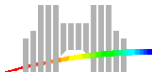
Master Informatique Fondamentale - M1 Compilation

Analyse Syntaxique

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr
perso.ens-lyon.fr/paul.feautrier

3 février 2007



Grammaires hors contexte

- Un ensemble $\mathcal{T}$ de symboles *terminaux*.
- Un ensemble $\mathcal{N}$ de symboles *non terminaux*.
- Un ensemble fini de *productions* :
  $T \longrightarrow w, T \in \mathcal{N}, w \in (\mathcal{N} \cup \mathcal{T})^*$.
- Un axiome $S \in \mathcal{N}$.
- Dérivation immédiate : si $T \longrightarrow w$ est une production, alors $uTv \longrightarrow uwv$.
- Dérivation : $u_0 \longrightarrow u_1 \longrightarrow \ldots u_n : u_0 \longrightarrow^* u_n$.
- Langage engendré : $\{w \in \mathcal{T}^* \mid S \longrightarrow^* w\}$.

Un exemple : les systèmes de parenthèses

- $\mathcal{T} = \{\underline{(}, \underline{)}, \underline{[}, \underline{]}, \underline{a}\}$.
- $\mathcal{N} = \{S\}$

$$S \to \underline{a} \qquad S \to SS$$
$$S \to \underline{(}S\underline{)} \qquad S \to \underline{[}S\underline{]}$$

$$S \to \underline{[}S\underline{]} \to \underline{[}SS\underline{]} \to \underline{[}a S\underline{]} \to \underline{[}a\underline{(}S\underline{)}\underline{]} \to \underline{[}a\underline{(}a\underline{)}\underline{]}$$

- Langage de Dick, ne peut être engendré par un automate fini.
- Le pouvoir expressif des grammaires hors contexte est supérieur à celui des grammaires rationelles.

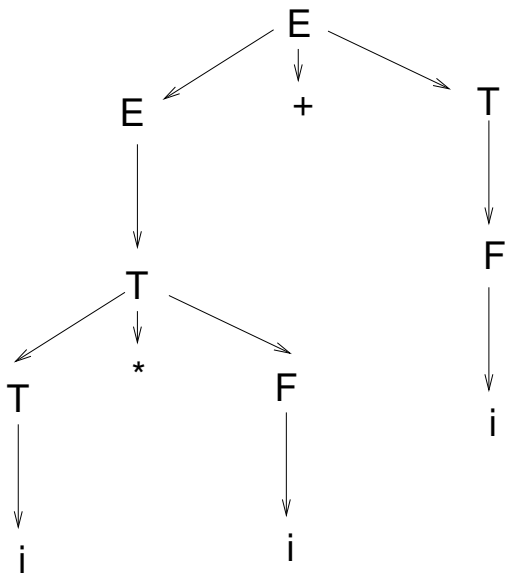
Exemple II : expressions arithmétiques

- $\mathcal{T} = \{i, e, +, -, *, /, (,)\}$
- $\mathcal{N} = \{E, T, F\}$

$$E \to E + T \quad E \to E - T \quad E \to T$$
$$T \to T * F \quad T \to T/F \quad E \to F$$
$$F \to (E) \quad F \to i \quad F \to e$$

$$E \to E+T \to E+F \to E+i \to T+i \to T*F+i \to T*i+i \to i*i+i$$

Arbre d’analyse syntaxique

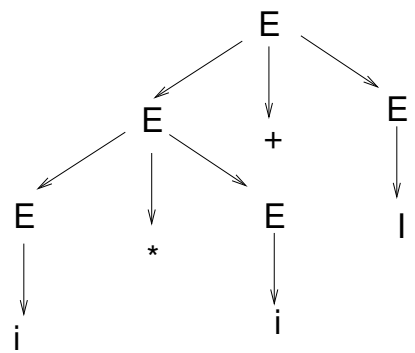


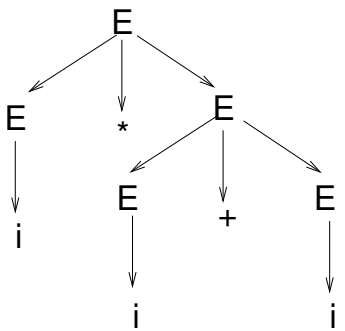
- Les noeuds sont les noms des productions (en général remplacés par le non terminal de gauche)
- Les feuilles sont les terminaux
- Pour chaque application d’une production on crée un noeud étiqueté par le nom de laproduction ayant pour fils les lettres du mot de droite.
- Le mot engendré est la frontière de l’arbre.

Ambiguïté, précédence

Une grammaire est ambiguë s'il existe plusieurs arbres ayant la même frontière.

$$
\begin{array}{rcl}
E & \rightarrow & E + E \\
 & | & E * E \\
 & | & (E) \\
 & | & i|e
\end{array}
$$





- On évite les grammaires ambiguës.
- On peut soit compliquer la grammaire, soit inciter l'analyseur à choisir l'une des analyses à l'aide de *règles de précédence*.
- Ici, $* > +$.

Classification

- Une grammaire est récursive à gauche si il existe un non terminal A telque $A \rightarrow^* Aw$.
- Il existe un algorithme permettant d'éliminer la récursivité à gauche sans changer le langage.
- Forme normale de Chomsky :

$$X \rightarrow a$$

$$X \rightarrow YZ$$

- De même, toute grammaire peut être mise en forme normale de Chomsky.

Algorithme de Cooke-Younger-Kasami

- On part d'une grammaire en forme de Chomsky.
- Etant donné une phrase w, on cherche à analyser les sous phrases $w[i..j], i \leq j$ dans l'ordre des longueurs croissantes.
- $T[i][j]$ contient la liste des nonterminaux $A \rightarrow^* w[i..j]$.

for $i = 1, |w|$ **do**
  $T[i][i] = \{A \mid A \rightarrow w[i]\}$;
for $d = 1, |w| - 1$ **do**
  for $i = 1, |w| - d$ **do**
    for $j = i, i + d$ **do**
      $V = \{A \mid A \rightarrow BC, B \in T[i][j], C \in T[j + 1][i + d]\}$;
      $T[i][i + d] = T[i][i + d] \cup V$;

w est accepté si l'axiome de la grammaire est dans $T[1][|w|]$.

Descente récursive, algorithme naïf, I

- L'idée : une fonction récursive pour reconnaître chaque production.
- Argument : la position courante dans le mot à analyser.
- Résultat : la position après reconnaissance de la production, ou -1 si la reconnaissance a échoué.
- On reste en forme normale de Chomsky, bien que ce ne soit pas indispensable.

Algorithme naïf, II

On suppose que le terminal A peut être développé de trois façons :

$$A \rightarrow a|BC|DE.$$

```
let rec checkBC p =
    let q = checkB p
    in if q < 0 then -1
       else checkC q

and checkDE p =
    let q = checkD p
    in if q < 0 then -1
       else checkE q

and checkA p =
    if w.(p) = 'a' then p+1
    else let q = checkBC p
         in if q > p then q
            else checkDE p

and ...
```

Le problème de la récursivité à gauche

Il est facile de voir que si la grammaire est récursive à gauche, le programme ci-dessus boucle. Il est toujours possible de réécrire la grammaire pour éliminer la récursivité à gauche sans changer le langage.

Exemple

```
instructions : instructions SEMICOLON instruction
             | instruction ;
```

peut être remplacé par :

```
instructions : instruction
             | instruction SEMICOLON instructions ;
```

ou mieux par :

```
instructions : instruction suite ;
suite :                              /* vide */
      | SEMICOLON instructions ;
```

Noter la syntaxe des productions, le second membre de longueur nulle, signalé par un commentaire, et la factorisation.

Elimination de la récursivité gauche

- La méthode ci-dessus se généralise au cas d'un nombre quelconque de récursion immédiates gauche :

$$A \longrightarrow w_0|Aw_1|Aw_2|\ldots|Aw_n$$

devient

$$A \longrightarrow w_0A' \;\; A' \longrightarrow w_1A' \;\; \ldots \;\; A' \longrightarrow w_nA' \; A' \longrightarrow \epsilon$$

- Mais il peut y avoir des récursions gauches non immédiates.
- Ordonner les symboles.
- Dérécursiver le premier symbole, si nécessaire.
- Pour dérécursiver le symbole numéro i, on commence par remplacer les symboles de rang $j < i$ par leurs valeurs, et on dérécursive le résultat.

Le problème du retour arrière

- L'algorithme de la descente récursive essaie toutes les productions possibles jusqu'à trouver la bonne : complexité non linéaire.
- Existe-t-il des grammaires pour lesquelles on peut “deviner” la bonne production ?
- Le seule information que l'on possède est le caractère courant de la chaîne d'entrée. Il faut que ce caractère détermine sans ambiguïté la production à utiliser.
- Par exemple, si toute production commence par un terminal, et si pour un non terminal donné, les premiers terminaux de gauche sont distincts, l'analyse syntaxique peut se faire sans retour arrière.

Grammaires prédictives

Pour généraliser, on définit deux fonctions **First** et **Follows** sur l'alphabet des terminaux et non terminaux :

- $x \in \textbf{First}(Y)$ si il existe w tel que $Y \rightarrow w$ et x est le premier caractère de w. $\epsilon \in \textbf{First}(Y)$ s'il existe une dérivation $Y \rightarrow \epsilon$.
- En particulier, si y est terminal, $\textbf{First}(y) = \{y\}$.
- $x \in \textbf{Follows}(Y)$ s'il existe une production $A \rightarrow uYZv$ et $x \in \textbf{First}(Z)$.

$\textbf{First}^+(X) = \textbf{if } \epsilon \in \textbf{First}(X) \textbf{ then } \textbf{First}(X) \cup \textbf{Follows}(X) \textbf{ else } \textbf{First}(X)$;

Une grammaire est prédictive si les membre droits de ses productions ont des ensembles $\textbf{First}^+$ disjoints.

Calcul de **First** et **Follows**

On construit un système d"équations :

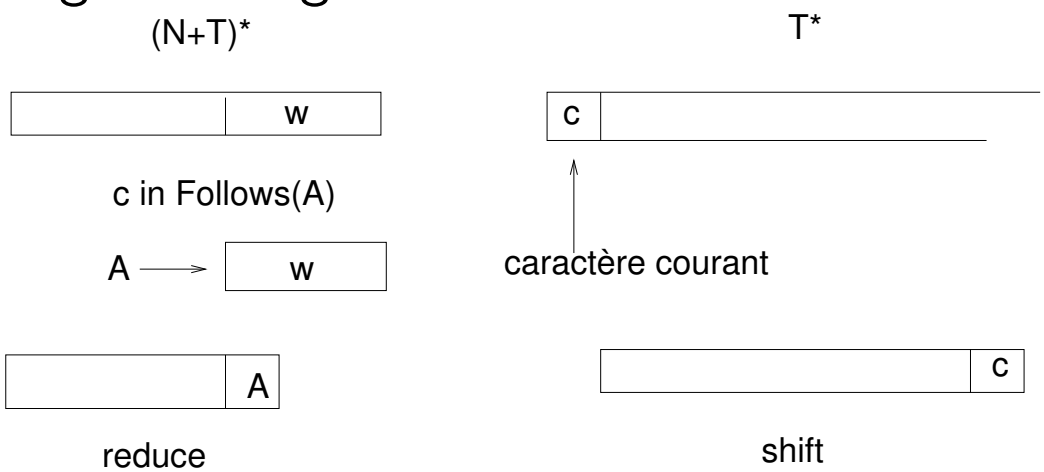
a est un terminal, X, Y des non terminaux, U une lettre quelconque, v, w des mots quelconques

$\mathbf{First}(a) = \{a\}$;
$\mathbf{First}(X) = \bigcup_{X \to Uw} \mathbf{First}^+(U)$;
$\mathbf{First}^+(X) = \mathbf{if}\ \epsilon \in \mathbf{First}(X)\ \mathbf{then}\ \mathbf{First}(X) \cup \mathbf{Follows}(X)\ \mathbf{else}\ \mathbf{First}(X)$;
$\mathbf{Follows}(X) = \bigcup_{X \to vXYw} \mathbf{First}^+(Y)$;

- On initialise à $\mathbf{First}(X) = \mathbf{Follows}(X) = \emptyset$ et on itère jusqu'à obtenir un point fixe.
- Convergence : le cardinal des ensembles ne peut que croitre et est borné par le nombre de caractères de l'alphabet.
- On peut recherchet un ordre efficace pour les itérations (par exemple basé sur un graphe de dépendance).

Analyse ascendante

Algorithme glouton



1. On a analysé un préfixe de la chaîne d'entrée.
2. On recherche une production $A \longrightarrow w$ telle que w soit un postfixe de la partie déja analysée et telle que le caractère courant puisse suivre A.
3. Si on trouve, on remplace w par A et on recommence (*reduce*).
4. Sinon, on transporte le caractère courant vers la partie déja analysée, on déplace le pointeur de lecture et on recommence (*shift*).
5. L'algorithme s'arrête quand tous les caractères ont été lu. La chaîne d'entrée est acceptée si la partie analysée se réduit à l'axiome.

Items, fermeture

On veut éviter de parcourir toute la grammaire à l'étape 2. Pour cela, on tient à jour un ensemble de productions candidates ou “items”.

- On note que la partie déja analysée est gérée comme une pile.
- On obtient un item en insérant un point dans le second membre d'une production : $A \longrightarrow u \bullet v$.
- Si $A \longrightarrow u \bullet v$ fait partie des productions candidates, cela signifie que u est un postfixe de la pile.
- Si $A \longrightarrow w\bullet$ est une production candidate, on peut réduire.
- Si $A \longrightarrow u \bullet Xv$ est candidate, il faut que l'on puisse pousser un X sur la pile. Toute production de la forme $X \longrightarrow \bullet u$ est donc candidate.
- On construit ainsi la fermeture (*closure*) d'un ensemble d'items.

Automate des items

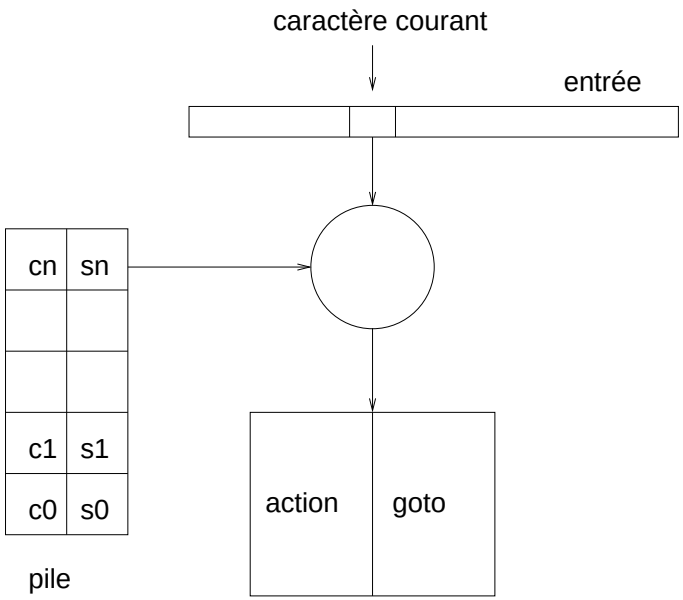
On note que l'ensemble des items est fini, et que l'on peut définir une fois pour toute l'effet de l'empilage d'un caractère quelconque. On construit un automate fini dont les états sont les ensembles d'items.

- Si I est un ensemble d'items, son successeur par le caractére x, terminal ou non terminal, est

$$I.x = \text{closure}(\{A \to ux \bullet v \mid A \to u \bullet xv \in I\})$$

- Pour simplifier, on introduit un nouvel axiome S' et la production $S' \to S$. L'état initial est $\{S' \to \bullet S\}$.
- On utilise un algorithme "à la demande" pour éviter de construire des états inaccessibles.
- On remarque qu'après une réduction on se retrouve dans un état antérieur de l'algorithme, à partir duquel on doit construire le nouvel état courant. Les états doivent donc également être empilés.

Algorithme glouton



Automate à pile

while *true* **do**
  $s = \text{sommet.de.pile};$
  $c = \text{caractere.courant};$
  switch $\text{Action}[s][c]$ **do**
    case *shift*
      push c;
      push $\text{goto}[s][c]$;
    case *reduce* $A \longrightarrow w$
      pop $2 \times |w|$ symboles;
      push A;
      push $\text{goto}[s][A]$;
    case *accept*
      stop;
    case *error*
      error;
      stop;

Ambiguïtés

Lors de la construction de la table Action, on peut rencontrer des ambiguïtés (encore appelées *conflits*) :

- Il peut y avoir dans l'état plusieurs productions réductibles : conflit reduce/reduce.
- On peut avoir le choix entre une réduction $A \longrightarrow w\bullet$ et un shift $B \longrightarrow u \bullet cv$ alors que le caractère courant est c.
- Pour résoudre le problème, on divise les états suivant le caractère courant attendu. Un item prend la forme $[A \longrightarrow u \bullet v, c]$.
- Signification : u est sur la pile, et si on parvient à réduire $u.v$ alors c sera le prochain caractère à lire en entrée.
- On améliore la précision, mais le nombre d'états possibles augmente énormément.

Fermeture

L'algorithme de fermeture est modifié comme suit :

Data: Une liste d'items LR(1)
Result: La fermeture de la liste argument
FIFO = la liste en argument;
resultat = $\emptyset$;
while *FIFO non vide* **do**
    extraire un item de la FIFO;
    l'adjoindre au résultat;
    if *l'item est de la forme* $[A \longrightarrow u \bullet Xv, c]$ **then**
        foreach $X \longrightarrow w$ **do**
            foreach $d \in$ **First**(vc) **do**
                $i = [C \longrightarrow \bullet w, d]$;
                if $i \notin$ *résultat* **then** ajouter i à la FIFO

La fonction **First** est étendue pour s'appliquer à un mot quelconque.

Construction de l’automate LR(1)

- Le calcul de l’automate est identique au cas où on n’utilise pas de symbole *look ahead*.
- Par contre la table des actions est différente :
  - S’il y a plusieurs réduction possibles, on discrimine en fonction du symbole d’entrée suivant. Il n’y a de conflit *reduce / reduce* que s’il y a deux items $[A \rightarrow w\bullet, a]$ et $[A' \rightarrow w\bullet, a]$ avec le même terminal a.
  - De même pour qu’il y ait conflit entre une réduction $[A \rightarrow w\bullet, a]$ et un shift $[B \rightarrow u \bullet cv, d]$ il faut que $a = c$.

Algorithme LALR

Le nombre d'états de l'automate peut devenir énorme :

$$2^{|G| \times L \times |T|}$$

où $|G|$ est le nombre de production, L leur longueur moyenne et $|T|$ la taille de l'alphabet terminal.
L'algorithme LALR (utilisé par Yacc) vise à compacter cet automate sans trop perdre de puissance.

- Le cœur d'un item $[A \longrightarrow u \bullet v, c]$ est la production potentielle $A \longrightarrow u \bullet v$.
- le cœur d'un état est l'ensemble des cœurs de ses items.
- Deux états ayant le même cœur sont équivalents pour l'automate des états. On fusionne donc en un seul tous les états ayant même cœur.
- Il se peut qu'au moment de calculer les actions, on trouve des conflits qui auraient pu être éviter par l'algorithme LR(1). Dans ce cas, l'algorithme échoue.

Résolution des conflits, I

Premier cas : la grammaire est donnée.

- Elle ne doit pas avoir de conflit.
- Si cependant il y en a, l'outil Yacc les résoud de manière arbitraire (en général, priorité au shift plutôt qu'au reduce).
- Il est possible d'agir sur ce choix en spécifiant des priorités ou des associativités.

Le problème du else

```
%{
type statement =
    Assign of string
  | Nop
  | If of string * statement * statement
;;
%}
%token IF THEN ELSE
%token COND
%token ASSIGN
%type <string> COND ASSIGN
%start stat
%type <statement> stat
%%
stat : ASSIGN
       {Assign of $1}
     | IF COND THEN stat
       {If ($2, $4, Nop)}
     | IF COND THEN stat ELSE stat
       {If ($2, $4, $6)}
     ;
%%
```

Le conflit

```
state 7
stat : IF COND THEN . stat  (2)
stat : IF COND THEN . stat ELSE stat  (3)
IF  shift 3
ASSIGN  shift 4
.  error
stat  goto 8

8: shift/reduce conflict (shift 9, reduce 2) on ELSE
state 8
stat : IF COND THEN stat .  (2)
stat : IF COND THEN stat . ELSE stat  (3)
ELSE  shift 9
$end  reduce 2
```

Exemple : une grammaire ambiguë

```
%{
type expression =
   Ident of string
 | Plus of expression * expression
;;
%}

%token IDENT PLUS
%type <string> IDENT
%start expr
%type <expression> expr
%%

expr : IDENT
     {Ident $1}
     | expr PLUS expr
     { Plus $1 $3}
     ;
%%
```

La solution

```
%{
type expression =
   Ident of string
 | Plus of expression * expression
;;
%}

%token IDENT PLUS
%left PLUS
%type <string> IDENT
%start expr
%type <expression> expr
%%

expr : IDENT
     {Ident $1}
     | expr PLUS expr
     { Plus $1 $3}
     ;
%%
```

Résolution des conflits, II

Deuxième cas : le langage est donné, mais pas la grammaire.

- On a probablement construit une grammaire ambigue.
- L'outil peut afficher l'automate LALR et signale les états qui comportent un conflit.
- Rechercher en particulier les paires de productions qui sont préfixe l'une de l'autre.
- Factoriser.

Résolution des conflits, III

On élabore simultanément le langage et son compilateur.

- On peut ajuster la syntaxe pour simplifier la compilation.
- Prévoir des marqueurs terminaux (par exemple des points-virgules après chaque instruction).
- Utiliser les parenthèses (`begin`, `end`, etc).
- “Refermer” toute les constructions :
  - `if .. then .. else .. fi` (exemple)
  - `do .. end_do`, `proc .. end_proc`

Gestion des erreurs

L'analyseur signale une erreur quand il n'existe aucune transition pour le caractère courant dans l'état courant.

- Le minimum est de signaler l'erreur en indiquant la ligne et le caractère (token) en cause.
- On peut ensuite soit terminer l'analyse (méthode Ocaml) soit essayer de poursuivre (méthode C).
  - Pour poursuivre, on choisit dans le langage une construction facile à repérer (par exemple le point virgule qui termine toute instruction en C).
  - On avale les caractères d'entrée jusqu'à passer un point virgule.
  - On dépile jusqu'à trouver une instruction et on poursuit.

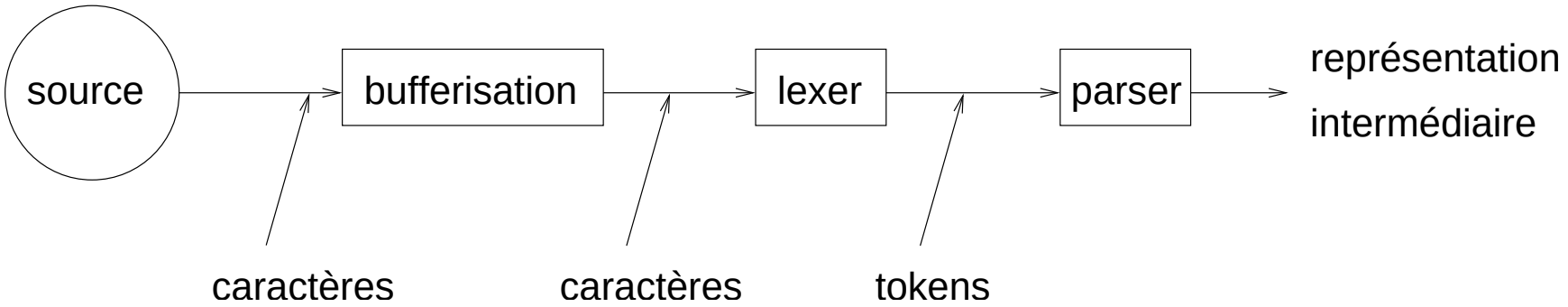
Interface avec Lex, I

- Lex et Yacc doivent utiliser le même système de codage des tokens : techniquement, c'est un type énuméré.
- Dans le contexte Ocaml, c'est ocamlyacc qui construit le type des jetons à partir des déclarations de la grammaire.
- Lex doit importer le fichier d'interface correspondant.

```
type token =
    IF
  | THEN
  | ELSE
  | COND of (string)
  | ASSIGN of (string)

val stat :
  (Lexing.lexbuf  -> token) -> Lexing.lexbuf -> statement
```

Interface avec Lex, II



- L'analyseur lexical et l'analyseur syntaxique fonctionnent en pipeline.
- Techniquement, l'analyseur syntaxique appelle l'analyseur lexical chaque fois qu'il a besoin du jeton suivant.

```
let lexbuf = Lexing.from_channel (open_in fichier)
in try (
    let theSource = Grammar.design Lexicon.token lexbuf
    in ...
  with Parse_error -> ...
```