

Correction du Partiel Compilation : Langages et Grammaires

EPITA – Promo 2005

Janvier 2003

Dans cette épreuve, les non-terminaux sont écrits en majuscules, les terminaux en minuscules ou entre guillemets, et ε désigne le mot vide. Lorsque l'on demande le type (de Chomsky) d'une grammaire ou d'un langage on demande **évidemment** le type le plus précis. On considère la hiérarchie de Chomsky « non stricte », c'est-à-dire que la grammaire $S \rightarrow \varepsilon$ est considérée hors-contexte.

Les résultats ne doivent pas être balancés comme « évidents », sous peine de disqualification pour tentative de bluffage. Néanmoins, une argumentation informelle mais convaincante, sera souvent suffisante.

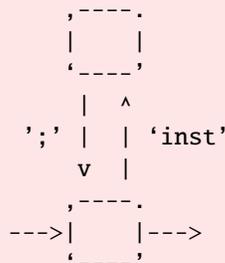
Une copie synthétique (= sans verbiage ou mauvaise traduction d'Appel), bien orthographiée, dont les résultats sont clairement affichés, sera toujours mieux notée qu'une copie qui aura demandé une quelconque forme d'effort de la part du correcteur.

1 Hiérarchie de Chomsky

Pour chacune des grammaires suivantes, préciser (i) son type dans la hiérarchie de Chomsky, (ii) si elle est ambiguë, (iii) le langage qu'elle engendre, (iv) le type du langage dans la hiérarchie, (v) un automate fini déterministe (sous forme d'un diagramme) qui reconnaisse le même langage. Justifier vos réponses.

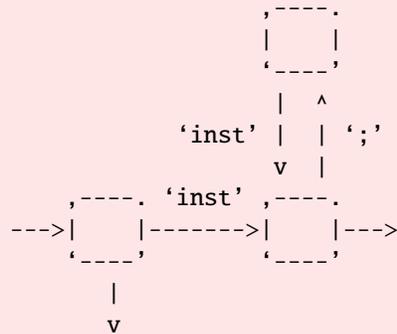
1. $P \rightarrow P \text{ inst } ' ; '$
 $P \rightarrow \varepsilon$

Correction: Linéaire à gauche, donc régulière. Elle est non ambiguë, et engendre une suite de zéro ou plusieurs *inst terminés* par des ;. Ce langage est infini, régulier (puisque engendré par une grammaire régulière) : type 3. Il existe donc un automate, comme par exemple :



2. $P \rightarrow P1$
 $P \rightarrow \varepsilon$
 $P1 \rightarrow P1 ' ; ' \text{ inst}$
 $P1 \rightarrow \text{inst}$

Correction: Linéaire à gauche, donc régulière. Elle est non ambiguë. P1 engendre une liste de une ou plusieurs *inst* séparés par des ;. Donc, cette grammaire engendre une liste de zéro ou plusieurs *inst* séparés par des ;. Ce langage est infini, régulier (puisqu'engendré par une grammaire régulière) : type 3. Il existe donc un automate, comme par exemple :



- 3. P → P1
- P → ε
- P1 → P1 ';' P1
- P1 → inst

Correction: Cette grammaire est très visiblement une version ambiguë de la grammaire précédente. On pourrait dire que dans la grammaire l'opérateur ; est associatif à gauche, ici il est associatif à droite et à gauche, i.e., une phrase comme *inst ; inst ; inst* peut se lire comme (*inst ; inst*) ; *inst* ou *inst ; (inst ; inst)*. Le langage, lui, reste évidemment de type 3, et reconnu par le même automate.

- 4. S → P
- P → p P Q R
- P → p q R
- R Q → Q R
- q Q → q q
- q R → q r
- r R → r r

Correction: Cette grammaire est visiblement monotone, non hors contexte. Bien qu'il ne soit pas simple de le montrer formellement, une "exécution" de cette grammaire à la main montre qu'elle n'est pas ambiguë.

On reconnaît l'exemple de grammaire engendrant $a^n b^n c^n$, i.e., le langage des mots commençant par un certain nombre (non nul) de a, puis d'autant de b, et enfin autant de c. Mais avec p, q et r. Ce langage est bien connu comme l'exemple type des langages sensibles au contexte (et non hors-contexte), comme vu en cours.

Bien entendu, il est impossible de trouver un automate fini (le langage n'est pas régulier), ni même un d'automate à pile (le langage n'est pas hors-contexte).

2 Parsage LL(1)

Soit le langage de la logique (dite propositionnelle) composée de deux symboles t (vrai) et f (faux), de l'opération unaire \neg (non), des opérations binaires \vee (ou) et \wedge (et), et des parenthèses. Ce langage inclut des mots tels que :

$$t \wedge t \quad t \vee f \quad \neg\neg(t \wedge t) \vee (f \wedge f)$$

Noter que l'on parle de tout ce langage, et non seulement du sous-langage des formules « vraies » ; il comprend donc aussi des mots tels que :

$$f \quad \neg t \quad \neg t \wedge f \wedge f \wedge t$$

1. Écrire une grammaire hors contexte naïve de la logique propositionnelle. On cherchera une formulation abstraite, courte et très lisible, au prix de l'ambiguïté.

Correction:

```
S → S ∧ S
S → S ∨ S
S → ¬ S
S → ( S )
S → t
S → f
```

2. Désambigüiser cette grammaire en considérant les règles suivantes :

- (a) \wedge et \vee sont associatives à gauche ;
- (b) \neg est prioritaire sur \wedge ;
- (c) \wedge est prioritaire sur \vee .

c'est-à-dire que $f \vee t \vee f \wedge \neg\neg t \wedge f$ se lit $(f \vee (t \vee ((f \wedge (\neg(\neg t))) \wedge f)))$.

Correction: Comme pour l'arithmétique, on introduit des symboles non-terminaux supplémentaires pour chaque étage de priorité. Par analogie avec l'arithmétique on prendra S (somme) pour l'étage \vee , T (terme) pour l'étage \wedge , et F (facteur) pour \neg , les parenthèses, et les valeurs littérales.

// Une somme est une somme de termes ou un terme.

```
S → S ∨ T
```

```
S → T
```

// Un terme est un produit de facteurs, ou un facteur.

```
T → T ∧ F
```

```
T → F
```

// Un facteur est une suite de négations, un littéral, ou des parenthèses.

```
F → ¬ F
```

```
F → ( S )
```

```
F → t
```

```
F → f
```

On remarque la récursivité à gauche des règles pour obtenir l'associativité à gauche.

3. Expliquer pourquoi cette grammaire ne peut pas être LL(1).

Correction: D'une part elle est récursive à gauche, d'autre part, plusieurs règles sont en concurrence. Par exemple les deux premières règles sont actives pour tout FIRST de T .

4. Transformer cette grammaire en une grammaire susceptible d'être LL(1).

Correction:

(a) Récursion à droite.

```
S → T ∨ S
S → T
T → F ∧ T
T → F
F → ¬ F
F → ( S )
F → t
F → f
```

(b) Factorisation à gauche

```
S → T S'
S' → ∨ T S'
S' →
T → F T'
T' → ∧ F T'
T' →
F → ¬ F
F → ( S )
F → t
F → f
```

5. Quelle critique formuler sur la grammaire obtenue ?

Correction: On a perdu l'associativité gauche des opérateurs.

6. Récrire cette grammaire en s'autorisant les extensions de l'EBNF. Par exemple,

$A \rightarrow a^*$.

Correction:

```
S → T ( ∨ T ) *
T → F ( ∧ F ) *
F → ¬ F
F → ( S )
F → t
F → f
```

7. Écrire en pseudo code un parseur LL(1) avec les bonnes priorités et associativités pour cette grammaire. Il suffira d'écrire une et une seule des routines de parsing, à condition qu'elle soit significative (comprendre que eat, aussi appelée accept, n'est pas demandée).

Correction: S se parse simplement par une routine :

```
routine S ()
  T ()
  tant que lookahead égale ∨ faire
    accepter ∨
    T ()
  fin tant que
fin routine S
et de façon similaire pour T. Enfin, toutes les règles de F commencent par un terminal différent.
```