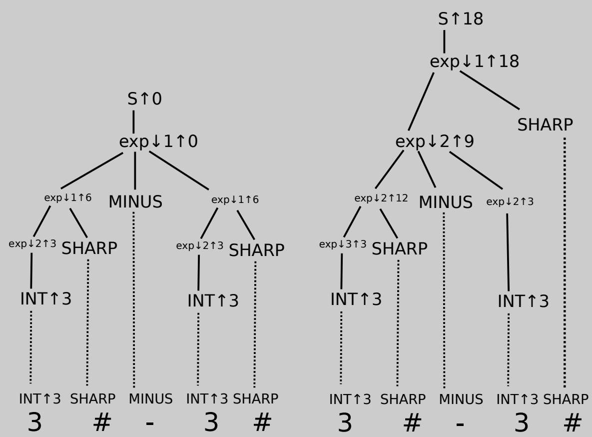
TD d'analyse syntaxique : feuille d'exercices

Exercice 1. On considère les terminaux `NAT`, `MINUS` qui correspondent aux langages de lexèmes du TD1, plus les terminaux `SHARP`, `OPAR` et `CPAR` qui correspondent respectivement aux singletons {`#`}, {`(`} et {`)`}. On considère aussi la BNF attribuée suivante avec les non-terminaux $\mathbf{S}\uparrow\mathbb{Z}$ et $\mathbf{exp}\downarrow\mathbb{N}\uparrow\mathbb{Z}$:

$$
\begin{array}{rrcll}
(1) & \mathbf{S}\uparrow n & ::= & \mathbf{exp}\downarrow 1\uparrow n & \\
(2) & \mathbf{exp}\downarrow p\uparrow n & ::= & \mathtt{NAT}\uparrow n & \\
(3) & & | & \mathtt{OPAR}\ \mathbf{exp}\downarrow p\uparrow n\ \mathtt{CPAR} & \\
(4) & & | & \mathbf{exp}\downarrow(p+1)\uparrow n_0\ \mathtt{SHARP} & n := n_0 \times 2^p \\
(5) & & | & \mathbf{exp}\downarrow p\uparrow n_1\ \mathtt{MINUS}\ \mathbf{exp}\downarrow p\uparrow n_2 & n := n_1 - n_2
\end{array}
$$

▷ **Question 1.** Cette BNF étant ambiguë, donner deux arbres d'analyse du mot "`3 # - 3 #`" pour lesquels le calcul d'attributs donne deux résultats différents. Décorer ces 2 arbres d'analyse avec leur calcul d'attributs.

Correction
Exemple : on a des calculs d'attributs similaire aux mots
"`(3 #) - (3 #)`" $= 6 - 6 = 0$
"`((3 #) - 3)#` $= (12 - 3) \times 2 = 18$.



▷ **Question 2.** Calculer les directeurs LL(1) de chacune des règles, y compris la règle (1). La BNF est-elle LL(1) ?

Correction
En suivant le cours, on calcule d'abord $\mathcal{E}(\mathbf{exp}) = \emptyset$, puis $\mathrm{Prem}(\mathbf{exp}) = \{\mathtt{NAT}, \mathtt{OPAR}\} \cup \mathrm{Prem}(\mathbf{exp})$ d'où :

```
Dir(2) = { NAT }
Dir(3) = { OPAR }
Dir(1) = Dir(4) = Dir(5) = Prem(exp) = { NAT, OPAR }
```

BNF non-LL(1) (on le savait car elle est ambiguë).

Exercice 2. Calculer les directeurs LL(1) des BNFs suivantes. Sont-elles LL(1) ? ambiguës ?

1. **S** ::= a**X** **X** ::= **S**b | b**S** | ε
2. **S** ::= **XX** | ε **X** ::= b**S**
3. **S** ::= **Y**a**XY**c**Y** **X** ::= a | ε **Y** ::= b**S** | ε

Correction

1. Directeurs LL(1) :
`Dir(S -> aX) = { a }`
`Dir(X -> Sb) = { a }`
`Dir(X -> bS) = { b }`
`Dir(X ->` ε`) = Suiv(X)`
On trouve la +petite solution du système d'équations des suivants :
`Suiv(S) = { $,b }` $\cup$ `Suiv(X)`
`Suiv(X) = Suiv(S)`
En appliquant la propriété donnée au chapitre 4 du cours
la +petite solution de $X = (X \cap \alpha) \cup \beta$ *vérifie aussi* $X = \beta$
on obtient :
`Suiv(X) = Suiv(S) = { $, b }`
Elle est LL(1) ssi pour chacune des paires de règles suivantes, les directeurs sont disjoints.
`paire 1: Dir(X -> Sb) et  Dir(X -> bS)`
`paire 2: Dir(X -> Sb) et  Dir(X ->` ε`)`
`paire 3: Dir(X -> bS) et  Dir(X ->` ε`)`
La BNF n'est pas LL(1) sur la paire 3 (on parle de conflit LL(1)).
La BNF est-elle ambiguë ? Si elle admet un mot qui a deux arbres d'analyse, alors l'analyse LL(1) de ce mot est "bloquée" par ce conflit. Pour trouver un tel mot, on cherche des mots/arbres où l'analyse LL(1) "bloque" sur ce conflit, comme "`aab`" et "`aabab`"... Mais, même parmi ces mots, on n'en trouve aucun qui a deux arbres d'analyses...
En fait, cette BNF n'est pas LL(1), mais elle est LL(2) : une généralisation de LL(1) quand on lit *autant que possible* deux tokens d'avance pour sélectionner la règle, au lieu d'un seul. Ça ne fait pas partie des compétences requises pour l'examen, mais c'est utile de le voir une fois pour améliorer la compréhension du cours (et la "culture"). Ici, on définit formellement le directeur LL(2) d'une règle par
$$\mathrm{Dir}_2(A \to \alpha) = \{\ w \in V_T^*.\{\varepsilon, \$\} \mid |w| \leq 2 \wedge (w \in V_T^* \Rightarrow |w| = 2) \wedge P(w)\ \}$$
où $P(w)$ est la proposition suivante :
il existe w_1, w_2, w', tels que $S.\$ \Rightarrow^* w_1.A.w_2$ et $\alpha.w_2 \Rightarrow^* w.w'$
En énumérant les arbres d'analyse de cette BNF avec une hauteur suffisamment grande, on trouve ces directeurs LL(2) :
`Dir2(S -> aX) = { a$, aa, ab }`
`Dir2(X -> Sb) = { aa, ab }`
`Dir2(X -> bS) = { ba }`
`Dir2(X ->` ε`) = { $, b$, bb }`
Ici, comme les 3 règles de membre gauche X ont des directeurs LL(2) deux à deux disjoints, la BNF est LL(2). Et donc, elle est non ambiguë....
REMARQUE : il ne peut pas exister d'algorithme (donc de méthode générale) pour décider si une BNF est ambiguë ou pas. Cf. derniers TDs de TL2.
REMARQUE de XN : on peut « simplifier » la déf de LL(2) et généraliser à LL(k) en dérivant depuis $S.\$^k$:
$$\mathrm{Dir}_k(A \to \alpha) = \{\ w \in V_T^*.\{\$\}^* \mid |w| = k \wedge P_k(w)\ \}$$
où $P_k(w)$ est la proposition suivante :

il existe w_1, w_2, w', tels que $S.\$^k \Rightarrow^* w_1.A.w_2$ et $\alpha.w_2 \Rightarrow^* w.w'$

Sur l'exemple, la seule différence est dans

`Dir2(X -> ε) = { $$, b$, bb }`

2. Directeurs LL(1) :

```
Dir(S -> XX) = Prem(X) = { b }
Dir(S -> ε) = Suiv(S)
Dir(X -> bS) = { b }
```

On trouve la +petite solution du système d'équations des suivants :

```
Suiv(S) = { $ } ∪ Suiv(X)
Suiv(X) = Prem(X) ∪ Suiv(S) = { b } ∪ Suiv(S)
```

soit :

`Suiv(X) = Suiv(S) = { b, $ }`

LL(1) ssi les 2 directeurs suivants sont disjoints :

`Dir(S -> XX)` et `Dir(S -> ε)`

Ils ne le sont pas (`b` est dans les 2 ensembles), donc la BNF n'est pas LL(1).

En cherchant des mots/arbres dont l'analyse LL(1) "bloque" sur ce conflit, on trouve que le mot "`bbbb`" a deux arbres d'analyse. Donc la BNF est ambiguë.

REMARQUE : on peut démontrer que le langage reconnu est $\{b^{2n} | 0 \leq n\}$. On reconnaît donc ce langage régulier avec la BNF LL(1) suivante :

`S -> bbS | ε`

On verra en cours que tout langage régulier est reconnu par une BNF LL(1).

3. Directeurs LL(1) :

```
Dir(S -> YaXYcY) = { a, b }
Dir(X -> a) = { a }
Dir(X -> ε) = Suiv(X)
Dir(Y -> bS) = { b }
Dir(Y -> ε) = Suiv(Y)
```

On trouve la +petite solution du système d'équations des suivants :

```
Suiv(S) = { $ } ∪ Suiv(Y)
Suiv(X) = Prem(YcY) = { b, c }
Suiv(Y) = Prem(aXYcY) ∪ Prem(cY) ∪ Suiv(S) = { a, c } ∪ Suiv(S)
```

soit :

`Suiv(X) = { b, c }` et `Suiv(Y) = Suiv(S) = { a, c ,$ }`

LL(1) ssi pour chacune des paires de règles suivantes, les directeurs sont disjoints.

```
paire 1: Dir(X -> a) = { a } et Dir(X -> ε)
paire 2: Dir(Y -> bS) = { b } et Dir(Y -> ε)
```

On en déduit que la BNF est LL(1) et donc qu'elle n'est pas ambiguë.

Exercice 3. On considère la BNF attribuée suivante, sur le vocabulaire terminal $\{a, b, c\}$, avec des non-terminaux de profil $\mathbf{S}{\downarrow}\mathbb{N}{\uparrow}\mathbb{N}$, $\mathbf{X}{\downarrow}\mathbb{N}{\uparrow}\mathbb{N}$ et $\mathbf{Y}{\downarrow}\mathbb{N}{\uparrow}\mathbb{N}$:

(1) $\mathbf{S}{\downarrow}h{\uparrow}\max(r_1, r_2)$ ::= `a` $\mathbf{X}{\downarrow}h{\uparrow}r_1$ $\mathbf{Y}{\downarrow}h{\uparrow}r_2$

(2) $\mathbf{X}{\downarrow}h{\uparrow}r$ ::= $\mathbf{S}{\downarrow}h{+}1{\uparrow}r$ `b`		(4) $\mathbf{Y}{\downarrow}h{\uparrow}r$ ::= `c` $\mathbf{Y}{\downarrow}h{+}1{\uparrow}r$ `a`	
(3) | ε	$r := 3 \times h$	(5) | ε	$r := 2^h$

▷ **Question 1.** La BNF est-elle LL(1) ? Est-elle ambiguë ? Justifier.

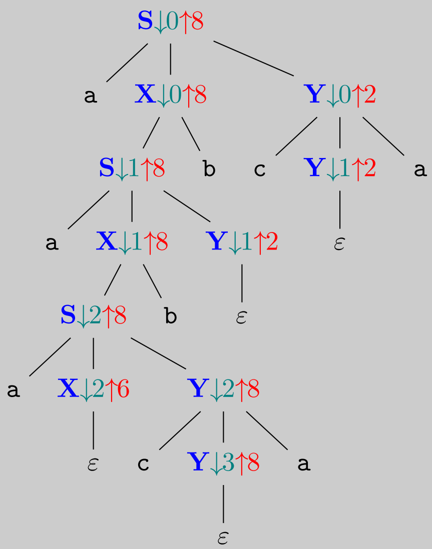
Correction

```
Suiv(S) = { b, $ }
Suiv(X) = Prem(Y) ∪ Suiv(S) = { b, c, $ }
Suiv(Y) = {a} ∪ Suiv(S) = { a, b, $ }
Dir(2) =  { a }
Dir(3) =  Suiv(X) = { b, c, $ }
Dir(4) =  { c }
Dir(5) =  Suiv(Y) = { a, b, $ }
```

La BNF est LL(1) car Dir(2) $\cap$ Dir(3) $= \emptyset$ et Dir(4) $\cap$ Dir(5) $= \emptyset$. elle est non-ambiguë.

▷ **Question 2.** Le mot "`aaacabbca`" est-il accepté par la BNF ? Si oui, en dessiner un arbre d'analyse avec la propagation d'attributs quand l'attribut hérité h à la racine vaut initialement 0.

Correction
Mot accepté avec l'arbre d'analyse suivant :



▷ **Question 3.** On suppose définie ci-dessous la machine à états des analyseurs syntaxiques LL(1) vue en cours et en TD, qui modifie une variable globale `current` contenant le token de pré-vision.

```python
TOKENS = tuple(range(4))
a, b, c, END = TOKENS    # END = token spécial de fin

def init_parser():    # initialise 'current' sur le premier token
def parse_token(t):   # vérifie 'current==t' et fait avancer 'current' sur le prochain token
```

Écrire le code PYTHON d'une fonction "`parse()`" qui implémente l'analyseur spécifié par la BNF attribuée ci-dessus : elle retourne un entier r correspondant à celui calculé par le système d'attributs lorsque h est initialisé à 0. On fera bien attention à rejeter un mot comme "`acaa`" qui n'est pas reconnu par la BNF. On pourra introduire des fonctions auxiliaires.

Correction

```python
def parse():
    init_parser()
    r = parse_S(0)
    parse_token(END)
    return r

def parse_S(h):
    parse_token(a)
    r1 = parse_X(h)
    r2 = parse_Y(h)
    return max(r1, r2)

def parse_X(h):
    if current == a:
       r = parse_S(h+1)
       parse_token(b)
    else:
       r = 3 * h
    return r

def parse_Y(h):
    if current == c:
       parse_token(c)
       r = parse_Y(h+1)
       parse_token(a)
    else:
       r = 2 ** h
    return r
```

Exercice 4. (Optionnel) Pour chacun des 2 langages suivants, donner une BNF LL(1)

$$L_1 = \{a^n b^m \mid 0 \leq n \leq m\} \quad \text{et} \quad L_2 = \{a^n b^m \mid 0 \leq m \leq n \leq m+2\}$$

Correction

1. Pour trouver la BNF, on peut décomposer L_1 sous forme $\{a^n b^p b^n\}$ ou $\{a^n b^n b^p\}$. Intuitivement, la première forme ne va pas donner une analyse LL(1), car on ne va pas savoir "quand sortir" de l'analyse de "b^p" pour revenir à "b^n" (on le vérifie facilement en écrivant une BNF qui correspond à cette idée et qui n'est pas LL(1)). La deuxième forme a l'air plus sympa : L_1 s'écrit $A.B$ où A et B sont deux langages LL(1). D'où la BNF :
   `S -> AB` `A -> aAb |` ε `B -> bB |` ε
   Ici, le calcul de suivant donne :
   `Suiv(S)={ $ }`
   `Suiv(A)=Prem(B)` $\cup$ `Suiv(S) = { b, $ }`
   `Suiv(B)=Suiv(S)= { $ }`
   On vérifie donc bien que la BNF est LL(1).
2. Pour trouver la BNF, on pourait décomposer L_2 sous forme

   $$L_2 = \{a^k | 0 \leq k \leq 2\}.\{a^m b^m | 0 \leq m\}$$

   Mais ça ne va pas donner une analyse LL(1) (même si c'est la concaténation de 2 langages LL(1)) : on ne sait pas quand sortir de a^k pour entrer dans a^m. On part plutôt sur la décomposition

   $$L_2 = \{\varepsilon\} \cup \{a\}.(\{\varepsilon, b\} \cup \{a\}.\{a^m b^m | 0 \leq m\}.\{b^k | 0 \leq k \leq 2\})$$

   `S -> aA |` ε `Suiv(S)={$}`
   `A -> aBX | b |` ε `Suiv(A)={$}`
   `B -> aBb |` ε `Suiv(B)={b,$}`
   `X -> bY |` ε `Suiv(X)={$}`
   `Y -> b |` ε `Suiv(Y)={$}`
   Elle est LL(1).

Exercice 5. Dans les questions ci-dessous, on étudie des BNFs reconnaissant des fragments du langage C. Pour chacune des BNFs, on aimerait trouver une (E)BNF LL(1) engendrant le même langage que la BNF initiale. On justifiera le caractère LL(1) des (E)BNFs proposées.

▷ **Question 1.** En C, les instructions peuvent commencer par des labels de "`goto`". Exemple :

`etat1: if (cc=='a') goto etat1;`

Voici la BNF à transformer en (E)BNF LL(1) :

$$\begin{array}{lcl} \textbf{inst} & ::= & \texttt{idf} : \textbf{inst} \mid \textbf{exp} \; ; \\ \textbf{exp} & ::= & \texttt{idf} = \textbf{exp} \mid \texttt{num} \end{array}$$

Correction

On modifie juste l'équation de **inst**, en conservant celle de **exp**. Version EBNF :

$$\begin{array}{lcl} \textbf{inst} & ::= & \texttt{idf} \; (: \textbf{inst} \mid = \textbf{exp} \; ;) \\ & \mid & \texttt{num} \; ; \end{array}$$

Version BNF LL(1) (directeurs évidents) :

$$\begin{array}{lcl} \textbf{inst} & ::= & \texttt{idf} \; \textbf{instX} \mid \textbf{num} \; ; \\ \textbf{instX} & ::= & : \textbf{inst} \mid = \textbf{exp} \; ; \end{array}$$

▷ **Question 2.** La syntaxe du langage C définit la notion de *lvalue*, pour "valeur à gauche" d'une affectation. C'est une catégorie d'expressions dont la valeur a une adresse mémoire. Typiquement, un littéral entier "`1`" n'est pas une lvalue. Mais une variable "`x`" en est une. Voici la BNF à transformer :

$$\begin{array}{lcl} \textbf{list} & ::= & \textbf{inst} \mid \textbf{inst} \; \textbf{list} \\ \textbf{inst} & ::= & \textbf{exp} \; ; \\ \textbf{exp} & ::= & \texttt{num} \mid \textbf{lvalue} \mid \textbf{lvalue} = \textbf{exp} \mid \textbf{lvalue} \; ++ \\ \textbf{lvalue} & ::= & \textbf{lvalue} \; . \; \texttt{idf} \mid \textbf{lvalue} \; [\; \textbf{exp} \;] \mid \texttt{idf} \end{array}$$

Correction

Version EBNF

$$\begin{array}{lcl} \textbf{list} & ::= & \textbf{inst} \; (\; \varepsilon \mid \textbf{list} \;) \\ \textbf{exp} & ::= & \texttt{num} \\ & \mid & \textbf{lvalue} \; (\; \varepsilon \mid = \textbf{exp} \mid ++ \;) \\ \textbf{lvalue} & ::= & \texttt{idf} \; (\; . \; \texttt{idf} \mid [\; \textbf{exp} \;] \;)^* \end{array}$$

BNF LL(1) et calcul de directeurs :

```
list ::= inst listX
listX ::=  ε            Suiv(listX)=Suiv(list)={ $ }
        | list          Prem(list) = Prem(exp) = { num, idf }
exp ::= num
      | lvalue expX     Prem(lvalue)=idf
expX ::= ε              Suiv(expX)=Suiv(exp)={;,]}∪Suiv(expX)={;,]}
       | = exp
       | ++
lvalue ::= idf lvalueX
lvalueX ::= . idf lvalueX
          | [exp] lvalueX
          | ε           Suiv(lvalueX)=Suiv(lvalue)=Prem(expX)∪Suiv(exp) = {=,++,;,]}
```

▷ **Question 3.** En C, une branche d'un "`if/else`" peut ne pas être délimitée par des accolades lorsqu'elle ne contient qu'une seule instruction (comme dans l'exemple de la question 1). Mais cela peut introduire des contre-sens sur la signification du programme (notamment si l'indentation est incorrecte). Voici ci-dessous une BNF G_1 qui reconnaît un fragment du langage C avec "`if/else`". Le symbole **exp** correspond à celui de la question 2. Les symboles **L** et **I** remplacent respectivement les **list** et **inst** précédents.

(1) **L** ::= **I L**	(3) **I** ::= **exp** ;	(5) **O** ::= `else` **B**	(7) **B** ::= { **L** }
(2) | ε	(4) | `if` (**exp**) **B O**	(6) | ε	(8) | **I**

1. Montrer que la BNF G_1 est ambiguë.
2. Comme G_1 est ambiguë, elle n'est pas LL(1). Indiquer une paire de règles dont les directeurs sont en conflit.
3. On considère la BNF G_2 obtenu en supprimant la règle (8) de G_1. Calculer les directeurs de G_2. Est-elle LL(1) ? Est-elle ambiguë ?
4. Dans la sémantique du langage C, les ambiguïtés des `if/else` sont éliminées en rattachant le "`else`" au "`if`" le plus proche (parmi les "`if`" ambiguës). Par exemple, la sémantique de "`if`(e_1)`if`$(e_2)e_3;$`else` $e_4;$" est équivalente à "`if`$(e_1)\{$`if`$(e_2)e_3;$`else` $e_4;\};$"
   Expliquer comment écrire un analyseur récursif inspiré d'une analyse LL(1) qui reconnaît le langage de G_1 et dont l'arbre des appels récursifs applique la règle de désambiguation ci-dessus : on donnera en particulier du pseudo-code pour `parse_O` et `parse_B`.

Correction

1. Un mot de forme "`if`(e_1)`if`$(e_2)e_3;$`else` e_4 ;" a 2 arbres d'analyses.
   Arbre 1 :
   ```
       if (exp) B O
                /   \
    if (exp) B O     ε
               |
               else B
   ```
   Arbre 2 :
   ```
       if (exp) B O
                /   \
    if (exp) B O    else B
               |
               ε
   ```
2. Le `else` est dans les directeurs de (5) et (6).
   `Dir(6) = Suiv(O) = Suiv(I)` $\supseteq$ `Suiv(B)` $\supseteq$ `Prem(O)`
3. $$\begin{aligned} \mathrm{Dir}(1) &= \{\mathtt{num}, \mathtt{idf}, \mathtt{if}\} \\ \mathrm{Dir}(2) &= \{\$, \}\} \\ \mathrm{Dir}(3) &= \{\mathtt{num}, \mathtt{idf}\} \\ \mathrm{Dir}(4) &= \{\mathtt{if}\} \\ \mathrm{Dir}(5) &= \{\mathtt{else}\} \\ \mathrm{Dir}(6) &= \{\mathtt{num}, \mathtt{idf}, \mathtt{if}, \$, \}\} \end{aligned}$$
   Notons que cette BNF ajoute) dans Suiv(**exp**) mais ça n'affecte pas le caractère LL(1) de la BNF de **exp** donnée en question 2. Pour arriver à ce calcul de directeurs, on doit résoudre le système d'équation des suivants :
   `Suiv(L) = { $ }` ∪ `{ } }` ∪ `Suiv(L)`
   `Suiv(I) = Prem(L)` ∪ `Suiv(L) = {num,idf,if}` ∪ `Suiv(L)`
   `Suiv(O) = Suiv(I)`

4. La BNF G_2 étant LL(1) : on part du code de son analyseur pour trouver en écrire un pour G_1. Pour que le "`else`" se raccroche au "`if`" le plus proche, il suffit d'être "glouton" sur la consommation du token "`else`". De plus, ajouter la règle (8) dans `parse_B` ne pose alors pas de problème : les règles (7) et (8) ont des directeurs disjoints.
$$\begin{array}{lcl} \mathrm{Dir}(7) & = & \{\{\} \\ \mathrm{Dir}(8) & = & \{\texttt{num}, \texttt{idf}, \texttt{if}\} \end{array}$$
Autrement dit, on écrit un code du style :

```
def parse_O():
    if current == 'else':
        parse_token('else')
        parse_B()

def parse_B():
    if current == '{':
        parse_token('{')
        parse_L()
        parse_token('}')
    else:
        parse_I()
```

▷ **Question 4** (avancée)**.** Il n'existe en fait pas de BNF LL(1) équivalente à G_1. Par contre, il existe une BNF non ambiguë équivalente (et même LALR).

$$\begin{array}{lcl} \mathbf{L} & ::= & \mathbf{L}\ \mathbf{I} \\ & | & \varepsilon \end{array} \qquad \begin{array}{lcl} \mathbf{I} & ::= & \mathbf{I_0} \\ & | & \mathbf{I_1} \end{array} \qquad \begin{array}{lcl} \mathbf{I_0} & ::= & \mathbf{exp}\ ; \\ & | & \texttt{if}\ (\ \mathbf{exp}\ )\ \mathbf{B_0}\ \texttt{else}\ \mathbf{B_0} \end{array} \qquad \begin{array}{lcl} \mathbf{B_0} & ::= & \{\ \mathbf{L}\ \} \\ & | & \mathbf{I_0} \end{array}$$

$$\mathbf{I_1} \ ::= \ \texttt{if}\ (\ \mathbf{exp}\ )\ \mathbf{I} \ \mid\ \texttt{if}\ (\ \mathbf{exp}\ )\ \{\ \mathbf{L}\ \} \ \mid\ \texttt{if}\ (\ \mathbf{exp}\ )\ \mathbf{B_0}\ \texttt{else}\ \mathbf{I_1}$$

1. Attacher sur cette BNF un système d'attributs qui associe à tout programme reconnu par G_1, un programme de même sémantique mais sans ambiguïté. Autrement dit, ce système d'attributs "ajoute" des accolades, comme dans l'exemple précédent, pour qu'on puisse comprendre le code sans avoir à appliquer la règle de désambiguation. On supposera écrit le non-terminal de profil $\mathbf{exp}{\uparrow}w$ où w est une chaîne de caractères correspondant à l'expression reconnue. On essaiera de minimiser les accolades ajoutées.
2. Appliquer votre système d'attributs sur le mot w_1="$\texttt{if}(e_1)\texttt{if}(e_2)e_3;\texttt{else if}(e_4)e_5;$" et le mot w_1 suivi de "$\texttt{else}\ e_6;$".

Correction

Remarque pour les étudiants qui poseraient la question : il n'est pas aisé de trouver une BNF non-ambiguë équivalente. Par exemple, en se restreignant déjà aux mots qui ne contiennent pas d'accolade, on pourrait être tenté de partir sur une BNF du style :

```
I ::= if (exp) I
    | X
X ::= if (exp) X else I
    | exp;
```

Dans cette BNF, le mot "$\texttt{if}(e_1)if(e_2)e_3;\texttt{else}\ e_4;$" a bien un unique arbre d'analyse. Mais le mot "$\texttt{if}(e_1)\texttt{if}(e_2)e_3;\texttt{else if}(e_4)e_5;\texttt{else}\ e_6;$" a toujours deux arbres d'analyse (le dernier `else` étant rattaché soit au premier, soit au dernier `if`).

Pour corriger ce type d'ambiguïté, on distingue dans $\mathbf{I}$ le sous-langage $\mathbf{I_1}$ des mots de $\mathbf{I}$ qui ont un `if` sans `else` non protégé par des accolades. On pose alors $\mathbf{I_0} = \mathbf{I} \backslash \mathbf{I_1}$.

Le caractère LALR de la BNF du sujet est garantit par Bison, cf. le fichier `pending_else.y` du dépôt.

1. Voilà le système d'attributs

$$
\begin{array}{rcll}
\mathbf{L}\uparrow l & ::= & \mathbf{L}\uparrow l' \ \mathbf{I}\uparrow i & l := l' \ i \\
 & | & \varepsilon & l := \varepsilon \\
\mathbf{I}\uparrow i & ::= & \mathbf{I_0}\uparrow i & \\
 & | & \mathbf{I_1}\uparrow i & \\
\mathbf{I_0}\uparrow i & ::= & \mathbf{exp}\uparrow e \ ; & i := e \ ; \\
 & | & \texttt{if} \ (\ \mathbf{exp}\uparrow e \ ) \ \mathbf{B_0}\uparrow b_1 \ \texttt{else} \ \mathbf{B_0}\uparrow b_2 & i := \texttt{if} \ (\ e \ ) \ b_1 \ \texttt{else} \ b_2 \\
\mathbf{B_0}\uparrow b & ::= & \{ \ \mathbf{L}\uparrow l \ \} & b := \{ \ l \ \} \\
 & | & \mathbf{I_0}\uparrow i & b := i \\
\mathbf{I_1}\uparrow i & ::= & \texttt{if} \ (\ \mathbf{exp}\uparrow e \ ) \ \mathbf{I}\uparrow i' & i := \texttt{if} \ (\ e \ ) \ \{ \ i' \ \} \\
 & | & \texttt{if} \ (\ \mathbf{exp}\uparrow e \ ) \ \{ \ \mathbf{L}\uparrow l \ \} & i := \texttt{if} \ (\ e \ ) \ \{ \ l \ \} \\
 & | & \texttt{if} \ (\ \mathbf{exp}\uparrow e \ ) \ \mathbf{B_0}\uparrow b \ \texttt{else} \ \mathbf{I_1}\uparrow i' & i := \texttt{if} \ (\ e \ ) \ b \ \texttt{else} \ i'
\end{array}
$$

   On ajoute des accolades sur une seule règle : la première de $\mathbf{I_1}$. On pourrait ajouter encore moins d'accolades à condition d'ajouter des règles. Sur le mot "$\texttt{if}(e_1)e_2;$", il est en effet inutile d'ajouter des accolades.

2. Le mot w_1 donne "$\texttt{if}(e_1)\{\texttt{if}(e_2)e_3;\texttt{else if}(e_4)\{e_5;\}\}$".
   Le mot w_1 suivi de "$\texttt{else}\ e_6;$" donne "$\texttt{if}(e_1)\{\texttt{if}(e_2)e_3;\texttt{else if}(e_4)e_5;\texttt{else}\ e_6;\}$".