

1 BNF

Backus-Naur Form, inventée pour les besoins du groupe de travail ALGOL (1960).

1.1 La BNF ... en BNF

```
1 syntax      ::= { rule }
2 rule        ::= identifieur " :="  expression
3 expression  ::= term { "|" term }
4 term        ::= factor { factor }
5 factor      ::= identifieur |
6               quoted_symbol |
7               "(" expression ")" |
8               "[" expression "]" |
9               "{" expression "}"
10 identifieur ::= letter { letter | digit }
11 quoted_symbol ::= "\"" { any_character } "\""
```

1.2 Exercices

Le langage PL/0 de N. Wirth est décrit par une grammaire de type E-BNF (extended BNF). Ecrivez quelques programmes dans ce langage.

```
1 program = block "." .
2
3 block =
4   ["const" ident "=" number {"," ident "=" number} ";"]
5   ["var" ident {"," ident} ";"]
6   {"procedure" ident ";" block ";"} statement .
7
8 statement =
9   ident " :=" expression
10  | "call" ident
11  | "begin" statement ";" {statement ";"} "end"
12  | "if" condition "then" statement
13  | "while" condition "do" statement .
14
15 condition =
16   "odd" expression
17   | expression ("="|"#"|"<"|<="|>"|>=") expression .
18
19 expression = ["+"|"−"] term {("+"|"−") term} .
20
21 term = factor {("*"|"/" ) factor} .
```

```

22
23 factor =
24     ident
25     | number
26     | "(" expression ")" .

```

Exercice. Voici des exemples de déclarations de type en langage Pascal. Fournissez une grammaire qui couvre au moins les exemples :

```

type chaine = array [1 .. 30] of char;
type date = record
    jour : 1..31;
    mois : 1..12;
    annee : integer
end;
type personne = record
    nom, prenom : chaine;
    naissance : date
end;

```

Exercice. Voici un programme écrit dans un langage jouet

```

function fac(n)
    local r = 1, i
    for i = 1 to n do
        let r = r * i
    endfor
    return r
endfunction

let encore = 1
while encore == 1 do
    print "valeur de n ? "
    read n
    if n < 0
    then
        print "n est négatif"
        let encore = 0
    else
        let r = fac(n)
        print "factorielle ", n, " = ", r
    endif
endwhile

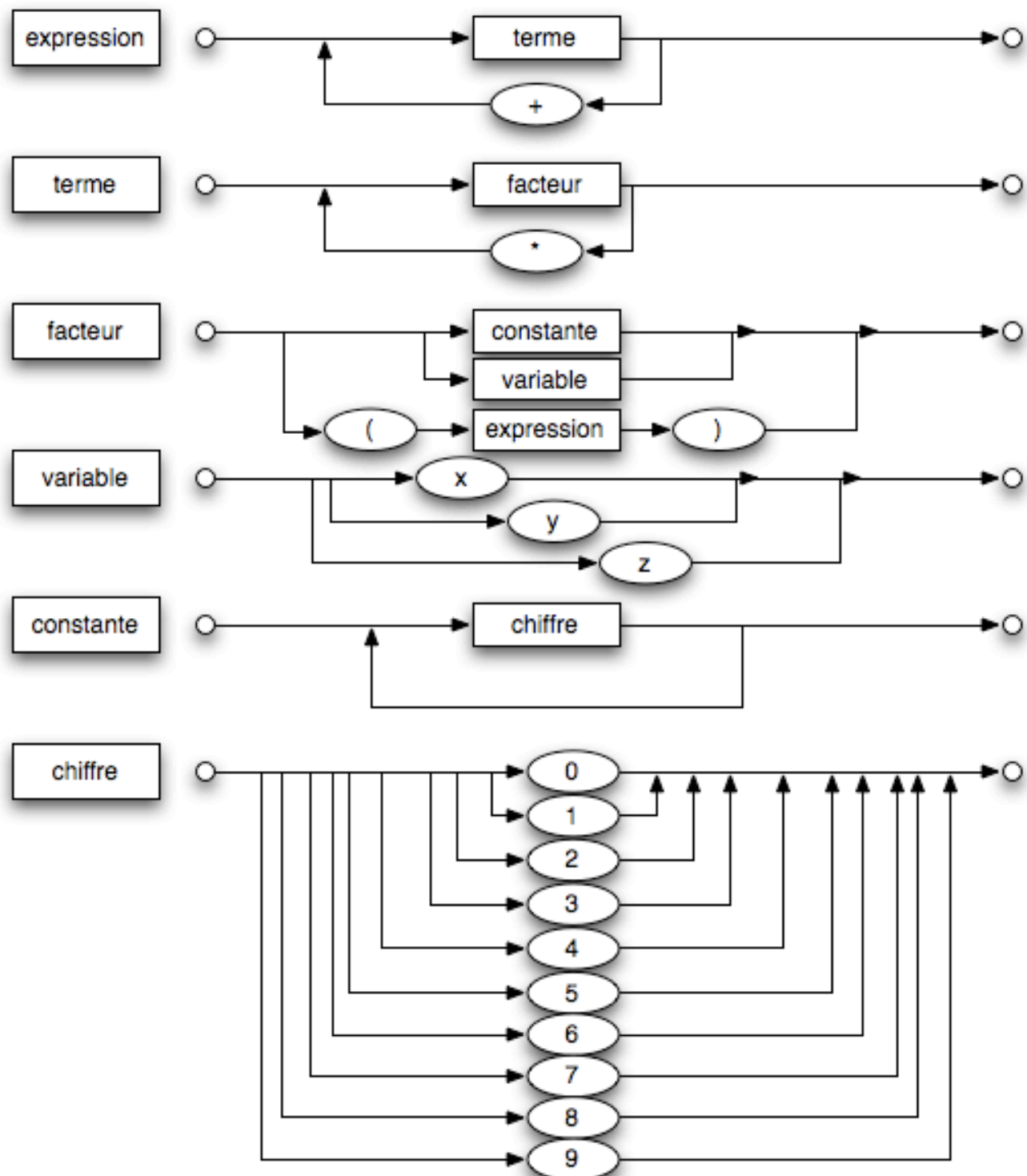
```

Fournir une description du langage en BNF étendue.

2 Diagrammes syntaxiques

Exemple : les expressions arithmétiques.

http://commons.wikimedia.org/wiki/File:Diagrammes_Syntaxiques.png



Exercice. Convertir la description de PL/0 en diagrammes syntaxiques.

3 Descente récursive, un exemple

Le programme ci-dessous analyse une expression arithmétique, et en fournit une paraphrase.

```
1 // compiler avec C++ version 11
2 // g++ -std=c++11 lecture-expr.cxx -o lecture-expr
3
4 #include <iostream>
5 using namespace std;
6
7 // -----
8
9 enum TypeLexeme {
10     OUVRANTE, FERMANTE,      PLUS, MOINS, ETOILE, BARRE,
11     NOMBRE, IDENTIFICATEUR,  FIN, ERREUR
12 };
13
14 // -----
15
16 class AnalyseurLexical {
17 private :
18     string m_chaine;
19     uint   m_longueur, m_position;
20     TypeLexeme m_typeLexeme;
21     string m_lexeme;
22
23 public :
24
25     AnalyseurLexical(const string & chaine) :
26         m_chaine(chaine),
27         m_longueur(chaine.size()),
28         m_position(0)
29     {
30         avancer();
31     }
32
33     void avancer()
34     {
35         m_lexeme = "";
36         while ((m_position < m_longueur) &&
37             isspace(m_chaine[m_position])) {
38             m_position++;
39         }
40         if (m_position == m_longueur) {
41             m_typeLexeme = FIN;
42             return;
43         }
44         char premier = m_chaine[m_position++];
45         m_lexeme = premier;
46         // nombres
47         if (isdigit(premier)) {
48             m_typeLexeme = NOMBRE;
49             while ((m_position < m_longueur) &&
50                 isdigit(m_chaine[m_position])) {
51                 m_lexeme += m_chaine[m_position++];
52             }
53             return;
54         }
55         // identificateurs
```

```

56         if (isalpha(premier)) {
57             m_typeLexeme = IDENTIFICATEUR;
58             while ((m_position < m_longueur) &&
59                 isalnum(m_chaine[m_position])) {
60                 m_lexeme += m_chaine[m_position++];
61             }
62             return;
63         }
64         // symboles
65         m_typeLexeme
66         = premier == '(' ? OUVRANTE
67           : premier == ')' ? FERMANTE
68           : premier == '+' ? PLUS
69           : premier == '-' ? MOINS
70           : premier == '*' ? ETOILE
71           : premier == '/' ? BARRE
72           : ERREUR;
73     }
74
75     TypeLexeme typeLexeme(void) const
76     {
77         return m_typeLexeme;
78     }
79     string lexeme(void) const
80     {
81         return m_lexeme;
82     }
83 };
84
85 void test_analyse_lexicale(const string & s)
86 {
87     cout << "—test_analyse_lexicale" << endl;
88     cout << "chaine: " << s << endl;
89     AnalyseurLexical lex(s);
90     while (lex.typeLexeme() != FIN) {
91         cout << "—" << lex.typeLexeme()
92              << " " << lex.lexeme() << endl;
93         lex.avancer();
94     };
95 }
96
97 // -----
98
99 class Expression {
100 public:
101     virtual ~Expression() {} ;
102     virtual void afficher() const = 0;
103 };
104
105 class ExpressionBinaire
106 : public Expression
107 {
108     string m_nom;
109     const Expression * m_gauche, *m_droite;
110
111 public:
112     ExpressionBinaire(const string & nom,
113                      const Expression * gauche,

```

```

114         const Expression * droite)
115     :       m_nom(nom),
116         m_gauche(gauche),
117         m_droite(droite)
118     {};
119
120     void afficher() const override
121     {
122         cout << "(" << m_nom << " de ";
123         m_gauche->afficher();
124         cout << " et ";
125         m_droite->afficher();
126         cout << ")";
127     }
128
129     ~ExpressionBinaire() {
130         delete m_gauche;
131         delete m_droite;
132     }
133 };
134
135 class ExpressionSimple
136 : public Expression
137 {
138     string m_type, m_nom;
139 public:
140     ExpressionSimple(const string & type, const string & nom)
141         : m_type(type), m_nom(nom)
142     {}
143     void afficher() const override
144     {
145         cout << "la " << m_type << " " << m_nom;
146     }
147 };
148
149 // -----
150 class AnalyseurSyntaxique
151 {
152     AnalyseurLexical m_lex;
153     Expression *m_expr = NULL;
154 public:
155     AnalyseurSyntaxique(const string & chaine)
156         : m_lex(chaine)
157     {}
158
159     Expression * expression() {
160         if (m_expr == NULL) {
161             m_expr = lireExpr();
162         }
163         return m_expr;
164     }
165
166     Expression * lireExpr()
167     {
168         Expression * expr = lireTerme();
169         for(;;) {
170             TypeLexeme t = m_lex.typeLexeme();
171             if (t == PLUS) {

```

```

172         m_lex.avancer();
173         Expression * terme = lireTerme();
174         expr = new ExpressionBinaire("la_somme",
175                                     expr, terme);
176     } else if (t == MOINS) {
177         m_lex.avancer();
178         Expression * terme = lireTerme();
179         expr = new ExpressionBinaire("la_difference",
180                                     expr, terme);
181     } else {
182         break;
183     }
184 }
185 return expr;
186 }
187
188 Expression * lireTerme() {
189     Expression * terme = lireFacteur();
190     for(;;) {
191         TypeLexeme t = m_lex.typeLexeme();
192         if (t == ETOILE) {
193             m_lex.avancer();
194             Expression * facteur = lireFacteur();
195             terme = new ExpressionBinaire("le_produit",
196                                           terme, facteur);
197         } else if (t == BARRE) {
198             m_lex.avancer();
199             Expression * facteur = lireFacteur();
200             terme = new ExpressionBinaire("le_quotient",
201                                           terme, facteur);
202         } else {
203             break;
204         }
205     }
206     return terme;
207 }
208
209 Expression * lireFacteur()
210 {
211     Expression *facteur = NULL;
212     if (m_lex.typeLexeme() == OUVRANTE) {
213         m_lex.avancer();
214         facteur = lireExpr();
215         if (m_lex.typeLexeme() != FERMANTE) {
216             cout << "**_oups_il_manque_une_fermante" << endl;
217         }
218     } else if (m_lex.typeLexeme() == NOMBRE) {
219         facteur = new ExpressionSimple("constante",
220                                       m_lex.lexeme());
221     } else if (m_lex.typeLexeme() == IDENTIFICATEUR) {
222         facteur = new ExpressionSimple("variable",
223                                       m_lex.lexeme());
224     }
225     else {
226         cout << "**_oups,_probleme_avec_"
227              << m_lex.lexeme() << endl;
228     }
229     m_lex.avancer();

```

```

230         return facteur;
231     }
232 };
233
234 void test_analyse_syntaxique(const string & s)
235 {
236     cout << "— test_analyse_syntaxique" << endl;
237     cout << "chaîne : " << s << endl;
238     AnalyseurSyntaxique a(s);
239     Expression * r = a.expression();
240     r->afficher();
241     cout << endl;
242     delete r;
243 }
244
245 // -----
246
247 int main(int argc, char **argv)
248 {
249     test_analyse_lexicale("beta*beta-(4*alpha*gamma)");
250     test_analyse_syntaxique("beta*beta-(4*alpha*gamma)");
251     test_analyse_syntaxique("HT*(100+TVA)/100");
252     return 0;
253 }

```

Résultat :

```

— test analyse lexicale
chaîne : beta * beta - (4*  alpha*gamma)
- 7 beta
- 4 *
- 7 beta
- 3 -
- 0 (
- 6 4
- 4 *
- 7 alpha
- 4 *
- 7 gamma
- 1 )
— test analyse syntaxique
chaîne : beta * beta - (4*  alpha*gamma)
( la difference de ( le produit de la variable beta et de la variable beta) et
  de ( le produit de ( le produit de la constante 4 et de la variable alpha)
    et de la variable gamma) )
— test analyse syntaxique
chaîne : HT * (100+TVA)/100
( le quotient de ( le produit de la variable HT et de ( la somme de la
  constante 100 et de la variable TVA) ) et de la constante 100)

```