6 - Grammaires non-contextuelles

Noam Chomsky, né en 1928, linguiste américain, MIT

Grammaires non-contextuelles

- Une grammaire G = (N,T,P,A) est non-contextuelle* (ou algébrique) si ses productions sont de la forme :
  B → w
  avec B ∈ N et w ∈ (N∪T)*
- Un langage L est algébrique si il existe une grammaire algébrique G telle que :
  L(G) = L

Conséquence : Tout langage rationnel est aussi algébrique.
Réciproque fausse !!!

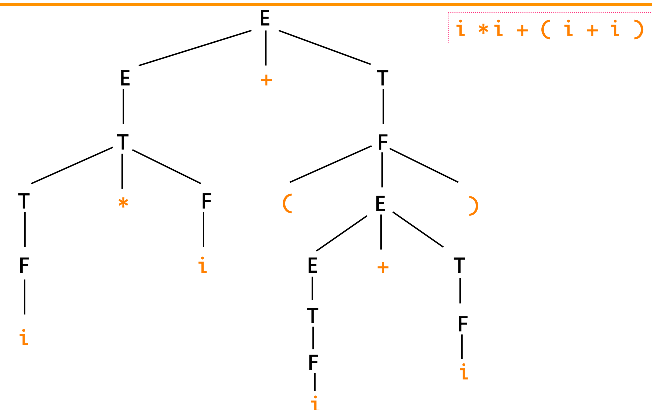
Un langage algébrique non rationnel
- n'est pas reconnu par un automate fini
- n'est pas décrit par une expression régulière
- il n'existe pas de grammaire régulière pour l'engendrer.

* *Context-free* (en anglais).

Exemple

- G = (N,T,P,E) où :
  - N = { E, T, F } ensemble des non-terminaux
  - T = { +, *, (,), i } ensemble des terminaux
  - P = { E → E + T
    E → T
    T → T * F
    T → F
    F → i
    F → (E) }
  - E est l'axiome
- L(G) : approximation de l'ensemble des expressions arithmétiques, présentes dans la plupart des langages de programmation.

Arbre de dérivation



Analyse syntaxique

- Les langages de programmation sont des langages algébriques. Ils sont spécifiés par des grammaires algébriques i.e. des procédés génératifs.
- Dans sa phase d'analyse syntaxique, un compilateur teste si un programme est bien un élément du langage de programmation dans lequel il est écrit.
- Pour un programme donné, le compilateur essaie de reconstituer l'arbre de dérivation* de bas en haut en procédant par réductions successives : on parle alors d'analyse ascendante.
- A l'issue de cette phase, on sait si notre programme est syntaxiquement correct ou pas ...

* aussi appelé pour cela arbre *syntaxique*

Dérivation « le plus à gauche »

- φ se dérive directement le plus à gauche en ψ et on note

$$\varphi \Rightarrow_g \psi$$

s'il existe des mots w_1 de T^* et w_2 de V^* tels que :
  - $\varphi = w_1 \; X \; w_2$
  - $\psi = w_1 \; w \; w_2$
  - P contient une production $X \rightarrow w$

w_1 et w_2 forment le contexte !

- φ se dérive le plus à gauche en ψ et on note

$$\varphi \Rightarrow_g^* \psi$$

s'il existe une suite de mots $w_0, w_1 \ldots w_n$ de V^* tels que :
  - $\varphi = w_0$
  - $\psi = w_n$
  - $\forall$ i, $0 \leq i < n$, $w_i \Rightarrow_g w_{i+1}$

- on note $\varphi \Rightarrow_g^+ \psi$ si $\varphi \Rightarrow_g^* \psi$ et φ ne se dérive pas *directement* le plus à gauche en ψ.

Dérivations équivalentes

- On définit pareil les dérivations à droite (directes ou pas) et on note :

$$\varphi \Rightarrow_d \psi$$

$$\varphi \Rightarrow_d^* \psi$$

Théorème *(admis)*
Soit G = (N,T,P,A) une grammaire algébrique, le langage engendré L(G)

vérifie :

$$\begin{cases} L(G) = \{ \varphi \in T^*, \; A \Rightarrow^* \varphi \} \\ L(G) = \{ \varphi \in T^*, \; A \Rightarrow_g^* \varphi \} \\ L(G) = \{ \varphi \in T^*, \; A \Rightarrow_d^* \varphi \} \end{cases}$$

- Une grammaire algébrique est ambiguë si elle engendre un mot par 2 dérivations à gauche différentes.
  ➥ on peut alors construire deux arbres de dérivation distincts pour ce mot.

Hélas, tout langage algébrique n'est pas engendré par une grammaire non-ambiguë ...

Exemple

- G = (N, T, P, E) où :
  - N = { E } ensemble des non-terminaux
  - T = { +, *, i, (,) } ensemble des terminaux
  - P = { E → E + E
    E → E * E
    E → i
    E → (E) }
  - E est l'axiome
- cette grammaire G est ambiguë !
- voici deux dérivations le plus à gauche distinctes pour

w = i + i * i

- $E \Rightarrow_g E + E \Rightarrow_g i + E \Rightarrow_g i + E * E \Rightarrow_g i + i * E \Rightarrow_g i{+}i{*}i$
- $E \Rightarrow_g E * E \Rightarrow_g E + E * E \Rightarrow_g i + E * E \Rightarrow_g i + i * E \Rightarrow_g i{+}i{*}i$

Exemple

- G' = (N',T,P',E) n'est plus ambiguë
- on a retiré la *récursivité à gauche* dans G
  - N' = { E, E' T, T', F }
  - T = { +, *, (,), i } comme dans la grammaire G précédente
  - E est l'axiome
  - P = {

```
P = {     E  →  T
                E  →  T E'
                E' →  + T
                E' →  + T E'
                T  →  F
                T  →  F T'
                T' →  * F
                T' →  * F T'
                F  →  i
                F  →  ( E )           }
```

- on obtient G' *équivalente* à la grammaire G c'est-à-dire :

L(G) = L(G')

Forme de Backus-Naur

- description analytique d'une grammaire informatique, aussi dite BNF (Algol 1960) :
  - alternative notée |
  - option notée entre [ ]
  - répétition (au moins une fois) notée entre { }
  - non-terminaux entre < >
  - ::== sépare les parties gauche et droite
- aujourd'hui, on utilise (*les multiples variantes de*) son extension EBNF :
  - les non-terminaux sont des noms
  - terminaux notés entre apostrophes
  - utilisation des symboles +, *, (), { }, ?, & ...

Extrait d'une grammaire sous forme BNF

```
<decimal number>       ::= [{space | tab}] <left decimal>
<left decimal>         ::= [+|-] <unsigned decimal>
<unsigned decimal>     ::= <finite number> | <infinity> | <NAN>
<finite number>        ::= <significand>  [<exponent>]
<significand>          ::= <integer> | [<mixed>]
<integer>              ::= <digits> [.]
<digits>               ::= {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}
<mixed>                ::= [<digits>] . <digits>
<exponent>             ::= E [+|-] <digits>
<infinity>             ::= [+|-] INF
<NAN>                  ::= NAN [([<digits>])]
```


Intégralité de la grammaire EBNF du langage de programmation RUBY

```
• PROGRAM : COMPSTMT
• COMPSTMT : STMT (TERM EXPR)* [TERM]
• STMT : CALL do ['|' [BLOCK_VAR] '|'] COMPSTMT end
        | undef FNAME
        | alias FNAME FNAME
        | STMT if EXPR
        | STMT while EXPR
        | STMT unless EXPR
        | STMT until EXPR
        | 'BEGIN' '{' COMPSTMT '}'
        | 'END' '{' COMPSTMT '}'
        | LHS '=' COMMAND [do ['|' [BLOCK_VAR] '|'] COMPSTMT end]
        | EXPR
• EXPR : MLHS '=' MRHS
        | return CALL_ARGS
        | yield CALL_ARGS
        | EXPR and EXPR
        | EXPR or EXPR
        | not EXPR
        | COMMAND
        | '!' COMMAND
        | ARG
• CALL : FUNCTION
        | COMMAND
• COMMAND : OPERATION CALL_ARGS
        | PRIMARY '.' OPERATION CALL_ARGS
        | PRIMARY '::' OPERATION CALL_ARGS
        | super CALL_ARGS
```

```
•   FUNCTION : OPERATION ['(' [CALL_ARGS] ')']
            | PRIMARY '.' OPERATION '(' [CALL_ARGS] ')'
            | PRIMARY '::' OPERATION '(' [CALL_ARGS] ')'
            | PRIMARY '.' OPERATION
            | PRIMARY '::' OPERATION
            | super '(' [CALL_ARGS] ')'
            | super
•   ARG : LHS '=' ARG
            | LHS OP_ASGN ARG
            | ARG '..' ARG
            | ARG '...' ARG
            | ARG '+' ARG
            | ARG '-' ARG
            | ARG '*' ARG
            | ARG '/' ARG
            | ARG '%' ARG
            | ARG '**' ARG
            | '+' ARG
            | '-' ARG
            | ARG '|' ARG
            | ARG '^' ARG
            | ARG '&' ARG
            | ARG '<=>' ARG
            | ARG '>' ARG
            | ARG '>=' ARG
            | ARG '<' ARG
            | ARG '<=' ARG
            | ARG '==' ARG
            | ARG '===' ARG
            | ARG '!=' ARG
            | ARG '=~' ARG
            | ARG '!~' ARG
            | '!' ARG
            | '~' ARG
            | ARG '<<' ARG
            | ARG '>>' ARG
            | ARG '&&' ARG
            | ARG '||' ARG
            | defined? ARG
            | PRIMARY
```



```
•   PRIMARY: '(' COMPSTMT ')'
            | LITERAL
            | VARIABLE
            | PRIMARY '::' IDENTIFIER
            | '::' IDENTIFIER
            | PRIMARY '[' [ARGS] ']'
            | '[' [ARGS [',']] ']'
            | '{' [(ARGS|ASSOCS) [',']] '}'
            | return ['(' [CALL_ARGS] ')']
            | yield ['(' [CALL_ARGS] ')']
            | defined? '(' ARG ')'
            | FUNCTION
            | FUNCTION '{' ['|' [BLOCK_VAR] '|'] COMPSTMT '}'
            | if EXPR THEN
              COMPSTMT
              (elsif EXPR THEN COMPSTMT)*
              [else COMPSTMT]
              end
            | unless EXPR THEN  COMPSTMT
              [else COMPSTMT]
              end
            | while EXPR DO COMPSTMT end
            | until EXPR DO COMPSTMT end
            | case COMPSTMT
              (when WHEN_ARGS THEN COMPSTMT)+
              [else COMPSTMT]
              end
            | for BLOCK_VAR in EXPR DO
              COMPSTMT
              end
            | begin  COMPSTMT  [rescue [ARGS] DO COMPSTMT]+
              [else COMPSTMT]
              [ensure COMPSTMT]
              end
            | class IDENTIFIER ['<' IDENTIFIER]
              COMPSTMT
              end
            | module IDENTIFIER
              COMPSTMT
              end
            | def FNAME ARGDECL
              COMPSTMT
              end
            | def SINGLETON ('.'| '::') FNAME ARGDECL
              COMPSTMT
              end
```



```
•   WHEN_ARGS : ARGS [',' '*' ARG]
            | '*' ARG
•   THEN : TERM
            | then
            | TERM then
•   DO : TERM
            | do
            | TERM do
•   BLOCK_VAR : LHS
            | MLHS
•   MLHS : MLHS_ITEM ',' [MLHS_ITEM (',' MLHS_ITEM)*] ['*' [LHS]]
            | '*' LHS
•   MLHS_ITEM : LHS
            | '(' MLHS ')'
•   LHS : VARIABLE
            | PRIMARY '[' [ARGS] '] '
            | PRIMARY '.' IDENTIFIER
•   MRHS : ARGS [',' '*' ARG]
            | '*' ARG
•   CALL_ARGS : ARGS
            | ARGS [',' ASSOCS] [',' '*' ARG] [',' '&' ARG]
            | ASSOCS [',' '*' ARG] [',' '&' ARG]
            | '*' ARG [',' '&' ARG]
            | '&' ARG
            | COMMAND
•   ARGS : ARG (',' ARG)*
•   ARGDECL : '(' ARGLIST ')'
            | ARGLIST TERM
•   ARGLIST : IDENTIFIER(','IDENTIFIER)*[','*'[IDENTIFIER]][','&'IDENTIFIER]
            | '*'IDENTIFIER[','&'IDENTIFIER]
            | ['&'IDENTIFIER]
•   SINGLETON : VARIABLE
            | '(' EXPR ')'
•   ASSOCS : ASSOC (',' ASSOC)*
•   ASSOC : ARG '=>' ARG
•   VARIABLE : VARNAME
            | nil
            | self
•   LITERAL : numeric
            | SYMBOL
            | STRING
            | STRING2
            | HERE_DOC
            | REGEXP
•   TERM : ';'| '\n'
```


Grammaire RUBY (fin) : la partie lexicale

```
•   OP_ASGN : '+=' | '-=' | '*=' | '/=' | '%=' | '**='
            | '&=' | '|=' | '^=' | '<<=' | '>>='
            | '&&=' | '||='
•   SYMBOL : ':'FNAME| ':'VARNAME
•   FNAME : IDENTIFIER | '..' | '|' | '^' | '&'
            | '<=>' | '==' | '===' | '=~'
            | '>' | '>=' | '<' | '<='
            | '+' | '-' | '*' | '/' | '%' | '**'
            | '<<' | '>>' | '~'
            | '+@' | '-@' | '[]' | '[]='
•   OPERATION : IDENTIFIER
            | IDENTIFIER'!'
            | IDENTIFIER'?'
•   VARNAME : GLOBAL
            | '@'IDENTIFIER
            | IDENTIFIER
•   GLOBAL : '$'IDENTIFIER|
             '$'any_char
            | '$''-'any_char
•   STRING : '"' any_char* '"'
            | ''' any_char* '''
            | ''' any_char* '''
•   STRING2 : '%'('Q'|'q'|'x')char any_char* char
•   HERE_DOC : '<<'(IDENTIFIER|STRING)
           any_char*
           IDENTIFIER
•   REGEXP: '/' any_char* '/'['i'|'o'|'p']
            | '%''r' char any_char* char
•   IDENTIFIER is the sequence of characters in the pattern of /[a-zA-Z_][a-zA-Z0-9_]*/.
```

Hiérarchie de Chomsky (1956)

Type	Langages	Grammaires
3	réguliers ou rationnels	régulières à droite (ou linéaires à droite) $A \to a$ $A \to aB$ $A \to \varepsilon$ $A,B \in N$ $a \in T$ et régulières à gauche (ou linéaires à gauche)
2	algébriques ou non-contextuels	algébriques non-contextuelles $A \to \alpha$ $A \in N$ $\alpha \in V^*$
1	contextuels	contextuelles monotones $\alpha \to \beta$ ou $A \to \varepsilon$ $\alpha, \beta \in V^+$, A axiome $\lvert\alpha\rvert \leq \lvert\beta\rvert$
0	récursivement énumérables	contextuelles avec effacement $\alpha \to \beta$ $\alpha \in V^+$, $\beta \in V^*$ soit *aucune contrainte aux productions*

Attention !!! un langage d'un type donné peut toujours être engendré par une grammaire de type plus compliqué ...

Grammaires propres

- Que des symboles productifs : chaque terminal et non-terminal doit apparaître dans la dérivation d'au moins un mot de L.
- Pas de renommage : G ne contient pas de production de la forme $A \to B$ avec $A, B \in N$
- Pas de cycle : G ne contient pas de cycle du type $A \Rightarrow^+ A$

Théorème Tout langage algébrique $L = L \setminus \{\varepsilon\}$ peut être engendré par une grammaire propre ne contenant pas de production vide.

- Si L contient ε, L est engendré par une grammaire propre G = (N,T,P,S) dont la seule production vide est : $S \to \varepsilon$ avec S absent des parties droites des productions de P
- Une fois nettoyées (algo. pour cela), on peut mettre les grammaires sous forme standard : les formes normales.

Forme normale de Chomsky

- Une grammaire algébrique G = (N,T,P,S) est sous forme normale de Chomsky si toute production est de la forme :

$$A \to a$$
$$A \to BC$$
$$S \to \varepsilon$$

avec $A,B,C \in N$ et $a \in T$ (et si on a $S \to \varepsilon$, S ne figure dans aucun membre droit d'une autre production)

- Il existe un algorithme qui transforme toute grammaire algébrique propre en grammaire sous FNC.
- Il existe une autre forme normale, tout aussi classique : la forme normale de Greibach dite FNG.

Sheila Greibach, née en 1939 Prof. CS, UCLA

Algorithme de mise sous FNC

On part de G = (N,T,P,S) propre, on construit G' = (N',T,P',S) équivalente et sous FNC :

1. $N' \leftarrow \{S\}$
   $P' \leftarrow \{A \to a,\ A \to BC$ et éventuellement $S \to \varepsilon$ pour $A,B,C \in N,\ a \in T\}$
2. pour chaque production de P de la forme :

$$A \to \alpha_1 \ldots \alpha_p \text{ avec } A \in N,\ \alpha_i \in V \text{ et } p > 2$$

on ajoute à P' les productions : $A \to A_1 X_1$
$X_1 \to A_2 X_2$
...
$X_{p-2} \to A_{p-1} A_p$

où $A_i = \alpha_i$ si $\alpha_i \in N$ ou alors $A_i = X_{\alpha_i}$ si $\alpha_i \in T$.

3. N' reçoit les A, les X_i et les A_i.

Algorithme (suite)

4 on remplace les productions de P de la forme :

$$A \rightarrow \alpha_1\alpha_2 \text{ avec } \alpha_1 \text{ ou } \alpha_2 \in T$$

par des productions dans P' de la forme :

$$A \rightarrow Y_1Y_2$$
$$Y_1 \rightarrow \alpha_1 \text{ si } \alpha_1 \in T,\ Y_1 = \alpha_1 \text{ sinon}$$
$$Y_2 \rightarrow \alpha_2 \text{ si } \alpha_2 \in T,\ Y_2 = \alpha_2 \text{ sinon}$$

5 pour chaque X_a créé plus haut, on ajoute à P' :

$$X_a \rightarrow a$$

6 N' reçoit les Y_i et les X_a précédents.

Exemple

- G = (N,T,P,E) une grammaire <u>propre</u> :
  - N = { E, T, F }
  - T = { +, *, (,), i }
  - P = { E → E + T | T * F | (E) | i
T → T * F | (E) | i
F → i
F → (E) }
  - E est l'axiome
- G' = (N',T,P',E) où <u>pour l'instant</u> :
  - N'= { E,... }
  - T = { +, *, (,), i }
  - P = { E → i
T → i
F → i }
  - E est l'axiome

Exemple (suite)

Reste à traiter : E → E + T | T * F | (E)
T → T * F | (E)
F → (E)

- E → E + T devient $E \rightarrow E\ X_1$ et $X_1 \rightarrow X_+\ T$

  $N' = \{ E, T, X_1, X_+ \}$ et $P' \leftarrow P' \cup \{ E \rightarrow E\ X_1,\ X_1 \rightarrow X_+\ T\}$

- E → T * F devient $E \rightarrow T\ X_2$ et $X_2 \rightarrow X_* F$

  $N' \leftarrow N' \cup \{ X_2, X_*, F\}$ et $P' \leftarrow P' \cup \{ E \rightarrow T\ X_2,\ X_2 \rightarrow X_*\ T\}$

- E → (E) devient $E \rightarrow X_(\ X_3$ et $X_3 \rightarrow E\ X_)$

  $N' \leftarrow N' \cup \{ X_(, X_3, X_) \}$ et
  $P' \leftarrow P' \cup \{ E \rightarrow X_(\ X_3\ ,\ X_3 \rightarrow E\ X_)\}$

Exemple (suite)

Reste à traiter : T → T * F | (E)
F → (E)

- T → T * F devient $T \rightarrow T\ X_4$ et $X_4 \rightarrow X_* F$

  $N' \leftarrow N' \cup \{ X_4 \}$ et $P' \leftarrow P' \cup \{ T \rightarrow T\ X_4,\ X_4 \rightarrow X_*\ F\}$

- T → (E) devient $T \rightarrow X_(\ X_5$ et $X_5 \rightarrow E\ X_)$

  $N' \leftarrow N' \cup \{ X_5 \}$ et $P' \leftarrow P' \cup \{ T \rightarrow X_(\ X_5,\ X_5 \rightarrow E\ X_) \}$

- F → (E) devient $F \rightarrow X_(\ X_6$ et $X_6 \rightarrow E\ X_)$

  $N' \leftarrow N' \cup \{ X_6 \}$ et $P' \leftarrow P' \cup \{ F \rightarrow X_(\ X_6,\ X_6 \rightarrow E_) \}$

Exemple (fin)

- on ajoute à P' les productions $X_a \to a$, $a \in T$ nécessaires
  $P' \leftarrow P' \cup \{ X_+ \to + , X_* \to * , X_(\to (, X_) \to) \}$

- $G' = (N', T, P', E)$ sous FNC :
  - $N' = \{ E, T, F, X_1, X_2, X_3, X_4, X_5, X_6, X_+, X_*, X_(, X_) \}$
  - $T = \{ +, *, (,), i \}$
  - E est l'axiome
  - P' = { $E \to E\ X_1 \mid T\ X_2 \mid i$
    $T \to T\ X_4 \mid X_(\ X_5 \mid i$
    $F \to X_(\ X_6 \mid i$
    $X_1 \to X_+\ T$
    $X_2 \to X_*\ T$
    $X_3 \to E\ X_)$
    $X_4 \to X_*\ F$
    $X_5 \to E\ X_)$
    $X_6 \to E\ X_)$
    $X_+ \to +$
    $X_* \to *$
    $X_(\to ($
    $X_) \to)$ }