# Correction du Partiel THL
## Théorie des Langages

EPITA – Promo 2009 – **Documents autorisés**

Novembre 2006 (1h30)

Le sujet et une partie de sa correction ont été écrits par Akim Demaille.

Écrire court, juste, et bien. Une argumentation informelle mais convaincante, sera souvent suffisante.

## 1 Incontournables

Les questions suivantes sont fondamentales. Une pénalité sur la note finale sera appliquée pour les erreurs.

**Correction:** Chaque erreur aux trois questions suivantes retire 1/6 de la note finale. Avoir tout faux divise la note par 2.

1. Un sous-langage d'un langage rationnel (i.e., un sous-ensemble) est rationnel. vrai/faux ?

   **Correction:** N'importe quel langage $L$ sur un alphabet $\Sigma$ vérifie $L \subset \Sigma^\star$, donc bien sûr que c'est faux.
   70% ont juste.

2. Le langage engendré par $S \to 0X \quad S \to \varepsilon \quad X \to S1$ est rationnel. vrai/faux ?

   **Correction:** Ce langage est $0^n1^n$ bien connu pour être hors-contexte, et non rationnel.
   69% ont juste.

3. Un automate non-déterministe est un automate qui n'est pas déterministe. vrai/faux ?

   **Correction:** Non, il peut être déterministe, d'où l'écriture en un seul mot « non-déterministe » par opposition à « non déterministe ».
   65% ont juste.

## 2 Culture Générale

1. Combien existe-t-il de sous-ensembles d'un ensemble de taille $n$ ?

   **Correction:** $2^n$. Pensez par exemple au codage en un vecteur de $n$ bits.
   **Seuls 11% de la promo a juste !**

2. Combien existe-t-il de n-uplets (c'est-à-dire une suite de $n$ éléments) d'un ensemble de taille $m$ ?

   **Correction:** $m^n$, on a $n$ fois le choix parmi $m$. La phrase est volontairement moins simple que celle vue en cours : « combien existe-t-il de mots de $n$ lettres prises dans un alphabet de taille $m$ ? ».
   **Seuls 5% de la promo a juste !**

3. Que vaut $2^{16}$ ?

   **Correction:** 65536. Pour ceux qui ne le savent pas (c'est bizarre pour un informaticien), la réponse était donnée dans la section 4 : $2^{16} = 2^{(4*4)} = (2^4)^4 = 65536$.
   79% de réussite.

4. Comment numéroter (i.e., étiqueter par un élément unique de $\mathbb{N}$) les réels ($\mathbb{R}$) ?

   **Correction:** Pas possible, comme le montre la diagonale de Cantor qui montre que $\mathbb{R}$ n'est pas dénombrable.
   27% de réussite. Affligeant. Certains proposent d'utiliser des tables de hachage...

5. Qui a inventé les algorithmes LR(k) ?

   **Correction:** Donald Knuth.
   28% de réussite.

   **Best-of:** "Knump", "Knoot", "Knouf", "Knout",...
   Certains petits rigolos ont cité Turing, Djisktra, Didier Verna, ou encore moi-même.

# 3 Hiérarchie de Chomsky

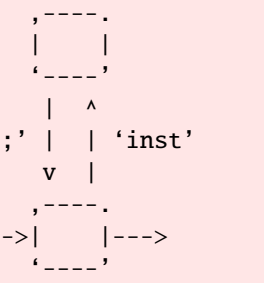
Pour chacune des grammaires suivantes, préciser (i) son type dans la hiérarchie de Chomsky, (ii) si elle est ambiguë, (iii) le langage qu'elle engendre, (iv) le type du langage dans la hiérarchie, (v) s'il existe un automate fini déterministe qui reconnaisse ce langage.

Justifier vos réponses.

**Correction:** D'une part cet exercice est très simple, et d'autre part, les corrections étaient disponibles. Certains ont d'ailleurs été jusqu'à recopier la correction des années précédentes.
Au départ je voulais faire une épreuve sans document, ce qui légitimait le choix de ces exercices, mais j'ai changé d'avis ensuite en oubliant de reprendre cet énoncé.

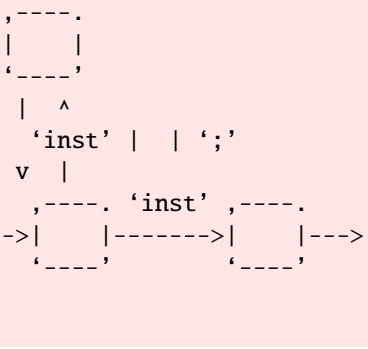
1. `P` $\rightarrow$ `P inst ';'`
   `P` $\rightarrow \varepsilon$

   **Correction:** 61% de réussite.
   Linéaire à gauche, donc régulière. Elle est non ambigüe, et engendre une suite de zéro ou plusieurs `inst` *terminés* par des `;`. Ce langage est infini, régulier (puisqu'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 ';' inst`
   `P1` $\rightarrow$ `inst`

**Correction:** 53% de réussite
Linéaire à gauche, donc régulière. Elle est non ambigüe. `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` → $\varepsilon$
   `P1` → `P1 ';' P1`
   `P1` → `inst`

**Correction:** 44% de réussite.
Cette grammaire est très visiblement une version ambigüe 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 Parsage LL

On rappelle que 1/2/3 = 1/6 et non pas 1.5, et $2^{4^4}$ n'est pas égal à $(2^4)^4 = 65536$, mais vaut :

$2^{(4^4)} = 115792089237316195423570985008687907853269984665640564039457584007913129639936$

1. Écrire la grammaire naïve de l'arithmétique avec les terminaux « n » (les entiers), « / » la division, « ^ » la puissance, « ( » et « ) » les parenthèses.

**Correction:** 67% de réussite.

```
E →  E / E
E →  E ^ E
E →  ( E )
E →  n
```

**Best-of:** `{ /, ^, [0-9], '(', ')' }*`. Ça c'est pas naïf, c'est stupide (ce n'est pas une grammaire), et faux (le langage dénoté contient des horreurs telles que `)(`, `/0` etc.

2. En plusieurs étapes, en faire une grammaire adaptée au parsage LL(1).

**Correction:** 27% de réussite, c'est très peu pour un exercice qui est un grand classique de l'épreuve THL.

(a) Priorité et associativité. Attention à l'associativité **droite** de puissance.

```
T →  T / F | F
F →  N ^ F | N
N →  ( T ) | n
```

(b) Élimination des récursions à droite.

```
T →  F / T | F
F →  N ^ F | N
N →  ( T ) | n
```

(c) Factorisation à gauche

```
T  →  F T'
T' →  / T | ε
F  →  N F'
F' →  ^ F | ε
N  →  ( T ) | n
```

(d) Simplification de $E', P'$.

```
T  →  F T'
T' →  / F T' | ε
F  →  N F'
F' →  ^ N F' | ε
N  →  ( T ) | n
```

3. Quel opérateur n'est pas lu correctement au terme de ces transformations ?

   **Correction:** La division a perdu son associativité gauche. Seuls 14% ont su le dire...

4. En s'autorisant l'utilisation de l'étoile de Kleene ($^*$) dans les parties droites de règles, modifier la grammaire précédente pour « réparer » l'opérateur abîmé.

   **Correction:** Réussi à 21%.

```
T →  F (/ F)*
F →  N ^ F | N
N →  n | (T)
```

5. Considérant les déclarations suivantes :

```
/* The lookahead. */
token_t la;
/* Its value when it's an ``n''. */
int value;
/* Check that the lookahead is equal to ``t'', and then advance.
   Otherwise, report an error, and throw tokens until ``t'' (or
   end of file) is found. */
void eat (token_t t);
```

écrire les routines d'analyse des non terminaux. On prendra garde à :
- retourner la valeur de l'expression,
- reporter les erreurs à l'utilisateur,
- soumettre du code *lisible*,

– ne pas se perdre dans les détails, rester abstrait
(e.g., on peut utiliser `afficher (la)` sans en fournir d'implémentation).

**Correction:** Seuls 13% ont su répondre quelque chose comme ce qui suit.

```
int
parse_T ()
{
  int res = parse_F();
  /* While implémente naturellement les associativités
     droites. */
  while (la == '/')
  {
    eat (/);
    res /= parse_F (); /* Certes, le cas 0. */
  }
  return res;
}

int
parse_F ()
{
  int res = parse_N();
  /* La récursion implémente naturellement les
     associativités gauches. */
  if (la == '^')
    {
       eat ('^');
       res ^= parse_F ()
    }
  return res;
}

int
parse_N ()
{
  switch (la)
  {
    case 'n':
      eat ('n');
      return value;
    case '(':
      eat ('(');
      int res = parse_T(); /* C99 rocks, ANSI-C sucks. */
      eat (')');
      return res;
    default:
      error (location, "unexpected %s, expected 'n' or '('",
             token_to_string (la));
      /* Reprise sur erreur: attendre un token dans
         FOLLOW (N). */
     while (la ∉ { '^', '/', ')', '$' })
       eat (la);
     /* Ne pas oublier de donner une valeur. */
     return 42 + 51 - 69;
  }
}
```

## 5 Automate LR(1)

Soit le fichier Yacc/Bison suivant :

```
%%
l: l l | 'n';  /* These "l" are "L", not "1", of course. */
```

1. Montrer que cette grammaire est ambiguë en exhibant deux arbres de dérivation du plus petit mot ambigu.

   **Correction:** 49% voient que `n n n` = `(n n) n` ou `n (n n)`.

2. Dessiner l'automate LR(1) en montrant bien les lookaheads.

**Correction:** 20% savent calculer cet automate très simple.
Voici ce qu'en dit Bison.

```
State 4 conflicts: 1 shift/reduce

Grammar

    0 $accept: l $end
    1 l: l l
    2  | 'n'


Terminals, with rules where they appear

$end (0) 0
'l' (108)
'n' (110) 2
error (256)

Nonterminals, with rules where they appear

$accept (5)
    on left: 0
l (6)
    on left: 1 2, on right: 0 1

state 0

    0 $accept: . l $end
    1 l: . l l
    2  | . 'n'

    'n'  shift, and go to state 1

    l  go to state 2


state 1

    2 l: 'n' .

    $default  reduce using rule 2 (l)


state 2

    0 $accept: l . $end
    1 l: . l l
    1  | l . l
    2  | . 'n'

    $end  shift, and go to state 3
    'n'   shift, and go to state 1

    l  go to state 4
```



```
state 3
```

3. Indiquer sur cette figure le conflit dû à l'ambiguïté.

   **Correction:** 12% de réussite.

4. Entre règles récursives à droite et à gauche, lesquelles préfèrent les parsers LR, et pourquoi.

   **Correction:** 20% de réussite.
   La récursion gauche économise la pile, puisqu'elle permet de réduire plus rapidement.

   **Best-of:**
   – LR préfère les règles récursives à droite parce que ça veut dire Like Right
   – LR ... parce qu'il est bottom-up
   – LR ... parce qu'il est Left to Right

5. Comment utiliser les directives de Yacc/Bison pour choisir cette « associativité » ?

**Correction:** 27% de réussite ont su faire ce qu'on a fait ensemble en classe. Le conflit est au moment où $\vdash ll \quad n \dashv$ : faut-il réduire vers $\vdash l \quad n \dashv$ (associativité gauche), ou décaler vers $\vdash lln \quad \dashv$ ? Le sujet demande l'associativité gauche, i.e., la règle 1, qui ne porte pas d'étiquette, doit l'emporter sur le token $n$. Deux solutions :

(a) Deux priorités différentes.

```
%nonassoc 'n'
%nonassoc 'l'
%%
l: l l %prec 'l'
 | 'n';
```

Voilà ce qu'en pense Bison.

```
state 4

    1 l: . l l  [$end, 'n']
    1  | l . l  [$end, 'n']
    1  | l l .  [$end, 'n']
    2  | . 'n'

    $default  reduce using rule 1 (l)

    l  go to state 4

    Conflict between rule 1 and token 'n' resolved
              as reduce ('n' < 'l').
```

(b) Un priorité associative à gauche.

```
%left 'n'
%%
l: l l %prec 'n'
 | 'n';
```

Voilà ce qu'en pense Bison.

```
state 4

    1 l: . l l  [$end, 'n']
    1  | l . l  [$end, 'n']
    1  | l l .  [$end, 'n']
    2  | . 'n'

    $default  reduce using rule 1 (l)

    l  go to state 4

    Conflict between rule 1 and token 'n' resolved
              as reduce (%left 'n').
```