S4 - Mathématiques

Théorie des langages

compléments

version du 23 juillet 2021

michel billaud
département informatique
iut de bordeaux

Table des matières

1 BNF

La notation BNF (Backus Naur Form) a été introduite par John Backus et Peter Naur pour décrire le langage de programmation Algol 60.

Plus précisement, elle a été (en grande partie) inventée par Backus pour décrire Algol 58 dans un rapport de recherche. Et Peter Naur (qui travaillait sur le même langage) s'est alors aperçu qu'il voyait autrement la syntaxe d'Algol 58.

Ils ont donc décidé, pour le travail sur Algol, de distribuer des descriptions écrites de la syntaxe pour que tout le monde parle de la même chose lors des réunions du groupe de travail.

Source : `http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html`

1.1 BNF de base

La notation de base (historique) de la BNF est simple
— la chaîne "`::=`" signifie "*est défini par*",
— la barre | sépare des alternatives,
— les chevrons "`<...>`" entourent des noms de catégories,
— les éléments terminaux sont représentés tels quels,
et se traduit directement en grammaire formelle.

Exemple

```
<program> ::= program
                  <déclarations>
              begin
                  <instructions>
              end ;

<instruction> ::=   <affectation>
                  | <instruction_tant_que>
                  | <instruction_si_alors_sinon>
                  | ...
```

Dans des textes plus modernes, on peut avoir d'autres conventions : terminaux en gras ou en police "machine à écrire", non-terminaux en italique, etc.

program ::= **program** *declarations* **begin** *declarations* **end**

1.2 Extensions

Quelques extensions s'avèrent pratiques, elles sont notées par des *méta-caractères* :

— des **crochets** pour entourer les éléments optionnels

instruction_si_alors_sinon ::=
if *condition*
then *instruction*
[**else** *instruction*]

— des **accolades** pour les éléments répétés (zero ou plusieurs fois)

liste_d'expressions ::=
( )
| **(** *expression* { **,** *expression* } **)**

— on peut aussi entourer les terminaux de guillemets.

Exercice : comment définir les listes d'expressions en BNF de base, sans ces méta-caractères ?

Un exemple plus complet, la syntaxe de la BNF ... en BNF

```
syntax      ::= { rule }
rule        ::= identifier "::=" expression
expression  ::= term { "|" term }
term        ::= factor { factor }
factor      ::= identifier |
                quoted_symbol |
                "(" expression ")" |
                "[" expression "]" |
                "{" expression "}"
identifier  ::= letter { letter | digit }
quoted_symbol ::= """ { any_character } """
```

1.3 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;
```

Notez qu'en Pascal le point-virgule est un séparateur, alors qu'en C c'est un terminateur. Il est donc facultatif après le dernier élément d'un *record*.

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
```

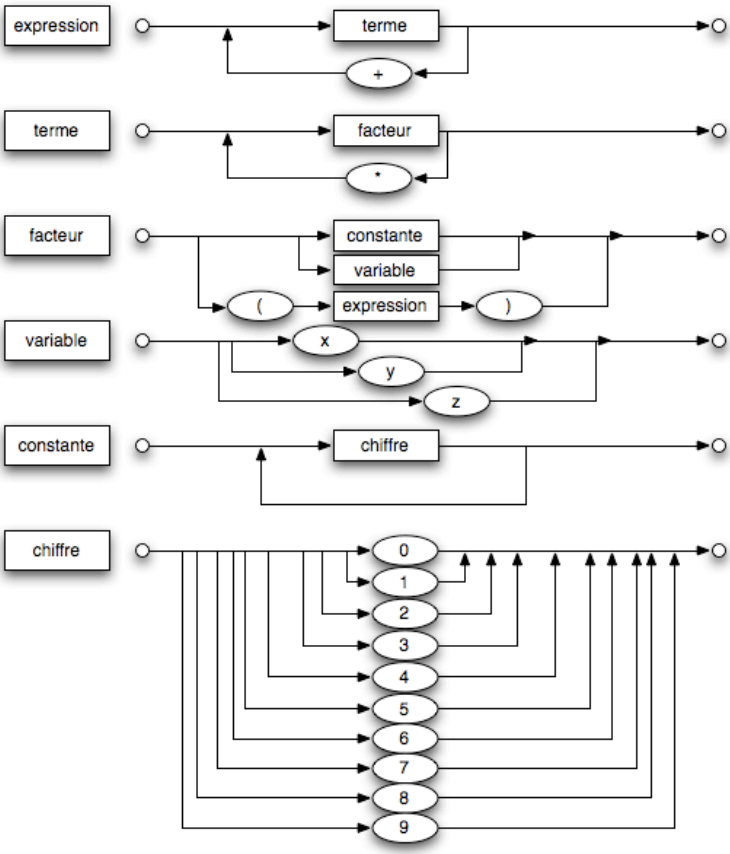
Fournir une description du langage en BNF étendue.

2 Diagrammes syntaxiques

Les diagrammes syntaxiques expriment essentiellement la même chose que la BNF mais sous une forme plus facile à appréhender

Exemple : les expressions arithmétiques.

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



Les diagrammes syntaxiques peuvent être vus comme des programmes (visuels) qui décrivent l'analyse d'une chaine de caractères, en termes d'appels de sous-programmes : pour reconnaître une expression, il faut d'abord reconnaitre un terme. Pour reconnaitre un terme il faut d'abord reconnaitre un facteur, etc.

C'est la base de la technique d'analyse récursive descendante, technique élégante qui a été appliquée dans nombre de compilateurs "écrits à la main" [1]

1. En effet, une autre option est d'utiliser un programme qui fabriquera un automate à pile à partir d'une grammaire du langage à reconnaitre.

2.1 Déterministe ou pas ?

A certains endroits du diagramme il y a des fourches, ce qui laisse craindre un non-déterminisme. Par exemple, dans *constante*, si on rencontre un second *chiffre*, doit-on boucler, ou sortir ?

De préférence, on voudra un analyseur déterministe, il faut donc étudier la grammaire sous-jacente pour montrer qu'on n'a en fait jamais le choix.

Méthode. on regarde en particulier

- l'ensemble $First(N)$ des terminaux qui peuvent apparaître en première position d'un non-terminal N.
- l'ensemble $Follow(N)$ des terminaux qui peuvent apparaître après un non-terminal N.

Pour *First*, on a évidemment

- $First(Chiffre) = \{0..9\}$
- et $First(Variable) = \{x, y, z\}$.
- et donc $First(Constante) = First(Chiffre) = \{0..9\}$

Par conséquent, les trois branches de la fourche d'entrée de *facteur* correspondent à des cas disjoints (le troisième est une parenthèse ouvrante), le choix peut même se faire en regardant seulement un seul caractère de la chaîne d'entrée.

Pour *Follow*. Les flèches de boucle posent la question de savoir si un "+" peut suivre une *expression*, une étoile un terme, un chiffre une constante. Donc de savoir dans quels contextes on peut rencontrer ces non-terminaux, *dans une chaîne valide*.

On suppose donc que le but est d'analyser une expression isolée, donc qu'on a une règle liant l'axiome$S$, les expressions et $ une marque de fin.

$$S \to \text{expression } \$$$

De cette règle en déduit immédiatement que

$$\$ \in Follow(expression)$$

Du troisième diagramme, on tire aussi qu'une parenthèse fermante (notons-l f) peut suivre une expression : $f \in Follow(expression)$.

Un certain nombre d'inclusions se déduisent également : si un symbole peut suivre une expression, alors il peut aussi suivre un terme (diag. 1). Un plus peut aussi suivre un terme.

2.2 Un systeme d'inequations

Le tableau ci-dessous résume nos trouvailles

origine			
axiome	$\$$	$\in$	$Follow(expression)$
diag. 1	$Follow(expression)$	$\subseteq$	$Follow(terme)$
diag. 1	$"+"$	$\in$	$Follow(terme)$
diag. 2	$Follow(terme)$	$\subseteq$	$Follow(facteur)$
diag. 2	$"*"$	$\in$	$Follow(facteur)$
diag. 3	$Follow(facteur)$	$\subseteq$	$Follow(constante)$
diag. 3	$Follow(facteur)$	$\in$	$Follow(variable)$
diag. 3	f	$\in$	$Follow(expression)$
diag. 4	$First(chiffre)$	$\subseteq$	$Follow(chiffre)$
diag. 4	$First(constante)$	$\subseteq$	$Follow(chiffre)$

C'est un système d'inéquations sur des ensembles ; nous en cherchons les plus petites solutions.

Pas de panique, la résolution n'est pas compliquée, elle se fait par un algorithmes itératif :

- on part d'ensembles vides E, T, F, Co, V, Ch qui représentent les différents "follow".
- on exécute une boucle qui interprète chaque inéquation comme une affectation
  - la première ajoute $\$$ à E
  - la seconde ajoute le contenu de E à T,
  - etc.
- il n'y a qu'un nombre fini de symboles, et on ne fait qu'ajouter des éléments, au bout d'un certain nombre de tours la situation va se stabiliser : on s'arrête.

2.3 Use the computer, Luke

L'ordinateur qui est votre ami va calculer la solution en moins de deux, si on écrit un programme Python comme celui-ci

```
from copy import deepcopy

F = { "e" : { "dollar", "fermante"},
      "t" : { "+"},
      "f" : { "*" },
      "co" : set(),
      "v" : set(),
      "ch" : {"chiffre"} }

```

```
def m(src, dst):
  F[dst].update(F[src])

n = 0
while True:
  n = n + 1
  old = deepcopy(F)
  m("e", "t")
  m("t", "f")
  m("f", "co")
  m("f", "v")
  m("co", "ch")
  if old == F: break

print ("arret apres %d iterations" % n)
for k in F:
  print (k + " => " + str(F[k]))
```

On obtient le résultat

```
arret apres 2 iterations
ch => set(['chiffre', '*', 'dollar', 'fermante', '+'])
co => set(['+', '*', 'dollar', 'fermante'])
f => set(['fermante', '+', '*', 'dollar'])
t => set(['fermante', '+', 'dollar'])
v => set(['+', '*', 'dollar', 'fermante'])
e => set(['dollar', 'fermante'])
```

Donc, résumons, les diagrammes syntaxiques sont déterministes parce que

- après une expression il ne peut pas y avoir un “+” (diag. 1)
- un facteur ne peut pas être suivi par “*” (diag. 2)
- une constante ne peut pas être suivie par un chiffre (diag. 3)

2.4 Compléments

- Ici nous avons montré qu'on peut choisir son chemin dans le diagramme de l'exemple, en regardant seulement le prochain caractère de la chaine à analyser.
  Certains langages nécessitent de regarder plusieurs caractères.
- En C, C++, c'est même plus compliqué : le nombre d'éléments à lire pour différencier, par exemple, une affectation d'un appel de fonction n'est pas borné. Voyons par exemple

```
f(....);
g(....)->a = b; // g fonction qui retourne un pointeur
```

et l'analyse syntaxique doit donc se baser également sur les types déclarés.

— Le calcul de $First$ et $Follow$ est un petit peu plus compliqué quand les non-terminaux peuvent produire un mot vide. Si on a une règle

$$A \to BC$$

, $First(A)$ contient $First(B)$, mais aussi $First(C)$ si B peut produire le mot vide. Et aussi $Follow(A)$ si B et C produisent le vide. $First(A)$ doit alors être interprété comme "le premier symbole que je peux rencontrer si je vais vers un bloc A dans le diagramme".

Tout ceci est fort intéressant et mérite d'être étudié de plus près, vous en saurez plus si vous continuez en Mastère ou Ecole d'ingénieur d'informatique.

3 Descente récursive, un exemple

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

Voici le résultat des tests

```
-- test analyse lexicale
chaine : beta * beta - (4*  alpha*gamma)
- 7 beta
- 4 *
- 7 beta
- 3 -
- 0 (
- 6 4
- 4 *
- 7 alpha
- 4 *
- 7 gamma
- 1 )
-- test analyse  syntaxique
chaine : 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
chaine : 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)
```

et le source dans les pages qui suivent.

```
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␣de␣";
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 << "chaine  :  " << 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 }
```