# Correction du Partiel de Compilation

Promo 2004 - Avril 2002

Une rédaction simple mais convaincante et une mise en évidence des résultats seront très appréciées des correcteurs...

## 1 Typologie des Langages

Soit le programme suivant, écrit pour mettre en valeur les différences entre *Mode*s de passage d'arguments aux routines :

```
var t    : integer
    foo  : array [1..2] of integer;

procedure shoot_my (x : Mode integer);
begin
  foo[1] := 51;
  t      := 2;
  x      := x + 69;
end;

begin
  foo[1] := 1;
  foo[2] := 2;
  t      := 1;
  shoot_my (foo [t]);
end.
```

Compléter le tableau suivant (sur votre copie) en explicitant les valeurs de `foo[1]`, `foo[2]`, et `t` à la fin du *programme*.

| *Mode* (Passage par...) | `foo[1]` | `foo[2]` | `t` |
|---|---|---|---|
| Valeur | | | |
| Valeur-Résultat (Algol W) | | | |
| Valeur-Résultat (Ada) | | | |
| Référence | | | |
| Nom (Algol 60) | | | |

On rappelle qu'en Algol W, l'adresse de la lvalue est calculée au retour, alors qu'en Ada elle l'est à l'appel. On rappelle également que les erreurs de calcul en Ing1 ne sont pas admissibles.

**Correction:**

| *Mode* (Passage par...) | `foo[1]` | `foo[2]` | `t` |
|---|---|---|---|
| Valeur | 51 | 2 | 2 |
| Valeur-Résultat (Algol W) | 51 | 70 | 2 |
| Valeur-Résultat (Ada) | 70 | 2 | 2 |
| Référence | 120 | 2 | 2 |
| Nom | 51 | 71 | 2 |

## 2 Parsage LL

1. Écrire la grammaire naïve de l'arithmétique avec les terminaux «1» (le seul nombre), «-» la soustraction binaire, «/» la division, «(» et «)» les parenthèses.

   **Correction:**
   ```
   E -> E - E
   E -> E / E
   E -> ( E )
   E -> 1
   ```

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

   **Correction:**

   (a) Priorité et associativité.

   ```
   E -> E - T
   E -> T
   T -> T / F
   T -> F
   F -> ( E )
   F -> 1
   ```

   (b) Récursion à droite.

   ```
   E -> T - E
   E -> T
   T -> F / T
   T -> F
   F -> ( E )
   F -> 1
   ```

   (c) Factoriation à gauche

   ```
   E  -> T E'
   E' -> - T E'
   E' -> /* Nothing. */
   T  -> F T'
   T' -> / F T'
   T' -> /* Nothing. */
   F -> ( E )
   F -> 1
   ```

3. Expliquer pourquoi l'implémentation LL(1) de cette grammaire sera sûrement plus performante que le parsage par automate LALR(1).

   **Correction:** Un compilateur ne peut rien comprendre de la structure d'un automate codé dans un grand tableau. Par contre, du code plus "traditionnel", avec des routines qui s'appellent les unes les autres, correspondent au modèle pour lequel il a été développé, et par conséquent sont plus aptes à toutes formes d'optimisation.
   Dès lors, étant donné que l'implémentation naturelle d'un parseur LL est par des routines, et qu'une implémentation naturelle d'un parseur LALR est par un tableau-automate, le premier peut espérer de meilleures performances.
   À remarquer qu'on peut implémenter LL par tableau-automate, et LALR (en se faisant un peu mal au crâne) par routines, cette question ne prend son sens que dans les cadres traditionnels.

# 3 Parsage LALR(1)

Considérons la grammaire suivante, écrite selon la syntaxe Bicc/Yason.

```
%%
sentence: "PROC"  argument '.'  | "PROC"  parameter ';'
        | "MACRO" argument ';'  | "MACRO" parameter '.'   ;
argument: "VARIABLE";
parameter: "VARIABLE";
%%
```

1. Quel est son type de Chomsky ?

   **Correction:** Elle n'est pas régulière (pas linéaire du tout), mais clairement hors-contexte.

2. Est-elle ambiguë ?

   **Correction:** Trivialement non : il suffit d'écrire son langage, et de constater que chaque mot n'offre qu'un arbre de parse.

3. Est-elle déterministe ?

   **Correction:** On ne peut répondre formellement à cette question qu'après avoir répondu à la suivante.

4. Est-elle LR(1) ?

   **Correction:** On peut répondre de deux façons : soit dessiner l'automate (pas très gros), soit simplement constater qu'on peut toujours prendre les décisions en regardant le prochain token. Par exemple, au point 0, savoir si le prochain token est `PROC` ou `MACRO` permet d'avancer sans ambiguïté, puis on shift le `VARIABLE`, et à ce moment-là, le lookahead `;` ou `.` permet de savoir s'il faut réduire en `argument` ou `parameter`.
   Donc elle est LR(1), donc elle est déterministe.

5. Sa compilation par Bicc/Yason échoue avec deux conflits réduction/réduction. Le rapport de génération de l'automate est fourni plus bas.

   Expliquer ce conflit, i.e., déchiffrer le rapport et expliquer dans quel cas on tombe sur ce conflit.

   **Correction:** Quand, par exemple, on a lu un `MACRO` puis `VARIABLE`, l'automate, confondant les lookaheads, ne sait plus s'il faut réduire en `argument` ou en `parameter`.

6. Expliquer l'*origine* de ce conflit, i.e., expliquer pourquoi cette grammaire contient un conflit.

   **Correction:** Cette grammaire est LR(1), mais LALR(1) montre un conflit. C'est donc que cette grammaire n'est pas LALR(1) est que le conflit vient d'un abus de simplification des ensembles de lookaheads par LALR(1).
   En clair, LALR(1) n'est pas une technologie adaptée à cette grammaire.

7. Est-ce qu'en dépit du conflit, le parseur est sain ? (On rappelle que par exemple dans le cas du `else` qui pendouille, le conflit est inoffensif.) Justifier.

**Correction:** Non, il est vraiment malsain : on voit dans le rapport de Bison que dans l'état 4 :

```
    argument  ->  "VARIABLE" .   (règle 5)
    parameter  ->  "VARIABLE" .   (règle 6)

    '.'         réduction par la règle 5 (argument)
    '.'         [réduction par la règle 6 (parameter)
    ';'         réduction par la règle 5 (argument)
    ';'         [réduction par la règle 6 (parameter)
```

autrement dit, quelque soit le lookahead, Bison va réduire une `VARIABLE` en `argument`. Donc toutes les phrases où il fallait faire un `parameter` sont vouées à un parse error.
Ce parser jette la moitié du langage : il ne comprend ni `PROC VARIABLE;`, ni `MACRO VARIABLE.`.

Le rapport de Bison est :

```
L'état 4 contient 2 conflicts réduction/réduction.

Grammaire

  Numéro, Ligne, Règle
    0   2 $axiom -> sentence $
    1   2 sentence -> "PROC" argument '.'
    2   2 sentence -> "PROC" parameter ';'
    3   3 sentence -> "MACRO" argument ';'
    4   3 sentence -> "MACRO" parameter '.'
    5   4 argument -> "VARIABLE"
    6   5 parameter -> "VARIABLE"

Terminaux, suivis des règles où ils apparaissent

$ (0) 0
'.' (46) 1 4
';' (59) 2 3
error (256)
"PROC" (258) 1 2
"MACRO" (259) 3 4
"VARIABLE" (260) 5 6

Non-terminaux, suivis des règles où ils apparaissent

$axiom (8)
    à gauche: 0
sentence (9)
    à gauche: 1 2 3 4, à droite: 0
argument (10)
    à gauche: 5, à droite: 1 3
parameter (11)
    à gauche: 6, à droite: 2 4

état 0
    $axiom  ->  . sentence $   (règle 0)

    "PROC"      décalage et aller à l'état 1
```

```
    "MACRO"     décalage et aller à l'état 2

    sentence    aller à l'état 3

état 1
    sentence  ->  "PROC" . argument '.'   (règle 1)
    sentence  ->  "PROC" . parameter ';'   (règle 2)

    "VARIABLE"  décalage et aller à l'état 4

    argument    aller à l'état 5
    parameter   aller à l'état 6

état 2
    sentence  ->  "MACRO" . argument ';'   (règle 3)
    sentence  ->  "MACRO" . parameter '.'   (règle 4)

    "VARIABLE"  décalage et aller à l'état 4

    argument    aller à l'état 7
    parameter   aller à l'état 8

état 3
    $axiom  ->  sentence . $   (règle 0)

    $           décalage et aller à l'état 9

état 4
    argument  ->  "VARIABLE" .   (règle 5)
    parameter  ->  "VARIABLE" .   (règle 6)

    '.'         réduction par la règle 5 (argument)
    '.'         [réduction par la règle 6 (parameter)
    ';'         réduction par la règle 5 (argument)
    ';'         [réduction par la règle 6 (parameter)
    $défaut     réduction par la règle 5 (argument)

état 5
    sentence  ->  "PROC" argument . '.'   (règle 1)

    '.'         décalage et aller à l'état 10

état 6
    sentence  ->  "PROC" parameter . ';'   (règle 2)

    ';'         décalage et aller à l'état 11

état 7
    sentence  ->  "MACRO" argument . ';'   (règle 3)

    ';'         décalage et aller à l'état 12

état 8
    sentence  ->  "MACRO" parameter . '.'   (règle 4)
```

```
    '.'         décalage et aller à l'état 13

état 9
    $axiom  ->  sentence $ .   (règle 0)

    $défaut     accepter

état 10
    sentence  ->  "PROC" argument '.' .   (règle 1)

    $défaut     réduction par la règle 1 (sentence)

état 11
    sentence  ->  "PROC" parameter ';' .   (règle 2)

    $défaut     réduction par la règle 2 (sentence)

état 12
    sentence  ->  "MACRO" argument ';' .   (règle 3)

    $défaut     réduction par la règle 3 (sentence)

état 13
    sentence  ->  "MACRO" parameter '.' .   (règle 4)

    $défaut     réduction par la règle 4 (sentence)
```

Les actions entre crochets sont celles qui, du fait d'un conflit, ne seront pas retenues.