Correction du Partiel Théorie des Langages

EPITA – Promo 2007

Juillet 2005 (1h30)

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

Ce sujet est l'exacte copie de l'épreuve THL originelle que vous avez échouée.

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.

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 et son nom, e.g., 3, langage rationnel). On demande **évidemment** le type le plus précis.
- proposer une grammaire *non ambiguë*, de même type de Chomsky, qui l'engendre. Les grammaires longues peuvent être esquissées et se terminer par « ... ».
- proposer un outil engendrant un analyseur adéquat pour saisir *la structure* du langage auquel il est fait référence. On ne demande pas l'outil minimum correspondant au rang du langage, mais celui qui permette de retrouver ce que l'on cherche dans chacun des cas.

1. Les mots du français.

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

   ```
   S ::= a
   S ::= abaisable
   S ::= abaissant
   ...
   ```

   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.

2. Les nombres entiers en base 3 non signés.

   Correction: Langage infini, mais trivialement rationnel —type 3— puisque représenté par l'expression rationnelle $(0|1|2)^+$. On peut par exemple en donner une grammaire linéaire à droite :

   ```
   N ::= 0 N | 1 N | 2 N
   N ::=  0  |  1  |  2
   ```

   Par contre, la grammaire suivante est bien évidemment fausse, puisqu'elle produit le mot vide :

   ```
   N' ::= 0 N | 1 N | 2 N | ε
   ```

3. Les expressions mathématiques composées de puissances (^), de parenthèses (« (», «) »), et de nombres entiers en base 3 non signés.

Correction: Il est bien connu qu'on est sorti des langages rationnels, pour tomber dans les langages hors-contexte —type 2. En effet, une grammaire hors-contexte serait (en utilisant `N`, la réponse de la question 2) :

```
S ::= S ^ T
S ::= T
T ::= ( S )
T ::= N
```

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

```
S ::= S ^ N
S ::= N
S ::= ( S )
```

car il ne serait plus possible d'engendrer `0^(0)`.

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 ¬ (non), des opérations binaires ∨ (ou) et ∧ (et), et des parenthèses. Ce langage inclut des mots tels que $f \wedge f$, $t \vee f$ et $\neg\neg(t \wedge t) \vee (f \wedge f)$.

1. Qu'avez-vous à dire de la grammaire suivante ?

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

   Correction: Elle correspond au langage recherché, elle est hors contexte, mais elle est ambiguë.

2. Qu'avez-vous à dire de la grammaire suivante ?

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

   Correction: Elle correspond au langage recherché, elle est hors contexte, et elle n'est plus ambiguë.

3. En déduire les priorités et associativités des opérateurs.

   Correction:
   (a) ∧ et ∨ sont associatives à gauche ;
   (b) ¬ est prioritaire sur ∧ ;
   (c) ∧ est prioritaire sur ∨.
   c'est-à-dire que $f \vee t \vee f \wedge \neg\neg t \wedge f$ se lit $(f \vee (t \vee ((f \wedge (\neg(\neg t))) \wedge f)))$.

4. Donner deux raisons évidentes pour lesquelles cette grammaire n'est pas LL(1).

   Correction: D'une part elle contient des règles récursives à gauche, d'autre part, plusieurs règles sont en concurrence (ont des préfixes non nuls communs donc des FIRST égaux).

5. Calculer NULLABLE, FIRST et FOLLOW pour `S`, `T` et `T` dans la grammaire 2.

   Correction:

	NNULABLE	FIRST	FOLLOW
S	n	$\neg(tf$	$\vee\wedge)$
T	n	$\neg(tf$	$\vee)$
F	n	$\neg(tf$	$)$

6. Qu'avez-vous à dire de la grammaire suivante ?

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

   Correction: Elle correspond toujours au langage, elle est hors contexte, non ambiguë, et même LL(1). Par contre, elle rend compte d'opérateurs associatifs à droite, au lieu d'à gauche.

7. Qu'avez-vous à dire de la grammaire suivante ?

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

Correction: Non ambiguë, engendre le bon langage, et rend compte des bonnes priorités et associativités. Elle est donnée en syntaxe EBNF, i.e., avec des opérateurs rationnels en partie droite. Nickel pour un LL programmé à la main.

8. Écrire en pseudo code la routine de parsage LL(1) de `T` dans la grammaire 7.

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.

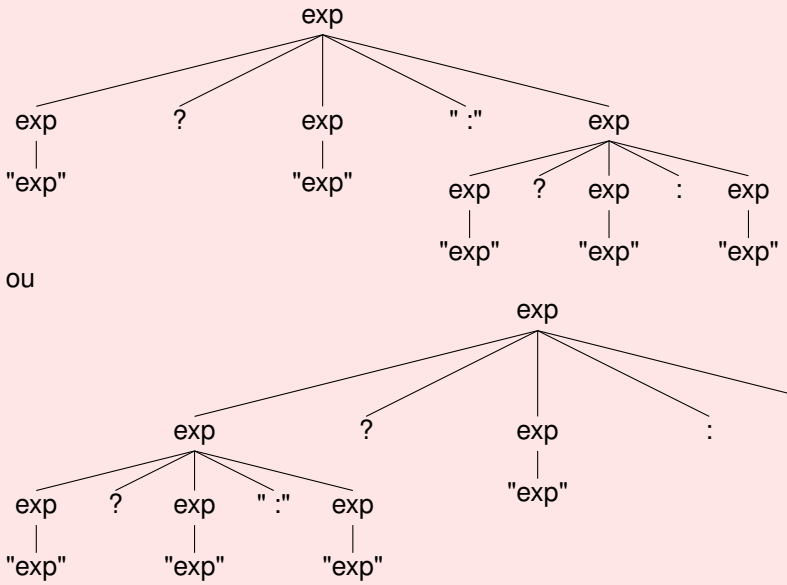
3 Opérateurs n-aires

Soit le fragment suivant d'une grammaire d'un langage concernant l'opérateur `?:` :

```
exp: "exp" | exp "?" exp ":" exp
```

1. Montrer que cette grammaire est ambiguë en exhibant tous les arbres de dérivation engendrant le plus petit mot ambigu.

   Correction: Ce mot est `"exp" ? "exp" ":" "exp" "?" "exp" : "exp"`, qui peut donner la dérivation

2. Cette ambiguïté est tout à fait comparable à une classe bien connue d'ambiguïté d'opérateurs, laquelle ?

   Correction: C'est un problème d'associativité : cet opérateur est-il associatif à droite, ou à gauche ? En fait, on se moque de la présence du `exp` du milieu (qui, encadré par ses "parenthèses" `?` et `:`, sera toujours facile à parser sans ambiguïté) : tout se passe comme si la grammaire était celle d'un opérateur $\otimes$ valant `? exp :`, i.e., `exp` : $''$`exp`$''$|`exp` $\otimes$ `exp`.

3. On résoudra cette ambiguïté en ne conservant que l'une de ces possibilités. Discuter tous les choix possibles et leurs vertus pour un langage de programmation.

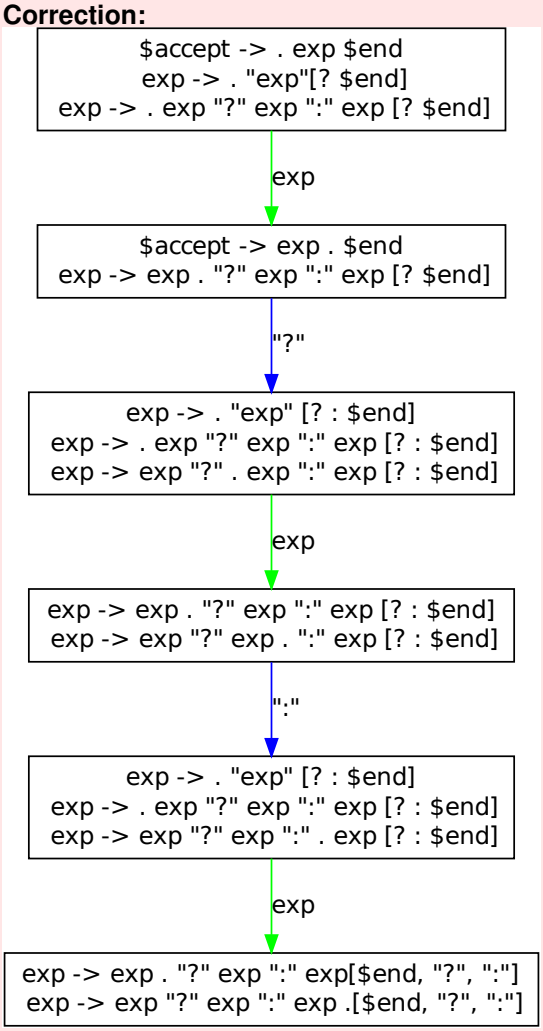
   Correction: Si l'on prête le sens habituel de if-then-else, alors l'associativité à gauche est difficile à lire, alors que l'associativité à droite est celle, bien habituelle, d'une cascade de else-if, else-if etc.
   N'empêche que s'il s'agit de faire plaisir à Bison/Yacc, alors prendre l'associativité gauche pour sauver la pile.

4. Écrire une grammaire non-ambiguë (sans annotation `%foo`) engendrant le même langage ne retenant qu'une interprétation.

   Correction:
   `exp: "exp" | "exp" "?" exp ":" exp`
   On a fixé le `exp` de gauche pour forcer l'associativité droite.

5. Dessiner la partie de l'automate LR(1) conduisant à l'état (aux états) exhibant le conflit (les conflits) résultant de l'ambiguïté de la grammaire originale.

Correction:



On voit "bien" que l'on veut réduire et décaler sur `?` dans le dernier des états représentés.

6. On rappelle qu'en Yacc, une règle hérite de la priorité/associativité de son terminal le plus à droite. Quelles sont les trois possibilités les plus simples d'annotation (`%foo`) de cette grammaire pour qu'il n'y ait plus de conflit ?

Correction: En fait il y en a plus de trois, mais il y a trois résultats possibles (en prenant `%prec` pour représenter `%left`, `%nonassoc`, ou `%right`) :

associatif à gauche Soit `%prec ':' %prec '?'`, soit `%left '?' ':'`.

nonassociatif `%nonassoc ':' '?'`.

associativité à droite Soit `%prec '?' %prec ':'`, soit `%right '?' ':'`.

7. Comparer les avantages du parsage de ce langage par, d'une part une implémentation de la grammaire désambiguïsée de la question 4 en Yacc, et, d'autre part, celle exploitant les directives de Yacc comme dans la question 6.

Correction: Utiliser les directives est beaucoup plus simple (“fixer” `exp` dans le cas d’une grammaire complète est beaucoup plus compliqué et demande l’introduction de symboles supplémentaires), et conduit à un automate plus petit, donc plus performant.

8. L’une des possibilités de la question 6 implique que la grammaire annotée ainsi définie n’engendre pas le même langage. Pourquoi ?

   Correction: Ben si c’est non associatif, alors la phrase qui nous a servi à montrer l’ambiguïté n’est plus reconnue. On a donc changé le langage.

9. Parmi les possibilités de la question 6 conduisant à une grammaire annotée en Yacc, quel est le choix exerçant la plus faible pression sur le parseur ?

   Correction: On l’a déjà dit : associatif à gauche.

10. Informellement, mais de façon convaincante, traiter l’extension suivante de cet opérateur :

```
exp: "exp"
   | exp "?" ":" exp
   | exp "?" exp ":" exp
   | exp "?" exp "," exp ":" exp
```

    On attend en particulier une description de l’ambiguïté de cette grammaire, et un mode d’implémentation Yacc.

    Correction: C’est pareil ! On n’a pas de problème au milieu, c’est bien délimité. Juste des problèmes d’associativité, et les solutions ci-dessus sont suffisantes.

11. Critiquer l’extension suivante :

```
exp: "exp"
   | exp "?" exp ":" exp
   | exp "?" exp ":" exp ":" exp
```

    Correction: Là par contre on introduit des problèmes supplémentaires comparable au “dangling else” quand il y a plusieurs `?`, à qui se rattache le dernier `:` ?