S4 - Mathématiques

Théorie des langages

notes de cours

version du 23 juillet 2021

michel billaud
département informatique
iut de bordeaux

Avant-Propos

Ce document contient mes notes personnelles pour préparer un cours de Théorie des Langages de 16 heures (sur 4 semaines) destiné aux étudiants de 2ème année d'IUT Informatique, au printemps 2014.

Il s'est trouvé que le programme pédagogique national (PPN) a changé récemment, et que ce cours donné traditionnellement au semestre 4 (seconde année) est maintenant fait au semestre 2. Pendant l'année de transition, il fallait faire ce cours en première année du nouveau PPN et en deuxième année de l'ancien. Il se trouve qu'en plus il fallait remplacer le collègue mathématicien qui assurait jusque-là ce cours, c'était l'occasion de me remettre à la théorie des langages.

Malgré le volume horaire restreint (16h), j'ai décidé de parler un peu des langages algébriques qui passent souvent à la trappe en DUT. Il me parait important de faire le lien avec la syntaxe (et la compilation) des langages de programmation, qui est quand même la motivation principale pour faire ce cours en DUT, du point de vue des informaticiens...

Dans ce cours, j'ai essayé de montrer un lien avec les applications relative traditionnelles à la programmation : automates de reconnaissance syntaxique, analyse des langages de programmation, ... tout en montrant des techniques mathématiques nouvelles (pour les étudiants) : travailler dans un produit cartésien (intersection de langages rationnels), dans l'ensemble des parties d'un ensemble (déterminisation), principe des tiroirs et des chaussettes (pour montrer que $\{a^n b^n\}$ n'est pas rationnel), etc.

En tout cas j'ai eu plaisir à préparer ce cours, même pour une seule fois, et j'espère ne pas avoir trop traumatisé mes étudiants !

Table des matières

Première partie

Pour commencer

1 Motivations

1.1 Linguistique

La *théorie des langages* est issue de problématiques de linguistique, pour l'étude de la structure des langues naturelles.

Une problématique était d'identifier des structures, comme par exemple la décomposition d'une phrase en groupe sujet/groupe verbal/complément, structures qui peuvent être communes à des groupes de langues

Remonter à une "langue originelle" et/ou identifier des mécanismes innés communs à l'espèce humaine, qui prédisposeraient à l'usage d'une langue.

1.2 Langages informatiques

À la fin des années 50 sont apparus les premiers langages de programmation. Au début, la programmation consistait à établir des listes d'instructions machine, mais assez rapidement est apparu le besoin d'écrire des programmes sous une forme plus "naturelle", dans laquelle les éléments (fonctions, instructions, déclarations, expressions, etc) sont combinés selon des règles précises.

Avec cela, la *définition du langage*, vient une autre préoccupation : comment écrire, à moindre frais, des *compilateurs* qui analysent le code source, et génèrent sa traduction ?

Inversement, peut-on identifier des familles de langages faciles à analyser ?

Applications

- reconnaissances des expressions régulières
- automatiser la fabrication des compilateurs, en fournissant une description du langage à reconnaître à un *méta-compilateur*, qui produira un compilateur.

1.3 Développements mathématiques

Au delà de ces motivations pratiques, la *théorie des langages* s'intéresse aux propriétés mathématiques des *langages formels*.

C'est un domaine de l'informatique théorique qui a connu une très grande activité à partir des années 70.

2 Langages formels

2.1 Alphabet, lettres, mots

Définitions et notations

- Un **alphabet** $A = \{a, b, c...\}$, est un ensemble fini [1] de **lettres**.
- Traditionnellement, les lettres sont notées $a, b, c \ldots$, et les variables qui parlent des lettres sont $x, y, z \ldots$
- un **mot** n est une séquence finie $(x, y, \ldots z)$ de lettres. On note un mot tout simplement en juxtaposant ses lettres, soit $xy \ldots z$.
- u, v, w désignent généralement des mots (w comme word)
- la **taille** d'un mot w se note $|w|$, c'est la longueur de sa séquence de lettres.
- la lettre ϵ désigne le **mot vide**, de taille nulle.
- A^n désigne l'ensemble des mots de longueur $n \in \mathbb{N}$,
- $A^* = \bigcup_{i \in \mathbb{N}} A^i = A^0 \cup A^1 \cup A^2 \cup \ldots$ l'ensemble de tous les mots sur A.

Exemples

- soit A l'alphabet à trois lettres $A = \{a, b, c\}$
- a, bc, aaa sont des mots de longueurs respectives 1, 2 et 3 sur A

2.2 Concaténation, facteurs

- **La concaténation** de deux mots consiste à les mettre bout à bout. Plus formellement, si $u = x_1 x_2 \ldots x_n$ et $v = y_1 y_2 \ldots y_p$ sont deux mots de longueurs respectives n et p, le mot $u.v$ (noté aussi uv) est la séquence $x_1 x_2 \ldots x_n y_1 y_2 \ldots y_p$ de longueur $n + p$.

Exercice. La concaténation

- est-elle associative ?
- est-elle commutative ?
- a-t-elle un élément neutre ?

1. dans le cadre de ce cours

— u est dit **préfixe** de w si il existe un $v \in A^*$ que $uv = w$, c'est-à-dire que les lettres de u sont identiques aux premières lettres de lettres de w.
— De la même façon, v est dit **suffixe** de w si il existe un $u \in A^*$ que $uv = w$,

Propriété . Soit trois mots u, v, w. on a équivalence entre

1. $uv = uw$
2. $vu = wu$
3. $v = w$

Le lemme de Levi est un petit résultat utile dans les preuves :

Lemme : Si u et v sont des préfixes d'un même mot w, alors l'un d'eux est un préfixe de l'autre.

Preuve par cas, selon l'ordre des longueurs. Idée : si $|u| \leq |v|$, les lettres de u sont identiques aux $|u|$ premières lettres de w qui sont identiques aux premières lettres de w, donc u est un préfixe de v.

Le théorème de commutation est un peu plus surprenant

Théorème. Deux mots commutent si et seulement si ils sont tous deux la répétition d'un facteur commun.

Autrement dit $uv = vu$ si et seulement si il existe un mot f et des entiers n et p tels que $u = f^n$ et $v = f^p$.

Preuve

— Évident dans un sens, puisque $f^n f^p = f^{n+p} = f^p f^n$
— Dans l'autre sens, par récurrence sur $N = |u| + |v|$.
  — vrai dans le cas de base, quand $N = 0$, alors $u = v = \epsilon$, commutent et sont une répétition de ϵ.
  — si $|u| = |v|$, la commutation $uv = vu$ entraîne que $u = v$
  — sinon, comme u et v sont des préfixes de $uv = vu$, l'un est préfixe (strict) de l'autre. On suppose que c'est u, il existe donc w tel que $v = uw$. L'égalité

$$uv = vu$$

s'écrit aussi

$$u(uw) = (uw)u$$

qui se simplifie (voir plus haut) en

$$uw = wu$$

Par hypothèse de récurrence, u et w sont des répétitions d'un même facteur f, et il en est donc de même pour $v = uw$, cqfd.

2.3 Langages

Définition Un **langage** est un ensemble de mots.

Exemples

— le langage des mots de deux lettres au plus sur $A = \{a, b\}$ est

$$L = \{\epsilon, a, b, c, aa, bb, cc, ab, ba, bc, cb, ac, ca\}$$

— $\{a^n \| 2 \leq n \leq 4\} = \{aa, aaa, aaaa\}$

2.4 Opérations sur les langages

Opérations ensemblistes : les langages sont des ensembles (de mots), on peut en faire

— l'intersection : par exemple si L_1 est l'ensemble des mots qui commencent par la lettre a, et L_2 ceux qui finissent par b, $L_1 \cap L_2$ contient les mots qui commencent a *et* finissent par b.
— l'union, souvent notée $+$: $L_1 + L_2$ contient les mots qui commencent a *ou* finissent par b (ou les deux) ;
— la différence, etc.

Le *produit* de deux langages est une opération spécifique, notée par un point : $L_1.L_2$ est l'ensemble des mots qui sont obtenus par concaténation d'un mot de L_1 avec un mot de L_2. Exemple : avec $L_1 = \{a, ab, c\}$ et $L_2 = \{\epsilon, b\}$

$$L_1.L_2 = \{a, ab, c, abb, cb\}$$

L'élévation à une puissance L^n d'un langage L consiste à concaténer n mots de L. Par exemple $\{a, ab\}^2 = \{aa, aab, aba, abab\}$

L'étoile L^n est l'union de tous les L^n, pour $n \geq 0$: c'est l'ensemble des mots que l'on peut décomposer en une suite de facteurs pris dans L. Il contient le mot vide, même si celui-ci n'est pas dans L.

Ceci rejoint la notation A^* pour le langage de tous les mots sur A.

2.5 Comment définir un langage ?

— par une **spécification** : l'ensemble des mots de 3 lettres sur $A = \{a, b\}$,
— par un **algorithme qui produit tous les mots** du langage.
— par un **algorithme qui détermine si le mot appartient au langage**

Exercice : quel langage est produit par l'algorithme suivant ?

```
pour n de 0 à l'infini, faire
 | pour i de 0 à n faire
 |   |  w = epsilon
 |   |  pour j de 0 à i faire
 |   |    | w = w.a
 |   |  pour j de i+1 à n faire
 |   |    | w = w.b
 |   |  afficher w
```

Exercice : écrire un algorithme qui produit tous les mots qui ne contiennent qu'un seul a (et autant de b qu'on veut avant et après).

Exercice : quel langage est reconnu par l'algorithme suivant ?

```
donnée : w, mot sur {a,b}
booléen t = faux
pour toute lettre x de w, faire
 | si x == b,  alors
 |     | t = vrai
 | sinon si t alors
 |     | retourner faux
retourner vrai
```

Exercice : algorithme qui reconnaît le langage des mots sur $\{a, b, c\}$ qui ont exactement autant de a que de b.

Exercice : quel langage reconnaît cet algorithme ?

```
donnée w
entier c = 0
pour toute lettre x de w, faire
 | selon x :
 |   si c'est un a, alors c++
 |   si c'est un b, alors c--
 | si c < 0, retourner faux
retourner vrai
```

2.6 Reconnaissance de motifs

En utilisant le *shell*, vous faites abondamment usage des méta-caractères "jokers" * et ?. Exemple

```
a2ps  *.cc
cp    prog-v?.*  archives
```

Une chaîne correspond à un *motif* comme `prog-v?.*`, si
— les caractères normaux sont identiques,
— un joker ? correspond à n'importe quel caractère,
— un joker * correspond àà n'importe quelle chaîne.

2.6.1 Backtracking

L'examen du motif et de la chaîne à reconnaître se fait en parallèle. Quand le caractère du motif est une étoile, il y a deux possibilités
— supposer qu'elle représente la chaîne vide, et examiner la même chaîne avec le reste du motif
— suppose qu'elle "avale" au moins un caractère du mot et reprendre l'examen

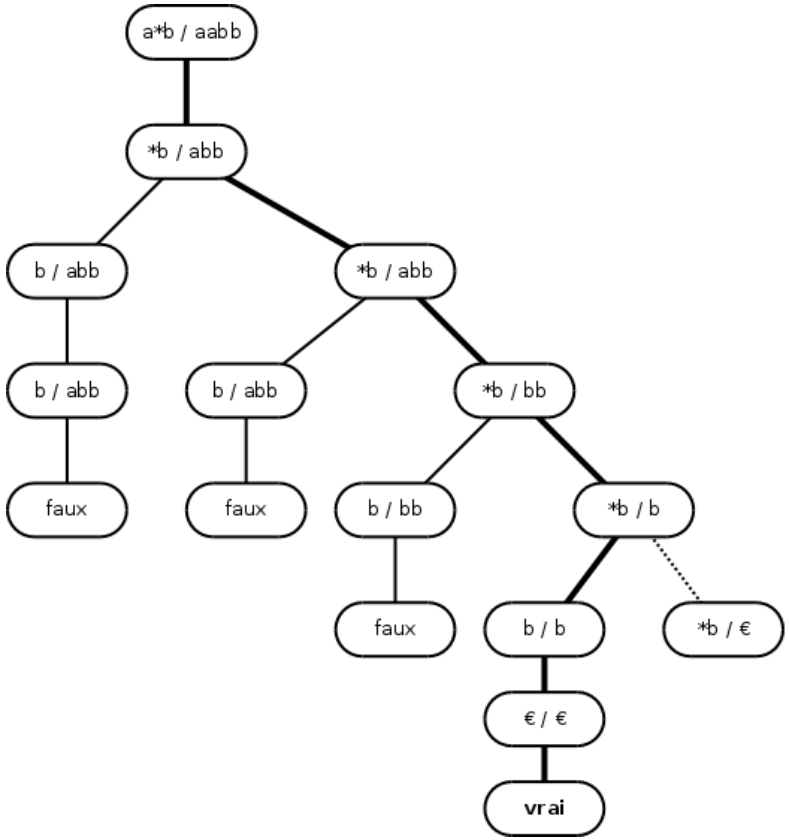
On peut formaliser l'algorithme de façon récursive, et séparant les cas : motif vide ou non, commençant par un point d'interrogation, un étoile ou autre chose, et chaîne vide ou non.

```
prédicat : reco (m , c)  où m est un motif
                         et c une chaîne
début
  selon que
   m est vide =>  voir si  c est vide
```

```
    m est x.m' =>  voir si  c est y.c',  x==y  et
                            et reco (m', c')
    m est ?.m' =>  voir si  c est y.c'
                            et reco (m', c')
    m est *.m' =>  voir si  reco(m', c)
                            ou  c est y.c'
                                et reco(m, c')
fin
```

Voici l'arbre de calcul pendant le déroulement de l'appel à `reco("a*b", "aaba")`.

Quand le motif commence par une étoile, il y a deux possibilités, soit on considère que l'étoile correspond à une sous-chaîne vide (on enlève l'étoile du motif, sous-arbre de gauche), soit qu'elle "mange" au moins un caractère de la chaîne (sous-arbre de droite).



On a là un exemple typique de procédure d'exploration d'un espace de possibilités par *backtracking* : on explore une alternative en conservant la possibilité de revenir en arrière.

2.6.2 Version récursive

Le programme C++ suivant utilise la récursivité pour gérer le backtracking par l'intermédiaire de la pile des appels.

```
/**                                                    1
 * reconnaissance d'un motif avec jokers               2
 * version  rcursive                                   3
 */                                                    4
                                                       5
```

```
bool reconnait(const char * motif,
               const char * chaine)
{
  if (*motif == '\0') {
    return *chaine == '\0';
  }
  // ? : on saute un caractre
  if (*motif == '?') {
    return (*chaine != '\0')
         && reconnait(motif+1, chaine+1);
  }
  // * : on saute l'toile ou un caractre
  if (*motif == '*') {
    return reconnait(motif+1, chaine)
        || ((*chaine != '\0')
            && reconnait(motif, chaine+1));
  }
  // autres caractres
  return (*motif == *chaine)
         && reconnait(motif+1, chaine+1);
}
```

Le programme est appelé ainsi :

```
/*
 * motifs.cc
 * reconnaissance d'un motif avec jokers * et ?
 */

#include <iostream>
#include <cassert>

#include "reconnait1.h"

using namespace std;

int main(int argc, char **argv)
{
  cout << "--␣debut␣tests" << endl;
       assert ( reconnait("abc", "abc"));
  assert (! reconnait("ab", "abc"));
```



```
  assert (! reconnait("abc", "ab"));
  assert ( reconnait("a?c", "abc"));
  cout << "--␣tests␣ok" << endl;
  for(;;) {
    string motif, chaine;
    cout << "motif␣:␣";
    getline(cin, motif);
    if (cin.fail()) break;
    cout << "chaine␣:␣";
    getline(cin, chaine);
    if (cin.fail()) break;
    cout << "la␣chaine␣" << chaine
         << (reconnait(motif.c_str(),
                       chaine.c_str()) ?
             "␣est␣" : "␣n'est␣pas␣")
         << "reconnue␣par␣" << motif << endl;
  }
  cout << endl << "ok." << endl;
       return 0;
}
```

2.6.3 Version itérative

On peut faire mieux, en gérant explicitement une pile des alternatives restant à explorer.

```
/**
 * reconnaissance d'un motif avec jokers
 * version itrative
 */

#include <stack>

bool reconnait(const char * motif,
               const char * chaine)
{
  stack <pair <const char *, const char * > >
        alternatives;
  const char *m = motif, *c = chaine;
  pair<const char *, const char *> p(m,c);
```

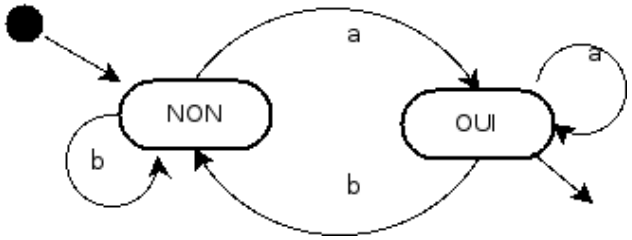
```
  alternatives.push(p);

  while (! alternatives.empty())
  {
    m = alternatives.top().first;
    c = alternatives.top().second;
    alternatives.pop();

    // le motif est examin lettre par lettre
    for ( ; *m; m++)
    {
      if (*m == '*') {
        if (*c != '\0') {
          // noter alternative : l'toile absorbe
          // au moins un caractre de la chaine
          p.first  = m;
          p.second = c+1;
          alternatives.push(p);
        };
      } else if (*c == '\0') { // fin de chaine
            break;
      } else if (*m == '?' || *m == *c) {
          c++;
      } else {                  // mauvais caractre
          break;
      };
     }
    // arriv au bout ?
    if (*m == '\0' && *c == '\0') {
       return true;
    }
  }
  // plus d'alternatives ?
  return false;
}
```


2.7 Éviter le backtracking

La problématique va être d'éviter ce backtracking, qui peut être extrêmement coûteux en temps de calcul.

Exercice. Calculez, en fonction de $n \in \mathbb{N}$ le nombre de noeuds de l'arbre de recherche pour le motif "*a" et le mot $b^n a$.

Pour reconnaître ces mots, il suffit d'un petit dessin :



On part de l'état de gauche, on suit les flèches au fur et à mesure de la lecture des lettres du mot à reconnaître, et à la fin on sait que le mot est reconnu si on est arrivé dans l'état OUI. C'est la notion d'automate

Deuxième partie

Langages rationnels

3 Automates finis déterministes

3.1 Définition

Un automate $\mathcal{A}$ est la donnée
- d'un alphabet A (fini)
- d'un ensemble fini d'**états** appelé traditionnellement Q
- d'un **état initial** $q^I \in Q$,
- d'un ensemble $Q_F \subseteq Q$ d'**états finaux** (ou terminaux) ;
- d'une **fonction de transition** $\delta : Q \times A \to Q$

Les automates se prêtent à une représentation sous forme de graphes. Les états sont des noeuds, les transitions des flèches entre les noeuds, portant l'étiquette de la lettre.

L'état initial est indiqué par un point, ou une flèche entrante. Les états finaux le sont par un double cercle, un triangle, une flèche sortante ...

Un mot est reconnu par un automate si, en partant de l'état initial et en suivant les transitions qui correspondent aux lettres successives du mot, on arrive dans un état final.

Plus formellement, à partir d'un w un mot de longueur n on construit une séquence de $n+1$ états $(q_0, q_1, \ldots, q_n)$ avec
- $q_0 = q^I$ (l'état initial)
- $q_i = \delta(q_{i-1}, w_i)$, pour tout i entre 1 et n

w est reconnu si et seulement si le dernier état est final, soit $q_n \in Q_F$.

3.2 Quelques exemples

3.3 Comment coder un automate

Un automate peut aisément être codé par un tableau (constant) à deux dimensions. La première dimension correspond à l'état et la seconde les catégories de caractères en entrée.

Pour noter les états et les catégories, on peut faire usage d'énumérations.


```
/*
 * automate
 *
 * exemple de codage d'automate,
 * reconnait les mots qui
 * ont un nombre pair de lettres
 * et au moins un chiffre
 * Les autres caractres sont ignors
 */
#include <iostream>
#include <cassert>

using namespace std;


enum Etat {   Lp0c, Li0c, Lp1c, Li1c };
enum Categorie { Lettre, Chiffre, Autre };


const Etat etatInitial = Lp0c;
const Etat etatFinal   = Lp1c;

Etat delta[][3] =
{
    { Li0c, Lp1c, Lp0c },
    { Lp0c, Li1c, Li0c },
    { Li1c, Lp1c, Lp1c },
    { Lp1c, Li1c, Li1c },
};

Categorie categorie(char x)
{
    return isdigit(x) ? Chiffre
           : isalpha(x) ? Lettre
           : Autre;
}

bool estValide (const char chaine[])
{
```

```
    Etat etat = etatInitial;
    for (int i=0; chaine[i] != '\0'; i++)
    {
        Categorie x = categorie(chaine[i]);
        etat = delta[etat][x];   // transition
    }
    return (etat == etatFinal);
}

// ------------------------------------------------------
int main(int argc, char **argv)
{
    cout << "-- debut tests" << endl;
    assert ( ! estValide(""));
    assert ( ! estValide("a"));
    assert ( estValide("5"));
    assert ( estValide("a1bcd"));
    assert ( ! estValide("aa"));
    assert ( ! estValide("z9"));
    assert ( estValide("ab3"));
    assert ( estValide("a3b"));
    assert ( estValide("a1b2"));
    assert ( estValide("314pi"));
    cout << "-- tests ok" << endl;
}
```

3.4 Un exemple plus détaillé

Construction d'un gros exemple : reconnaissance d'une ligne de fichier CSV

Le contenu d'une feuille de calcul (tableur) peut être exporté au format CSV (comma-separated values), dans lequel

- les cellules d'une même ligne de la feuille sont représentées par une ligne de texte
- si une case n'est pas vide, son contenu est entouré de guillemets,
- les guillemets qui sont dans les cases sont doublés
- les représentations des cases sont séparés par des virgules

Exemple, la rangée de cellules

	123	route de paris	lieu dit "Le bourg"

est codée par la ligne

```
,"123","route de paris","lieu dit ""Le bourg"""""
```

On demande de construire une fonction qui détermine si une ligne de texte est valide ou non.

3.5 Applications

Définissez des automates pour les applications suivantes

Vérifier la ponctuation d'un texte : en français une ponctuation double, comme deux-points ou point-virgule, doit être précédée et suivie d'un ou plusieurs espaces, contrairement aux ponctuations simples (espaces après mais pas avant). Faire un automate qui vérifie la ponctuation d'une ligne de texte (qui ne peut pas commencer par une ponctuation, même précédée d'espaces).

On travaillera sur un alphabet $A = \{e, s, d, l\}$ de catégories de caractères : e pour les espaces, s et d pour les ponctuations simples et doubles, l pour les lettres.

Vérifier une chaîne de caractères en C : Une chaîne de caractère en C est entourée par des guillemets. Si on doit coder des guillemets à l'intérieur, on les précède par un "backslash". Si on doit représenter un backslash, on le double. On travaillera sur un alphabet $A = \{g, b, a\}$ de catégories de caractères : g pour les guillemets, b pour backslash, a pour les autres.

3.6 Langages rationnels

Définition. Un langage est dit **rationnel** si il existe un automate qui le reconnaît.

Les langages rationnels constituent une des familles les plus importantes de la hiérarchie définie par Chomsky.

Quelques propriétés :

- La famille des langages rationnels sur un alphabet A est close par par complément, union, intersection, différence (voir ci dessous) étoile (preuve plus tard, avec les automates non-déterministes)
- on peut déterminer si deux automates reconnaissent le même langage (par un calcul sur les automates)
- on peut calculer (automatiquement) un automate minimal.

3.7 Un langage qui n'est pas rationnel

Tous les langages ne sont pas rationnels, mais encore faut-il le prouver. L'exemple classique est $L = \{a^n b^n, n \in \mathbb{N}\}$

L'idée de la preuve, c'est que pour reconnaître si un mot est dans L, il va falloir compter combien on a rencontré de a, pour voir ensuite si on a autant de b. Le seul moyen de compter, c'est d'être dans des états différents. Comme le nombre d'états est fini, on ne peut pas y arriver.

Preuve par l'absurde

1. on suppose qu'il existe un automate à N états qui reconnaît L, et on considère le mot $a^N b^N$ qui est dans L.
2. on regarde le parcours dans l'automate pendant la reconnaissance du préfixe a^N. On part de l'état initial, et on fait N "pas".
3. Par le principe de poteaux et des intervalles, on est donc passé par $N+1$ états.
4. Principe des tiroirs et des chaussettes : comme il n'y a que N états distincts, on est forcement passé au moins deux fois au même endroit : on a donc fait une boucle de transitions étiquetées a, soit B la longueur de cette boucle.
5. en partant de l'état initial, on arrive donc au même état que par a^N, en faisant un tour de plus, c'est à dire par $a^N a^B = a^{N+B}$
6. de là, suivre N transitions "b" mène à un état terminal, puisque $a^N b^N$ est reconnu.
7. donc $a^{N+B} b^N$ est également reconnu par l'automate.
8. Or il ne fait pas partie du langage : c'est une contradiction.

3.8 Complément d'un langage rationnel

Il est facile de voir que le complément (par rapport à A^*) d'un langage rationnel L sur A est lui-même rationnel.

Preuve

1. L est rationnel, il existe un automate $\mathcal{A}$ qui le reconnaît.

2. construisons $\mathcal{A}'$ identique à $\mathcal{A}$, avec les mêmes états, les mêmes transitions, le même état initial, mais en prenant comme états terminaux ceux qui ne le sont pas dans $\mathcal{A}$: $Q'_F = Q \setminus Q_F$.

Ainsi, les mots reconnus par $\mathcal{A}'$ sont ceux qui ne sont pas dans L, et inversement. Le langage L' reconnu par $\mathcal{A}'$ est le complément de L.

3.9 Intersection et Union, automate produit

Montrons maintenant que l'intersection de deux langages rationnels L_1 et L_2 est elle-même rationnelle.

L'idée intuitive est très simple : pour voir si un mot w appartient à **l'intersection** $L_1 \cap L_2$, on utilise deux automates $\mathcal{A}_1, \mathcal{A}_2$ qui reconnaissent L_1 et L_2.

Jusqu'ici, on reconnaissait un mot en pointant du doigt l'état initial, et en suivant les flèches correspondant aux lettres. Le mot est reconnu si on s'arrête sur un état final.

Maintenant on fait la même chose avec deux doigts, un par automate. Au départ on pointe la paire d'états initiaux, et on progresse simultanément dans les deux automates. Le mot est reconnu si on s'est arrêté sur une paire d'états finaux.

On travaille donc sur des **paires d'états** des deux automates. C'est la notion de produit d'automates.

Plus formellement, on construit ainsi le produit $\mathcal{A}$ qui reconnaît l'intersection

- l'alphabet A est le même,
- l'ensemble Q des états est le produit cartésien $Q_1 \times Q_2$,
- l'état initial est la paire d'états initiaux $({q^I}_1, {q^I}_2)$
- les états finaux sont les paires d'états finaux des automates $Q_F = Q_{1F} \times Q_{2F}$,
- la fonction de transition δ combine les transitions dans les deux automates

$$\delta((q_1, q_2), x) = (\delta_1(q_1, x), \delta_2(q_2, x))$$

pour toute lettre $x \in A$, et pour tous états $q_1 \in Q_1, q_2 \in Q_2$.

On notera que cette construction peut faire apparaître des états inaccessibles depuis l'état initial. On peut les ignorer.

Exercice. Donnez des automates pour
- L_1 les mots qui commencent par a.
- L_2 les mots qui finissent par b.
- $L_3 = L_1 \cap L_2$
- L_4 mots qui contiennent au moins deux a consécutifs,
- $L_5 = = L_3 \cap L_4$

Pour l'union de deux langages rationnels la construction est similaire : un mot est reconnu si on arrive dans un état final pour au moins l'un des automates.

Autrement dit on prend comme états terminaux l'ensemble

$$Q_F = (Q_{1F} \times Q_2) \cup (Q_1 \times Q_{2F})$$

.

3.10 Concaténation de langages

Il est tentant de vouloir appliquer la même idée pour déterminer si un mot w appartient à la concaténation de deux langages L_1 et L_2 : on part de l'état initial du premier automate, et on le suit jusqu'à obtenir un état final. Puis on part de l'état initial du second, et on continue avec les lettres qui restent.

Mais malheureusement ce n'est pas aussi simple : il peut y avoir plusieurs préfixes u de w qui conduisent à des états finaux de $\mathcal{A}_1$: a priori on ne sait pas pour lequel il faut choisir de passer dans le second automate.

Pour aborder ces problèmes, nous allons donc introduire une généralisation de la notion d'automate : les **automates non déterministes avec ϵ-transitions**.

Nous verrons que si cette notion est plus générale, et permet d'exprimer plus facilement les idées (notamment la reconnaissance d'un produit de langage, ou l'étoile d'un langage), en fait on peut toujours les ramener à des automates déterministes qui reconnaissent le même langage.

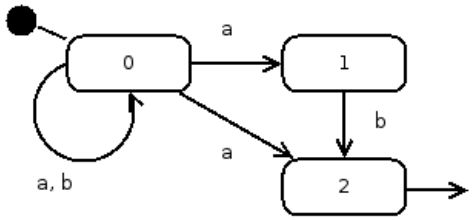
4 Automates non déterministes

4.1 Un automate non-déterministe

$\mathcal{A}$ sur un alphabet A est la donnée de
- un ensemble d'états Q
- un ensemble $Q_I \subseteq Q$ d'états initiaux
- un ensemble $Q_F \subseteq Q$ d'états finaux
- un ensemble $\Delta \subseteq (Q \times A \times Q)$ de transitions

Sur le schéma d'un automate, il y a donc éventuellement
- plusieurs entrées possibles au lieu d'une seule pour les automates déterministes
- en partant d'un état, il peut y avoir plusieurs transitions concernant la même lettre, ou aucune.

Exemple : dans l'automate ci-dessous, en lisant des a depuis l'état 0, on peut rester dans 0, aller en 1 ou encore en 2. Par contre dans l'état 1, on ne peut aller nulle part en lisant un b.



Un mot w est reconnu par un automate non-déterministe si et seulement il existe un chemin dans l'automate
- qui parte d'un état initial
- qui soit étiqueté par les lettres successives du mot
- qui mène à un état final

Sur l'automate, on voit assez facilement que pour aller de 0 à 2, il faut un mot qui se termine par la lettre a ou par le suffixe ab.

L'intérêt pratique des automates non déterministes, c'est qu'ils sont plus faciles à concevoir que les automates non déterministes.

Exemple : définissez un automate qui reconnaît les mots qui se terminent par $ababab$.

4.2 Retour sur l'union

En 3.9, nous avons vu une façon de construire un automate déterministe qui reconnaît l'union de deux langages rationnels.

Exemple : construisez un automate reconnaissant les mots

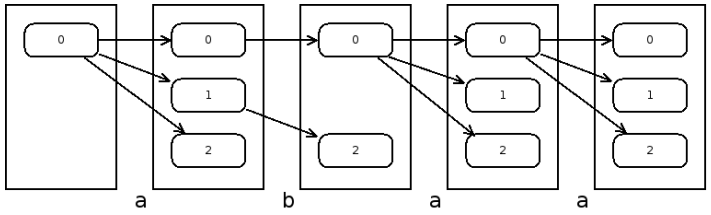
Exercice : construire des automates non-déterministes pour les mots sur $A = \{l, c, a\}$ (lettre, chiffre, autre) qui reconnaissent
- les nombres entiers, suites d'au moins un chiffre,
- les identificateurs (une lettre, suivie de lettres ou de chiffres)
- les nombres et les identificateurs

La construction générale de l'automate pour l'union.
- l'automate est l'union de deux copies distinctes des automates
- états : union des deux ensembles d'états
- états initiaux (resp. finaux) : union des deux ensembles d'états initiaux (resp. finaux)
- transitions : union des ensembles de transitions.

4.3 Déterminisation

En partant de l'état initial, la lettre a fait passer dans l'état 0, l'état 1 ou l'état 2.

Ceci conduit à l'idée de considérer *l'ensemble des états* où l'on peut se trouver après avoir lu une séquence de lettres, par exemple $abaa$
- au début on ne peut être que dans l'état initial, soit $\{0\}$;
- le premier a mène dans $\{0, 1, 2\}$,
- le b dans $\{0, 2\}$,
- le a dans $\{0, 1, 2\}$,
- le dernier a dans $\{0, 1, 2\}$.



Parmi ces états se trouve un état terminal : le mot est donc reconnu.

Le procédé de déterminisation est donc simple : partant d'un automate $\mathcal{A}$ non déterministe, on construit un automate déterministe $\mathcal{A}'$ de la façon suivante :
- les états sont les parties de Q : $Q' = \mathcal{P}(Q)$
- l'état initial regroupe tous les états initiaux : $q'_I = Q_I$
- les états finaux sont ceux qui contiennent au moins un état final : $Q'_F = \{q' \in Q' | q' \cap Q_F \neq \emptyset\}$
- la fonction de transition fait passer dans l'état qui regroupe tous les états accessibles par une lettre : $\delta'(q', x) = \{q_2 | \exists (q_1, x, q_2) \in \Delta, q_1 \in q'\}$ pour tout état $q' \in Q'$, et toute lettre $x \in A$.

En pratique, on n'a pas besoin de calculer tous les états, seulement ceux qui sont accessibles depuis l'état initial.

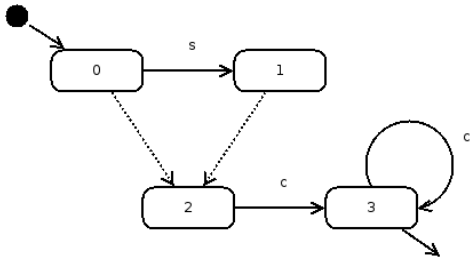
Exercice : déterminiser l'automate ci-dessus.

Conséquence : les langages reconnus par les automates non-déterministes sont rationnels (et inversement).

4.4 Automates avec ϵ-transitions

En ajoutant des "epsilon-transitions", on donne la possibilité à l'automate, de passer d'un état à un autre "spontanément", sans lire de lettre.

Exemple, reconnaître les nombres formés de chiffres (c), qui commencent éventuellement par un signe (s). Les ϵ-transitions sont représentées par des flèches en pointillés :



Ces transitions apportent une facilité d'expression des idées. Elles peuvent être éliminées (nous verrons comment) pour revenir des automates non-déterministes "simples", qui eux-mêmes se ramènent à des automates déterministes.

Conséquence. Les automates non-déterministes avec ϵ-transitions reconnaissent les langages rationnels.

4.5 Concaténation de langages

La construction d'un automate qui reconnaît la concaténation de deux langages, à partir de leurs automates, est simple :

— les états initiaux sont ceux du premier automate
— les états finaux sont ceux du second
— les transitions sont conservées
— on y ajoute des ϵ-transitions entre les états finaux du premier et les états initiaux du second

Voir sur l'exemple précédent, qui est le produit de deux langages
— celui qui reconnaît un signe facultatif
— celui qui reconnaît une suite de chiffres.

Application : faire un automate qui reconnaît les nombres écrits en notation scientifique, comme par exemple $-3.14E+23$. Ils comportent une mantisse et éventuellement un exposant (commençant par E). La mantisse a éventuellement un signe, peut être entière ou décimale (mais si il y a un point décimal, il y a au moins un chiffre à gauche ou à droite). L'exposant peut aussi être signé

4.6 Étoile

Et de même l'étoile d'un langage s'obtient, par exemple
— en ajoutant un état supplémentaire à l'automate, qui sera le seul état initial et final.
— en ajoutant des ϵ-transitions de cet état vers les (anciens) états initiaux, et depuis les (anciens) états finaux

4.7 Élimination des ϵ-transitions

Pour tout automate avec ϵ-transitions, on peut construire un automate équivalent qui n'en a pas. Méthode
— pour chaque état q, on détermine l'ensemble $S(q)$ de ses successeurs, qui peuvent être atteints par une ou plusieurs ϵ-transitions.
— pour chaque transition "normale" qui mène à q, on ajoute une transition similaire (même origine, même lettre), vers tous les états de $S(q)$

— pour chaque transition "normale" qui part de $q' \in S(q)$, on ajoute une transition similaire partant de q (même lettre, même destination)
— si q est un état initial, tous les successeurs deviennent initiaux
— si au moins un des successeurs de q est final, q devient final.

Cette méthode permet d'éliminer les "raccourcis" que sont les ϵ-transitions.

Application : automate qui reconnaît si une ligne contient une suite de nombres entiers (éventuellement aucun) Les entiers peuvent avoir des signes, ils sont séparés par des espaces (au moins un). Il peut y avoir des espaces en début et en fin de ligne.

4.8 Conséquence : le théorème de Kleene

Ce théorème célèbre dit que l'ensemble des langages rationnels, qui sont définis à partir des singletons (mots d'une lettre), et des unions, produits et étoiles de langages rationnels, coïncide avec l'ensemble des langages reconnus par des automates finis.

Nous l'avons démontré par petits bouts, en introduisant des notions qui facilitent la preuve : automates non-déterministes, ϵ-transitions, mais qui peuvent toujours se ramener aux automates déterministes.

En le combinant le théorème de Kleene avec d'autres propriétés que nous avons déjà remarquées (par exemple l'intersection et la différence de deux rationnels est rationnelle), cela fournit des arguments pour montrer qu'un langage est rationnel (ou pas).

Exemple : le langage L des mots sur $A = \{a, b\}$ qui n'ont pas le même nombre de a que de b est-il rationnel ? deux lettres.

Raisonnement :

1. si il l'était, son complémentaire L_1, qui contient les mots qui ont autant de a que de b, le serait aussi.
2. L'intersection de L_1 avec un autre rationnel le serait aussi. Soit $L_2 = L_1 \cap a^*b^*$
3. or $L_2 = \{a^n b^n, n \in \mathbb{N}\}$, le fameux exemple de langage qui n'est pas rationnel.
4. donc L n'est pas rationnel.

Exercice : montrer que le "langage des parenthèses" n'est pas rationnel. Exemple de mot de ce langage : "`(()())()((()))`". En tout il y a autant d'ouvrantes que de fermantes, et dans un préfixe il ne peut y avoir moins de fermantes que d'ouvrantes.

4.9 Quelques autres propriétés

Une application du théorème de Kleene, c'est que pour prouver une propriété des langages rationnels, on peut se ramener à des opérations sur les automates qui les reconnaissent.

Exemples :

- Le **miroir** $\widetilde{w}$ d'un mot $w_1 w_2 \dots w_n$ s'obtient en inversant les lettres : $\widetilde{w} = w_n \dots w_2 w_1$. Comment montrer que le miroir d'un langage rationnel est lui-même rationnel ?
- Appelons $\gamma_x(w)$ le mot obtenu en effaçant de w toutes les apparitions de la lettre x. Par exemple $\gamma_a aabcabbac = bcbbc$. Comment montrer que pour tout langage rationnel L et pour toute lettre x, $\gamma_x(L)$ est rationnel ?

Application : on considère les expressions arithmétiques bien parenthésées, du genre "`3*(x-5)+1/(y*(z-3)`". Est-ce un langage rationnel ?

4.10 Cas pratiques

1. Comment analyser des fichiers de configuration qui ressemblent à ceci :

```
;
; config
;
[database]
name = "mabase"
type = "mysql"
port = 2134
login = toto     ; à changer

[images]
directory = "/usr/local/images"
...
```


donnez-en une description formelle complète.

2. Même question pour le langage de description d'automates ci-dessous

```
automate Premier
état ZERO initial
       a -> ZERO    b -> ZERO UN
état UN final

automate Second
état ZERO initial final
       a -> ZERO UN
       b-> UN
état UN
      a -> ZERO
```

5 Expressions régulières

5.1 Notion

Un exemple : $ab(c^*) + (a+b)^*c$

Les **expressions régulières** sont un moyen de noter des langages en partant

— de singletons comme a, b, c et du mot vide ;

— de l'étoile, du produit et de l'union (notée $+$) de langages.

. Les parenthèses servent à désambiguïser. Comme vous le savez, ces opérations suffisent pour noter tous les langages rationnels.

Les expressions régulières sont aussi utilisées comme outil de programmation (bibliothèque *regex*) pour valider des chaînes de caractères, ou les découper en morceaux.

5.2 De l'expression à l'automate

La construction d'un automate (non déterministe) à partir de l'expression régulière ne pose pas de difficulté.

Exercice : application à l'exemple.

Elle peut d'ailleurs facilement être automatisée.

5.3 De l'automate à l'expression régulière

Beaucoup plus amusant : à partir d'un automate, on peut construire une expression du langage, en utilisant le *lemme d'Arden* (voir plus loin).

Rappelons que le langage reconnu par un automate, ce sont les mots qui mènent d'un état initial à un état final.

A chaque état q_i (on suppose qu'ils sont numérotés), on associe le langage L_i des mots qui mènent de l'état q_i à un état final. L'automate reconnaît donc l'union des L_i, pour les $q_i \in Q_F$.

Observons les chemins qui partent d'un état q_i

— si q_i est final, alors $\epsilon \in L_i$

— si il y a une transition x de q_i à q_j, alors $L_i \supseteq xL_j$: parmi les chemins qui vont de q_i à un état terminal, il y a ceux qui partent de q_j, après un premier pas x.

Plus précisément on a une équation pour chaque état final

$$L_i = \bigcup_{(q_i,x,q_j)\in\Delta} xL_j \cup \{\epsilon\}$$

et pour chaque état non final

$$L_i = \bigcup_{(q_i,x,q_j)\in\Delta} xL_j$$

Ceci permet de traduire un automate non-déterministe en système d'équations portant sur les langages

Exercice : de quel automate viennent ces équations ?

$$\begin{aligned} L = L_1 &= (a+b)L_1 + bL_2 \\ L_2 &= bL_2 + \epsilon \end{aligned}$$

Pour résoudre ce système, on applique les bonnes vieilles méthodes de substitution et d'élimination.

Intuitivement (et en regardant l'automate), L_2 est une suite, éventuellement vide de b. Autrement dit $L_2 = b^*$.

Donc en substituant :

$$L_1 = (a+b)L_1 + bb^*$$

et là encore, intuitivement, L_1 est une répétition de a ou de b suivie par une répétition d'au moins un b.

$$L_1 = (a+b)^* + bb^*$$

pour ce faire nous avons utilisé un petit résultat : le lemme d'Arden

5.4 Résultat : le lemme d'Arden

Lemme : soient B et C deux langages rationnels : la plus petite solution de l'équation

$$X = BX + C$$

est le langage $X = B^*A$. Et c'est l'unique solution si $\epsilon \notin B$.

Remarque. Que se passe-t-il quand $\epsilon \in B$, par exemple le cas particulier de l'équation $X = X + C$? Alors dans ce cas il n'y a pas *une seule* solution : tout ensemble est solution si et seulement si il contient C. À partir du moment où $\epsilon \in B$, l'ensemble A^* de tous les mots est également solution. En mathématiques, dans ce type de situation, on recherche - si c'est possible - la plus petite (ou la plus grande) solution.

Preuve

— B^*C est toujours solution de l'équation parce qu'on a la propriété $L^* = LL^* + \epsilon$ pour tout langage L. En remplaçant en partie droite,

$$\begin{aligned} BX + C &= B(B^*C) + C \\ &= (BB^*)C + \epsilon C \\ &= (BB^* + \epsilon)C \\ &= B^*C \end{aligned}$$

— on montre très facilement que toutes les solutions contiennent $B^*C = B^0C + B^1C + B^2C + \ldots$.
Soit S tel que $S = BS+C$. Alors en remplaçant successivement

$$\begin{aligned} S &= BS + C \\ = B^1S + B^0C &= B^1(BS + C) + B^0C \\ = B^2S + B^1C + B^0C &= B^2(BS + C) + B^1C + B^0C \\ = B^3S + B^2C + B^1C + B^0C &= \ldots \end{aligned}$$

Donc S contient tous les ensembles B^nC et donc B^*C qui est leur union.

— si B ne contient pas ϵ, tout mot de B a au moins une lettre, et un mot w de longueur n ne peut pas appartenir à $B^{n+1}L$ où L est un langage quelconque.
Or une solution S se "déplie" en

$$S = B^{n+1}S + B^nC + B^{n-1}C + \ldots + B^0C$$

et si w de longueur n appartient à S, il appartient forcément à un des B^kC, et donc à B^*C qui est leur union pour tout k entier. Donc tout mot w de S appartient à B^*C, qui est contenu (voir plus haut) dans toutes les solutions : $S = B^*C$, la solution est unique.

Troisième partie

Langages algébriques

6 Grammaires algébriques

6.1 Définitions

Une **grammaire algébrique** (ou *context-free*, hors-contexte, etc) $\mathcal{G}$ est la donnée

— d'un ensemble fini V_T de symboles terminaux (similaire à l'alphabet),

— d'un ensemble fini V_N de symboles non-terminaux (notés par des majuscules), dont on distingue un élément particulier, l'**axiome** S ;

— d'un ensemble de **règles de production**, qui sont des paires formées d'un non-terminal et et d'une suite de terminaux et de non-terminaux.

Exemple

$$\begin{aligned} S &\to aSSb \\ S &\to T \\ T &\to cT \\ T &\to \epsilon \end{aligned}$$

Dérivation directe. On dit qu'un mot (sur $V_T \cup V_N$, donc une suite de symboles terminaux ou pas) **dérive directement** d'un autre si on peut l'obtenir en remplaçant un de ses non-terminaux par la partie droite d'une de ses règles de production. Par exemple $aaaSbb$ dérive directement de $aaSb$.

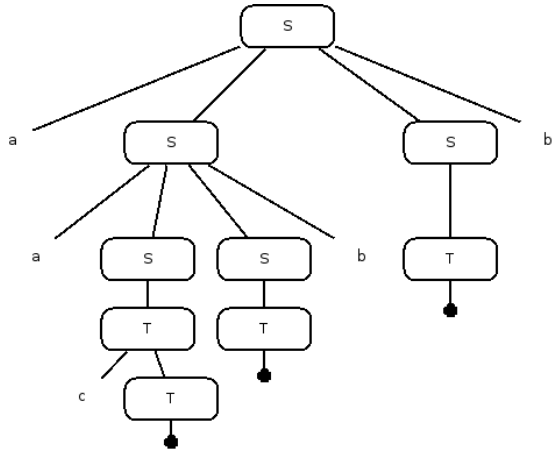
Plus généralement, on parle de dérivation quand on peut obtenir un mot après une suite de dérivations directes.

Le langage reconnu par $\mathcal{G}$ est l'ensemble des mots qui dérivent de l'axiome.

Exemple : montrez que $aaccbb \in L(\mathcal{G})$

Solution : $S \to aSSb \to aaSSbSb \to \ldots$

Une autre façon de voir est de construire l'**arbre de dérivation**, dont les feuilles correspondent à des terminaux, et les noeuds internes à des non-terminaux.



Une grammaire est **ambiguë** si il existe au moins un mot qui peut être obtenu de différentes façons.

Exercice : montrez que la grammaire de l'exemple est ambiguë.

L'ambiguïté peut être une source de problèmes. Par exemple pour l'analyse de expressions arithmétiques. La grammaire

$$\begin{aligned} E &\rightarrow E+E \\ &\rightarrow E-E \\ &\rightarrow E*E \\ &\rightarrow E/E \\ &\rightarrow (E) \\ &\rightarrow nombre \end{aligned}$$

reconnaît les expressions arithmétiques correctes, mais fournit deux arbres de dérivation pour "$1+2*3$", dont un qui ne correspond pas aux priorités habituelles.

7 Langages algébriques

Un langage est algébrique si il existe au moins une grammaire context-free qui le reconnaît.

Exercices : donnez des grammaires, si possible non-ambiguës, pour les les langages suivants :

1. $\{a^n b^n, n \geq 0\}$
2. mots qui ont autant de a que de b
3. systèmes de parenthèses
4. expression arithmétique

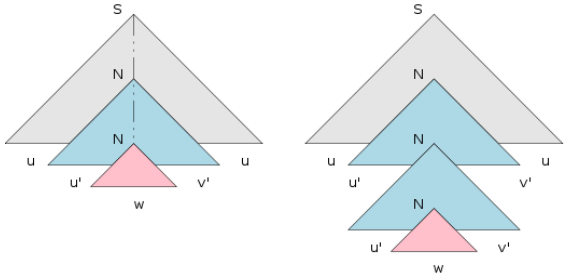
Remarques

— Il existe des langages non-algébriques, comme $\{a^n b^n c^n, n \geq 0\}$

— il existe des langages algébriques intrinséquement ambigus, c'est-à-dire qui reconnus par des grammaires qui sont toutes ambigües. Exemple $\{a^n b^m c^p, n = m$ ou $m = p\}$

Une idée de la preuve pour le premier résultat :

1. on peut transformer toute grammaire en une grammaire équivalente dont l'axiome n'apparaît jamais en partie droite d'une règle (il suffit de définir dont dérive l'ancien axiome) et dont aucun non-terminal, sauf éventuellement l'axiome, ne produit le mot vide (on les remplace).
2. si on prend un mot $a^n b^n c^n$ assez long, son arbre de dérivation aura une branche de taille supérieure au nombre de non-terminaux, et il y aura donc un non-terminal N qui apparaît deux fois sur une branche. De l'axiome S dérive dont un mot uNv (u et v suite de terminaux, qui ne sont pas tous les deux vides), d'où dérive encore $uu'Nv'v$ qui produit $uu'wv'u$ qui appartient au langage.



3. on peut donc construire un autre mot dérivé en répétant la seconde étape, c'est $uu'u'wv'v'v$, dont il est facile de voir qu'il n'appartient pas au langage : si u' (ou v') contient deux types de lettres, $u'u'$ contient une alternance de lettres. si u' et v'

sont des répétitions d'une même lettre, il y aura forcément un déséquilibre avec le nombre d'occurrences de la troisième lettre.

8 Automates à pile

De la même manière que les langages rationnels sont reconnus par les automates finis, les langages algébriques le sont par les automates à pile.

Qu'est-ce qu'un **automate à pile** ?

— il possède une **pile**, qui est une suite de symboles, don un mot sur un vocabulaire de pile. Par commodité, on considère que la pile vide est représentée par un symbole particulier, appelé "fond de pile" ($\perp$).
— il possède des **états**, dont un état initial, et des états finaux.
— il a des **transitions**, qui indiquent, en fonction d'une situation donnée, c'est à dire
  — l'état courant,
  — une lettre du mot
  — le symbole qui est en sommet de pile
  la nouvelle situation
  — le nouvel état
  — par quoi il faut remplacer le sommet de la pile

Il existe plusieurs types d'automates à pile,

— mot reconnu si pile vide ou pas à la fin ?
— l'automate peut être déterministe (pour un état, une lettre, un symbole, il a une seule transition possible), ou non.

Exemple d'automate , pour reconnaître $a^n b^n$

— deux états : OK (initial) et ERREUR
— symbole de pile : c. La taille de la pile sert de compteur.
— transitions

état avant	lettre	sommet de pile	état après	à empiler
NORMAL	a	$\perp$	NORMAL	$c\perp$
NORMAL	a	c	NORMAL	cc
NORMAL	b	$\perp$	ERREUR	$\perp$
NORMAL	b	c	ERREUR	ϵ
ERREUR	a	c	ERREUR	ϵ
ERREUR	b	c	ERREUR	ϵ
ERREUR	a	$\perp$	ERREUR	$\perp$
ERREUR	b	$\perp$	ERREUR	$\perp$

Ici on dira qu'un mot est reconnu si, après avoir effectué toutes les transitions, l'état est final et la pile est vide ($\perp$).

Exercice : construire un automate pour les mots qui ont autant de a que de b. Idée : utiliser la pile comme compteur du nombre de a moins le nombre de b. Empiler des p pour une valeur positive, et des n quand c'est négatif.

9 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`

9.1 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**

9.2 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 } """
```

9.3 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
```

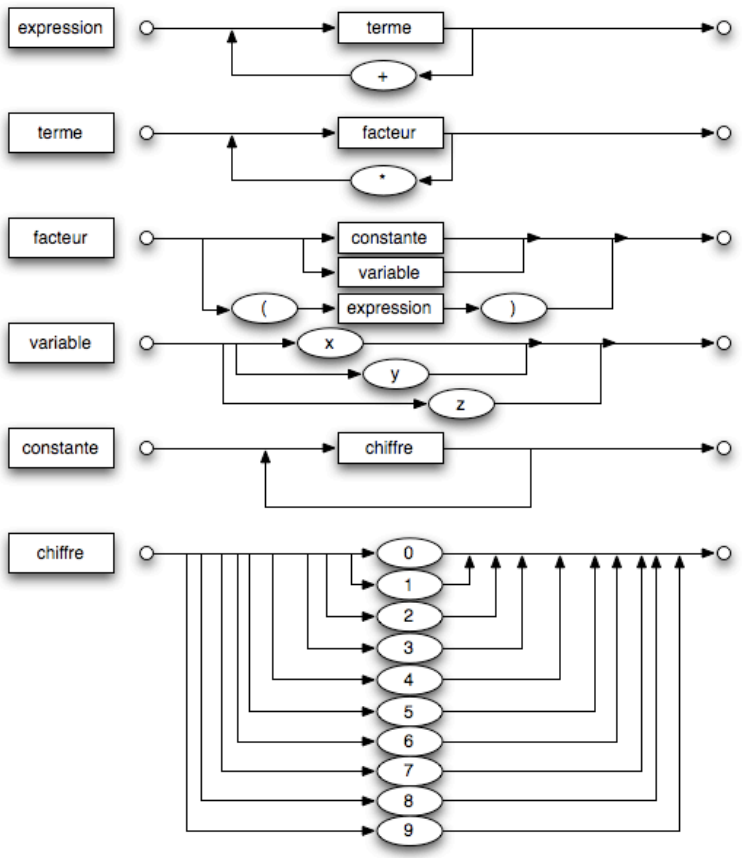
Fournir une description du langage en BNF étendue.

10 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" [2]

2. 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.

10.1 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.

$$S \rightarrow \text{expression } \$$$

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.

10.2 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.

10.3 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)

10.4 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

$$A \rightarrow BC$$

, $First(A)$ 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.

11 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
51 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()
52 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)
53 118     {};
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
54 144     {
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(;;) {
55 170             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␣",
56 196                                              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 }
```