Correction du Partiel Compilation : Langages et Grammaires

EPITA – Promo 2006 – **Tous documents autorisés** *

Janvier 2004 (1h30)

Le sujet et une partie de sa correction ont été écrits par Akim Demaille. Thomas Claveirole, Valentin David et Clément Vasseur en ont assuré la correction, et ont reporté, à la demande expresse d'Akim Demaille, la partie "best-of".

Une copie synthétique, bien orthographiée, avec un affichage clair des résultats, sera toujours mieux notée qu'une autre demandant une quelconque forme d'effort de la part du correcteur.

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.

Dans cette épreuve, les non-terminaux sont écrits en majuscules, les terminaux en minuscules.

1 Hiérarchie de Chomsky

Pour chacun des langages suivants,
- préciser son type dans la hiérarchie de Chomsky :
  - son rang, e.g., 3
  - son nom, e.g., langage régulier

  On demande **évidemment** le type le plus précis.
- proposer une grammaire
  - en syntaxe BNF
  - qui l'engendre
  - qui ait le même type de Chomsky
  - qui soit non ambiguë.

  Les grammaires longues peuvent être esquissées et se terminer par « ... ».

1. Les logins de la promo 2006 de l'EPITA.

*"Tout document autorisé" signifie que notes de cours, livres, annales, etc. sont explicitement consultables pendant l'épreuve. Le zèle de la part des surveillants n'a pas lieu d'être, mais dans un tel cas contacter le LRDE au 01 53 14 59 22.

Correction: Il s'agit d'un langage fini, rang 4. Une grammaire longue mais finie est :

```
S ::= adam_n
S ::= agrapa_n
S ::= agreda_g
...
```

ou bien même en oubliant la finitude du nombre de comptes, les logins sont composés d'au plus de 8 caractères :

```
S ::= a
S ::= b
...
S ::= z
S ::= aa
S ::= ab
...
S ::= zzzzzzzy
S ::= zzzzzzzz
```

(sans montrer les caractères non alphabétiques).

Barème:

- Le type dans la hiérarchie de Chomsky est correct : 1 point.
- La dénomination du type dans la hiérarchie de Chomsky est correcte : 1 point.
- La grammaire génère un langage valide : 1 point.
- Le grammaire est non-ambiguë et de même type que le langage dans la hiérarchie de Chomsky : 1 point.

Best-of:
- C'est un langage régulier.
- C'est un langage régulier hors contexte de type 2.
- Il ne peut y avoir plus de six lettres dans la partie gauche du login, donc il faut pouvoir compter les lettres. Ce n'est donc pas un langage régulier, mais simplement hors contexte.
- Classification dans la hiérarchie de Chomsky : non classifié (Il y a une dépendance par rapport au nombre maximal de lettres : 6. Ce n'est pas modélisable dans les grammaires classifiées par Mr Chomsky)
- La grammaire des logins EPITA ne peut être engendrée au mieux que par une grammaire hors contexte.
- Voici une grammaire contextuelle pour représenter les logins :
  `S ::= L F F F F F _ L`
  `L ::= a | b | c | d | ...`
  `F _ ::= a _ | b _ | c _ | ...`
  `F ::= - | ϵ | a | b | c | ...`
  `- F ::= - a | - b | - c | ...`
  `ϵ F ::= ϵ ϵ`
- Cette grammaire gère bien les priorités.
- (L'élève fait toute une démonstration pour démontrer la finitude des logins...) Ceci n'est pas une représentation BNF évidemment mais elle exprime bien ce que fait cette grammaire.
- La première règle ne décris pas du tout une grammaire à choix finis, (du style $A \rightarrow \gamma$) mais sinon on ne s'en sort plus.
- Voici la grammaire :
  `L ::= kliout_a | farcy_f | gronly_p | ... | viella_a`
  Ici, ... représente tous les autres logins (qui sont en nombre fini et explicitables à l'aide de `ypcat passwd` et `grep`, mais pas sur copie.)
- Beaucoup d'élèves, qui ont par ailleurs répondus correctement à cette question, estiment que Akim, Pierre Testemale ou encore Fabrice Bardèche font partis de la promo 2006 de l'EPITA.

2. Les nombres entiers non signés écrits en base 4 (i.e., composés de « 0 », « 1 », « 2 », « 3 »).

Correction: Langage infini, mais trivialement régulier —type 3— puisque représenté par l'expression régulière `(0|1)`$^+$. On peut par exemple en donner une grammaire linéaire à droite :
`S ::= 0 S | 1 S | 2 S | 3 S`
`S ::= 0 | 1 | 2 | 3`
Par contre, la grammaire suivante est bien évidemment fausse, puisqu'elle produit le mot vide :
`S ::= 0 S | 1 S | 2 S | 3 S | ϵ`

Best-of:
- On peut créer un automate fini de type 2.
- Et en plus, elle est non ambiguë !
- Cette grammaire contient au moins un terminal et pas de terminaux. Donc elle est régulière.
- Les nombres entiers non-signés écrits en base 4 peuvent s'écrire de la manière suivante : `(0)+(1)+(2)+(3)+`.
- La grammaire parle d'elle même.

3. Les sommes (« + ») de nombres binaires.

Correction: Ça reste régulier, type 3, puisque représenté par l'expression régulière « num(+num)* » où num représente une expression régulière représentant les entiers binaires. On obtient par exemple « (0|1)+(+(0|1)+)* ».
Un grammaire régulière serait :

```
S ::= 0 E | 1 E
    | 0   | 1
E ::= + 0 E | + 1 E
    |   0 E |   1 E
    |   0   |   1
```

Toute réponse de ce style :

```
S ::= S + T | T
T ::= 0 | 1 | 0 T | 1 T
```

donne une grammaire valide et non ambiguë, mais il ne s'agit plus alors d'une grammaire régulière (on perd la linéarité dans la première production.)
Parmi les erreurs fréquemment rencontrées, on trouve aussi les élèves qui oublient qu'un nombre binaire peut être composé de plusieurs chiffres :

```
S ::= S + T | T
T ::= 0 | 1
```

Best-of:
– Cette grammaire est un fragment de la grammaire classique de l'arithmétique, bien connue pour ne pas être ambiguë.
– Type de la grammaire : ambiguë.

4. Les soustractions de nombres binaires avec parenthèses (« (», «) »).

Correction: Il est bien connu qu'on est sorti des langages réguliers, pour tomber dans les langages hors-contexte —type 2. En effet, une grammaire hors-contexte serait :

```
S ::= T - S
S ::= T
T ::= ( S )
T ::= B
B ::= 0 B | 1 B
    | 0 | 1
```

Il n'est pas possible de remonter les parenthèses dans la production S de la sorte :

```
S ::= B - S
S ::= B
S ::= ( S )
B ::= 0 B | 1 B
    | 0 | 1
```

car alors il ne serait plus possible de générer (0)-0.
De même, on ne peux pas regrouper les productions T et B de la sorte :

```
T ::= 0 T | 1 T
    | 0 | 1
    | ( S )
```

car alors on pourrait générer des mots de la forme 0(0) qui ne sont bien évidemment pas dans le langage.

Best-of:

– Soustractions de nombres binaires avec parenthèses :

```
S ::= ( F ) | F
F ::= B Q
Q ::= ( P ) | P
P ::= - B T
T ::= P | ϵ
B ::= ( A ) | A
A ::= 0 | 1
```

– Je suppose que ce n'est pas de type 3 surtout parce que je suis incapable de prouver que c'est le cas.

– Cette grammaire est incluse dans celle de l'arithmétique, qui est ambiguë.

– Alors la... Je dirais que c'est un langage hors contexte, type 2.

– C'est une grammaire hors contexte (context free) (ça fait mieux en anglais non ?)

2 Parsage LL(1)

Lorsque l'on s'attaque à des problèmes sur des structures évoluées (telles que des arbres), on constate souvent que le code mélange à la fois le traitement proprement dit (par exemple changer l'étiquette d'un nœud), et le parcours (appliquer à tous les fils, ou bien encore essayer tel traitement, puis en cas de succès, en appliquer tel autre etc.).

Une façon puissante d'exprimer de tels programmes consiste en la séparation des traitements et des parcours. Ces modes de parcours sont appelés *stratégies*. On s'intéressera alors à un langage dédié aux stratégies, i.e., permettant de composer des parcours. Stratego est un exemple de tel langage, dédié à la transformation de programmes (`http://www.stratego-language.org/twiki/bin/view/Stratego`).

Les stratégies de base comptent la stratégie `0` (échec), la stratégie `1` (succès), ou encore `s` qui dénote une quelconque stratégie atomique de l'utilisateur. À partir de deux stratégies s_1 et s_2 peuvent être construites les stratégies s_1 `+` s_2 (choix non déterministe), s_1 `<+` s_2 (choix gauche), s_1 `.` s_2 (composition séquentielle). Les parenthèses permettent de grouper.

Ce langage inclut des mots tels que :

```
0 <+ 1        (s + s) . 1        s . s <+ s <+ 1
```

1. Écrire une grammaire hors contexte naïve de ce langage de stratégies. On cherchera une formulation abstraite, courte et très lisible, au prix de l'ambiguïté.

Correction:

```
S ::= S .  S
    | S <+ S
    | S +  S
    | ( S )
    | 0 | 1 | s
```

Barème:

– Grammaire correcte : 4
– Étourderie, mais grammaire naïve et simple : 2

Best-of:

```
– S -> E $
  E -> T | ϵ
  T -> C <+ C | (C + C) | T . 1 | T <+ T | C | ϵ
  C -> 0 | 1 | ϵ
– S -> "0" | "1" | "s"
  T -> "(" | ")" | "." | "<+"
  P -> S P1
  P1 -> T S P1 | ϵ
– S => C A C
  A => 0|1|s
  B => <+ | .
  C => ϵ | ( | )
  A => C A B C
  B => C B A C
```

2. Désambiguïser cette grammaire en considérant les règles suivantes :
   (a) Toutes les opérations binaires sont associatives à gauche ;
   (b) `.` est prioritaire sur `<+` ;
   (c) `<+` est prioritaire sur `+`.

c'est-à-dire que

```
0 + s . s <+ s <+ 1
```

se lit comme suit.

```
0 + (((s . s) <+ s) <+ 1)
```

Correction: Comme pour l'arithmétique, on introduit des symboles non-terminaux supplémentaires pour chaque étage de priorité. On prendra `S` (somme) pour l'étage `+`, `C` pour le choix `<+`, `T` (terme) pour l'étage `.`, et `F` (facteur) pour les valeurs littérales et les parenthèses.

```
// Une somme est une somme de choix ou un choix.
S ::= S + C
S ::= C
// Un choix est un choix parmi des termes, ou un terme.
C ::= C <+ T
C ::= T
// Un terme est un produit de facteurs, ou un facteur.
T ::= T . F
T ::= F
// Un facteur est un littéral.
F ::= 0 | 1 | s | ( S )
```

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

Barème:
- Bonne désambiguïsation : 4
- Priorités strictement inversées : 2

Best-of:
- pas le temps.
  récursion à droite + factorisation à gauche.
- (= somme) pour `+`
  T (Terme) pour `<+`
  F (Fuckem) pour `.`
  et E pour stratégie (pourquoi pas ?)
- (a)
```
S -> S + E | E
  E -> E . F | F
  F -> F <+ G | G
  G -> 0 | 1 | s (S)
```
  (b)
```
S -> S <+ E | E
  E -> E + F | F
  F -> F . G | G
  G -> 0 | 1 | s (S)
```
  (c)
```
S -> S + E | E
  E -> E <+ F | F
  F -> F . G | G
  G -> 0 | 1 | s (S)
```
- Bon, alors là, je ne trouve pas le sujet clair. Est-ce que si je n'explique pas comment j'arrive à cette grammaire, je "bluffe" ? La méthode est dans les 4 supports que j'ai emmené (notes de cours, th-lang.pdf, Appel, [? ? ?]), on l'a fait et refait en cours... bon après bien sur j'ai l'air con si j'ai fait une erreur.

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 (celles de S) sont actives pour tout FIRST de C.

Barème:
– Récursion : +2
– Concurence : +2

Best-of:
– Cette grammaire ne peut pas être LL(1) à cause des priorités des opérateurs qui nécessitent d'aller voir plus loin.
– C'est similaire à la grammaire du partiel de l'année dernière, "les deux premières règles sont actives pour tout FIRST de T".
– LL(1) signifie que l'on part systématiquement de l'élément le plus à gauche, et qu'on ne regarde que le token suivant, rien de plus. On ne peut donc pas (avec la grammaire LL(1)) écrire une grammaire comme celle de l'exercice (logique).
– Cette grammaire ne peut pas être LL1 car elle possède la régularité à gauche. elle n'est donc pas régulière.
– Cette grammaire ne peut pas être LL(1) car d'après le cours, toute grammaire LL(1) est non ambiguë, comme celle-ci est ambiguë, elle n'est pas LL(1).
– Elle est récursive à droite.
– LL ne sait pas gérer les récursions gauche.
⇒ on dérécursionne (ré-récursionne ?) à droite.
– Cette grammaire n'est pas LL car pour des grammaires tels que U on a trop de productions possibles.

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

Correction:

(a) Récursion à droite.

```
S ::= C + S  | C
C ::= T <+ C | T
T ::= F . T  | F
F ::= 0 | 1 | s | ( S )
```

(b) Factorisation à gauche En utilisant ’’ pour désigner le mot vide.

```
S  ::= C S'
S' ::= + S  | ''
C  ::= T C'
C' ::= <+ C | ''
T  ::= F T'
T' ::= . T  | ''
F  ::= 0 | 1 | s | ( S )
```

(c) Simplification des récursions mutuelles.

```
S  ::= C S'
S' ::= + C S'  | ''
C  ::= T C'
C' ::= <+ T C' | ''
T  ::= F T'
T' ::= . F T'  | ''
F  ::= 0 | 1 | s | ( S )
```

Barème:
– Récursion et factorisation : 4
– Récursion seule (mais concorde avec la réponse 3) : 2

Best-of:
– On va donc mettre la récursion à droite :
```
S -> E $
E -> I "+" E | I
I -> T "<+" I | T
T -> F "." T | F
F -> 0 | 1 | s | (E)
```
(Note : un doute subsiste à savoir si "." doit appartenir à l'etage des parenthèses, ou bien rester à son étage).
Ainsi il aurait peut etre suffit d'écrire :
```
S -> E $
E -> T + E | T
T-> F <+ T F
F ->  0 | 1 | s | (E) | E.F
```
(ce qui me gène ici, c'est le fait que "." soit binaire et non unaire).
– On force donc l'utilisation des parenthèses :
```
S -> S + (E) | E
E -> E <+ (F) | F
F -> F . (G) | G
G -> (S) | L
L -> s | 0 | 1
```

5. Quelle critique formuler sur la grammaire obtenue ?

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

Barème:
– Bonne réponse- : 4 points.
– Mauvaise réponse- : 0 point.

Best-of:
– C'est illisible et totalement contre-intuitif.
– La grammaire obtenue engendre des langages trop parenthésés, ce qui est lourd et contraignant. On se croirait en LISP.

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

```
A ::= a*.
```

Correction:
```
S ::= C (+ C)*
C ::= T (<+ T)*
T ::= F (. F)*
F ::= 0 | 1 | s | ( S )
```

Barème:
– Grammaire correcte- : 4 points.
– Mauvaise priorité- : 3 points.
– Pour chaque faute dans la grammaire- : -1 point.

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 parsage, à condition qu'elle soit significative (comprendre que `eat`, aussi appelée `accept`, n'est pas demandée). On prendra soin de ne pas cacher la gestion des erreurs.

Correction: `S` se parse simplement par une routine :

```
routine parse-S ()
  parse-C ()
  tant que lookahead égale '+' faire
    accepter '+'
    parse-C ()
  fin tant que
fin routine parse-S
```

et de façon similaire pour `C` et `T`. Enfin, toutes les règles de `F` commencent par un terminal différent donc aucun problème pour LL.

Barème:
- Code correct : 4 points.
- Erreur d'étourderie (mauvais opérateur) : -1 point.