université virtuelle Burkina Faso

Sciences du Numérique

Théorie des Langages et Compilation
MSI-02
Analyse Syntaxique

Dr. Tegawendé F. BISSYANDE
tegawende.bissyande@gmail.com

Semestre 1
"Sécurité des Réseaux et des Logiciels"

Analyse syntaxique: Kesako?

- Objectif
  - Structurer une chaîne d'unités lexicales en unités syntaxiques (*syntagmes*)
- Exemple
  - *Langages impératifs*: variables, expressions, instructions, déclarations, etc.
  - *Langages fonctionnels*: variables, expressions, définition, etc.
  - *Langages logiques*: variables, termes, buts, clauses, etc.
- Résultat

  Souvent une représentation intermédiaire, en général sous forme d'arbre syntaxique

Spécification

- Principe

  Pour un grand nombre de langages, la structure syntaxique des programmes peut être décrite par une grammaire non contextuelle (*context-free grammar*)

- Remarque
  - Les expressions régulières son en général insuffisantes (cas des structures emboîtées par exemple)
  - Pour chaque grammaire non contextuelle, on peut construire un automate à pile (non déterministe) acceptant le langage décrit par cette grammaire

Grammaires

- Définition

  Une **grammaire** G est définie par un quadruplet (V_T, V_N, S, P), où :

  - V_T est un alphabet (les **terminaux**)
  - V_N est un alphabet disjoint de V_T (les **non-terminaux**)
  - S est une lettre de V_N appelée **axiome**
  - P est un ensemble fini de couples de $(V_T \cup V_N)^+ \times (V_T \cup V_N)^*$, appelés **règle de productions**

Grammaires pour les expressions arithmétiques

$$
\begin{array}{lcl}
E & ::= & E \texttt{ + } E \\
E & ::= & E \texttt{ - } E \\
E & ::= & E \texttt{ * } E \\
E & ::= & E \texttt{ / } E \\
E & ::= & \texttt{(}E\texttt{)} \\
E & ::= & \texttt{int}
\end{array}
\qquad\qquad
\begin{array}{lcl}
E & ::= & E \texttt{ + } E \\
 & | & E \texttt{ - } E \\
 & | & E \texttt{ * } E \\
 & | & E \texttt{ / } E \\
 & | & \texttt{(}E\texttt{)} \\
 & | & \texttt{int}
\end{array}
$$

Grammaire non contextuelle

Une Grammaire est dite **non contextuelle** (ou algébrique) si et seulement si :

$$P \subset V_N \times (V_T \cup V_N)^*$$

En d'autres termes, ssi il n'y a qu'un seul non-terminal en partie gauche des règles de production

Langage engendré

Le langage $L(G)$ engendré par une grammaire G est l'ensemble des mots produits en partant du symbole de départ S et en appliquant la démarche suivante aux mots α :

1. Si α n'est formé que de terminaux, alors α est un mot ω de $L(G)$
2. Sinon, α peut se découper en $\beta A \gamma$, où A est un non-terminal
3. Alors, on considère une production $A ::= \delta$, on remplace A dans α par δ, noté $\alpha \Rightarrow \beta\delta\gamma$

L'opération décrite ici s'appelle une **dérivation**

Arbre de dérivation syntaxique

- Définition

  Un arbre de dérivation syntaxique pour la grammaire G de racine $X \in V_N$ et de feuilles $\omega \in V_T^*$ est un arbre ordonné dont la racine est X, les feuilles sont étiquetés par des terminaux formant le mot ω et les nœuds internes par des non-terminaux tels que si Y est un nœud interne dont les p fils sont étiquetés par les symboles $\alpha_1 \dots \alpha_p$ alors $Y ::= \alpha_1 \dots \alpha_n$ est une production de P.

Ambiguïtés

- Une phrase $x \in L(G)$ est ambiguë si elle admet plusieurs arbres syntaxiques

- Exemple

Analyse descendante/ascendante

- Définition

  Analyser un mot, c'est établir si le mot appartient au langage engendré par la grammaire. En pratique, on construit dans le cas positif une dérivation du mot à partir du symbole initial de la grammaire ou bien un arbre de dérivation syntaxique

- Analyse descendante

  Part du symbole de départ (axiome) et l'expanse jusqu'à obtenir le mot.

- Analyse ascendante

  Factorise le mot en reconnaissant des parties droites de production jusqu'à retomber sur l'axiome.

Automate à pile

Tout comme les langages réguliers sont reconnus par des automates finis, les langages non contextuels (algébriques) sont reconnus par des automates à pile.

La pile permet de mémoriser des informations au cours des transformations pour décider de la prochaine transition à effectuer. Cette pile est non bornée et peut mémoriser des informations de taille variable en fonction de l'entrée.

Analyse ascendante

- Principe

  On part du texte source (lecture de gauche à droite) et on cherche à remonter jusqu'à l'axiome en procédant par des actions de lecture ou de réduction.

- Lecture (*shift*)

  Consommer et empiler un lexème.

- Réduction (*reduce)*

  Reconnaître la partie droite d'une production au sommet de la pile et la transformer en le non-terminal correspondant.

Evolution dans l'automate

- État initial

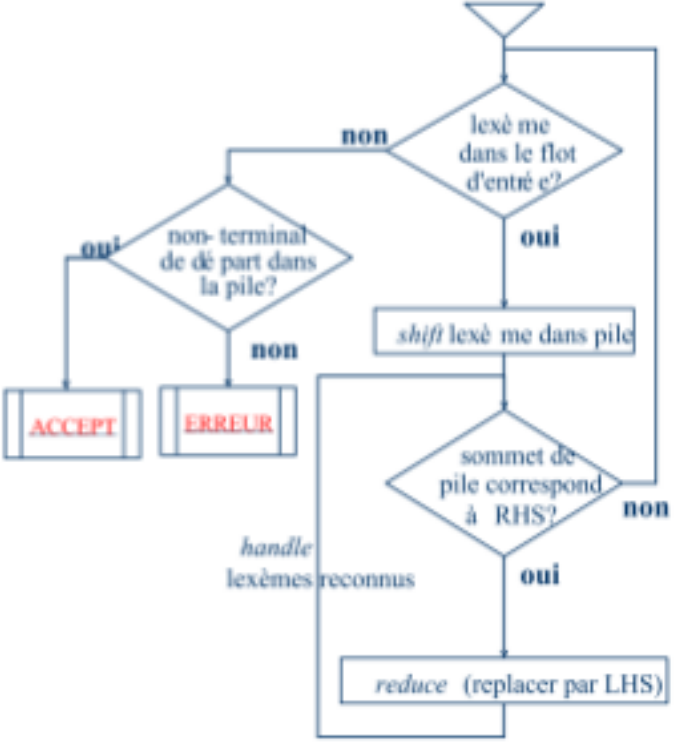
  La pile est vide.

- État intermédiaire

  L'automate évolue tant qu'une action peut se produire (lecture ou réduction).

- État final
  - Plus aucune action possible.
  - Si l'entrée est totalement lue et l'état de la pile est le symbole de départ S alors le mot a été reconnu, sinon échec.

Algorithme


lexème dans le flot d'entrée?
non
oui
non-terminal de départ dans la pile?
oui
non
ACCEPT
ERREUR
shift lexème dans pile
sommet de pile correspond à RHS?
non
oui
handle lexèmes reconnus
reduce (replacer par LHS)

Exemple

$$G_0 =< \{a, b, c\}, \{S, X, Y\}, \{S\}, \{S ::= XY, X ::= aX|c, Y ::= b\} >$$

Pile	Texte à analyser	Action	Représentation
	acb	shift	.acb
a	cb	shift	a.cb
ac	b	reduce	ac.b
aX	b	reduce	aX.b
X	b	shift	X.b
Xb		reduce	Xb.
XY		reduce	XY.
S			S.

Développement d'un analyseur syntaxique

Un véritable analyseur syntaxique est

- très gros
- très complexe
- très optimisé

Il existe des outils qui, à partir de spécifications basées sur des familles de grammaires algébriques, génèrent automatiquement l'analyseur syntaxique

- yacc, bison en C, ocamlyacc en OCaml, JavaCC en Java, etc.

Ocamlyacc

Structure du fichier source

- **Prélude** : Morceau de code du langage cible (Ocaml) qui sera mis en entête du fichier cible.
- **Déclarations** : Déclarations des unités lexicales utilisées dans les productions de la grammaire. Contient également des instructions permettant de résoudre les ambiguïtés de la grammaires.
- **Règles** : Composées des productions de la grammaire et de leur action sémantique associée. Les actions utilisent les primitives du langage cible (Ocaml).
- **Épilogue** : Il s'agit du code du langage cible qui est copié à la fin du fichier cible engendré.

Structure d'un fichier source

```
%{
  Prélude
%}
  Déclarations
%%
  Règles
%%
  Épilogue
```

Exemple 1 (début)

```
%{
  (* Rien à déclarer *)
%}

%token <int> INT
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN
%token EOL

%start expression
%type <unit> expression

%%
```

Exemple 1 (grammaire et actions sémantiques)

```
[...]

%%

expression :
  expr EOL            { }
| expr EOL expression { } ;

expr :
  expr PLUS expr     { }
| expr MINUS expr    { }
| expr TIMES expr    { }
| expr DIV expr      { }
| LPAREN expr RPAREN { }
| INT                { }
;
```

Exemple 1 (main.ml)

```
(* File exemple1/main.ml *)

let _ =
  let lexbuf = Lexing.from_channel stdin in (* Fabrication du flux *)
  try
    Parser.expression Lexer.token lexbuf
  with
    Lexer.Eof ->
      Printf.printf "Syntax ok\n";
      exit 0
  | Parsing.Parse_error -> Printf.printf "Syntax error\n"
```

Grammaires ambiguës

$$
\begin{array}{lcl}
E & ::= & T \\
 & | & \texttt{Id} \\
T & ::= & \texttt{Id}
\end{array}
$$

- Si Id est dans la pile, deux réductions possibles :
  - Réduction T
  - Réduction E

Grammaires ambiguës : shift-reduce

$$
\begin{array}{lcl}
E & ::= & E \texttt{ + } E \\
 & | & E \texttt{ * } E \\
 & | & \texttt{Id}
\end{array}
$$

- Si on considère le mot suivant de la grammaire $E + E \bullet + E$, alors deux actions possibles :
  - Si le mot est considéré comme $(E + E) + E$, alors reduce
  - Si le mot est considéré comme $E + (E + E)$, alors shift

Gestion des ambiguïtés par Ocamlyacc

- Par défaut
  - Conflit shift-reduce ⇒ shift
  - Conflit reduce-reduce ⇒ ordre
- Directives de Ocamlyacc
  - Règles de priorités
  - Définition de règles d'associativités
- Informations sur les conflits

  La commande `ocamlyacc -v parser.mly` génère le fichier `parser.output` contenant l'automate produit et le détail des conflits.

Priorité et associativité des opérateurs

```
%token <int> INT

%left  PLUS  MINUS     /* priorité la plus faible */
%left  TIMES DIV       /* priorité plus forte */
```

- Associativité (left, right)
- Définition dans l'ordre croissant de priorité

Priorité et associativité des opérateurs

```
%token <int> INT

%left  PLUS  MINUS     /* priorité la plus faible */
%left  TIMES DIV       /* priorité moyenne        */
%nonassoc UMINUS       /* priorité la plus forte  */

expr: MINUS expr %prec UMINUS   { }
```

- `%nonassoc` est un opérateur non-associatif
- `UMINUS` a la priorité la plus importante
- `%nonassoc` est fréquemment utilisé avec `%prec`

Ambiguïté If-Else

Conflit shift-reduce pour la construction

$$\begin{array}{rcl} \textit{stmt} & ::= & \texttt{IF}\ \textit{expr}\ \textit{stmt} \\ & | & \texttt{IF}\ \textit{expr}\ \textit{stmt}\ \texttt{ELSE}\ \textit{smt} \end{array}$$

Avec l'exemple :

IF *expr* IF *expr stmt* • ELSE *stmt*

- Deux choix possibles :
  1. Shift ELSE
  2. Reduce IF expr stmt

Ambiguïté If-Else

1. Shift

   IF *expr* IF *expr* *stmt* • ELSE *stmt*
   IF *expr* IF *expr* *stmt* ELSE • *stmt*
   IF *expr* IF *expr* *stmt* ELSE *stmt* •
   IF *expr* *stmt* •

2. Reduce

   IF *expr* IF *expr* *stmt* • ELSE *stmt*
   IF *expr* *stmt* • ELSE *stmt*
   IF *expr* *stmt* ELSE • *stmt*
   IF *expr* *stmt* ELSE *stmt* •

Ambiguïté If-Else

- La première solution (shift) est communément utilisée dans les langages de programmation modernes

  IF *expr* IF *expr* *stmt* • ELSE *stmt*
  IF *expr* IF *expr* *stmt* ELSE • *stmt*
  IF *expr* IF *expr* *stmt* ELSE *stmt* •
  IF *expr* *stmt* •

- Compatible avec les règles par défaut de Ocamlyacc

  shift-reduce ⇒ shift

Eliminer le conflit shift-reduce

- Utilisation d'une directive Ocamlyacc

```
%nonassoc IF_LOWER_ELSE
%nonassoc ELSE

stmt:
  IF expr stmt %prec IF_LOWER_ELSE
| IF expr stmt ELSE stmt
```

IF *expr* IF *expr* *stmt* • ELSE *stmt*

IF *expr* IF *expr* *stmt* ELSE • *stmt*

IF *expr* IF *expr* *stmt* ELSE *stmt* •

IF *expr* *stmt* •

Eliminer le conflit shift-reduce

- Réécriture de la grammaire

stmt	::=	*stmt-complete*
	|	*stmt-incomplete*
stmt-complete	::=	`IF` *expr* *stmt-complete* `ELSE` *stmt-complete*
	|	*autre*
stmt-incomplete	::=	`IF` *expr* *stmt*
	|	`IF` *expr* *stmt-complete* `ELSE` *stmt-incomplete*

Exemple 2 (lexer.mll)

```
{ open Parser          (* The type token is defined in parser.ml *)
  exception Illegal_character }

let ident = ['a'-'z' 'A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9' '_']*

rule token = parse
   [' ' '\t' '\n' '\r']  { token lexbuf }     (* skip blanks *)
   | ['0'-'9']+     { INT(int_of_string (Lexing.lexeme lexbuf)) }
   | '+'            { PLUS }
   | '-'            { MINUS }
   | '*'            { TIMES }
   | '/'            { DIV }
   | '('            { LPAREN }
   | ')'            { RPAREN }
   | '='            { EQUAL }
   | eof            { EOF }
   | "VAR"          { VAR }
   | "var"          { VAR }
   | ';'            { SEMICOLON }
   | ident          { IDENT(Lexing.lexeme lexbuf) }
   | _              { raise Illegal_character }
```

Exemple 2 (parser.mly)

```
%{ let vars = Hashtbl.create 17 %}
%token <int> INT
%token <string> IDENT
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN
%token SEMICOLON EQUAL VAR
%token EOF

%left PLUS MINUS
%left TIMES DIV
%nonassoc UMINUS

%start calcul
%type <int> calcul

%%

calcul:
  declarations expr EOF { $2 }
| expr EOF              { $1 }
;
```

```
declarations:
  declaration SEMICOLON                { }
| declaration SEMICOLON declarations   { }
;

declaration:
  VAR IDENT EQUAL expr   { Hashtbl.add vars $2 $4 }
;

expr:
  expr PLUS expr           { $1 + $3 }
| expr MINUS expr          { $1 - $3 }
| expr TIMES expr          { $1 * $3 }
| expr DIV expr            { $1 / $3 }
| MINUS expr %prec UMINUS  { - $2 }
| LPAREN expr RPAREN       { $2 }
| INT                      { $1 }
| IDENT                    { Hashtbl.find vars $1 }
;
```

Exemple 2 (main.ml)

```
(* File exemple2/main.ml *)


let _ =
  let lexbuf = Lexing.from_channel stdin in (* Fabrication du flux *)
  try
    let result = Parser.calcul Lexer.token lexbuf in
    Printf.printf "Resultat : %d\n" result
  with
    Parsing.Parse_error ->
      let loc_start = Lexing.lexeme_start lexbuf in
      let loc_end   = Lexing.lexeme_end lexbuf in
      Printf.printf "Syntax error: (%d,%d)\n" loc_start loc_end
```

Grammaires récursives droite et gauche

$$
\begin{array}{lcl}
\textit{list} & ::= & \texttt{id} \\
 & | & \textit{list}\ \texttt{', '}\ \texttt{id}
\end{array}
$$

id1 •, id2, id3, id4
list •, id2, id3, id4
list, id2 •, id3, id4
list •, id3, id4
list, id3 •, id4
list •, id4
list, id4 •
list •

$$
\begin{array}{lcl}
\textit{list} & ::= & \texttt{id} \\
 & | & \texttt{id}\ \texttt{', '}\ \textit{list}
\end{array}
$$

id1 •, id2, id3, id4
id1, id2 •, id3, id4
id1, id2, id3 •, id4
id1, id2, id3, id4 •
id1, id2, id3, list •
id1, id2, list •
id1, list •
list •