Introduction à la Compilation

Cours 6 : L'analyse syntaxique en pratique

Yann Régis-Gianas
yrg@pps.jussieu.fr

PPS - Université Denis Diderot – Paris 7

vendredi 5 mars 2010

Construction de l'arbre de production

Algorithmes étudiés

- Nous avons étudiés les algorithmes suivants :
  - Unger
  - LL(1)
  - CYK
  - Analyse syntaxique par recherche
  - Earley
  - LR(1)
- Le problème de la construction explicite de l'arbre de production a été ignoré.

Exercice

Adaptez chacun de ces algorithmes pour qu'il produise un (ou plusieurs ?) arbre(s) de production.

Grammaire des arbres de production

- Soit G, une grammaire hors-contexte.
- Soit w, un mot de longueur n.
- On définit l'alphabet $\mathcal{P}$ des non-terminaux de production ainsi :

$$\{s_{[i;l]} \mid s \in T \cup N, 1 \leq i \leq n, 1 \leq l \leq n\}$$

- Le non-terminal $s_{[i;l]}$ représente le symbole s qui a produit le sous-mot de w de longueur l à la position i.
- Une grammaire d'arbres de production du mot w par rapport à G a :
  - le même alphabet de terminaux que G ;
  - pour alphabet de non-terminaux les non-terminaux de production ;
  - pour règles les instances des règles de G utilisées pour produire w ;
  - pour langage $\{w\}$.

Exemple

- Soit la grammaire :

$$\begin{array}{rcl} S & ::= & S + S \\ & | & a \end{array}$$

- Une grammaire de production du mot « $a + a + a$ » par la grammaire G est :

$$\begin{array}{rcl} S & ::= & S_{[1;5]} \\ S_{[1;5]} & ::= & S_{[1;3]} + S_{[5;1]} \\ & | & S_{[1;1]} + S_{[3;3]} \\ S_{[1;3]} & ::= & S_{[1;1]} + S_{[3;1]} \\ S_{[3;3]} & ::= & S_{[3;1]} + S_{[5;1]} \\ S_{[1;1]} & ::= & a_{[1;1]} \\ a_{[1;1]} & ::= & a \\ S_{[3;1]} & ::= & a_{[3;1]} \\ a_{[3;1]} & ::= & a \\ S_{[5;1]} & ::= & a_{[5;1]} \\ a_{[5;1]} & ::= & a \end{array}$$

$\Rightarrow$ Cette grammaire produit ce mot de deux façons différentes.

- En construisant l'ensemble de ces dérivations, on produit une forêt d'arbres.

Application sur l'algorithme de Unger

- À chaque fois que l'on essaie une partition, on rajoute la règle correspondante à la grammaire de production.
- Par exemple, pour la grammaire précédente, pour le partitionnement du mot « $\mid a + \mid a + a$ » et la règle « $S \longrightarrow S + S$ », on produit :

$$S_{[1;5]} ::= S_{[1;0]} \ +_{[1;2]} \ S_{[3;3]}$$

- La plupart des règles rajoutées, comme celle-ci par exemple, ne mèneront à la production d'aucun mot lorsqu'elles correspondent à des essais voués à l'échec. (Ici, par exemple le non terminal $+_{[1;2]}$ ne peut produire aucun mot de longueur 2.)

Application sur l'algorithme CYK

- À chaque fois qu'un non-terminal A est inséré dans une case $R_{i,l}$ parce qu'il existe une règle $A \longrightarrow BC$ telle que $B \in R_{i,k}$ et $C \in R_{i+k,l-k}$ alors on ajoute la règle suivante dans la grammaire de production :

$$A_{[i;l]} \longrightarrow B_{[i;k]} C_{[i+k;l-k]}$$

Application sur LL et LR

- Pour LL(1) lors des phases de prédiction et pour LR(1) lors des phases de réduction, on peut maintenir suffisamment d'information sur la pile pour produire la règle de la grammaire de production concernée par l'action en cours.

Exercice

Précisez ce point.

Résolution des conflits

Les conflits de l'analyse syntaxique

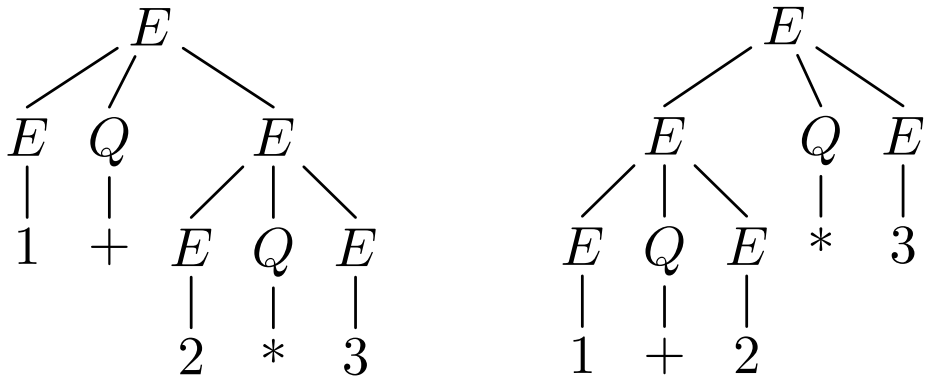
- Une grammaire G est ambiguë si un mot a plusieurs dérivations à travers G.
- Lorsqu'un mot peut être analysé de plusieurs façons, cela signifie que l'analyse syntaxique mène à plusieurs arbres de production, c'est-à-dire une forêt.
- Il y a deux solutions à ce problème :

1. On peut appliquer des critères externes à la grammaire pour filtrer *a posteriori* ces arbres de production, en espérant qu'un unique arbre résulte de ce processus.
2. On peut appliquer un filtrage au fur et à mesure de l'analyse de façon à ne produire qu'un arbre, au plus.

Exemple

$$
\begin{array}{lcl}
E & ::= & E\ Q\ E \mid 0 \mid 1 \mid \ldots \\
Q & ::= & + \mid *
\end{array}
$$

- Le mot « $1 + 2 * 3$ » est associé aux arbres de production :

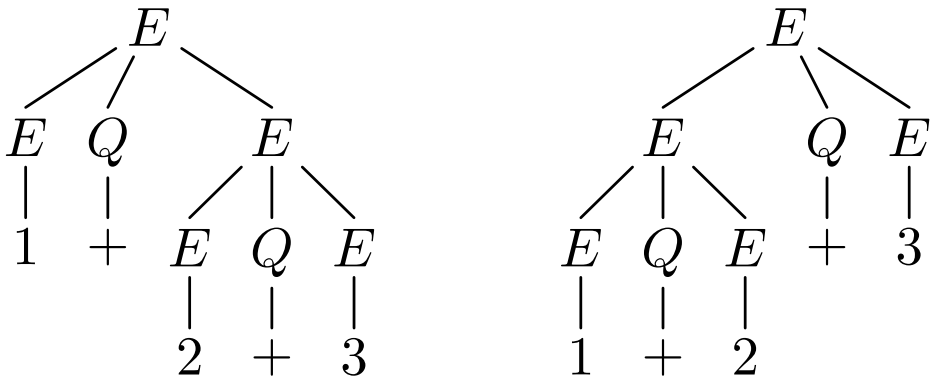


$\Rightarrow$ Le critère est :
La règle « $Q \longrightarrow +$ » doit être “au dessus de” la règle « $Q \longrightarrow *$ » dans l'arbre.

Exemple

$$
\begin{array}{lcl}
E & ::= & E\ Q\ E \mid 0 \mid 1 \mid \ldots \\
Q & ::= & + \mid *
\end{array}
$$

- Le mot « $1 + 2 + 3$ » est associé aux arbres de production :



$\Rightarrow$ Le critère est :
Une séquence d'application de la règle « $Q \rightarrow +$ » doit formée un arbre "qui penche à gauche".

Inconvénients de la résolution *a posteriori*

- L'application de critères de filtrage sur une forêt d'arbres nécessite un algorithme qui produit effectivement une forêt (ce qui n'est pas le cas des algorithmes déterministes LL et LR que nous avons étudiés).
- Certaines ambiguités des grammaires peuvent conduire à la construction de forêts de taille exponentielle.

$\Rightarrow$ Cela pose des problèmes de représentation en mémoire et le temps nécessaire au filtrage peut s'avérer très important.

Résolution des conflits pour les algorithmes déterministes

- Les grammaires possédant des conflits LL et LR sont par définition inutilisables avec ces algorithmes.
- On peut cependant supprimer ces conflits par deux procédés :
  - l'utilisation d'une information externe à la grammaire (priorité, associativité) ;
  - la réécriture de la grammaire en une grammaire équivalente.

Conflits LL

- Nous avons vu qu'il existe 3 types de conflits LL :
  - FIRST/FIRST
  - FIRST/FOLLOW
  - FOLLOW/FOLLOW

Exemple

$$
\begin{array}{lcl}
E & ::= & E\ L\ E \mid 0 \mid 1 \mid \ldots \\
L & ::= & L\ Q \mid \epsilon \\
Q & ::= & T\ O \mid T \\
O & ::= & + \mid * \mid \epsilon \\
T & ::= & . \mid \epsilon
\end{array}
$$

Exercice

Calculer la table de prédiction LL(1) de cette grammaire et classifier ses conflits.

Techniques de résolution de conflits LL(1)

- La récursion à gauche provoque systématiquement un conflit FIRST/FIRST.

$\Rightarrow$ Nous savons comment l'éliminer !

- Une source de conflits FIRST/FIRST provient du partage d'un préfixe entre plusieurs règles d'un même terminal.

$\Rightarrow$ Une factorisation à gauche permet de retarder le choix d'une de ces règles.

- Parfois, d'autres transformations moins systématiques suffisent mais il faut s'assurer qu'on préserve le langage représenté par la grammaire...

Exemple de factorisation à gauche

- La grammaire suivante a un conflit FIRST/FIRST :

$$N \quad ::= \quad ab \mid ac$$

qui est résolu immédiatement par factorisation à gauche :

$$\begin{array}{lll} N & ::= & aM \\ M & ::= & b \mid c \end{array}$$

Exemple

$$
\begin{array}{lcl}
E & ::= & E\ L\ E \mid 0 \mid 1 \mid \ldots \\
L & ::= & L\ Q \mid \epsilon \\
Q & ::= & T\ O \mid T \\
O & ::= & + \mid * \mid \epsilon \\
T & ::= & . \mid \epsilon
\end{array}
$$

Exercice

Existe-t-il une grammaire LL(1) équivalente à cette grammaire ?

Conflits LR(1)

- Les conflits LR sont de deux natures :

  décalage/réduction ou réduction/réduction

Exemple

$$
\begin{array}{lcl}
S & ::= & E\# \\
E & ::= & E \ \textbf{then} \ E \\
 & | & 0 \mid 1 \mid \ldots \\
 & | & E \ E
\end{array}
$$

Exercice

Calculer l'automate LALR(1) de cette grammaire.

Explication du premier conflit

```
** Conflict (shift/reduce) in state 6.
** Tokens involved: THEN INT
** The following explanations concentrate on token THEN.
** This state is reached from s after reading:

e e

** The derivations that appear below have the following common factor:

s
e EOF
(?)

** In state 6, looking ahead at THEN, reducing production
** e -> e e
** is permitted because of the following sub-derivation:

e THEN e // lookahead token appears
e e .

** In state 6, looking ahead at THEN, shifting is permitted
** because of the following sub-derivation:

e e
  e . THEN e
```

Résolution 1 du premier conflit

- Supposons que l'on désire lire le mot « INT INT THEN INT »

  ainsi : « INT (INT THEN INT) ».
- Dans l'automate, cela signifie que l'on donne la priorité dans l'état 6 à l'action de décalage qui passe à l'état 4 à la lecture d'un terminal THEN vis-à-vis de l'action de réduction par la règle 1 « $e \longrightarrow e\ e$ ».
- Ce choix est exprimé à l'aide d'une priorité : il suffit d'écrire dans le prélude de la spécification de la grammaire que THEN a une priorité plus forte que la règle 1.
- La priorité d'une règle est donnée par la priorité de son terminal le plus à droite.
- Or, la règle 1 n'a pas de terminal !
- On utilise la syntaxe :

```
e :  e e %prec app
```

  qui introduit un terminal virtuel « app » qui sert juste à dénoter la priorité de la règle.

Résolution 2 du premier conflit

- Une autre façon de supprimer cet ambiguité s'obtient en réécrivant la grammaire ainsi :

$$
\begin{array}{rcl}
S & ::= & E_0\# \\
E_0 & ::= & E_0\ E_1 \\
 & | & E_1 \\
E_1 & ::= & 0 \mid 1 \mid \ldots \\
 & | & E_0\ \textbf{then}\ E_0
\end{array}
$$

- Cette stratification de la grammaire explicite les priorités relatives des opérateurs.

Explication du second conflit

```
** Conflict (shift/reduce) in state 5.
** Tokens involved: THEN INT
** The following explanations concentrate on token THEN.
** This state is reached from s after reading:

e THEN e

** The derivations that appear below have the following common factor:

s
e EOF
(?)

** In state 5, looking ahead at THEN, shifting is permitted
** because of the following sub-derivation:

e THEN e
       e . THEN e

** In state 5, looking ahead at THEN, reducing production
** e -> e THEN e
** is permitted because of the following sub-derivation:

e THEN e // lookahead token appears
e THEN e .
```

Résolution 1 du second conflit

- Ce conflit porte sur un décalage et une réduction provoqués par un terminal qui est le non-terminal le plus à droite de la règle à réduire.
- Supposons que l'on désire lire « INT THEN INT THEN INT»

  ainsi : « (INT THEN INT) THEN INT »
- Cela signifie que l'on veut favoriser la réduction devant le décalage.
- Il suffit de rajout une directive « `%left THEN` » dans le prélude pour exprimer ce choix.
- La directive « `%right THEN` » donnerait la priorité au décalage.
- Tandis que la directive « `%nonassoc THEN` » rejetterait cette entrée.

Résolution 2 du second conflit

- Encore une fois, on peut réécrire la grammaire pour résoudre ce conflit.

$$
\begin{array}{rcl}
S & ::= & E_0\# \\
E_0 & ::= & E_0\ E_1 \\
 & | & E_1 \\
E_1 & ::= & 0 \mid 1 \mid \ldots \\
 & | & E_0\ \textbf{then}\ E_1
\end{array}
$$

Exemple de conflit réduction/réduction

$$
\begin{array}{rcl}
S & ::= & E\ \# \\
E & ::= & E\ Q_1\ E \\
 & | & E\ Q_2\ E \\
 & | & 0 \mid 1 \mid \ldots \\
Q_1 & ::= & + \mid \epsilon \\
Q_2 & ::= & * \mid \epsilon
\end{array}
$$

Exercice

Calculer l'automate LALR(1) de cette grammaire.

Résolution des conflits réduction/réduction

- En général, la résolution des conflits réduction/réduction nécessitent une reformulation de la grammaire.
- La “bonne” formulation de la grammaire précédente est :

$$
\begin{array}{lcl}
S & ::= & E\ \# \\
E & ::= & T\ E \\
 & | & T \\
T & ::= & F\ +\ T \\
 & | & F \\
F & ::= & n\ *\ F \\
 & | & 0 \mid 1 \mid \ldots
\end{array}
$$

Une grammaire LR(2)

$$
\begin{array}{lcl}
G & ::= & R\ G \mid R \\
R & ::= & \text{'}i\text{'}\ \text{' :'}\ D \\
D & ::= & P \\
 & \mid & D\ \text{'}|\text{'}\ P \\
 & \mid & D\ \text{';'} \\
P & ::= & P\ \text{'}i\text{'} \\
 & \mid & \epsilon
\end{array}
$$

- Après avoir lu « i : D | P », si l'analyseur syntaxique voit un terminal « i », il ne peut pas savoir si ce « i » doit être intégré dans la liste des « i » de P ou bien si ce « i » commence un nouveau R.
- Pourtant, en lisant un terminal plus loin, il pourrait décider :
  - si ce terminal est un ' :' alors on commence un R ;
  - si ce terminal est un 'i' alors on doit continuer l'analyse du P.

$\Rightarrow$ Ceci est caractéristique d'une grammaire LR(2).

Synthèse générale sur l'analyse syntaxique

Savoir et savoir-faire

- À l'issue de cette première partie du cours de compilation sur l'analyse syntaxique, vous devez être en mesure de :
  - spécifier une grammaire ;
  - appliquer un algorithme d'analyse syntaxique général (Earley, Unger...) ;
  - déterminer si une grammaire est LL(1), si elle est LR(1), ... ;
  - calculer les tables associées à ces algorithmes ;
  - résoudre les conflits LL ou LR ;
  - utiliser `Menhir` pour produire l'analyseur syntaxique d'une grammaire LR(1).

Les générateurs d'analyseurs syntaxiques

- Dans ce cours, nous utilisons Menhir.
- Pour chaque langage de programmation, on trouve un outil similaire :
  - GNU Bison : C, C++, ...
  - SableCC : Java.
  - Wisent : Python.
  - Happy : Haskell.
- Il existe aussi des générateurs d'analyseurs syntaxiques qui s'appuient sur LL :
  - ANTLR : C, C++, C#, Objective-C, Java, Python,...
  - Coco/R : C, C++, C#, Objective-C, Java, Python,...

Les algorithmes non vus en cours

- Il existe de nombreux raffinements des algorithmes vus en cours.
- GLR généralise LR en admettant une forme de non-déterminisme qui permet de traiter efficacement l'ensemble des grammaires hors-contextes non ambiguës.
- Il est implémenté dans les générateurs d'analyseurs syntaxiques :
  - GNU Bison
  - Elkhound
  - Dypgen