**Correction de l'examen – 18 mai 2022**

# 1 Introduction

On désire écrire un interpréteur d'expressions parenthésées reconnaissant un mot parenthésé ET construisant une forêt (liste d'arbres). Soit la grammaire $G = (\{(;);LITCH;,\},\{foret, arbres, arbre, suite\}, R, foret)$ avec les règles $R$ suivantes :

$$\begin{aligned} foret &\rightarrow \ '(' \ arbres \ ')' \mid \varepsilon \\ arbres &\rightarrow \ arbre \ suite \\ suite &\rightarrow \ ',' \ arbre \ suite \mid \varepsilon \\ arbre &\rightarrow \ LITCH \ foret \end{aligned}$$

Les espaces et tabulations ne sont que des séparateurs et seront donc filtrés. Un littéral chaîne ($LITCH$) est un mot non vide constitué de lettres minuscules ou majuscules et/ou de chiffres décimaux et sera reconnu par l'analyseur lexical. Par exemple le mot `( a (a1, a2), b )` représente la forêt suivante représentée sous forme indentée :

```
a
    a1
    a2
b
```

Le fonctionnement de l'interpréteur est un cycle consistant à lire une ligne de commande saisie par l'utilisateur (terminée par une fin de ligne), puis à afficher la forêt correspondante sous forme indentée.

# 2 Théorie

**Question 1 (3 points)** Donnez la liste des Premiers et des Suivants de chaque symbole non terminal de $G$.

| | Premiers | Suivants |
|---|---|---|
| $foret$ | $\{'(', \varepsilon\}$ | $\{',', ')', \$\}$ |
| $arbres$ | $\{LITCH\}$ | $\{')'\}$ |
| $suite$ | $\{',', \varepsilon\}$ | $\{')'\}$ |
| $arbre$ | $\{LITCH\}$ | $\{',', ')'\}$ |

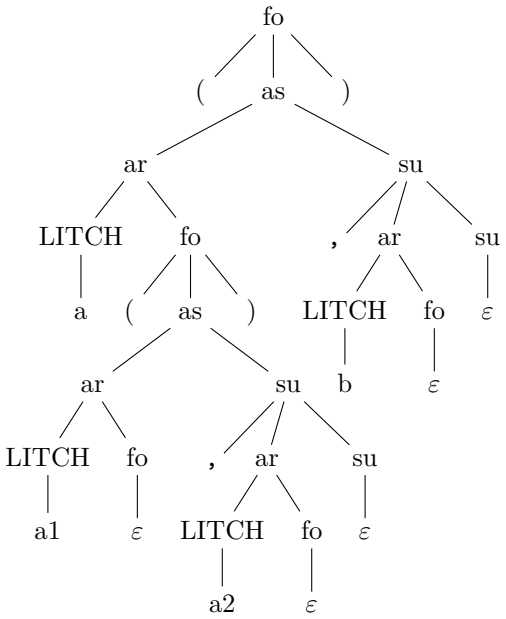
**Question 2 (4 points)** Calculez la table d'analyse descendante de l'automate à pile et représentez-la en abrégeant les symboles non terminaux : foret (fo), arbres (as), suite (su), arbre (ar).

| | ( | ) | LITCH | , | $ |
|---|---|---|---|---|---|
| fo | $fo \rightarrow' (' \ as \ ')'$ | $fo \rightarrow \varepsilon$ | | $fo \rightarrow \varepsilon$ | $fo \rightarrow \varepsilon$ |
| as | | | $as \rightarrow ar \ su$ | | |
| su | | $su \rightarrow \varepsilon$ | | $su \rightarrow',' \ ar \ su$ | |
| ar | | | $ar \rightarrow LITCH \ fo$ | | |

**Question 3 (2 points)** Effectuez la reconnaissance du mot `( a (a1, a2), b )` en représentant sur chaque ligne : la pile, le suffixe du mot commençant par le jeton courant, la règle de grammaire activée.

| Pile | Flot d'entrée | Règle |
|---|---|---|
| $ | `( a (a1, a2), b )$` | $fo \rightarrow' (' \ as \ ')'$ |
| $)as( | `( a (a1, a2), b )$` | |
| $)as | `a (a1, a2), b )$` | $as \rightarrow ar \ su$ |
| $)su ar | `a (a1, a2), b )$` | $ar \rightarrow LITCH \ fo$ |
| $)su fo a | `a (a1, a2), b )$` | |
| $)su fo | `(a1, a2), b )$` | $fo \rightarrow' (' \ as \ ')'$ |
| $)su )as( | `(a1, a2), b )$` | |
| $)su )as | `a1, a2), b )$` | $as \rightarrow ar \ su$ |
| $)su )su ar | `a1, a2), b )$` | $ar \rightarrow LITCH \ fo$ |
| $)su )su fo a1 | `a1, a2), b )$` | |
| $)su )su fo | `, a2), b )$` | $fo \rightarrow \varepsilon$ |
| $)su )su | `, a2), b )$` | $su \rightarrow',' \ ar \ su$ |
| $)su )su ar , | `, a2), b )$` | |
| $)su )su ar | `a2), b )$` | $ar \rightarrow LITCH \ fo$ |
| $)su )su fo a2 | `a2), b )$` | |
| $)su )su fo | `), b )$` | $fo \rightarrow \varepsilon$ |
| $)su )su | `), b )$` | $su \rightarrow \varepsilon$ |
| $)su ) | `), b )$` | |
| $)su | `, b )$` | $su \rightarrow',' \ ar \ su$ |
| $)su ar , | `, b )$` | |
| $)su ar | `b )$` | $ar \rightarrow LITCH \ fo$ |
| $)su fo b | `b )$` | |
| $)su fo | `)$` | $fo \rightarrow \varepsilon$ |
| $)su | `)$` | $su \rightarrow \varepsilon$ |
| $) | `)$` | |
| $ | `$` | ACCEPT |

**Question 4 (1 point)** Dessinez l'arbre de dérivation correspondant à la reconnaissance précédente.




**Question 5 (1 point)** La grammaire $G$ est-elle LL(1) ? Justifiez votre réponse.

La table d'analyse descendante ne contient aucun conflit, donc la grammaire est non ambiguë. Toute grammaire non ambiguë est LL(1), donc la grammaire $G$ est également LL(1).

# 3 Pratique

**Question 6 (3 points)** Écrire un source flex réalisant l'analyse lexicale.

```
%{
#include <stdio.h>

#define LITCH 300
%}
%option noyywrap
%%
[ \t]           { /* Filtrer */ }
[a-zA-Z0-9]+    { return LITCH; }
.|\n            { return yytext[0]; }
%%
```

# 4 Analyse syntaxique et sémantique

On utilisera les méthodes C++ suivantes de la classe Forêt contenue dans `foret.h` :

```
class Foret {
  public:
    Foret(const char* rac, Foret *sousArbres); // Constructeur d'une forêt
        // contenant un arbre de racine rac; sousArbres peut être NULL.
    Foret *ajouter(Foret *f); // ajoute la forêt f à la fin de la forêt courante.
    void afficher(int nb=0); // affiche la forêt sous forme indentée
};
```

**Question 7 (6 points)** Écrire un analyseur descendant récursif (C/C++) réalisant cet interpréteur en utilisant la fonction `yylex()` définie précédemment à l'aide du source flex. L'interpréteur doit boucler tant que la ligne saisie n'est pas "q" (pour quitter).

```
#include "foret.h"

Foret *fo(); Foret *as(); Foret *su(Foret *); Foret *ar();

int jeton;
int numcar = 0;

Foret *fo() {
    if (jeton == '(') {
        AVANCER
        Foret *arbres = as();
        TEST_AVANCE(')')
        return arbres;
    }
    return NULL;
}

Foret *as() {
    return su(ar());    // On aurait aussi pu faire ar()->ajouter(su())
                        // si su() ne prenait pas d'argument..
}

Foret *su(Foret *left) {
    if (jeton == ',') {
        AVANCER
        return left->ajouter(su(ar()));
    }
    return left;
}

Foret *ar() {
    if (jeton == LITCH) {
        char *res = yytext;
```

```
        AVANCER
        return new Foret(res, fo());
    }
    else {
        ERREUR_SYNTAXE
    }
}

int main() {
    AVANCER
    while (jeton != 'q') {
        Foret *foret = fo();
        if (jeton == '\n') {
            AVANCER
            printf("\nMot reconnu\n");
            foret->afficher();
        }
        else {
            ERREUR_SYNTAXE
        }
        delete foret;
    }
    return 0;
}
```