Grammaires
Analyse grammaticale

Didier Rémy

Octobre 2000

http://cristal.inria.fr/~remy/poly/compil/8/

http://w3.edu.polytechnique.fr/profs/informatique/Didier.Remy/compil/8/

Grammaires algébriques (context-free)

Une grammaire permet une représentation finie d'un langage (éventuellement infini). Noam Chomsky (c.f. author of Manufacturing Conscent) a inventé la notion de grammaire formelle.

Une grammaire algébrique (ou context free) G est un quadruplet (Σ, V, S, P) où :

- Σ est l'alphabet des terminaux (lexèmes ou tokens),
- V est l'alphabet des non-terminaux (disjoint de Σ),
- $S \in V$ est l'unique axiome.
- P est un ensemble fini de règles de production, de la forme $\alpha \to \beta$ avec $\alpha \in V$, et $\beta \in (\Sigma \cup V)^*$

Le langage $L(G)$ associé à la grammaire G est la fermeture transitive du singleton $\{S\}$ par la règle d'induction :

$$u\alpha v \in L \wedge \alpha \to \beta \in P \Longrightarrow u\beta v \in L$$

Dérivations

Pour montrer qu'un mot w est dans le langage L il faut donc exhiber une suite finie d'applications de règles de production partant de S et aboutissant à w, appelée une dérivation.

On peut représenter une dérivation par

$$\boxed{S = w_0 \xrightarrow{P_1} w_1 \xrightarrow{P_2} \dots \xrightarrow{P_n} w_n = w}$$

où $w_k = u_k\beta_k v_k = u_{k+1}\alpha_{k+1}v_{k+1}$ et $P_k = \alpha_k \longrightarrow \beta_k$.

On laisse P_i implicite lorsqu'il n'y a pas d'ambiguité, ou on écrit :

$$\begin{aligned} S = \quad & u_1\underline{\alpha_1}v_1 \longrightarrow u_1\overline{\beta_1}v_1 = u_2\underline{\alpha_2}v_2 \longrightarrow u_2\overline{\beta_2}v_2 = \quad \dots \\ & u_n\underline{\alpha_n}v_n \longrightarrow u_n\overline{\beta_n}v_n \end{aligned}$$

Exemples : expressions arithmétiques

Les terminaux (lexèmes) sont `NUM`, `ID`, `PLUS` et `MINUS`.

$$S \to E \quad E \to \texttt{NUM} \quad E \to \texttt{ID} \qquad E \to E \ \texttt{PLUS} \ E \qquad E \to E \ \texttt{MINUS} \ E$$

Pour reconnaître l'expression `1 - 1 + x`, i.e. après analyse lexicale, `NUM MINUS NUM PLUS ID`, il existe de nombreuses dérivations possibles. L'ensemble de ces dérivations peut être représenté par un ensemble d'arbres syntaxiques (ici deux) :

$$S \to E \to \left\{ \begin{array}{l} E \to \left\{ \begin{array}{l} E \to \texttt{NUM} \\ \texttt{MINUS} \\ E \to \texttt{NUM} \end{array} \right. \\ \texttt{PLUS} \\ E \to \texttt{ID} \end{array} \right. \qquad S \to E \to \left\{ \begin{array}{l} E \to \texttt{NUM} \\ \texttt{MINUS} \\ E \to \left\{ \begin{array}{l} E \to \texttt{NUM} \\ \texttt{PLUS} \\ E \to \texttt{ID} \end{array} \right. \end{array} \right.$$

Chaque arbre syntaxique représente lui même un ensemble de dérivations obtenues en fixant un ordre d'application des règles.

Dérivations

Par exemple, la dérivation la plus à gauche (resp. à droite) de l'arbre syntaxique de gauche est obtenue en réduisant toujours le non-terminal le plus à gauche (resp. à droite).

$$
\begin{array}{l}
\underbrace{S} \\
\overbrace{\underbrace{E}} \\
\overbrace{\underbrace{E}\ \mathtt{PLUS}\ E} \\
\overbrace{\underbrace{E}\ \mathtt{MINUS}\ E}\ \mathtt{PLUS}\ E \\
\overbrace{\mathtt{NUM}}\ \mathtt{MINUS}\ \underbrace{E}\ \mathtt{PLUS}\ E \\
\mathtt{NUM}\ \mathtt{MINUS}\ \overbrace{\mathtt{NUM}}\ \mathtt{PLUS}\ \underbrace{E} \\
\mathtt{NUM}\ \mathtt{MINUS}\ \mathtt{NUM}\ \mathtt{PLUS}\ \overbrace{\mathtt{ID}}
\end{array}
\qquad \text{resp.} \qquad
\begin{array}{l}
\underbrace{S} \\
\overbrace{\underbrace{E}} \\
\overbrace{E\ \mathtt{PLUS}\ \underbrace{E}} \\
\underbrace{E}\ \mathtt{PLUS}\ \overbrace{\mathtt{ID}} \\
\overbrace{E\ \mathtt{MINUS}\ \underbrace{E}}\ \mathtt{PLUS}\ \mathtt{ID} \\
\underbrace{E}\ \mathtt{MINUS}\ \overbrace{\mathtt{NUM}}\ \mathtt{PLUS}\ \mathtt{ID} \\
\overbrace{\mathtt{NUM}}\ \mathtt{MINUS}\ \mathtt{NUM}\ \mathtt{PLUS}\ \mathtt{ID}
\end{array}
$$

Parmi l'ensemble des dérivations d'un arbre syntaxique, il en existe toujours une la plus à gauche (resp. la plus à droite).

Grammaires régulières

Les règles de production sont de la forme $\alpha \to \beta$ où $\beta \in \{\epsilon\} \cup \Sigma V$.

Langage régulier Le langage décrit par une grammaire régulière peut être reconnu par un automate fini, et réciproquement.

- À (Σ, V, S, P), on associe $(V \cup \{F\}, \Sigma, \delta, S, \{F\})$ où $q' \in \delta(q, a)$ si $q \to aq'$ et $q \in F$ si $q \to \epsilon$.
- À un automate fini déterministe $(Q, \Sigma, \delta, q_0, F)$ on associe la grammaire (Σ, Q, q_0, P) où P contient $q \to aq'$ si $q' \in \delta(q, a)$ et $q \to \epsilon$ si $q' \in F$.

Exemple à l'expression régulière $(\mathtt{ab})*$ correspond la grammaire composée des lexèmes `a` et `b` et des règles :

$$S \longrightarrow \qquad\qquad S \longrightarrow \mathtt{a}I \qquad\qquad I \longrightarrow \mathtt{b}S$$

Un exemple de dérivation :

$$\underline{S} \longrightarrow \overline{\mathtt{a}\underline{I}} \longrightarrow \mathtt{a}\overline{\mathtt{b}\underline{S}} \longrightarrow \mathtt{ab}\overline{\mathtt{a}\underline{I}} \longrightarrow \mathtt{aba}\overline{\mathtt{b}\underline{S}} \longrightarrow \mathtt{abab}$$

Grammaires ambigües

Dérivations et arbres de dérivations

- En général, il existe de nombreuses dérivations d'un même arbre syntaxique, mais ce n'est pas gênant.
- Lorsqu'il existe plusieurs arbres syntaxiques dérivables pour une même expression, la grammaire est dite ambigüe.

Élimination des ambiguités

L'existence de plusieurs arbres syntaxiques pose un problème, car il faudra faire un choix dans la construction de l'arbre de syntaxe abstraite qui va en général influer sur la sémantique (par exemple, la soustraction des expressions arithmétiques n'est pas associative).

Heureusement, il est souvent possible d'éliminer les ambiguités, ce qui consiste à ré-écrire certaines règle de façon à obtenir une grammaire non-ambigüe reconnaissant le même langage.

Élimination des ambiguïtés

Typiquement, on peut séparer un terminal E en deux terminaux E et T.

$$S \to E \qquad\qquad E \to T$$
$$E \to E \ \texttt{PLUS}\ T \qquad\qquad T \to \texttt{NUM}$$
$$E \to E \ \texttt{MINUS}\ T \qquad\qquad T \to \texttt{ID}$$

Fin de fichier/phrase on ajoute un lexème `EOF` et

– on change les règles $S \to \alpha$ en $S \to \alpha\ \texttt{EOF}$, ou bien

– on introduit un symbole S' et on prend $S' \to S\ \texttt{EOF}$ pour axiome (S devient un non-terminal).

Grammaires prédictives LL(1)

Simples, elles permettent l'écriture manuelle d'analyseurs.

Pour $\beta \in (\Sigma \cup V)^*$, on définit l'ensemble $first(\beta)$ des lexèmes qui peuvent commencer une dérivation de β. Formellement,

$$first(\beta) = \{a \in \Sigma \mid \exists \gamma \in (\Sigma \cup V)^*, \beta \to^* a\gamma\}$$

Une grammaire est prédictive (i.e. LL(1)), si pour tout non terminal $\alpha \in V$, pour toutes productions distinctes $\alpha \to \beta$ et $\alpha \to \beta'$ alors $first(\beta) \cap first(\beta') = \emptyset$.

Généralisation possible à LL(k) mais peu utilisée (inefficacité)

Exemple : la grammaire des expressions arithmétiques donnée ci-dessus n'est pas prédictive, car :

$$first(E \texttt{ PLUS } T) = \{\texttt{NUM}, \texttt{ID}\} = first(E \texttt{ MINUS } T) = \{\texttt{NUM}, \texttt{ID}\}$$

Élimination de la récursion gauche

La récursion gauche est une source de d'ambiguité (conflit). On peut l'éliminer en par tansformation (réécriture) en des règles équivalentes, sans récursion gauche.

Exemple $\left(\begin{array}{l} E \rightarrow E \ \texttt{PLUS} \ T \\ E \rightarrow E \ \texttt{MINUS} \ T \\ E \rightarrow T \end{array}\right) \Longrightarrow \left(\begin{array}{l} E \rightarrow T \ E' \\ E' \rightarrow \texttt{PLUS} \ T \ E' \\ E' \rightarrow \texttt{MINUS} \ T \ E' \\ E' \rightarrow \epsilon \end{array}\right)$

Cas général $\left(\begin{array}{l} X \rightarrow X \ \beta_1 \\ X \rightarrow X \ \beta_2 \\ X \rightarrow \gamma_1 \\ X \rightarrow \gamma_2 \end{array}\right) \Longrightarrow \left(\begin{array}{l} X \rightarrow \gamma_1 \ X' \\ X \rightarrow \gamma_2 \ X' \\ X' \rightarrow \beta_1 \ X' \\ X' \rightarrow \beta_2 \ X' \\ X' \rightarrow \epsilon \end{array}\right)$

Factorisation gauche

Elle se traite de façon similaire.

Exemple du if-then-else

$$\left(\begin{array}{l} E \rightarrow \texttt{IF}\; B\; \texttt{THEN}\; E\; \texttt{ELSE}\; E \\ E \rightarrow \texttt{IF}\; B\; \texttt{THEN}\; E \end{array}\right) \Longrightarrow \left(\begin{array}{l} E \rightarrow \texttt{IF}\; B\; \texttt{THEN}\; E\; X \\ X \rightarrow \texttt{ELSE}\; E \\ X \rightarrow \epsilon \end{array}\right)$$

Analyseur pour les grammaires prédictives

Pour chaque non-terminal $\alpha \in V$, on peut décider de la production à appliquer en fonction du prochain lexème $a \in \Sigma$: c'est l'unique règle $\alpha \to \beta$ avec $a \in first(\beta)$ (puisque deux règles dérivant α ne peuvent pas commencer par le même lexème).

On peut alors construire une table à deux dimensions $V \times \Sigma$ à valeurs dans P.

Exemple pour les expressions arithmétiques (après élimination de la récursion gauche).

	`PLUS`	`MINUS`	`ID`	`NUM`	`EOF`
S			E `EOF`	E `EOF`	
E			$T\ E'$	$T\ E'$	
E'	`PLUS` $T\ E'$	`MINUS` $T\ E'$			ϵ
T			`ID`	`NUM`	

Programme d'analyse

La table ci-dessus peut être implémentée directement en interprétant le tableau ou bien par filtrage. (Il faudrait aussi ajouter un rattrapage d'erreur dans les cases vides du tableau) :

Fichier expr.ml

```
type lexème = PLUS | MINUS | ID | NUM | EOF;;
let  buf = ref [];;
let  lit_lexème () =
  match !buf with h::t -> buf:= t; h | [] ->  EOF;;
let  lexème = ref  (EOF : lexème);;
let  avance() = lexème := lit_lexème();;
let  accepte a =
  if  ! lexème = a then avance() else failwith  "accepte"
```

```
let rec _S() =
  match !lexème with
  | ID | NUM -> _E(); accepte EOF
and _E() =
  match !lexème with
  | ID | NUM -> _T(); _E'()
and _E'() =
  match !lexème with
  | PLUS -> accepte PLUS; _T(); _E'()
  | MINUS -> accepte MINUS; _T(); _E'()
  | EOF -> ()
and _T() =
  match !lexème with
  | ID -> accepte ID;
  | NUM -> accepte NUM

let parse l = buf := l; avance(); _S();;
```

Langages LR(1)

Left-to-right parse, **R**ight-most derivation **(1**-token lookahead**)**

Les grammaires LL(1) permettent l'écriture facile donc manuelle d'analyseurs, mais l'écriture d'une grammaire LL(1) pour un langage est contraignante et revient à une forme de “compilation manuelle”. De plus, celle-ci n'est pas toujours possible.

Les grammaires LR sont plus générales : une grammaire LR(1) est aussi LL(1) donc la classe des langages reconnus par une grammaire LR(1) contient celle des langages LL(1).

Les grammaires LR(1) sont également plus souples (moins contraignantes) dans l'écriture des règles.

Principe

Leur moteur utilise une pile auxiliaire. Le flux de lexème est lu en consultant au plus 1 lexème non empilé. Il faut alors pouvoir décider d'une action à effectuer (en fonction de la pile et du prochain lexème) :

- $shift$, i.e. consommer et empiler un lexème, ou
- $reduce$, i.e. réduire une règle. Cela revient à appliquer une règle de production sur le sommet (partie droite) de la pile.

Si pour une même configuration de pile et du prochain lexème, plusieurs axiomes sont possibles, alors la grammaire n'est pas LR(1) et dite ambigüe.

Généralisation possible en regardant les k prochains lexèmes (grammaires LR(k)) mais peu utilisée (inefficacité).

Trace d'exécution

Pile	Flux de lexèmes	Action
	NUM MINUS NUM PLUS ID EOF	*shift*
NUM	MINUS NUM PLUS ID EOF	*reduce*
T	MINUS NUM PLUS ID EOF	*reduce*
E	MINUS NUM PLUS ID EOF	*shift*
E MINUS	NUM PLUS ID EOF	*shift*
E MINUS NUM	PLUS ID EOF	*reduce*
E MINUS T	PLUS ID EOF	*reduce*
E	PLUS ID EOF	*shift*
E PLUS	ID EOF	*shift*
E PLUS ID	EOF	*reduce*
E PLUS T	EOF	*reduce*
E	EOF	*shift*
E EOF		*reduce*
S		

Reconnaissance par un automate à pile

Une grammaire LR(1) est reconnaissable par un automate à pile qui mécanise le calcul précédent.

Description

- Les éléments sur la pile sont des paires (X_i, s_i) d'un état s_i et d'un lexème $X \in \Sigma \cup V$ sauf le fond de pile qui est l'état initial s_0.
- L'automate est déterminé par deux fonctions partiellement définies :
  - $goto(s, X)$ pour $X \in \Sigma \cup V$.
  - $action(s, a)$ qui peut prendre les valeurs $shift$ ou $reduce(P)$ où P est une règle de production

Transitions

Dans une configuration de pile $s_0(X_1, s_1)...(X_m, s_m)$ (sommet à droite), l'automate consulte le lexème suivant a et par cas sur la valeur de $action(s_m, a)$ effectue l'une des deux actions :

- $shift$: le prochain lexème est consommé et $(a, goto(s_m, a))$ est empilée
- $reduce(X \to \alpha)$: k éléments où k est la longueur de α sont dépilés et $(X, goto(s_{m-k}, X))$ est empilé.

Lorsque $action(s_m, a_i)$ est indéfinie, l'automate s'arrête, avec succès si la pile est réduite à l'axiome, avec échec sinon.

Construction des états

Un état I est composé d'un ensemble de configurations C.

Une configuration C est une paire composée :

1. d'une règle de production pointée $\alpha \to \beta.\gamma$. i.e. $\alpha \to \beta\gamma$ est une règle de production et β est le sommet de pile.
2. du prochain lexème possible $a \in \Sigma$.

La fermeture d'un ensemble de configurations I est le plus petit ensemble contenant I et satisfaisant

$$((X \to \alpha \cdot Y\beta, a) \in I \wedge Y \to \gamma \in G \wedge b \in first(\beta a)) \Longrightarrow (Y \to \cdot\gamma, b) \in I$$

La transition d'une configuration I par un lexème $X \in \Sigma \cup V$ est

$$goto(I, Y) = fermeture(\{(X \to \alpha Y \cdot \beta, a) \mid (X \to \alpha \cdot Y\beta, a) \in I\})$$

Construction des tables

L'état initial est la fermeture de l'axiome. Les états sont les ensembles de configurations fermées non vides atteignables par une suite de transitions arbitraires à partir de l'état initial.

Les actions sont de 4 types :

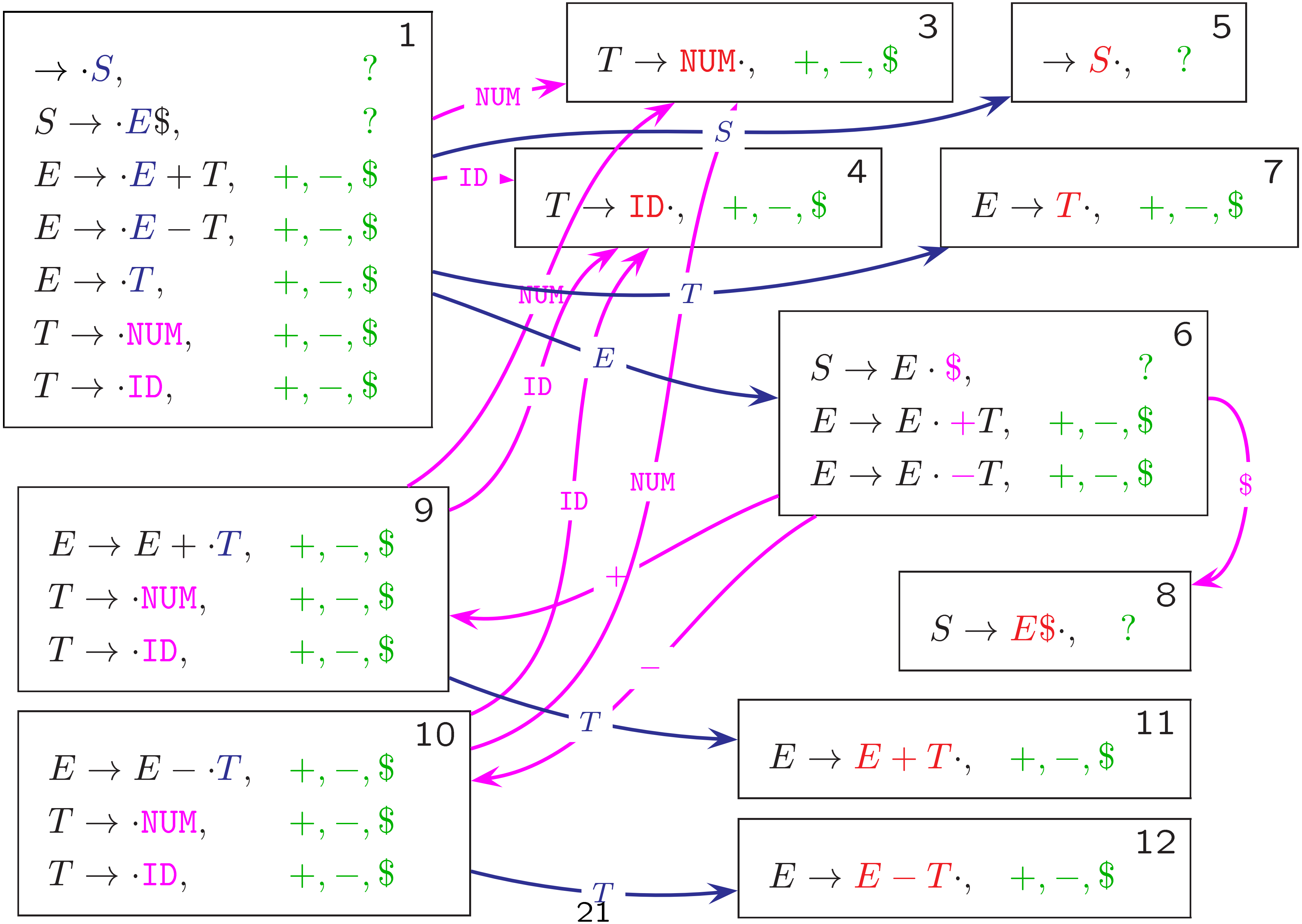
1. Si $(X \to \alpha \cdot a\beta, b) \in I$ et $goto(I, a) = J$, alors $action(I, a) = shift$ (on peut représenter simultanément $goto(I, a)$ en écrivant $shift(J)$)
2. Si $(X \to \alpha\cdot, a) \in I$, alors $action(I, a) = reduce(X \to \alpha)$.

   Sauf, si $X \to \alpha$ est l'axiome, alors $action(I, a) = accept$.

(Il reste à représenter la table $goto(I, Y)$ pour $Y \in V$.)

Enfin, la grammaire n'est pas LR(1) s'il y a des conflits, i.e. plusieurs valeurs possibles pour $action(I, a)$.

Pour les autres cases, $action(I, a) = echec$.


1
$\to \cdot S,\quad ?$
$S \to \cdot E\$,\quad ?$
$E \to \cdot E + T,\quad +, -, \$$
$E \to \cdot E - T,\quad +, -, \$$
$E \to \cdot T,\quad +, -, \$$
$T \to \cdot \mathtt{NUM},\quad +, -, \$$
$T \to \cdot \mathtt{ID},\quad +, -, \$$
3
$T \to \mathtt{NUM}\cdot,\quad +, -, \$$
4
$T \to \mathtt{ID}\cdot,\quad +, -, \$$
5
$\to S\cdot,\quad ?$
7
$E \to T\cdot,\quad +, -, \$$
6
$S \to E \cdot \$,\quad ?$
$E \to E \cdot +T,\quad +, -, \$$
$E \to E \cdot -T,\quad +, -, \$$
8
$S \to E\$\cdot,\quad ?$
9
$E \to E + \cdot T,\quad +, -, \$$
$T \to \cdot \mathtt{NUM},\quad +, -, \$$
$T \to \cdot \mathtt{ID},\quad +, -, \$$
10
$E \to E - \cdot T,\quad +, -, \$$
$T \to \cdot \mathtt{NUM},\quad +, -, \$$
$T \to \cdot \mathtt{ID},\quad +, -, \$$
11
$E \to E + T\cdot,\quad +, -, \$$
12
$E \to E - T\cdot,\quad +, -, \$$
NUM
ID
S
T
E
NUM
ID
ID
NUM
$\$$
$+$
$-$
T
T

LR(0) v.s. LR(1)

Dans la construction précédente, l'ensemble des lexèmes prossibles après le déclanchement d'une règle est presque toujours $\texttt{EOF}$, $\texttt{PLUS}$, $\texttt{MINUS}$, et ne sert jamais à déambigüer deux états.

En effet, l'exemple considérer est aussi LALR(0).

Exercice 1 *On considère la grammaire suivante*

$$S \to E\ \texttt{EOF} \qquad E \to T\ \texttt{PLUS}\ E \qquad E \to T \qquad T \to \texttt{ID}$$

Vérifier que la grammaire n'est pas LR(0). (On construit les états comme pour une grammaire LR(1), mais dans lesquelles les configurations ne comporte que les premières composante (une règle pointée. Il suffit alors de vérifier qu'il existe deux transition possibles avec une même étiquette).

Vérifier que la grammaire est LR(1). $\square$

Construction de la table de l'automate

On construit un tableau de dimension 2 avec les états en ordonnées et les éléments de lettre de l'alphabet $\Sigma \cup V$ en abscisse.

On remplit le tableau ligne par ligne. Pour chaque état I,

– Pour chaque flèche étiquettée par un terminal $I \rightarrow^{a} J$, on associe une action $shift(J)$ dans la colonne a.
– Pour chaque flèche étiquettée par un non-terminal $I \rightarrow^{a} J$, on associe on mémorise la valeur J de $goto(I, a)$ dans la case a. (pseudo-action $shift(J)$).

Enfin, pour chaque règle pointée à la fin (en rouge) on associe une action réduce dans la case associé à chacun des terminaux suivant possibles (en vert).

Tables

I	$+$	$-$	NUM	ID	\$	S	E	T
1			$s3$	$s4$		5	6	7
3	$r5$	$r5$			$r5$			
4	$r6$	$r6$			$r6$			
5								
6	$s9$	$s10$			$s8$			
7	$r4$	$r4$			$r4$			
8	$r1$	$r1$			$r1$			
9			$s3$	$s4$				11
10			$s3$	$s4$				12
11	$r2$	$r2$			$r2$			
12	$r3$	$r3$			$r3$			

r	règle
1	$S \to E\$$
2	$E \to E + T$
3	$E \to E - T$
4	$E \to T$
5	$T \to \texttt{NUM}$
6	$T \to \texttt{ID}$

Pile	Flux de lexèmes	Action
$_1$	NUM − NUM + ID$	*shift*
$_1\mathtt{NUM}_3$	− NUM + ID$	*reduce*
$_1 T_7$	− NUM + ID$	*reduce*
$_1 E_6$	− NUM + ID$	*shift*
$_1 E_6 -$	NUM + ID$	*shift*
$_1 E_6 -_{10} \mathtt{NUM}_3$	+ ID$	*reduce*
$_1 E_6 -_{10} T_{12}$	+ ID$	*reduce*
$_1 E_6$	+ ID$	*shift*
$_1 E_6 +_9$	ID$	*shift*
$_1 E_6 +_9 \mathtt{ID}_4$	$	*reduce*
$_1 E_6 +_9 T_{11}$	$	*reduce*
$_1 E_6$	$	*shift*
$_1 E\$_8$		*reduce*
$_1 S_5$		

Correction

Exercice 2 (Correction) *Vérifier que lorsque l'algorithme ci-dessus trouve une solution, alors il est possible de construire une dérivation.* $\square$

Exercice 3 (Complétude) *Vérifier que lorsque l'algorithme ne trouve pas de solution, alors il n'existe pas de dérivation.* $\square$

Conflits

Un conflit se produit lorsqu'il existe plusieurs une configuration (I, a) pour laquelle $action(I, a)$ admet plusieurs valeurs possibles.

On parle de conflit :

- $reduce - reduce$ lorsque $(X \to \alpha\cdot, a) \in I$ et $(Y \to \beta\cdot, a) \in I$.
  Ces conflits sont graves et correspondent généralement à une forte ambiguïté dans la grammaire. Il faut réécrire la grammaire.
- $shift - reduce$ lorsque $(X \to \alpha \cdot a\gamma, b) \in I$ et $(Y \to \beta\cdot, a) \in I$.
  Ces conflits sont moins graves, et correspondent souvent à des priorités qu'il faut rendre explicites, soit par réécriture, soit par ajout de règles de priorité.

(Par construction, les conflits $shift - shift$ ne sont pas possibles)

Règles d'associativité

La grammaire suivante contient un conflit $shift/\,reduce$

$$1 : E \to E + E \qquad\qquad 2 : E \to \mathtt{NUM}$$

En effet, la séquence NUM PLUS NUM PLUS NUM est reconnue par

$$shift, reduce\,2, shift, shift, reduce\,2, \begin{cases} \underline{reduce\,1}, shift, reduce\,2, reduce\,1 \\ \underline{shift}, reduce\,2, reduce\,1, reduce\,1 \end{cases}$$

La première solution reconnaît $(E + E) + E$, l'autre $E + (E + E)$.

On peut réécrire la grammaire pour forcer l'une ou l'autre règle, mais il est aussi possible et préférable (bien que sortant du formalisme LR(1)) de choisir entre les deux actions possibles par des règles d'associativité et de priorité.

(Les règles de priorité s'appliquent localement pour ordonner les actions possibles à partir d'un état ambigü.)

Règles de priorité

Selon que $+$ est déclaré associative à gauche ou à droite, l'action retenue sera $reduce$ ou $shift$.

On peut également déclarer des lexèmes plus prioritaires que d'autres, ou des règles plus prioritaires que d'autres.

Grammaires LALR(1)

Look **A**head **LR(1)**

Les grammaires LALR(1) sont une restriction des grammaires LR(1) afin d'obtenir des tables plus petites. Le langage reconnu est un peu plus restrictif, mais ce n'est pas une gêne en pratique.

Dans une grammaire LALR(1) les états qui ne sont distingués que par le lexème suivant (mais qui ont exactement le même ensemble de règles de production) sont identifiés (avant construction des tables).

Yacc, Ocamlyacc, Bison, etc., sont des compilateurs de grammaires LALR(1).

Ocamlyacc

Fichier foo.mly

```
%{
  (* prelude : code Ocaml reproduit verbatim *)
%}
  /* declarations yacc */
%%
  /* règles yacc */
%%
  (* postlude : code Ocaml reproduit verbatim *)
```

Pour compiler la grammaire

```
ocamlyacc -v foo.mly        # produit foo.ml et foo.output
ocamlc foo.ml
```

L'option `-v` copie les tables dans un fichier `foo.output` (utile pour la mise au point)

Exemple

exp.mly

```
%token EOP PLUS MINUS
%token <int> NUM
%token <string> ID
%start _S
%type <string> _S
%%
_S :  _E EOP              { ($1 : string) }
_E :  _E PLUS _T          { "Plus ("^$1^", "^$3^")" }
    | _E MINUS _T         { "Minus ("^$1^", "^$3^")" }
    | _T                  { $1 }
_T : NUM                  { "Num "^(string_of_int $1) }
    | ID                  { "Id "^$1 }
%%
```

Indication des priorités

Dans la partie déclaration :

```
%noassoc lower
%left PLUS MINUS
%left TIMES DIV
%right FOO
%noassoc BAR
%noassoc higher
```

Par défaut, la priorité d'une règle est celle de son dernier lexème.

La directive %**prec** peut être utilisée à la fin d'une règle pour indiquer la priorité à donner à cette règle.

Mise en œuvre

Il faut combiner le parseur avec un lexeur. voir `lexp.mll`.

Le parseur doit être compilé en premier, car il produit simultanément la définition du type des `token` utilisé par le lexeur (le nom de l'identificateur token est fixé par convention).

Le programme principal appelle le parseur qui appelle le lexeur comme ci-dessus.

Report d'erreur

Il existe un mécanisme de report d'erreur dans yacc, mais qui n'est pas facile à mettre en œuvre. En cas d'erreur, il est simple et souvent suffisant de signaler le dernier lexème lu et son emplacement (voir le compilateur Pseudo-Pascal main.ml)

Exercices

Exercice 4 (Langage de parenthèses) *Écrire un parseur (et un lexeur) pour le langage de parenthèses.*

Comparer avec la reconnaissance des parenthèses directement dans le lexeur. Quelle approche est-elle préférable ? *Réponse*

□

Exercice 5 (Expressions arithmétiques) *Écrire un parseur (et un lexeur) pour reconnaître les expressions arithmétiques sans variables. Retourner l'arbre de syntaxe abstraite.*

En déduire une calculette en remplaçant la génération de l'arbre par le calcul.

Pour obtenir une Super calculette, on pourra

- *Ajouter des mémoires (set n et get n).*
- *Ajouter la comparaison et les expressions conditionnelles.*

□

Travaux dirigés

Exercice 6 (Pseudo Pascal) *Écrire un lexeur pour le langage Pseudo Pascal. (On réutilisera le lexeur de cet exercice).*

Un exemple de programme Pseudo Pascal est fact.p.

La syntaxe programmes est donnée dans une forme BNF (Backus Normal Form) que l'on peut lire comme des règles de production, une par ligne (le symbole $|$ remplace un saut à la ligne).

La notation $u\,s\ldots u$ (resp. $u\,s\ldots u\,s$) représente une séquence possiblement vide de u séparés par des s (resp. et terminée par un s).

Programmes

$p ::=$ `program` $v; \ldots v;\ d; \ldots d;\ i\ ;;$		*Programme*

Déclarations

$v ::=$ `var` $x : t$	*Décl. de variable*
$d ::=$ `function` $x($ `var` $x : t, \ldots$ `var` $x : t\ ) : t;\ v; \ldots v;\ i$	*Décl. de fonction*
`procedure` $x($ `var` $x : t, \ldots$ `var` $x : t\ );\ v; \ldots v;\ i$	*Décl. de procédure*

Instructions

$i ::= x := e$	*Affectation*
`begin` $i; \ldots i$ `end`	*Séquence*
`if` e `then` i `else` i	*Conditionnelle*
$x(\ e, \ldots e\ )$	*Appel de procédure*
`read` (x)	*Lecture*
`write` (e)	*Écriture*
`writeln` (e)	*Écriture avec newline*
$e[e] := e$	*Affectation de tableau*

Exercice (Suite)

Expressions

e	$::=$	(e)	*Expression parenthèsée*
		x	*Variables*
		n	*Entiers*
		$\texttt{true} \mid \texttt{false}$	*Booléens*
		$x(\, e, \ldots e\,)$	*Appel de fonction*
		$\texttt{alloc}\,(e : t)$	*Allocation*
		$e\ bin\ e$	*Opération binaire*
		$e[e]$	*Accès dans un tableau*

Opérations

bin	$::=$	$+ \mid - \mid \times \mid /$	*Arithmétiques*
		$< \mid > \mid <= \mid >= \mid = \mid <>$	*Comparaison*

Types

t	$::=$	`integer`
		`boolean`
		`array of` t

Exercice (Suite)

Dans un premier temps, on pourra laisser les actions vides (on retour unit). Ensuite, on produira un arbre de syntaxe abstraite défini par le type concret program *du fichier d'interface pp.mli.*

Réponse

Enfin, on pourra compléter le programme avec un imprimeur qui imprime le programme (arbre de syntaxe abstraire reconnu) dans la sortie standard. En particulier, cela permet de vérifier que le programme a été compris correctememt. *Réponse*

□