SERIE DE TD N°:2 COMPILATION
ANALYSE SYNTAXIQUE : INTRODUCTION, RAPPELS ET COMPLEMENTS

Exercice 1

1) Soit la grammaire G = ({a, b }, {S, A, B, C}, S, P) où P est défini par :

S → AB | BC

A → BA | a

B → CC | b

C → AB | a

Analyser la chaîne **babaab** de manière descendante puis ascendante, en construisant, à chaque fois, son arbre de dérivation.

Exercice 2

1) Soit la grammaire des expressions arithmétiques G = ({+, -, *, /, a, b, c, (,) }, {E}, E, P) où P est défini par :

E → E + E | E – E | E * E | E / E | (E) | -E | a | b | c

Donner l'arbre de dérivation de la chaîne **b + a * b - c** ? Que peut-on en déduire ?

2) Soit la grammaire G' = = ({+, -, *, /, a, b, (,) }, {E, T, F, L}, E, P) où P est défini par :

E → E + T | E – T | T

T → T * F | T / F | F

F → -F | L

L → (E) | a | b | c

a) Donner l'arbre de dérivation de la chaîne **b + a * b - c**. Que peut-on en déduire ?

b) Donner l'arbre abstrait correspondant à la chaîne **b + a * b - c**.

c) Donner les arbres syntaxiques concret et abstrait de la chaîne **b - a + c**.

Exercice 3

1) Soit la grammaire des expressions booléennes G = ({ou , et, non, vrai, faux, (,) }, {A}, A, P) où P est défini par :

A → A ou A | A et A | non A | (A) | vrai | faux

Donner l'arbre de dérivation de la chaîne **non faux ou vrai et vrai** ? Que peut-on en déduire ?

2) Proposer une grammaire G' pour éliminer le problème posé par la grammaire G.

3) Donner l'arbre de dérivation de la chaîne **non faux ou vrai et vrai** en utilisant la grammaire G'.

Exercice 4

Soit la grammaire G d'un langage proche de Pascal, exprimée sous forme EBNF de la manière suivante, exprimer la grammaire G sous forme de diagrammes syntaxiques.

<Programme> ::= Program ident ; <Bloc>.

<Bloc> ::= [Const <SuitConst> ;] [Var <SuitVar> ;] { Procedure ident ; <Bloc> ;}
Begin <Inst> {; <Inst> } End

<SuitConst> ::= <DecConst> { , < DecConst>}

<DecConst> ::= ident = nbEnt

<SuitVar> ::= ident {, ident}

<Inst> ::= ident := <Exp> | If <Cond> Then <Inst> [Else <Inst>]
| Repeat <Inst> Until <Cond> | While <Cond> Do <Inst> | Begin <Inst> {; <Inst> } End

<Cond> ::= <Exp> (= | <> | < | > | <= | >=) <Exp>

<Exp> ::= <Terme> { (+ | -) <Terme>}

<Terme> ::= <Facteur> { (* | /) <Facteur>}

<Facteur> ::= ident | nbEnt | (<Exp>)