Analyse Syntaxique ascendante

- Définition d'analyse ascendante LR
- Définition de poignée et préfixe viable
- Structure d'un analyseur ascendant
- Définition et construction des Items LR(k)
- Définition et construction des tables LR(k)
- Un exemple LR(0)
- Un exemple LR(1)
- Construction des tables SLR
- Construction des tables LALR(1)

Analyse ascendante

Cette méthode cherche a construire une dérivation droite en la reconstituant à l'envers à partir de la chaîne de terminaux vers le symbole initial, ou de façon équivalente, cherche à construire un arbre de dérivation a partir des feuilles selon un parcours inverse d'un parcours en profondeur d'abord de gauche a droite.

Pour savoir simplement quand l'analyse s' arrête avec succès pour une grammaire G, on travaille toujours sur une grammaire dite *augmentés* G' qui est G avec un nouveau symbole S' comme symbole de départ, un nouveau symbole terminal \$ et une transition supplémentaire

$$S' \quad \to \quad S\$$$

Terminologie

Notation 1.1 *Dans ce qui suit, on utilisera les conventions suivantes:*

des lettres majuscules $A, \ldots Z$ *indiquent des symboles non-terminaux*

des lettres grecques $\alpha, \beta, \gamma, \ldots$ *indiquent des* séquences *de symboles terminaux ou non-terminaux*

des lettres minuscules $a, b, c, \ldots$ *indiquent des symboles terminaux*

des lettres minuscules $\ldots, u, v, x, y, w, z$ *indiquent des* séquences *de symboles terminaux*

Définition 1.2 *Analyseurs LR Les analyseurs ascendants le plus connus sont dans la classe* LR *des analyseurs qui lisent le flot de tokens en entrée* de gauche a droite *(le* L *dans* LR*) pour reconstruire une dérivation* droite *(le* R *dans* LR*).*

Définitions techniques

Définition 1.3 (Dérivation droite) *Une dérivation droite est une dérivation qui remplace à chaque étape le symbole non terminal le plus à droite.*
On notera $\alpha \Rightarrow_d \beta$ *une étape de dérivation droite entre* α *et* β*, et on notera* $\alpha \Rightarrow_d^* \beta$ *une série (même vide) d'étapes de dérivation droite entre* α *et* β*.*

Définition 1.4 (Protophrase d'une grammaire G**)** *Une protophrase est une séquence de symboles terminaux et non terminaux qui peut apparaître en cours d'une dérivation du symbole initial* S *d'une grammaire* G*.*
On parle de protophrase droite *(resp.* gauche*) lorsque cette séquence peut apparaître dans une dérivation droite (resp. gauche) de* G*.*

Définitions techniques, suite

Définition 1.5 (Poignée (handle)) *Dans une protophrase ϕ, la séquence γ est une* poignée *á la position n pour la grammaire G si ell'est la partie gauche d'une production $X \rightarrow \gamma$, et que cette production doit être appliquée à ϕ en position n pour construire la protophrase précédente dans une dérivation droite á partir de S vers ϕ avec la grammaire G.*

Définition 1.6 (Préfixe viable) *Une séquence γ est un* préfixe viable *pour une grammaire G si γ est un préfixe de $\alpha\beta$, où $\phi = \alpha\beta w$ est une protophrase droite de G est β est une poignée dans cette protophrase.*
Autrement dit, un préfixe viable est un préfixe γ d'une protophrase ϕ, mais qui ne s' étend pas plus à droite d'une poignée β de ϕ.

Un exemple

Sur la grammaire augmentée

$$\begin{array}{lcl} S & \rightarrow & E\ \$ \\ E & \rightarrow & E + T \mid T \\ T & \rightarrow & id \end{array}$$

l'unique dérivation droite pour $id + id + id\ \$$ est la suivante:

$$\begin{array}{ll} \overline{S} & S \\ \overline{E}\ \$ & \underline{E\ \$} \\ E + \overline{T}\ \$ & \underline{E + T}\ \$ \\ \overline{E} + id\ \$ & E + \underline{id}\ \$ \\ E + \overline{T} + id\ \$ & \underline{E + T} + id\ \$ \\ \overline{E} + id + id\ \$ & E + \underline{id} + id\ \$ \\ \overline{T} + id + id\ \$ & \underline{T} + id + id\ \$ \\ id + id + id\ \$ & \underline{id} + id + id\ \$ \end{array}$$

Chaque ligne est une protophrase droite, en surligné les symboles produits, et en souligne les poignées.
Les préfixes viables sont les préfixes qui ne s'étendent pas plus loin qu'une poignée. Par exemple, sur la protophrase $E + \underline{id} + id\ \$$ ce sont ϵ, E, $E+$, $E + id$.

Définitions techniques, fin

Définition 1.7 **($FIRST_k(\gamma)$)** *Étant donnée une grammaire G, l'ensemble $FIRST_k(\gamma)$ contient les préfixes de longueur k des séquences de non terminaux de longueur au moins k dérivables à partir de γ dans G,* et *les séquences de non terminaux de longueur inférieur à k dérivables depuis γ.*

Définition 1.8 **($EFF_k(\gamma)$ ($\epsilon - free\ FIRST_k$))** *Étant donnée une grammaire G, EFF_k est le sousensemble de $FIRST_k$ obtenu en considérant seulement les dérivations qui ne réduisent pas sur ϵ un non terminal en tête de chaîne.*

Exemples

Pour la grammaire:

$$\begin{array}{lcl} S & \rightarrow & A\ B \\ A & \rightarrow & B\ a \mid \epsilon \\ B & \rightarrow & C\ b \mid C \\ C & \rightarrow & c \mid \epsilon \end{array}$$

on a:

$FIRST_2(S) = \{\epsilon, a, b, c, ab, ac, ba, ca, cb\}$

$EFF_2(S) = \{ca, cb\}$

Grammaires LR(k)

On peut maintenant donner la définition formelle de la classe des grammaires $LR(k)$.

Définition 1.9 (Grammaire LR(k)) *Une grammaire G est dans LR(k) (k* ≥ 0*) si les trois conditions suivantes:*

- $S \Rightarrow_d^* \alpha A w \Rightarrow_d \alpha\beta w$
- $S \Rightarrow_d^* \gamma B x \Rightarrow_d \alpha\beta y$
- $FIRST_k(w) = FIRST_k(y)$

impliquent $\alpha = \gamma, A = B, x = y$

Autrement dit, il suffit de regarder les premiers k symboles en entrée au moment ou il faut choisir une production pour la réduction dans la dérivation droite.

Structure de l'analyseur

Les grammaires $LR(k)$ sont celles dont le langage est reconnu par un analyseur déterministe $LR(k)$. Cet analyseur utilise une pile et le flot d'entrée, qui décrivent une *configuration* de l'analyseur, notée

$$(X_1 \dots X_j \qquad , \qquad a_i \dots a_n)$$

où les X sont de symboles, terminaux ou non terminaux, stoqués sur la pile, alors que les a sont seulement des symboles terminaux, et correspondent aux terminaux non encore lus sur le flot d'entrée.

L'analyseur travaille en effectuant quatre actions possibles:

shift (décalage) on transfert le terminal a_i du flot d'entrée vers la pile

reduce (réduction) on reconnaît sur le sommet de la pile une partie droite d'une production $Y \to X_{j-k} \dots X_j$, on l'enlève et on la remplace par sa partie gauche Y

erreur l'analyseur s'arrête et signale un erreur

accept l'analyseur s'arrête et signale que la phrase a été reconnue

Pour choisir les actions, on utilise une table d'analyse que l'on verra plus avant.

Un exemple

Sur la grammaire augmentée

$$\begin{array}{lcl} S & \rightarrow & E\ \$ \\ E & \rightarrow & E + T \mid T \\ T & \rightarrow & id \end{array}$$

Une possible séquence de reconnaissance pour $id + id + id\ \$$ pour un analyseur ascendant serait:

$$\begin{array}{llrl}
(& , & id + id + id\ \$) & shift \\
(\underline{id} & , & +id + id\ \$) & reduce \\
(\underline{T} & , & +id + id\ \$) & reduce \\
(E & , & +id + id\ \$) & shift \\
(E+ & , & id + id\ \$) & shift \\
(E + \underline{id} & , & +id\ \$) & reduce \\
(\underline{E + T} & , & +id\ \$) & reduce \\
(E & , & +id\ \$) & shift \\
(E+ & , & id\ \$) & shift \\
(E + \underline{id} & , & \$) & reduce \\
(\underline{E + T} & , & \$) & reduce \\
(E & , & \$) & accept
\end{array}$$

Remarques Importantes

Configurations et protophrases: la concaténation de la partie gauche et droite d'une configuration d'un analyseur ascendant pour une grammaire G est toujours une protophrase *droite* de G (si l'analyse se termine avec succès).

Préfixes viables: un préfixe viable peut toujours se compléter en une protophrase droite. En d'autre termes, il n'y a pas d'erreur au cours de l'analyse tant que l'on a sur la pile un préfixe viable.

Un autre exemple, avec look-ahead

Sur la grammaire augmentée

$$\begin{array}{lcc} S & \rightarrow & E\,\$ \\ E & \rightarrow & T + E \mid T \\ T & id & \end{array}$$

Une possible séquence de reconnaissance pour $id + id + id\,\$$ pour un analyseur ascendant serait:

$$\begin{array}{llrl}
(& , & id + id + id\,\$) & shift \\
(\underline{id} & , & +id + id\,\$) & reduce \\
(T & , & +id + id\,\$) & shift \\
(T+ & , & id + id\,\$) & shift \\
(T + \underline{id} & , & +id\,\$) & reduce \\
(T + T & , & +id\,\$) & shift(*) \\
(T + T+ & , & id\,\$) & shift \\
(T + T + \underline{id} & , & \$) & reduce \\
(T + T + \underline{T} & , & \$) & reduce(*) \\
(T + \underline{T + E} & , & \$) & reduce \\
(\underline{T + E} & , & \$) & reduce \\
(E & , & \$) & accept
\end{array}$$

Analyseurs LR

Un analyseur LR est composé de

une pile et un flot d'entrée comme vu dans les exemples

une table d'analyse qui décrit un automate à états finis augmenté avec des actions á effectuer éventuellement sur la pile (shift, reduce, accept, error)

L'exécution de l'automate est censée décaler sur la pile des symboles jusqu'à atteindre une préfixe viable maximale (i.e. pas extensible à droite, i.e. contenant une poignée γ en fond à droite, i.e. en sommet de pile), puis réduire la poignée en la remplaçant avec la partie droite X de la production $X \rightarrow \gamma$ concernée.

Fonctionnement d'un analyseur LR

Sur un état d'analyseur $(\alpha \quad , \quad xw)$ le fonctionnement de l'analyseur LR est le suivant:

- exécuter l'automate à partir de l'état initial s_1 sur la pile α, ce qui nous laisse sur un état s_k
- exécuter l'action décrite dans la table d'analyse associée au symbole terminal x en entrée pour l'état s_k

  shift (noté s) déplacer le symbole d'entrée x sur la pile,

  reduce n (noté rn) sur le sommet de la pile il y a la partie gauche de la règle numéro n, disons $X \to \gamma$; dépiler γ et empiler X

  accept (noté a) arrêter avec succès

  error (noté par une case vide) arrêter sur erreur

- recommencer avec le nouvel état d'analyseur

Exemple d'exécution avec une table d'analyse LALR(1)

$$
\begin{array}{llcl@{\qquad}llcl}
0 & S & \rightarrow & E\,\$ & 2 & E & \rightarrow & T \\
1 & E & \rightarrow & T + E & 3 & T & \rightarrow & id
\end{array}
$$

	Action			Transition				
	id	$+$	$\$$	id	$+$	$\$$	E	T
1	s			5			2	3
2			a					
3		s	$r2$		4			
4	s			5			6	3
5		$r3$	$r3$					
6			$r1$					

$$
\begin{array}{llrl}
(_1 & , & id + id\,\$) & shift \\
(_1\underline{id}_5 & , & +id\,\$) & reduce \\
(_1T_3 & , & +id\,\$) & shift(*) \\
(_1T_3 +{}_4 & , & id\,\$) & shift \\
(_1T_3 +{}_4\underline{id}_5 & , & \$) & reduce \\
(_1T_3 +{}_4\underline{T}_3 & , & \$) & reduce(*) \\
(\underline{_1T_3 +{}_4E_6} & , & \$) & reduce \\
(_1E_2 & , & \$) & accept
\end{array}
$$

Ici on a marqué en bas les états de l'automate après lecture de chaque symbole sur la pile.

Analyseur avec états sur la pile

Si on garde les états sur la pile, en modifiant la notion de configuration pour que chaque symbole soit suivi par un état, on peut éviter de relire toute la pile à chaque fois: sur une configuration d'analyseur $(s_1 X_1 \ldots s_{k-1} X_{k-1} s_k \quad , \quad xw)$ le fonctionnement de l'analyseur LR est le suivant:

- exécuter l'action décrite dans la table d'analyse associée au symbole terminal x en entrée pour l'état s_k

  shift k déplacer le symbole d'entrée x sur la pile, et empiler l'état numéro k

  reduce n sur le sommet de la pile il y a la partie gauche de la règle numéro n, disons $X \to \gamma$; dépiler γ et tous les états associés, en découvrant l'état s'; empiler X et l'état s'' contenu dans la table á la ligne s', colonne X

  accept arrêter avec succès

  error arrêter sur erreur

- recommencer avec le nouvel état d'analyseur

Comment produire une table d'analyse?

Il faut savoir reconnaître les préfixes viables, et savoir déterminer quelles productions utiliser pour les réductions, éventuellement en utilisant k tokens en entrée pour aider dans la décision.

Pour reconnaître les préfixes viables, on définit d'abord

Définition 1.10 (ITEM LR(k)) *Un* ITEM LR(k) *pour une grammaire G est une production $X \quad \rightarrow \quad \gamma$ de G plus une position j dans γ et une séquence w de longueur $\leq k$. Cela est noté, si $\gamma = \alpha\beta$ avec j la longueur de α*

$$X \quad \rightarrow \quad \alpha \cdot \beta, w$$

sauf dans le cas LR(0) pour lequel on écrit simplement $X \quad \rightarrow \quad \alpha \cdot \beta$

L'intuition de $(A \ \rightarrow \ \alpha \cdot \beta, w)$ est que l'on a déjà vu en entrée le préfixe α d'une protophrase et que l'on attende sur l'entrée une séquence dérivable à partir de βw.

Reconnaître les préfixes viables: la fermeture

Si on a $(A \rightarrow \alpha \cdot X\beta, z)$, i.e. on a a déjà vu en entrée le préfixe α et on attende une séquence dérivable à partir de $X\beta z$, on est aussi en condition d'attendre une séquence dérivable depuis X, suivie d'une séquence dérivable depuis βz. C'est cela que capture la notion suivante de fermeture

Définition 1.11 (Fermeture (Closure) LR(k))

Fermeture(I) =
répéter tant que I grandit
pour tout item $(A \rightarrow \alpha \cdot X\beta, z)$ dans I
  pour toute production $X \rightarrow \gamma$
    pour tout $w \in FIRST\_k(\beta\ z)$
      $I \leftarrow I \cup \{(X \rightarrow \cdot\gamma, w)\}$
retourner I

Reconnaître les préfixes viables: GOTO

Définition 1.12 (GOTO) *Supposons d'avoir* $(A \rightarrow \alpha \cdot X\beta, z)$, *pour un symbole* terminal ou non terminal X*: on a donc déjà vu en entrée le préfixe* α *et on attende une séquence dérivable à partir de* $X\beta z$*. Si maintenant l'on reconnaît* X, *alors on a vu* αX *et on attende une séquence dérivable à partir de* βz*.*
C'est cela que capture la notion suivante de GOTO

Goto(I,X) =
$J \leftarrow \emptyset$
pour tout item $(A \rightarrow \alpha \cdot X\beta, z)$ dans I
$\quad J \leftarrow J \cup \{(X \rightarrow \alpha\ X \cdot \beta, z)\}$
retourner Fermeture(J)

L'automate qui reconnaît les préfixes viables

Soit G un grammaire augmentée, et soit $\mathcal{I}$ la collection $\{s_0; s_1; \ldots; s_k\}$ d'ensembles d'ITEMS LR(k) attaignables depuis la fermeture de l'item $s_0 = (S' \rightarrow \cdot S\$, \epsilon)$ par la fonction GOTO.
On peut alors construire l'automate à état fini suivant:

états $\{s_0; s_1; \ldots; s_k\}$ avec s_0 état initial et comme états finaux ceux qui contiennent au moins un ITEM LR(k) avec le point au fond à droite (i.e. de la forme $(X \rightarrow \gamma\cdot, w)$)

transitions on a une transition de l'état s_i vers l'état s_j sur le symbole X si $GOTO(s_i, X) = s_j$

La construction de la table LR(k)

Soit G un grammaire augmentée, pour laquelle on a construit l'automate.
La table d'analyse a une ligne par état et un colonne par séquence de symboles terminaux de longueur $\leq k$ (le look-ahead) et une colonne par symbole terminal et non-terminal, que l'on remplit de la façon suivante:
Pour tout état $s_i \in \mathcal{I}$,

- on met $reduce\ n$ dans la case s_i, u si $(A \rightarrow \beta\cdot, u) \in s_i$ et $A \rightarrow \beta$ est la production numéro $n \geq 1$
- on met $accept$ dans la case $s_i, \$$ si $(S' \rightarrow \beta\cdot, \$) \in s_i$
- on met $shift$ dans la case s_i, u si $(A \rightarrow \beta_1 \cdot \beta_2, v) \in s_i$ et $u \in EFF_k(\beta_2 v)$
- on laisse vide (i.e. on signale erreur) autrement

Theorem 1.13 (fondamental de l'analyse ascendante) *Si une grammaire G est LR(k), alors l'automate construit reconnaît les préfixes viables de G et tout état contenant un ITEM LR(k) de la forme $(X \rightarrow \gamma\cdot, w)$ ne contient pas un ITEM $(X' \rightarrow \gamma_1 \cdot \gamma_2, u)$ avec $w \in EFF_k(\beta_2 v)$. (en d'autre terme, on n'aura pas dans la table des entrées multiples décaler et réduire ou entre deux réductions).*

Comment l'analyseur peut-il choisir l'action à effectuer?

Un analyseur LR dispose de plus d'information qu'un analyseur LL pour déterminer la prochaine action.

Imaginons d'avoir en entrée une chaîne uvw, et d'avoir déjà lu u. Pour déterminer la production à appliquer

- un analyseur LL(k) connaît u et $\text{FIRST}_k(vw)$
- un analyseur LR(k) connaît uv (en effet, il connaît un préfixe viable γ obtenu à partir de uv) et $\text{FIRST}_k(w)$

Un exemple LR(0)

La grammaire $G\,1$ suivante est $LR(0)$

$$\begin{array}{llll@{\qquad}llll@{\qquad}llll} 0 & S' & \rightarrow & S\,\$ & 1 & S & \rightarrow & (L) & 3 & L & \rightarrow & S \\ & & & & 2 & S & \rightarrow & x & 4 & L & \rightarrow & L, S \end{array} \tag{1}$$

Voici la construction complète de la table LR(0) de $G\,1$

États et transitions (2,4,6,7,9 sont terminaux)

1. $\{(S' \rightarrow \cdot S\$), (S \rightarrow \cdot(L)), (S \rightarrow \cdot x)\}$ **goto**$(1, S) = 4$ **goto**$(1, () = 3$, **goto**$(1, x) = 2$
2. $\{(S \rightarrow x\cdot)\}$
3. $\{(S \rightarrow (\cdot L)), (L \rightarrow \cdot S), (L \rightarrow \cdot L, S), (L \rightarrow \cdot S)(S \rightarrow \cdot(L)), (S \rightarrow \cdot x)\}$
   goto$(3, () = 3$ **goto**$(3, x) = 2$ **goto**$(3, S) = 7$ **goto**$(3, L) = 5$
4. $\{(S' \rightarrow S \cdot \$)\}$
5. $\{(S \rightarrow (L\cdot)), (L \rightarrow L\cdot, S)\}$ **goto**$(5,)) = 6$ **goto**$(5, ,) = 8$
6. $\{(S \rightarrow (L)\cdot)\}$
7. $\{(L \rightarrow S\cdot)\}$
8. $\{(L \rightarrow L, \cdot S), (S \rightarrow \cdot(L)), (S \rightarrow \cdot x)\}$ **goto**$(8, x) = 2$ **goto**$(8, () = 3$ **goto**$(8, S) = 9$
9. $\{(L \rightarrow L, S\cdot)\}$

La table d'analyse LR(0)

Pour remplir la table LR(0), on écrit la table de transition de l'automate et on introduit les actions de décalage (s pour $shift$) comme décrit plus en haut.
Pour les réductions (rk pour $reduce$ *avec la production* k), n'ayant pas de look-ahead dans les états, on met rk dans toute la ligne action de l'état j si l'état j contient un ITEM LR(0) $X \to \gamma\cdot$, et que $X \to \gamma$ est la production numéro j.

	Action					*Transition*						
	(	)	x	,	\$	(	)	x	,	\$	S	L
1	s		s			3		2			4	
2	$r2$	$r2$	$r2$	$r2$	$r2$							
3	s		s			3		2			7	5
4					a							
5		s		s			6		8			
6	$r1$	$r1$	$r1$	$r1$	$r1$							
7	$r3$	$r3$	$r3$	$r3$	$r3$							
8	s		s			3		2			9	
9	$r4$	$r4$	$r4$	$r4$	$r4$							

La table d'analyse LR(0), versions compacte

Remarque Les générateurs d'analyseurs, comme Yacc, fusionnent les colonnes des actions et des transitions pour les terminaux: plutôt que d'avoir une case case $(1, x)$ qui contient s(hift) pour les actions, et une case $(1, x)$ qui contient 2 pour les transitions, on préfère avoir une seule case $(1, x)$ qui contient $s2$, pour "l'action est un shift et la transition est vers l'état 2".
Dans ce cas, on écrit souvent gk dans les colonnes transitions restantes (celles des non-terminaux). C'est une abréviation pour "goto k", plus lisible que juste k.

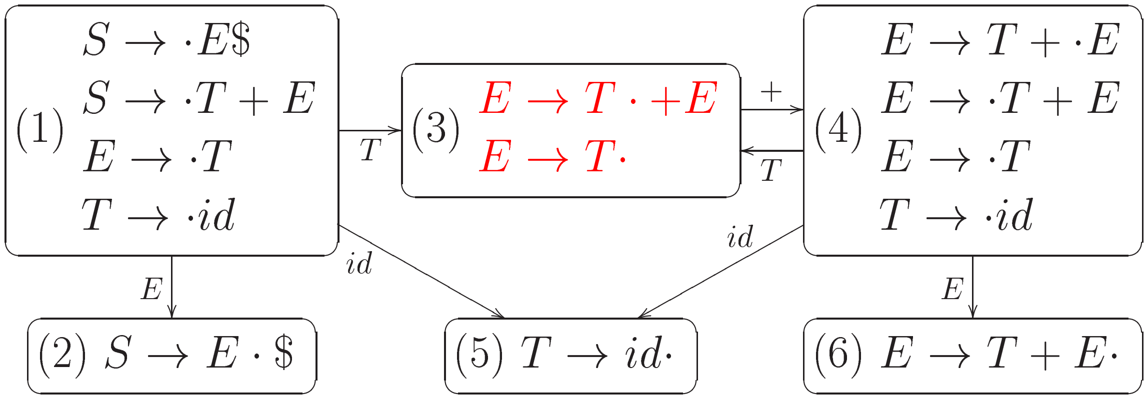
	Action					*Transition*	
	(	)	x	,	\$	S	L
1	$s3$		$s2$			$g4$	
2	$r2$	$r2$	$r2$	$r2$	$r2$		
3	$s3$		$s2$			$g7$	$g5$
4					a		
5		$s6$		$s8$			
6	$r1$	$r1$	$r1$	$r1$	$r1$		
7	$r3$	$r3$	$r3$	$r3$	$r3$		
8	$s3$		$s2$			$g9$	
9	$r4$	$r4$	$r4$	$r4$	$r4$		

Un exemple LR(1) non LR(0)

La grammaire $G\ 2$

$$\begin{array}{llcllllcl} 0 & S & \to & E\,\$ & \quad & 2 & E & \to & T \\ 1 & E & \to & T + E & & 3 & T & \to & id \end{array} \tag{2}$$

est une grammaire $LR(1)$ mais pas $LR(0)$.
En effet, dans l'automate on a un problème pour l'état $\{(E \to T \cdot +E), (E \to T\cdot)\}$.



Un exemple LR(1) non LR(0) (suite)

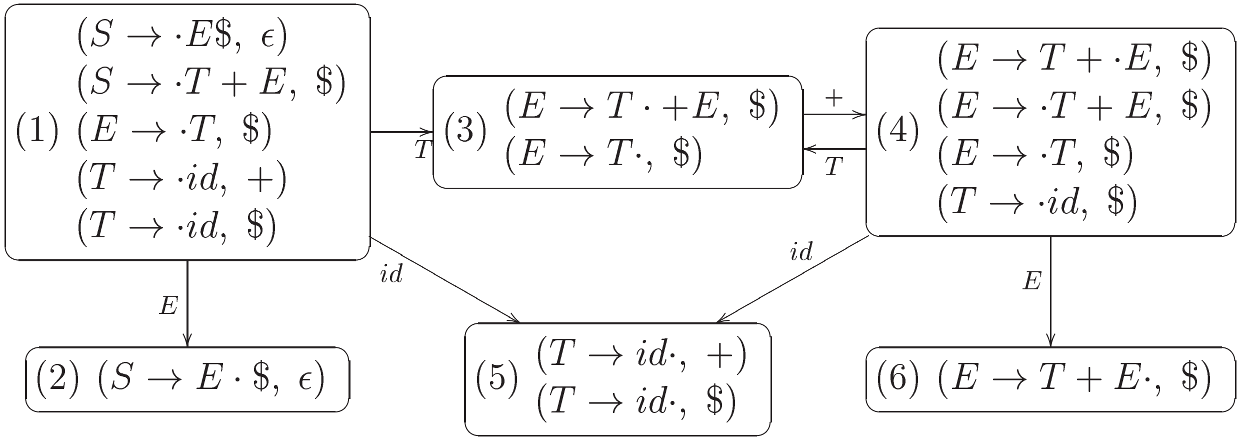
Donc dans la table d'analyse $LR(0)$ on trouve un conflit $shift/reduce$ dans la case $3, +$

	$Action$			$Transition$	
	$+$	id	$\$$	E	T
1		$s5$		$g2$	$g3$
2			a		
3	$\boxed{\text{r2,s4}}$	$r2$	$r2$		
4		$s7$		$g6$	$g3$
5	$r3$	$r3$	$r3$		
6	$r1$	$r1$	$r1$		

Les états LR(1) de G 2

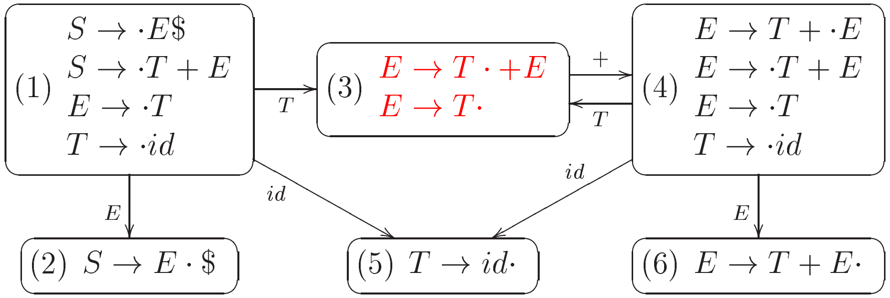
Regardons alors la construction LR(1), qui garde trace des look-aheads dans les états…

États et transitions (2, 5 et 6 sont terminaux)

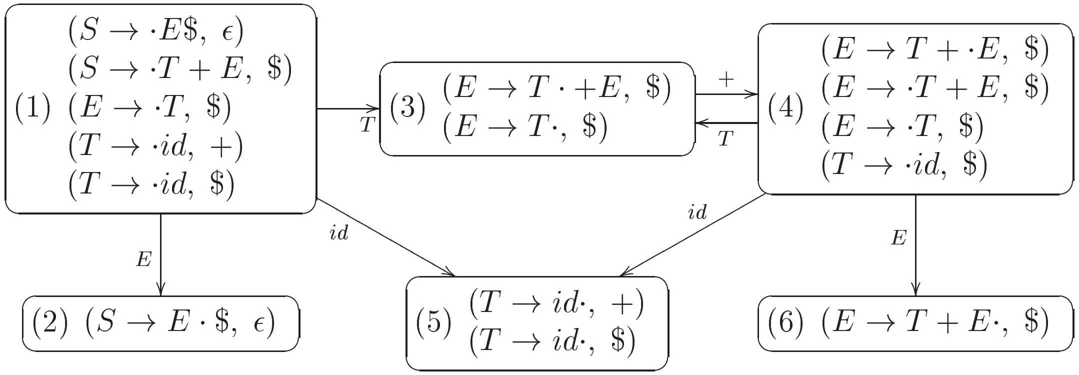


Comparons les automates LR(0) et LR(1) pour G 2

$LR(0)$



$LR(1)$



La table d'analyse LR(1) de $G\,2$

On garde trace des look-ahead pour introduire dans la table les actions *reduce*, donc il y en a moins et le conflit disparaît!

	Action			*Transition*				
	$+$	id	$\$$	$+$	id	$\$$	E	T
1		s			5		2	3
2			a					
3	s		$r2$	4				
4		s			5		6	3
5	$r3$		$r3$					
6			$r1$					

La table d'analyse LR(1) de G 2, version compacte

	Action			*Transition*	
	$+$	id	$\$$	E	T
1		$s5$		$g2$	$g3$
2			a		
3	$s4$		$r2$		
4		$s5$		$g6$	$g3$
5	$r3$		$r3$		
6			$r1$		

Comparons les tables LR(0) et LR(1) pour G 2

$LR(0)$

	Action			*Transition*	
	$+$	id	$\$$	E	T
1		$s5$		$g2$	$g3$
2			a		
3	$\boxed{\text{r2,s4}}$	$r2$	$r2$		
4		$s7$		$g6$	$g3$
5	$r3$	$r3$	$r3$		
6	$r1$	$r1$	$r1$		

SLR

	Action			*Transition*	
	$+$	id	$\$$	E	T
1		$s5$		$g2$	$g3$
2			a		
3	$s4$		$r2$		
4		$s5$		$g6$	$g3$
5	$r3$		$r3$		
6			$r1$		

Trouver sa place parmi les LR(k)

Comme nous venons de voir, la classe d'analyseurs LR(0) est trop faible pour traiter les langages de programmations: même le simple langage des expressions pose problème.

Les classes LR(2), LR(3), ... ont par contre une table d'analyse trop grosse en pratique en raison du nombre de colonnes pour le "look-ahead": un analyseur moderne utilise plusieurs dizaines de tokens, et une colonne pour chaque séquence de token de longueur inférieur ou égale à k, pour $k \geq 2$ est déraisonnable.

Exercice: combien de séquences de longueur inférieure ou égale à k y-at-il si on se donne n tokens différents?

Trouver sa place entre LR(0) et LR(1)

Heureusement, la classe LR(1) est largement suffisante pour les langages modernes, et la table n'a qu'*une colonne* par token.
Mais là, c'est *le nombre d'états* qui grandit trop, en raisons de la présence de look-ahead dans les états qui départage des états qui sont très peu différents.

C'est pour cela que dans la pratique on utilise deux types d'analyseurs dont la puissance est comprise entre celle de LR(0) et celle de LR(1): SLR et LALR(1)

Analyseurs SLR

SLR (Simple LR) est un analyseur dont l'automate est celui de LR(0), donc la partie transition est la même que LR(0), et les actions de décalage aussi, mais la table d'analyse est construite de une façon plus fine: on pallie à l'absence de look-ahead dans les état avec l'information contenue dans les ensembles FOLLOW construits à partir de la grammaire. La règle de placement des réductions devient alors:

> si l'état j contient un ITEM LR(0) $X \to \gamma \cdot$, et que $X \to \gamma$ est la production numéro $j \geq 1$, on met rk dans toutes les cases (j, t) telles que $t \in FOLLOW(X)$.

SLR pour la grammaire G 2

L'automate SLR étant le même que celui LR(0), on ne le montrera pas à nouveau, mais maintenant la table SLR contiendra des entrées *reduce* seulement sur certains nonterminaux, pas tous! En particulier, on pourra éviter le conflit $shift/reduce$ dans l'état $\{(E \to T \cdot +E), (E \to T \cdot)\}$.

	$Action$			$Transition$	
	$+$	id	$\$$	E	T
1		$s5$		$g2$	$g3$
2			a		
3	$s4$		$r2$		
4		$s5$		$g6$	$g3$
5	$r3$		$r3$		
6			$r1$		

Dans ce cas précis, SLR fait aussi bien que LR(1), avec bien moins d'effort.

Analyseurs LALR(1)

LALR(1) (*Look-Ahead LR(1)*) est une classe d'analyseurs dont l'automate est obtenu de l'automate LR(1) en *fusionnant* les états qui diffèrent seulement par leur look-ahead.

On dit aussi que l'on fusionne les états ayant le même *coeur*, le coeur d'un état étant l'ensemble des parties gauches des ITEMS LR(1) qu'il contient, i.e. sans le look-ahead, i.e. des ITEMS LR(0).
Donc un analyseur LALR(1) a autant d'états qu'un LR(0) ou SLR.

Les analyseurs LALR(1) sont les plus utilisés parce que, même s'ils ont moins d'états qu'un analyseur LR(1), il est très rare qu'on retrouve un conflit dans la table LALR(1) quand il n'y en a pas dans la table LR(1).
En particulier, on peut prouver que si un analyseur LR(1) n'a pas de conflits *shift/reduce*, l'analyseur LALR(1) n'en a pas non plus.
Par contre, on peut introduire des conflits *reduce/reduce*.

LALR(1) pour $G\ 2$

Dans le cas précis de cette grammaire, l'automate LR(1) pour $G\ 2$ n'ayant pas d'états différents avec le même coeur, la table d'analyse LALR(1) de $G\ 2$ est la même que celle LR(1).

> Mais la grammaire suivante, qui capture un sous-ensemble des expressions du langage C, est un exemple de grammaire LALR(1) qui n'est pas SLR et pour laquelle l'automate LALR(1) est plus petit que l'automate LR(1).

$$\begin{array}{llcllllcl} 0 & S' & \rightarrow & S\,\$ & & 3 & E & \rightarrow & V \\ 1 & S & \rightarrow & V = E & & 4 & V & \rightarrow & id \\ 2 & S & \rightarrow & E & & 5 & V & \rightarrow & *E \end{array} \tag{3}$$

LR préfère l'associativité à gauche

Contrairement à ce qui se passe dans le cas des analyseurs LL, dans l'analyse ascendante on a plutôt intérêt à utiliser des grammaires récursives à gauche.

Considérons les analyseurs LR pour la grammaire récursive à droite

$$\begin{array}{ccccccc} S & \rightarrow & E\,\$ & \qquad & E & \rightarrow & T \\ E & \rightarrow & T + E & \qquad & T & \rightarrow & id \end{array}$$

vus en cours: pour reconnaître $id + id + \ldots + id\$$, ils empilent toute la suite de symboles (en réduisant id sur T à chaque coup) avant de faire la prémière reduction non triviale.
Par contre, la grammaire recursive à gauche

$$\begin{array}{ccccccc} S & \rightarrow & E\,\$ & \qquad & E & \rightarrow & T \\ E & \rightarrow & E + T & \qquad & T & \rightarrow & id \end{array}$$

mantiendra la dimension de la pile à un minimum.

Utilisation de grammaires ambiguës

Une grammaire ambiguë n'est jamais LR(k), quelque soit k.
Pourtant, on a intérêt à essayer d'utiliser une grammaire ambiguë, quitte à trafiquer l'automate $LR(k)$, si on peut.

efficacité dans une grammaire obténue par désambiguation, l'analyseur passe beaucoup de temps à reduire des productions triviales (comme $E \rightarrow T$ dans l'exemple précédent), dont le seul but était d'expliciter *dans la grammaire* les priorités entre opérateurs et leur associativité droite ou gauche.

praticité si on peut décrire de façon concise ces priorités entre opérateurs et leur associativité droite ou gauche, sans toucher à la grammaire, on obtient une description plus modulaire du langage qui nous intéresse.

Exemple

Considérons la grammaire (ambiguë) suivante:

$$
\begin{array}{lcl}
S & \rightarrow & E\ \$ \\
E & \rightarrow & E * E \\
E & \rightarrow & E + E \\
E & \rightarrow & id
\end{array}
$$

...