Compilation

III. Grammaires non-contextuelles

Jacques Farré

Jacques.Farre@unice.fr

http://deptinfo.unice.fr/~jf/Compil-L3

Rappels de quelques définitions

Alphabets et mots

- un *alphabet* X (ou un *vocabulaire* V) est un ensemble fini de symboles atomiques
  - on emploie de préférence *alphabet* si les symboles sont simples, *vocabulaire* s'ils sont plus complexes
  - $X = \{0, 1\}$, $V = \{le, la, chat, souris, mange\}$
  - en analyse syntaxique, on emploie aussi le terme de *lexème* ou de *terminal*, ou encore de *token*
- un *mot* sur X ou une *chaîne* sur V est une suite **finie** de symboles de X ou de V
  - 010110, *le souris mange la chat*
  - a^k désigne le mot formé de k fois la lettre a
- le nombre de symboles d'un mot (ou d'une chaîne) φ est appelé sa *longueur* et noté $|\varphi|$
- le *mot vide*, c'est-à-dire de longueur nulle, sera noté ε

Rappels de quelques définitions

monoïdes

- un *monoïde* sur X (resp. V) est l'ensemble des mots (resp. chaînes) sur X, muni
  - d'une loi de concaténation associative (notée en général par la simple juxtaposition)
  - d'un élément neutre à gauche et à droite, le *mot vide*, noté ε : $\varepsilon\omega = \omega\varepsilon = \omega$
- X^i est l'ensemble des mots de longueur i
  - par exemple, avec $X = \{0, 1\}$, X^8 est l'ensemble des valeurs possible d'un octet
- le *monoïde libre* sur X, noté X^*, est l'ensemble de tous les ensembles de mots de longueur quelconque :

$$X^* = \bigcup_i X^i \cup \{\varepsilon\}, i > 0$$

- $X^+ = X^* - \{\varepsilon\}$: par exemple, avec $X = \{0, 1\}$, X^+ est l'ensemble des nombres binaires

Rappels de quelques définitions

Langages formels

- un *langage formel* est un sous-ensemble, fini ou non, de X^* ; ses éléments sont des *phrases*
  - exemple avec $X = \{0, 1\}$
    - $L_1 = \{0^*1^*\}$ (un nombre quelconque de 0 suivis d'un nombre quelconque de 1)
    - $L_2 = \{0^n1^n \mid n \geq 0\}$ (n 0 suivis de n 1)
    - $L_3 = \{\varphi\tilde{\varphi} \mid \varphi \in X^*\}$ (par exemple 0101 1010)
    - $L_5 = \{0^n1^n0^n \mid n \geq 0\}$ (langage non algébrique)
- si φ, α, β et ω sont des mots sur X^* (des chaînes sur V^*) :
  - si $\varphi = \alpha\beta$, α est un *préfixe* (ou une *tête*) de φ ; c'est un *préfixe propre* si $\beta \neq \varepsilon$: *tr* est un préfixe de *truc*
    - de manière analogue, β est un *suffixe* (ou une *queue*) de φ ; c'est un *suffixe propre* si $\alpha \neq \varepsilon$: *uc* est un suffixe de *truc*
    - si $\varphi = \alpha\beta\omega$, β est un *facteur* de φ : *ru* est un facteur de *truc*
  - β est un *sous-mot* (une *sous-chaîne*) de φ s'il est formé d'une suite de symboles extraits de φ par effacement : *tu* est un sous-mot de *truc*

Rappels de quelques définitions

Systèmes de réécriture et dérivations

- une *production* (ou *règle*) est une paire (φ, ψ) de mots sur $X^* \times X^*$, notée $\varphi \rightarrow \psi$ (φ produit ψ)
  - φ sera appelé *partie gauche* et ψ *partie droite*
- une *dérivation*, notée $\Rightarrow$, est l'application d'une règle permettant de réécrire un mot : par exemple $\alpha\varphi\beta \Rightarrow \alpha\psi\beta$ par application de la règle $\varphi \rightarrow \psi$
  - on dit que $\alpha\varphi\beta$ est *directement dérivable* de $\alpha\psi\beta$
  - ou encore que $\alpha\psi\beta$ *se réduit directement* à $\alpha\varphi\beta$
- la fermeture transitive de $\Rightarrow$ est notée $\Rightarrow^*$ (on applique des règles un certain nombre de fois, y compris 0 fois)
  - si $\alpha \Rightarrow^* \beta$, on dit que α *se dérive* en β (ou que β *se réduit* à α)
- on écrira $\Rightarrow^+$ si on applique des règles au moins une fois
- le couple (V, P) formé du vocabulaire V et de l'ensemble fini des productions P est un *système de réécriture*

Exemple de dérivations

- soient les règles

  $\{S \to aB, S \to bA, S \to \varepsilon, aB \to abS, aB \to aSb, bA \to baS, bA \to bSa\}$

- on a par exemple les dérivations suivantes

  $$S \Rightarrow aB \Rightarrow abS \Rightarrow abbA \Rightarrow abbSa \Rightarrow abbbAa \to abbbSaa \to abbbaa$$

  ou

  $$S \Rightarrow bA \to baS \to babA \to babaS \to babaaB \to babaabS \to babaab$$

- exercice : montrer que ces règles permettent d'engendrer tous les mots qui ont autant de a que de b

Rappels sur les grammaires

- une grammaire G est un quadruplet (N, T, P, A) tel que :
  - N est un ensemble fini de *symboles non-terminaux*
  - T est un ensemble fini de *symboles terminaux* ($T \cap N = \emptyset$)
  - P est un ensemble fini de *règles* (ou *productions*) dans $V^+ \times V^*$, avec $V = N \cup T$
  - A est l'*axiome*, $A \in N$
- une grammaire est un système de réécriture
- le langage engendré par une grammaire $G = (N, T, P, A)$ est $L(G) = \{w \mid w \in T^*, A \Rightarrow^+ w\}$
- 2 grammaires sont *équivalentes* si elles engendrent le même langage

Hiérarchie de *Chomsky*

- **type 0 :** grammaires *contextuelles avec effacement*, engendrant les langages *récursivement énumérables*

$$P = \{\sigma \to \tau \mid \sigma \in V^+, \tau \in V^*\}$$

- **type 1 :** Grammaires *contextuelles* ou *monotones*, engendrant les langages *contextuels*

$$P = \{\mu B \varphi \to \mu \psi \varphi \mid \mu, \varphi \in V^*; B \in N; \psi \in V^+\}$$

- **type 2 :** Grammaires *non-contextuelles*, engendrant les langages *algébriques* (ou non-contextuels)

$$P = \{B \to \psi \mid B \in N; \psi \in V^*\}$$

- **type 3 :** Grammaires *régulières*, engendrant les langages *rationnels* (ou réguliers)
  - règles restreintes à $A \to a$ ($a \in T \cup \{\varepsilon\}$) ou
    - *régulières à droite* : $A \to aB$ ($a \in V, B \in N$)
    - *régulières à gauche* : $A \to Ba$ ($a \in V, B \in N$)

Exemple de grammaire

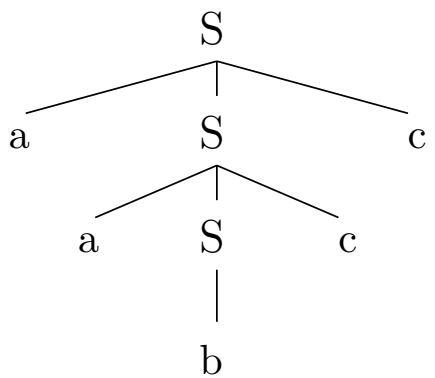
- soit la grammaire contextuelle

$$N = \{Z, S, P, A, D, N, C, V, W\},\ T = \{a : z\} \cup \{\_\},\ axiome = Z,$$
$$P = \{Z \to SANV,\ Z \to PANV,$$
$$SA \to DS,\ PA \to DP,$$
$$SN \to SCS,\ PN \to PCP,$$
$$SV \to SW,\ PV \to PWnt$$
$$D \to le,\ C \to chat,\ W \to miaule,\ S \to \_,\ P \to s\_\}$$

  (_ signifiant un espace)
- cette grammaire ne produit que 2 phrases :
  1. *le chat miaule* :
     $Z \Rightarrow SANV \Rightarrow DSNV \Rightarrow DSCSV \Rightarrow DSCSW \Rightarrow^{+} leSchatSmiaule$,
     et comme $S \to \_$ on a bien *le chat miaule*
  2. *les chats miaulent* : de manière identique,
     $Z \Rightarrow PANV \Rightarrow DPNV \Rightarrow DSCPV \Rightarrow DPCPWent \Rightarrow^{+} lePchatPmiaulent$,
     et comme $P \to s\_$ on a bien *les chats miaulent*

Rappels sur les grammaires non-contextuelles

- du type 2 de *Chomsky* : règles restreintes à $N \times V^*$, donc du genre $A \to \varphi$; elles engendrent des langages *algébriques*
- *arbre de dérivation* (ou *arbre syntaxique*)
  - racine = l'axiome S
  - feuilles = symboles de $T \cup \{\varepsilon\}$
  - nœuds = symboles de N ; les fils d'un nœud A sont les X_i tels que $A \to X_1 X_2 \ldots X_n$ est une règle
  - si w est le mot résultant de la concaténation des feuilles, alors $S \Rightarrow^* w$
  - par exemple, avec les règles $\{S \to aSc, S \to b\}$, *aabcc* est un mot du langage, qui a pour arbre de dérivation



Conventions de notations pour la suite

- symboles et mots
  - lettres latines capitales du début de l'alphabet (A, $B\ldots$) pour les symboles de N ; exception : axiome S
  - lettres latines minuscules du début de l'alphabet (a, $b\ldots$) pour les symboles de T
  - lettres latines capitales de la fin de l'alphabet (W, $X\ldots$) pour les symboles de $V = N \cup T$
  - lettres latines minuscules de la fin de l'alphabet (u, $v\ldots$) pour les mots de T^*(donc un mot éventuellement vide)
  - lettres grecques minuscules (α, $\beta\ldots$) pour les mots de V^*
- règles
  - $A \to \alpha_1 | \alpha_2 | \cdots | \alpha_n$ pour l'ensemble $\{A \to \alpha_1, A \to \alpha_2, \cdots A \to \alpha_n\}$
- dérivations
  - dérivation la plus à droite : $\underset{\text{rm}}{\Rightarrow}$ (et $\underset{\text{rm}}{\Rightarrow}^*$, $\underset{\text{rm}}{\Rightarrow}^+$), par exemple $\alpha AAw \underset{\text{rm}}{\Rightarrow} \alpha A\beta w$
  - dérivation la plus à gauche : $\underset{\text{lm}}{\Rightarrow}$ ($\underset{\text{lm}}{\Rightarrow}^*$, $\underset{\text{lm}}{\Rightarrow}^+$), par exemple $wAA\alpha \underset{\text{rm}}{\Rightarrow} w\beta A\alpha$

Encore quelques définitions ...

- on appelle *forme sentencielle* toute chaîne φ telle que $S \Rightarrow^* \varphi$, $\varphi \notin T^*$; si $\varphi \in T^*$, on dira que c'est une *phrase*
  - exemple : dans $S \Rightarrow cSeS \Rightarrow cieS \Rightarrow ciei$, *ciei* est une phrase et les autres chaînes des formes sententielles
- un nonterminal A est *annulable* s'il existe une dérivation $A \Rightarrow^* \varepsilon$; il est dit *nul* si toutes ses dérivations produisent ε
  - la notion s'étend à une forme sentencielle φ
- une règle de la forme $A \rightarrow \alpha$ est *récursive* si $\alpha \Rightarrow^* \varphi A \psi$ (et directement récursive si $\alpha = \varphi A \psi$)
  - récursive gauche si $\varphi = \varepsilon$, droite si $\psi = \varepsilon$
  - récursive *cachée* gauche si $\varphi \Rightarrow^* \varepsilon$, droite si $\psi \Rightarrow^* \varepsilon$
  - par exemple $\{S \rightarrow ASa, S \rightarrow b, A \rightarrow \varepsilon\}$ qui produit le langage $\varepsilon^n ba^n$, $N \geq 0$ contient une récursivité gauche cachée
- une grammaire est *cyclique* s'il existe une dérivation $A \Rightarrow^+ A$ (et notamment quand $A \Rightarrow^+ \varphi A \psi$, φ et ψ annulables)

... et d'autres définitions

- un symbole X est *génératif* s'il existe $w \in T^*$ tel que $X \Rightarrow^* w$
- un symbole X est *accessible* s'il existe α, β tels que $S \Rightarrow^* \alpha X \beta$
- un symbole X est *utile* s'il est génératif et accessible ; sinon il est inutile
- dans $S \rightarrow AB$, $A \rightarrow a$, $C \rightarrow b$, seuls A et S sont des symboles utiles
  - B n'est pas génératif
  - C n'est pas accessible
- une grammaire est *bien formée* si elle ne contient que des symboles utiles et n'a pas de cycles
  - on supposera pour le reste du cours que l'on travaille uniquement sur des grammaires bien formées

Notation des grammaires

- Notation *BNF* (Backus Normal Form) : productions uniquement de la forme $A \rightarrow \alpha$, par exemple

$$\texttt{expr} \rightarrow \texttt{expr + terme}$$
$$\texttt{expr} \rightarrow \texttt{expr - terme}$$
$$\texttt{expr} \rightarrow \texttt{terme}$$
$$\texttt{expr} \rightarrow \texttt{- terme}$$

- Notation *EBNF* (Extended BNF) : productions acceptant des expressions régulières en partie droite
  - $\alpha \mid \beta \Rightarrow$ soit α, soit β
  - $[\alpha] \Rightarrow \alpha$ optionnel, noté aussi parfois $(\alpha)?$
  - $(\alpha)^* \Rightarrow$ 0, 1 ou plusieurs α (on peut parfois aussi utiliser $(\alpha)^+$)

  par exemple

$$\texttt{expr} \rightarrow \texttt{[-] terme ((+|-) terme)*}$$

Transformation de grammaires

- ces transformations ont pour objectif de transformer la grammaire en une autre, **sans évidemment modifier le langage engendré**
- elles peuvent être rendues nécessaires si l'on a une grammaire du langage (dans un manuel de définition par exemple) qui n'est pas compatible avec la méthode que l'on compte utiliser pour produire l'analyseur syntaxique
- les plus courantes
  - passage de l'EBNF à la BNF
  - transformation des récursivités droites en récursivités gauches (et inversement)
  - factorisation à gauche
  - élimination des productions vides

Transformation simples de grammaires

transformation EBNF $\rightarrow$ BNF

- $[\alpha]$: $A' \rightarrow \alpha$, $A' \rightarrow \varepsilon$, mais on peut expanser A' dans la partie droite
  - $A \rightarrow \beta[\alpha]\gamma$: $A \rightarrow \beta\gamma$, $A \rightarrow \beta\alpha\gamma$
  - par exemple `expr` $\rightarrow$ `[-] terme ( (+|-) terme )*` devient

    `expr` $\rightarrow$ `terme ( (+|-) terme )*`

    `expr` $\rightarrow$ `- terme ( (+|-) terme )*`
- $(\alpha)^*$: $A' \rightarrow A'\alpha$ (ou $A' \rightarrow \alpha A'$), $A' \rightarrow \varepsilon$
  - $A \rightarrow \beta(\alpha)^*$: $A \rightarrow \beta$, $A \rightarrow A\alpha$
  - par exemple `expr` $\rightarrow$ `terme ( (+|-) terme )*` devient

    `expr` $\rightarrow$ `terme`

    `expr` $\rightarrow$ `expr (+|-) terme`
- $(\alpha \mid \beta)$ devient $A' \rightarrow \alpha$, $A' \rightarrow \beta$
  - $A \rightarrow \gamma(\alpha \mid \beta)\rho$ devient $A \rightarrow \gamma\alpha\rho$, $A \rightarrow \gamma\beta\rho$
  - par exemple `expr` $\rightarrow$ `expr (+|-) terme` devient

    `expr` $\rightarrow$ `expr + terme`

    `expr` $\rightarrow$ `expr - terme`

Transformation simples de grammaires

transformation des récursivités

- on passe facilement d'une récursivité droite directe à une récursivité gauche directe (et inversement)
  - en effet, si on a $A \rightarrow \alpha A \mid \beta$, on voit que $A \Rightarrow^+ \alpha^* \beta$, et il suffit de réécrire les règles comme $A \rightarrow A'\beta, A' \rightarrow A'\alpha \mid \varepsilon$
  - pour $A \rightarrow A\alpha \mid \beta$, on voit que $A \Rightarrow^+ \beta\alpha^*$, qui peut être aussi produit par $A \rightarrow \beta A', A' \rightarrow \alpha A' \mid \varepsilon$
  - pour les récursivités indirectes
    - indirecte gauche $A \Rightarrow B\alpha \Rightarrow^+ A\beta\alpha$ par exemple : pour toute règle $B \rightarrow \gamma$, transformer $A \rightarrow B\alpha$ en $A \Rightarrow \gamma\alpha$ et recommencer jusqu'à obtenir une récursivité directe $A \rightarrow A\rho$, transformée comme précédemment
  - un grand classique, les expressions arithmétiques $a + a \cdots + a$

$$E \rightarrow E + T \mid T, T \rightarrow a$$

$$E \rightarrow T\ S, T \rightarrow a, S \rightarrow \varepsilon \mid +E$$

Transformation simples de grammaires

factorisation des préfixes communs

- factorisation à gauche : des règles comme $S \rightarrow \alpha\beta \mid \alpha \mid \gamma$, dont une partie droite est préfixe d'une autre, peuvent être facilement modifiées en $S \rightarrow \alpha A \mid \gamma, A \rightarrow \beta \mid \varepsilon$
  - un grand classique, *if then else*

    instruction $\rightarrow$ **if** *condition instruction*
    $\mid$ **if** *condition instruction* **else** *instruction*
    $\mid$ **while** *condition instruction*

    devient

    instruction $\rightarrow$ **if** *condition instruction* E
    $\mid$ **while** *condition instruction*

    $E \rightarrow \varepsilon \mid$ **else** *instruction*

Transformation simples de grammaires

Élimination des productions vides

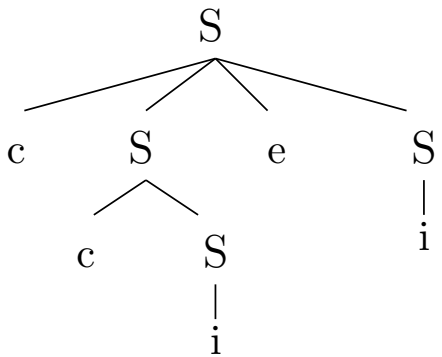
- élimination des symboles nuls : si on a $A \rightarrow \alpha E \beta, E \rightarrow \varepsilon \mid \gamma$ (noté aussi $A \rightarrow \alpha[\gamma]\beta$), alors $A \rightarrow \alpha\gamma\beta \mid \alpha\beta$ engendre le même langage $L(A)$
  - par exemple pour SQL,la requête *select* peut être informellement définie par :

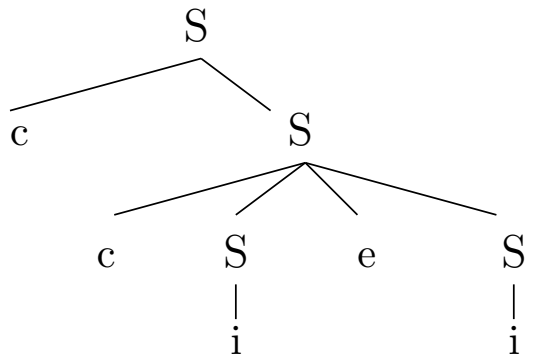
    SELECT [DISTINCT] exp1 [[AS] nom1], exp2 [[AS] nom2], ...
  - qui peut être décrite par la grammaire

    select → **SELECT** *expressions*
    | **SELECT DISTINCT** *expressions*
    expressions → *expression sexpressions*
    sexpressions → ε | , *expression sexpressions*
    expression → *expr* **AS** *nom*
    | *expr nom*
    | *expr*

Grammaires ambiguës

- une grammaire est ambiguë s'il existe 2 arbres de dérivation différents pour un même mot ; exemple avec $S \rightarrow cSeS|cS|i$





c'est le problème du `if-else` de nombreux langages

```
if condition
  if condition
     instruction
  else
     instruction
```

```
if condition
  if condition
    instruction
else
  instruction
```

- en d'autres termes, il existe 2 dérivations gauches (ou droites) différentes qui produisent le même mot
- **le problème de savoir si une grammaire non-contextuelle est ambiguë est indécidable** (dans le cas général)

Ambiguïté et nombre d'arbres

- une grammaire ambigüe peut produire un nombre énorme d'arbres
- par exemple avec $\{E \rightarrow E + E \mid a\}$
  - $a+a$: un seul arbre
  - $a+a+a$: deux arbres correspondant à $[a + a] + a$ et $a + [a + a]$
  - $a+a+a+a$: 4 arbres ($[[a + a] + a] + a$, $[a + a] + [a + a]$, $a + [[a + a] + a]$ et $[a + [a + a]] + a$
  - donc, si on a n + dans l'expression, 2^{n-1} arbres !
- problématique pour le traitement des langues naturelles, pour lesquelles les grammaires sont nécessairement ambigües (entre autre car les langues le sont), avec n assez grand (10 à 20 mots par phrase)

Comment concevoir une grammaire non contextuelle ?

pour un langage de programmation, de description . . .

- un premier principe : ne pas essayer de mettre des aspects contextuels dans une grammaire non contextuelle
  - ces aspects seront traités par la passe de *sémantique statique*
- un second principe : être général est souvent plus simple que prendre les choses par le petit bout de la lorgnette
  - y compris accepter **syntaxiquement** un langage plus large, quitte à restreindre dans la sémantique statique
- définir les constructions principales du langage : par exemple, pour un langage de programmation, *déclarations* (de type, de variable, de fonction), *expressions*, *instructions*, *désignations* de variable . . .

Quelques exemples

- ne pas essayer de mettre des aspects contextuels
  - en C, `int t[3*N + M]` est autorisé à condition que `N` et `M` soient des constantes, mais il est difficile de définir syntaxiquement ce qu'est une expression constante $\Rightarrow$ accepter syntaxiquement une expression et vérifier plus tard que tous ses opérandes représentent des constantes
- être général est souvent plus simple
  - déclaration de méthode Java (extrait de la doc de Javacc)

```
void MethodDeclaration() : {} {
  ( "public" | "protected" | "private" | "static" |
    "abstract" | "final" | "native" | "synchronized" )*
  ResultType() MethodDeclarator() [ "throws" NameList() ]
  ( Block() | ";" )
}
```

    on accepte les répétitions (et même des mots contradictoires comme `public` et `private`) : plus facile que de prévoir toutes les combinaisons légales et on règle la question par un tableau de bits par exemple

Conception (partielle) d'une grammaire pour C

le niveau global

- un programme C est suite de déclarations (variables, types ou fonctions)

```
programmeC ::= ( declaration )*
declaration ::= declVars | declType | declFonc | defFonc
```

- une déclaration de variables (oublions pointeurs et tableaux) est une suite de noms avec éventuellement une initialisation

```
declVars ::= type declVar ( "," declVar )*
declVar ::= IDENT [ "=" expr ]
```

- une déclaration/définition de fonction a des paramètres et éventuellement un corps

```
declFonc ::= entete ";"
defFonc ::=  entete bloc
entete ::= ( type | "void" ) IDENT "(" [parFormels] ")"
parFormels ::= parFormel ( "," parFormel)*
parFormel ::= type [ IDENT ]
```

Conception (partielle) d'une grammaire pour C

les blocs

- un bloc est une suite d'instructions précédée éventuellement de déclarations

  ```
  bloc ::= "{" ( declBloc )* ( instruction )* "}"
  declBloc ::= declVar | declType | declFonc
  ```

  on aurait pu utiliser *declaration* : on aurait alors accepté syntaxiquement une définition de fonction dans un bloc
  - ce qui aurait l'avantage de ne pas provoquer une erreur syntaxique difficilement rattrapable si le cas arrivait
  - on peut toujours positionner un booléen si on a une définition de fonction, et émettre un message d'erreur (qui de plus sera plus explicite, du genre *"on n'a pas le droit de définir une fonction dans un bloc"*, plutôt que *"{ non attendu"*)

Conception (partielle) d'une grammaire pour C

les instructions

- on a évidemment plusieurs types d'instructions

```
instruction ::= bloc | ( [expression ] ";" ) |
                si | tantque | ...
```

  - l'affectation est une expression en C et `a + b;` est légal bien que sans intérêt
  - comme un appel de fonction est une expression, on règle le cas `F(...);`
  - le fait que l'expression soit optionelle autorise l'instruction vide (bien utile dans `while (t[i++] != n);`)
  - évidemment, si seuls l'affectation et l'appel sont autorisés, il faudra écrire `( ( affectation | appel ) ";" )` au lieu de `( expression ";" )`
    et `( [ affectation | appel ] ";" )` si on veut garder l'instruction vide

Conception (partielle) d'une grammaire pour C

l'instruction si-sinon

- le *si* est simple

```
si ::= "if" "(" condition ")" instruction
                [ "else" instruction ]
```

  puisqu'un bloc est une instruction, on règle sans problème la question des `{...}`

- la grammaire est ambiguë : elle provoque un conflit quelque soit le générateur d'analyseur utilisé : en général, la résolution par défaut des conflits associe le `else` au `if` le plus proche qui n'a pas de `else`, ce que l'on veut
- une grammaire non ambiguë (pas LL(k), mais LALR(1))

```
instruction ::= complet | incomplet
complet ::= "if" "(" condition ")" complet "else" complet | ...
incomplet ::= "if" "(" condition ")"
                ( instruction | ( complet "else" incomplet )
```

  les instructions complètes sont toutes les instructions sauf les si sans sinon et les si avec sinon qui contiennent des si sans sinon

Conception (partielle) d'une grammaire pour C

les conditions

- en général, ne pas essayer de "spécialiser" les expressions en distinguant les conditions (booléennes), les indices de tableaux (entières) ...
- on peut en effet écrire (entre autres exemples)

```
if (b) ...
if (a == b + 1) ...
if (estPair(i) && b) ...
while ( (c=getchar()) != EOF) ...
```

  car ce qui compte, ce n'est pas la forme syntaxique de l'expression, mais son type
- donc `condition ::= expression`
  - en Java, une affectation n'est pas une expression, mais une instruction :
    `affectation ::= ( variable "=" )+ expression`

Conception (partielle) d'une grammaire pour C

les expressions

- sauf à écrire une grammaire ambiguë (si le générateur d'analyseur utilisé donne des moyens de lever les ambiguïtés, par exemple bison mais pas JavaCC), il est préférable d'écrire des règles qui reflètent la priorité et l'associativité des opérateurs, par exemple

```
expression ::= termBool ( "||" termBool )*
termBool ::=  factBool ( "&&" factBool )*
factBool ::= expr [ ("=="|"!="|"<"|"<="|">"|">=") expr ]
expr ::= terme ( ( "+" | "-" ) terme )*
terme ::= fact ( ( "*" | "/" | "%" ) fact )*
fact ::= litteral | appel | variable | ("(" expression ")")
```

  - ce qui permet **syntaxiquement** d'accepter des phrases comme `"abc" || a + 1 && 3.0 % (a == b)`
  - il serait trop fastidieux d'écrire une grammaire qui ne permettrait pas, par exemple, `"abc" % true`
  - c'est la sémantique statique qui vérifiera que le type de chaque opérande est correct pour son opérateur

Conception (partielle) d'une grammaire pour C

les variables

- les variables ne sont pas seulement de simples identificateurs
  - ce sont aussi des variables indicées (`t[exp]`), des sélections de champs (`a.b.c`), des déréférenciations de pointeurs (`*p`), et possiblement une combinaison de tout ça : `*t[i].a.b[j]`, et aussi `(*t[i]).a.b[j]`
  - donc (en oubliant que `*p++` est aussi possible, parmi bien d'autres)

```
variable ::= pvar
           | '*' variable
pvar     ::= svar
           | pvar "[" expression "]"
           | pvar "." IDENT
svar     ::= IDENT
           | "(" variable ")"
```

Conception (partielle) d'une grammaire

les variables dans un langage à objets

- dans un langage à objets, une méthode pouvant retourner un objet, on peut avoir `this.a.M(i,j).b.F(k).c`

- donc (sans les tableaux) :

```
variable ::= prefixe var
appel ::= prefixe app
var ::= IDENT
app ::= IDENT "(" [ arguments ] ")"
prefixe ::= [ THIS "." ] ( ( var | app ) "." )*
arguments ::= expression ( "," expression )*
```

Analyse syntaxique des langages algébriques

- on peut reconnaître un mot d'un langage algébrique en *interprétant* sa grammaire
  - c'est le cas de l'algorithme *CYK* que vous avez déjà vu (son inconvénient est qu'il faut mettre la grammaire en *forme normale de Chomsky*
  - on peut aussi utiliser l'algorithme de *Earley* qui ne nécessite pas de transformation de la grammaire
  - dans les 2 cas, algorithme cubique en temps sur la longueur du mot (dans le pire des cas)
- on peut *compiler* la grammaire vers un automate à pile
  - solution plus efficace car en général reconnaissance linéaire en temps et en espace sur la longueur du mot
  - deux types d'automates selon la manière dont ils construisent (virtuellement) l'arbre de dérivation : de haut en bas (analyse *descendante*) ou de bas en haut (analyse *ascendante*)

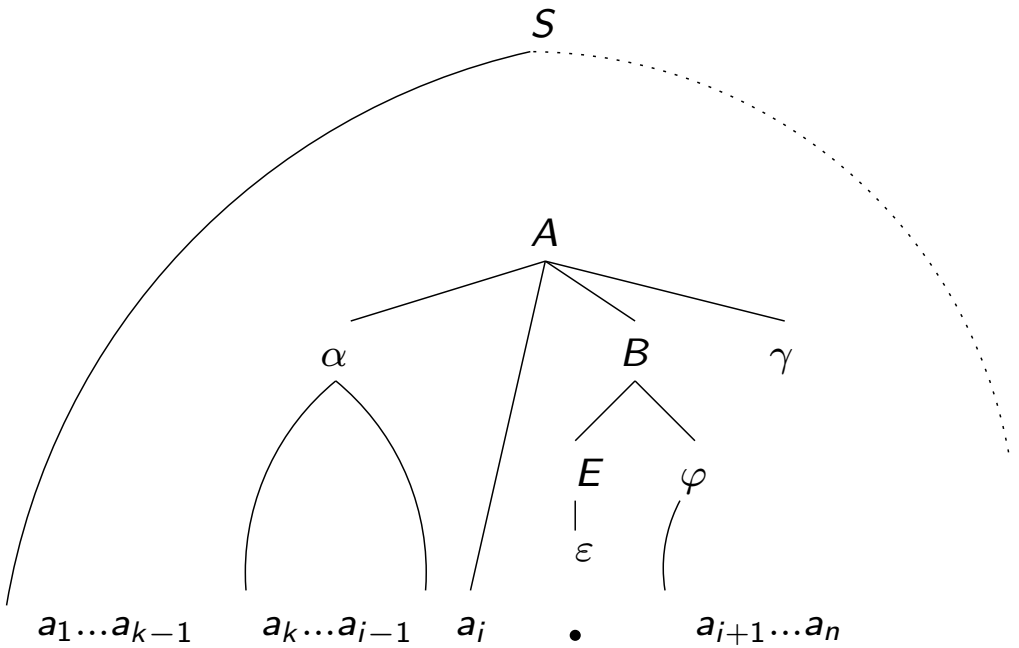
Méthode d'Earley

- pour une entrée $a_1 \cdots a_n$, l'algorithme construit une suite d'états $q_0 \cdots q_n$ tels que
  - en q_0, aucun caractère n'a encore été lu
  - $q_{i-1} \stackrel{a_i}{\Longrightarrow} q_i$ (on passe de q_{i-1} à q_i quand on lit a_i), donc dans l'état q_i, on a lu $a_1 \cdots a_i$
  - à chaque état q_i est associé un ensemble d'*items* ; un item est de la forme $[A \to \alpha \cdot \beta, k]$, qui signifie que
    - on est dans la reconnaissance d'une portion de texte engendrée par A, portion qui commence à partir de a_k
    - on a reconnu α (qui couvre donc $a_k \cdots a_i$)
    - on s'apprête à reconnaître β (si $\beta \neq \varepsilon$), qui couvre donc $a_{i+1} \cdots a_j$
  - dans un ensemble d'items, on distingue
    - le sous-ensemble, appelé *complets*, des items de la forme $[A \to \alpha \cdot, k]$
    - le sous-ensemble, appelé *actifs*, des items de la forme $[A \to \alpha \cdot X \beta, k]$

Calcul des états dans la méthode d'Earley - I

- une transition $q_{i-1} \overset{a_i}{\Longrightarrow} q_i$ a lieu si q_{i-1} contient au moins un item de la forme $[A \rightarrow \alpha \cdot a_i \beta, k]$; sinon le mot n'appartient pas au langage décrit par la grammaire
- le *noyau* de q_i est formé des items $[A \rightarrow \alpha a_i \cdot \beta, k]$, et
  - si $\beta = B\gamma$, il faut faire une *fermeture*, c'est-à-dire ajouter à q_i les items $[B \rightarrow \cdot \varphi, i+1]$ (on s'apprête à reconnaître φ à partir du prochain terminal a_{i+1})
  - si $\beta = \varepsilon$, il faut *compléter*, c'est-à-dire ajouter à q_i les items $[C \rightarrow \gamma A \cdot \varphi, k']$, tels qu'il existe $[C \rightarrow \gamma \cdot A\varphi, k']$ dans q_{k-1}
  - et ceci récursivement selon la forme de φ
  - le traitement des productions vides est donc pris en charge automatiquement
- l'entrée $a_1 \cdots a_n$ appartient au langage si et seulement si q_n contient $[S \rightarrow \sigma \cdot, 1]$

Items d'Earley et arbre de dérivation



$A \to \alpha a_i B \gamma$

$B \to E \varphi$

$E \to \varepsilon$

$[A \to \alpha a_i \cdot B\gamma, k]$

$[B \to \cdot E\varphi, i+1]$

$[E \to \cdot, i+1]$

$[B \to E \cdot \varphi, i+1]$

on voit bien sur l'arbre que quand le $\cdot$ est entre a_i et a_{i+1}, il est aussi avant B, et puisque E produit le vide, avant φ.

Calcul des états dans la méthode d'Earley - I

- fermeture et complétion : c'est l'ensemble minimum défini par

$$C(q_i) = q_i$$
$$\cup \{[B \to \cdot\gamma, i+1] \mid [A \to \alpha \cdot B\beta, k] \in C(q_i), B \to \gamma \in P\}$$
$$\cup \{[B \to \gamma A \cdot \varphi, k'] \mid [A \to \alpha \cdot, k] \in C(q_i), [B \to \gamma \cdot A\varphi, k'] \in q_{k-1}\}$$

- ou autrement dit

```
C (EnsItems noyau, entier i, SuiteItems etats) -> res : EnsItems
  res := noyau
  repeter
    pour tout nouvel item qui vient d'etre ajoute
      s'il est de la forme [A -> . B beta, k]
        pour toute regle B -> gamma,
          ajouter [B -> . gamma, i+1] a res s'il n'y est pas
      sinon s'il est de la forme [A -> alpha ., k]
        pour tout [B -> beta . A gamma, k'] de etats[k-1]
          ajouter [B -> beta A. gamma, k'] a res s'il n'y est pas
  tant qu'on ajoute un nouvel item
```

Calcul des états dans la méthode d'Earley - II

- etat initial :

$$q_0 = C(\,\{[S \rightarrow \cdot\alpha, 1] \mid S \rightarrow \alpha \in P\}\,)$$

  autrement dit, l'état initial comprend comme noyau les items correspondant aux règles de l'axiome, et leur fermeture-complétion

- transitions :

$$q_{i+1} = C(\,\{[A \rightarrow \alpha a \cdot \beta, k] \mid [A \rightarrow \alpha \cdot a\beta, k] \in q_i\}\,)$$

  autrement dit, on déplace le $\cdot$ à droite pour les items qui *prédisent* le caractère courant a pour former le noyau du prochain état, puis on fait sa fermeture-complétion

Exemple de fonctionnement de la méthode d'Earley - I

- On prend la grammaire des nombres en notation scientifique :

$S \to NDX$	$N \to C$	$N \to NC$	$D \to .N$	$D \to \varepsilon$
$X \to e + N$	$X \to e - N$	$X \to \varepsilon$		
$C \to 0$	$C \to 1$	$\cdots$	$\cdots$	$C \to 9$

- L'entrée 1 donne (en haut de l'état, items complets)

$q_0 =$

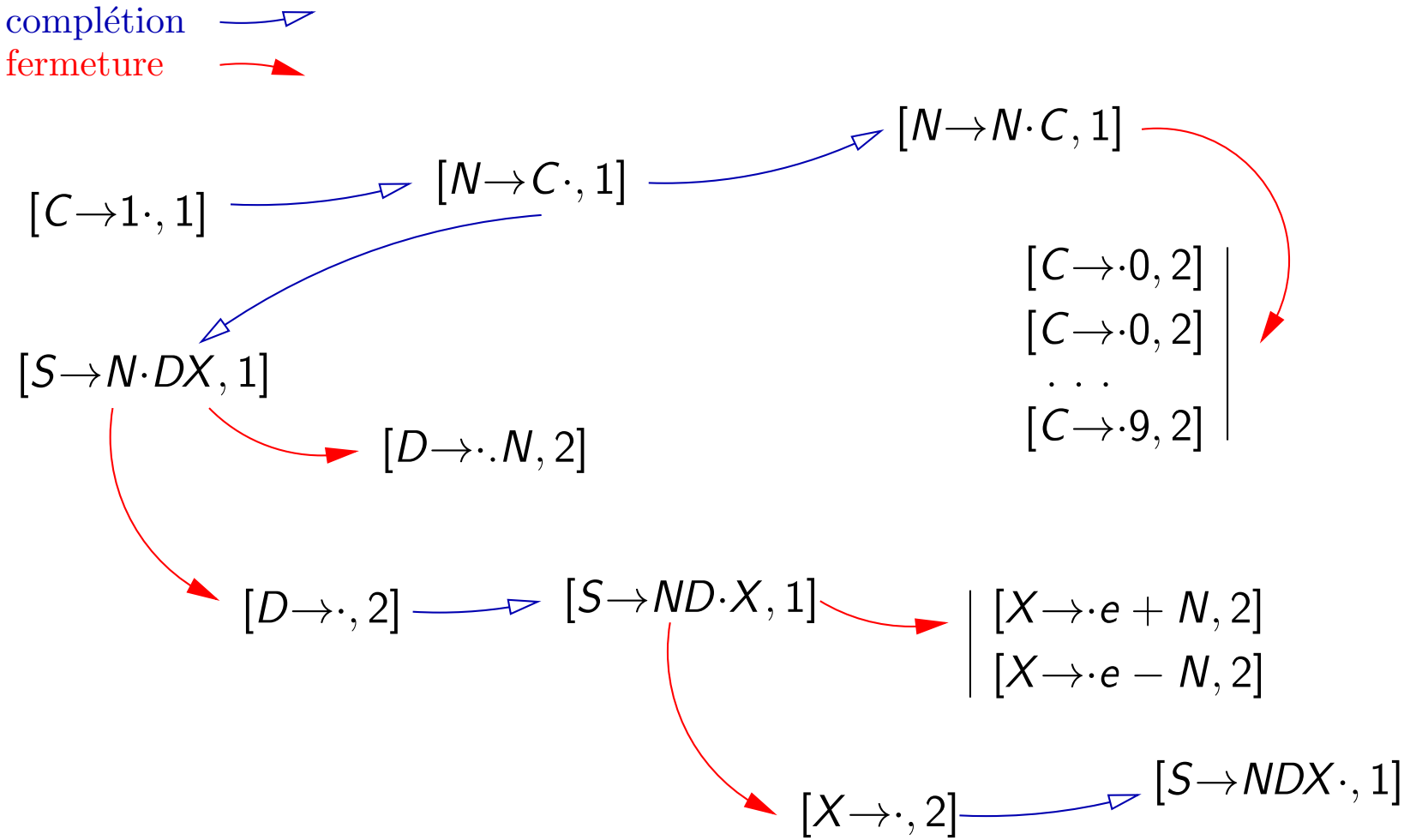
pas d'items complets			
$[S \to \cdot NDX, 1]$	$[N \to \cdot C, 1]$	$[N \to \cdot NC, 1]$	
$[C \to \cdot 0, 1]$	$[C \to \cdot 1, 1]$	$\cdots$	$[C \to \cdot 9, 1]$

$\xrightarrow{1}$

$q_1 =$

$[C \to 1\cdot, 1]$	$[N \to C\cdot, 1]$	$[D \to \cdot, 2]$	$[X \to \cdot, 2]$
$[S \to NDX\cdot, 1]$			
$[S \to N \cdot DX, 1]$	$[N \to N \cdot C, 1]$	$[D \to \cdot .N, 2]$	
$[C \to \cdot 0, 2]$	$[C \to \cdot 1, 2]$	$\cdots$	$[C \to \cdot 9, 2]$
$[S \to ND \cdot X, 1]$	$[X \to \cdot e + N, 2]$	$[X \to \cdot e - N, 2]$	

Détail de la fermeture-complétion de l'état q_1 précédent



Exemple de fonctionnement de la méthode d'Earley - IIa

- L'entrée $12.3e + 4$ donne

$q_0 =$

$[S \to \cdot NDX, 1]$	$[N \to \cdot C, 1]$	$[N \to \cdot NC, 1]$
$[C \to \cdot 0, 1]$	$[C \to \cdot 0, 1]$	$\cdots$ $[C \to \cdot 9, 1]$

$\stackrel{1}{\Longrightarrow}$

$q_1 =$

$[C \to 1 \cdot, 1]$	$[N \to C \cdot, 1]$	$[D \to \cdot, 2]$
$[X \to \cdot, 2]$	$[S \to NDX \cdot, 1]$	
$[S \to N \cdot DX, 1]$	$[N \to N \cdot C, 1]$	$[D \to \cdot N, 2]$
$\cdots$ $[C \to \cdot 2, 2]$	$\cdots$	$[C \to \cdot 9, 2]$
$[S \to ND \cdot X, 1]$	$[X \to \cdot e + N, 2]$	$[X \to \cdot e - N, 2]$

$\stackrel{2}{\Longrightarrow}$

$q_2 =$

$[C \to 2 \cdot, 2]$	$[N \to NC \cdot, 1]$	$[D \to \cdot, 3]$
$[X \to \cdot, 3]$	$[S \to NDX \cdot, 1]$	
$[S \to N \cdot DX, 1]$	$[N \to N \cdot C, 1]$	$[D \to \cdot N, 3]$
$[C \to \cdot 0, 3]$	$\cdots$	$[C \to \cdot 9, 3]$
$[S \to ND \cdot X, 1]$	$[X \to \cdot e + N, 3]$	$[X \to \cdot e - N, 3]$

$\stackrel{\cdot}{\Longrightarrow}$

Exemple de fonctionnement de la méthode d'Earley - IIb

$q_3 =$

$[D \to . \cdot N, 3]$	$[N \to \cdot C, 4]$	$[N \to \cdot NC, 4]$
$\cdots \quad [C \to \cdot 3, 4]$	$\cdots$	$[C \to \cdot 9, 4]$

$\stackrel{3}{\Longrightarrow}$

$q_4 =$

$[C \to 3 \cdot, 4]$	$[N \to C \cdot, 4]$	$[D \to . N \cdot, 3]$
$[X \to \cdot, 5]$	$[S \to NDX \cdot, 1]$	
$[N \to N \cdot C, 4]$	$[S \to ND \cdot X, 1]$	
$[C \to \cdot 0, 5]$	$\cdots$	$[C \to \cdot 9, 5]$
$[X \to \cdot e + N, 5]$	$[X \to \cdot e - N, 5]$	

$\stackrel{e}{\Longrightarrow}$

$q_5 =$

$[X \to e \cdot + N, 5]$	$[X \to e \cdot - N, 5]$

$\stackrel{+}{\Longrightarrow}$

$q_6 =$

$[X \to e + \cdot N, 5]$	$[N \to \cdot C, 7]$	$[N \to \cdot NC, 7]$
$\cdots \quad [C \to \cdot 4, 7]$	$\cdots$	$[C \to \cdot 9, 7]$

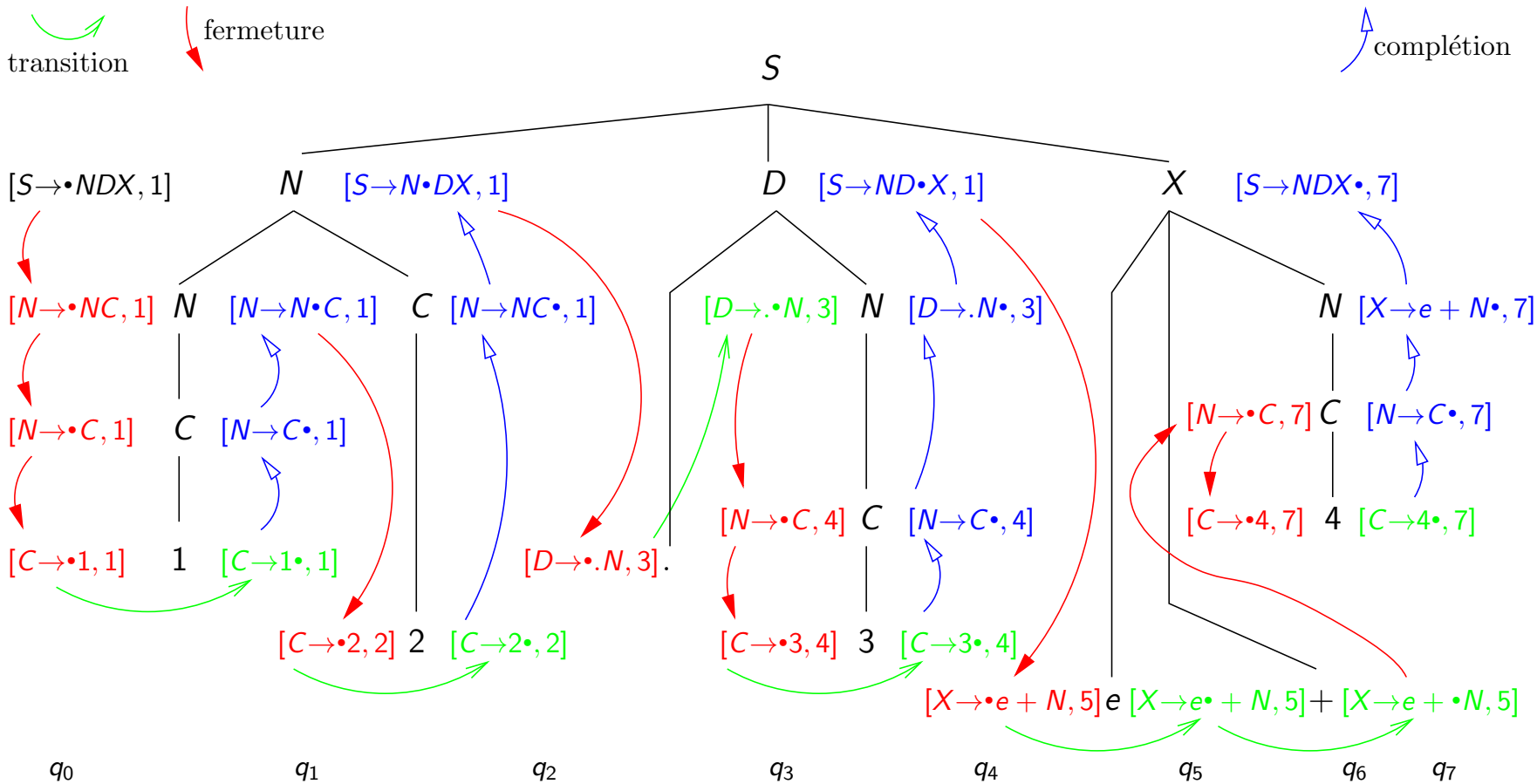
$\stackrel{4}{\Longrightarrow}$

$q_7 =$

$[C \to 4 \cdot, 7]$	$[N \to C \cdot, 7]$	$[X \to e + N \cdot, 5]$	$[S \to NDX \cdot, 1]$
$[N \to N \cdot C, 7]$	$[C \to \cdot 0, 8]$	$\cdots$	$[C \to \cdot 9, 8]$

Méthode d'Earley et arbres de dérivation

- les items des états de la méthodes d'Earley *qui participent à l'obtention de l'item final* $[S \to \sigma \cdot, n]$ représentent un parcours (virtuel) de l'arbre de dérivation ; par exemple, pour $12.3e + 4$



Méthode d'Earley et dérivations gauches

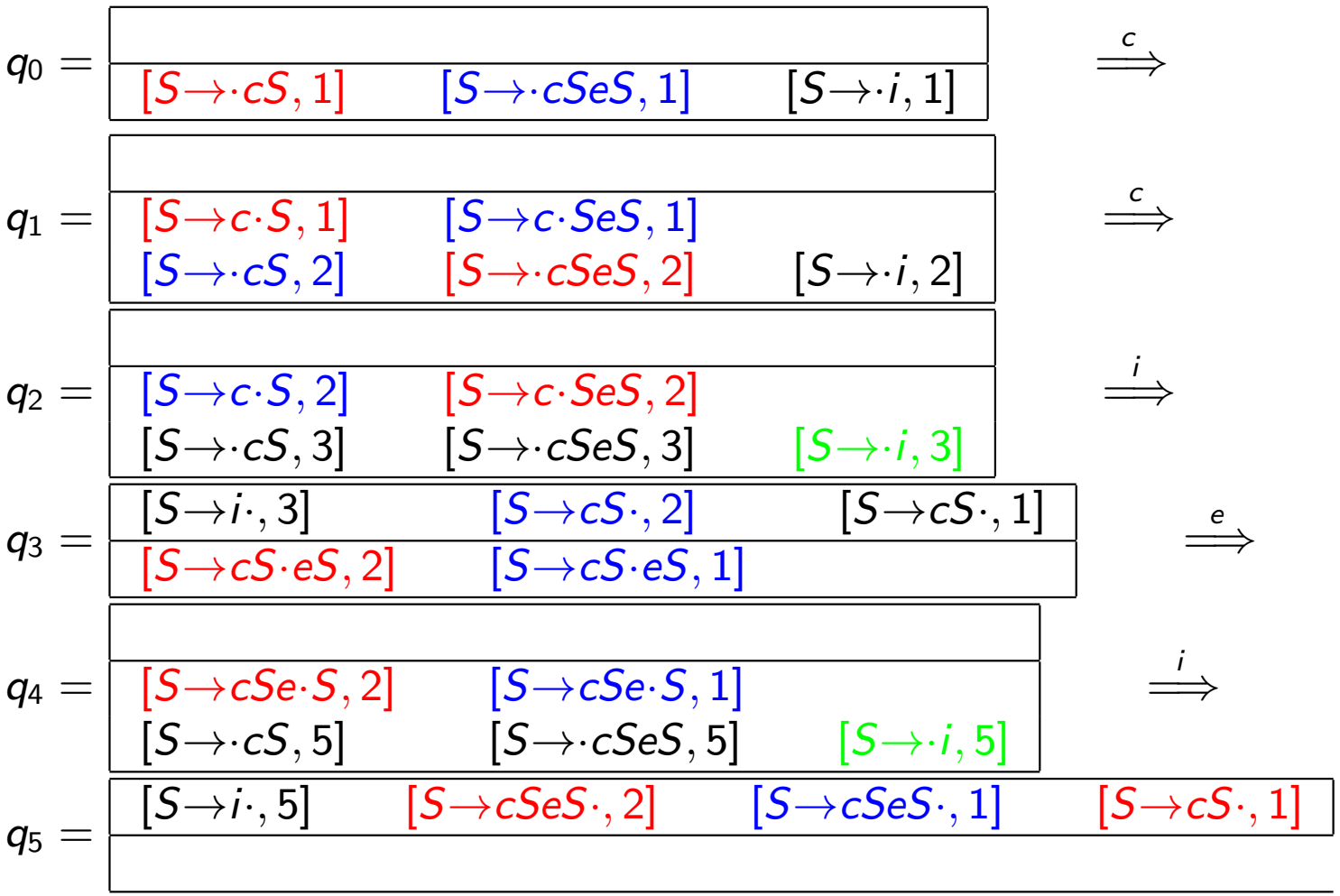
- l'algorithme d'Earley simule aussi une dérivation gauche
  $S \Rightarrow NDX \Rightarrow NCDX \Rightarrow CCDX \Rightarrow 1CDX \Rightarrow 12DX \Rightarrow$
  $12.NX \Rightarrow 12.CX \Rightarrow 12.3X \Rightarrow$
  $12.3e+N \Rightarrow 12.3e+C \Rightarrow 12.3e+4$
- il faut bien voir que dans un état, *tous les pas de dérivation possibles* à cette étape sont envisagés
  - un item comme $[N \rightarrow \cdot NC, 1]$ de l'état q_1 ne prédit pas seulement un nombre à 2 chiffres, mais **tous** les nombres d'au moins 2 chiffres
  - en effet, quand 2 chiffres auront été reconnus (dans l'état q_2), la complétion de l'item $[N \rightarrow NC \cdot 1]$ ne donnera pas uniquement $[S \rightarrow N \cdot DX, 1]$, mais aussi $[N \rightarrow N \cdot C, 1]$ qui permettront de reconnaître soit le point décimal, soit un $3^{\text{ème}}$ chiffre
  - de même, dans un état q_k atteint après avoir reconnu k chiffres, l'algorithme se préserve la possibilité d'en reconnaître un $k+1^{\text{ème}}$

Méthode d'Earley et grammaires ambiguës - I

- on peut donc dire que dans un état q_k, les items de cet état prédisent **tous** les arbres de dérivation *compatibles* avec le préfixe $a_1 \cdots a_k$ du mot à lire
- les arbres potentiels s'éliminent au fur et à mesure de l'avancée dans le mot, puisque les transitions ne retiennent que les items compatibles avec le nouveau caractère lu
- ce qui permet à l'algorithme d'analyser des mots de grammaires ambiguës

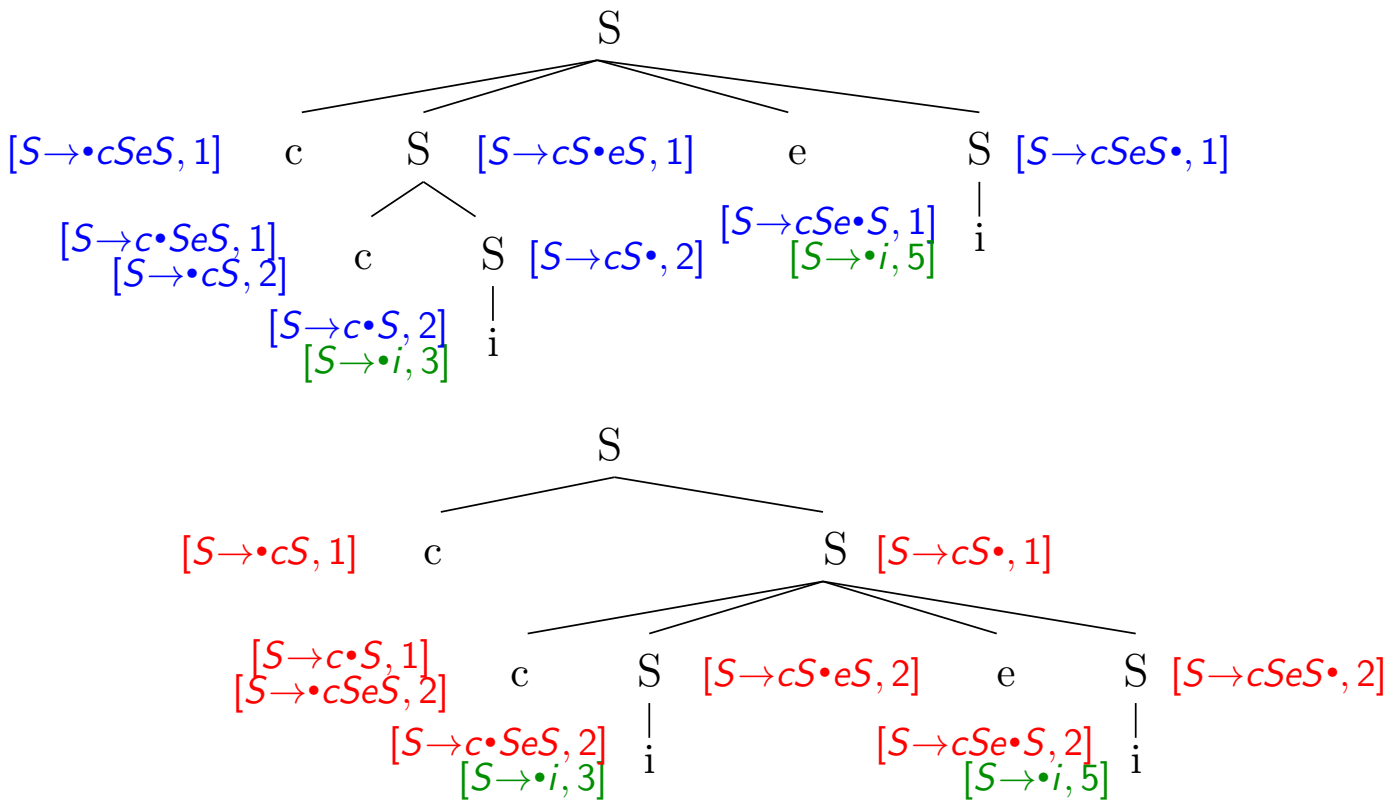
Méthode d'Earley et grammaires ambiguës - II

- soit par exemple, avec la grammaire déjà vue $S \to cSeS \mid cS \mid i$ et l'entrée *cciei*



Méthode d'Earley et grammaires ambiguës - III

- on a donc 2 arbres de dérivation possibles



- pour lever l'ambiguïté, on pourrait supprimer l'item $[S \to cS \cdot eS, 1]$ de l'état q_4 et garder $[S \to cS \cdot eS, 2]$ car $2 > 1$ (on associe le e au c le $+$ proche) mais c'est plus compliqué que cela dans le cas général

Méthode d'Earley et récursivité gauche

- avec la grammaire $\mathcal{G}_{ASa}, P = \{S \to ASa,\ S \to b,\ A \to \varepsilon\}$, qui a des récursivités gauches cachées, et en entrée le mot $ba \cdots a$, on obtient

$$q_0 = \begin{array}{|c|c|c|}\hline [A \to \cdot, 1] & & \\ \hline [S \to \cdot ASa, 1] & [S \to \cdot b, 1] & [S \to A \cdot Sa, 1] \\ \hline \end{array} \quad \stackrel{b}{\Longrightarrow}$$

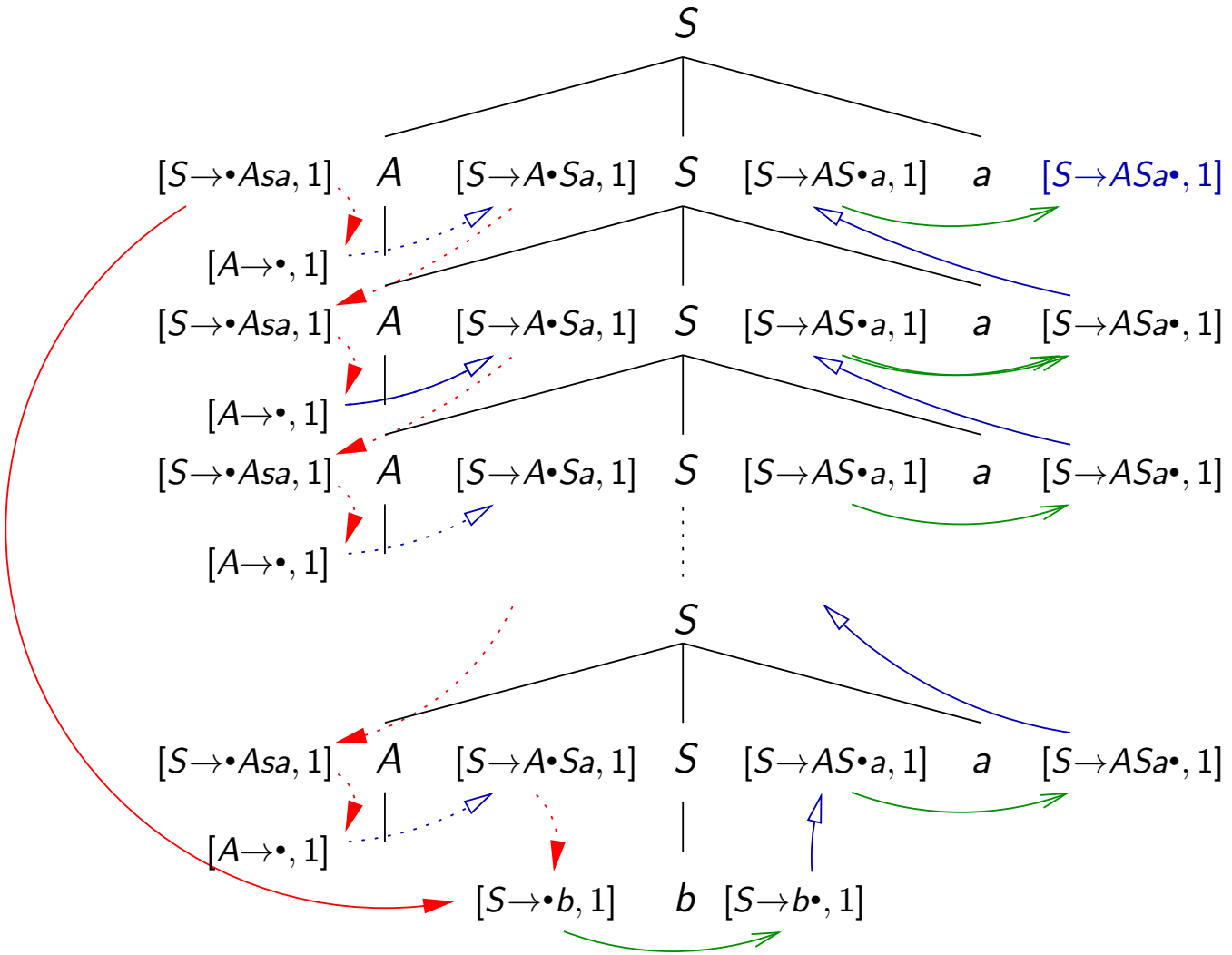
$$q_1 = \begin{array}{|c|}\hline [S \to b \cdot, 1] \\ \hline [S \to AS \cdot a, 1] \\ \hline \end{array} \quad \stackrel{a}{\Longrightarrow} \quad q_2 = \begin{array}{|c|}\hline [S \to ASa \cdot, 1] \\ \hline [S \to AS \cdot a, 1] \\ \hline \end{array} \quad \stackrel{a}{\Longrightarrow} \cdots \stackrel{a}{\Longrightarrow}$$

$$q_n = \begin{array}{|c|}\hline [S \to ASa \cdot, 1] \\ \hline [S \to AS \cdot a, 1] \\ \hline \end{array}$$

- puisqu'on effectue immédiatement la transition sur b, a-t-on un moyen de compter le nombre de A, c'est à dire le nombre de réductions du vide ?

Arbre de dérivation de $\mathcal{G}_{ASa}$

et circulation vituelle des items



Principes de construction de l'arbre de dérivation

à partir des états

- l'état final q_n contient (au moins) un item $[S \rightarrow X_1 X_2 \cdots X_r \cdot, j]$
- il s'agit de produire l'arbre de racine S et de fils $X_1, X_2 \cdots X_r$, et pour chaque X_k qui est un non-terminal, de produire le sous arbre dont il est la racine
- soit $\Pi(q_i, [A \rightarrow X_1 \cdots X_r \cdot, j])$ la fonction produisant le sous-arbre de racine A pour l'item en paramètre qui se trouve dans l'état q_i
  - soit X_r est un terminal, alors il est une feuille, et l'état $q_k, k = i - 1$ contient nécessairement $[A \rightarrow X_1 \cdots X_{r-1} \cdot X_r, j])$
  - soit X_r est un non-terminal, alors l'état q_i contient (au moins) un item $[X_r \rightarrow \alpha \cdot, j']$
    - il faut appeler $\Pi(q_i, [X_r \rightarrow \alpha \cdot, j'])$ pour produire le sous-arbre dont X_r est la racine
    - l'état $q_k, k = j' - 1$ contient $[X_r \rightarrow \cdot \alpha, j']$, et donc nécessairement un item $[A \rightarrow X_1 \cdots X_{r-1} \cdot X_r, j])$
  - on recommence avec $X_{r-1}, X_{r-2} \cdots X_1$ et avec i prenant les valeurs successives de k

Algorithme de construction de l'arbre de dérivation

fonction Π

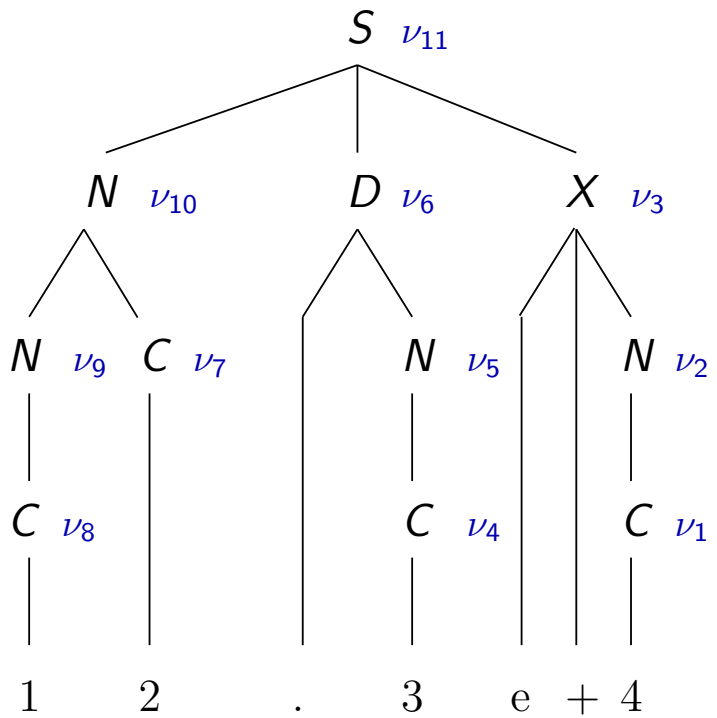
- les fils de l'arbre à produire seront rangés dans un vecteur $\pi_0 \cdots \pi_r$, où r est la longueur de la partie droite qu'on traite
- L'algorithme de Π est le suivant

  $\Pi(q_i, [A \to X_1 \cdots X_r \cdot, j])$
  $k := r$
  tantque $k > 1$ **faire**
  si $X_k \in T$ **alors**
  $\pi_k := X_k$; $i := i - 1$
  sinon // il existe $[X_k \to \beta \cdot, j']$ dans q_i
  $\pi_k := \Pi(q_i, [X_k \to \beta \cdot, j'])$; $i := j' - 1$
  fin tantque
  produire l'arbre de racine $\pi_0 = A$ et de fils $\pi_1 \cdots \pi_r$
  retourner π_0
- on l'appelle par $\pi := \Pi(q_n, [S \to \alpha \cdot, 1])$
- si la phrase est ambiguë, cet algorithme ne donnera qu'un seul des arbres possibles

Construction de l'arbre pour la grammaire des nombres

- avec l'entrée $12.3e+4$: ordre des appels de Π et de construction des nœuds de l'arbre

$\Pi(q_7, [S \to NDX\cdot, 1])$
$\quad \Pi(q_7, [X \to e+N\cdot, 5])$
$\quad\quad \Pi(q_7, [N \to C\cdot, 7])$
$\quad\quad\quad \Pi(q_7, [C \to 4\cdot, 7]) \quad \to\nu_1$
$\quad\quad \to\nu_2$
$\quad \to\nu_3$
$\quad \Pi(q_4, [D \to .N\cdot, 3])$
$\quad\quad \Pi(q_4, [N \to C\cdot, 4])$
$\quad\quad\quad \Pi(q_4, [C \to 3\cdot, 4]) \quad \to\nu_4$
$\quad\quad \to\nu_5$
$\quad \to\nu_6$
$\quad \Pi(q_2, [N \to NC\cdot, 1])$
$\quad\quad \Pi(q_2, [C \to 2\cdot, 2]) \quad \to\nu_7$
$\quad\quad \Pi(q_1, [N \to C\cdot, 1])$
$\quad\quad\quad \Pi(q_1, [C \to 1\cdot, 1]) \quad \to\nu_8$
$\quad\quad \to\nu_9$
$\quad \to\nu_{10}$
$\to\nu_{11}$



Construction de l'arbre pour G_{ASa}