6 – Grammaires hors-contexte

Noam Chomsky, né en 1928, linguiste américain, MIT

Grammaires hors-contexte

- Une grammaire G = (N,T,P,A) est hors-contexte * (ou algébrique) si ses productions sont de la forme :

  $B \rightarrow w$
  avec $B \in N$ et $w \in (N \cup T)^*$

- Un langage L est hors-contexte ou algébrique si il existe une grammaire hors-contexte G telle que :

  $L(G) = L$

Conséquence : Tout langage rationnel est aussi hors-contexte

Réciproque fausse !!!

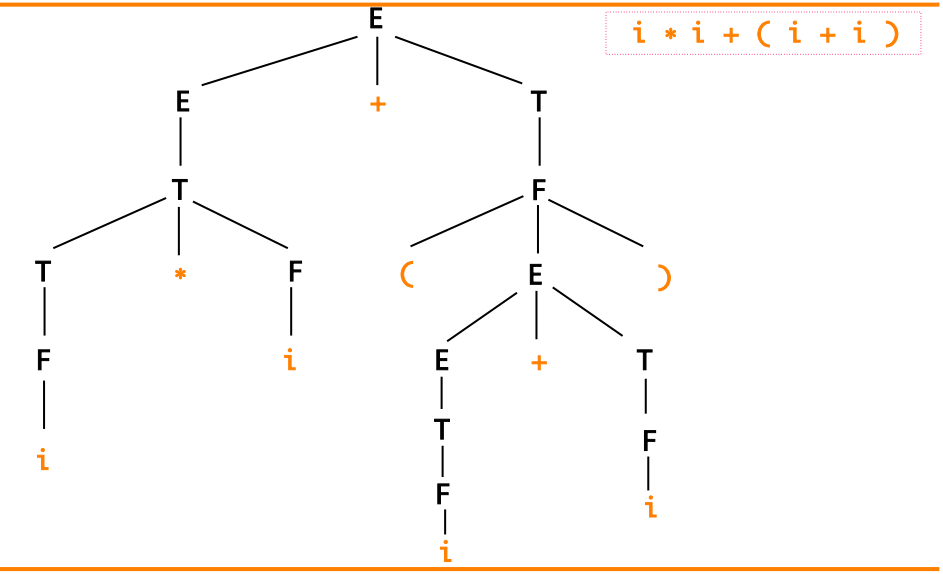
Un langage hors-contexte non rationnel
- n'est pas reconnu par un *automate fini*
- n'est pas décrit par une *expression régulière*
- il n'existe pas de *grammaire régulière* pour l'engendrer.

* *Context-free* (en anglais).

Exemple

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

Arbre de dérivation



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 $X \rightarrow w$

- 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 :
  - $\varphi = w_0$
  - $\psi = w_n$
  - $\forall\ i,\ 0 \le i < n,\quad w_i \Rightarrow_g w_{i+1}$

- 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 de même 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 hors-contexte, le langage L(G) engendré 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 hors-contexte est ambiguë si elle engendre un mot par deux dérivations à gauche différentes
  ↙ on peut alors construire deux arbres de dérivation distincts pour ce mot

Hélas, il n'existe pas de grammaire non-ambiguë pour tout langage hors-contexte ...

Exemple

- G = (N, Σ, P, E) où :
  - N = { E } ensemble des non-terminaux
  - Σ = { +, *, 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 2 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', Σ, P', E) n'est plus ambiguë
- de plus, G' est *équivalente* à la grammaire G c'est-à-dire :

$$L(G) = L(G')$$

  - N' = { E, E' T, T', F }
  - Σ = { +, *, (,), i } comme dans la grammaire G précédente
  - E est l'axiome
  - 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) }
- G' est exempte de *« récursivité à gauche »* mais contient des *« renommages »*.

Extrait de la grammaire Java

```
FloatingPointLiteral:
        Digits . Digits_opt ExponentPart_opt FloatTypeSuffix_opt
        . Digits ExponentPart_opt FloatTypeSuffix_opt
        Digits ExponentPart FloatTypeSuffix_opt
        Digits ExponentPart_opt FloatTypeSuffix
ExponentPart: ExponentIndicator SignedInteger
ExponentIndicator: one of e E
SignedInteger: Sign_opt Digits
Sign: one of + -
FloatTypeSuffix: one of f F d D
DecimalNumeral:
        0
        NonZeroDigit Digits_opt
Digits:
        Digit
        Digits Digit
Digit:
        0
        NonZeroDigit
NonZeroDigit: one of 1 2 3 4 5 6 7 8 9
```


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 +, *, (), { }, ?, & ...

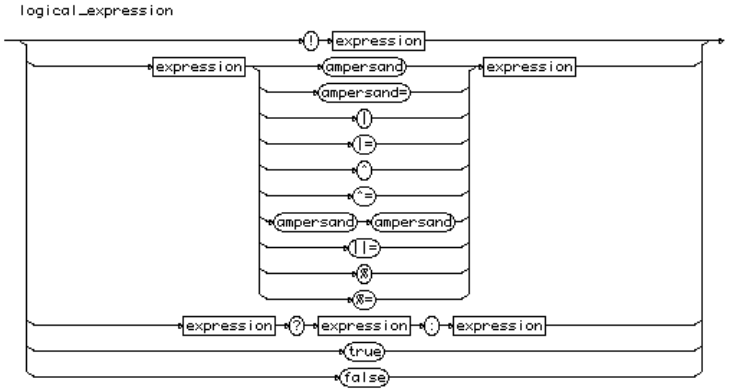
Extrait d'une grammaire sous forme BNF

```
<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>])]
```


Grammaire sous forme EBNF

```
logical_expression
      ::=
      ( "!" expression )
      | ( expression
      ( "ampersand"
      | "ampersand="
      | "|"
      | "|="
      | "^"
      | "^="
      | ( "ampersand" "ampersand" )
      | "||="
      | "%"
      | "%=" )
      expression )
      | ( expression "?" expression ":" expression )
      | "true"
      | "false"
```

Diagramme



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 (linéaires à droite) $A \to a \quad A \to aB \quad A \to \varepsilon$ $A,B \in N \quad a \in T$ et régulières à droite (linéaires à droite)
2	algébriques ou hors-contexte	algébriques hors-contexte $A \to \alpha$ $A \in N \quad \alpha \in V^*$
1	contextuels	contextuelles monotones $\alpha \to \beta$ $\alpha, \beta \in V^+$ $\lvert\alpha\rvert \leq \lvert\beta\rvert$
0	récursivement énumérables	contextuelles avec effacement $\alpha \to \beta$ $\alpha \in V^+, \beta \in V^*$ soit *aucune contrainte aux productions*

En particulier, les grammaires **hors-contexte** savent engendrer les langages **rationnels.**

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 hors-contexte $L = L\setminus\{\varepsilon\}$ 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 (il existe des algo. pour cela), on peut mettre les grammaires sous forme standard : les formes normales.

Forme normale de Chomsky

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

$$A \to a$$
$$A \to B\,C$$
$$S \to \varepsilon$$

avec A,B,C ∈ N et a ∈ T (et si on a S → ε, S ne figure dans aucun membre droit d'une autre production)

- Il existe un algorithme transformant toute grammaire hors-contexte propre en grammaire sous F.N.C.

- Il existe une autre forme normale, tout aussi classique : la forme normale de Greibach dite F.N.G. que l'on verra au prochain cours.

Sheila Greibach, née en 1939
Prof. CS, UCLA

Algorithme de mise sous F.N.C.

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

1 N' ← { S }
P' ← { A → a, A → B C *et éventuellement* S → ε pour A,B,C ∈ N, a ∈ T }

2 pour chaque production de P de la forme :

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

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

avec soit $A_i = \alpha_i$ si $\alpha_i \in N$, soit $A_i = X_{\alpha_i}$ si $\alpha_i \in T$

3 pour chaque X_{α_i} créé précédemment, on ajoute à P' :

$$X_{\alpha_i} \to \alpha_i$$

Algorithme (suite)

4 N' reçoit les non-terminaux A_i, X_i et X_{α_i} précédemment créés

5 on remplace les productions de P de la forme :

$$A \to \alpha_1\,\alpha_2 \qquad \text{avec } \alpha_1 \text{ ou } \alpha_2 \in T$$

par des productions dans P' de la forme :

$$A \to Y_1\,Y_2$$

si $\alpha_1 \in T$ alors $Y_1 \to \alpha_1$ sinon $Y_1 = \alpha_1$
si $\alpha_2 \in T$ alors $Y_2 \to \alpha_2$ sinon $Y_2 = \alpha_2$

6 N' reçoit les Y_i précédents.

Exemple

- G = (N, Σ,P,E) une grammaire *propre* :
  - N = { E, T, F }
  - Σ = { +, *, (,), i }
  - P = { E → E + T | T * F | (E) | i
    T → T * F | (E) | i
    F → i
    F → (E) }
  - E axiome

- G' = (N', Σ',P',E) où pour commencer :
  - N'= { E, ... }
  - Σ' = { +, *, (,), i }
  - P = { E → i
    T → i
    F → i }
  - E axiome

Exemple (suite)

Reste à traiter : $E \to E + T \mid T * F \mid (E)$
$T \to T * F \mid (E)$
$F \to (E)$

- $E \to E + T$ devient $E \to E\ X_1$ et $X_1 \to X_+\ T$

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

- $E \to T * F$ devient $E \to T\ X_2$ et $X_2 \to X_*F$

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

- $E \to (E)$ devient $E \to X_(\ X_3$ et $X_3 \to E\ X_)$

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

Exemple (suite)

Reste à traiter : $T \to T * F \mid (E)$
$F \to (E)$

- $T \to T * F$ devient $T \to T\ X_4$ et $X_4 \to X_*\ F$

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

- $T \to (E)$ devient $T \to X_(\ X_5$ et $X_5 \to E\ X_)$

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

- $F \to (E)$ devient $F \to X_(\ X_6$ et $X_6 \to E\ X_)$

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

- on ajoute à P' les productions $X\alpha_i \to \alpha_i$, $\alpha_i \in T$ nécessaires :

  $P' \leftarrow P' \cup \{ X_+ \to +\ ,\ X_* \to *\ ,\ X_(\to (\ ,\ X_) \to) \}$

Exemple (fin)

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

Récursivité à gauche

- Une grammaire est récursive à gauche dès qu'elle contient une production de la forme :

  $B \to B\ w$
  avec $B \in N$ et $w \in (N \cup T)^*$

- la récursivité à gauche provoque des exécutions infinies ... mais elle se retire facilement

- *il peut arriver que du même coup, on supprime l'ambiguïté*

Théorème: Pour toute grammaire hors-contexte récursive à gauche, il existe une grammaire hors-contexte équivalente exempte de récursivité à gauche.

Suppression de la récursivité à gauche

En la supprimant, on la transforme alors en récursivité à droite, non gênante

- prenons un sous-ensemble de productions :

$$B \to B\,\alpha$$
$$B \to \beta$$

avec $B \in N$, α, $\beta \in (N \cup T)^*$
et β ne commence pas par B

- il engendre le langage :

$$\beta\,\alpha^*$$

- ce langage est aussi engendré par :

$$B \to \beta \mid \beta\,Z$$
$$Z \to \alpha \mid \alpha\,Z$$

avec $Z \notin (N \cup T)^*$

Exemple

- Reprise : G = (N,T,P,E) est ambiguë et récursive à gauche :
  - N = { E }
  - E axiome
  - T = { +, *, i, (,) }
  - P = { E → E + E
    E → E * E
    E → i
    E → (E) }

- en ôtant la récursivité à gauche de G, on obtient G' = (N',T,P',E) :
  - N' = { E, Z, Z' }
  - T = { +, *, (,), i }
  - E axiome
  - P = { E → i | i Z
    E → (E) | (E) Z
    Z → + E | + E Z
    E → i Z'
    E → (E) Z'
    Z' → * E | * E Z' }

- G' n'est plus récursive à gauche ... est-elle ambiguë ?