Grammaires et Analyse Syntaxique - Cours 7
LR(0) et LR(1)

Ralf Treinen

INFORMATIQUE
Sciences
Université de Paris

INSTITUT
DE RECHERCHE
EN INFORMATIQUE
FONDAMENTALE

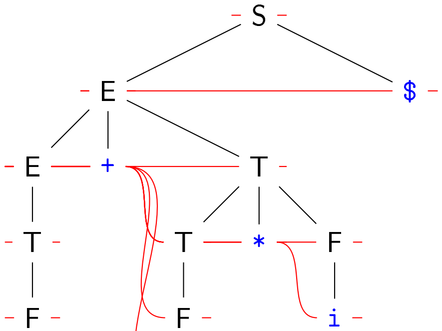
treinen@irif.fr

10 mars 2022

Exemple de l'exécution d'un parseur *shift/reduce*

$S \to E\ \$ \quad E \to E + T \mid T \quad T \to T * F \mid F \quad F \to (\ E\ ) \mid i$

Situation initiale Après *Shift* Après *Reduce* $F \to i$ Après *Reduce* $T \to F$ Après *Reduce* $E \to T$ Après *Shift* Après *Shift* Après *Reduce* $F \to i$ Après *Reduce* $T \to F$ Après *Shift* Après *Shift* Après *Reduce* $F \to i$ Après *Reduce* $T \to T * F$ Après *Reduce* $E \to E + T$ Après *Shift* Après *Reduce* $S \to E\ \$$ *Accept !*



Préfixes réductibles

Définition

$\alpha \in (\Sigma \cup N)^*$ est *préfixe réductible* ssi

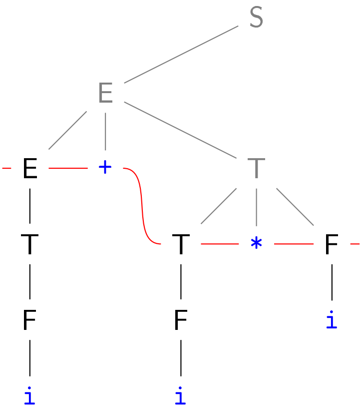
- il y a une décomposition $\alpha = \alpha_1\alpha_2$
- et une règle N $\rightarrow \alpha_2$
- et un mot $w \in \Sigma^*$ tel que
- $S \overset{d}{\rightarrow}^* \alpha_1 \mathrm{N} w \overset{d}{\rightarrow} \alpha_1\alpha_2 w = \alpha w$

- Seulement sur un préfixe réductible on peut appliquer une action **reduce**, avec l'espoir de pouvoir construire l'arbre jusqu'à l'axiome.
- Il faut faire attention qu'il est toujours possible de compléter le contenu de la pile (par des actions **shift**) et obtenir un préfixe réductible.

Comment reconnaître un préfixe réductible ?

- Pour reconnaître un préfixe réductible il faut en principe trouver le reste de l'arbre.
- Ce qui est "en dessous" des symboles de la pile n'importe pas car on l'a déjà consommé.
- On ne veut pas regarder tout le reste de l'entrée, on va donc assurer qu'il est possible de trouver un reste de l'entrée convenable.
- Reste à trouver la partie de l'arbre entre la racine et les symboles de la pile.

Exemple : préfixe réductible E+T∗F



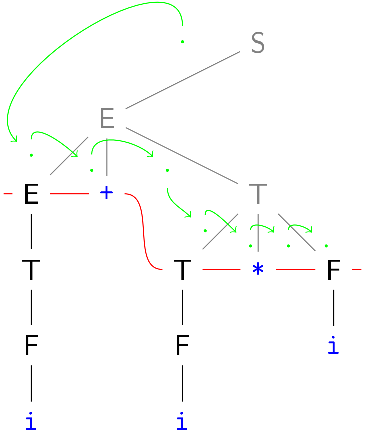
En gris la partie "à deviner".

Quelle est la structure de la partie “à deviner” ?

- C’est un morceau d’arbre qui commence par l’axiome S.
- Tous les nœuds à deviner sont étiquetés par des non terminaux, et leurs enfants constituent le côté droit d’une règle pour ce non-terminal.
- On peut dans l’arbre soit descendre vers un fils, soit passer au frère suivant.
- Quand on passe au frère suivant on vérifie que c’est justifié par un symbole qu’on trouve sur la pile.

Exemple : préfixe réductible E+T*F

$$S \to E\,\$ \quad E \to E + T \mid T \quad T \to T * F \mid F \quad F \to (\,E\,) \mid i$$



Un automate

- Il s'agit de l'exécution d'un automate !
- Les états sont les points verts - l'information dans un état est la production de la grammaire, et l'endroit où on est dans le côté droit de la règle.
- L'automate peut "deviner" quelque chose car il s'agit d'un automate non déterministe.
- L'automate peut descendre par une transition ϵ, ou passer d'un enfant au suivant par une transition étiquetée par un symbole de la pile.
- On appelle cet automate l'*automate caractéristique* de la grammaire. Nous allons le définir précisément dans la suite.

Les *items*

Définition

Soit $G = (\Sigma, N, S, P)$ une grammaire. Un *item* de G est une expression de la forme

$$[N \rightarrow \alpha.\beta]$$

où $N \rightarrow \alpha\beta \in P$.
Un item de la forme $[N \rightarrow \alpha.]$ est *complet*.

- C'est à dire, un item consiste en une règle de la grammaire, plus une position (indiqué par le symbole ".") dans le côté droit de la règle.
- α et β peuvent être ϵ.
- Une règle N $\rightarrow \epsilon$ donne lieu à un seul item : $[N \rightarrow .]$

Exemple

- Soit la grammaire E → (E+E) | i   S → E $
- Les items sont :
  [E → .(E+E)], [E → (.E+E)], [E → (E.+E)],
  [E → (E+.E)], [E → (E+E.)], [E → (E+E).],
  [E → .i], [E → i.],
  [S → .E $], [S → E.$], [S → E $.]
- Donc 11 états même pour une grammaire si simple.
- En général : Pour k règles de la grammaire avec longueurs des côtés droits $n_1, \ldots, n_k$, le nombre d'items est

$$\sum_{i=1\ldots k} (n_i + 1)$$

- Dans le cas de la grammaire pour les expressions arithmétiques avec priorités : 21 items.

L'automate caractéristique non-déterministe : états

- L'ensemble d'états : c'est l'ensemble des items.
- Les états initiaux sont

$$\{[\mathsf{S} \rightarrow .\alpha] \mid \mathsf{S} \rightarrow \alpha \in P\}$$

  où S est l'axiome de la grammaire.
- Sur l'exemple : l'état initial est $[\mathsf{S} \rightarrow .\mathsf{E}\ \$]$
- Les états acceptants sont les items complets :

$$\{[\mathsf{N} \rightarrow \alpha.] \mid \mathsf{N} \rightarrow \alpha \in P\}$$

- Sur l'exemple : les états acceptants sont :
  $[\mathsf{E} \rightarrow (\mathsf{E}+\mathsf{E}).]$, $[\mathsf{E} \rightarrow \mathtt{i}.]$, $[\mathsf{S} \rightarrow \mathsf{E}\ \$.]$

L'automate caractéristique non-déterministe : transitions

1. Alphabet : $\Sigma \cup N$
2. Premier type de transitions :

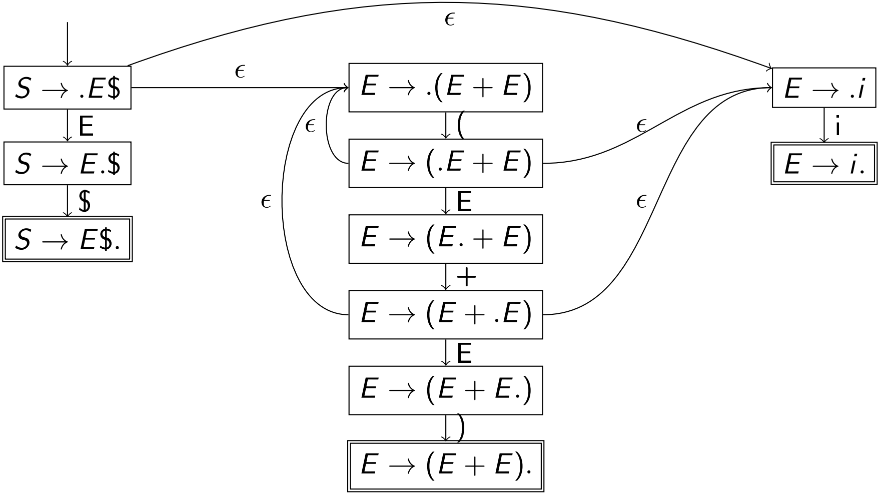
$$[\mathsf{N} \rightarrow \alpha.x\beta] \stackrel{x}{\Rightarrow} [\mathsf{N} \rightarrow \alpha x.\beta]$$

   où $\mathsf{N} \rightarrow \alpha x\beta \in P$, $x \in \Sigma \cup N$.
3. Deuxième type de transitions :

$$[\mathsf{N} \rightarrow \alpha.\mathsf{M}\beta] \stackrel{\epsilon}{\Rightarrow} [\mathsf{M} \rightarrow .\gamma]$$

   où $\mathsf{N} \rightarrow \alpha\mathsf{M}\beta, \mathsf{M} \rightarrow \gamma \in P$

L'automate caractéristique non-déterministe sur l'exemple



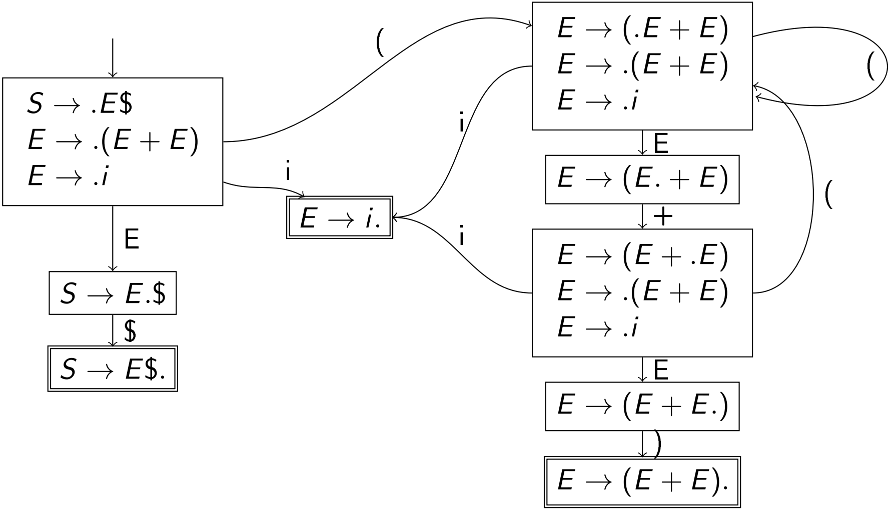
Rappel : Déterminiser et éliminer les transitions ϵ

- Soit $A = (\Sigma, Q, \delta, I, F)$ un automate non-déterministe avec ϵ-transitions.
- On définit d'abord pour $P \subseteq Q$:

$$\epsilon\text{-cloture}(P) = \{q \in Q \mid \exists p \in P, q \in \delta^*(p, \epsilon)\}$$

- On construit $A' = (\Sigma, 2^Q, \delta', I', F')$ avec
  - $I' = \epsilon\text{-cloture}(I)$
  - $\delta'(P, a) = \bigcup_{p \in P} \epsilon\text{-cloture}(\delta(p, a))$
  - $F' = \{P \subseteq Q \mid P \cap F \neq \emptyset\}$
- Il convient de construire les états de A' au fur et à mesure.

Déterminiser avec élimination des ϵ-transitions



Comment se servir de l'automate caractéristique ?

- On applique l'automate au mot sur la pile (la pile est lue du bas vers le haut).
- L'état du dernier élément de la pile peut nous dire quoi faire :
  - Si l'état contient un item complet (c.-à-d. de la forme $[N \rightarrow \alpha.]$) on peut appliquer une action **reduce** pour cette règle.
  - Si l'état contient un item non complet (c.-à-d. de la forme $[N \rightarrow \alpha.\beta]$ avec $\beta \neq \epsilon$) on peut appliquer une action **shift**.
- C'est une solution efficace qui évite de chercher en haut de la pile : il suffit de regarder l'état au sommet de la pile.

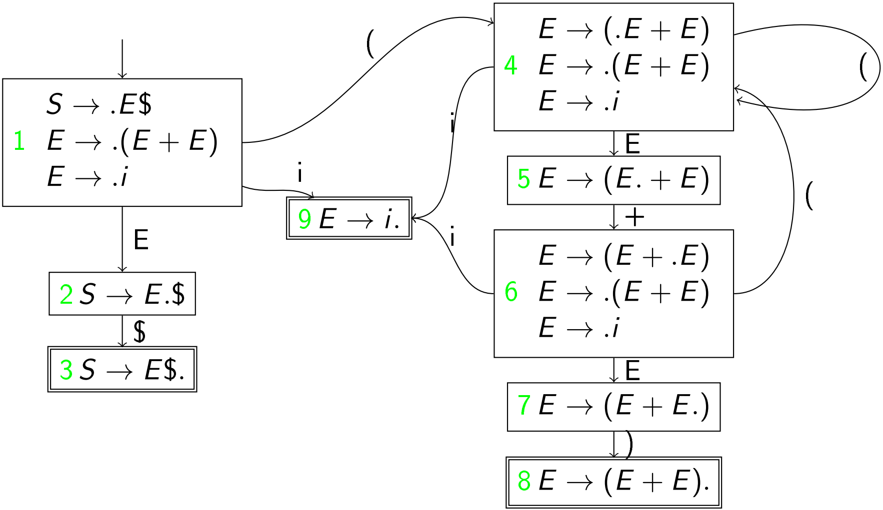
Comment se servir de l'automate caractéristique (2) ?

- Une action **reduce** remplace en haut de la pile une séquence α par un non-terminal N. Comment calculer son état ?
- On stocke sur la pile l'état avec *chaque* élément. Cela permet de calculer l'état de N, en faisant une transition de l'automate.
- On avait aussi dit (cours 6) qu'on doit toujours pouvoir compléter le contenu de la pile par des terminaux et obtenir un préfixe réductible. Comment est-ce assuré ?
- Notre automate est incomplet, et n'a pas d'états puits. Il suffit donc d'assurer pendant l'analyse que l'automate ne bloque pas.

Les grammaires **LR(0)** ?

- Dans l'automate caractéristique non-déterministe on a que chaque état contient exactement un item. Après déterminisation, un état peut contenir plusieurs items !
- Un état (ensemble d'items) a un *conflit shift-reduce* quand il contient à la fois un item complet et un item incomplet.
- Un état (ensemble d'items) a un *conflit reduce-reduce* quand il contient deux items complets différents.
- Une grammaire est dite *LR(0)* quand son automate caractéristique déterministe n'a pas de conflit (ni shift-reduce, ni reduce-reduce).

Sur l'exemple (avec les états numérotés en vert)



C'est donc bien une grammaire LR(0) !

Grammaire augmentée

- On appelle une grammaire (Σ, N, S, P) *augmentée* quand
  1. il y une seule règle avec côté gauche S,
  2. elle est de la forme $S \rightarrow$ E\$ pour un E $\in N$ et \$ $\in \Sigma$,
  3. S parait sur aucun côté droit d'une règle.
- On peut toujours transformer une grammaire non augmentée en une grammaire augmentée, en définissant un nouvel axiome.
- Les grammaires augmentées facilitent la description de l'algorithme LR(0), mais on pourrait en principe aussi se débrouiller avec des grammaires non augmentées.

L'algorithme d'analyse grammaticale LR(0)

- Initialement on met sur la pile l'état initial qui contient $[S \rightarrow .\mathrm{E}\$]$.
- Tant que le sommet de la pile n'est pas $\{[S \rightarrow \mathrm{E}\$.]\}$:
  - Si l'état sur le sommet de la pile
    - est de la forme $\{[N \rightarrow \alpha.]\}$: faire un **reduce** $\{[N \rightarrow \alpha.]\}$, mettre à jour l'état sur la pile ;
    - contient un item incomplet de la forme $[N \rightarrow \alpha.a\beta]$ et le symbole suivant est a : faire un **shift**, mettre à jour l'état sur la pile.
    - dans les autres cas **échec**.
- **accept**

Les cas d'échec

- Il y a des possibilités d'échec à deux niveaux différents :
  1. Notre construction de l'automate caractéristique peut nous amener à un conflit reduce-reduce ou shift-reduce. Dans ce cas la grammaire est rejetée car elle n'est pas LR(0).
  2. Quand notre grammaire est LR(0) l'analyse d'un texte d'entrée peut échouer, c'est le cas précisément quand l'entrée ne correspond pas à la grammaire.
- Ces deux possibilités existent aussi pour l'analyse LL(1).

Retour à l'exemple des expressions arithmétiques avec priorités

- Rappel :
  $S \to E\,\$ \quad E \to E + T \mid T \quad T \to T * F \mid F \quad F \to (\,E\,) \mid i$
- Est elle LR(0) ?
- Essayons de construire l'automate caractéristique déterministe !
- On commence par l'état initial et on ajoute les autres états au fur et à mesure, mais nous allons pour cet exemple nous arrêter au premier problème rencontré.

Construction de l'automate déterministe

$$S \to E\,\$ \quad E \to E + T \mid T \quad T \to T * F \mid F \quad F \to (\,E\,) \mid i$$

$$\begin{tikzcd}
{} \arrow[r] & \boxed{\begin{array}{l} S \to .E\$ \\ E \to .E{+}T \\ E \to .T \\ T \to .T{*}F \\ T \to .F \\ F \to .(E) \\ F \to .i \end{array}} \arrow[r, "T"] & \boxed{\boxed{\begin{array}{l} E \to T. \\ T \to T.{*}F \end{array}}}
\end{tikzcd}$$

- On peut déjà s'arrêter là car on a un *conflit shift-reduce* !
- Si on continue la construction on trouve un deuxième conflit shift-reduce.

Les conflits sur l'exemple

- S → E $ E → E + T | T T → T * F | F F → (E) | i
- On peut trouver les conflits aussi sur l'automate non-déterministe, les transitions de l'automate sont notés ⇒ :
  1. shift-reduce conflit obtenu par T :
     $[S \to .E\$] \stackrel{\epsilon}{\Rightarrow} [E \to .T] \stackrel{T}{\Rightarrow} [E \to T.]$ reduce
     $[S \to .E\$] \stackrel{\epsilon}{\Rightarrow} [E \to .T] \stackrel{\epsilon}{\Rightarrow} [T \to .T{*}F] \stackrel{T}{\Rightarrow} [T \to T. * F]$ shift
  2. shift-reduce conflit obtenu par $E + T$:
     $[S \to .E\$] \stackrel{\epsilon}{\Rightarrow} [E \to .E{+}T] \stackrel{E}{\Rightarrow} [E \to E.{+}T] \stackrel{+}{\Rightarrow}$
     $[E \to E{+}.T] \stackrel{T}{\Rightarrow} [E \to E{+}T.]$ reduce
     $[S \to .E\$] \stackrel{\epsilon}{\Rightarrow} [E \to .E{+}T] \stackrel{E}{\Rightarrow} [E \to E.{+}T] \stackrel{+}{\Rightarrow}$
     $[E \to E{+}.T] \stackrel{\epsilon}{\Rightarrow} [T \to .T{*}F] \stackrel{T}{\Rightarrow} [T \to T.{*}F]$ shift

Faire un *lookahead*

- La solution dans des cas comme l'exemple des expressions arithmétiques et de permettre un regard en avant (lookahead).
- Pour cela on va ajouter à un item $[N \rightarrow \alpha.\beta]$ un ensemble de symboles qui peuvent suivre à un mot produit à partir de $\alpha\beta$.
- On fait cela en utilisant l'information comment on est arrivé à cet item.

Les items LR(1)

- Soit $G = (\Sigma, N, S, P)$ une grammaire.
- Un item LR(1) est une expression de la forme

$$[K \to \alpha.\beta, L]$$

  où
  - $K \to \alpha\beta \in P$ est une règle de la grammaire
  - $L \subseteq \Sigma \cup \{\epsilon\}$
- $[K \to \alpha.\beta]$ est son *noyau*.
- L est son *lookahead*
- La longueur des lookahead est limité à 1, dont le nombre 1 dans “LR(1)”.

L'automate caractéristique non-déterministe LR(1)

- L'automate caractéristique LR(1) est construit très similaire à l'automate pour LR(0), la différence est seulement dans la gestion des lookahead.
- Dans l'automate déterministe on aura une définition de conflit plus fine que pour LR(0) car elle prend aussi en compte le lookahead.
- Ce raffinement de la détection de conflit est la raison pourquoi on passe des LR(0) aux LR(1).
- Nous allons décrire la version non déterministe de cet automate, mais sur l'exemple nous allons construire tout de suite la version déterministe.

L'automate caractéristique non-déterministe LR(1)

- Grammaire augmentée (Σ, N, S, P)
- L'ensemble des états est l'ensemble des items LR(1).
- Les états initiaux sont les items $[S \to .\alpha, \{\epsilon\}]$
- Les états acceptants sont les items LR(1) avec un noyau complet, c-a-d de la forme $[K \to \alpha., L]$
- Transitions par un symbole x $[K \to \alpha.x\beta, L] \overset{x}{\Rightarrow} [K \to \alpha x.\beta, L]$
- Transitions ϵ : $[K \to \alpha.N\beta, L] \overset{\epsilon}{\Rightarrow} [N \to .\gamma, \mathrm{First}_1(\beta L)]$ quand $N \to \gamma \in P$, où $\beta L = \{\beta w \mid w \in L\}$.
- $\mathrm{First}_1(\beta L)$ est trivial à calculer quand β commence sur un symbole terminal. L est seulement pris en compte quand β peut produire ϵ.
- Pour chaque item $[N \to \alpha.\beta, L]$ on a que $L \subseteq \mathrm{Follow}_1(\mathrm{N})$.

Exemple d'un automate LR(1)

- On va construire au tableau (au moins le début de) l'automate LR(1) pour la grammaire des expressions arithmétiques :

  $$S \to E \,\$ \quad E \to E + T \mid T \quad T \to T * F \mid F \quad F \to (E) \mid i$$

- Nous avons vu que cette grammaire n'est pas LR(0), à cause de deux conflits shift-reduce, dans les états suivants de l'automate LR(0) :
  1. $\{[E \to T.], [T \to T.*F]\}$
  2. $\{[E \to E+T.], [T \to T.*F]\}$

La taille de l'automate

- $S \rightarrow E\ \$ \quad E \rightarrow E + T \mid T \quad T \rightarrow T * F \mid F \quad F \rightarrow (E) \mid i$
- Il y 21 items LR(0), 6 symboles terminaux, ça fait
  - $2^6 = 64$ possibilités pour L,
  - en principe $21 * 64 = 1344$ états dans l'automate LR(1) non déterministe,
  - en principe 2^{1344} états dans l'automate LR(1) déterministe.
- On va donc plutôt pas dessiner cet automate avec tous les états, mais on va construire tout de suite l'automate déterministe qui contient seulement les états accessibles à partir de son état initial.

Quel est l'état initial ?

- S → E `$`   E → E `+` T | T   T → T `*` F | F   F → `(` E `)` | `i`
- État initial de l'automate non déterministe : $[S \rightarrow .E\$, \{\epsilon\}]$
- Quels états peut on atteindre à partir de là par des ϵ-transitions ?
- $[E \rightarrow .E{+}T, \{\$\}]$ car $\mathrm{First}_1(\$\{\epsilon\}) = \{\$\}$
- $[E \rightarrow .E{+}T, \{+\}]$ car $\mathrm{First}_1(+\{\$\}) = \{+\}$
- Dans l'automate déterministe on regroupe ces deux en $[E \rightarrow .E{+}T, \{\$, +\}]$.
- Et on continue ...

Conflits reduce-reduce dans le cas LR(1)

- Un ensemble d'items LR(1) a un **conflit reduce-reduce** quand il contient deux items complets

$$[N_1 \to \alpha_1., L_1]$$
$$[N_2 \to \alpha_2., L_2]$$

avec $L_1 \cap L_2 \neq \emptyset$

- Quand il n'y a pas de conflit reduce-reduce et on atteint un état avec des items complets :

$$[N_1 \to \alpha_1., L_1]$$
$$\ldots$$
$$[N_i \to \alpha_i., L_i]$$
$$\ldots$$

alors on fait une action reduce $N_i \to \alpha_i$ quand le symbole suivant de l'entrée appartient à L_i.

Conflits shift-reduce dans le cas LR(1)

- Un ensemble d'items LR(1) a un **conflit shift-reduce** quand il contient deux items

$$[N_1 \to \alpha_1., L_1]$$
$$[N_2 \to \alpha_2.c\beta_2, L_2]$$

avec $c \in L_1$.

- Quand il n'y a pas de conflit shift-reduce et on est dans un état qui contient un item

$$[N_2 \to \alpha_2.c\beta_2, L_2]$$

et le symbole suivant de l'entrée est c alors on fait un shift.

Les grammaires **LR(1)**

- Une grammaire est LR(1) quand son automate LR(1) déterministe n'a pas de conflits (ni shift-reduce, ni reduce-reduce)

Sur l'exemple

- Dans notre exemple on a obtenu pour LR(0) des conflits shift-reduce dans ces deux états :

$$\{[E \to T.], [T \to T. * F]\}$$
$$\{[E \to E + T.], [T \to T. * F]\}$$

- Quand on refait l'automate pour LR(1) on obtient les états

$$\{[E \to T., \{\$, +\}], [T \to T. * F, \{\$, +, *\}]\}$$
$$\{[E \to E + T., \{\$, +\}], [T \to T. * F, \{\$, +, *\}]\}$$

  et il n'y a pas de conflits LR(1) car $* \notin \{\$, +\}$.

- Cette grammaire est donc LR(1) mais elle n'est pas LR(0).

Représentation habituelle

- Souvent on représente l'information contenue dans l'automate sous forme de deux tables : *table d'action* et *table de transition* (angl : goto table).
- Il s'agit seulement d'une représentation plus commode pour le code de l'analyseur grammaticale. Toute information utile est déjà présente dans l'automate.
- Table de transitions : chaque ligne correspond à un état de l'automate déterministe, chaque colonne à un non terminal.
- Les entrées dans cette table disent simplement vers quel nouvel état il faut aller à partir de tel état, et en lisant tel symbole non-terminal. C'est la fonction de transition restreinte aux non-terminaux.
- Cette table est consultée après un reduce, pour déterminer l'état pour le non-terminal qu'on a mis sur la pile.

La table d'action

- Chaque ligne correspond à un état, P chaque colonne à un symbole terminal, a.
- Si P contient $[S \to \alpha.\$]$ et $a = \$$: entrée *accept*
- Sinon, si P contient un item $[N \to \alpha.a\beta, L]$: entrée dans la table *shift* Q, où $\delta(P, a) = Q$.
- Sinon, si P contient un item $[N \to \alpha., L]$ avec $a \in L$: entrée dans la table *reduce* $N \to \alpha$.
- Sinon : entrée dans la table *error*.
- Voir l'exemple complèt donné au tableau de la grammaire suivante : $\mathrm{S} \to \mathrm{N\$} \quad \mathrm{N} \to \mathrm{aNb} \mid \epsilon$

L'algorithme d'analyse LR(1)

- On suppose les opérations suivantes sur la pile :
  - `push(n)` met *n* sur la pile
  - `pop()` supprime le sommet de la pile
  - `top()` donne le sommet de la pile (sans de modifier la pile)
- On suppose les opérations suivantes sur l'entrée :
  - `lookahead()` donne le symbole suivant (sans de le consommer)
  - `eat()` consomme un symbole de l'entrée

L'algorithme d'analyse LR(1)

- `etat-initial` : état qui contient $[S \rightarrow .\alpha\$, \{\epsilon\}]$.
- Utilise des tables `action` et `goto`.
- ```
  push (etat-initial);
  while true do
    match action(top(),lookahead()) with
    | shift(n) -> eat(); push(n);
    | reduce (N->w) -> for i in 1..length(w)
                            do pop();
                         push(goto(top(),N))
    | accept -> exit success
    | error -> exit error
  ```
  ```

Autres approches

- On trouve dans la littérature (en particulier quand elle date un peu) souvent des autres approches qui sont plus fortes que LR(0) mais plus faibles que LR(1) : SLR(1) et LALR(1).
- Ces approches ont l'avantage que leurs ensembles d'états sont plus petits que l'ensemble d'états de la construction LR(1) car il y a une analyse moins fine des lookahead.
- Aujourd'hui, les ordinateurs sont bien capables de construire des automates LR(1) même si c'est un peu pénible à la main, les restrictions SLR(1) et LALR(1) ont donc un peu perdu l'intérêt.

Conclusion sur LR(0) et LR(1)

- Nous avons la construction d'analyseurs grammaticales LR(1).
- La construction de l'automate caractéristique est, pour des grammaires réalistes, fastidieuse quand on la fait à la main.
- Il nous faut donc un *générateur* qui prend une grammaire en entrée et qui produit le module qui fait l'analyse grammaticale.
- Le générateur échoue quand la grammaire n'est pas LR(1). Dans ce cas il faut comprendre les conflits indiqués par le générateur, et les résoudre.