Correction du Partiel THL
Théorie des Langages

EPITA – Promo 2010 – **Documents autorisés**

Janvier 2008(1h30)

> **Correction:** Le sujet et sa correction ont été écrits par Akim Demaille. Le best of est tiré des copies des étudiants. Les erreurs, en particulier d'orthographe, sont les leurs !

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

1 Incontournables

Les questions suivantes sont fondamentales. Une pénalité sur la note finale sera appliquée pour les erreurs. Répondez sur les feuilles de QCM qui vous sont remises.

> **Correction:** Chaque erreur aux trois questions suivantes retire 1/6 de la note finale. Avoir tout faux divise la note par 2. D'assez nombreux étudiants n'ont pas été capables de mettre les réponses sur le formulaire, mais l'ont fait dans leur copie. Dans ce cas, ils ont eu faux aux trois questions.

1. Tout automate non-déterministe n'est pas déterministe. a. vrai/b. 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 en promo 2009, 74% en 2010.

2. Le langage engendré par $A \to \rangle B \quad A \to \times \quad B \to A\langle$ est rationnel. a. vrai/b. faux ?

   > **Correction:** Ce langage est $\rangle^n \times \langle^n$ bien connu pour être hors-contexte, et non rationnel.
   > 69% ont juste en 2009, 71% en 2010.

3. Toute partie d'un langage rationnel est rationnelle. a. vrai/b. faux ?

   > **Correction:** N'importe quel langage L sur un alphabet Σ vérifie $L \subset \Sigma^\star$, donc bien sûr que c'est faux.
   > 70% ont juste en 2009, 58% en 2010.

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 2009 a juste !** 69% pour la 2010.

Best-of:
- Une infinité
- $n-1$
- $n+1$
- $n!$
- $n!+1$
- n^n+1
- n^2-n-1
- n^2-n+1
- $(n-1)^2+n+1$
- $n+\sum_{i=2}^{n-1}\sum_{k=1}^{i}k+1$
- $2+\sum_{i=1}^{n-1}(n-i)\times n^i$
- $\sum_{i=1}^{n} i!$
- $(n(n-1))/2-1$
- Il y a n sous-ensemble (sic) : chaque élément peut-être un sous-enemble.
- Il existe un nombre fini de sous-ensembles. Ce nombre est dénombrable.
- Pour un ensemble de taille n, il existe N sous-ensembles possibles où $N =$ toutes les combinaisons possibles des n éléments (ceci incluant l'ensemble vide).
- Il existe $n/(taille(sous-ensemble))$

2. Combien existe-t-il de mots de n lettres écrits dans un alphabet de m symboles ?

Correction: Seuls 5% de la 2009 a juste ! 90% en 2010.

Best-of:
- C_m^n
- En supprimant les doublons (mot identique construit de façon différente), l'on obtient : $mots = m^n$, soit le nombre de symbol(s) possible puissance le nombre de lettre qui forme un mot unique.

3. Combien de valeurs différentes peut coder un octet ?

Correction: 256. 73% en 2010.

Best-of:
- Une valeur (techniquement cette réponse est juste, mais je l'ai comptée fausse).
- 2
- Il y a 8 valeurs différentes pour coder un octet.
- 16 : de 0 à F(=15)
- $2^7 = 64$
- 64
- 128
- $2^8 - 1$
- 255 bien évidemment !
- Un octet = 8 bit soit $2^8 - 1$ possibilité = 256.
- $2^9 = 256$
- De 0 à 128, donc 129 valeurs possibles.
- 258.
- Il faut huit valeurs (car "octo" en Grec veut dire huit).
- $2^8 - 1 = 511$.
- 1 octet = 8 bits $\Rightarrow 2^7 = 128$ valeurs différentes.
- $2^9 - 1$
- 2^9
- 1024
- 8^8
- Tout dépend du point de vue :
  - En prenant un octet sans y appliqué d'algorithme $\rightarrow 256$.
  - En prenant le bit de poid fort comme bit de signe $\rightarrow 256$ aussi (-128,128).
  - En y appliquant des algorithmes (loi A, loi y) on peut obtenir un panel plus important.

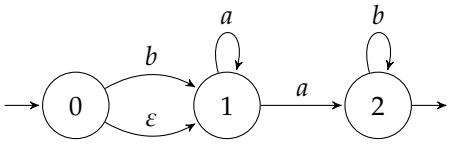
4. À quel linguiste les informaticiens doivent-ils le défrichage d'une partie importante de la théorie des langages formels ?

Correction: Chomsky. 68% en 2010.

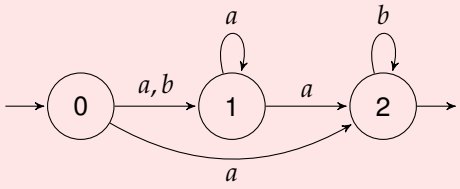
Best-of:
- Donald Knuth est souvent cité.
- Puis vient John Backus.
- Niklaus Wirth.
- Chuck Noris.
- Alan Turing.
- Chowski.
- Excellente question dont je ne connais malheureusement pas la réponse.
- Il s'agit du linguiste Noam Chomsky qui a apporté beaucoup à la hiérarchie de Chomsky.
- Tigrou.
- Huffman.
- sigour_b.

3 Automates

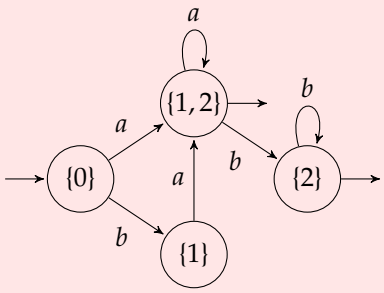
Déterminiser rigoureusement l'automate suivant.



Correction: Bien sûr il faut commencer par effectuer l'ϵ-fermeture :



automate déterministe :



Barème:

0 rien de bon

1 résultat correct mais qui vient de l'espace ; ou démarche correcte, résultat manifestement faux (e.g., pas déterministe, pas d'état d'acceptation ou pas d'état initial, des transitions spontanées, etc.).

2 résultat correct, mais peu justifié. Par exemple, l'élimination des transitions spontanées n'est pas explicite, mais le déterminisé montre les numéros de l'automate de départ.

3 démarche correcte, résultat faux

4 tout bon

Ainsi noté, le score de la promo 2010 est 21%.

Best-of: J'ai vraiment vu beaucoup de bétises, et je n'ai pas eu le courage de toutes les noter. Quelques unes que j'ai retenue :
– C'est un automate à pile.
– Cet automate est LR0 (sic).
– Cet automate n'est pas rationel.
– Beaucoup d'étudiants donne une regexp comme seul résultat...
– Certains me donnent une grammaire linéaire.
– Après bcp de calcul, certains concluent que l'automate n'est pas déterministe.
– « L'automate suivant n'est pas rationnelles. On ne peu pas le déterminiser. Il n'est pas reconnaissable par un automate fini. Exemple impossible de déterminisation : [un automate simple pour `aa*bb*`, parfaitement déterministe, sur lequel l'étudiant note "boucle" sur les boucles, et écrit ensuite :] Impossible car il reste des boucles. »
– « Je vais tenter de déterminiser rigoureusement l'automate.

```
S  →  A                          S  →  A | b A
S  →  b A                     →  A  →  a B | a A B          ».
A  →  a B                        B  →  b B | ε
A  →  a A B
B  →  b B
B  →  ε
```

4 Parsage LL et LR

Soit U un langage pour la programmation de robots offrant des primitives de composition de processus séquentielle et parallèle :

```
p | q  # Exécuter p et immédiatement à la fin d'icelui, exécuter q.
p & q  # Démarrer p et q de façon synchrone.
p ; q  # Exécuter p puis q après la fin de p (possiblement bien plus tard).
p , q  # Lancer p puis sans attendre sa fin, exécuter q (peut-être plus tard).
```

Les accolades, '{' et '}', groupent les processus.

1. Écrire la grammaire naïve de ce langage en utilisant 'p' pour désigner les processus élémentaires. On prendra garde d'utiliser des conventions typographiques permettant de distinguer les symboles du langage U de ceux du formalisme des grammaires.

Correction:

`S → S ';' S | S ',' S | S '&' S | S '|' S | '{' S '}' | p`

Beaucoup trop d'étudiants, sans doute croyant bien faire, commence déjà à régler certaines ambiguïtés (typiquement celles de l'associativité). C'est bien trop tôt : non seulement il faut commencer par se faire une idée du langage seul, débarassé des problèmes d'associativité et priorité, mais en plus l'énoncé n'a pas encore donné d'indication à leur sujet ! Sans compter par ailleurs que ça ne les gênent pas ensuite pour dessiner des arbres de dérivation sans rapport avec leur grammaire, puisqu'ils en font deux, alors que leur grammaire n'en accepte qu'un. Les accolades sont aussi souvent oubliées. Enfin, certains me rajoutent un symbole pour EOF : c'est totalement inutile ici, ça n'est intéressant que lorsqu'on calcule un parseur (LL ou LR), mais quand on fait une étude théorique de la grammaire.

Best-of:
- IMPORTANT. Dans les grammaires qui vont suivre, les éléments de U (donc les terminaux) seront entre cotes. Celà uniquement pour un soucis de clairvoyance du formalisme des grammaires.
- Par souci de lecture, je remplace le '|' de séparation des éléments de la grammaire par '/'

```
C  →  L
L  →  p O / p
O  →  | L / , L / ; L / & L
```

[Je sais pas vous, mais moi j'ai un mal de chien à lire, il fallait à la rigueur faire le contraire.]
- Voici une grammaire simple pour exécuter les commandes avec des conditions simples.

```
I  →  if E then B
B  →  execute A | execute A and A
E  →  A ended
A  →  p | q
```

[J'arrête de recopier, mais les arbres de dérivation par exemple, sont de la même veine !]

- `S → S ';' S | S ',' S | S '&' S | S '|' S | '{' S '}' | L`
  `L → 'a' | 'b' | 'c' | ...`

  [ben oui, sinon on est embêté pour 'p'...]

2. Dessiner les arbres de dérivation et leur arbre de syntaxe abstraite de '`p, q ; r`' correspondant à deux opérateurs associatifs à gauche. Faire de même en associativité à droite.

   Correction: Trop d'étudiants dessinent des ASTs au lieu d'arbres de dérivation. Et chose surprenante : les arbres d'associativité gauche sont pris pour de l'associativité droite, et inversement. Un étudiant note même « ça paraît bizarre que l'arbre à gauche penche à droite, et l'arbre à droite penche à gauche ». Par ailleurs, les règles « terminales » ($S \to p$) sont souvent oubliées, donnant lieu à des dérivations du genre $S \to p; q$ etc.

3. La sémantique de '`p ; q`' est la même qu'en shell. Celle de '`p , q`' est comparable à celle de '`p& q`' en shell avec l'importante différence que dans '`{p, q} ; r`', '`r`' attendra la fin de '`p`' et de '`q`' pour commencer.
   Tous deux,'`,`' et '`;`', ont même priorité. Étant donnée la sémantique voulue et pour que '`,`' se comporte comme '`&`' en shell dans une phrase comme '`p, q ; r`', quelle associativité faut-il prendre pour '`,`' et '`;`'?

   Correction: Il s'agissait de réfléchir à la sémantique du langage, et comment l'implémenter le plus naturellement possible. Bien sûr un AST peut être complètement tordu, tant que le code qui suit est capable de s'en débrouiller. Bien entendu, ce n'était pas l'esprit de la question.
   Si on prend associativité gauche pour les deux opérateurs, alors '`p, q ; r`' se comprend '`(p, q) ; r`', ce qui signifie que l'on va attendre la fin de '`(p, q)`' pour commencer '`r`', ce qui est contraire à la sémantique voulue pour '`,`'. On prendra donc une associativité droite pour les deux.
   Les étudiants qui donnent une associativité différente à ces deux opérateurs ne répondent pas à la question d'une utilisation des deux opérateurs ensemble, comme c'est le cas ici.

4. Étant donnée leur sémantique, discuter l'associativité et la commutativité de '`&`' puis celles de '`|`'.

Correction: Bien sûr '&' est commutatif, et '|' ne l'est pas. Enfin, l'associativité des deux tombe sous le sens.

Best-of:
- Il faudrait que '&' soit associatif à gauche et à droite, comme par exemple pour la division.
- '|' peut être commutatif à gauche aussi.
- Pour '&' on peut choisir l'associativité double. Pareil pour la commutativité.
- '&' est commutatif, et son associativité devrait être supérieure à '|'.
- '&' n'est pas associatif car 'P & (Q | R) ≠ (P & Q) | (P & R)'. '|' est associatif car 'P | (Q & R) = (P | Q) & (P | R)'.
- La commutativité est logiquement impossible pour '|' il ne s'agit pas d'un "et" ou "ou" logique.
- Pour 'p & q', p et q s'exécutent en même temps, donc l'associativité n'a pas d'importance.
- Ils ne sont pas commutatifs, car le résultat de 'P | Q & R' est différent de 'R & P | Q'.

5. Pour rester semblable au modèle du C, on donne une priorité supérieure à '&' sur '|'. Pour aider LL, quelle associativité leur donner ?

Correction: Comme on l'a vu de nombreuses fois, les implémentations récursives de LL partent en boucle infinie si on a une récursion gauche, directe ou indirecte. Il faut donc prendre une récursion droite.

Best-of:
- Cette question est ma plus grosse surprise : la très large majorité des étudiants répond ici « récursion gauche », et je ne comprends pas pourquoi.
- Pour aider LL qui n'aime pas les récursions à droite, on leur donne une associativité droite.

6. Sachant que ',' et ';' sont les moins prioritaires, donner une grammaire non ambiguë de U.

Correction: Merci de bien lire l'énoncé : la question précédente vient de donner une priorité supérieure à '&' sur '|' ! La vaste majorité des étudiants n'en ont pas tenu compte et sont restés accrochés à une n-ième re-syntaxisation de l'arithmétique.

```
S → T | T ';' S | T ',' S
T → F | F '|' T
F → P | P '&' F
P → p | '{' S '}'
```

Best-of:
- La grammaire précédemment donnée pour ce langage n'est pas ambiguë, je peux donc la réutiliser :

```
R → S | { S }
S → p | p X
X → '&' R
  | '|' R
  | ';' R
  | ',' R          Ici l'ordre des priorités est correct.
```

7. Pourquoi cette grammaire n'est pas LL(1) ?

Correction: Il est impossible en regardant le lookahead de distinguer les règles d'un nonterminal donné. Par exemple, les trois règles sur S ont le même préfixe T.
Beaucoup d'étudiants ont donné une grammaire avec des opérateurs récursifs à gauche, et dans ce cas j'ai accepté leur explication : pas LL(1) car récursif à gauche.

Best-of:
– Cette grammaire n'est pas LL(1) car nous n'avons pas introduit ϵ (le mot vide).
– « segfault » (et fin de la copie).
– Cette grammaire n'est pas LL(1) à cause de '`{ T }`' qui engendre un langage non rationnel.

8. Est-elle LL(2) ?

Correction: Non, pour les mêmes raisons : T est un non terminal, et k lookahead ne suffiront jamais pour trouver le terminal qui le suit, ce qui permettrait de distinguer les trois règles de S.

Best-of:
– [Après avoir répondu que la grammaire précédente était bien LL(1)] Non.
– Cette grammaire n'est pas LL(2) car les tokens se lisent un par un et nous deux à deux.

9. Généraliser cette grammaire en utilisant des opérateurs rationnels (EBNF).

Correction:

```
S  →  T ((';' | ',') T)*
T  →  F ('|' F)*
F  →  P ('&' P)*
P  →  '{' S '}' | p
```

Best-of:
– Je ne m'y risquerais pas en ce moment vus les taux de la BNF.

10. Écrire la routine de parsage d'une implémentation conventionnelle de LL(1) en C (parseur prédictif récursif descendant) pour l'axiome (le symbole de tête) de cette grammaire EBNF. Pour ce faire, utiliser les déclarations suivantes :

```c
/* The type of the processus trees. */
typedef ... proc_t;
/* The connectives: ';', ',', '&', '|'. */
typedef enum { conn_semicolon, conn_comma, conn_and, conn_pipe } conn_t;

/* Return a processus which composes lhs and rhs with conn. */
proc_t* proc_new (conn_t conn, proc_t* lhs, proc_t* rhs);

/* The lookahead. */
token_t la;

/* 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);
```

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: Il était demandé une seule routine, ne nombreux d'étudiants ont cru intelligent de faire du remplissage et de donner tout (ou presque) le parser. Bien entendu, plus vous en écrivez, plus la probabilité qu'il y ait quelque chose de faux est importante.

```
proc_t*
parse_S ()
{
  proc_t* res = parse_T();
  while (la == ';' || la == ',')
    switch (la)
    {
      case ';':
        eat(';');
        res = proc_new(conn_semicolon, res, parse_T());
        break;
      case ',':
        eat(',');
        res = proc_new(conn_semicolon, res, parse_T());
        break;
    }
  switch (la)
  {
    case '}':
    case EOF:
       // These are in FOLLOW(S), correct.
       break;
    default:
       // Not in FOLLOW(S), parse error.
       error (location,
              "unexpected ‘%s’, expected ‘;’ or ‘:’ or ‘)’ or %EOF",
              token_to_string (la));
       break;
  }
  return res;
}
```

Correction: Mais les lecteurs assidus remarqueront que cette réponse est fausse, car contrairement aux années précédentes, ces opérateurs sont associatifs à droite, cas pour lequel la récursion est plus naturelle.

```
proc_t*
parse_S ()
{
  proc_t* res = parse_T();
  switch (la)
  {
    case ';':
      eat(';');
      res = proc_new(conn_semicolon, res, parse_T());
      break;
    case ',':
      eat(',');
      res = proc_new(conn_semicolon, res, parse_T());
      break;
    case '}':
    case EOF:
       // These are in FOLLOW(S), correct.
       break;
    default:
       // Not in FOLLOW(S), parse error.
       error (location,
              "unexpected '%s', expected ';' or ':' or ')' or EOF",
               token_to_string (la));
       break;
  }
  return res;
}
```

J'ai été **très** généreux sur la correction. Personne ne m'a donné la bonne solution.

11. On souhaite à présent étudier la possibilité d'une implémentation en Bison de la grammaire précédente. Traduire la grammaire de la question 6 en une grammaire Bison exploitant les primitives d'associativité et de priorité.

Correction: Comme '&' et '|' sont associatifs, pour aider le parser, on les prendra associatifs à gauche.

```
%right ',' ';'
%left '|'
%left '&'
%%
p: p ',' p
 | p '|' p
 | p '&' p
 | p ';' p
 | '{' p '}'
 | 'p';
```

Best-of:
– Les bisons ont été exterminés par les cowboys malheureusement.
– '%%'

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

Correction: 20% de réussite en 2009, 70% en 2010.
La récursion gauche économise la pile, puisqu'elle permet de réduire plus rapidement.

Best-of:
- (Réponse d'un étudiant qui avait donné cette réponse à la question similaire mais en LL, puis qui l'avait cerclée en écrivant « bidon » à côté) Enfin je sais ! (Puis la bonne réponse, et enfin :) Oui, celà ressemble à ma réponse de la question 5, mais celle-là est la bonne... Je dis n'importe quoi à la 5...
- Les parsers LR préfèrent la récursion gauche car ceux-ci sont récursif droite [Je suppose que c'est une forme de sexualité chez les parseurs : ils cherchent leur contraire.]. Cependant la récursion droite n'est pas un problème pour eux [les parseurs ne sont pas homophobes], mais pour la mémoire.
- « Les parsers LR préfèrent » [sic]
- « Les parsers LR préfèrent les règles récursives à gauche (à cause des conflits shift/reduce) ».
- « Les parsers LR préfèrent la règle récursive à droite ». Ce qui est rigolo dans ce cas précis, c'est que c'est la seule question traitée sur les 14 de tout ce problème.
- « Les parsers LR prefere les recursions à droite cqr ils utilisent une pile, pour les recursions à gauche il faut LL. »
- LR prefaire recursive gauche pour éviter que le lookahead parte trop loin.

13. Compléter la séquence de décalages/réductions de la phrase suivante.

```
   ⊢           p, p, p ⊣
s  ⊢'p'          , p, p ⊣
```

Correction: 32 % pour les 2010. Score bien faible, pour un exercice **très** facile.

```
    ⊢           p, p, p ⊣
 s  ⊢'p'          , p, p ⊣
 r  ⊢p            , p, p ⊣
 s  ⊢p',            p, p ⊣
 s  ⊢p',' 'p'         , p ⊣
 r  ⊢p                , p ⊣
 s  ⊢p',                p ⊣
 s  ⊢p',' 'p'             ⊣
 r  ⊢p                    ⊣
acc ⊢p⊣
```

Best-of:

```
—   p, p, p ⊣
     , p, p ⊣
       p, p ⊣
         , p ⊣
           p ⊣
             ⊣


—  ⊣    p, p, p ⊣
   ⊣ p    , p, p ⊣
   ⊣ p,     p, p ⊣
   ⊣ p, p     p, p ⊣
   ⊣ p, p,      p ⊣
   ⊣ p, p, p      ⊣
```

14. On souhaite également utiliser le langage U avec un interprète de commande interactif, comme pour un shell Unix. Et, toujours comme pour un shell, on souhaite que '`p,`' lance '`p`' en « tâche de fond ».
    Expliquer pourquoi cette implémentation en Bison ne convient pas.

    Correction: Question très mal posée : je voulais faire remarquer que si un opérateur est associatif à droite (comme '`;`' et '`,`'), alors le parser attendra la dernière commande pour traiter toutes celles qui sont stockées. En clair, l'interactivité nécessite l'associativité gauche, ce qui est contraire à ce que l'on veut ici. Il faut trouver d'autres implémentations, comme par exemple en faire des opérateurs n-aires. Ou bien en faire des terminateurs au lieu de séparateurs.
    J'ai accepté les réponses notant que ce sont des opérateurs binaires, et non des terminateurs, et que par conséquent on devra toujours attendre le membre de droite.

    Best-of:
    – Le token '`,`' n'est pas un terminal.
    – Cela suppose qu'il y a une communication entre les deux processus (père, fils).

5 À propos de ce cours

Bien entendu je m'engage à ne pas tenir compte des renseignements ci-dessous pour noter votre copie. Ils ne sont pas anonymes, car je suis curieux de confronter vos réponses à votre note. En échange, quelques points seront attribués pour avoir répondu. Merci d'avance.

Vous pouvez cocher plusieurs réponses par question. Répondez sur les feuilles de QCM qui vous sont remises.

4. Travail personnel
   a Rien
   b Bachotage récent
   c Relu les notes entre chaque cours
   d Fait les anales
   e Lu d'autres sources
5. Ce cours
   a Est incompréhensible et j'ai rapidement abandonné
   b Est difficile à suivre mais j'essaie
   c Est facile à suivre une fois qu'on a compris le truc
   d Est trop élémentaire
6. Ce cours
   a Ne m'a donné aucune satisfaction
   b N'a aucun intérêt dans ma formation
   c Est une agréable curiosité
   d Est nécessaire mais pas intéressant
   e Je le recommande
7. L'enseignant
   a N'est pas pédagogue
   b Parle à des étudiants qui sont au dessus de mon niveau
   c Me parle
   d Se répète vraiment trop
   e Se contente de trop simple et devrait pousser le niveau vers le haut