Éléments de base sur les analyseurs grammaticaux

- Rappels de quelques notions de théorie des langages formels
- Technique d'analyse descendante (top-down)
  - analyse descendante récursive
  - analyse descendante prédictive
- Technique d'analyse montante (bottom-up)
  - Shift-Reduce parsing
- Techniques d'analyse génériques
  - CYK
  - Earley

Éléments de vocabulaire

Définition:

Une **grammaire** est définie par un quadruplet $< N, T, R, S >$ où:

N est l'ensemble des symboles non-terminaux
T est l'ensemble des symboles terminaux
R est l'ensemble des règles de production (de la forme: $\alpha \rightarrow \beta$)
S est l'axiome de départ

Convention de notation:

- V est le vocabulaire de la grammaire $V = N \cup T$.
- Les lettres minuscules désignent dans ce document des symboles terminaux ($a, b, \ldots$).
- Les lettres majuscules désignent des symboles non-terminaux ($A, B, \ldots$).
- Les symboles grecs désignent n'importe quelle suite (éventuellement vide) de symboles terminaux et non-terminaux ($\alpha, \beta, \ldots$).
- ϵ désigne une chaîne vide.

Éléments de vocabulaire

- Soit une chaîne $\alpha A \beta$ et la règle de grammaire $A \rightarrow \gamma \in R$, alors on note par $\alpha A \beta \Rightarrow \alpha \gamma \beta$ le fait que la première chaîne produit la seconde par *substitution* de A par γ.
- Une séquence de zéro ou plus de ces substitutions est appelée une *dérivation* et est indiquée par $\stackrel{\star}{\Rightarrow}$.

Exemple: Soit la grammaire: $S \rightarrow (S)S|\epsilon$

Alors $S \stackrel{\star}{\Rightarrow} (()())()$ car:

$$S \underset{1}{\Rightarrow} (S)\underline{S} \underset{1}{\Rightarrow} (S)(S)\underline{S} \underset{2}{\Rightarrow} (S)(\underline{S})$$

$$\underset{2}{\Rightarrow} (\underline{S})() \underset{1}{\Rightarrow} ((\underline{S})S)() \underset{2}{\Rightarrow} (()\underline{S})()$$

$$\underset{1}{\Rightarrow} (()(S)\underline{S})() \underset{2}{\Rightarrow} (()(\underline{S}))() \underset{2}{\Rightarrow} (()())()$$

Éléments de vocabulaire

Il existe souvent plusieurs façons de *réduire* une chaîne (c'est par exemple le cas à plusieurs reprises dans l'exemple précédent). On appelle la *dérivation gauche* (*left-most derivation*), la dérivation obtenue en substituant le non-terminal le plus à gauche de la chaîne à réduire.

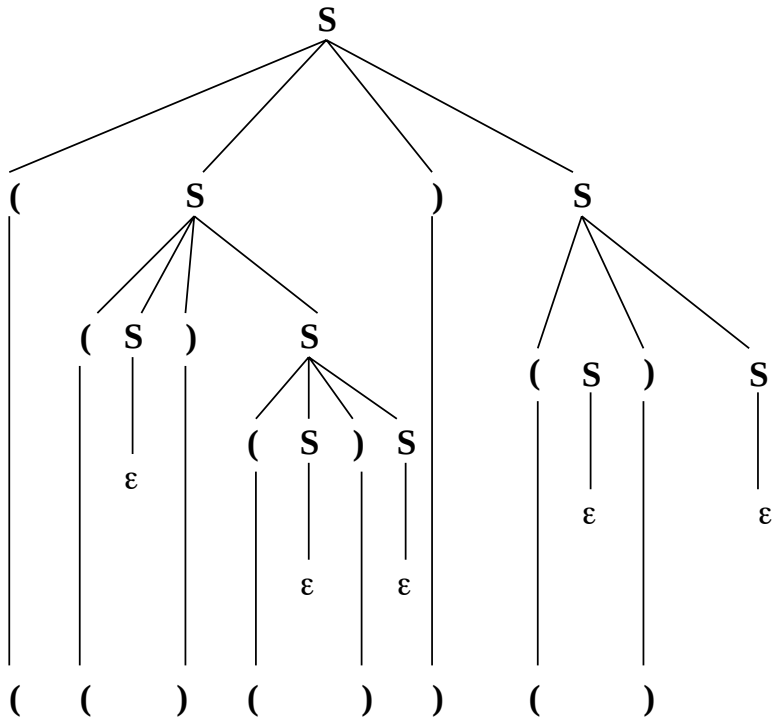
$$\underline{S} \underset{1}{\Rightarrow} (\underline{S})S \underset{1}{\Rightarrow} ((\underline{S})S)S \underset{2}{\Rightarrow} (()\underline{S})S$$

$$\underset{1}{\Rightarrow} (()(\underline{S})S)S \underset{2}{\Rightarrow} (()()\underline{S})S \underset{2}{\Rightarrow} (()())\underline{S}$$

$$\underset{1}{\Rightarrow} (()())(\underline{S})S \underset{2}{\Rightarrow} (()())()\underline{S} \underset{2}{\Rightarrow} (()())()$$

Un arbre d'analyse (*parse tree*, ou *arbre de dérivation*) est une représentation graphique d'une dérivation dans laquelle on fait abstraction de l'ordre dans lequel les non-terminaux sont dérivés.

Éléments de vocabulaire



Un parcours en profondeur de l'arbre permet d'obtenir la chaîne analysée

Éléments de vocabulaire

Une grammaire qui produit plus d'un arbre syntaxique pour une chaîne donnée est dite *ambiguë* (cad qu'elle produit plus d'une dérivation gauche (ou droite) d'une même chaîne).

Ex: $E \rightarrow E + E | E \times E | (E) | \text{id}$

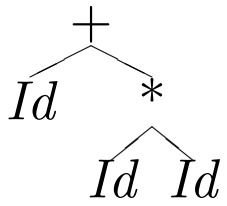
Il existe deux dérivations de la chaîne: id + id × id:

$$\begin{aligned} \underline{E} &\underset{1}{\Rightarrow} \underline{E} + E \underset{4}{\Rightarrow} \text{id} + \underline{E} \underset{2}{\Rightarrow} \text{id} + \underline{E} \times E \\ &\underset{4}{\Rightarrow} \text{id} + \text{id} \times \underline{E} \underset{4}{\Rightarrow} \text{id} + \text{id} \times \text{id} \end{aligned}$$

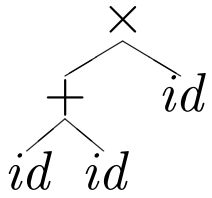
$$\begin{aligned} \underline{E} &\underset{2}{\Rightarrow} \underline{E} \times E \underset{1}{\Rightarrow} \underline{E} + E \times E \underset{4}{\Rightarrow} \text{id} + \underline{E} \times E \\ &\underset{4}{\Rightarrow} \text{id} + \text{id} \times \underline{E} \underset{4}{\Rightarrow} \text{id} + \text{id} \times \text{id} \end{aligned}$$

Chacune de ces dérivations correspond à un arbre syntaxique différent.

Éléments de vocabulaire



$$\begin{aligned}
\underline{E} &\underset{1}{\Rightarrow} \underline{E} + E \\
&\underset{4}{\Rightarrow} \text{id} + \underline{E} \\
&\underset{2}{\Rightarrow} \text{id} + \underline{E} \times E \\
&\underset{4}{\Rightarrow} \text{id} + \text{id} \times \underline{E} \\
&\underset{4}{\Rightarrow} \text{id} + \text{id} \times \text{id}
\end{aligned}$$



$$\begin{aligned}
\underline{E} &\underset{2}{\Rightarrow} \underline{E} \times E \\
&\underset{1}{\Rightarrow} \underline{E} + E \times E \\
&\underset{4}{\Rightarrow} \text{id} + \underline{E} \times E \\
&\underset{4}{\Rightarrow} \text{id} + \text{id} \times \underline{E} \\
&\underset{4}{\Rightarrow} \text{id} + \text{id} \times \text{id}
\end{aligned}$$

Lever une ambiguïté

Idée:

- on introduit les opérateurs les moins prioritaires en premier dans la grammaire (par ajout de non-terminaux),
- les opérateurs associatifs à gauche (ex: $a + b + c = ((a + b) + c)$) sont traduits par des règles récursives à gauche ($A \rightarrow A\alpha$).

$E \rightarrow E+T$	$E \rightarrow E\text{-}T$	$E \rightarrow T$
$T \rightarrow T*F$	$T \rightarrow T/F$	$T \rightarrow F$
$F \rightarrow id$	$F \rightarrow (E)$	

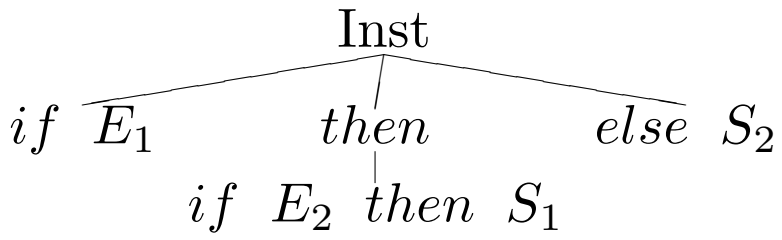
$\hookrightarrow$ seule la première dérivation est possible avec cette nouvelle grammaire.

Autre cas classique d'ambiguïté: la construction *if-then-else*

La grammaire suivante est ambiguë:

Inst	$\rightarrow$	**if** Expr **then** Inst
Inst	$\rightarrow$	**if** Expr **then** Inst **else** Inst
Inst	$\rightarrow$	**other**

Car la chaîne: **if** E_1 **then if** E_2 **then** S_1 **else** S_2 a deux arbres syntaxiques différents.

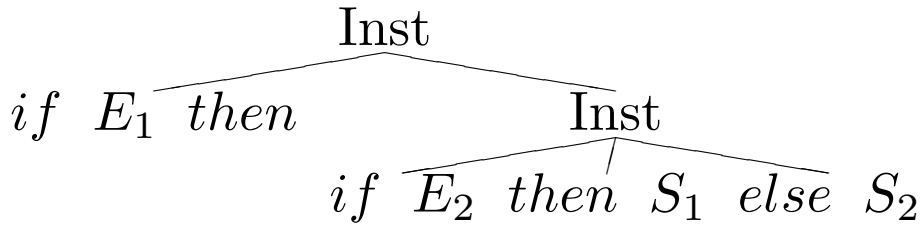


la construction *if-then-else*

La grammaire suivante est ambiguë:

Inst	$\rightarrow$	**if** Expr **then** Inst
Inst	$\rightarrow$	**if** Expr **then** Inst **else** Inst
Inst	$\rightarrow$	**other**

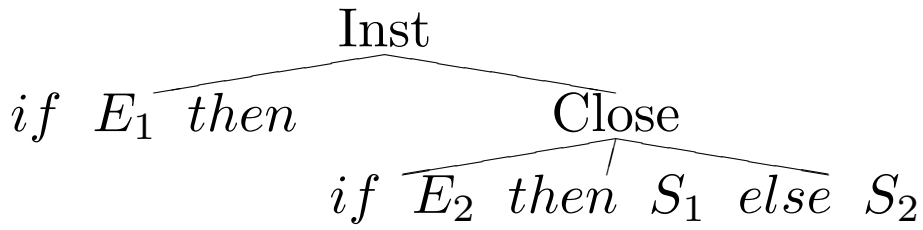
Car la chaîne: **if** E_1 **then if** E_2 **then** S_1 **else** S_2 a deux arbres syntaxiques différents.



la construction *if-then-else*

En voici une version non ambiguë

Inst	$\rightarrow$	**if** Expr **then** close **else** Inst
	$\rightarrow$	**if** Expr **then** Inst
	$\rightarrow$	**Close**
Close	$\rightarrow$	**if** Expr **then** close **else** close
	$\rightarrow$	**other**



Note: il n'existe pas d'algorithme capable de déterminer si une grammaire est ambiguë ou pas

Classification chomskienne

type	contraintes sur R	langages	analyse
3	A $\rightarrow$ aB ou A $\rightarrow$ a	réguliers	$\mathcal{O}(n)$
2	A $\rightarrow \alpha$	hors-contexte	$\mathcal{O}(n^3)$
1	$\alpha \rightarrow \beta$ $avec$ $\lvert\alpha\rvert \leq \lvert\beta\rvert$	contextuels	$\mathcal{O}(n^6)$
0	$\alpha \rightarrow \beta$	récursivement énumérable	—

Les langages réguliers

$\hookrightarrow$ représentables par une expression régulière (ou un automate, ou une grammaire régulière)

Définition: d'une expression régulière (ER)

- les éléments terminaux sont des ER,
- si a et b sont des ER, alors ab est une ER (concaténation),
- si a est une ER, alors $a^\star$ est une ER (nb. quelconque de fois de a),
- si a est une ER, alors (a) est une ER
- si a et b sont des ER, alors $a|b$ est une ER (a OU b)

Note: on utilise souvent des ER étendues:

$(a\|b)+$	une fois ou plus	$(a\|b)(a\|b)^\star$
$[a-z]$	de a à z	$a\|b\|c\|d\|\ldots\|z$

Les langages réguliers

Ex 1: $\{a^\star b^\star\}$ représente l'ensemble des chaînes constituées d'un nombre quelconque (éventuellement nul) de a suivi d'un nombre quelconque (éventuellement nul) de b.

abb, a, b, bbb sont des exemples de telles chaînes.

$$\text{grammaire équivalente} \begin{cases} A \rightarrow aA|bB|\epsilon \\ B \rightarrow bB|\epsilon \end{cases}$$

Ex 2: $\{0\,|\,(0|1)^\star 0\}$ représente l'ensemble des chaînes binaires paires (simplifiable).

Ex 3: un exemple de (mauvaise) ER capable de reconnaître les adresses emails dans un document (en vue par exemple d'envoyer du spam):
$[A-Z, a-z, 0-9, -, , .] + @[A-Z, a-z, 0-9, -, , .]^\star.[a-z, A-Z]^\star$

Les langages réguliers et Unix

```
>cat fic
```

```
chaffars:Chaffar, Soumaya:  XXXXX:
drouinta:Drouin-Trempe, Antoine:  XXXXX:
leberrej:Le Berre, Jean-François:  XXXXX:
leflocam:Le Floch, Amélie:  XXXXX:
leplusth:Leplus, Thomas:  XXXXX:
mabroukm:Mabrouk, Moez:  XXXXX:
merdaoub:Merdaoui, Badis:  XXXXX:
morissm:Morissette, Marc-André:  XXXXX:
talebikb:Taleb, Ikbal:  XXXXX:
tawbebil:Tawbe, Bilal:  XXXXX:
ulrichal:Ulrich, Alexis:  XXXXX:
```

```
>sed -e "s/\([^:]*\):\([^:]*\),\(.*\):\([^:]*\):/\3 \2/g" fic
```

Les langages réguliers et Unix

```
>sed -e "s/\([^:]*\):\([^:]*\),\(.*\):\([^:]*\):/\3 \2/g" fic
```

```
Soumaya Chaffar
Antoine Drouin-Trempe
Jean-François Le Berre
Amélie Le Floch
Thomas Leplus
Moez Mabrouk
Badis Merdaoui
Marc-André Morissette
Ikbal Taleb
Alexis Ulrich
```

Note: `>sed -e "s/[^:]*:\([^:]*\),\(.*\):[^:]*:/\2 \1/g" fic`

Considérations concernant les langages

Voici des exemples de langages non réguliers:

- $\{a^n b^n / n \geq 1\}$
- le langage des expressions correctement parenthésées
- le langage de l'exemple suivant, etc.

$\hookrightarrow$ On peut exprimer un plus grand nombre de langages à l'aide de grammaires hors-contexte (type 2).

Exemple de langage de type 2: le langage dont les chaînes sur l'alphabet $\{a, b\}$ contiennent un nombre égal (éventuellement nul) de a et de b.

S $\rightarrow$ aSbS
S $\rightarrow$ bSaS
S $\rightarrow \epsilon$

Considérations concernant les langages

Il existe des langages qui ne peuvent être décrits par une grammaire hors-contexte:

Ex1: $\{wcw/w \in (a|b)^*\}$ (ex: $aabcaab$)

$\hookrightarrow$ mais le langage $\{wcw^R\}$ (ex: $aabcbaa$) est hors-contexte, car ($S \rightarrow aSa|bSb|c$)

Ex 2: $\{a^n b^m c^n d^m / n \geq 1, m \geq 1\}$ (ex: $aabbbccddd$)

$\hookrightarrow$ mais le langage $\{a^n b^m c^m d^n / n \geq 1, m \geq 1\}$ (ex: $aabcdd$) est hors-contexte car:

$$S \rightarrow aSd|aAd$$
$$A \rightarrow bAc|bc$$

Ex 3: $\{a^n b^n c^n / n \geq 0\}$ (ex: $aabbcc$)

$\hookrightarrow$ mais le langage $\{a^n b^n / n \geq 1\}$ (ex: $aabb$) est hors-contexte ($S \rightarrow aSb|ab$)

$\hookrightarrow$ mais le complément de ce langage est hors-contexte

Considérations concernant les langages

Exemple de grammaire contextuelle pour le langage $a^n b^n c^n$:

S $\rightarrow$ aBC	bC $\rightarrow$ bc
S $\rightarrow$ SABC	aA $\rightarrow$ aa
CA $\rightarrow$ AC	aB $\rightarrow$ ab
BA $\rightarrow$ AB	bB $\rightarrow$ bb
CB $\rightarrow$ BC	cC $\rightarrow$ cc

Exemple:

S $\Rightarrow$ SABC $\Rightarrow$ aBCABC $\Rightarrow$ aBACBC $\Rightarrow$ aBABCC $\Rightarrow$ aABBCC $\Rightarrow$ aaBBCC $\Rightarrow$ aabBCC $\Rightarrow$ aabbCC $\Rightarrow$ aabbcC $\Rightarrow$ aabbcc

Considérations concernant les langages

Question: peut-on représenter une langue naturelle par une grammaire hors-contexte ?

- S $\rightarrow$ NP VP ne capture pas l'accord en genre et nombre entre le sujet et le groupe verbal. Mais $S_{plur} \rightarrow NP_{plur} VP_{plur}$ le fait.

- Dépendances arbitrairement longues[1], où le groupe prépositionnel n'a pas d'objet (implicite):

  Whom did Fred give the ball to ?
  Whom does Alice believes Fred wants to give the ball to ?

  $\hookrightarrow$ On peut néanmoins représenter ces phrases avec une grammaire hors-contexte en introduisant de nouveaux symboles non terminaux: $S \rightarrow$ whom S/NP.

[1] Pris dans Charniak [1993], p.9

Considérations concernant les langages

Question: peut-on représenter une langue naturelle par une grammaire hors-contexte ?

En fait plusieurs chercheurs se sont posé la question (Jurafsky and Martin [2000]).

$\hookrightarrow$ Deux preuves (non réfutées) semblent infirmer l'hypothèse pour le suisse allemand et le bambara. . . .

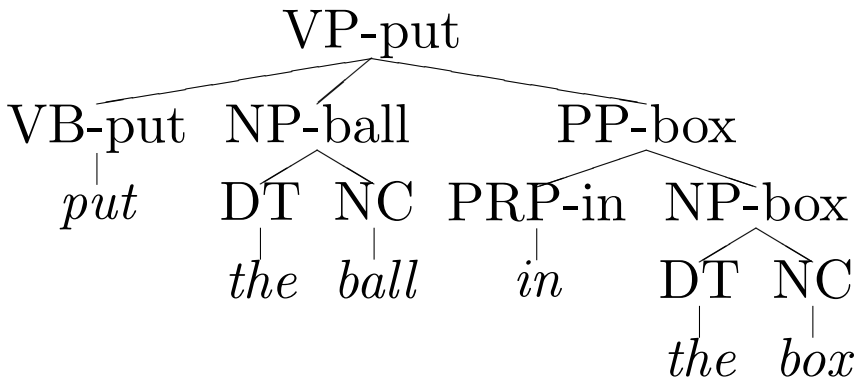
Question: l'être humain a-t-il une représentation hors-contexte du langage ?

$\hookrightarrow$ pas de réponse claire. Lire encore une fois Jurafsky and Martin [2000] p.350-352 pour plus d'information à ce sujet.

Analyse descendante récursive

- intuitif, simple à mettre en œuvre,
- construit un arbre du haut vers le bas (en pré-ordre)
- le plus simple de la famille des analyseurs LL (lecture de gauche à droite (left-to-right) de l'input à analyser, produit une dérivation gauche (leftmost-derivation))

```
VP(VERB("put","put").
   NP(DET("the").
      NOUN("ball"),
      "ball").
   PP( PREP("in","in").
       NP(DET("the").
       NOUN("box"),
       "box"),"box") ,"put");
```



Analyse descendante récursive

input:

```
VP(VERB("put","put").
   NP(DET("the").
      NOUN("ball"),
      "ball").
   PP( PREP("in","in").
       NP(DET("the").
       NOUN("box"),
       "box"),"box")
   ,"put");
```

LaTeX:

```
\begin{parsetree}
(.VP-put.
   (.VB-put. ‘put’)
   (.NP-ball.
     (.DT. ‘the’)
     (.NC. ‘ball’)
   )
   (.PP-box.
     (.PRP-in. ‘in’)
     (.NP-box.
        (.DT. ‘the’)
        (.NC. ‘box’)
     )
   )
)
\end{parsetree}
```

Analyse descendante récursive

Sent $\rightarrow$ PH(Node)
Node $\rightarrow$ Cat(Fils Suite) Frere
Frere $\rightarrow \epsilon$ | . Node
Fils $\rightarrow$ Mot | Node
Suite $\rightarrow \epsilon$ | , Mot
Cat $\rightarrow$ *tout sauf un separateur*
Mot $\rightarrow$ "*suite de lettres*"

- une méthode par symbole non-terminal
- on entre dans une méthode avec le premier symbole de l'input à analyser
- on sort d'une méthode avec le premier symbole que la méthode n'a pas analysé

Analyse descendante récursive

Plus facile si on implémente un analyseur lexical:

- soit `lex` le prochain **lexème** à analyser
- soit `next()` une fonction qui récupère le prochain lexème (réaffecte `lex`)

Exemple pour la règle: Sent $\rightarrow$ PH(Node):

```
methode sent() {
  if (lex != "PH") Error("PH attendu");
   next();
   if (lex != "(") Error("( attendue");
   next();
   node();
   if (lex != ")") Error(") attendue");
}
```

Note: l'émission d'une erreur systématique enlève la possibilité de backtrack

Analyse descendante récursive: problèmes

Ne fonctionne qui si on est capable de décider à tout moment de la bonne action à prendre au cours de l'analyse (la grammaire doit être LL_1).

$\hookrightarrow$ Une alternative avec retour-arrière (back-tracking) est possible mais inefficace.

S $\rightarrow$ c A d
A $\rightarrow$ a b
A $\rightarrow$ a

chaîne à analyser $w = cad$, alors:

$$S \underset{1}{\Rightarrow} cAd \underset{2}{\Rightarrow} cabd \text{ (**blocage**)}$$
$$\underset{3}{\Rightarrow} cad$$

Analyse descendante récursive: problèmes

Boucle si la grammmaire est récursive à gauche (directement ou pas). Il existe un algorithme pour rendre une grammaire non récursive à gauche.

Exemple de récursion (gauche) directe:

$$A \rightarrow A + B$$

Exemple de récursion (gauche) indirecte:

$$A \rightarrow B\ a\ C$$
$$B \rightarrow A\ b$$

En fait la récursion gauche est un problème pour tout analyseur descendant (*top-down*).

Factorisation gauche

Idée: si $A \rightarrow \alpha\beta_1|\alpha\beta_2$, alors réécrire ces deux règles avec: $A \rightarrow \alpha A'$ et $A' \rightarrow \beta_1|\beta_2$

Algo: Pour chaque non-terminal A, chercher le plus long préfixe α commun de deux ou plusieurs de ses alternatives. Si ce préfixe existe, alors réécrire toutes les A-productions: $A \rightarrow \alpha\beta_1| \ldots |\alpha\beta_n|\gamma$ par:

$$\begin{array}{lcl} A & \rightarrow & \alpha A'|\gamma \\ A' & \rightarrow & \beta_1| \ldots |\beta_n \end{array}$$

où γ désigne ici toutes les autres alternatives ne commencant pas par α. Recommencer jusqu'à ce qu'aucune alternative d'un non-terminal ne partage un préfixe commun.

$$\left\{ \begin{array}{lcl} S & \rightarrow & iEtS|iEtSeS|a \\ E & \rightarrow & b \end{array} \right\} \text{devient:} \left\{ \begin{array}{lcl} S & \rightarrow & iEtSS'|a \\ S' & \rightarrow & eS|\epsilon \\ E & \rightarrow & b \end{array} \right\}$$

Récursivité gauche: directe

Définition: Une grammaire est récursive gauche s'il existe (au moins) un A tel que $A \stackrel{\star}{\Rightarrow} A\alpha$.

Élimination de la récursivité gauche directe:

- On remplace les règles $A \rightarrow A\alpha|\beta$ par:

$$\begin{array}{lcl} A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A'|\epsilon \end{array}$$

- Soit dans le cas plus général:
  $A \rightarrow A\alpha_1|A\alpha_2|\ldots|A\alpha_m|\beta_1|\beta_2|\ldots|\beta_n$ avec β_i ne commencant pas par A, se remplace par:

$$\begin{array}{lcl} A & \rightarrow & \beta_1 A'|\ldots|\beta_n A' \\ A' & \rightarrow & \alpha_1 A'|\ldots|\alpha_m A'|\epsilon \end{array}$$

Récursivité gauche indirecte

Il existe un algorithme qui est garanti de fonctionner si la grammaire de départ n'a ni **cycle** ($A \stackrel{\star}{\Rightarrow} A$) ni **$\epsilon$-production** ($A \rightarrow \epsilon$).

fixer un ordre $(1 \rightarrow n)$ sur les non-terminaux
for all $i \in [1, n]$ **do**
  for all $j \in [1, i-1]$ **do**
    Remplacer $A_i \rightarrow A_j\gamma$ par $A_i \rightarrow \delta_1\gamma| \ldots |\delta_k\gamma$ où:
    $A_j \rightarrow \delta_1| \ldots |\delta_k$ sont toutes les A_j-productions courantes
  Eliminer les récursions gauches directes parmi les A_i-productions

Note: Cet algorithme peut produire une grammaire qui contient des ϵ-productions.

Récursivité gauche

Soit la grammaire: $S \rightarrow Aa|b \quad A \rightarrow Ac|Sd|\epsilon$
Ordre (arbitraire): S, A

$i = 1$ pas de récursivité directe sur $S \hookrightarrow$ rien à faire

$i = 2$ (pour $j = 1$) il existe une règle $A \rightarrow Sd$

que l'on remplace par $A \rightarrow Aad|bd$.

On retire alors la récursivité directe sur A de $A \rightarrow Ac|Aad|bd|\epsilon$

Ce qui nous donne la grammaire:

$$\begin{array}{lcl} S & \rightarrow & Aa|b \\ A & \rightarrow & bdA'|A' \\ A' & \rightarrow & cA'|adA'|\epsilon \end{array}$$

Analyseur prédictif non récursif

Input: une chaîne w à analyser, une table M, S l'axiome
Output: une dériv. gauche si w appartient au langage, une erreur sinon

push \$; push S; $I = w\$$; $ip \leftarrow 1$
repeat
  $X \leftarrow top()$ et $a \leftarrow I[ip]$
  if X est un terminal ou \$ **then**
    if $X = a$ **then** POP(X); ip++ **else** error()
  else
    if $M[X, a] = X \rightarrow Y_1 Y_2 \ldots Y_k$ **then**
      pop X
      push $Y_k Y_{k-1} \ldots Y_1$
      output $X \rightarrow Y_1 Y_2 \ldots Y_k$
    else
      error()
until X = \$ /* pile vide */

Analyseur prédictif non récursif

- Soit la grammaire (non récursive gauche):

$$E \to TE' \qquad E' \to +TE'|\epsilon \quad T \to FT'$$
$$T' \to \times FT'|\epsilon \quad F \to (E)|id$$

- la table d'analyse associée:

	Input symbols					
N	id	+	×	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to \times FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

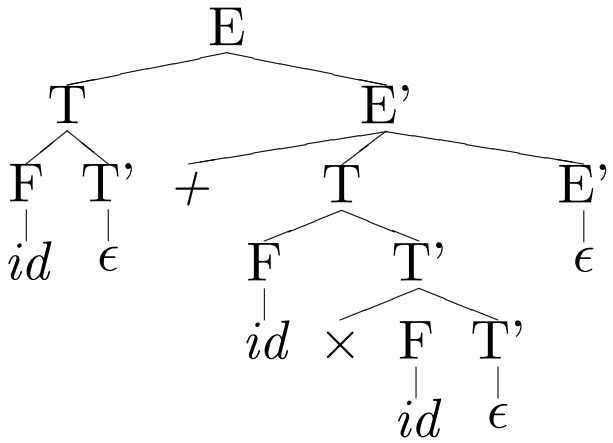
- et la chaîne à analyser $id + id \times id$

Analyseur prédictif non récursif

pile	input	output
\$E	id + id × id \$	
\$E'T	id + id × id \$	$E \rightarrow TE'$
\$E'T'F	id + id × id \$	$T \rightarrow FT'$
\$E'T'id	id + id × id \$	$F \rightarrow id$
\$E'T'	+ id × id \$	
\$E'	+ id × id \$	$T' \rightarrow \epsilon$
\$E'T+	+ id × id \$	$E' \rightarrow +TE'$
\$E'T	id × id \$	
\$E'T'F	id × id \$	$T \rightarrow FT'$
\$E'T'id	id × id \$	$F \rightarrow id$
\$E'T'	× id \$	
\$E'T'F×	× id \$	$T' \rightarrow \times FT'$
\$E'T'F	id \$	
\$E'T'id	id \$	$F \rightarrow id$
\$E'T'	\$	
\$E'	\$	$T' \rightarrow \epsilon$
\$	\$	$E' \rightarrow \epsilon$
succès		

Analyseur prédictif non récursif

$\hookrightarrow$ Il est alors facile de reconstruire la structure car ce que sort l'analyseur prédictif est la liste des dérivations gauches à appliquer pour dériver la chaîne depuis l'axiome.



$\Rightarrow id + (id \times id)$

Construction de la table d'analyse

Deux opérateurs définissant des ensembles à valeur dans T (les symboles terminaux):

FIRST(α) : pour toute chaîne α de terminaux et non-terminaux

C'est l'ensemble des terminaux qui commencent les chaînes dérivées de α.

cad: $\{a \in T / \alpha \stackrel{\star}{\Rightarrow} a\beta\}$

Si $\alpha \stackrel{\star}{\Rightarrow} \epsilon$ alors ϵ est ajouté à FIRST(α).

FOLLOW(A) : pour tout non-terminal

C'est l'ensemble des terminaux qui peuvent apparaître dans les S-dérivations directement à droite de A.

cad: $\{a \in T / \exists\ S \stackrel{\star}{\Rightarrow} \alpha A a \beta\}$

Si A peut être le symbole le plus à droite dans une S-dérivation, alors ajouter \$ à FOLLOW(A).

Analyseur prédictif non récursif: Construction de la table d'analyse

Idée: S'il existe $A \rightarrow \alpha$ et que a est dans FIRST(α) alors appliquer $A \rightarrow \alpha$ à la lecture de a.

Input: Grammaire G
Output: Table d'analyse M

1. Pour chaque production $A \rightarrow \alpha$ appliquer 2 et 3
2. Pour chaque terminal a dans FIRST(α), $M[A, a]$ += $A \rightarrow \alpha$
3. Si ϵ est dans FIRST(α), alors $M[A, b]$ += $A \rightarrow \alpha$ pour tout terminal b dans FOLLOW(A).
   Si ϵ est dans FIRST(α) et que \$ est dans FOLLOW(A) alors $M[A, \$]$ += $A \rightarrow \alpha$
4. Les entrées vides sont des erreurs.

Analyseur prédictif non récursif: Calcul de FIRST

1. Appliquer en boucle les règles suivantes, tant que l'on effectue des ajouts:
   - si $X \in T$, alors FIRST$(X) = \{X\}$
   - si $X \to \epsilon$ alors ajouter ϵ à FIRST(X)
   - si $X \to Y_1 Y_2 \ldots Y_k$ alors ajouter FIRST(Y_1) (sauf ϵ) à FIRST(X).
     - si ϵ est dans FIRST(Y_1) alors ajouter FIRST(Y_2) à FIRST(X); etc.
     - si ϵ est dans tous les Y_i alors ajouter ϵ à FIRST(X).

2. Pour toute séquence $X = X_1 X_2 \ldots X_k$: ajouter tous les symboles (sauf ϵ) de FIRST(X_1) à FIRST(X).
   - si ϵ est dans FIRST(X_1), alors ajouter les non-ϵ symboles de FIRST(X_2) à FIRST(X); etc.
   - si $\epsilon \in$ FIRST(X_i), $\forall i \in [1, k]$ alors ajouter ϵ à FIRST(X)

Calcul de FIRST: exemple

$$
\begin{array}{lll}
\mathsf{E} \to \mathsf{T\ E'} & \mathsf{E'} \to + \mathsf{\ T\ E'} & \mathsf{E'} \to \epsilon \\
\mathsf{T} \to \mathsf{F\ T'} & \mathsf{T'} \to \times \mathsf{\ F\ T'} & \mathsf{T'} \to \epsilon \\
\mathsf{F} \to \mathsf{(E)} & \mathsf{F} \to \mathsf{id} &
\end{array}
$$

Par exemple:

$$
\begin{array}{ll}
\mathsf{FIRST(E)} & = \mathsf{FIRST(T)} = \mathsf{FIRST(F)} = \{(, id\} \\
\mathsf{FIRST(E')} & = \{+, \epsilon\} \\
\mathsf{FIRST(T')} & = \{\times, \epsilon\} \\
\mathsf{FIRST(E'T')} & = \{+, \times, \epsilon\}
\end{array}
$$

Calcul de FOLLOW

Appliquer ces règles tant que l'on peut ajouter des symboles:

1. Mettre \$ dans FOLLOW(S) où S est l'axiome de départ
2. si $A \rightarrow \alpha B \beta$, alors mettre tous les non-ϵ symboles de FIRST(β) dans FOLLOW(B)
3. si il existe $A \rightarrow \alpha B$ ou $A \rightarrow \alpha B \beta$ avec $\epsilon \in$ FIRST(β), alors tout ce qui est dans FOLLOW(A) est dans FOLLOW(B).

Exemple:

$$\begin{aligned} \text{FOLLOW(E)} &= \text{FOLLOW(E')} = \{), \$\} \\ \text{FOLLOW(T)} &= \text{FOLLOW(T')} = \{+,), \$\} \\ \text{FOLLOW(F)} &= \{+, \times,), \$\} \end{aligned}$$

Grammaire LL1

Définition: Une grammaire pour laquelle on peut calculer une table d'analyse de telle manière qu'il n'existe au plus qu'une seule règle dans chaque case de M est dite LL(1).

Note: Le premier L est pour left-to-right, le second pour parce que l'on produit une dérivation gauche, le chiffre entre parenthèse est ce qu'on appelle le $look\ ahead$.

Propriété: Une grammaire LL1 ne peut pas être ambiguë ou récursive à gauche.

Propriété: G est LL1 **ssi** pour toute paire de productions $A \rightarrow \alpha|\beta$:

1. Il n'existe pas de termimal a tel que $\alpha \stackrel{\star}{\Rightarrow} a\alpha_1$ **et** $\beta \stackrel{\star}{\Rightarrow} a\beta_1$
2. Au plus α ou β peut dériver ϵ
3. Si $\beta \stackrel{\star}{\Rightarrow} \epsilon$, alors α ne peut pas dériver une chaîne qui débute par un terminal dans FOLLOW(A).

Analyse par décalage réduction (Shift-Reduce Parsing)

Technique d'analyse montante (bottom-up) qui n'a pas de problème avec les grammaires récursives à gauche. Nécessite: une file de lecture (I), une pile (P) et une table d'analyse comportant l'une des 4 actions:

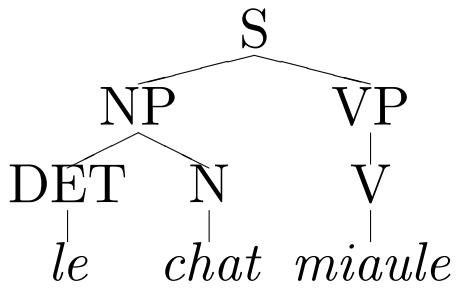
- shift du symbole en tête de I sur le sommet de la pile P.
- reduction de la séquence s en sommet de pile par une partie gauche de règle qui a s pour partie droite
- accept si I est vide et P contient l'axiome
- erreur

Shift-Reduce Parsing: Exemple

Pile	Input	Actions
	le chat miaule	shift
le	chat miaule	reduce DET $\rightarrow$ le
DET	chat miaule	shift
DET,chat	miaule	reduce N $\rightarrow$ chat
DET,N	miaule	reduce NP $\rightarrow$ DET N
NP	miaule	shift
NP,miaule		reduce V $\rightarrow$ miaule
NP,V		reduce VP $\rightarrow$ V
NP,VP		reduce S $\rightarrow$ NP VP
S		accept

Shift-Reduce Parsing: Exemple

Arbre associé (si on lit les réductions en commencant par le bas, on a une dérivation droite de la chaîne analysée):



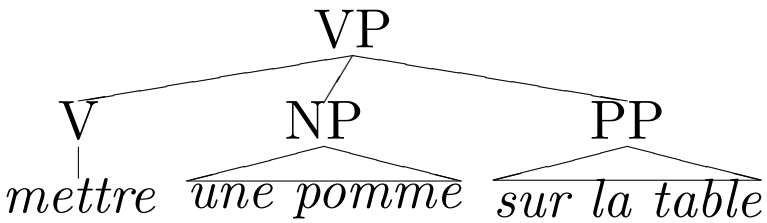
Deux types de conflits peuvent se produire avec ce type de méthode:

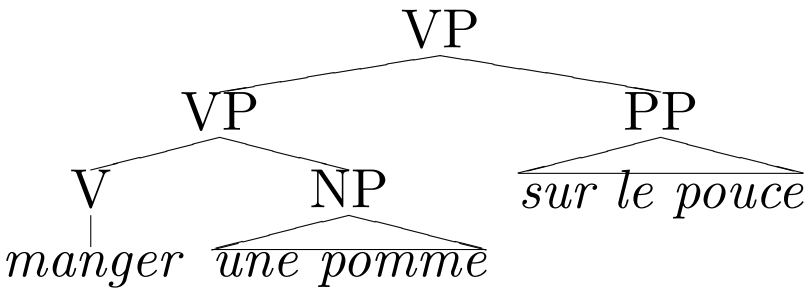
- shift-reduce: on ne sait pas s'il faut réduire ou décaler
- reduce-reduce: il existe plusieurs réductions possibles dans un état donné

Conflit Shift-Reduce

Soit la grammaire: VP $\rightarrow$ V NP PP | VP $\rightarrow$ VP PP | VP $\rightarrow$ V NP

Pile	Input	Action
V,NP	sur la table	shift ou reduce VP $\rightarrow$ V NP ?

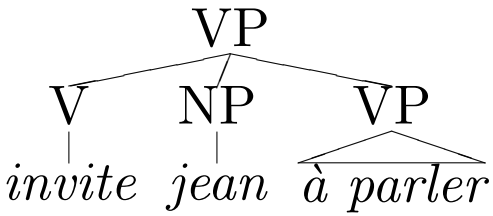


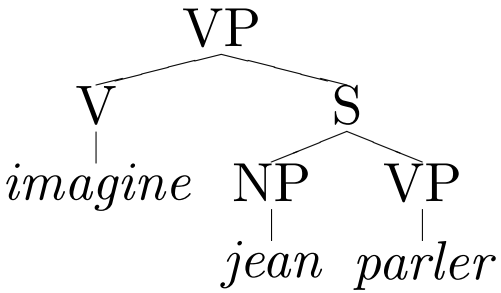


Conflit Reduce-Reduce

Soit la grammaire: VP $\rightarrow$ V NP VP | VP $\rightarrow$ V S | S $\rightarrow$ NP VP

Pile	Input	Action
V,NP,VP		reduce VP $\rightarrow$ V NP VP ou S $\rightarrow$ NP VP ?





Analyse shift-reduce populaire: analyse LR(k)

Il existe des programmes (ex: `Yacc`) capables de construire une table d'analyse LR(k) étant donnée une grammaire hors-contexte.

Le principe consiste à pré-calculer (sous forme d'un automate) toutes les substitutions possibles d'une forme S-dérivée que l'on pourra rencontrer lors de l'analyse d'une chaîne étant donnée une grammaire. Cet automate fournit toutes les informations dont on a besoin pour compiler une table d'analyse LR.

Note: toute grammaire hors-contexte ne peut pas nécessairement être analysée à l'aide de cette technique.

Analyse LR(k): construction de la table

- La construction de l'automate consiste à construire des *ensembles d'items*.

- L'opération de base permettant de construire un ensemble d'items est la *fermeture*. La fermeture d'un item $I \equiv \alpha \bullet A\beta$ est définie par l'ensemble des items $A \rightarrow \bullet\gamma$ que l'on peut créer à partir des règles de G de la forme: $A \rightarrow \gamma$

- La fermeture d'un ensemble d'items est constituée de l'union de la fermeture de chacun de ses items.

Les états de l'automate sont les ensembles d'items construits. Les transitions correspondent à tout symbole (terminal ou non) se situant directement à droite de $\bullet$ dans un ensemble d'items.

Analyse LR(k): construction de la table

Soit G: $\{\{E, T\}, \{id, (,), +)\}, E, \{\mathsf{E} \to \mathsf{E} + \mathsf{T}, \mathsf{E} \to \mathsf{T}, \mathsf{T} \to \mathsf{id}, \mathsf{T} \to (\mathsf{E})\}\}$

Le premier ensemble (I_0) contient l'item $S' \to \bullet S\$$ où S' est un méta-axiome et sa fermeture. Sur la lecture d'une parenthèse ouvrante (par exemple), on obtient un nouvel ensemble d'items (I_6).

I_0

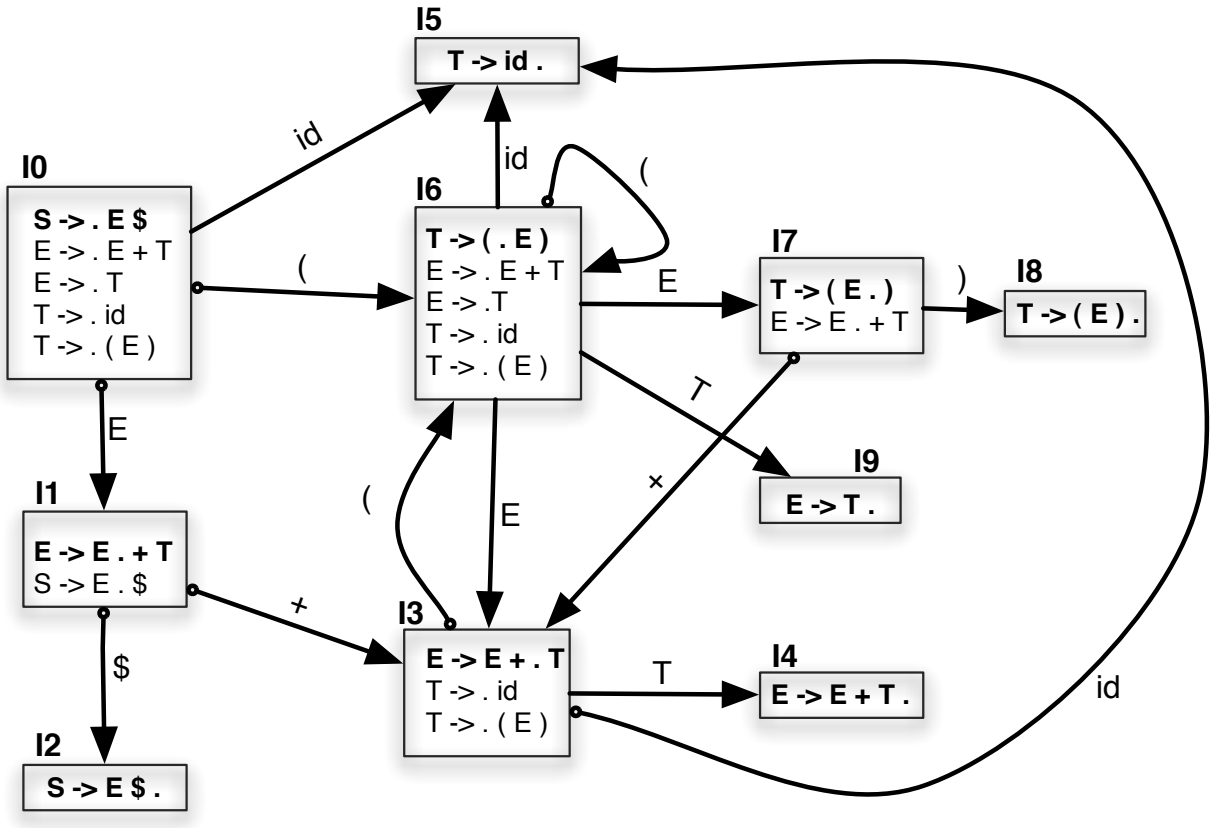
S → • E
E → • E + T
E → • T
T → • id
T → • (E)

I_6

T → (• E)
E → • E + T
E → • T
T → • id
T → • (E)

Analyse LR(k): construction de la table

En continuant ce processus, on obtient un automate:



Analyse LR(k): construction de la table SLR(0)

La table d'analyse est construite comme suit (table LR(0)):

- Une transition de l'état I vers l'état I' sur un symbole terminal a correspond à une action: $\text{ACTION}[\text{I,a}] = \text{SHIFT-I'}$
- Une transition de l'état I vers l'état I' sur un symbole non-terminal A correspond à un goto: $\text{GOTO}[\text{I,A}] = \text{I'}$
- Pour tous les états I contenant un item $r \equiv \text{A} \rightarrow \alpha \bullet$, ajouter l'action: $\text{ACTION}[\text{I,a}] = \text{RÉDUIRE-}r$ pour tout $a \in T$
- Pour tout état I contenant l'item $\text{S} \rightarrow \alpha \bullet$ ajouter l'action: $\text{ACTION}[\text{I,a}] = \text{ACCEPT}$ pour tout $a \in T$
- Les cases vides sont des erreurs d'analyse

Une grammaire pour laquelle une telle table peut être construite sans qu'aucune case ne contienne deux actions est dite LR(0).

Analyse LR(k): construction de la table SLR(0)

Voici la table correspondant à l'automate précédent:

2 $E \rightarrow E + T$
3 $E \rightarrow T$
4 $T \rightarrow id$
5 $T \rightarrow (E)$

	id	(	)	+	$	S	E	T
0	S5	S6					1	9
1				S3	S2			
2	accept							
3	S5	S6						4
4	R2	R2	R2	R2	R2			
5	R4	R4	R4	R4	R4			
6	S5	S6					7	9
7			S8	S3				
8	R5	R5	R5	R5	R5			
9	R3	R3	R3	R3	R3			

Analyse LR(k)

push(0) // état initial

loop
  s ← top()
  if (ACTION[s,first(w)] == SHIFT-n) **then**
    push(n); remove-first(w)
  else if (ACTION[s,first(w)] == REDUCE-rule) **then**
    soit $rule \equiv \text{A} \rightarrow A_1 \ldots A_k$
    k times **do** pop()
    push(GOTO[top(),A])
  else if (ACTION[s,first(w)] == ACCEPT) **then**
    succes()
  else
    error()

Analyse LR(k)

pile	input	actions		
0/S	id + (id + id)	S5		
0/S 5/id	+ (id + id)	R4	T → id	GOTO 9
0/S 9/T	+ (id + id)	R3	E → T	GOTO 1
0/S 1/E	+ (id + id)	S3		
0/S 1/E 3/+	(id + id)	S6		
0/S 1/E 3/+ 6/(	id + id)	S5		
0/S 1/E 3/+ 6/(5/id	+ id)	R4	T → id	GOTO 9
0/S 1/E 3/+ 6/(9/T	+ id)	R3	E → T	GOTO 1
0/S 1/E 3/+ 6/(1/E	+ id)	S3		
0/S 1/E 3/+ 6/(1/E 3/+	id)	S5		
0/S 1/E 3/+ 6/(1/E 3/+ 5/id	)	R4	T → id	GOTO 4
0/S 1/E 3/+ 6/(1/E 3/+ 4/T	)	R2	E → E+T	GOTO 7
0/S 1/E 3/+ 6/(7/E	)	S8		
0/S 1/E 3/+ 6/(7/E 8/)	$	R5	T → (E)	GOTO 4
0/S 1/E 3/+ 4/T	$	R2	E → E+T	GOTO 1
0/S 1/E	$	S2		
0/S 1/E 2/$	$	ACCEPT		

Analyse LR(k): illustration d'un conflit reduce-reduce

Si dans un ensemble d'items, deux items sont de la forme A $\rightarrow \alpha \bullet$ alors, nous avons un conflit reduce-reduce. On peut tenter d'éviter le conflit en changeant la façon de construire la table $\rightarrow$ tables SLR(1):

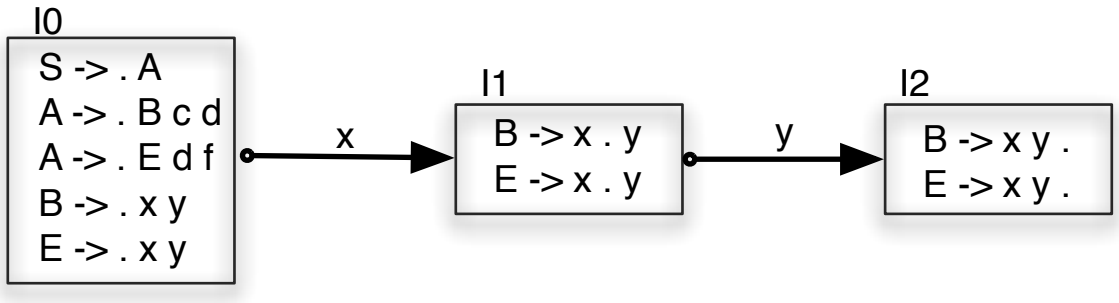
pour un item A $\rightarrow \alpha \bullet$ dans un ensemble I, ajouter pour tout a dans FOLLOW(A):

ACTION[I,a] = REDUCE-A $\rightarrow \alpha$

Si le conflit n'est toujours pas résolu, alors il existe d'autres heuristiques pour construire la table (LALR, LR(k)). Si le conflit persiste, l'expert doit trancher.

Analyse LR(k): illustration d'un conflit reduce-reduce

$A \rightarrow Bcd$
$A \rightarrow Edf$
$B \rightarrow xy$
$E \rightarrow xy$



	c	d	f	x	y	$	A	B	E
I_2	R3 R4	R3 R4	R3 R4	R3 R4	R3 R4	R3 R4			

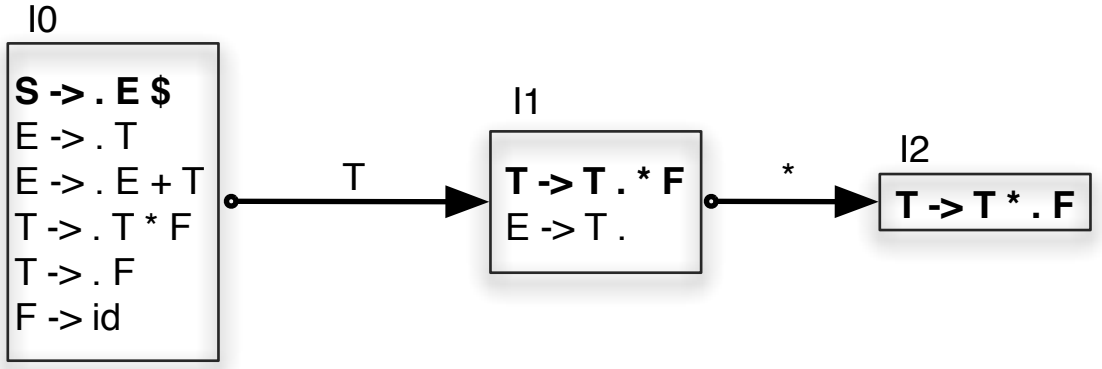
avec la construction SLR(1), ($\textsc{follow}(B) = \{c\}$ et $\textsc{follow}(E) = \{d\}$):

	c	d	f	x	y	$	A	B	E
I_2	R3	R4							

Analyse LR(k): illustration d'un conflit shift-reduce

Si dans un ensemble d'items, il en existe un de la forme A $\rightarrow \alpha \bullet$ et un de la forme B $\rightarrow \beta \bullet \gamma$ alors on est en présence d'un conflit shift-reduce.

E $\rightarrow$ E+T
E $\rightarrow$ T
T $\rightarrow$ T * F
T $\rightarrow$ F
F $\rightarrow$ id



Conflit Shift-2 / Reduce-E $\rightarrow$ T sur l'état I_1 en lecture de *. Avec une construction SLR(1) on applique la réduction seulement sur les éléments terminaux de FOLLOW(E) = {+,$}:

	+	*	$	E	T	F	S
I_1	R2	S2	R2				

CYK: un parseur "universel"

Cocke [1965], Younger [1967], Kasami [1965]

Hypothèse: les grammaires analysées sont exprimées en forme normale de Chomsky (CNF)[2]: $A \rightarrow BC|a$

Structure centrale: un tableau de booléens à trois dimensions dont chaque élément $\langle i, A, j \rangle$ est vrai ssi $A \stackrel{\star}{\Rightarrow} w_i \dots w_{j-1}$.

On suppose dans l'énoncé de l'algorithme qui suit qu'il y a N non terminaux et que la chaîne à analyser contient n mots: $w_1 \dots w_n$.

On cherche à vérifier si: $\langle 1, S, n+1 \rangle$ est vrai (analyse réussie) ou pas (analyse ratée).

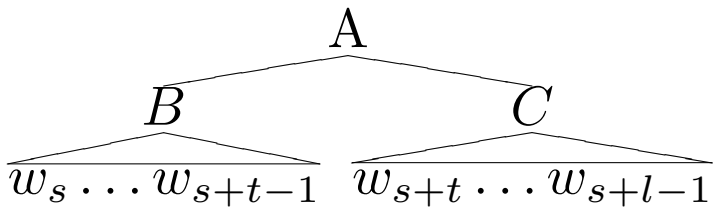
Note: C'est un exemple d'algorithme **bottom-up**.

[2]Toute grammaire CF peut-être exprimée par une grammaire CNF qui lui est faiblement équivalente: même langage engendré, dérivations possiblement différentes.

CYK: Idée générale

Définition: Un span est un ensemble de positions adjacentes dans la chaîne à analyser qui sont dominées par un symbole non-terminal ($A \stackrel{\star}{\Rightarrow} w_i \dots w_j$ est un span de longueur $j - i + 1$)

CYK considère tous les regroupements de spans adjacents en commencant par les plus petits:



Dans l'algorithme suivant, un span ($A \stackrel{\star}{\Rightarrow} w_i \dots w_j$) est dénoté par: $\langle i, A, j + 1 \rangle$

CYK[3]

// init
$\langle 1..n, 1..N, 1..n+1 \rangle \leftarrow$ false
// règles lexicales
for $s \leftarrow 1$ à n **do**
  for all rule $A \rightarrow w_s$ **do**
    $\langle s, A, s+1 \rangle \leftarrow$ true
// règles internes
for all length l, shortest (2) to longest (n) **do**
  for all valid start $s \in [1, n]$ **do**
    for all split length t **do**
      for all rule $A \rightarrow BC$ **do**

$$\langle s, A, s+l \rangle \leftarrow \langle s, A, s+l \rangle \vee (\langle s, B, s+t \rangle \wedge \langle s+t, C, s+l \rangle)$$

return $\langle 1, S, n+1 \rangle$

[3] Pour grammaire CNF. Je reprends ici l'algorithme tel que décrit dans Goodman [1998]

CYK: Exemple

S $\rightarrow$ NP VP	Det $\rightarrow$ a
NP $\rightarrow$ Det N	N $\rightarrow$ circle | square | triangle
VP $\rightarrow$ VT NP	VT $\rightarrow$ touches
VP $\rightarrow$ VI PP	VI $\rightarrow$ is
PP $\rightarrow$ P NP	P $\rightarrow$ above | below

Et la chaîne: *a circle touches a triangle*

1. pour $l = 1$: initialiser à vrai les cinq cellules:
   $\langle 1, Det, 2\rangle, \langle 2, N, 3\rangle, \langle 3, VT, 4\rangle, \langle 4, Det, 5\rangle, \langle 5, N, 6\rangle$
2. pour $l = 2$: $\langle 1, NP, 3\rangle, \langle 4, NP, 6\rangle$.
3. Pour $l = 3$: $\langle 3, VP, 6\rangle$
4. Pour $l = 4$: $\langle 1, S, 6\rangle$
   $\leftarrow$ analyse réussie

CYK: Table d'analyse

On peut ranger les spans dans une table d'analyse où la case (i, j) contient le non-terminal A ssi $A \stackrel{\star}{\Rightarrow} w_i \dots w_j$

triangle	S		VP	NP	N
a				Det	
touches			VT		
circle	NP	N			
a	Det				
	a	circle	touches	a	triangle

Earley: un autre algorithme universel Earley [1968]

- un item $[i, A \rightarrow \alpha \bullet \beta, j]$, avec la sémantique suivante:
  - la position courante dans la chaîne d'entrée est i: $w_1 \dots w_{i-1}$ a déjà été analysé.
  - la règle $A \rightarrow \alpha\beta$ est considérée. Elle a été appliquée en position j et $\alpha \stackrel{\star}{\Rightarrow} w_j \dots w_{i-1}$.

- un état S_i qui constitue l'ensemble des tentatives viables pour une position de l'entrée donnée $i \in [0, n]$:

$$S_i = \{[i, A \rightarrow \alpha \bullet \beta, j], \forall A, \alpha, \beta, j\}$$

Idée: Construire les états S_i. L'analyse est positive si S_n contient l'item $[n, S' \rightarrow S\bullet, 0]$, négative sinon. S' est un non terminal ajouté à la grammaire (nouvel axiome).

Earley: Trois opérations

prédiction: pour tout item $[i, X \to \lambda \bullet Y\mu, k]$ et pour toute règle $Y \to \alpha$
ajouter $[i, Y \to \bullet\alpha, i]$

lecture: pour tout item $[i, X \to \lambda \bullet a\mu, k]$ où a est le terminal w_i
ajouter $[i+1, X \to \lambda a \bullet \mu, k]$

complétion: pour tout item $[i, Y \to \alpha\bullet, j]$ et pour tout item qui a Y à droite du $\bullet$ dans l'état j $(\leq i)$: $[j, X \to \lambda \bullet Y\mu, k]$
ajouter $[i, X \to \lambda Y \bullet \mu, k]$

Départ: $[0, S' \to \bullet S, 0]$

But: $[n, S' \to S\bullet, 0]$

Earley: Exemple (input: *a circle touches a triangle*)

a	circle	touches	a
start:	lecture:	lecture:	lecture:
$[S' \rightarrow \bullet S, 0]$	[Det $\rightarrow a\bullet, 0]$	$[N \rightarrow$ circle$\bullet, 1]$	$[VT \rightarrow$ touches$\bullet, 2]$
prédiction:	complétion:	complétion:	complétion:
$[S \rightarrow \bullet$NP VP$, 0]$	$[NP \rightarrow$ Det $\bullet\ N, 0]$	$[NP \rightarrow$ Det N$\bullet, 0]$	$[VP \rightarrow VT \bullet NP, 2]$
[NP $\rightarrow \bullet$Det N$, 0]$	prédiction:	$[S \rightarrow NP \bullet VP, 0]$	prédiction:
[Det $\rightarrow \bullet$a$, 0]$	$[N \rightarrow \bullet$circle$, 1]$	prédiction:	$[NP \rightarrow \bullet$Det N$, 3]$
	$[N \rightarrow \bullet$square$, 1]$	$[VP \rightarrow \bullet$VT NP$, 2]$	[Det $\rightarrow \bullet a, 3]$
	$[N \rightarrow \bullet$triangle$, 1]$	$[VP \rightarrow \bullet$VI PP$, 2]$	
		$[VT \rightarrow \bullet$touches$, 2]$	
		$[VI \rightarrow \bullet$is$, 2]$	
S_0	S_1	S_2	S_3

triangle	
lecture:	lecture:
[Det $\rightarrow a\bullet, 3]$	$[N \rightarrow$ triangle$\bullet, 4]$
complétion:	complétion:
$[NP \rightarrow$ Det $\bullet\ N, 3]$	$[NP \rightarrow$ Det N$\bullet, 3]$
prédiction:	$[VP \rightarrow$ VT NP$\bullet, 2]$
$[N \rightarrow \bullet$circle$, 4]$	$[S \rightarrow$ NP VP$\bullet, 0]$
$[N \rightarrow \bullet$square$, 4]$	$[S' \rightarrow S\bullet, 0]$
$[N \rightarrow \bullet$triangle$, 4]$	
S_4	S_5

S $\rightarrow$ NP VP	Det $\rightarrow$ a
NP $\rightarrow$ Det N	VT $\rightarrow$ touches
VP $\rightarrow$ VT NP	VI $\rightarrow$ is
VP $\rightarrow$ VI PP	N $\rightarrow$ circle
PP $\rightarrow$ P NP	N $\rightarrow$ square
P $\rightarrow$ above	N $\rightarrow$ triangle
P $\rightarrow$ below	

Références

Eugene Charniak. *Statistical Language Learning*. MIT Press, 1993.

Cocke. peu probable que vous mettiez la main dessus(document à diffusion très restreinte), 1965.

J. Earley. *An efficient context-free parsing grammar*. PhD thesis, Carnegie-Mellon University, 1968.

Joshua Goodman. *Parsing inside-out*. PhD thesis, Harvard University, 1998.

Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Prentice Hall, 2000.

J. Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Technical report, University of Hawaii, 1965.

D.H. Younger. Recognition and parsing of context-free grammars in time n^3. *Information and Control*, 10:189–208, 1967.