l'analyse SLR(1)

en pratique, la classe SLR(1) reste trop restrictive

exemple :

$$\begin{aligned} S &\rightarrow E\# \\ E &\rightarrow G = D \\ &\quad | D \\ G &\rightarrow *D \\ &\quad | \text{id} \\ D &\rightarrow G \end{aligned}$$

	=	
1	...	...
2	shift 3 reduce $D \rightarrow G$	...
3	⋮	⋱



on introduit une classe de grammaires encore plus large, **LR(1)**, au prix de tables encore plus grandes

dans l'analyse LR(1), les *items* ont maintenant la forme

$$[X \rightarrow \alpha \bullet \beta, a]$$

dont la signification est : « je cherche à reconnaître  $X$ , j'ai déjà lu  $\alpha$  et je dois encore lire  $\beta$  puis vérifier que le caractère suivant est  $a$  »



les transitions de l'automate LR(1) non déterministe sont

$$\begin{aligned}
 [Y \rightarrow \alpha \bullet a\beta, b] &\xrightarrow{a} [Y \rightarrow \alpha a \bullet \beta, b] \\
 [Y \rightarrow \alpha \bullet X\beta, b] &\xrightarrow{X} [Y \rightarrow \alpha X \bullet \beta, b] \\
 [Y \rightarrow \alpha \bullet X\beta, b] &\xrightarrow{\epsilon} [X \rightarrow \bullet \gamma, c] \quad \text{pour tout } c \in \text{FIRST}(\beta b)
 \end{aligned}$$

l'état initial est celui qui contient  $[S \rightarrow \bullet \alpha, \#]$

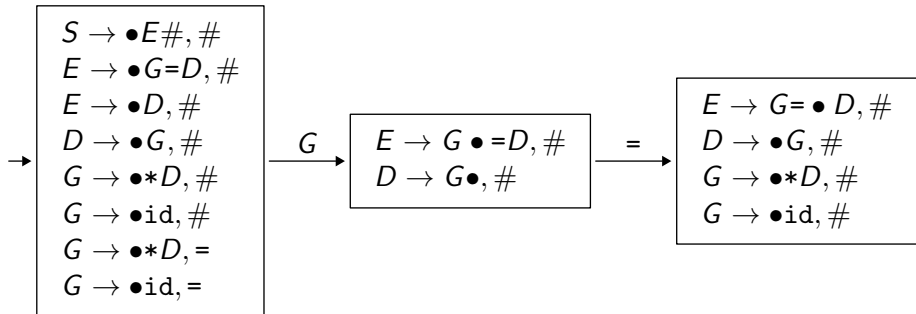
comme précédemment, on peut déterminer l'automate et construire la table correspondante ; on introduit une action de réduction pour  $(s, a)$  seulement lorsque  $s$  contient un item de la forme  $[X \rightarrow \alpha \bullet, a]$

### Définition (classe LR(1))

*Une grammaire est dite LR(1) si la table ainsi construite ne contient pas de conflit.*

$S \rightarrow E\#$   
 $E \rightarrow G = D$   
 $\quad | D$   
 $G \rightarrow * D$   
 $\quad | id$   
 $D \rightarrow G$

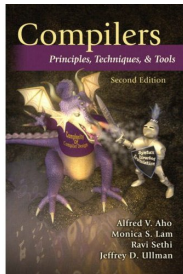
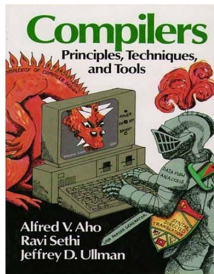
	#	=	
1	...	...	...
2	reduce $D \rightarrow G$	shift 3	...
3	:	:	..



la construction LR(1) pouvant être coûteuse, il existe des approximations

la classe LALR(1) (*lookahead LR*) est une telle approximation, utilisée notamment dans les outils de la famille yacc

plus d'info : voir par exemple *Compilateurs : principes techniques et outils* (dit « le dragon ») de A. Aho, R. Sethi, J. Ullman, section 4.7

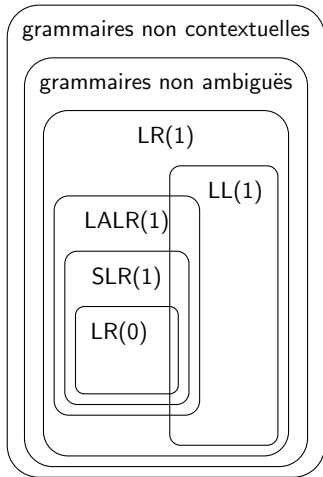


on peut également procéder par **analyse descendante** = expansions successives du non terminal le plus à gauche en partant de  $S$ , en se servant d'une table d'expansions

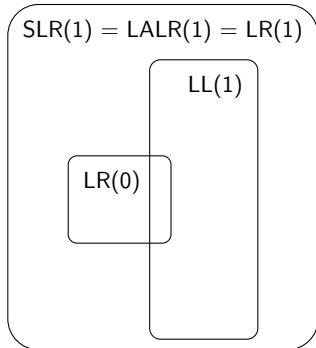
ce sont les classes de grammaires  $LL(k)$ ; cf poly chapitre 4

les analyseurs  $LL(1)$  sont relativement simples à écrire mais ils nécessitent d'écrire des grammaires peu naturelles

## grammaires



## langages



l'analyse ascendante est puissante mais le calcul des tables est complexe

le travail est automatisé par de nombreux outils

c'est la grande famille de `yacc`, `bison`, `ocamlyacc`, `cup`, `menhir`, ...  
(YACC signifie *Yet Another Compiler Compiler*)

on présente ici `cup` (pour Java) et `menhir` (pour OCaml)

on poursuit l'exemple du langage d'expressions arithmétiques avec

- des constantes entières
- des parenthèses
- une soustraction

on suppose la syntaxe abstraite et l'analyseur lexical déjà réalisés

---

## **l'outil CUP (Java)**



dans un fichier `Parser.cup`, on commence par un entête où sont déclarés les symboles terminaux et non terminaux

```
terminal Integer INT;  
terminal LPAR, RPAR, MINUS;
```

```
non terminal Expr file;  
non terminal Expr expr;
```

...

on écrit ensuite les règles de grammaires et les actions

```
start with file;

file ::=
  expr:e
  {: RESULT = e; :}
;

expr ::=
  INT:n
  {: RESULT = new Ecst(n); :}
| expr:e1 MINUS expr:e2
  {: RESULT = new Esub(e1, e2); :}
| LPAR expr:e RPAR
  {: RESULT = e; :}
;
```

on compile le fichier Parser.cup avec

```
java -jar java-cup-11a.jar -parser Parser Parser.cup
```

ce qui provoque ici une erreur :

```
Warning : *** Shift/Reduce conflict found in state #6  
between expr ::= expr MINUS expr (*)  
and      expr ::= expr (*) MINUS expr  
under symbol MINUS  
Resolved in favor of shifting.
```

```
Error : *** More conflicts encountered than expected  
-- parser generation aborted
```

on y remédie en déclarant MINUS comme étant associatif à gauche

```
precedence left MINUS;
```

(ce qui favorisera la réduction)

s'il y a plusieurs opérateurs, on les énumère par ordre de priorité croissante

```
precedence left PLUS, MINUS;  
precedence left TIMES, DIV, MOD;
```

maintenant la commande CUP termine avec succès et produit deux fichiers Java :

- `sym.java` contient la déclaration de constantes pour les lexèmes (INT, LPAR, RPAR, etc.)
- `Parser.java` contient l'analyseur syntaxique et fournit notamment un constructeur

```
Parser(Scanner scanner)
```

et une méthode

```
Symbol parse()
```

on combine les classes produites par jflex et CUP de la façon suivante

```
Reader reader = new FileReader(file);
Lexer lexer = new Lexer(reader);
Parser parser = new Parser(lexer);
Expr e = (Expr)parser.parse().value;
try {
    System.out.println(e.eval());
} catch (Error err) {
    System.out.println("error: " + err.toString());
    System.exit(1);
}
```

le programme doit utiliser la bibliothèque `java-cup-11a-runtime.jar`

---

## **l'outil Menhir (OCaml)**

dans un fichier `parser.mly`, on commence par un entête où sont déclarés les symboles terminaux et non terminaux

```
%{  
  (* code OCaml arbitraire *)  
%}
```

```
%token MINUS LPAR RPAR EOF
```

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

```
%start <expr> file
```

...

(à la différence de CUP, il faut déclarer EOF)



on écrit ensuite les règles de grammaires et les actions

```
%%  
  
file:  
  e = expr; EOF  { e }  
;  
  
expr:  
| e1 = expr; MINUS; e2 = expr { Sub (e1, e2) }  
| LPAR; e = expr; RPAR      { e }  
| i = INT                    { Cte i }  
;  
  
%%
```

(à la différence de CUP, il faut ajouter EOF)

on compile le fichier `arith.mly` de la manière suivante

```
menhir -v arith.mly
```

ce qui provoque ici un avertissement

```
Warning: one state has shift/reduce conflicts.
```

```
Warning: one shift/reduce conflict was arbitrarily resolved.
```

lorsque la grammaire n'est pas LR(1), Menhir présente les **conflits** à l'utilisateur

- le fichier `.automaton` contient une description de l'automate LR(1); les conflits y sont mentionnés
- le fichier `.conflicts` contient, le cas échéant, une explication de chaque conflit, sous la forme d'une séquence de lexèmes conduisant à deux arbres de dérivation

on y remédie en déclarant MINUS comme étant associatif à gauche

```
%left MINUS
```

(ce qui favorisera la réduction)

s'il y a plusieurs opérateurs, on les énumère par ordre de priorité croissante

```
%left PLUS MINUS  
%left TIMES DIV MOD
```

maintenant la commande `menhir` termine avec succès et produit deux fichiers OCaml `arith.ml(i)` qui contiennent notamment

- la déclaration d'un type `token`

```
type token = RPAR | MINUS | LPAR | INT of int | EOF
```

- une fonction

```
val file: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> int
```

on combine ocamllex et menhir de la façon suivante

```
let c = open_in file in
let lb = Lexing.from_file c in
let e = Parser.file Lexer.next_token lb in
...
```

- TD 3
  - analyse syntaxique de mini-Turtle
- lire les chapitres 3 et 4 du poly
- prochain cours **jeudi 28**
  - typage
  - TD : début du projet

