923. Analyse lexicale et syntaxique. Applications

→ Compilers. Aho Sethi Ullman

I_ COMPILATION

Un compilateur est un programme qui lit le code d'un autre programme dans un langage (langage source) et le traduit dans un autre langage (langage cible). Il permet en plus de détecter des erreurs dans le programme source.

Pour cela, dans une première partie d'analyse, un compilateur rompt le programme source en différentes composantes (phase d'analyse lexicale), puis leur impose une structure grammaticale afin d'obtenir une représentation intermédiaire du programme (phase d'analyse syntaxique). enfin, il vérifie le sens de la représentation (phase d'analyse sémantique). En même temps, le compilateur crée une table des symboles pour stocker les informations obtenues.

II_ ANALYSE LEXICALE

A_ Vocabulaire

Un analyseur lexical lit donc les caractères du programme source. Il les regroupe en chaîne de caractères ayant un sens, puis renvoie la composante associée, en ajoutant si besoin une entrée dans la table de symboles.

Def 1: Les chaînes de caractères ayant un sens sont les lexèmes, les composantes sont les unités lexicales.

Ex 2: double = nombre x 2 ← programme source
"double" "=" "nombre" "x" "2" ← décomposition
<id, 1> <=> <id, 2> <*> <int, 2> ← suite d'unités lexicales
avec la première entrée de la table de symboles qui est "double" et la seconde "nombre".

L'analyseur lexical supprime donc au passage les caractères inutiles pour la suite comme les blancs ou commentaires

B_ Principe

De manière générale, les lexèmes peuvent être reconnus par des expressions rationnelles. Le motif associé à l'unité lexicale.

Ex 3. lexème 42, motif : $(-+\varepsilon)(chiffre)^+$ avec chiffre $=\{0,\dots,9\}$
unité lexicale : <int, 42>

A chaque unité lexicale et donc motif, on peut associer un automate fini déterministe. Si $\{p_i, 1 \leq i \leq n\}$ est l'ensemble des motifs, on obtient $\{A_i, 1 \leq i \leq n\}$

On peut alors construire un automate pour reconnaître tous les motifs, grâce à des ε-transitions (Figure 1)

C_ Résolution des conflits

n'intervient pas à la construction du compilateur

Pb 4: La déterminisation de l'automate obtenu est coûteuse.

Pb 5: Il faut savoir choisir le plus long préfixe ← en fait, fin de l'identificateur

Ex 6: 12 doit être reconnu comme <int, 12> et non <int, 1><int, 2>.

Pb 7: Certains lexèmes peuvent correspondre à plusieurs unités lexicales, notamment au niveau des identifiants et mots clés.
Il faut établir une priorité entre les unités lexicales

Ex 8: if doit être reconnu comme le mot clé <if> et non <id, i> où la table de caractères contient dans la ligne i "if".

D_ Applications et utilisation

Appl 9: Coloration syntaxique. Les programmes comme emacs colorent au fur et à mesure le code dans une couleur correspondant à l'unité lexicale.
On peut alors particulièrement bien voir le système de priorités:
els → else → elsev ...
<id, .> <else> <id, .>

Rq 10: Certains programmes font une analyse lexicale simplifiée, comme les calculatrices programmables pour lesquelles les mots clés doivent être choisis depuis un menu.

Outil 11: Lex est un constructeur d'analyseur lexical : en lui donnant en entrée les unités lexicales et motifs associés, ainsi que les priorités entre les unités lexicales, il crée un analyseur lexical.

Ex 12: "+" {return PLUS}
[0-9]+ {yylval.num = atoi(yytext); return INT}
extrait de programme lex permettant de reconnaître le symbole + et les entiers.

III - ANALYSE SYNTAXIQUE

A - Principe

Un analyseur syntaxique lit donc la suite d'unités lexicales renvoyées par l'analyseur lexical et vérifie qu'elle peut être générée par la grammaire du langage source. Si ce n'est pas le cas, il doit renvoyer l'erreur la plus précise possible. Si c'est le cas, il doit construire l'arbre de la dérivation.

Rq 13 : Les langages de programmation ont généralement une grammaire algébrique $G = (\Sigma, V, R, S)$.

Méthode (14) 1 : L'algorithme de Cocke-Younger-Kasami utilise la programmation dynamique pour décider si la suite d'unités lexicales est reconnue par la grammaire, après que la grammaire a été mise sous forme normale de Chomsky.

Der 1

Dev (15) 1 : On peut se contenter de mettre la grammaire en 2NF ; alors l'algorithme CYK se passe en temps $O(m^3|G|)$.
En pratique, ce n'est pas assez efficace. ← seul avantage : grammaire qcque.

Méthode (16) 2 : Analyse descendante : On construit l'arbre de dérivation à partir du symbole initial de G.

Méthode (17) 3. Analyse ascendante : On construit l'arbre de dérivation à partir des feuilles : les unités lexicales.

Rq 18 : Il faut choisir en plus, pour les deux dernières méthodes dans quel ordre lire la suite d'unités lexicales (gauche à droite ou droite à gauche). Dans la suite, de gauche à droite (L).

B - Analyse syntaxique descendante

Ex 19 : Pour la grammaire G $E \to E+T \mid T$; $T \to T \times F \mid F$, $F \to (E) \mid id$, dont l'axiome est E et de terminaux $\{+, \times, id, (,)\}$, la formule $id + id$ a pour arbre de dérivation la Figure 2.

Pb 20 : Deux problèmes se posent alors :
Quelle règle choisir ?
Quels terminaux de l'entrée choisir par rapport à la règle ?

Algo 21 : Pour chacun des non-terminaux A on définit ; pour w l'entrée
Desc_A()
- Choisir une règle $A \to X_1 \dots X_k$.
- Pour $i = 1$ à k
  - Si X_i est un non terminal, alors Desc_X_i()
  - Sinon si $X_i = w[p]$, alors $p \leftarrow p+1$
  - Sinon Erreur

Pb 22 : On peut avoir mal choisi une règle $A \to X_1 \dots X_k$, dans ce cas, il faut revenir en arrière et en choisir une autre.

Def 23 : Une grammaire est dite $LL(k)$, $k \geq 0$ s'il est possible de faire un analyseur syntaxique descendant qui lit k symboles en avance, et n'a jamais à faire de retour en arrière.

Def 24 : $Premier(\alpha) = \{a \in \Sigma, \exists u \in \Sigma^*, \alpha \to^* au\} \cup \{\varepsilon$ si $\alpha \to^* \varepsilon\}$. Il est défini pour $\alpha \in (\Sigma \cup V)^*$ et est calculable récursivement. Pour $A \in V$, $Suivant(A) = \{a \in \Sigma, \exists u, v \in (\Sigma \cup V)^*, S \to^* uAav\} \cup \{\varepsilon$ si $\exists u \in (\Sigma \cup V)^*$, $S \to^* uA\}$. Il est calculable récursivement à l'aide de premier.

Prop 25. Une grammaire est $LL(1)$ ssi pour tout A non terminal ayant 2 règles dont il est la partie gauche : $A \to \alpha \mid \beta$ de G, on a :
(i) $Premier(\alpha) \cap Premier(\beta) = \emptyset$
(ii) $\varepsilon \in Premier(\alpha) \Rightarrow Premier(\beta) \cap Suivant(A) = \emptyset$
$\varepsilon \in Premier(\beta) \Rightarrow Premier(\alpha) \cap Suivant(A) = \emptyset$

On peut alors regrouper les informations dans un tableau.

Def 26 : La table d'analyse syntaxique d'une grammaire $LL(1)$ est le tableau à double entrée $M[A, a]$ où $A \in V$ et $a \in \Sigma \cup \{\varepsilon\}$. $M[A, a]$ contient les α tq $A \to \alpha$ et ($a \in Premier(\alpha)$ (i)) ou ($\varepsilon \in Premier(\alpha)$ et $a \in Suivant(A)$ (ii)).

Prop 27 : Une grammaire est $LL(1)$ si chaque case ne contient qu'au plus une production, les cases vides correspondant aux erreurs.

Dev (28) 2 : G' : $E \to TE'$ d'axiome E est $LL(1)$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to \times FT' \mid \varepsilon$
$F \to (E) \mid id$

Dev 2

C. Analyse syntaxique ascendante

*Ex42:

L'analyse syntaxique ascendante correspond à un procédé de lecture (il faut ajouter des lettres avant d'appliquer une règle) et de réduction (la fin de ce qu'on a lu et réduit jusqu'à présent correspond exactement au membre droit d'une règle, qu'on remonte alors de droite à gauche).

On obtient alors une analyse LR : lecture L : de gauche à droite, R : on applique les règles de droite à gauche.

Pb 29. A nouveau, deux problèmes se posent :
Doit-on lire ou réduire ?
Si on choisit de réduire, quelle partie doit-on réduire ?

Def 30 : Une grammaire est dite $LR(k)$, $k \geq 0$ s'il est possible de faire un analyseur syntaxique ascendant qui lit k symboles en avance, et n'a jamais à faire de retour en arrière (pas d'erreurs).

La structure qui apparaît directement est l'automate à pile, où la pile contient l'état actuel des lectures-réductions.

Def 31 : Un item est un triplet $[A \to \alpha \cdot \beta]$ où $A \to \alpha\beta$ est une règle de la grammaire et $\alpha, \beta \in (\Sigma \cup V)^*$.

Dev (32) 3 : On peut alors construire l'automate fini déterministe des items, ainsi que l'automate à pile $LR(0)$ pour la grammaire G_0 : $S \to SS+$ d'axiome S et de terminaux $\{x, +\}$.
$S \to x$

Prop 33. Une grammaire est $LR(0)$ si l'automate à pile associé est déterministe.

Pour une grammaire $LR(0)$, on peut à nouveau construire un tableau regroupant les informations.

Def 34 : On augmente la grammaire avec une nouvelle règle $S' \to S$ avec $S' \notin \Sigma \cup V$ et S l'axiome de la grammaire.

Def 35 : À un ensemble d'items, on lui associe sa fermeture calculée récursivement : Pour tout item $[A \to \alpha \cdot B\beta]$ avec $A, B \in V$, $\alpha, \beta \in (\Sigma \cup V)^*$, on ajoute l'ensemble $\{[B \to \cdot \gamma], B \to \gamma$ une règle$\}$.

Def 36 : Le premier état est la fermeture de $\{[S' \to \cdot S]\}$. On construit récursivement les autres états : si q est un état, on obtient les états $q.u$, $u \in (\Sigma \cup V)$ où $q.u$ est la fermeture de $\{[A \to \alpha u \cdot \beta], [A \to \alpha \cdot u\beta] \in q\}$.

Def 37 : La table d'analyse syntaxique $LR(0)$ est alors le tableau $M[q, a]$ où q est un état et $a \in \Sigma \cup V$. Elle contient alors :
(i) pour $a \in \Sigma$: lecture(q') si on doit faire une étape de lecture, écrire q' sur la pile, et aller à l'état q'.
(ii) pour $a \in \Sigma$: réduction(i) si on doit faire une étape de réduction par la règle i : on doit alors dépiler autant d'éléments que la longueur de la règle droite ($A \to \alpha$) donc $|\alpha|$, puis aller en $M[q', A]$ où q' est le nouvel élément en haut de pile.
(iii) pour $a \in V$: q' si on doit écrire q' sur la pile et aller à l'état q'.

Prop 38 : Une grammaire est $LR(0)$ si lorsqu'on a une réduction dans la case $M[q, a]$ pour un $a \in \Sigma$, alors $M[q, b]$, $b \in \Sigma$ contient la même réduction. De plus chaque case ne contient qu'au plus une action.

D. Applications et utilisation

Appl 39 : Les navigateurs internet utilisent un analyseur syntaxique pour analyser le HTML des pages.

Outil 40 : Yacc est un constructeur d'analyseur syntaxique : en lui donnant en entrée la grammaire du langage source, il crée un analyseur syntaxique.

Ex 41 : La règle $E \to E + E$ se traduirait pour faire un programme yacc, par exemple en :
expr : expr '+' expr { $$ = node('+', $1, $3); }

(Ex 42 : Pour la grammaire de l'exemple 19, la formule id+id a pour arbre de dérivation la Figure 3)

Figure 1 Assemblage de l'automate pour l'analyse lexicale :

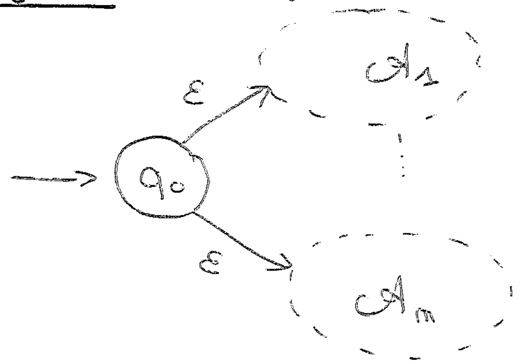

Figure 2 Analyse syntaxique descendante de id + id :

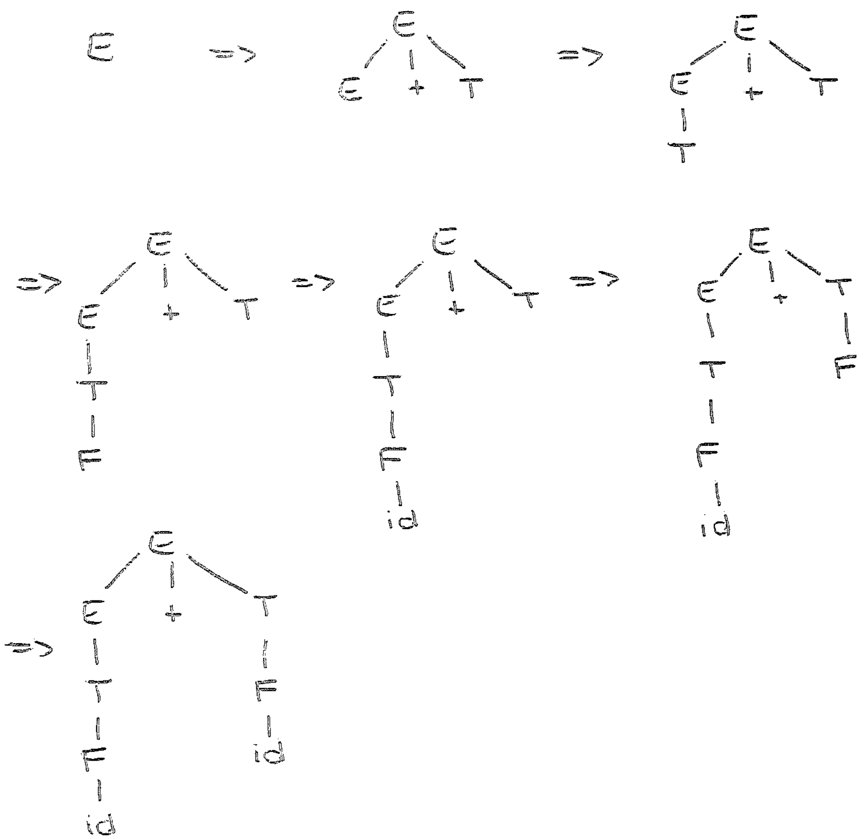

Figure 3 Analyse syntaxique ascendante de id + id