# Correction du Partiel THL
## Théorie des Langages

EPITA – Promo 2008 – **Tous documents autorisés**

Novembre 2005 (1h30)

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

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. Une argumentation informelle mais convaincante, sera souvent suffisante.

## 1 Incontournables

Il n'est pas admissible d'échouer sur une des questions suivantes : **chacune représente 25% de la note finale**. En d'autres termes, deux erreurs dans cette partie réduisent la note de la suite de moitié.

1. Le langage engendré par $S \to aX \quad S \to b \quad X \to Sc$ est rationnel. vrai/faux ?

   **Correction:** Ce langage est $a^n b^n$ bien connu pour être hors-contexte, et non rationnel.

2. 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.

3. LR(k) est plus puissant que LL(k). vrai/faux ?

   **Correction:** Oui, c'est vrai.

4. Si un analyseur GLR a des conflits « reduce/reduce » alors il est incorrect. vrai/faux ?

   **Correction:** Non, tout l'intérêt (et le danger) de GLR c'est que l'on peut tolérer les conflits : à l'exécution toutes les pistes seront suivies.

## 2 Langages rationnels

On pourra admettre le résultat principal de cette section dans la suite de l'épreuve.

1. Comment montre-t-on que l'intersection de deux langages rationnels est rationnelle ?

   **Correction:** Par exemple en construisant le produit synchronisé de deux automates finis déterministes reconnaissant ces langages. Cf. le cours avec le cas de l'intersection entre « les mots sur $a, b$ avec un nombre impair de $a$ » et « les mots sur $a, b$ avec un nombre impair de $b$ » .

2. Comment montre-t-on que si un langage est rationnel, alors son « renversé » (tous les mots écrits à l'envers) l'est aussi ?

   **Correction:** Il suffit d'inverser toutes les flèches (y compris initiale et finales) d'un automate reconnaissant le premier.

3. En déduire que $\{a^n b^m | m \leq n\}$ n'est pas rationnel.

> **Correction:** S'il l'était, alors l'intersection avec son renversé le serait aussi. On montrerait alors que $a^n b^n$ est rationnel...

# 3 Si alors sinon

## 3.1 Grammaires

Considérons l'extrait de grammaire suivant :

```
<stm>   : :=   if <exp> then <stm> else <stm>
           |   if <exp> then <stm>
           |   ...
```

1. Quelle est sa catégorie de Chomsky ? Justifier.

> **Correction:** Grammaire hors-contexte : un seul symbole à gauche, et c'est un non-terminal. Certaines règles ont plusieurs non-terminaux à gauche, elle n'est donc pas rationnelle.

2. En considérant que `<exp>` et `...` sont des terminaux, quelle est la catégorie de Chomsky du langage engendré ? Justifier.

> **Correction:** Supposons ce langage rationnel. Alors son intersection avec le langage rationnel $\texttt{then}^*.\texttt{else}^*$ doit l'être aussi. Or il est alors facile de voir qu'il ne peut pas y avoir plus de `else` que de `then`. D'une façon certes un peu approximative, on voir que le résultat serait
>
> $$\texttt{then}^n.\texttt{else}^m \quad | \quad n \leq m$$
>
> La section 2 montre que ce langage n'est pas rationnel, ce qui permet de conclure que le langage engendré est hors-contexte et non rationnel.
> Noter que l'on ne peut pas juste se contenter de dire que « ce langage contient $\texttt{then}^n.\texttt{else}^m$ qui n'est pas rationnel, donc il n'est pas rationnel. « Contenir » est bien trop flou, comme le montre la question 1.2.

3. Cette grammaire est-elle ambiguë ? Justifier.

> **Correction:** C'est le « dangling-else » bien connu.

4. Produire deux arbres de dérivation différents d'une phrase la plus courte possible.
5. Pourquoi la syntaxe du shell comprend-elle un terminal `fi` ?

> **Correction:** Pour désambiguïser.

## 3.2 Désambiguïsation

À partir de ce point on ne retiendra que l'interprétation la plus répandue dans les langages de programmation : on attache un `else` au `if` le plus proche.

1. On rencontre fréquemment des macros comparables aux suivantes.

```
#define LOG(Msg)                                        \
  if (using_syslog)                                     \
    vsyslog (LOG_INFO, "%s\n", Msg);                    \
  else                                                  \
    fprintf (stderr, "%s: %s\n", program_name, Msg)

#define TRACE_INT(Int)                                  \
  do {                                                  \
    if (trace)                                          \
      fprintf (stderr, "Trace: %s = %d\n", #Int, Int);  \
  } while (0)
```

Comment expliquez-vous cette différence de style ?

2. En introduisant <stm-if-then> et <stm-if-then-else>, proposer une grammaire non ambiguë équivalente à celle-ci :

<stm-if> : := if <exp> then <stm-if> else <stm-if>
| if <exp> then <stm-if>

**Correction:** Commençons par remarquer que cette question est étrangement posée : dans un soucis de simplification, on en tombe presque dans le ridicule puisque cette grammaire, techniquement, ne produit aucun langage. La grammaire suivante, non ambiguë, est donc une réponse valide :

<stm-if> : :=

Il faudrait vraiment qu'il y ait un terminal quelque part... Le sujet cherchait à retarder son introduction à la question suivante. Si l'on s'attache donc à travailler dans l'esprit de la question, introduisons d'abord les deux symboles :

<stm-if> : := <stm-if-then>
| <stm-if-then-else>
<stm-if-then> : := if <exp> then <stm-if>
<stm-if-then-else> : := if <exp> then <stm-if> else <stm-if>

Puis déroulons les <stm-if> et rayons les cas invalides : aucun <stm-if-then> ne doit être dans un <stm-if-them-else>.

<stm-if> : := <stm-if-then>
| <stm-if-then-else>
<stm-if-then> : := if <exp> then <stm-if-then>
| if <exp> then <stm-if-then-else>
<stm-if-then-else> : := ~~if <exp> then <stm-if-then> else <stm-if-then>~~
| ~~if <exp> then <stm-if-then> else <stm-if-then-else>~~
| if <exp> then <stm-if-then-else> else <stm-if-then>
| if <exp> then <stm-if-then-else> else <stm-if-then-else>

On arrive donc à la solution suivante :

<stm-if> : := <stm-if-then>
| <stm-if-then-else>
<stm-if-then> : := if <exp> then <stm-if>
<stm-if-then-else> : := if <exp> then <stm-if-then-else> else <stm-if>

ou

<stm-if> : := if <exp> then <stm-if>
| <stm-if-then-else>
<stm-if-then-else> : := if <exp> then <stm-if-then-else> else <stm-if>

Cependant, comme l'a remarqué Jean-Philippe Garcia Ballester, cette solution est incorrecte, car si elle interdit bien à un <stm-if-thn> de voler le `else` de son père, ça n'interdit le même problème d'apparaître avec son *grand*-père :

```
if <exp>
then if <exp>
     then <stm-if>
     else if <exp>
          then <stm-if>
          else <stm-if>
```

```
if <exp>
then if <exp>
     then <stm-if>
     else if <exp>
          then <stm-if>
else <stm-if>
```

Il faut aller plus loin en se permettant de distinguer les **chaînes** de `if-then-else`.

<stm-if> : := if <exp> then <stm-if>
| if <exp> then <stm-if-then-else> else <stm-if>
<stm-if-then-else> : := if <exp> then <stm-if-then-else> else <stm-if-then-else>

3. Dans la réalité la grammaire originelle est plutôt la suivante. Quel problème allons-nous rencontrer pour adapter le travail précédent ?

```
<stm>   ::=  if <exp> then <stm> else <stm>
         |   if <exp> then <stm>
         |   while <exp> do <stm>
         |   ...
```

**Correction:** Il va falloir faire très attention à ne pas réintroduire `<stm-if>` dans `<stm>` au risque de laisser passer les `<stm-if-then>` en partie `then` d'un `<stm-if-then-else>`.

```
                <stm>   ::=  if <exp> then <stm>
                         |   if <exp> then <stm-if-then-else> else <stm>
                         |   <stm-all-but-if>

 <stm-if-then-else>   ::=  if <exp> then <stm-if-then-else> else <stm-if-then-else>
                         |   <all-but-if>

     <stm-all-but-if>   ::=  while <exp> do <stm>
                         |   ...
```

C'est vraiment très laid. Et aucune des <stm-all-but-if> ne doit de plus « s'amuser » à utiliser un `else` optionnel...

### 3.3 Analyse LL

1. La grammaire suivante est-elle LL(k) ? Justifier.

```
<stm>   ::=  if <exp> then <stm> else <stm>
         |   if <exp> then <stm>
         |   while <exp> do <stm>
         |   ...
```

**Correction:** Non, les deux premières règles ont le même *first* de partie droite.

2. Proposer en BNF une grammaire LL(1) engendrant le même langage.

**Correction:** Il faut factoriser à gauche.

```
          <stm>   ::=  if <exp> then <stm> <stm-else-opt>
                   |   while <exp> do <stm>
                   |   ...

 <stm-else-opt>   ::=  else <stm-if>
 <stm-else-opt>    |   ε
```

3. Proposer en EBNF une grammaire LL(1) engendrant le même langage.

**Correction:**

```
                <stm>   ::=  if <exp> then <stm> ( else <stm-if> ) ?
                         |   while <exp> do <stm>
                         |   ...
```

4. Déduire de la grammaire de l'item 3 une implémentation de `parse_stm`. Toutes les routines d'analyse retournent un arbre de type `ast_t *` et utilisent `token_if` ... pour les terminaux. Les routines telles que `ast_t *ast_if (ast_t *c, ast_t* s1, ast_t* s2)` construisent ces arbres.

**Correction:**

```
ast_t *
parse_stm (void)
{
  ast_t *res;
  switch (lookahead)
  {
    case token_if:
      {
        ast_t *c = 0, *s1 = 0, *s2 = 0;
        eat (token_if);
        cond = parse_exp ();
        eat (token_then);
        s1 = parse_stm ();
        if (lookahead == token_else)
        {
          eat (token_else);
          s2 = parse_stm ();
        }
        return ast_if (c, s1, s2);

    case token_while:
      ...
  }
}
```

## 3.4 Analyse LR

1. Proposer une implémentation LALR(1) de la grammaire suivante qui tire parti des fonctionnalités de Yacc.

   <stm> ::= if <exp> then <stm> else <stm>
   | if <exp> then <stm>
   | while <exp> do <stm>
   | ...

   **Correction:**

   ```
   %nonassoc "then"
   %nonassoc "else"
   %%
   stm:
     "if" exp "then" stm "else" stm
   | "if" exp "then" stm
   | "while" exp "do" stm
   | ...
   ;
   ```

   On pourrait aussi écrire (ce qui a ma préférence pour sa lisibilité) :

   ```
   %right "then" "else"
   ```

2. Pourquoi préférer cette solution plutôt que celle retenue dans la section 3.2 ou dans la section 3.3 ?