Université Laval
Faculté des sciences et de génie
Département d'informatique et de génie logiciel
IFT-3101

Danny Dubé
Automne 2018
Version : 18 octobre

Corrigé du travail pratique #1

Réponses

Note : les réponses aux différentes questions ne sont pas toujours uniques.

1. (a) Oui. Le fait de prendre un préfixe ne fait que laisser tomber un certain nombre des derniers caractères (ou peut-être pas), lesquels sont pris de la classe `[A-Za-z_0-9]`. En revanche, comme le préfixe est non-vide, on préserve le premier caractère, lequel est pris de la classe `[A-Za-z_]` et lequel restera à la première position dans le préfixe extrait.

   (b) Non. Par exemple, si on choisit l'identificateur `R2D2`, on peut extraire le suffixe non-vide `2D2`, qui n'est pas une chaîne conforme au motif des identificateurs.

2. (a) Pour générer ce langage, on porte attention à la longueur des suites de '`a`' consécutifs. Il faut limiter ces suites à trois '`a`' consécutifs. Le motif $\texttt{a}^?\texttt{a}^?\texttt{a}^?$ permet de générer n'importe quelle suite de '`a`' de longueur 0 à 3. Ensuite, il faut prendre soin de séparer les suites de '`a`' les unes des autres. Je fournis une définition régulière générant le langage.

$$\begin{aligned} s &\rightarrow \texttt{a}^?\texttt{a}^?\texttt{a}^? \\ r &\rightarrow s\,(\texttt{[bc]}\,s)^* \end{aligned}$$

   (b) La grammaire que je fournis choisit tout d'abord l'une des trois inégalités, puis prend soin de générer une chaîne qui respecte cette inégalité. Il faut noter qu'il est possible que d'autres inégalités soient "accidentellement" respectées par la même occasion mais ça n'empêche pas l'inégalité choisie d'être respectée.

$$\begin{array}{lcll} S & \rightarrow & D\,C \mid E \mid A\,F & // \text{ Choix de l'inégalité} \\ A & \rightarrow & \texttt{a}\,A \mid \epsilon & // \text{ Génère } \{\texttt{a}\}^* \\ B & \rightarrow & \texttt{b}\,B \mid \epsilon & // \text{ Génère } \{\texttt{b}\}^* \\ C & \rightarrow & \texttt{c}\,C \mid \epsilon & // \text{ Génère } \{\texttt{c}\}^* \\ D & \rightarrow & \texttt{a}\,D\,\texttt{b} \mid B & // \text{ Respecte } i \leq j \\ E & \rightarrow & \texttt{a}\,E\,\texttt{c} \mid B\,C & // \text{ Respecte } i \leq k \\ F & \rightarrow & \texttt{b}\,F\,\texttt{c} \mid C & // \text{ Respecte } j \leq k \end{array}$$

(c) Le truc consistait à réaliser que la condition utilisée dans l'ensemble est toujours vraie. Si on suppose que la condition est fausse, alors : on aurait $i > j$ et $j > k$ et $k > i$; ceci impliquerait $i > i$; ceci mènerait à une contradiction. Donc, on peut conclure qu'il n'y a pas de contrainte sur les exposants i, j et k.

$$\mathtt{a}^* \, \mathtt{b}^* \, \mathtt{c}^*$$

(d) L'ensemble contient toutes les chaînes sur $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. On peut s'en convaincre facilement en appliquant la recette suivante. Prenons une chaîne quelconque $v \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}^*$. Soit $l = |v|$ la longueur de v. Alors, on choisit $w = (\mathtt{abcd})^l$. Pour extraire la sous-séquence v de w, il suffit d'abandonner trois symboles sur quatre dans chaque copie de `abcd`. Dans la i-ème copie de `abcd` dans w, on ne conserve que le i-ème symbole de v.

$$(\mathtt{a} \mid \mathtt{b} \mid \mathtt{c} \mid \mathtt{d})^*$$

(e) Soit L le langage à générer. La difficulté ici vient du fait que l'ordre des symboles dans les mots de L n'est pas contraint. Donc, il faut arriver à compter correctement les symboles tout en accommodant la variété de combinaisons possibles. Pour la plupart des gens, il est assez aisé de concevoir une grammaire qui construit des chaînes correctes mais il est plus difficile de s'assurer que celle-ci construise *toutes* les chaînes correctes.

Premièrement, pour nous aider à raisonner à propos de nos chaînes, adoptons la notation suivante. Pour avoir le nombre de '`a`' dans w, on écrit $|w|_{\mathtt{a}}$; pour le nombre de '`b`', on écrit $|w|_{\mathtt{b}}$. Ainsi, $L = \{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid 2\,|w|_{\mathtt{b}} \leq |w|_{\mathtt{a}} \leq 3\,|w|_{\mathtt{b}}\}$.

Deuxièmement, il faut noter qu'il n'est pas suffisant de construire des chaînes en concaténant des segments comme `aba`, `baaa`, `aaba`, `aab` ; i.e. en concaténant des éléments de $\{x \in \{\mathtt{a}, \mathtt{b}\}^* \mid 3 \leq |x| \leq 4 \text{ et } |x|_{\mathtt{b}} = 1\}$. En effet, en se contentant de concaténer de tels segments on n'arrive pas à générer une chaîne comme $\mathtt{bbaaaaaaab} \in L$.

Troisièmement, partitionnons L en deux sous-ensembles :

$$L_2 = \{w \in L \mid 2\,|w|_{\mathtt{b}} = |w|_{\mathtt{a}}\} \quad \text{et} \quad L_+ = \{w \in L \mid 2\,|w|_{\mathtt{b}} < |w|_{\mathtt{a}} \leq 3\,|w|_{\mathtt{b}}\}.$$

Bien entendu, on a $L_2 \cup L_+ = L$ et $L_2 \cap L_+ = \emptyset$. Observons que la chaîne vide ϵ fait partie de L ; plus précisément, $\epsilon \in L_2$. On va montrer qu'une chaîne w telle que $\epsilon \neq w \in L_2$ peut être obtenue à partir de $v \in L_2$, dans laquelle on insère deux '`a`' et un '`b`' à des positions appropriées. On va aussi montrer qu'une chaîne $w \in L_+$ peut être obtenue à partir de $v \in L$, dans laquelle on insère trois '`a`' et un '`b`' à des positions appropriées.

Quatrièmement, nous proposons la grammaire G suivante pour générer L. Les paragraphes suivants ne servent qu'à justifier que la grammaire génère L au com-

plet.

$$
\begin{array}{rcl}
S & \to & \epsilon \\
 & | & S\,\mathtt{a}\,S\,\mathtt{a}\,S\,\mathtt{b}\,S \\
 & | & S\,\mathtt{a}\,S\,\mathtt{b}\,S\,\mathtt{a}\,S \\
 & | & S\,\mathtt{b}\,S\,\mathtt{a}\,S\,\mathtt{a}\,S \\
 & | & S\,\mathtt{a}\,S\,\mathtt{a}\,S\,\mathtt{a}\,S\,\mathtt{b}\,S \\
 & | & S\,\mathtt{a}\,S\,\mathtt{a}\,S\,\mathtt{b}\,S\,\mathtt{a}\,S \\
 & | & S\,\mathtt{a}\,S\,\mathtt{b}\,S\,\mathtt{a}\,S\,\mathtt{a}\,S \\
 & | & S\,\mathtt{b}\,S\,\mathtt{a}\,S\,\mathtt{a}\,S\,\mathtt{a}\,S
\end{array}
$$

Une propriété fort utile de G est que, à n'importe quelle position à l'intérieur d'un mot de $L(G)$, on peut décider d'insérer un autre mot de $L(G)$. Plus formellement, soient $w, x \in L(G)$; i.e. $S \Rightarrow^* w$ et $S \Rightarrow^* x$. Subdivisons w en un préfixe u et un suffixe v, i.e. $u\,v = w$, afin de désigner l'endroit où on souhaite insérer x. Notez qu'il existe un arbre t_w de dérivation pour w et un autre arbre t_x de dérivation pour x. Recherchons une feuille dans t_w telle que, à sa gauche, on retrouve les feuilles qui épellent u et, à sa droite, on retrouve les feuilles qui épellent v. La grammaire est conçue de telle façon que les feuilles des arbres de dérivation alternent entre une ou plusieurs apparitions de ϵ et un symbole. Ainsi, il existe bel et bien une feuille ϵ qui marque la séparation entre u et v. Cette feuille ϵ est l'enfant qui a crû grâce à la production $S \to \epsilon$. Il suffit alors de remplacer le mini-arbre correspondant à $S \to \epsilon$ par l'arbre t_x et on obtient alors un arbre de dérivation qui démontre que $u\,x\,v \in L(G)$.

Cinquièmement, nous montrons que la grammaire peut générer toutes les chaînes non-vides de L_2. Soit $w \in L_2 - \{\epsilon\}$. Considérons toutes les sous-chaînes de longueur 3 qu'il est possible d'extraire de w ;[1] soit $\Sigma_3 = \{y \mid xyz = w \text{ et } |y| = 3\}$. Naturellement, on a $|y|_{\mathtt{b}} \in \{0, 1, 2, 3\}$, pour tout $y \in \Sigma_3$. Nous prétendons qu'il existe une sous-chaîne $y_1 \in \Sigma_3$ telle que $|y_1|_{\mathtt{b}} = 1$. Notons que, s'il existe $y_i \in \Sigma_3$ telle que $|y_i|_{\mathtt{b}} = i$ et $y_j \in \Sigma_3$ telle que $|y_j|_{\mathtt{b}} = j$, où $i < j$, alors il existe $y_k \in \Sigma_3$ telle que $|y_k|_{\mathtt{b}} = k$, pour tout k tel que $i \leq k \leq j$. Pour s'en convaincre, on n'a qu'à penser à deux sous-chaînes $y, y' \in \Sigma_4$ extraites à deux positions voisines dans w ; alors, $|y|_{\mathtt{b}}$ et $|y'|_{\mathtt{b}}$ ne peuvent différer que d'une unité au maximum. Donc, si on suppose que y_1 n'existe pas, alors on aurait $|y|_{\mathtt{b}} = 0$, pour tout $y \in \Sigma_3$ ou on aurait $|y|_{\mathtt{b}} \in \{2, 3\}$, pour tout $y \in \Sigma_3$. Dans le premier cas, ça impliquerait qu'il y aurait trop peu de '$\mathtt{b}$' dans w pour avoir $w \in L_2$. Dans le second cas, ça impliquerait qu'il y aurait trop de '$\mathtt{b}$' dans w pour avoir $w \in L_2$. Donc, il existe $y_1 \in \Sigma_3$ et on peut écrire w ainsi : $w = uy_1v$. Un calcul simple nous permettrait de constater que la chaîne restante uv contiendrait un nombre de '$\mathtt{a}$' et un nombre de '$\mathtt{b}$' tels que $uv \in L_2$. Comme $|uv| < |w|$, on peut utiliser l'hypothèse d'induction pour conclure que $uv \in L(G)$. Enfin, comme $y_1 \in L(G)$, on peut insérer y_1 dans uv et conclure que $w \in L(G)$.

Sixièmement, nous montrons que la grammaire peut générer toutes les chaînes de L_+. Nous procédons d'une manière similaire au cas des chaînes non-vides de L_2.

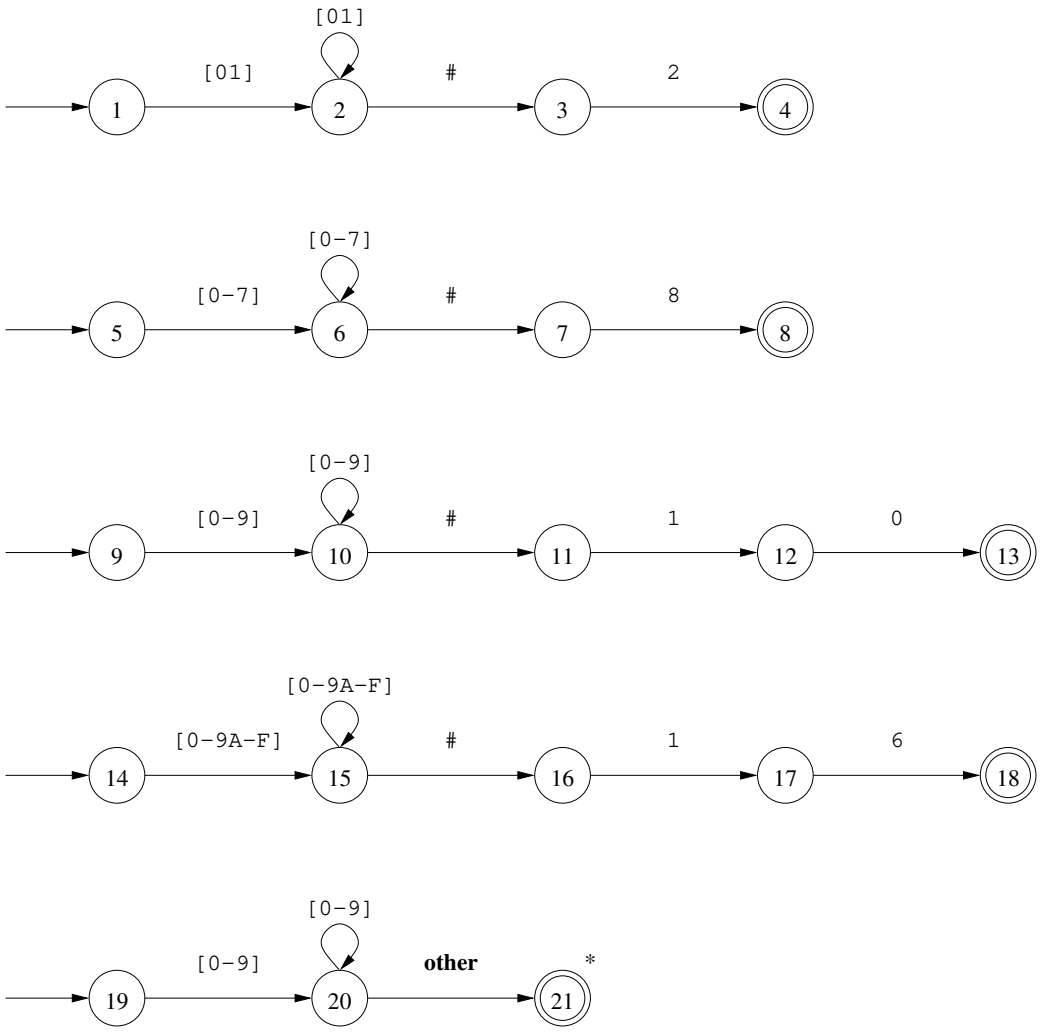
1. Il est à noter qu'il existe au moins une telle sous-chaîne. En effet, toute chaîne dans L_2 a une longueur qui est un multiple de 3.

Soit $w \in L_+$. Souvenons-nous que $w \neq \epsilon$. Considérons toutes les sous-chaînes de longueur 4 qu'il est possible d'extraire de w ;[2] soit $\Sigma_4 = \{y \mid xyz = w \text{ et } |y| = 4\}$. On a $|y|_{\texttt{b}} \in \{0, 1, 2, 3, 4\}$, pour tout $y \in \Sigma_4$. Il doit exister une sous-chaîne $y_1 \in \Sigma_4$ telle que $|y_1|_{\texttt{b}} = 1$ à cause du même raisonnement sur les sous-chaînes voisines que celui utilisé pour L_2. Donc, on peut écrire w comme étant uy_1v. Un calcul simple nous permettrait de constater que la chaîne restante uv contiendrait un nombre de '`a`' et un nombre de '`b`' tels que $uv \in L$ (ce n'est pas nécessairement le cas qu'on a $uv \in L_+$). Comme $|uv| < |w|$, on peut conclure que $uv \in L(G)$. Comme $y_1 \in L(G)$, on peut aussi conclure que $w \in L(G)$.

Notez que j'ai fourni cette réponse longue et détaillée à des fins pédagogiques. Je ne m'attends pas à ce que soyez allés jusqu'à un tel niveau de détails, dans votre travail.

2. Il est à noter qu'il existe au moins une telle sous-chaîne. En effet, les chaînes les plus courtes dans L sont ϵ, qui est dans L_2, puis `aab`, `aba` et `baa`, qui sont aussi dans L_2. Donc, toutes les chaînes dans L_+ ont une longueur d'au moins 4.

3. Pour éviter les complications, nous choisissons de séparer le traitement sur cinq automates. Les quatre premiers se chargent des constantes dans les différentes bases avec un suffixe. Le cinquième se charge des constantes en décimal sans suffixe. Notez que le cinquième automate doit consommer des caractères jusqu'à temps qu'on confirme que la constante est terminée en tombant sur un caractère qui n'est pas un chiffre. Ce dernier caractère est retournée dans l'entrée grâce à une étoile.



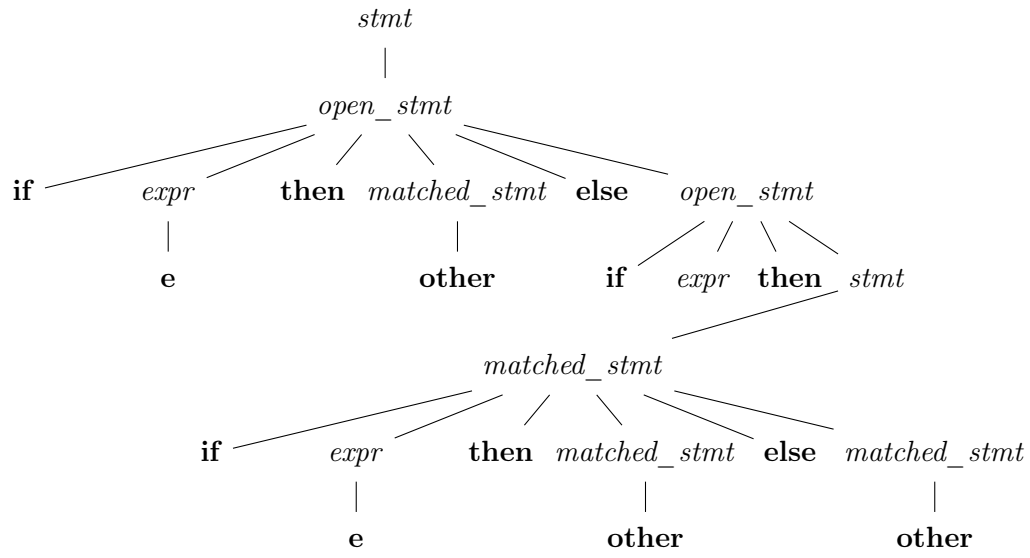
4. Voici le pseudo-code.

```
while (true)
{
  switch(state)
  {
    ...
    case 1:
      c := buffer[forward];
      forward ++;
      if (c == ' ')
        state := 2;
      else if (is_digit(c))
        state := 4;
      else
        state := fail();
      break;
    case 2:
      c := buffer[forward];
      forward ++;
      if (c == ' ')
        state := 2;
      else
        state := 3;
      break;
    case 3:
      forward --;
      jeton := JETON_WS;
      attribut := ...;
      token_beginning := forward;
      return jeton;
    case 4:
      c := buffer[forward];
      forward ++;
      if (is_digit(c))
        state := 4;
      else if (c == '/')
        state := 5;
      else
        state := fail();
      break;
    case 5:
      c := buffer[forward];
      forward ++;
      if (is_digit(c))
        state := 6;
      else
        state := fail();
      break;
    case 6:
      c := buffer[forward];
      forward ++;
      if (is_digit(c))
        state := 6;
      else
        state := 7;
      break;
    case 7:
      forward --;
      jeton := JETON_RATIONAL;
      attribut := getlexeme();
      token_beginning := forward;
      return jeton;
    case 8:
      c := buffer[forward];
      forward ++;
      if (is_digit(c))
        state := 9;
      else
        state := fail();
      break;
    case 9:
      c := buffer[forward];
      forward ++;
      if (is_digit(c))
        state := 9;
      else
        state := 10;
      break;
    case 10:
      forward --;
      jeton := JETON_INTEGER;
      attribut := getlexeme();
      token_beginning := forward;
      return jeton;
    ...
  }
}
```

5. Dérivation à gauche d'abord :

$\Rightarrow$ *stmt*
$\Rightarrow$ *open_stmt*
$\Rightarrow$ **if** *expr* **then** *matched_stmt* **else** *open_stmt*
$\Rightarrow$ **if e then** *matched_stmt* **else** *open_stmt*
$\Rightarrow$ **if e then other else** *open_stmt*
$\Rightarrow$ **if e then other else if** *expr* **then** *stmt*
$\Rightarrow$ **if e then other else if e then** *stmt*
$\Rightarrow$ **if e then other else if e then** *matched_stmt*
$\Rightarrow$ **if e then other else if e then**
if *expr* **then** *matched_stmt* **else** *matched_stmt*
$\Rightarrow$ **if e then other else if e then**
if e then *matched_stmt* **else** *matched_stmt*
$\Rightarrow$ **if e then other else if e then if e then other else** *matched_stmt*
$\Rightarrow$ **if e then other else if e then if e then other else other**

Arbre de dérivation :



6. La chaîne $w = \texttt{aaabbbaaa}$ fait partie du langage mais nous prétendons qu'elle ne peut pas être générée par la grammaire donnée. Pour s'en convaincre, nous analysons quelle pourrait être la première production utilisée pour remplacer S. Premièrement, on voit que w a exactement deux fois plus de '`a`' que de '`b`'. De plus, w n'est pas vide. Donc, on peut se restreindre aux productions données dans les rangées 2 à 4. Deuxièmement, parmi ces productions, on voit qu'on ne peut en choisir une qui se trouve dans la première colonne, à cause que ces productions forcent l'inclusion d'un '`b`' dans l'une des trois premières positions. Pour des raisons similaires, on ne peut en choisir une qui se trouve dans la dernière colonne. Nous devons donc choisir une production qui se trouve dans le groupe des 6 productions en haut, au centre. Troisièmement, on voit que w commence et finit avec des '`a`', donc on doit exclure la première et la dernière rangée du groupe de 6. Finalement, parmi les deux productions qui restent, on voit qu'elles forceraient d'insérer un '`b`' soit à la deuxième, soit à l'avant-dernière position de w. On conclut donc que la grammaire ne peut pas générer w.

7. Oui, la grammaire est ambiguë. La clé se trouve dans le fait que les chaînes qui ont une longueur qui est à la fois multiple de 3 et de 5 peuvent être générées de deux façons. Donc, nous nous intéressons aux chaînes qui ont une longueur qui est multiple de 15. Bien entendu, la grammaire a été conçue pour que ϵ, la première chaîne de cette sorte, soit générée sans ambiguïté. Si on se tourne plutôt vers la prochaine chaîne, soit $\texttt{a}^{15}$, on peut exhiber deux arbres de dérivation différents. J'omets de dessiner les deux arbres ici, étant donnée leur simplicité.

8. Le langage permet un déséquilibre, jusqu'à concurrence d'un rapport de 2 entre le nombre de '`a`' et le nombre de '`b`'. Le membre droit $\texttt{aa}S\texttt{b}$ sert à surreprésenter les '`a`' et le membre droit $\texttt{a}S\texttt{bb}$, l'inverse. Une des sources d'ambiguïté vient du fait qu'on peut favoriser les '`a`' pendant quelques substitutions, puis favoriser les '`b`' et vice-versa. Favoriser les deux symboles à des moments différents revient à utiliser le membre droit $\texttt{a}S\texttt{b}$ un certain nombre de fois. Ceci constitue une autre source d'ambiguïté. Enfin, même si on ne choisit de favoriser qu'un seul des deux symboles, par exemple '`a`', on peut quand même alterner arbitrairement entre l'usage du membres droits $\texttt{aa}S\texttt{b}$ et $\texttt{a}S\texttt{b}$, ce qui constitue encore une autre source d'ambiguïté.

   Conscients de toutes ces sources d'ambiguïté, nous concevons une autre grammaire, laquelle est sans ambiguïté. Celle-ci décide dans un premier temps si le mot généré va surreprésenter les '`a`' ou les '`b`' ou bien s'il sera équilibré. Lorsque le mot doit être déséquilibré, on force l'utilisation du membre droit déséquilibré avant de permettre l'usage du membre droit équilibré.

$$\begin{aligned} S &\rightarrow \texttt{aa}A\texttt{b} \mid \texttt{a}B\texttt{bb} \mid C \\ A &\rightarrow \texttt{aa}A\texttt{b} \mid C \\ B &\rightarrow \texttt{a}B\texttt{bb} \mid C \\ C &\rightarrow \texttt{a}S\texttt{b} \mid \epsilon \end{aligned}$$

9. Nous commençons avec la grammaire originale.

$$\begin{aligned} E &\rightarrow E \oplus T \mid F \dagger E \mid T \\ T &\rightarrow T\surd \mid E \ominus F \mid F \\ F &\rightarrow \textbf{cte} \mid \textbf{var} \mid (\, E \,) \end{aligned}$$

Pour $i = 1$ **:** On nettoie les E-productions.

— La boucle sur j est vide. (C'est-à-dire qu'il n'y a aucun non-terminal placé *avant* E dans l'ordre des non-terminaux.)

— Il faut éliminer la récursion à gauche immédiate ; c'est-à-dire, toute production de la forme $E \rightarrow E\alpha$. Il y en a une. On obtient la grammaire :

$$\begin{aligned} E &\rightarrow F \dagger E\, E' \mid T\, E' \\ E' &\rightarrow \oplus\, T\, E' \mid \epsilon \\ T &\rightarrow T\surd \mid E \ominus F \mid F \\ F &\rightarrow \textbf{cte} \mid \textbf{var} \mid (\, E \,) \end{aligned}$$

Pour $i = 2$ **:** On nettoie les T-productions.

— La boucle sur j se résume à $j = 1$. Il faut éliminer toute production de la forme $T \rightarrow E\alpha$. Il y en a une. On obtient la grammaire :

$$\begin{aligned} E &\rightarrow F \dagger E\, E' \mid T\, E' \\ E' &\rightarrow \oplus\, T\, E' \mid \epsilon \\ T &\rightarrow T\surd \mid F \dagger E\, E' \ominus F \mid T\, E' \ominus F \mid F \\ F &\rightarrow \textbf{cte} \mid \textbf{var} \mid (\, E \,) \end{aligned}$$

— Il faut éliminer la récursion à gauche immédiate. Il y a deux productions de la forme $T \rightarrow T\alpha$. On obtient la grammaire :

$$\begin{aligned} E &\rightarrow F \dagger E\, E' \mid T\, E' \\ E' &\rightarrow \oplus\, T\, E' \mid \epsilon \\ T &\rightarrow F \dagger E\, E' \ominus F\, T' \mid F\, T' \\ T' &\rightarrow \surd T' \mid E' \ominus F\, T' \mid \epsilon \\ F &\rightarrow \textbf{cte} \mid \textbf{var} \mid (\, E \,) \end{aligned}$$

Pour $i = 3$ **:** On nettoie les F-productions.

— Pour $j = 1$, il faut éliminer toute production de la forme $F \rightarrow E\alpha$. Rien à faire.

— Pour $j = 2$, il faut éliminer toute production de la forme $F \rightarrow T\alpha$. Rien à faire.

— Il n'y a pas de récursion à gauche immédiate chez F. Rien à faire.

La dernière grammaire donnée est la grammaire souhaitée.

10. Voici le pseudo-code.

```
void stmt()
{
  if (peekToken().type == "while")
    {
      readToken("while");
      readToken("expr");
      readToken("do");
      list();
      readToken("end");
    }
  else if (peekToken().type == "if")
    {
      readToken("if");
      readToken("expr");
      readToken("then");
      list();
      tail();
    }
  else
    {
      readToken("id");
      readToken(":=");
      readToken("expr");
      readToken(";");
    }
}
```

```
void tail()
{
  if (peekToken().type == "else")
    {
      readToken("else");
      list();
      readToken("end");
    }
  else
    {
      readToken("end");
    }
}

void list()
{
  if ((peekToken().type == "while") ||
      (peekToken().type == "if") ||
      (peekToken().type == "id"))
    {
      stmt();
      list();
    }
  else
    {
    }
}
```

11.

Pile	Entrée	Sortie
S \$	abbaaabcbaaabba\$	$S \to \mathtt{a}\,S\,\mathtt{a}$
a S a \$	abbaaabcbaaabba\$	(Consomme a)
S a \$	bbaaabcbaaabba\$	$S \to \mathtt{b}\,S\,\mathtt{b}$
b S b a \$	bbaaabcbaaabba\$	(Consomme b)
S b a \$	baaabcbaaabba\$	$S \to \mathtt{b}\,S\,\mathtt{b}$
b S b b a \$	baaabcbaaabba\$	(Consomme b)
S b b a \$	aaabcbaaabba\$	$S \to \mathtt{a}\,S\,\mathtt{a}$
a S a b b a \$	aaabcbaaabba\$	(Consomme a)
S a b b a \$	aabcbaaabba\$	$S \to \mathtt{a}\,S\,\mathtt{a}$
a S a a b b a \$	aabcbaaabba\$	(Consomme a)
S a a b b a \$	abcbaaabba\$	$S \to \mathtt{a}\,S\,\mathtt{a}$
a S a a a b b a \$	abcbaaabba\$	(Consomme a)
S a a a b b a \$	bcbaaabba\$	$S \to \mathtt{b}\,S\,\mathtt{b}$
b S b a a a b b a \$	bcbaaabba\$	(Consomme b)
S b a a a b b a \$	cbaaabba\$	$S \to \mathtt{c}$
c b a a a b b a \$	cbaaabba\$	(Consomme c)
b a a a b b a \$	baaabba\$	(Consomme b)
a a a b b a \$	aaabba\$	(Consomme a)
a a b b a \$	aabba\$	(Consomme a)
a b b a \$	abba\$	(Consomme a)
b b a \$	bba\$	(Consomme b)
b a \$	ba\$	(Consomme b)
a \$	a\$	(Consomme a)
\$	\$	(Succès)

12. (a) Pour nous simplifier la vie, je commence par déterminer quels ensembles FIRST contiennent ϵ. Clairement, $\epsilon \notin$ FIRST(*stmt*) et $\epsilon \notin$ FIRST(*tail*) car tous les membres droits concernés contiennent des terminaux. À l'inverse, $\epsilon \in$ FIRST(*list*) car *list* $\Rightarrow \epsilon$.

Maintenant, posons les contraintes sur les ensembles FIRST qui sont nécessaires au calcul des ensembles FIRST.

$$\begin{aligned}
\text{FIRST}(\textit{stmt}) &= \text{FIRST}(\textbf{while expr do } \textit{list} \textbf{ end}) \cup \\
&\quad \text{FIRST}(\textbf{if expr then } \textit{list } \textit{tail}) \cup \text{FIRST}(\textbf{id } := \textbf{ expr ;}) \\
&= \{\textbf{while}\} \cup \{\textbf{if}\} \cup \{\textbf{id}\} \\
&= \{\textbf{while}, \textbf{if}, \textbf{id}\} \\
\text{FIRST}(\textit{tail}) &= \text{FIRST}(\textbf{else } \textit{list } \textbf{end}) \cup \text{FIRST}(\textbf{end}) \\
&= \{\textbf{else}\} \cup \{\textbf{end}\} \\
&= \{\textbf{else}, \textbf{end}\} \\
\text{FIRST}(\textit{list}) &= \text{FIRST}(\textit{stmt } \textit{list}) \cup \text{FIRST}(\epsilon) \\
&= \text{FIRST}(\textit{stmt}) \cup \{\epsilon\} \\
&= \{\textbf{while}, \textbf{if}, \textbf{id}, \epsilon\}
\end{aligned}$$

La plus petite (et la seule) solution à ces contraintes nous donne les ensembles suivants.

	FIRST
stmt	{**while**, **if**, **id**}
tail	{**else**, **end**}
list	{**while**, **if**, **id**, ϵ}

(b) Posons les contraintes sur les ensembles FOLLOW.

$$
\begin{array}{rcl}
\text{FOLLOW}(\textit{stmt}) & \supseteq & \{\$\} \\
 & & \text{car symbole de départ} \\
\text{FOLLOW}(\textit{list}) & \supseteq & \text{FIRST}(\textbf{end}) - \{\epsilon\} \;=\; \{\textbf{end}\} \\
 & & \text{car apparition dans } 1^{\text{ere}} \textit{ stmt}\text{-production} \\
\text{FOLLOW}(\textit{list}) & \supseteq & \text{FIRST}(\textit{tail}) - \{\epsilon\} \;=\; \{\textbf{else}, \textbf{end}\} \\
 & & \text{car apparition dans } 2^{\text{e}} \textit{ stmt}\text{-production} \\
\text{FOLLOW}(\textit{tail}) & \supseteq & \text{FIRST}(\epsilon) - \{\epsilon\} \;=\; \{\} \\
 & & \text{car apparition dans } 2^{\text{e}} \textit{ stmt}\text{-production} \\
\text{FOLLOW}(\textit{list}) & \supseteq & \text{FIRST}(\textbf{end}) - \{\epsilon\} \;=\; \{\textbf{end}\} \\
 & & \text{car apparition dans } \textit{tail}\text{-production} \\
\text{FOLLOW}(\textit{stmt}) & \supseteq & \text{FIRST}(\textit{list}) - \{\epsilon\} \;=\; \{\textbf{while}, \textbf{if}, \textbf{id}\} \\
 & & \text{car apparition dans } \textit{list}\text{-production} \\
\text{FOLLOW}(\textit{list}) & \supseteq & \text{FIRST}(\epsilon) - \{\epsilon\} \;=\; \{\} \\
 & & \text{car apparition dans } \textit{list}\text{-production} \\
\text{FOLLOW}(\textit{tail}) & \supseteq & \text{FOLLOW}(\textit{stmt}) \\
 & & \text{car apparition en fin de } 2^{\text{e}} \textit{ stmt}\text{-production} \\
\text{FOLLOW}(\textit{stmt}) & \supseteq & \text{FOLLOW}(\textit{list}) \\
 & & \text{car apparition en fin de } \textit{list}\text{-production} \\
\text{FOLLOW}(\textit{list}) & \supseteq & \text{FOLLOW}(\textit{list}) \\
 & & \text{car apparition en fin de } \textit{list}\text{-production}
\end{array}
$$

La plus petite solution à ces contraintes nous donne les ensembles suivants.

	FOLLOW
stmt	{**while**, **if**, **id**, **else**, **end**, \$}
tail	{**while**, **if**, **id**, **else**, **end**, \$}
list	{**else**, **end**}

13. La production *stmt* → *matched_stmt* est insérée uniquement dans les colonnes correspondant à FIRST(*matched_stmt*), car $\epsilon \notin$ FIRST(*matched_stmt*).

    La production *stmt* → *open_stmt* est insérée uniquement dans les colonnes correspondant à FIRST(*open_stmt*), car $\epsilon \notin$ FIRST(*open_stmt*).

    La production *matched_stmt* → **if** *expr* **then** *matched_stmt* **else** *matched_stmt* est insérée uniquement dans la colonne correspondant à FIRST(**if** ...) = {**if**}, car $\epsilon \notin$ FIRST(**if** ...).

    La production *matched_stmt* → **other** est insérée uniquement dans la colonne correspondant à FIRST(**other**) = {**other**}, car $\epsilon \notin$ FIRST(**other**).

    La production *open_stmt* → **if** *expr* **then** *stmt* est insérée uniquement dans la colonne correspondant à FIRST(**if** ...) = {**if**}, car $\epsilon \notin$ FIRST(**if** ...).

    La production *open_stmt* → **if** *expr* **then** *matched_stmt* **else** *open_stmt* est insérée uniquement dans la colonne correspondant à FIRST(**if** ...) = {**if**}, car $\epsilon \notin$ FIRST(**if** ...).

Non-terminal	Symbole d'entrée				
	if	**then**	**else**	**other**	$
stmt	*stmt* → *matched_stmt* *stmt* → *open_stmt*			*stmt* → *matched_stmt*	
matched_stmt	*matched_stmt* → **if** ...			*matched_stmt* → **other**	
open_stmt	*open_stmt* → ... *stmt* *open_stmt* → ... *open_stmt*				