Compilation

V. Analyse syntaxique ascendante

Jacques Farré

Jacques.Farre@unice.fr

http://deptinfo.unice.fr/~jf/Compil-L3

Introduction

- l'analyse ascendante (ou par *décalage-réduction*) construit (virtuellement) l'arbre de dérivation à partir du bas
  - ce qui revient à suivre à l'envers une chaîne de dérivations droites
  - si $S \underset{\text{rm}}{\Rightarrow}^* \varphi A w \underset{\text{rm}}{\Rightarrow} \varphi \alpha w$, réduire α à A
- exemple avec $S \longrightarrow ABC$, $A \longrightarrow a \mid Aa$, $B \longrightarrow b \mid bB$, $C \longrightarrow c$ et la chaîne *aabbc*
  - $S \underset{\text{rm}}{\Rightarrow} ABC \underset{\text{rm}}{\Rightarrow} ABc \underset{\text{rm}}{\Rightarrow} AbBc \underset{\text{rm}}{\Rightarrow} Abbc \underset{\text{rm}}{\Rightarrow} Aabbc \underset{\text{rm}}{\Rightarrow} aabbc$
  - réduire *a* à *A*, puis *Aa* à *A*, ...
  - problème : quel *b* réduire à *B* ; clairement pas le 1er, car on aurait alors *ABbc* avec impossibilité de réduire *Bb*
  - quels sont les critères pour effectuer une réduction ?
  - quelle réduction effectuer ?
- une méthode puissante : LR (variantes SLR, LALR)

Préfixes viables et poignées

- $\varphi\alpha$ est un *préfixe viable* si $S \underset{\text{rm}}{\Rightarrow}^* \varphi\alpha w$; si $\varphi\alpha$ peut être obtenu par $S \underset{\text{rm}}{\Rightarrow}^* \varphi A w \underset{\text{rm}}{\Rightarrow} \varphi\alpha w$, alors α est une *poignée* qu'on peut *réduire* à A
- la pile d'un analyseur LR contient toujours un préfixe viable
  - s'il y a une poignée en haut de la pile, on réduit, sinon on *décale* (avance et empile) dans le texte
- reprenons avec $S \underset{\text{rm}}{\Rightarrow} ABC \underset{\text{rm}}{\Rightarrow} ABc \underset{\text{rm}}{\Rightarrow} AbBc \underset{\text{rm}}{\Rightarrow} Abbc \underset{\text{rm}}{\Rightarrow} Aabbc \underset{\text{rm}}{\Rightarrow} aabbc$

pile	texte restant	action
	aabbc	décaler
a	*abbc*	réduire *a* à *A*
A	*abbc*	décaler
Aa	*bbc*	réduire *Aa* à *A*
A	*bbc*	décaler
Ab	*bc*	réduire *b* à *B* ? non car *ABb* ne serait pas un préfixe viable ⇒ décaler
Abb	*c*	réduire *b* à *B* ? oui car *AbB* est un préfixe viable
AbB	*c*	réduire *bB* à *B*, etc...

Les types de conflit qu'on peut avoir

- supposons qu'en plus de la pile, l'automate peut regarder k symboles du texte restant
- la grammaire est LR(k) si, et seulement, dans les cas suivants
  - elle n'a pas de conflit réduction/réduction :
    si $S \underset{\text{rm}}{\Rightarrow}^* \varphi A w \underset{\text{rm}}{\Rightarrow} \varphi \alpha w$ et $S \underset{\text{rm}}{\Rightarrow}^* \varphi' B w' \underset{\text{rm}}{\Rightarrow} \varphi' \beta w'$, alors $\varphi\alpha \neq \varphi'\beta$ ou $k : w \neq k : w'$
    exemple de conflit r/r (pour $k = 1$) :
    $S \rightarrow fAcd \mid faBce, A \rightarrow ab, B \rightarrow b$ (avec fab en pile, réduire ab à A ou b à B ?)

  - elle n'a pas de conflit décalage/réduction :
    si $S \underset{\text{rm}}{\Rightarrow}^* \varphi A w \underset{\text{rm}}{\Rightarrow} \varphi \alpha w$ et $S \underset{\text{rm}}{\Rightarrow}^* \varphi' B x \underset{\text{rm}}{\Rightarrow} \varphi' \beta y x$, alors $\varphi\alpha \neq \varphi'\beta$ ou $k : w \neq k : yx'$
    exemple de conflit d/r (pour $k = 1$) :
    $S \rightarrow fAbc \mid fBce, A \rightarrow a, B \rightarrow ab$ (avec fa en pile, réduire a à A ou décaler b pour réduire ab à B ?)

Principes de l'analyse LR

- un théorème important : les préfixes viables appartiennent à un langage régulier
- ces préfixes peuvent être regroupés en un *nombre fini* de classes d'équivalences (par exemple la classe des préfixes correspondant à une certaine poignée)
- on peut donc reconnaître les préfixes viables et les poignées par un automate fini qui explore la pile
  - en fait pas besoin d'explorer la pile : si on met dans la pile les états de l'automate fini, chaque état correspond à la reconnaissance d'un ensemble de préfixes
  - le sommet de pile "synthétise" la classe d'équivalence du préfixe présent dans la pile (ou, autrement dit, représente leur classe d'équivalence)

Différence importante entre analyse LR et LL

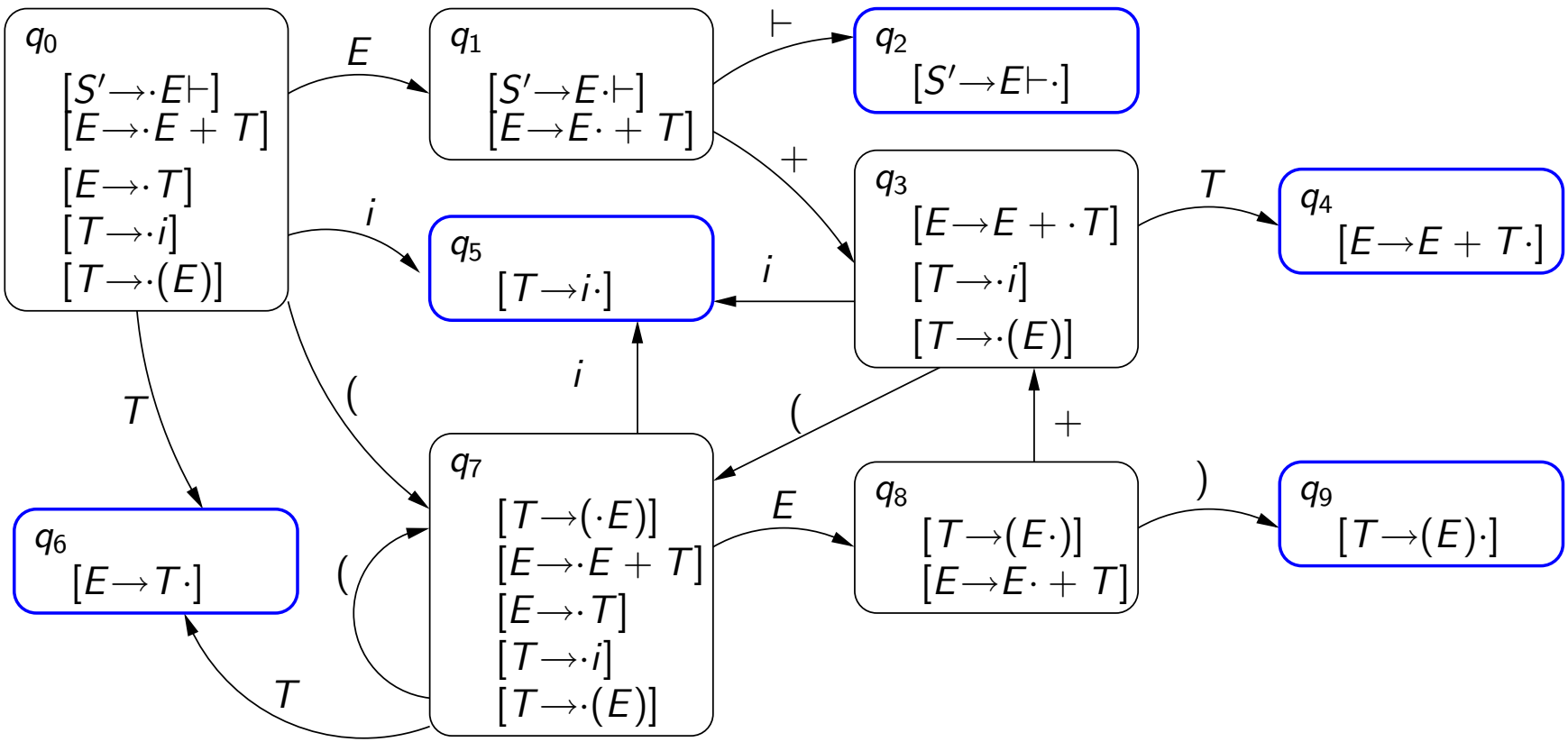
- en LR, le contenu de la pile permet de déterminer si on réduit une poignée (les k symboles de fenêtre permettant au besoin de choisir quelle réduction effectuer)
- en LL, la pile ne permet pas de savoir selon quelle règle remplacer le non terminal (l'examen de k symboles est nécessaire)
- LR accepte donc les récursivités gauche (et dans un sens, les préfère aux récursivités droites)
- exemple avec $S \rightarrow dSe \mid aab \mid aac$ ($\mathcal{L}(S) = d^n aa(b|c)e^n$)
  - LL(3), car il faut regarder jusqu'au b pour décider quelle règle appliquer entre $S \rightarrow aab$ et $S \rightarrow aac$
  - LR(0), car tout préfixe d^*aab permet de décider la réduction $A \rightarrow aab$ (d^*aab est la classe d"equivalence des $d^naab, n \geq 0$)

Analyse LR(0)

- la construction d'un analyseur LR(0) ressemble fortement à la construction de la suite des états de l'algorithme de Earley
- on utilise des *items LR* de la forme $[A \rightarrow \alpha \cdot \beta]$ indiquant que la partie α de la règle $A \rightarrow \alpha\beta$ a été reconnue, et qu'il faut reconnaître β
- les items de la forme $[A \rightarrow \alpha \cdot X \beta]$ produisent une transition vers un état initialement composé des items de la forme $[A \rightarrow \alpha X \cdot \beta]$
- un item de la forme $[A \rightarrow \alpha \cdot B \beta]$ implique une *fermeture*, c'est à dire l'ajout à l'état d'items de la forme $[B \rightarrow \cdot \gamma]$, pour toute règle $B \rightarrow \gamma$ de la grammaire
- un item de la forme $[B \rightarrow \gamma \cdot]$ signifie qu'on a trouvé une poignée
- l'état initial est la fermeture de l'item $S' \rightarrow \cdot S \vdash$

Exemple de construction d'automate LR(0)

- soit la grammaire $G_1 = \{E \rightarrow E + T \mid T,\ T \rightarrow i \mid (\ E\ )\}$
- son automate LR(0) des préfixes viables



Actions d'un automate LR(0)

- initialement, la pile contient l'état q_0
- on a 2 types d'actions, commandées uniquement par l'état en sommet de pile :
  - réduire selon une règle, dans les états avec un item de la forme $[A \rightarrow \alpha\cdot]$: on dépile $|\alpha|$ états
  - avancer, dans les états avec un item de la forme $[A \rightarrow \alpha\cdot a\beta]$
- selon le sommet de pile et le symbole lu (avancer) ou réduit (A si on réduit selon $A \rightarrow \alpha$), une table de saut permet d'empiler un nouvel état
- si un état contient des items décidant d'actions différentes, on a un conflit LR(0)
  - conflit décalage-réduction (*shift-reduce*) si $[A \rightarrow \alpha\cdot]$ et $[B \rightarrow \beta\cdot a\gamma]$
  - conflit réduction-réduction (*reduce-reduce*) si $[A \rightarrow \alpha\cdot]$ et $[B \rightarrow \beta\cdot]$, ($A \neq B$ ou $\alpha \neq \beta$)

Tables de l'automate LR(0) pour G_1

état	action	table de saut						
		i	$+$	$($	$)$	$\vdash$	E	T
q_0	avancer	q_5		q_7			q_1	q_6
q_1	avancer		q_3			q_2		
q_2	accepter							
q_3	avancer	q_5		q_7				q_4
q_4	réd. $E \rightarrow E + T$							
q_5	réd. $T \rightarrow i$							
q_6	réd. $E \rightarrow T$							
q_7	avancer	q_5		q_7			q_8	q_6
q_8	avancer		q_3		q_9			
q_9	réd. $T \rightarrow (E)$							

les entrées vides provoquent une erreur

Exécution de l'automate LR(0) pour G_1 sur $i + (i + i)$

pile	texte restant	action et saut
q_0	$i + (i + i)\vdash$	avancer et $(q_0, i) \rightarrow q_5$
$q_0 q_5$	$+(i + i)\vdash$	red. $T \rightarrow i$ et $(q_0, T) \rightarrow q_6$
$q_0 q_6$	$+(i + i)\vdash$	red. $E \rightarrow T$ et $(q_0, E) \rightarrow q_1$
$q_0 q_1$	$+(i + i)\vdash$	avancer et $(q_1, +) \rightarrow q_3$
$q_0 q_1 q_3$	$(i + i)\vdash$	avancer et $(q_3, () \rightarrow q_7$
$q_0 q_1 q_3 q_7$	$i + i)\vdash$	avancer et $(q_7, i) \rightarrow q_5$
$q_0 q_1 q_3 q_7 q_5$	$+i)\vdash$	red. $T \rightarrow i$ et $(q_7, T) \rightarrow q_6$
$q_0 q_1 q_3 q_7 q_6$	$+i)\vdash$	red. $E \rightarrow T$ et $(q_7, E) \rightarrow q_8$
$q_0 q_1 q_3 q_7 q_8$	$+i)\vdash$	avancer et $(q_8, +) \rightarrow q_3$
$q_0 q_1 q_3 q_7 q_8 q_3$	$i)\vdash$	avancer et $(q_3, i) \rightarrow q_5$
$q_0 q_1 q_3 q_7 q_8 q_3 q_5$	$)\vdash$	red. $T \rightarrow i$ et $(q_3, T) \rightarrow q_4$
$q_0 q_1 q_3 q_7 q_8 q_3 q_4$	$)\vdash$	red. $E \rightarrow E + T$ et $(q_7, E) \rightarrow q_8$
$q_0 q_1 q_3 q_7 q_8$	$)\vdash$	avancer et $(q_8,)) \rightarrow q_9$
$q_0 q_1 q_3 q_7 q_8 q_9$	$\vdash$	red. $T \rightarrow (E)$ et $(q_3, T) \rightarrow q_4$
$q_0 q_1 q_3 q_4$	$\vdash$	red. $E \rightarrow E + T$ et $(q_0, E) \rightarrow q_1$
$q_0 q_1$	$\vdash$	avancer et $(q_1, \vdash) \rightarrow q_2$
q_2		accepter

Analyse SLR(1)

- prenons maintenant la grammaire
  $G_2 = \{E \rightarrow E + T \mid T,\ T \rightarrow T * F \mid F,\ F \rightarrow i\}$
- il est facile de voir que les items $[E \rightarrow \cdot T]$ et $[T \rightarrow \cdot T * F]$ de q_0, après transition sur T, donneront un état contenant $[E \rightarrow T \cdot]$ et $[T \rightarrow T \cdot * F]$: conflit décalage-réduction !
- on peut voir que les suivants de E sont $+$ et $\vdash$, mais pas $*$
- on peut donc décider le décalage sur $*$ et la réduction sur les suivants de E
- on obtient un automate *SLR(1)*, soit *simple* LR(1) (la notion peut s'étendre à SLR(k)) :
  - $[A \rightarrow \alpha \cdot]$ décide une réduction uniquement sur les suivants de A
  - $[A \rightarrow \alpha \cdot a\beta]$ décide un décalage uniquement sur a

Tables SLR(1)

- la table d'actions de l'analyseur SLR(1) regarde l'état en sommet de pile et le prochain symbole du texte
  - au lieu de mettre simplement *avancer* dans une entrée de la table, on peut indiquer l'état où mène ce décalage
  - et supprimer les colonnes des terminaux dans la table de sauts
- la table SLR(1) de la grammaire G_1 est donc :

état	table d'action					table de saut	
	i	$+$	$($	$)$	$\vdash$	E	T
q_0	q_5		q_7			q_1	q_6
q_1		q_3			accepter		
q_2			**état devenu inutile**				
q_3	q_5		q_7				q_4
q_4		$r_{E\longrightarrow E+T}$		$r_{E\longrightarrow E+T}$	$r_{E\longrightarrow E+T}$		
q_5		$r_{T\longrightarrow i}$		$r_{T\longrightarrow i}$	$r_{T\longrightarrow i}$		
q_6		$r_{E\longrightarrow T}$		$r_{E\longrightarrow T}$	$r_{E\longrightarrow T}$		
q_7	q_5		q_7			q_8	q_6
q_8		q_3		q_9			
q_9		$r_{T\longrightarrow (E)}$		$r_{T\longrightarrow (E)}$	$r_{T\longrightarrow (E)}$		

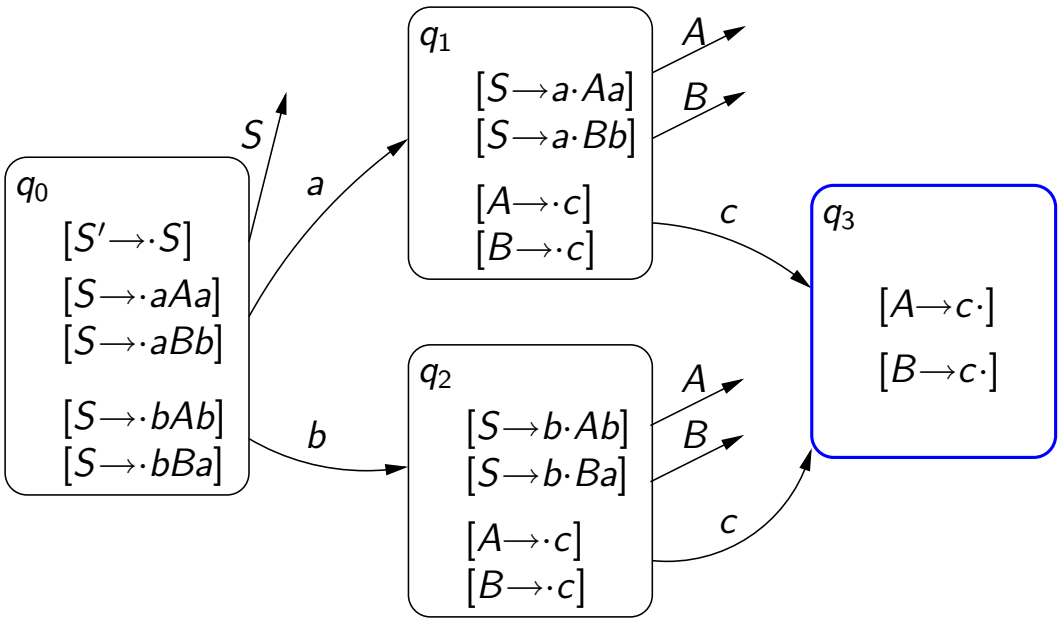
Moteur d'un analyseur LR(1)

- ce moteur est valable pour tout analyseur de la classe LR(1) (LR, SLR, LALR), avec *action* et *saut* les tables d'action et de saur respectivement

initialement, la pile contient q_0
répéter
  fini := *faux*
  soit s l'état en sommet de pile et c le prochain symbole
    si *action[s,c]* = q_i **alors**
      empiler q_i *et avancer dans le texte*
    sinon si *action[s,c]* = $r_{A \longrightarrow \alpha}$ **alors**
      dépiler $|\alpha|$ *états de la pile*
      soit s' l'état qui est maintenant en sommet de pile
        empiler saut[s',A]
    sinon si *action[s,c]* = *accepter* **alors**
      fini := *vrai*
    sinon
      erreur()
tantque non *fini*

Grammaire non SLR(1)

- soit la grammaire $G_3 = \{S \rightarrow aAa \mid bAb \mid aBb \mid bBa,\ A \rightarrow c,\ B \rightarrow c\}$ et un fragment de son automate LR(0)



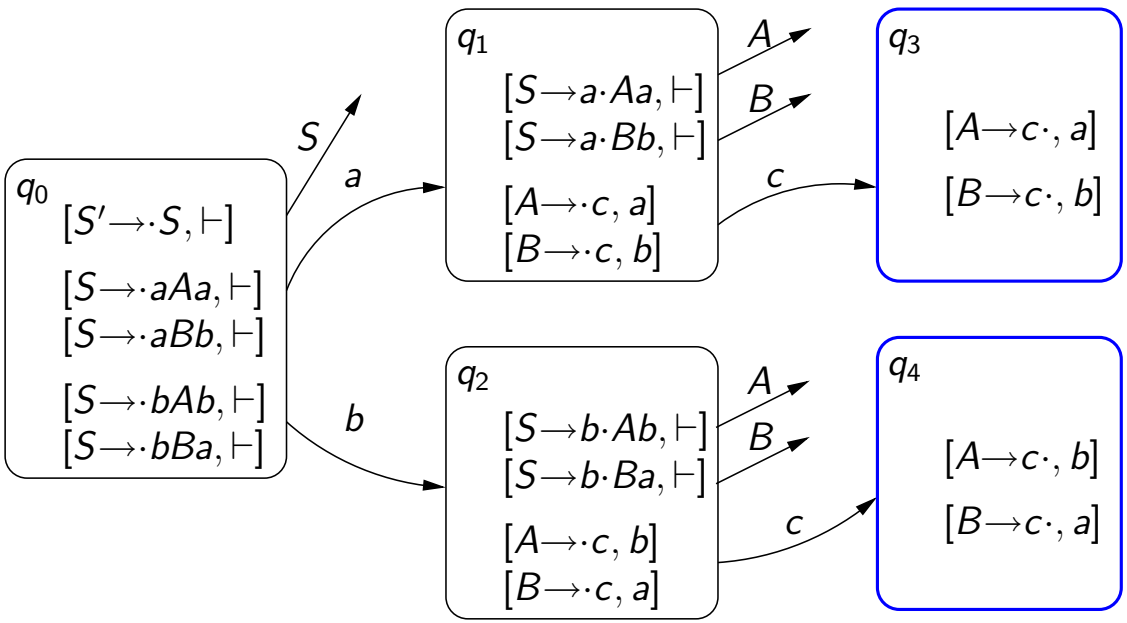
- cette grammaire n'est pas SLR(1) : $Suiv(A) = Suiv(B) = \{a, b\}$ ne permet pas de résoudre le conflit en q_3
- pourtant, ac suivi de a commande clairement $A \rightarrow c$, et suivi de b il commande $B \rightarrow c$ (et inversement pour le préfixe bc)

Analyse LR(1)

- la faiblesse de l'analyse SLR(1) est qu'elle ne tient pas compte de ce qui a été reconnu (synthétisé dans la pile) pour prédire les suivants qu'on peut attendre dans ce contexte
- l'analyse LR y rémédie en ajoutant aux items LR(0) les suivants attendus
  - un *item LR(1)* est de la forme : $[A \rightarrow \alpha \cdot \beta, t]$, où t est un symbole qui peut suivre A **dans le contexte courant de** A
  - on peut étendre t à des mots de longueur k pour un analyseur LR(k)
- la différence avec la construction de l'automate LR(0) réside dans la fermeture :
  - pour les items de la forme $[A \rightarrow \alpha \cdot B\beta, t]$, on ajoute les items $[B \rightarrow \cdot\gamma, u]$ où $u \in Prem(\beta t) : B$, dans ce contexte, ne peut être suivi que d'un premier de β, ou de t si β est vide ou produit le vide
  - l'état q_0 est la fermeture de $[S' \rightarrow \cdot S, \vdash]$

Automate (partiel) LR(1) de G_3

- les transitions sur c à partir de q_1 et q_2 donnent maintenant des items différents, et donc 2 états différents



- on sait maintenant comment réduire en regardant le prochain symbole : la grammaire est bien LR(1)

Construction des tables d'analyse LR(1)

- pour la table d'action, si, dans un état q_i
  - on a un item $[A \rightarrow \alpha \cdot a\beta, t]$, $action[q_i, a] = q_j$, si q_j est l'état atteint à partir de q_i par la transition sur a
  - on a un item $[A \rightarrow \alpha \cdot, t]$, $action[q_i, t] = r_{A \rightarrow \alpha}$, (sauf pour $[S' \rightarrow S \cdot, \vdash]$ où l'action est *accepter*)
  - toute entrée non remplie correspond à une erreur (la phrase analysée n'appartien pas au langage)
  - toute entrée ayant 2 actions différentes implique du non déterminisme : la grammaire n'est pas LR(1))
    - une méthode, *LR généralisé* (GLR), très utilisée pour les langues naturelles, permet de gérer ce non déterminisme (bison peut fonctionner en mode GLR)
- pour la table de saut
  - pour un état q_i transitant pas A (A non terminal) sur q_j, $saut[q_i, A] = q_j$

Force et faiblesse de la méthode LR(1)

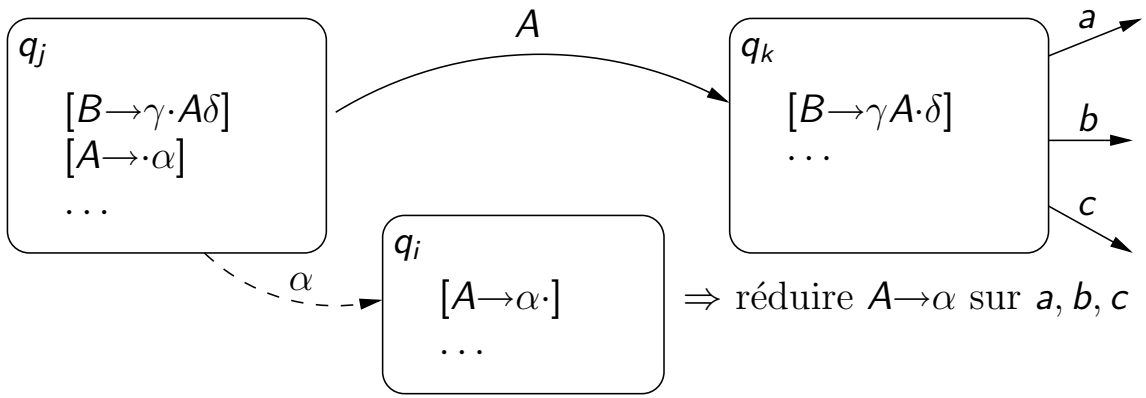
- sa force : tout langage dit *déterministe* a une grammaire LR(k)
  - et toute grammaire LR(k) peut être transformée mécaniquement en grammaire LR(1)
  - mais il existe des langages non ambigus dits *non déterministes* qui n'ont pas de grammaire LR(k)
- sa faiblesse : son grand nombre d'états par rapport à LR(0)/SLR(1)
  (on a vu qu'ajouter le suivant dans les items LR(1) faisait éclater certains états)
  - soit $|G|$ la *taille d'une grammaire* (le nombre total de symboles apparaissant dans les règles)
  - à une règle $A \rightarrow \alpha$ correspondent $|A\alpha|$ items LR(0), donc le nombre total d'items LR(0) différents est $|G|$
  - un état est un sous-ensemble d'items LR(0) : on peut avoir au plus $2^{|G|}$ états LR(0) (puisque le nombre de sous-ensembles d'un ensemble de n éléments est 2^n)
  - on a $|G| \times |T|$ items LR(1) possibles (à cause de l'introduction des suivants), donc $2^{|G| \times |T|}$ états possibles : en pratique, l'automate LR(1) peut facilement être 10 à 100 fois plus gros que le LR(0) !

Un compromis : l'analyse LALR(1)

- en pratique, il est très fréquent que des états LR(1) ayant les mêmes noyaux LR(0) d'items, mais pas les mêmes suivants, puissent être fusionnés sans créer de conflit
  - par exemple, $q_i = \{[A \rightarrow \alpha \cdot, t], [B \rightarrow \beta\alpha \cdot, u]\}$ et $q_j = \{[A \rightarrow \alpha \cdot, v], [B \rightarrow \beta\alpha \cdot, w]\}$
- donc, on peut ramener la taille de l'automate à celle de LR(0), tout en conservant en grande partie le bénéfice du calcul plus précis des suivants de la méthode LR(1)
- toutefois, l'idée de construire un automate LR(1) et de fusionner des états quand c'est possible, même si elle est correcte d'un point de vue théorique, pose le problème de la complexité (en temps et en espace) de la construction d'un automate LR(1)
- il existe un moyen plus efficace pour calculer les suivants LR(1) en se basant sur un automate LR(0)

Principes de construction d'un analyseur LALR(1) à partir d'un automate LR(0)

- tout repose sur la remarque suivante :
  - si un état LR(0) q_i contient $[A \rightarrow \alpha\cdot]$, alors il existe (au moins) un état q_j transitant par α sur q_i qui contient $[A \rightarrow \cdot\alpha]$ et par conséquent $[B \rightarrow \gamma\cdot A\delta]$
  - l'état q_k par lequel q_j transite par A (il contient donc $[B \rightarrow \gamma A\cdot\delta]$ contient lui-même des transitions sur des nonterminaux (qui sont en fait les premiers de δ)
  - donc, on remontant la transition α et en suivant la transition A, on arrive à q_k, toute transition de q_k par un non terminal t indique que t est un suivant de A *dans ce contexte*



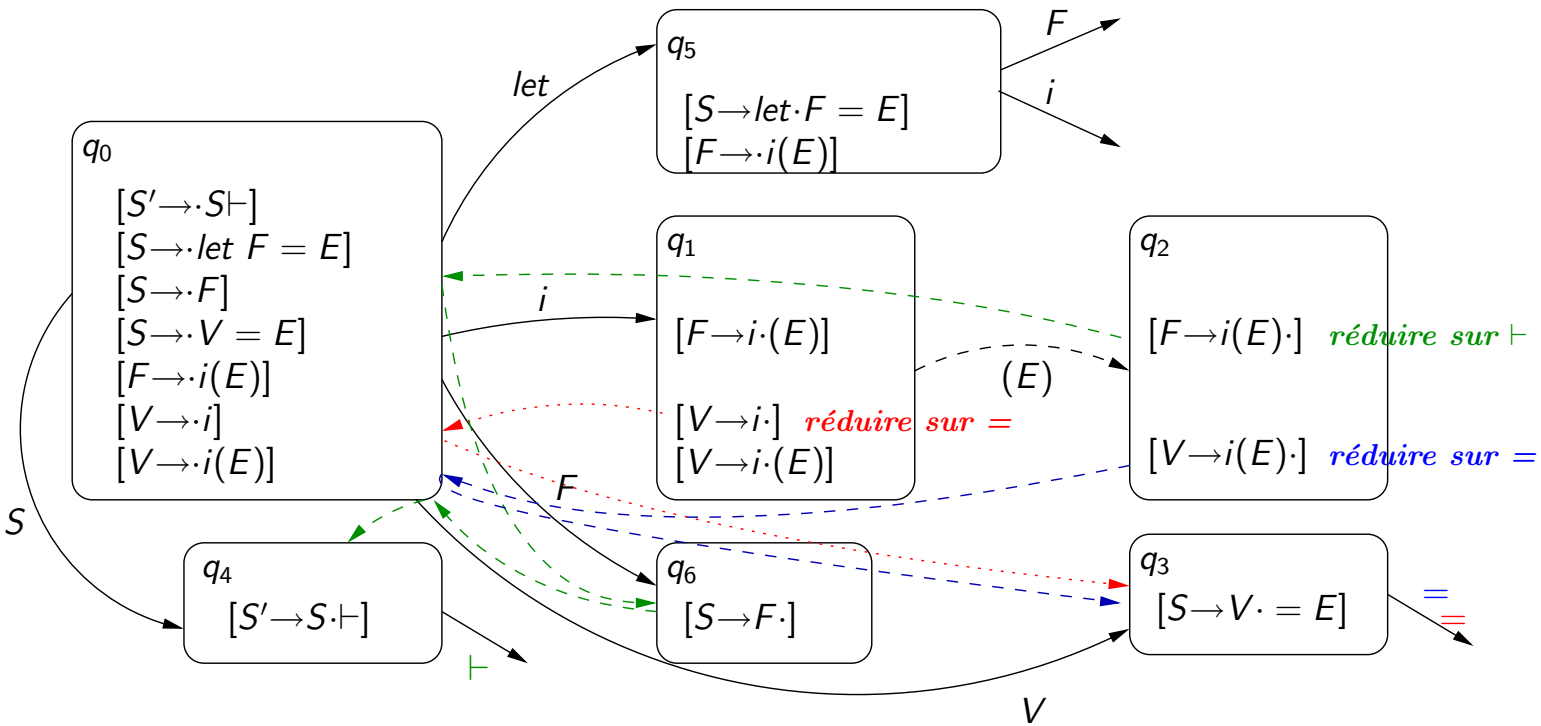
Construction d'un analyseur LALR(1)

- construire l'automate LR(0) de la grammaire
- pour tout état q_i contenant un conflit LR(0), calculer l'ensemble $SC(A \rightarrow \alpha, q_i)$ des *suivants contextuels* pour chaque réduction $r_{A \rightarrow \alpha}$ de q_i
  - décider la réduction $r_{A \rightarrow \alpha}$ sur tout terminal $t \in SC(A \rightarrow \alpha, q_i)$ (il y a évidemment un conflit LALR(1) si un même t décide plusieurs actions différentes)

- calcul de $SC(A \rightarrow \alpha, q_i)$:
  - pour tout q_j transitant par α sur q_i, soit q_k atteint de q_j par A $SC(A \rightarrow \alpha, q_i) = \{t \mid q_k \text{ transite par } t \text{ sur un autre état}, t \in T\}$
  - si q_k contient $[B \rightarrow \varphi A \cdot]$, ajouter $SC(B \rightarrow \varphi A, q_k)$ à $SC(A \rightarrow \alpha, q_i)$, et ceci récursivement

Exemple de construction d'un analyseur LALR(1)

- soit la grammaire $S \rightarrow let\ F = E$, $S \rightarrow F$, $S \rightarrow V = E$, $V \rightarrow i$, $V \rightarrow i(E)$, $E \rightarrow V$, $F \rightarrow i(E)$
- et soit le fragment de son automate LR(0) et les suivants contextuels LALR(1)



- la grammaire est bien LALR(1) (mais pas SLR(1) car = est un suivant de F)

Forces et faiblesses de LALR(1)

- forces :
  - utilise un automate LR(0), donc avec nettement moins d'états que LR(1)
  - calcule les suivants dans le contexte presque aussi exactement que LR(1) (sauf pour des cas assez particuliers)
  - est en général beaucoup plus puissante que LL(1) : il est rare qu'une grammaire LL(1) ne soit pas LALR(1), mais beaucoup de grammaires LALR(1) ne sont pas LL(1) ne serait-ce qu'à cause de la récursivité gauche ou les parties droites partageant des préfixes communs
- faiblesses :
  - pas toujours facile de voir comment sont calculés les suivants, et par là, pas toujours facile de comprendre pourquoi il y a un conflit et comment le résoudre

Bison : un générateur d'analyseurs syntaxiques LALR(1)

- *Bison* crée un analyseur LALR(1), auquel il est possible d'ajouter des actions sémantiques et d'indiquer comment réparer les erreurs
- Forme générale d'une grammaire pour *Bison*

```
[définitions]
%%
[règles]
[%%
 code C
]
```

  sauf le premier %%, tout ce qui est indiqué entre [ ] est optionnel.
- Les espaces (blanc, fin de ligne, tabulation) ne sont pas significatifs. Les commentaires sont comme en C.

Règles

- Elles sont de la forme générale (forme EBNF non acceptée) :

```
NT   : S1 S2 ... { action en C }
     | ...
     ;
```

  S_i est un ident de non terminal, ou de terminal (déclaré par `%token` et reconnu par l'analyseur lexical), ou une chaîne de **1 seul** caractère

- exemple pour les expressions arithmétiques

```
expr : expr '+' term
     | expr '-' term
     | term
     ;
term : term '*' fact
     | term '/' fact
     | fact
     ;
fact : NOMBRE
     | '(' expr ')'
     ;
```

Déclaration des terminaux

- déclaration normale

```
%token IDENT NOMBRE KWHILE
```

- déclaration avec priorité et associativité (pour les grammaires ambiguës notamment)

```
%nonassoc EQU '<' '>' NEQ LEQ GEQ
%left '+' '-'
%left '*' '/'
%%
exp : exp '*' exp
```

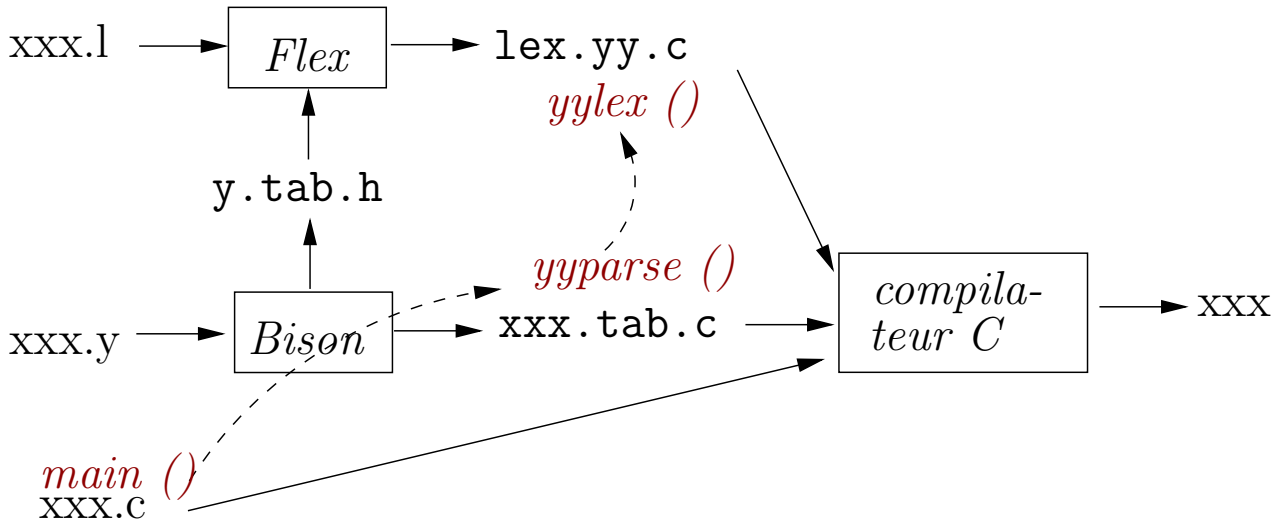
`a * b * c` ambigu (et conflit d/r), mais l'associativité gauche permet de résoudre le conflit en faveur de la réduction `a * b` à *expr*

- modification locale de la priorité dans une règle

```
exp : '-' exp  %prec '*'
```

`- a * b` ambigu (et conflit d/r), mais `-a` réduit à *expr* car **ce** `-` prend l'associativité de `*`

Le couple Flex-Bison



- `flex xxx.l` produit un programme C dans `lex.yy.c`
- `bison xxx.y` produit un programme C dans `xxx.tab.c` (option `-d` ⇒ définitions dans `xxx.tab.h`, option `-y` ⇒ `y.tab.c`, `y.tab.h` au lieux de `xxx.tab[ch]`)
- compiler, relier avec les bibliothèques :
  `gcc xxx.yy.c xxx.tab.c xxx.c -ll -ly`

Interface Flex-Bison

- les déclarations de token dans Bison donnent une macro (dans `xxx.tab.h`), par exemple `%token ID` donne `#define ID 256`
- inclure `xxx.tab.h` dans le source flex
- les actions dans flex doivent retourner le code du token

```
[A-Za-z]+   return ID;
```

- les tokens peuvent calculer des attributs, par exemple en Bison

```
%union {
   int ival;
   ...

%token<ival> NB
```

en Flex, l'attribut d'un token est calculé dans la variable globale `yylval`

```
[0-9]+  { yylval.ival = atoi (yytext); return NB; }
```

Automate LALR

- il est produit sur `xxx.output` par l'option `-v`
- exemple avec la grammaire `nonlalr.y`

```
expr : var '=' expr
     | var
     | '(' expr ')'
     ;
var : ID
    | var '[' expr ']'
    | '*' expr
    ;
```

- appel à Bison :

```
bison -v nonlalr.y
conflicts: 2 shift/reduce
```

Automate LALR – 1

- automate : conflit dans l'état 2 (le • des items LR est noté _)

```
2: shift/reduce conflict (shift 6, red'n 2) on =
2: shift/reduce conflict (shift 7, red'n 2) on [
state 2
        expr :  var_= expr
        expr :  var_    (2)
        var :  var_[ expr ]

        =  shift 6
        [  shift 7
        .  reduce 2
```

- var en pile, = ou [en fenêtre : soit réduire var à expr, soit décaler
- le décalage (action par défaut) semble correct ici (on peut aussi indiquer %right '=' pour être plus sûr)

Automate LALR – 2

- mais l'état 5, qui marque la rencontre de `*`, va sur l'état 2

```
state 5
        var :  *_expr

        ID  shift 4
        (  shift 3
        *  shift 5
        .  error

        expr  goto 9
        var  goto 2
```

- donc `* t[x]` sera analysé comme `* (t[x])` (correct),
- mais `*p = x` sera analysé comme `* (p = x)` (incorrect)

Automate LALR – 3

- il faut soit changer la grammaire

```
expr : var '=' expr | var
     ;
var  : v | '*' var
     ; /*  réduction avec = en fenêtre */

v    : ID | '(' expr ')' | v '[' expr ']'
     ;  /* décalage avec [ en fenêtre */
```

- soit garder la grammaire originale avec :

```
%right '='
%right '*'
%left '['
%%
expr : var '=' expr
     | var                  %prec '*'
     | '(' expr ')'
     ;
var  : ID | var '[' expr ']' | '*' expr
     ;
```

Résolution des conflits LALR(1)

- S'il y a des conflits LALR(1), les règles appliquées par défaut sont :
  - conflit décalage/réduction (shift/reduce) : effectuer le shift.
  - conflit réduction/réduction (reduce/reduce) : effectuer la réduction par la règle définie la première parmi les règles en conflit.

```
exp : exp '-' exp
    | exp '*' exp
    | NOMBRE
```

  donc `3 - 2 * 4` $\longrightarrow$ analysé par défaut comme `(3 - 2) * 4`
- Les conflits peuvent être levés grâce aux précédences des lexèmes

```
%left '-'
%left '*' /* plus prioritaire que + car déclaré après */
```

Définition des types des attributs

- définition de l'union des attributs

```
%union {
  int v;
  struct S {char* c; int n;} s;
}
```

- attributs des lexèmes

```
%token <s> '+' X1
%left <v> X2 '('
```

- attributs des non-terminaux

```
%type <s> NT1 NT2, ...
%type <v> NT3 NT4, ...
```

- on verra l'utilité des attributs lors de l'analyse sémantique

Reprise sur erreurs

- quand une erreur est détectée, la fonction `yyerror (char*)` est appelée
- par défaut, le paramètre est `"parse error"`
- Bison permet d'avoir des messages plus précis en définissant dans la partie C des déclarations

```
#define YYERROR_VERBOSE 1
```

- si aucune forme de reprise d'erreur n'est donnée, l'analyse s'arrête
- reprise d'erreur = mode *panique* : on avance dans la phrase jusqu'à un certain caractère, et on remet la pile en état

```
ligne : expression '\n'         { ... }
      | expression error '\n'   { yyerrok; }
```