Division par deux, Chomsky

`http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/IF/09/`

1 Forme normale de Chomsky

Selon le cours, une grammaire est en forme normale de Chomsky lorsque toutes ses productions sont de l'une des trois formes suivantes.

$$A \to a, \quad A \to BC, \quad S \to \epsilon$$

Où les majuscules désignent des non-terminaux, les minuscules des terminaux et S est le symbole de départ. Dans tous les exercices on ne considère que des grammaires qui n'engendrent pas le mot vide, c'est à dire que l'on se passe de la troisième sorte de production ci-dessus.

1. On considère la grammaire ambiguë des expressions arithmétiques.

   $$S \to E \qquad E \to \mathbf{i} \qquad E \to E \texttt{+} E \qquad E \to E \texttt{*} E$$

   Identifier les règles qui font que cette grammaire n'est pas en forme normale de Chomsky.
2. Mettre la grammaire en forme normale de Chomsky. On procédera à trois transformations successives des productions.
   (a) Éliminer les productions de la forme $A \to \dots a \dots$.
   (b) Éliminer les productions de la forme $A \to BCD \dots$.
   (c) Éliminer les productions de la forme $A \to B$.
3. On représente les grammaires par les objets de la classe `Rules` suivante.

   ```
   class Rules {
     char lhs ; List rhs ; Rules next ;

     Rules (char c, List s, Rules r) { lhs = c ; rhs = s ; next = r ; }
   }
   ```

   Tandis que les membres droits sont représentés par des objets de la classe

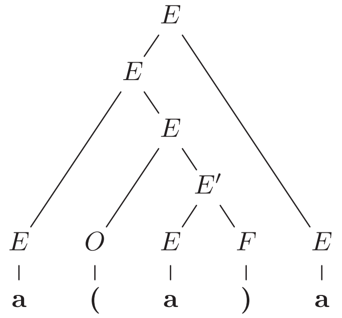
   ```
   class List {
     char val ; List next;

     List (char c, List r) { val = c ; next = r ; }
   }
   ```

   Écrire une méthode **static boolean** `isCNF(Rules rs)` qui vérifie qu'une grammaire est en forme normale de Chomsky. On adopte la convention que les non-terminaux sont les lettres majuscules (identifiées par `Character.isUpperCase(`**char** `c)`) et les terminaux les autres caractères qui ne sont pas des espaces (identifiés par `Character.isSpaceChar(`**char** `c)`).

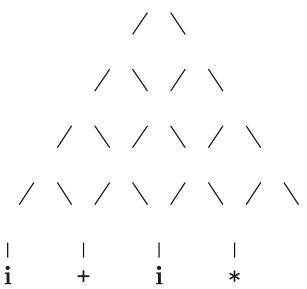
2 Analyse avec une pyramide

Soit un mot de terminaux, représenté comme une chaîne $\alpha[0 \dots 0 + \ell[$ (ℓ est la longueur de α). Plus généralement on note $\alpha[i \dots i + j[$ la sous-chaîne de α de longueur j et commençant à l'indice i. On peut représenter une analyse de α à l'aide d'une pyramide, chaque étage de la pyramide correspondant à des sous-chaînes d'une longueur donnée, le sommet correspondant à la longueur ℓ et la base à la longueur 1. Par exemple voici une analyse de `a(a)a`, dans la grammaire suivante qui est une forme normale de Chomsky de la grammaire des suites de « **a** » bien parenthésées.

$$E \to EE \qquad E \to OE' \qquad E' \to EF \qquad E \to \mathbf{a} \qquad O \to \texttt{(} \qquad F \to \texttt{)}$$



1. Dessiner une pyramide reflétant une autre analyse de `a(a)a`. On remarque que la pyramide n'est pas autre chose qu'un arbre de dérivation.
2. Systématiser le procédé en dessinant une pyramide reflétant *toutes* les analyses possibles de `i+i*i` dans la grammaire de Chomsky des expressions arithmétiques. On procédera par décoration des sommets de la pyramide « *complète* » suivante.



   Le i-ème sommet de la j-ème ligne est à décorer par l'ensemble des non-terminaux dont $\alpha[i \ldots i+j[$ peut dériver. Donner la condition qui prouve que α est bien un mot du langage de la grammaire.
3. Pour toute grammaire de Chomsky (qui n'engendre pas le mot vide) et tout mot non-vide α, donner le procédé général de décoration de la pyramide, en distinguant la base des autres sommets. On note $m_{j,i}$ la décoration portée par le $i+1$-ième sommet de la j-ème ligne de la pyramide.

3 Programmation de l'analyse

1. Commencer par identifier les structures de données dont nous allons avoir besoin.
   (a) Pour la pyramide elle-même et ses décorations.
   (b) Pour les grammaires supposées en forme normale de Chomsky.

   Il n'est pas demandé d'écrire le code des méthodes, seulement de les spécifier. La démarche consiste surtout à définir les classes dont on aura besoin, les spécifications des méthodes peuvent être complétées plus tard.
2. Programmer la méthode d'analyse

```
static boolean parse(Chomsky g, String alpha) ;
```

   On commencera évidemment pas remplir la base de la pyramide, puis on procédera du bas vers le haut, jusqu'à calculer $m_{\ell,0}$.
3. Programmer la classe `Chomsky`. Cette classe offre un constructeur.

```
Chomsky(char start, Rules rs)
```

Où `start` est le symbole de départ et `rs` sont les productions de la grammaire comme à l'exercice 1. On supposera que toutes les productions sont conformes et on utilisera la classe `Hashtable` de la librairie de Java. En effet on va construire une fois pour toute une association des membres droits de production vers les membres gauches correspondants.

Les tables de hachage sont des structures de données qui se comportent en gros comme des tableaux d'objets, mais dont les indices sont n'importe quels objets. On utilisera principalement deux méthodes (des objets de la classe `Hashtable`) :

```
// Renvoie l'objet associé à key, ou null si il n'y en a pas
 Object get(Object key)
 // Associe o à key, l'ancien objet associé à key est renvoyé
 Object put(Object key, Object o)
```

L'idée est bien évidemment d'associer des ensembles de membres gauches aux membres droits.

Les tables de hachage de Java demande de (re-)définir les méthodes `equals` et `hashCode` des clefs (à moins de prendre les clefs dans une classe standard, comme `String`).

```
// Égalité de this et de o
 public boolean equals(Object o)
// Renvoie un entier, deux objets égaux doivent renvoyer le même entier
 public int hashCode()
```

4 Bonus : mise en forme normale de Chomsky

Un document joint trouvé sur le Web donne une procédure générale de mise en forme normale de Chomsky. Nous allons nous intéresser à une étape en particulier : l'élimination des production vides $A \to \epsilon$.

On procède en deux temps, il faut d'abord identifier les non-terminaux *nullables*, c'est-à-dire ceux dont on peut dériver le mot vide.

On montre facilement qu'un non-terminal A est nullable si et seulement si :
- il existe une production $A \to \epsilon$,
- il existe une production $A \to BCD\ldots$, dont le membre droit est un mot de non-terminaux nullables.

Nous pouvons définir un opérateur $\mathcal{N}$ sur les ensembles de non-terminaux :

$$\mathcal{N}(S) = U \cup \{\, A, A \to \alpha,\ \text{avec } \alpha \in S^* \,\}$$

On construit ensuite la suite N_i d'ensembles de non-terminaux :

$$N_0 = \emptyset, \quad N_{i+1} = \mathcal{N}(N_i)$$

1. Montrer que la suite (N_i) converge vers l'ensemble des nullables.
2. On suppose donnée une classe `Symbols` des ensembles (de symboles) codée selon un style impératif :

```
class Symbols {
  // Constructeur de l'ensemble vide
  Symbols ()
  // Test d'appartenance
  boolean mem(char c)
  // Ajout d'un élément, renvoie true si this est modifié (ie, this ne
  // contenait pas déjà c)
  boolean add (char c)
}
```

Coder la méthode :

```
static Symbols getNullables(Rules rs)
```

Qui renvoie les nullables de la grammaire passée en argument.

3. Il s'agit maintenant de transformer la grammaire en une grammaire équivalente sans non-terminaux nullables. Le procédé est le suivant.
   - Remplacer les productions de la forme $A \to \alpha B \beta$, où B est nullable, par deux productions $A \to \alpha B \beta$ et $A \to \alpha\beta$. Attention, il faut considérer tous les nullables du membre droit, de sorte que si le membre droit contient n nullables, la production sera remplacée par 2^n productions (dont l'une peut d'ailleurs être de la forme $A \to \epsilon$).
   - Supprimer purement et simplement les productions de la forme $A \to \epsilon$.

   Coder la méthode :

   ```
   static Rules removeEmpty(Rules rs)
   ```

   Qui transforme la grammaire passée en argument en une grammaire équivalente, sans non-terminaux nullables.

Solutions des exercices

Exercice 1

1. Les deux productions $E \to E \texttt{+} E$ et $E \to E \texttt{*} E$ ne sont pas en forme normale de Chomsky car, d'une part leur membre droit contient trois symboles et d'autre part l'un de ces symboles est un terminal. La production $S \to E$ n'est pas non plus en forme de Chomsky.
2. On procède donc en trois étapes.
   (a) Il suffit de remplacer les terminaux par des non terminaux et d'ajouter les productions correspondantes.

   $$S \to E \qquad E \to \mathbf{i} \qquad E \to EPE \qquad P \to \texttt{+} \qquad E \to EME \qquad M \to \texttt{*}$$

   (b) Ici encore c'est assez facile : remplacer une règle $A \to BCD$ par deux règles $A \to BE$ et $E \to CD$.

   $$S \to E \qquad E \to \mathbf{i}$$
   $$E \to EE_p \qquad E_p \to PE \qquad P \to \texttt{+}$$
   $$E \to EE_m \qquad E_m \to ME \qquad M \to \texttt{*}$$

   (c) C'est à peine plus dur, il faut remplacer la production $S \to E$ par les productions $S \to \alpha$ où les α sont les membres droits des productions $E \to \alpha$.

   $$S \to \mathbf{i} \qquad S \to EE_p \qquad S \to EE_m$$
   $$E \to \mathbf{i}$$
   $$E \to EE_p \qquad E_p \to PE \qquad P \to \texttt{+}$$
   $$E \to EE_m \qquad E_m \to ME \qquad M \to \texttt{*}$$

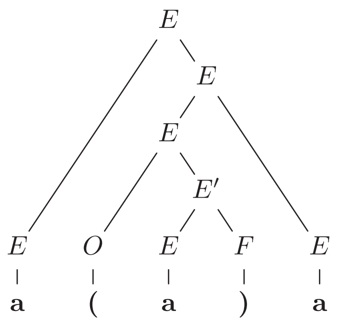
3. Code de `isCNF`, qui abuse des **return**, mais c'est tellement pratique.

```
static boolean isTerm(char c) {
  return !(isNonTerm(c) || Character.isSpaceChar(c)) ;
}

static boolean isNonTerm(char c) {
  return Character.isUpperCase(c) ;
}

static boolean isCNF(Rules rs) {
  for ( ; rs != null ; rs = rs.next) {
    List rhs = rs.rhs ;
    if (rhs == null) return false ; // Production vide
    if (rhs.next == null) {
      if (!isTerm(rhs.val)) return false ; // Production A → B
    } else {
      if (rhs.next.next == null) {
        if (!(isNonTerm(rhs.val) && isNonTerm(rhs.next.val)))
           return false ; // Membre droit de taille 2, distinct de AB
      } else {
        return false ; // Membre droit de taille >2
      }
```

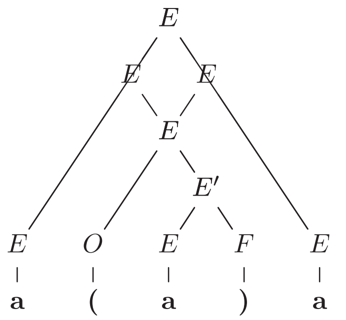
```
      }
    }
    return true ;
  }
```

Exercice 2

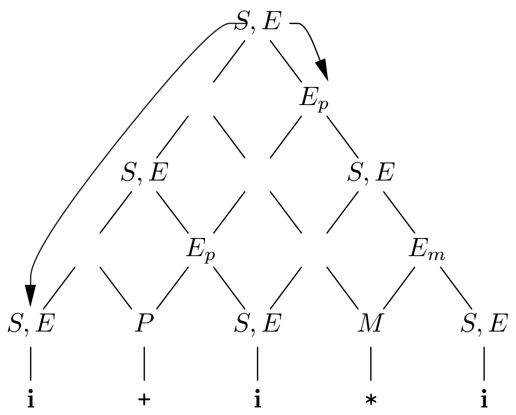
1. On peut y arriver sans se poser trop de questions. La modification porte sur le regroupement de `(a)a` en E (au lieu de `a(a)`). Les deux étages supérieurs de la pyramide sont modifiés.



Il n'est pas sans intérêt de faire se recouvrir les deux pyramides.



2. La condition demandée est S (symbole de départ) apparaît au sommet de la pyramide.



On peut justifier chaque décoration directement, ou plus subtilement à l'aide d'une décomposition de l'intervalle $[i \dots i+j[$ en un premier sous-intervalle $[i \dots k[$ complété par un second $[i{+}k \dots i{+}k{+}(j{-}k)[$, où B apparaît sur le sommet (k, i), C apparaît sur le sommet $(j{-}k, i{+}k)$

et $A \rightarrow BC$ est une production de la grammaire. Plus graphiquement, pour décorer un sommet donné on décompose la (sous-)pyramide définie par ce sommet en deux sous-pyramides qui couvre complètement sa base. On reconstitue ainsi un arbre de dérivation à partir de ses deux sous-arbres. Une reconstitution possible du sommet est indiquée, il en existe une autre qui regroupe la première sous-pyramide de la troisième ligne et la dernière de la deuxième ligne, selon la production $S \rightarrow SE_m$.

3. La base de la pyramide est de longueur ℓ.
   (a) On commence par calculer les $m_{1,i}$ (avec $0 \leq i < \ell$).

   $$m_{1,i} = \{ A, A \rightarrow c_i \in G \}$$

   Où c_i est le i-ème caractère de α.

   (b) En on peut ensuite calculer les lignes de la base vers le sommet :

   $$m_{j,i} = \{ A, A \rightarrow BC \in G, B \in m_{k,i}, C \in m_{j-k,i+k} \}$$

Exercice 3

1. Voici les classes utiles.
   (a) La pyramide sera représentée par un tableau des lignes de décorations, une ligne sera un tableau de décoration de la longueur appropriée, les décorations seront des ensembles de symboles (classe `SymbolSet`, les symboles sont des caractères). La déclaration de la pyramide sera donc

   ```
   SymbolSet [] [] m ; // Tableau de tableaux d'ensembles de symboles
   ```

   Pour éviter de nous fatiguer trop, définissons la classe `SymbolSet` comme des listes de caractères sans répétitions, l'ensemble vide étant représenté par **null**.

   ```
   class SymbolSet {
     char val ; SymbolSet next ;

     // test d'appartenance
     static boolean mem(char c, SymbolSet e)
     // ajouter un ^^e9l^^e9ment
     static SymbolSet add(char c, SymbolSet e)
     // union des ensembles
     static SymbolSet union(SymbolSet e1, SymbolSet e2)

   }
   ```

   Les champs `val` et `next` sont exposés pour nous permettre d'itérer facilement sur les ensembles.

   (b) Lors de la construction de la pyramide, nous avons en fait besoin de retrouver les ensembles de non-terminaux correspondant à un membre droit donné, ces membres droits sont de deux sortes possibles : un simple symbole (qui est toujours terminal) ou une paire de symboles (qui sont toujours non-terminaux). En faisant fi de la distinction entre terminaux et non-terminaux, nous avons donc besoin d'une classe `Pair` des paires de symboles.

   ```
   class Pair {
     char fst, snd ;

     Pair (char c1, char c2) { fst = c1 ; snd = c2 ; }
   }
   ```

La classe `Chomsky` proposera trois méthodes, deux méthodes pour retrouver tous les non-terminaux définis par un membre droit donné, et une méthode qui donne le symbole de départ.

```
class Chomsky {
  SymbolSet getLhs(char t)
  SymbolSet getLhs(Pair p)
  char getStartSymbol() ;
}
```

2. Bon nous voilà prêts. Commençons par traiter le cas de l'entrée vide et par déclarer notre pyramide.

```
static boolean parse(Chomsky g, String alpha) {
  int len = s.length() ;
  if (len == 0) return false ;
  SymbolSet [] [] m = new SymbolSet [len+1] [];
```

L'initialisation de `m` est une initialisation de tableau, dont les éléments sont des tableaux (valant ici initialement **null**). On notera que le tableau `m` comprend $\ell + 1$ éléments, ainsi `m[j][i]` sera $m_{j,i}$. La ligne d'indice zéro ne sera pas employée.
Remplissons la base, qui est la ligne d'indice 1.

```
  SymbolSet [] m1 = new SymbolSet[len] ;
  for (int i = 0 ; i < len ; i++) {
    m0[i] = g.getLhs(alpha.charAt(i)) ;
  }
  m[1] = m1 ;
```

C'est tout à fait facile, puisque la majeure partie du travail (regrouper les productions de membre droit identique) est faite par la méthode `getLhs` de la classe `Chomsky`.
Enfin construisons les lignes de la base vers de sommet.

```
for (int j = 2 ; j <= len ; j++) { // Pour toutes les longueurs j
  SymbolSet [] row = new SymbolSet [len-j+1] ;
  for (int i = 0 ; i <= len-j ; i++) {
  // Pour tous les indices i de sous-cha^^eenes de longueur j
    SymbolSet set = null ;
    for (int k = 1 ; k < j ; k++) {
      // Pour tous les d^^e9coupages de [i..i+j[ en [i..i+k[ et ...
      for (SymbolSet p = m[k][i] ; // [i..i+k[
           p != null ;
           p = p.next) {
        for(SymbolSet q = m[j-k][i+k] ; //[i+k..i+k+k'[, k+k' = j
            q != null ;
            q = q.next) {
          set =
            SymbolSet.union(g.getLhs(new Pair(p.val, q.val)), set) ;
        }
      }
    }
    row[i] = set ;
  }
  m[j] = row ;
}
```

Ce n'est pas si difficile que ça en a l'air. Une fois fixée une décomposition en deux de la sous-chaîne $\alpha[i \ldots i+j[$ dont la première composante est de longueur k (avec $1 \leq k \leq j-1$) on obtient les deux ensembles $E = m_{k,i}$ et $E' = m_{k',i+k}$ (avec $k + k' = j$). Il faut ensuite considérer toutes les paires (A, A') (avec $A \in E$ et $A' \in E'$) et accumuler les membres droits des productions de la forme $B \rightarrow AA'$. D'où les cinq boucles imbriquées...

Et on peut enfin savoir si α dérive du symbole de départ de `g`.

```
    return SymbolSet.mem(g.getStartSymbol(),m[len][0]) ;
  }
```

3. Les clefs des tables de hachage seront donc des charactères (classe `Character`) et nos paires (classe `Pair`).

```
class Pair {
  char fst, snd ;

  Pair (char c1, char c2) {
    fst = c1 ; snd = c2 ;
  }

  public boolean equals(Object o) {
    if (o == null) return false ;
    try {
      Pair p = (Pair)o ;
      return fst == p.fst && snd == p.snd ;
    } catch (ClassCastException e) {
      return false ;
    }
  }

  public int hashCode() {
    int f = fst ;
    int s = snd ;
    int h = (f << 16) | s ;
    return h ;
  }

  public String toString() {
    return "(" + fst + "," + snd + ")" ;
  }
}
```

On remarque que `equals` doit traiter le cas où l'objet `o` n'est pas une paire, car nous avons deux classes de clefs différentes pour une même table. La méthode `hashCode` est naïve et exploite que les caractères sont des entiers sur 16 bits (tandis que les **int** sont sur 32 bits).

Et voici la classe `Chomsky`.

```
import java.util.* ;

class Chomsky {

  private char start ;
  private Hashtable t_lhs ;

  char getStartSymbol() { return start ; }
```

```
  static private void error(String msg) {
    System.err.println("Chomsky:␣" + msg) ;
    System.exit(2) ;
  }

  private void add(Object key, char c) {
    SymbolSet p = (SymbolSet)(t_lhs.get(key)) ;
    t_lhs.put(key, SymbolSet.add (c, p)) ;
  }


  Chomsky(char c, Rules rs) {
    start = c ;
    t_lhs = new Hashtable () ;

    for (Rules p = rs ; p != null ; p = p.next) {
      char lhs = p.lhs ;
      List rhs = p.rhs ;
      if (rhs == null) error("r^^e8gle␣vide") ;
      if (rhs.next == null) {
        add(new Character(rhs.val), lhs) ;
      } else if (rhs.next.next == null) {
        add(new Pair(rhs.val, rhs.next.val), lhs) ;
      } else
        error("r^^e8gle␣trop␣longue") ;
    }
  }

  SymbolSet getLhs(Pair p) {
    return (SymbolSet)t_lhs.get(p) ;
  }

  SymbolSet getLhs(char t) {
    return (SymbolSet)t_lhs.get(new Character(t)) ;
  }
}
```

Exercice 4

1. La suite (N_i) est croissante au sens large (pour $\subseteq$) et bornée (par l'ensemble de tous les non-terminaux). Elle converge donc et sa limite est l'ensemble des non-terminaux nullables, selon l'introduction.
2. C'est un classique calcul par point fixe, on itère $\mathcal{N}$ jusqu'à stabilisation. Le booléen renvoyé par la méthode `add` des des objets de la classe `Symbols` aide un peu.

```
  static boolean isNullable(List rhs, Symbols nullable) {
    for (List q = rhs ; q != null ; q = q.next) {
      if (!nullable.mem(q.val)) return false ;
    }
    return true ;
  }

  static Symbols getNullables(Rules rs) {
```

```
    Symbols r = new Symbols() ;
    boolean changed = true ;
    while (changed) {
      changed = false ;
      for (Rules p = rs ; p != null ; p = p.next) {
        if (isNullable(p.rhs, r)) {
          changed = r.add(p.lhs) || changed ;
        }
      }
    }
    return r ;
  }
```

3. C'est techniquement assez casse-pieds. Le plus simple me semble d'abord d'expanser les membres droits contenant des nullables, puis d'effectuer une passe de nettoyage pour enlever toutes les productions vides.

```
  static Rules removeEmpty(Rules rs) {
    Symbols nullables = getNullables(rs) ;

    Rules r = null ;
    for (Rules p = rs ; p != null ; p = p.next) {
      r = expandNullables(p.lhs, nullables, p.rhs, r) ;
    }

    Rules rr = null ;
    for (Rules p = r ; p != null ; p = p.next) {
      if (p.rhs != null)
        rr = new Rules (p.lhs, p.rhs, rr) ;
    }

    return rr ;
  }
```

On peut noter que la suppression des productions vides (dernière boucle) inverse la liste `r`, cela n'a pas d'importance ici.

Notons aussi que, par hypothèse, puisque le mot vide n'est pas engendré, le symbole de départ S n'est pas nullable. Si tel était le cas, on pourrait ajouter une production $S \rightarrow \epsilon$ tout à la fin.

Et voici `expandNullable`, qui expanse les membres droits.

```
  static Rules expandNullables(char lhs, Symbols nullables, List rhs, Rules k)
    if (rhs == null)
      return new Rules (lhs, null, k) ;
    else {
      Rules p = expandNullables(lhs, nullables, rhs.next, k) ;
      Rules r ;
      if (nullables.mem(rhs.val)) {
        r = p ;
      } else {
        r = k ;
      }
      for ( ; p != k ; p = p.next)
        r = new Rules (lhs, new List(rhs.val, p.rhs), r) ;
      return r ;
```

```
    }
  }
```

L'expansion est réalisée sur le principe récursif suivant.

- Pour $A \to \epsilon$ l'expansion est $A \to \epsilon$.
- Sinon, pour $A \to B\alpha$ où B est nullable. On expanse $A \to \alpha$ en $A \to \alpha_0$, …, $A \to \alpha_n$ et on renvoie les $2 \times n$ productions $A \to B\alpha_0$, …, $A \to B\alpha_n$, $A \to \alpha_0$, …, $A \to \alpha_n$. Si B n'est pas nullable, (en particulier si B est terminal), on se contente de renvoyer $A \to B\alpha_0$, …, $A \to B\alpha_n$.

On peut noter la condition d'arrêt de la dernière boucle, assez subtile, elle évite une concaténation (ajouter k à la fin de toutes les listes de productions ci-dessus). Le code est en fait encore un peu plus subtil, il évide les recopie des listes $A \to B\alpha_0$, …, $A \to B\alpha_n$

Et en bonus final (dans la version web), un programme de démonstration un peu prototypal.

Un complément intéressant serait, à partir de la pyramide, de retrouver une dérivation du mot analysé.