Théorie des langages

Notes de Cours

François Yvon et Akim Demaille
with the collaboration of
of Pierre Senellart

20 09 2020
(rev. 3c06641)

Comments about reading this document

These notes document the course de théorie des langages enseigné dans le cadre de la BCI d'informatique. Despite our best efforts, and the best efforts of several generations of students who have helped us to improve the text, and efforts of A. Amarilli, this document probably still contains errors. If you still find errors, please notify `jonathan@lrde.epita.fr`.

Dans ce polycopié, sont au programme du cours (et de l'examen):

— Généralités sur les langages (chapter 2); les notions d'ordre, de quotient, de distance ne sont pas spécifiquement au programme et seront réintroduites dans l'examen si elles y sont utiles.

— Langages rationnels, expressions rationnelles, automates (chapters 3 and 4, cf. aussi TD 1, TP 1).

— Introduction aux grammaires génératives et à la hiérarchie de Chomsky (chapter 5).

— Grammaires et langages hors-contexte: définitions, arbres de dérivation, lemme de pompage pour les grammaires hors-contexte, problématique de l'analyse de grammaire (chapters 5 and 6, cf. aussi TD 2, TP 2).

— Notions de calculabilité (chapter 10); la définition formelle des modèles de calcul (machines de Turing) n'est pas au programme.

Le chapter 7 est partiellement couvert en cours mais n'est pas au programme de l'examen. Les chapters 8 and 9 ne sont pas au programme de l'examen; certains sujets d'annales font référence à ces notions (analyseurs LL ou LR) car le contenu du cours a changé depuis. Le chapter 11 (Compléments historiques) n'est pas au programme mais pourra être utilisé pendant le cours.

Chapter 1

Contents

Part I

Lecture Notes

Chapter 2

Words and Languages

The objective of this chapter is to give an introduction to the models used in computer science for describing, representing, and computing finite sequences of symbols.

Before speaking formally of these concepts in section section 2.2, we first present several application domains which illustrate the usefulness of the theory. This introduction also highlights multiple connections to sub-domains of information theory in which these concepts find their origin along side of compiler theory and formal linguistics.

2.1 Several "real" languages

2.1.1 Compilation

We call any device a compiler, if it can translate a set of written instructions in a programming language to another language (as an example, a sequence of instructions that can be run by a computer). A compiler should perform preliminary tasks such as identifying sequences of characters that can be matched to the language's keywords, to legal variable names, or to real numbers. This step is called *lexical analysis*. The inputs written with an user's keyboard are expressed as a finite sequence of characters. From a formal point of view, a lexical analyzer should be able to solve the characterization and the discrimination of finite sequences of symbols in order to split a program (given as a stream of input symbols) into coherent units that can later be classified according to their type: keywords, variables, constants...

A compiler should also be able to find syntax errors and, in order to do so, identify sequences in the set of lexical categories (keywords, variables, etc...), operators ($+, *, -, :, ...$), and auxiliary symbols ($\{, \}$, ...) that can be matched to well-formed programs. Note that a program being well-formed doesn't mean it will be bug-free or behave according to the programmer's wishes. *Syntactic analysis* ensures that arithmetic expressions and programming blocks are well-formed. Obviously, some programs are not syntactically corrects, and every programmer has experienced the subsequent warning messages generated by the compiler. The set of syntactically correct programs for a given programming language is therefore a particular subset of the set of all finite sequences generated by the language's atomic elements.

The syntactic analyzer's role goes beyond merely checking that expressions are well-formed: it should also highlight the internal structure of the sequences of symbols it reads. As an example, an arithmetic expression compiler should be able to analyze a sequence such as $Var+Var*Var$ and interpret it as $Var+(Var*Var)$ in order to perform the adequate computations.

Computer scientists must therefore deal with three major problems: define well-formed programs, discriminate syntactically correct atomic sequences, and outline the internal structure of programs in order to determine the correct sequence of instructions to be performed.

2.1.2 Bioinformatics

Molecular biology and genetic are "natural" examples that can be modeled as linear sequences of symbols on a finite alphabet.

Each chromosome that carries an individual's genetic material features two strands of DNA: if we abstract the chromosome's tridimensional structure, each strand can be modeled as a sequence of nucleotides, each nucleotide being made of sugar, a nitrogenous base, and at least one phosphate group. There are four different bases; two purine bases (guanine G and adenine A) and two pyrimidine bases (cytosine C and thymine T) that act as pairs, since adenine always binds with thymine and guanine with cytosine. The information encoded by these bases is significant enough that it is useful to model a DNA strand as a (very long) linear sequence of its nitrogenous bases encoded by a four letter alphabet (ATCG).

Using this model, one can look for particular sequences of nucleotides or compare two (or more) different strands of DNA. These comparisons may be used to quantify the genetic distance between two populations or individuals. Looking for sequences and quantifying differences are therefore two common problems in the field of bioinformatics.

These computations are not restricted to genes and may be applied to proteins as well. Indeed, a protein's structure can be modeled as a simple linear sequence of its amino-acids that determines many relevant properties of the protein. There is only a finite amount of amino-acids (20), hence, a protein can be modeled as a finite sequence on a twenty letter alphabet.

2.1.3 "Natural languages"

By *natural languages*, we simply mean the languages spoken (and often written) by human beings. There are multiple interpretations of these languages as sequences of symbols:

— The sounds articulated by human beings in order to communicate can be, despite their acoustic variability, interpreted as a linear, unidimensional sequence of symbols belonging to a finite set of phonetic transcriptions. Every sequence of sounds isn't, however, a sentence that can be understood or even vocalized.

— Writing systems universally rely on finite alphabets that can be used to represent words as linear, finite sequences of symbols. Obviously, a random sequence of letters may not be an actual word. Proper words instead belong to dictionaries [1].

1. This is not true of neologisms, proper nouns, slang words, etc. . .

— If we assume, as an approximation, that only a finite number of actual words exist for a given language, then sentences written in said language can be interpreted as linear sequences of symbols belonging to a finite set. A proper sentence has to be grammatically correct, but this is not a sufficient condition: the sentence also has to make sense.

In order to process natural languages automatically, one has to consider multiple approaches to discriminate what actually belongs to a language. This could allow a program to perform or propose corrections. Automatic language processing should also be able to identify the underlying structure of a sentence ("where is the subject?", "where is the verb?" ...) in order to check that grammar rules are properly applied ("does the inflection of the verb matches the subject?") or to understand the meaning of the input, and maybe translate it. Such problems can be modeled using the theoretical framework of language theory; indeed, significant progress have been achieved thanks to the works of formal linguists.

2.2 Terminology

2.2.1 Bases

Given a *finite* set of symbols Σ, called the *alphabet*, by *word* we mean a possibly empty finite sequence of elements of Σ. By convention, the *empty word* is denoted ε; some authors denote this word with 1 or 1_Σ. The *word length* u, denoted $|u|$, is the total number of symbols in u (each symbol being counted as many times as it appears). In particular, $|\varepsilon| = 0$. We take, $|u|_a$ to mean the total number of occurrences of symbol a in word u. Naturally: $|u| = \sum_{a\in\Sigma} |u|_a$.

The set of all words whose elements come exclusively from alphabet Σ is denoted $\Sigma^\star$. By *language* over Σ, we mean a subset of $\Sigma^\star$. One particular such subset, which we denote Σ^+, is the set of non-empty words.

The operation *concatenation* of two words $u, v \in \Sigma^\star$ results in a possibly new word uv, consisting of the juxtaposition of the symbols u followed by the symbols of symbols v. We therefore have $|uv| = |u| + |v|$ and a similar equality for the number of occurrences. The concatenation operation is associative on $\Sigma^\star$, but in general not commutative. ε is the neutral element for concatenation: $u\varepsilon = \varepsilon u = u$; thus motivating 1 and 1_Σ. By convention, we will take u^n to mean the concatenation of n copies de u, including the special case of $u^0 = \varepsilon$. If the word u can be factored into the form $u = xy$, we may write $y = x^{-1}u$ and $x = uy^{-1}$.

$\Sigma^\star$, equipped with the concatenation operation, conforms to the structure of a *monoid* (recall: a monoid is a set equipped with an associative binary operation and a neutral element. In case there is no neutral element, we refer to the set as a *semi-group*). This monoid is the *free monoid* formed by Σ: every word u can be uniquely decomposed into a concatenation of symbols of from Σ.

Several examples of languages defined under the alphabet $\Sigma = \{a, b, c\}$.

— $\Sigma^\star = \{\varepsilon, a, b, c, aa, ab, ac, ba, \ldots\}$, the set of all words composed of the letters from Σ;

— $\{\varepsilon, a, b, c\}$, the set of all words whose length is strictly less than 2;

— $\{ab, aab, abb, acb, aaab, aabb, \ldots\}$, the set of all words which start with a and end with b;

— $\{\varepsilon, ab, aabb, aaabbb, \ldots\}$, the set of all words which start with some number, n, of a's and end with the exactly as many b's. This language is denoted $\{a^n b^n \mid n \geq 0\}$;

— $\{\varepsilon, abc, aabbcc, aaabbbccc, \ldots\}$, the set of words starting with n occurrences of the letter a, followed by n occurrences of the letter b, and ending with n occurrences of the letter c. This language is denoted $\{a^n b^n c^n \mid n \geq 0\}$.

— $\{aa, aaa, aaaaa, \ldots\}$, the set of all words composed of a *prime* number of occurrences of the letter a.

There are countably many words in $\Sigma^\star$, but uncountably many languages of $\Sigma^\star$. Among these, not all are at the disposition of computer scientists: there exist, in effect, some languages which "resist" being computed, i.e., more precisely, which cannot be enumerated algorithmically.

2.2.2 Some notions of computability

Precisely, the theory of calculability introduces the following distinctions:

Definition 2.1 (Recursively enumerable language)**.** *A language L is said to be* recursively enumerable *(equivalently* semi-decidable*) if there exists an algorithm, A, which enumerates every word in L.*

In other words, a language, L, is recursively enumerable if there exists an algorithm, A, (or equivalently if there exists a Turing machine) such that any word of L is produced by a finite number of steps of the execution of A. That is to say, if L is recursively enumerable and $u \in L$, then if the algorithm A runs "long enough", the algorithm will produce u.

An equivalent characterization of recursively enumerable languages are those languages L for which there exists a Turing machine which *recognizes* all the words of L and only the words of L. This means that the machine stops in an accepting state given any word in L, and does not stop in an accepting start given a word not in L. Attention, this description does specify whether the Turing machine must stop, given a word not in L, it may well loop forever.

Definition 2.2 (Recursive language)**.** *A language, L, is said to be* recursive *(equivalently* decidable*), if there exists an algorithm A, which when given a word $u \in \Sigma^\star$, responds* true *if $u \in L$ and responds* false*, if $u \notin L$. We say, in this case, that the algorithm A* recognizes *the language L, and that A implements an* indicator *function.*

A language is recursive if it is possible to construct an algorithm for its indicator function.

Every recursive language is recursively enumerable. To construct an enumeration of L, it suffices to to take any procedure whatsoever which enumerates $\Sigma^\star$ (for example military order) and to submit each word in turn to the indicator function of L, i.e., to algorithm, A. If the response from A is *true*, we emit the current word, else we skip it and repeat for the subsequent word. This procedure effectively enumerates all the words of L. Only recursive languages have any real practical interest, because they correspond to the distinctions which are calculable between words in L and outside of L

From these definitions, we see a fundamental limitation in our knowledge as computer scientists: There exist languages which we cannot enumerate. These will be of little interest to us. Most of our attention will be focused on languages which we know hot to recognize.

Beyond the problems of recognition and decision, there are other types of computations which we will consider and which have good applications practices in the various application areas mentioned above:

— compare two words; evaluate their similarity
— find a pattern in a word
— compare two languages
— learn a language by examples
— ...

2.3 Operations on words

2.3.1 Factors and subwords

We say that u is a *subword* (or *factor*) of v if there exist $u_1, u_2 \in \Sigma^\star$ such that $v = u_1 u u_2$. If $u_1 = \varepsilon$ (resp. $u_2 = \varepsilon$), then u is said to be a *prefix* (resp. *suffix*) of v. If w can be factored into $u_1 v_1 u_2 v_2 \dots u_n v_n u_{n+1}$, where all the u_i and v_i are in $\Sigma^\star$, then $v = v_1 v_2 \dots v_n$ is said to be a *subsequence* or *scattered subword* [2] of w. As opposed to factors (subwords), subsequences are constructed from fragments (not necessarily contiguous) of the original word. Moreover, the order in which these fragments appear respects the order of the original word. When we refer to a *proper factor* of u, we mean a factor of u which is different from u; analogously for *proper prefix*, *proper suffix*, and *proper substring*.

We notice that $|pref|_k(u)$ (resp. $|suff|_k(u)$) the prefix (resp. the suffix) of length k of u. If $k \geq |u|$, then $|pref|_k(u)$ designates u itself.

The notions of prefix and suffix those in linguistics: everyone agrees with the claim that *in* is a prefix of *infinite*; but computer scientists think that the same goes for *i*, *inf* or even *infini*. In the same way, everyone agrees that *ness* is a suffix of *happiness*; but computer scientists think that also *ppiness* is another suffix of *happiness*.

A non-empty word $u \neq \varepsilon$ is said to be *primitive* if the equation $u = v^i$ has no solution when $i > 1$.

Two words, x and y, are said to be *conjugates*, if there exist words u, v such that $x = uv$ and $y = vu$. I.e., the words are derived from one another by exchanging prefix with suffix are called (of each other). It is easy to verify that conjugation (x is a conjugate of y) is an equivalence relation.

The *mirror* or *transpose* u^R of word $u = a_1 \dots a_n$, where $a_i \in \Sigma$, is defined by: $u^R = a_n \dots a_1$. A word is a *palindrome* if it is equal to its transpose. *radar*, *ablewasIereIsawelba* are palindromes.

2. French and English: *subword* (English) or *substring* (English) correspond to *facteur* (French), and *subsequence* (English) or *scattered subword* (English) correspond to *sous-mot* (French). In this document we will also use the word *factor* (English) to mean *subword* as it is unambiguous, even if non-standard.

We can verify that the prefixes of u^R are precisely the transposes of the suffixes of u and vice versa.

2.3.2 Quotient of words

Definition 2.3 (Right quotient of a word)**.** *The* right quotient *of word u by the word v, denoted, uv^{-1}, or $u_{/v}$, is defined as follows:*

$$u_{/v} = uv^{-1} = \begin{cases} w & \text{if } u = wv \\ \text{not defined} & \text{if } v \text{ is not a suffix of } u \end{cases}$$

For example, $abcde(cde)^{-1} = ab$; and $abd(abc)^{-1}$ is not defined. The notation uv^{-1} should not be seen as a product because v^{-1} does not represent a word. In similar fashion we can define the left product of v by u as $u^{-1}v$ or ${}_{u\backslash}v$.

2.3.3 Orders on words

A family of partial orders

The relations prefix, suffix, factor, and subsequence each introduce a *order relation* on $\Sigma^\star$: these are, in effect, relations which are reflexive, transitive, and antisymmetric. For example, we may say that $u \preceq_p v$, if u is a prefix of v. Two arbitrary words are not necessarily comparable with respect to these relations; thus the relations are *partial orders*.

Total orders

It is possible to define orders *total* on $\Sigma^\star$, if we have a total order $\leq$ on Σ at our disposal.

Definition 2.4 (Lexicographic order)**.** *The* lexicographic order *on $\Sigma^\star$, denoted $\leq_l$, is defined by $u \leq_l v$ if and only if*

- *either u is a prefix of v,*
- *or if not, $u = tu'$, $v = tv'$ with $u' \neq \varepsilon$ and $v' \neq \varepsilon$, and the first symbol in u' precedes that of v' with respect to $\leq$.*

This order leads to counterintuitive results when we deal with infinite languages. For example, there are infinitely many predecessors of the word b in $\{a, b\}^\star$: $\{\varepsilon, a, aa, ab, aaa, \ldots\}$.

Military order (or *alphabetical order*) likewise uses $\leq$ defined on Σ, gives priority to length of words.

Definition 2.5 (Military order)**.** *The* military order *on $\Sigma^\star$ denoted $\leq_a$ is defined by $u \leq_a v$ if and only if*

- *either $|u| \leq |v|$,*
- *or $|u| = |v|$ and $u \leq_l v$*

When compared to lexicographic order, this is a *well-founded order*: there are finitely many words smaller than any given word u. Moreover, for every w, w', if $u \leq_a v$ then $wuw' \leq_a wvw'$, which is not the case with lexicographic order (e.g.: $a \leq_l ab$, but $c \cdot a \cdot d >_l c \cdot ab \cdot d$).

We have noted above that we sometimes use the term *alphabetical order* to refer to military order. To avoid confusion, keep in mind that by our definitions, natural language dictionaries use lexicographic order, not alphabetical order.

2.3.4 Distances between words

A family of distances

For any pair of words, there exists a longest common prefix (resp. suffix, factor, subsequence). For the case of suffixes and prefixes, the longest factor is unique.

If we denote the longest prefix common to u and v as $\text{lcp}(u, v)$, then the function $d_p(u, v)$ defined by:

$$d_p(u, v) = |uv| - 2|\text{lcp}(u, v)|$$

defines a distance in $\Sigma^\star$, the *prefix distance*. We can verify that:

— $d_p(u, v) \geq 0$

— $d_p(u, v) = 0 \Leftrightarrow u = v$

— $d_p(u, w) \leq d_p(u, v) + d_p(v, w)$.
The verification of this inequality uses the fact that the longest prefix common to u and w is also the longest prefix common to $\text{lcp}(u, v)$ and $\text{lcp}(v, w)$.

Rather than considering prefixes, equally valid would be a distance defined by longest common suffix (d_s), longest factors (d_f), or longest common subsequence (d_m).

Proof. In each case the only property that is difficult to prove is the triangle inequality. In the case of suffixes, it is proven the same was as for prefixes.

For the case of factors and subsequences, it is useful to consider the words from a slightly different perspective. It is possible to envision a word, $u \in \Sigma^+$, as a *function* from the interval $I = [1 \dots |u|]$ to Σ, which associates each integer, i, in the interval with the i^{th} symbol of u: $u(i) = u_i$. Every strictly increasing sequence of indices corresponds to a subsequence of u; if the indices are consecutive we obtain a factor.

In the following, we consider subsequences and note that $lcss(u, v)$, the longest sub sequence common to u and v.

Showing $d_m(u, w) \leq d_m(u, v) + d_m(v, w)$ amounts to showing:

$$|uw| - 2|lcss(u, w)| \leq |uv| - 2|lcss(u, v)| + |vw| - 2|lcss(v, w)|$$

which amounts to showing:

$$|lcss(u, v)| + |lcss(v, w)| \leq |v| + |lcss(u, w)|$$

Letting I and J be subsequences of indices of $[1 \dots |v|]$ corresponding respectively to $lcss(u, v)$ and $lcss(v, w)$, we notice that:

$$\begin{aligned} |lcss(u,v)| + |lcss(v,w)| &= |I| + |J| \\ &= |I \cup J| + |I \cap J| \end{aligned}$$

We further notice that the subsequence of v constructed by considering the symbols at positions $I \cap J$ is a subsequence of both u and w, thus necessarily of the longest subsequence of $lcss(u, w)$. From this we conclude that: $|I \cap J| \leq |lcss(u, w)|$. Since $|I \cup J| \leq |v|$, we can conclude that:

$$|lcss(u,w)| + |v| \geq |I \cup J| + |I \cap J|$$

We obtain precisely what we wished to prove. The case of factors can be treated in a similar manner. □

Edit distance and variations

Another commonly used distance on $\Sigma^\star$ is the so-called *edit distance*, or *Levenshtein distance*, defined as being the smallest number of elementary editing operations necessary to transform word u into word v. The elementary editing operations are deletion and insertion of a symbol. As an example we see that the distance from *chien* to *chameau* (French for *dog* and *camel*) is no more than 6, because we can transform *chien* to *chameau* by the following set of consecutive operations: delete i, insert a, insert m, delete n, insert a, insert u. This metamorphosis from one word to another is decomposed in table 2.1.

current word	operation
chien	delete i
chen	insert a
chaen	insert m
chamen	delete n
chame	insert a
chamea	insert u
chameau	

Two consecutive words are of distance 1 from each other. The order of elementary operations is arbitrary.

Table 2.1 – Metamorphosis from *chien* to *chameau*

Multiple variants of this notion of distance have been proposed, which use different sets of operations and considering different weights for the different operations. Considering a real example, if we wish to implement an application for "correcting" typographical errors (errors made while typing on a keyboard), it is useful to consider weights which take into account the higher probability of fingers touching keys which are close to the desired letters. We consider also, for example, that *camel* is a better correction for *canel* than is *caxel* because the distance on the keyboard from m to n is closer than the distance from x to n,[3], even if both words are transformations of *canel* by a sequence of two elementary operations.

3. This is a first approximation. To do better we could also take into account the relative frequency of proposed words. But this is better than nothing

The Unix `diff` utility uses a form of distance calculation. This utility makes it possible to compare of two files and prints all the differences between their respective contents.

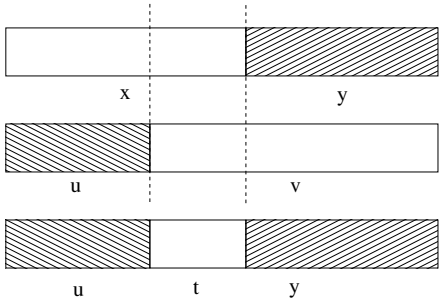
2.3.5 Some elementary combinatorial results

The following property is respected trivially:

Lemma 2.6.

$\forall u, v, x, y \in \Sigma^\star, uv = xy \Rightarrow \exists t \in \Sigma^\star$ *such that either* $u = xt$ *and* $tv = y$, *or* $x = ut$ *and* $v = ty$.

This property is illustrated in the following figure.



This result is used for proving two other elementary results which connect non-commutativity and concatenation.

Theorem 2.7. *If* $xy = yz$, *with* $x \neq \varepsilon$, *then* $\exists u, v \in \Sigma^\star$ *and an integer* $k \geq 0$ *such that:* $x = uv$, $y = (uv)^k u = u(vu)^k$, $z = vu$.

Proof. If $|x| \geq |y|$, then the preceding result allows us to directly write $x = yt$, which identifies u with y, and v with t, and allows us to directly derive the desired equalities for $k = 0$.

The case when $|x| < |y|$ can be treated by induction on the length of y. The case when $|y| = 1$ being immediate, if we suppose the relation is true for all y whose length is less than n, and considering y with $|y| = n + 1$. There therefore exists a t such that $y = xt$, from which we can derive $xtz = xxt$, or still $tz = xt$, with $|t| \leq n$. The inductive hypothesis guarantees the existence of u and v such that $x = uv$ and $t = (uv)^k u$, for which $y = uv(uv)^k u = (uv)^{k+1} u$. □

Theorem 2.8. *If* $xy = yx$, *with* $x \neq \varepsilon, y \neq \varepsilon$, *then* $\exists u \in \Sigma^\star$ *and two indices* i *and* j *such that* $x = u^i$ *and* $y = u^j$.

Proof. These results can again be obtained by induction on the length of xy. For a length of 2, the result follows trivially. Suppose it holds for lengths up to an including n, and consider xy of length $n + 1$. By the previous theorem, there exist u and v such that $x = uv$, $y = (uv)^k u$, from which we deduce: $uv(uv)^k u = (uv)^k uuv$, thus $uv = vu$. Using the inductive hypothesis we have: $u = t^i$, $v = t^j$, and then $x = t^{i+j}$ and $y = t^{i+k(i+j)}$, which is the result we are looking for. □

The interpretation of these results is that equations of the form $xy = yx$ only produce *periodic* results: that is to say, sequences which are constructed by repeating the same base pattern.

2.4 Operations on languages

2.4.1 Boolean set operations

Languages being sets, all classical Boolean set operations are applicable. Among these are union, intersection, and complementation (relative to $\Sigma^\star$) a defined for $L, L_1, L_2 \subseteq \Sigma^\star$ by:

$$\begin{aligned} L_1 \cup L_2 &= \{u \in \Sigma^\star \mid u \in L_1 \text{ or } u \in L_2\} \\ L_1 \cap L_2 &= \{u \in \Sigma^\star \mid u \in L_1 \text{ and } u \in L_2\} \\ \overline{L} &= \{u \in \Sigma^\star \mid u \notin L\} \end{aligned}$$

The operations $\cup$ and $\cap$ are associative and commutative.

2.4.2 Concatenation, Kleene Star

The operation of concatenation defined between words extends naturally to the *concatenation of languages* (we also say the *product of languages*, but this is not a Cartesian product):

$$L_1 L_2 = \{u \in \Sigma^\star \mid \exists (x, y) \in L_1 \times L_2 \text{ such that } u = xy\}$$

We again remark that this operation is associative but it is not commutative. As before, the concatenation of n copies of the language L is denoted L^n, with, by convention: $L^0 = \{\varepsilon\}$. Attention: do not confuse L^n with the language containing the n^{th} powers of the words in L which would be defined by $\{u \in \Sigma^\star \mid \exists v \in L, u = v^n\}$.

The operation of *Kleene closure* (or simply *star*) of the language L is defined as:

$$L^\star = \bigcup_{i \geq 0} L^i = \{w \in \Sigma^\star \mid \exists n \geq 0 \text{ such that } w \in L^n\}$$

$L^\star$ contains all the words which are possible to construct by concatenating a finite number (possibly zero) of elements of L. We note that if Σ is an alphabet, $\Sigma^\star$, such as already defined, represents [4] the set of finite sequences constructed by concatenating symbols of Σ. We also remark that by definition, $\emptyset^\star$ is not empty because it contains ε.

We also define

$$L^+ = \bigcup_{i \geq 1} L^i = \{w \in \Sigma^\star \mid \exists n < 0 \text{ such that } w \in L^n\}$$

4. Note that so doing is an abuse of notation. Σ represents both the language of singleton words, and also the alphabet.

The distinction between $L^\star$ and L^+ is that $\varepsilon \in L^\star$, but $\varepsilon \in L^+$ only if $\varepsilon \in L$. We also have that $L^+ = LL^\star$.

One advantage of this notation is that they make it possible for us to formally and compactly express complex potentially infinite languages in terms of simpler languages. The set of sequences containing 0 and 1 which also contain 111 can be expressed for example: $\{0,1\}^\star\{111\}\{0,1\}^\star$. The notation $\{0,1\}^\star$ allows for arbitrarily many 0 et 1 before the sequence 111. The notion of rational expression, introduced in chapter 3, further develops this intuition.

2.4.3 Other operations in $\mathcal{P}(\Sigma^\star)$

For a language $L \subseteq \Sigma^\star$, we define the following concepts:

Definition 2.9 (Language of prefixes)**.** *Let $L \subseteq \Sigma^\star$, we define the* language of prefixes *of L, denoted,* Pref(L)*, by:*

$$\text{Pref}(L) = \{u \in \Sigma^\star \mid \exists v \in \Sigma^\star, uv \in L\}$$

Attention not to confuse this notion with a *prefix language*. We say that L is a *prefix language* if for all $u, v, \in L$, $u \neq v$, we have $u \notin \text{Pref}(v)$. By using a finite prefix language, it is possible to define an encoding giving rise to *prefix codes*. In particular, codes produced by Huffman codes are prefix codes.

Exercise 2.10. *Small application of the concept: show that the product of two prefix languages is again a prefix language.*

Definition 2.11 (Language of suffixes)**.** *Let $L \subseteq \Sigma^\star$, we define the* language of suffixes *of L, denoted* Suff(L) *by:*

$$\text{Suff}(L) = \{u \in \Sigma^\star \mid \exists v \in \Sigma^\star, vu \in L\}$$

Definition 2.12 (Language of factors)**.** *Let $L \subseteq \Sigma^\star$, we define the* language of factors *of L, denoted $Fac(L)$ by:*

$$Fac(L) = \{v \in \Sigma^\star \mid \exists u, w \in \Sigma^\star, uvw \in L\}$$

The left and right quotients of words extend additively to languages.

Definition 2.13 (Right quotient of a language)**.** *The* right quotient of a language *L by the word u is defined as:*

$$L_{/u} = Lu^{-1} = \bigcup_{v \in L} \{vu^{-1}\} = \{w \in \Sigma^\star \mid wu \in L\}$$

The right quotient of L by u is therefore the set of words $w \in \Sigma^\star$ for which $wu \in L$. Likewise, the left quotient of L by u, $u^{-1}L =_{u\setminus} L$, is the set of words $w \in \Sigma^\star$ for which $uw \in L$.

Definition 2.14 (Right congruence)**.** *A* right congruence *of $\Sigma^\star$ is a relation $\mathcal{R}$ of $\Sigma^\star$ which verifies :*

$$\forall w, w' \in \Sigma^\star, w \mathcal{R} w' \Rightarrow \forall u, wu \mathcal{R} w'u$$

Definition 2.15 (Right congruence associated with a language L)**.** *Let L be a language et let w_1, w_2 two words such that $L_{/w_1} = L_{/w_2}$. It is clear, by definition of quotient, that we therefore have $\forall u, L_{/w_1 u} = L_{/w_2 u}$. This deduces a right congruence $\mathcal{R}_L$ "naturally" associated with L and defined by:*

$$w_1 \mathcal{R}_L w_2 \Leftrightarrow L_{/w_1} = L_{/w_2}$$

2.4.4 Morphisms

Definition 2.16 (Morphism)**.** *A* morphism *of a monoid M into monoid N is a function ϕ such that:*

— $\phi(\varepsilon_M) = \varepsilon_N$*: the image of the neutral element of M is the neutral element of N.*

— $\phi(uv) = \phi(u)\phi(v)$

The length function is a morphism of the monoid $(\Sigma^\star, .)$ into $(N, +)$: we have trivially $|\varepsilon| = 0$ et $|uv| = |u| + |v|$. We simply verify that this also holds for the occurrence counting functions.

Definition 2.17 (Code)**.** *Un code is an injective (onto) morphism: $\phi(u) = \phi(v)$ follows from $u = v$.*

Chapter 3

Languages and rational expressions

We introduce the family of *rational languages*. This family consists in particular of all the finite languages, but also a number of infinite languages. The characteristic of all these languages is the possibility to *describe* them by very simple formulas (we also say *patterns*, *motifs* in French). The use of these formulas, known as rational expressions, [1] is generally considered a "good" way to describe patterns representing sets of words.

After introducing the principal formal concepts (see section 3.1), we will student several classic systems which apply the concepts in applications.

The concepts introduced in this and the next chapter are illustrated by using Vcsn (`http://vcsn.lrde.epita.fr`). This software implements a platform for manipulating automata (state machines) and rational expressions. The software application is a result of collaboration between Télécom ParisTech, Laboratoire Bordelais d'Informatique (LaBRI), and Laboratoire de R&D de l'EPITA (LRDE). The documentation is available online at: `http://vcsn-sandbox.lrde.epita.fr/notebooks/Doc/index.ipynb`.

The shell command '`vcsn notebook`' opens an interactive session using IPython. By default, it is available online at: `http://vcsn-sandbox.lrde.epita.fr`.

Create a new page, or open an existing page. In this environment ENTER inserts a newline, and SHIFT-ENTER launches the evaluator on the current cell, and advances the cursor to the next cell. Once the interactive session is launched, enter the Python command: '`import vcsn`'.

1. The literature also uses the term *regular expression*. We consider this term misleading, and we avoid it in this course.

3.1 Rationality

3.1.1 Rational Languages

Among the operations defined on $\mathcal{P}(\Sigma^\star)$ in section 2.4, three qualify as *rational* : union, concatenation, and star. *On the contrary*, note that We have not mentioned whether complement and intersection are not rational operations, and for now we will not many any assumption regarding this. This distinction makes it possible for us to define an important family of languages: The *rational* languages.

Definition 3.1 (Rational language)**.** *Let Σ be an alphabet. The* rational languages *on (over) Σ are defined inductively as follows:*

(i) $\{\varepsilon\}$ *and* $\emptyset$ *are rational languages*

(ii) $\forall a \in \Sigma, \{a\}$ *is a rational language,* singleton languages

(iii) if L_1 and L_2 are rational languages, then $L_1 \cup L_2$, $L_1 L_2$, and ${L_1}^\star$ are also rational languages.

In this inductive definition we may apply operators in (iii) finitely many times.

All finite languages are rational, because they can be expressed as finitely many applications of union and concatenation of singleton languages. By definition, the set of rational languages is the closure of these three rational operations. We way that the set is rationally closed.

The family (set) of rational languages corresponds precisely to the smallest set of languages which (i) contain the finite languages, and (ii) is rationally closed.

Any rational language may be decomposed to the form of a finite formula, corresponding to rational operations which designate its construction. Take, for example, the language over $\{0, 1\}$ containing all the words in which the factor 111 appears at least once. The language may be designated by: $(\{0\} \cup \{1\})^\star\{1\}\{1\}\{1\}(\{0\} \cup \{1\})^\star$, signifying that the words of the language are constructed by taking any two words from $\Sigma^\star$ and inserting the word 111 between them: we can deduce that such a language is rational. *Rational expressions* define a system of formulas which simplify and extend this type of notation of the rational languages.

3.1.2 Rational Expressions

Definition 3.2 (Rational expression)**.** *Let Σ be an alphabet. The* rational expressions *(RE) on (over) Σ are defined inductively by:*

(i) ε *and* $\emptyset$ *are rational expressions*

(ii) $\forall a \in \Sigma, a$ *is a rational expression*

(iii) if e_1 and e_2 are two rational expressions, then $(e_1 + e_2)$, $(e_1 e_2)$, and $({e_1}^\star)$ are also likewise rational expressions.

A rational expression is therefore any formula constructed by *finitely* many applications application of step (iii).

Let's illustrate this new concept by taking the characters from the French (in particular including the acute accented *é*) as the set of symbols, i.e., the alphabet:

— $r, e, d, é$, are RE (by (ii))
— (re) and $(dé)$ are RE (by (iii))
— $((((fa)i)r)e)$ is an RE (by (ii) and (iii))
— $((re) + (dé))$ is an RE (by (iii))
— $((((re) + (dé)))^\star)$ is an RE (by (iii))
— $(((((re) + (dé)))^\star)((((fa)i)r)e))$ is an RE (by (iii))
— ...

How are these formulas useful? As previously claimed, they serve to denote rational languages. The interpretation (the semantics) of an expression is defined by the following inductive rules:

(i) ε designates the language $\{\varepsilon\}$ and $\emptyset$ designates the empty language.
(ii) $\forall a \in \Sigma$, a designates the singleton language $\{a\}$.
(iii.1) $(e_1 + e_2)$ designates the union of the languages designated by e_1 and e_2 respectively.
(iii.2) $(e_1 e_2)$ designates the concatenation of the languages designated by e_1 and e_2 respectively.
(iii.3) $(e^\star)$ designates the star of the language designated by e.

For lessening the burden of the notation (and limit the number of parentheses), we impose the following precedence rules: the star ($\star$) is the highest priority operator, then comes concatenation ($\cdot$), then union ($+$). The binary operators are left-associative. As such, $aa^\star + b^\star$ is interpreted the same as $((a(a^\star)) + (b^\star))$, and $(((((re) + (dé)))^\star)((((fa)i)r)e))$ can be written $(re + dé)^\star faire$.

Returning to the preceding formula $(re + dé)^\star faire$ designating the set of words formed by freely iterating the two prefixes *re* and *dé*, and then concatenating the suffix *faire*: this set describes a set of potentially existing words in the French language which are formed by regular prefixing of verbs.

By construction, rational expression makes possible the designation of precisely all the rational languages and nothing more. If a language is rational, then there exists at least one rational expression which designates it.

TODO TODO TODO TODO TODO TODO TODO TODO TODO TODO: Ceci se montre par une simple récurrence sur le nombre d'opérations rationnelles utilisées pour construire le language.

Conversely, if a language is denoted by a rational expression, then it is itself rational (by induction over the sets in the number of steps in the expression). This final point is important, for it provides our first method for *proving* that a language is rational: it suffices to exhibit a rational expression which designates the language.


Vcsn works through rational expressions (and automata) of types more general. This concept of type is called "context" in Vcsn. We use the simplest context, '`lal_char(a-z), b`', which designates expressions over the alphabet $\{a, b, \ldots, z\}$ which "compute" a Boolean ($\mathbb{B}$). The cryptic '`lal_char`' signifies that the symbols are letters and that a letter is a simple character ('`char`').

We define this context.

```
>>> import vcsn
>>> ctx = vcsn.context('lal_char(abc), b')
```

Then construction several rational expressions.

```
>>> ctx.expression('(a+b+c)*')
>>> ctx.expression('(a+b+c)*abc(a+b+c)*')
```

The syntax of rational expressions is documented in the page Expressions.

3.1.3 Equivalence and reduction

La correspondence between expression and language not an isomorphism: each expression designates a unique language, but a given language corresponds to (infinitely) many different expressions. Case in point, the two following expressions: $a^\star(a^\star ba^\star ba^\star)^\star$ et $a^\star(ba^\star ba^\star)^\star$ are two different notations of the same language over $\Sigma = \{a, b\}$.

Definition 3.3 (equivalent rational expressions)**.** *Two rational expressions are said to be* equivalent *if they designate the same language.*

How can we automatically determine whether two rational expressions are equivalent? Does there exists a canonical corresponding to the most concise, designation of the language? This is not an innocuous question.

One suggested way to determine whether two rational expressions are equivalent is to start with the expression which seems simpler, and minimize the number of operations needed to accomplish it. To accomplish such a transformation the elementary equivalences shown in table 3.1 may be useful.

$$
\begin{array}{rcl@{\qquad}rcl}
\emptyset e & \equiv & \emptyset & \varepsilon e & \equiv & e \\
e\emptyset & \equiv & \emptyset & e\varepsilon & \equiv & e \\
\emptyset^\star & \equiv & \varepsilon & \varepsilon^\star & \equiv & \varepsilon \\
e + f & \equiv & f + e & e + \emptyset & \equiv & e \\
e + e & \equiv & e & (e^\star)^\star & \equiv & e^\star \\
e(f + g) & \equiv & ef + eg & (e + f)g & \equiv & eg + fg \\
(ef)^\star e & \equiv & e(fe)^\star & & & \\
(e + f)^\star & \equiv & e^\star(e + f)^\star & (e + f)^\star & \equiv & (e^\star + f)^\star \\
(e + f)^\star & \equiv & (e^\star f^\star)^\star & (e + f)^\star & \equiv & (e^\star f)^\star e^\star
\end{array}
$$

Table 3.1 – Rational Expression Identities

Using these identities, it becomes possible to effectuate transformations purely syntactically [2] which preserve the designated language, in particular to simplify the expression. An example of reduction obtained by application of these rules is the following:

$$\begin{aligned} bb^\star(a^\star b^\star + \varepsilon)b &\equiv b(b^\star a^\star b^\star + b^\star)b \\ &\equiv b(b^\star a^\star + \varepsilon)b^\star b \\ &\equiv b(b^\star a^\star + \varepsilon)bb^\star \end{aligned}$$


Observe certain simplifications.

```
>>> ctx.expression('(c+a+b+a)*+\z')
>>> ctx.expression('\e*')
```

It would be an exceedingly difficult project to implement an efficient algorithm to reduce a rational expression by using reductions such as those in table 3.1. For this reason the most often used approach for testing the equivalence of two given rational expression is not based on the identities. Rather, a common approach is by transforming the expressions to finite automata which will be presented in the next chapter (cf., but rather section 4.2.2).

3.2 Notational extensions

Rational expressions constitute a powerful tool for describing simple (rational) languages. The necessity to describe such languages being commonplace in computer science, these formulas are thus useful, but with many different extensions, and different tools in current use.

For example, `grep` is an tool available in UNIX for finding occurrences of a word or pattern within a text file. Its usage is simple:

```
> grep 'string' my-file
```

will print all lines of the '`my-file`' to stdout which contain an occurrence of the word `string`.

In fact, `grep`, allows us to do a bit more: rather than searching for a word, we can print all the occurrences of any word in a rational language—the language being designated by a rational expression. For example,

```
> grep 'str*ing' my-file
```

will find and print all occurrences of any word from the language $str^\star ing$ within the file '`my-file`'. Being given a pattern in the form of the rational expression e, `grep` analyzes the

2. By *syntactic transformation* we intend to rewrite the rational expressions themselves without regard to their semantics. For example, the rules to rewrite $x + 0 \rightsquigarrow x$ and $0 + x \rightsquigarrow x$ explains *syntactically* the neutrality of the value of the word 0 for the operation represented by + without recognizing the value nor the operation.

test, line by line, testing each line as to whether it belongs to the language $\Sigma^\star(e)\Sigma^\star$; the default alphabet being ASCII (or also in play, the characters such as ISO Latin 1).

The syntax of `grep` uses the characters '`*`' and '`|`' to denote respectively the $\star$ and + operators. This implies that if you with to use the explicit symbol '`*`' in a pattern, you must prevent that it be interpreted as an operator. To denote this, the special character must be escaped '`\`'. The same goes for other operators ('`|`', '`(`', '`)`')... and thus also for pour '`\`'.

The complete syntax of the `grep` command includes numerous notational extensions which makes it possible to greatly simplify regular expression at the price of `grep` having defined new *special characters*. The most important of these extensions are presented in table 3.2.

Expression	Designation	Comment
Classes of characters		
'`[abc]`'	$a+b+c$	a, b, c are the characters
'`[a-z]`'	$a+b+c+\cdots+z$	uses the order of the ASCII characters
'`[^abc]`'	$\Sigma\backslash\{a,b,c\}$	don't include the end of line `\n`
'`.`'	Σ	any symbol except `\n`
Quantifiers		
'`e?`'	$\varepsilon + e$	
'`e*`'	$e^\star$	
'`e+`'	$ee^\star$	
'`e{n}`'	e^n	
'`e{n,}`'	$e^n e^\star$	
'`e{,m}`'	$\varepsilon + e + e^2 + \cdots + e^m$	
'`e{n,m}`'	$e^n + e^{n+1} + \cdots + e^m$	only when $n \leq m$
Anchors/Predicates		
'`\<e`'	e	e must appear at the beginning of a word, i.e. preceding a word separator (space, comma, beginning of line, ...)
'`e\>`'	e	e must appear at the end of a word, i.e. after a word separator (space, comma, end of line, ...)
'`^e`'	e	e must appear at the beginning of a line
'`e$`'	e	e must appear at the end of a line
Special characters		
'`\.`'	.	
'`\*`'	*	
'`\+`'	+	
'`\n`'		denotes an end of line
...		

Table 3.2 – Definition of some patterns for `grep`

Suppose, for purpose of illustration, that we would like to count the usages of the French grammatical form *imparfait* of the *subjonctif* within a novel of Balzac, supposedly available in a large file '`Balzac.txt`'. To start, a little bit of information about this grammatical form:

which endings are possible? French verbs are divided into several groups. In the first group, the endings for *imparfait du subjonctif* are: `asse`, `asses`, `ât`, `assions`, `assiez`, `assent`. We will thus find all the forms of verbs from the first verb group with [3]:

```
> grep -E '(ât|ass(e|es|ions|iez|ent))' Balzac.txt
```

A little more difficult is the second verb group, where the endings of *imparfait du subjonctif* are: `isse`, `isses`, `ît`, `issions`, `issiez`, `issent`. We have a new pattern:

```
> grep -E '([îâ]t|[ia]ss(e|es|ions|iez|ent))' Balzac.txt
```

In the third verb group the verbs of the *imparfait du subjonctif* grammatical form have the endings: `usse` and `insse`. This brings us to the pattern

```
> grep -E '([îâû]n?t|[iau]n?ss(e|es|ions|iez|ent))' Balzac.txt
```

It turns out that this pattern is too general because it also matches endings such as `unssiez`; but for the moment we will be happy with it. To continue, we return to our initial problem: find the verbs. It is important that the verb endings appear as word suffixes in the text. How can we do this? One way is to impose that the pattern contain a punctuation mark or space after the verb ending: We may now update the pattern to:

```
> grep -E '([îâû]n?t|[iau]n?ss(e|es|ions|iez|ent))[,;.!:? ]' Balzac.txt
```

which indicates that the verb ending must be followed by a word separator. `grep` understands yet another notation which is more general, using: '`[:punct:]`', which includes all the punctuation and '`[:space:]`', which includes all the various white space characters (space, tab,...). This is still not satisfactory, because we might also have a verb at the end of a line, not followed by punctuation or white space.

We can use \>, which is a notation for ε only when it is found at the end of a word. The condition that the ending is really at the end of a word can be written as follows:

```
> grep -E '([îâû]n?t|[iau]n?ss(e|es|ions|iez|ent))\>' Balzac.txt
```

One last problem: reduce the noise. Our formulation is still weak because it recognizes works like `masse` and `passions`, which are not verbs of the form *imparfait du subjonctif*. An exact solution is out of the question: we would have to find all the words in the French dictionary which are an exception to our rule. This would be possible because the dictionary is finite, but it would be far to arduous. A reasonable approximation would be to require that the ending appear after a root of at least three letters. So finally, after adding \< to specify the beginning of a word:

3. The '`-E`' option gives access to notational extensions

```
> grep -E \
  '\<[a-zéèêîôûç]{3,}([îâû]n?t|[iau]n?ss(e|es|ions|iez|ent))\>' \
  Balzac.txt
```

Other UNIX programs use the similar types of notational extensions, each time with their own minor variants. This is the case in particular with `(f)lex`, a generator for lexical analyzers; for `sed`, a batch editor; for `perl`, a programming language for manipulation text files; for `(x)emacs`... There are many pages of documentation for each of these programs describing their precise individual notations. There also exist programming libraries which allow programs to manipulate rational expressions. Case in point, for the C programmer, the library, `regexp`, allows the programmer to "compile" rational expressions and to "execute" them for testing word recognition. There are analogous libraries for for C++, for Java...

Attention There is a common confusion which you should avoid. For expressing sets of file names in various UNIX shells, we use notations similar but semantically different than rational expression. For example, '`foo*`' designates files whose name starts with `foo` (like for example '`foo.java`'). This pattern, '`foo*`', does not represent the pattern $fo(o^\star)$.

Chapter 4

Finite Automata

In this chapter we succinctly introduce finite automata[1]. For the readers interested in the formal aspects of the theory of finite automata, we can particularly recommend several chapters of Hopcroft et Ullman (1979) or the first few chapters of Sakarovitch (2003) (available in English and French) The limited description presented here loosely follows that presented in Sudkamp (1997).

4.1 Finite Automata

4.1.1 Foundation

In this section, we introduce the most simple model of finite automata: the complete deterministic automaton. This model allows the definition of computation methods and the language associated with an automaton. We finish this section with the definition of equivalence of automata and of the utility of a state.

Definition 4.1 (Deterministic finite automaton (complete))**.** *A* complete deterministic finite automaton *(*DFA*, deterministic finite automaton) is defined by a structure* $A = (\Sigma, Q, q_0, F, \delta)$*, in which:*

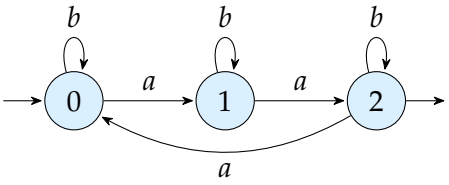
- Σ *is a finite set of symbols (called the alphabet)*
- Q *is a finite set of what are called* states
- $q_0 \in Q$ *is the initial* initial
- $F \subset Q$ *is a set of* final *states*
- δ *is a* total *function of* $Q \times \Sigma \to Q$*, called the* transition function.

The *deterministic* qualifier will be justified later in the introduction of *non-deterministic* automata (definition 4.8).

A finite automaton corresponds to a directed graph. The vertices correspond to states, and some vertices are distinguished as final or initial (or both). The graph edges (*transitions*),

1. The singular form is *automaton*; the plural form is *automata*.

are labeled by symbols from Σ. A transition is thus an element (a triple) of $Q \times \Sigma \times Q$. If $\delta(q, a) = r$, we say that a is the *label* of the transition (q, a, r); q is the l'*origin* of the transition, and r is the *destination* of the transition. Automata may be represented graphically as in automate 4.1. TODO TODO TODO change LaTeX code for this figure reference from automate to automaton.



Automaton 4.1 – A deterministic finite automaton

In this representation, the initial state 0 is distinguished by an incoming graph edge having no origin, and each final state (here there is only one final state, 2) is distinguished by an outgoing edge having no destination. The transition function corresponding to the graph is also made explicit by a matrix:

δ	a	b
0	1	0
1	2	1
2	0	2

To define an automaton in Vcsn, list, line by line, the transitions, using the syntax: '*source* -> *destination label*, *label*...'. The initial states are denoted by '$ -> 0', and the final states by '2 -> $'.

automate 4.1 can be written as:

```
>>> a = vcsn.automaton('''
context = "lal_char(ab), b"
$ -> 0
0 -> 0 b
0 -> 1 a
1 -> 1 b
1 -> 2 a
2 -> 2 b
2 -> 0 a
2 -> $
''')
```

A *computation* in A is a sequence of transitions $e_1 \dots e_n$ of A, such that for each pair of successive transitions e_i, e_{i+1}, the destination state of e_i is the origin state of e_{i+1}. The *label of a computation* is the word constructed by concatenating the labels of $e_1 \dots e_n$. A computation in

A is *successful* if the origin of the first transition, e_1, is an initial state of A and the destination of e_n is a final state of A. The language *recognized* by the automaton, A, denoted $L(A)$, is the set of concatenated labels of all successful computations. In the previous example, the word *baab* appears in the recognized language, because it is concatenation of labels of the successful computation: $(0, b, 0)(0, a, 1), (1, a, 2)(2, b, 2)$.

The relation $\vdash_A$ (read *yield*) allows us to formalize the notion of a single computation step. As such we will write, for $a \in \Sigma$ and $v \in \Sigma^\star$: $(q, av) \vdash_A (\delta(q, a), v)$ to denote a step in the computation which uses the transition $(q, a, \delta(q, a))$. The reflexive and transitive closure of $\vdash_A$ is denoted $\vdash_A^\star$; $(q, uv) \vdash_A^\star (p, v)$ if there exists a sequence of $q = q_1 \dots q_n = p$ such that $(q_1, u_1 \dots u_n v) \vdash_A (q_2, u_2 \dots u_n v) \dots \vdash_A (q_n, v)$. The reflexivity of this relation signifies that for every (q, u), $(q, u) \vdash^\star (q, u)$. With this notation, we have:

$$L(A) = \{u \in \Sigma^\star \mid (q_0, u) \vdash_A^\star (q, \varepsilon), \text{ with } q \in F\}$$

This notation allows us to view an automaton as a machine which *recognizes* words: every path starting at q_0 "consumes" the symbols, one by one, of a word being considered. The process stops when a word is completely consumed. If the state attained by such a traversal is a final state, then the word belongs to the language recognized by the automaton.

We can derive an algorithm for testing whether a word is contained in the language recognized by a deterministic finite automaton.

```
// u = u_1 ... u_n is the word to recognize.
// A = (Σ, Q, q_0, δ, F) is the DFA.
q := q_0;
for i := 1 to n do
  |  q := δ(q, u_i)
if q ∈ F then return true else return false ;
```

Algorithm 4.2– Recognition by a DFA

The complexity of this algorithm can be determined by observing that every step of the computation corresponds to an application of the function $\delta()$, which itself reduces to reading an entry from a table and an assignment— two operations which can be achieved in constant time. Recognizing a word, u, is done in exactly $|u|$ steps.

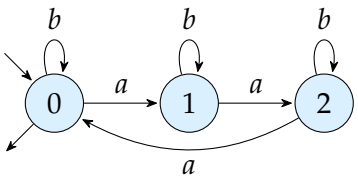
Definition 4.2 (Recognizable Language)**.** *A language is* recognizable *if there exists a finite automaton which recognizes it.*

In Vcsn, an automaton may be viewed as a function which computes a Boolean representing the membership of a word in the language of the automaton:

```
>>> a('baab')
>>> a('a')
```

The routine `'automaton.shortest'` obtains the first n elements of the list of accepted words.

```
>>> a.shortest(10)
```



Automaton 4.3 – an deterministic finite automaton counting a (modulo 3)

The figure in automate 4.3 is an example similar to the first one. The 0 state is both initial and final. The automaton recognizes the language containing words u such that $|u|_a$ is divisible by 3: each state corresponds to a remainder when divided by 3: a successful computation necessarily corresponds to a remainder equal to 0.

By similar reasoning to this one used to define a computation of any length, it is possible to recursively obtain the function $\delta^\star : Q \times \Sigma^\star \to Q$ from the transition function δ:

— $\delta^\star(q, \varepsilon) = q$

— $\delta^\star(q, au) = \delta^\star(\delta(q, a), u)$

We note that since δ is a total function, $\delta^\star$ is also a total function: the image of $\delta^\star$ of any word in $\Sigma^\star$ is well defined, i.e. exists and is unique. This new notation introduces an alternative notation for then language recognized by an automaton, A:

$$L(A) = \{u \in \Sigma^\star \mid \delta^\star(q_0, u) \in F\}$$

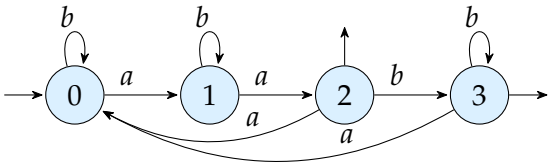
At the moment we have seen the automaton as a machine which recognizes words. It is also possible to view it as a system of *production*: starting in the initial state, every walk leading to a final state constructs a word by concatenating the labels along the traversed edges.

Every finite automaton recognizes a single language, but the converse is not true: several automata may recognize the same language. As was the case with rational expressions, we say that such automata are *equivalent*.

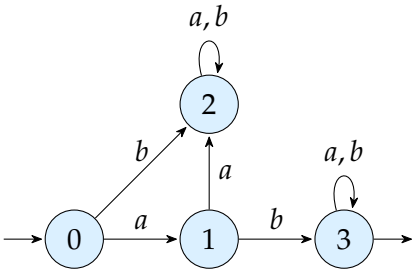
Definition 4.3 (Equivalent Automata)**.** *Two finite automata, A_1 and A_2, are said to be* equivalent *if they recognize the same language.*

As such, par example, is automate 4.4 equivalent to automate 4.1? Both recognize the language of words containing a number of a's congruent to 2 modulo 3.

We have previously noted that, $\delta^\star$ is defined for all words in $\Sigma^\star$. This implies that the recognition algorithm (algorithm 4.2) requires exactly $|u|$ steps of computation, corresponding to a complete execution of the loop. This may be inefficient in some cases such as the example of automate 4.5, recognizing $\{ab\}\{a, b\}^\star$. In this particular case is it possible to accept or reject some words without visiting all the symbols in the word.



Automaton 4.4 – A deterministic finite automaton equivalent to 'automate 4.1



Automaton 4.5 – a deterministic finite automaton for $ab(a+b)^\star$

4.1.2 Partial specification

For further considering the problem of stopping the computation as soon as possible, in this section we introduce alternative definitions which are in fact equivalent in terms of the automaton and the computation.

Definition 4.4 (Deterministic finite automaton)**.** *A* deterministic finite automaton *is a structure* $A = (\Sigma, Q, q_0, F, \delta)$*, where:*

- Σ *is a finite set of symbols (the alphabet)*
- Q *is a finite set of states*
- $q_0 \in Q$ *is the initial state*
- $F \subset Q$ *are the final states*
- δ *is a* partial *function* $Q \times \Sigma \to Q$

The difference with respect to definition 4.1 is that here δ is defined as a partial function rather than a total function, whose domain of definition is a subset of $Q \times \Sigma$. According to this new definition, it is possible to encounter a situation where the computation stops before reading an end of its input. This happens as soon as the automaton reaches a configuration (q, au) for which there is no transition with origin q labeled by a.

The definition 4.4 is in fact strictly equivalent to the preceding one, in as much as the partially specified automata could be completed by adding a *sink state* which absorbs the transitions missing in the original, without any change in the recognized language. Formally, let $A = (\Sigma, Q, q_0, F, \delta)$ be an partially specified automaton. We define $A' = (\Sigma, Q', q'_0, F', \delta')$ with:

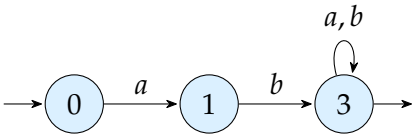
- $Q' = Q \cup \{q_p\}$
- $q'_0 = q_0$

— $F' = F$

— $\forall q \in Q, a \in \Sigma, \delta'(q,a) = \delta(q,a)$ if $\delta(q,a)$ exists, $\delta'(q,a) = q_p$ otherwise.

— $\forall a \in \Sigma, \delta'(q_p,a) = q_p$

The sink state, q_p, is the state for which we are directed to in A' in the case of a failure in A; once in q_p, it is impossible to transition to any other state of A and thus cannot reach a final state. This transformation is illustrated in automate 4.6, for which the transformed automaton is precisely automate 4.5— state 2 playing the role of the sink.



Automaton 4.6 – An partially specified automaton

A' recognizes the same language as A because:

— If $u \in L(A)$, $\delta^\star(q_0,u)$ exists and belongs to F. In this case, the same computation exists in A' and also finishes as well in a final state.

— If $u \notin L(A)$, two cases are possible: either $\delta^\star(q_0,u)$ exists but is not final, and the same thing happens in A'; or the computation stops in A after the prefix v: we thus have $u = va$ and $\delta(\delta^\star(q_0,v),a)$ is missing from A. The corresponding computation in A' lands in the same state, at which point a transition exists with destination q_p. Thereafter, the automaton rests in this state until the computation finishes; Since this state is not final, the computation fails and the input string is rejected.

For every automaton (in the sense of definition 4.4), there therefore exists an completely specified automaton (or *automaton complete*) which is equivalent.

Exercise 4.5 (BCI-0506-1). *We say that a language L over $\Sigma^\star$ is* local *if there exists $I, F \subseteq \Sigma$ and $T \subseteq \Sigma^2$ (the set of words in $\Sigma^\star$ of length 2) such that $L = (I\Sigma^\star \cap \Sigma^\star F) \setminus (\Sigma^\star T \Sigma^\star)$. In other words, a local language is defined by a finite set of prefixes and suffixes of length 1 (I and F) and a set T of "forbidden factors" of length 2. These local languages play a particular, important role for inferring language L given a set of example words in L.*

1. *Show that the language $L_1 = aa^\star$ is a local language; that the language $L_2 = ab(ab)^\star$ is a local language. In these two cases, $\Sigma = \{a,b\}$.*
2. *Show that he language $L_3 = aa^\star + ab(ab)^\star$ is not local.*
3. *Show that the intersection of two local languages is local; that the union of two local languages local languages is not necessarily local.*
4. *Show that every local language is rational.*
5. *Let $\Sigma = \{a,b,c\}$. Let C be the finite set $C = \{aab, bca, abc\}$. Suggest and justify a construction pour the automaton recognizing the smallest (in the sense of inclusion) local language local which is a superset of C. You may, for example, construct the sets I, F, and T and from them deduce the form of the automaton.*
6. *Recall that a morphism is a function $\phi : \Sigma_1^\star \to \Sigma_2^\star$ such that (i) $\phi(\varepsilon) = \varepsilon$ and (ii) if $u, v \in \Sigma_1^\star$, then $\phi(uv) = \phi(u)\phi(v)$. A morphism is said to be a* letter-by-letter morphism *if it is induced*

by an injection $\phi : \Sigma_1 \to \Sigma_2$, and is recursively extended by $\phi(au) = \phi(a)\phi(u)$. For example, for $\phi(a) = \phi(b) = x, \phi(c) = y$ we have $\phi(abccb) = xxyyx$.

Show that every rational language, L, is the image of a letter-by-letter morphism of a local language L'.

Clue: If $A = (\Sigma, Q, q_0, F, \delta)$ is a deterministic finite automaton recognizing L, define the local, L', over the alphabet in which the symbols are triples $[p, a, q]$, with $p, q \in Q$ and $a \in \Sigma$. Now consider the morphism induced by $\phi([p, a, q]) = a$.

4.1.3 Useful States

A second result concerning equivalence of automata requires the introduction of the following complementary definitions.

Definition 4.6 (Accessible, co-accessible, useful, trim)**.** *A state q of A is said to be* accessible *if there exists $u \in \Sigma^\star$ such that $\delta^\star(q_0, u) = q$. q_0 is trivially accessible (by $u = \varepsilon$). An automaton for which every state is a accessible is itself said to be accessible.*

A state q of A is said to be co-accessible *if there exists a $u \in \Sigma^\star$ such that $\delta^\star(q, u) \in F$. Every final state is trivially co-accessible (by $u = \varepsilon$). An automaton for which every state is co-accessible is itself is said to be co-accessible.*

A state q in A is said to bet useful *if it is both accessible and co-accessible. An automaton for which every state is useful is said to be* émondé *(in French* émondé*).*

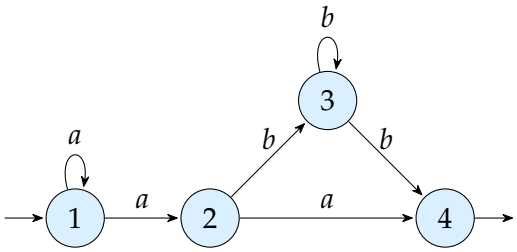
The useful states are thus the state which participate in at least one successful computation. We can reach them along some path which starts the initial state and ends at a final state. The other states, the non-useful states, do not serve much purpose; they do not contribute to populating $L(A)$. This is precisely what is shown in the following theorem.

Theorem 4.7 (Trimming)**.** *If $L(A) \neq \emptyset$ is a recognizable language, then it is also recognized by a trim automaton.*

Proof. Let A be an automaton recognizing L, and $Q_u \subset Q$ be the subset of useful states; $Q_u \neq \emptyset$ because $L(A) \neq \emptyset$. The restriction δ' of δ to Q_u induces automaton $A' = (\Sigma, Q_u, q_0, F, \delta')$. A' is equivalent to A. Q_u begin a subset of Q, we immediately have $L(A') \subset L(A)$. Take $u \in L(A)$. All the states along a path which computes u are useful by definition (by definition of useful). This computation also exists in A' and terminates on a final state. u is also recognized by A'. □

4.1.4 Non-deterministic Automata

In this section, we extend the model of automaton of definition 4.4 by authorizing multiple transitions with the same symbol to from a state. Such automata are known as *non-deterministic*. We show that such a generalization does not in any way change the expressivity of the model. The languages recognized by non-deterministic automata are the same as those recognized by deterministic automata.



Two transitions originating at 1 are labeled by a: $\delta(1,a) = \{1,2\}$. An input of aa causes a successful computation passing successfully through 1, 2, and 4, which is final; aa also causes the $(1,a,1)(1,a,1)$, which is not a successful computation.

Automaton 4.7 – A non-deterministic automaton

Non-determinism

Definition 4.8 (Non-deterministic Finite Automaton)**.** *An finite automaton* non-deterministic (NFA, *nondeterministic finite automaton) is defined by the structure* $A = (\Sigma, Q, I, F, \delta)$*, where:*

- Σ *is a finite set of symbols (the alphabet)*
- Q *is a finite set of states*
- $I \subset Q$ *are the* initial *states*
- $F \subset Q$ *are the* final *states*
- $\delta \subset Q \times \Sigma \times Q$ *is a relation; each triplet* $(q,a,q') \in \delta$ *is a transition.*

We can equally consider δ to be a function of $Q \times \Sigma \to 2^Q$. The image by δ of the pair (q,a) is a possibly empty subset of Q ($\delta(q,a) \subset Q$). $\delta(q,a) = \emptyset$ in the case that there is no transition labeled a whose origin is q. Without loss of generality we could require that the initial state be unique.

The novelty introduced by this definition is the indeterminacy exhibited in the transitions. For the pair (q,a), there may be several possible transitions. In such a case, we say that reading a symbol a from the input causes a choice (or an indeterminacy, a *non-determinism*), and multiple transitions are possible.

Note that this definition correctly generalizes the notion of finite automaton; definition 4.4 is a special case of definition 4.8, in which for each (q,a), the set $\delta(q,a)$ is a singleton set. In other words, every deterministic automaton is non-deterministic. Take special note that the vocabulary is confusing. Being *non-deterministic* does not mean that it is deterministic! For this reason we write "non-deterministic" with a hyphen rather than writing "non deterministic".

The notions of *computation* and *successful computation* are defined exactly as in the case of deterministic. As well, we define on the extended transition function $\delta^\star : q \times \Sigma^\star \to 2^q$ by:

- $\delta^\star(q,\varepsilon) = \{q\}$
- $\delta^\star(q,au) = \bigcup_{r \in \delta(q,a)} \delta^\star(r,u)$

These notions are illustrated in automate 4.7.

The language recognized by a non-deterministic automaton is defined by:

$$L(A) = \{u \in \Sigma^\star \mid \exists q_0 \in I : \delta^\star(q_0, u) \cap F \neq \emptyset\}$$

For a word to belong to the language recognized by the automaton, is suffices that there exist at least one successful computation, among all the possible computations— that is to say a computation which consumes all the symbols of u between an initial state and a final state; the recognition fails if *all the computations* lead to a failing situation. This implies that to compute the membership of a word to a language, we must, in principle, examine in turn all possible paths, and thus backtrack when the search encounters an impasse. This exploration can however be reduced to a search problem in the underlying graph of the automaton, because a word is recognized by the automaton if it is possible to prove that at least one final state is accessible from an initial state along some path labeled by the letters of u. In this new model, the recognition of a word is thus polynomial in the length of the input word.

Non-determinism does not pay

The generalization of the finite automata model leading to the introduction of non-deterministic transitions is inconsequential from the point of view of language recognition. Any language recognized by a non-deterministic finite automaton is also recognized par a deterministic finite automaton.

Theorem 4.9 (Determinization)**.** *For every NFA, A (with n states) we can construct a DFA, A', (having at most 2^n states) which is equivalent to A.*

Take $A = (\Sigma, Q, I, F, \delta)$, and consider A' defined by $A' = (\Sigma, 2^Q, I, F', \delta')$ with:

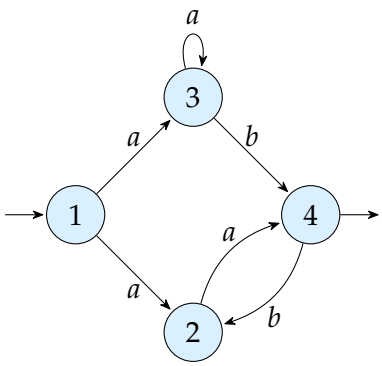
— $F' = \{G \subset Q \mid F \cap G \neq \emptyset\}$.

— $\forall G \subset Q, \forall a \in \Sigma, \delta'(G, a) = \bigcup_{q \in G} \delta(q, a)$.

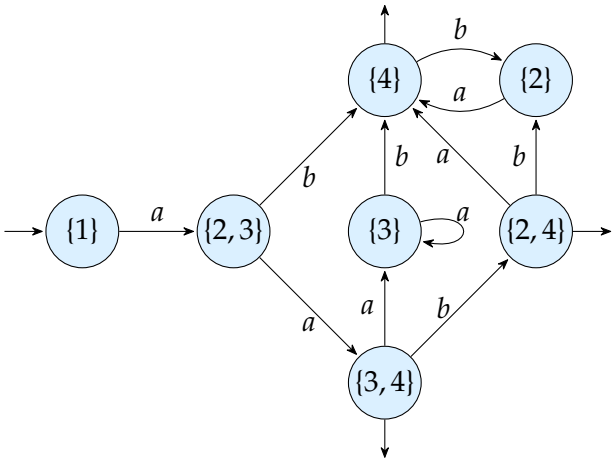
The states of A' are thus associated in a 1-to-1 manner with the subsets of Q (there are finitely many of them). The initial state is the set I. Each subset containing a final state of A corresponds to a final state of A'. The transition with origin begin subset E, labeled by a, has destination being the set of all states of Q which is the destination of some state in E with transition labeled a. A' is the *determinization* of A. We illustrate such a construction in automate 4.8.

automate 4.8 has 4 states. Its determinization will thus have $2^4 = 16$, corresponding to the number of subsets of $\{1, 2, 3, 4\}$. Its initial state is the singleton $\{1\}$. Its final states final are all the subsets of $\{1, 2, 3, 4\}$ which contain 4. There are exactly 8 such subsets: $\{4\}$, $\{1, 4\}$, $\{2, 4\}$, $\{3, 4\}$, $\{1, 2, 4\}$, $\{1, 3, 4\}$, $\{2, 3, 4\}$,$\{1, 2, 3, 4\}$. Consider, for example, the transitions with origin 1 having label a. There are two such transitions, having destinations 2 and 3. Thus $\{1\}$ will be the origin of a transition labeled a, having destination $\{2, 3\}$. The determinization is shown in automate 4.9. We note that this figure only represents the useful states of the determinism. As such, $\{1, 2\}$ is not represented since there is no means of reaching that state.

What would happen if we draw an additional transition on automate 4.8, looping on the state 1 with a label a? Construct the determinization of this new automaton.



Automaton 4.8 – An automaton to determinize



Automaton 4.9 – The result of determining automate 4.8

We now prove theorem 4.9; et pour grasp the sense of the proof, we refer to automate 4.8, and consider the computations of the words prefixed by *aaa*. The first *a* leads to an indeterminacy between 2 and 3. Le second *a* leads to 4 (if we chose initially to go to 2) but to 3 (if we initially chose to go to 3). Reading the third *a* eliminates the ambiguity, since there is no transition with origin 4 having label *a*. The only possible state after reading *aaa* is 3. Now consider automate 4.9 with the same input. The initial ambiguities correspond to states $\{2,3\}$ (after reading the first *a*) and $\{3,4\}$ (after reading the second *a*). After the third *a*, there is no more doubt, and we arrive at the singleton $\{3\}$. We now formalize these ideas by proving the result we are interested in.

Proof of theorem 4.9. First remark: A' is a deterministic finite automaton, since the image under function δ' of a pair (H, a) is uniquely defined.

Next we will show that each computation in A corresponds to exactly one computation in A'. Formally:

$$\begin{array}{llll} \text{if} & \exists q_0 \in I : (q_0, u) \vdash_A^\star (p, \varepsilon) & \text{then} & \exists G \subset Q : p \in G, (\{I\}, u) \vdash_{A'}^\star (G, \varepsilon) \\ \text{if} & (\{I\}, u) \vdash_{A'}^\star (G, \varepsilon) & \text{then} & \exists q_0 \in I : \forall p \in G : (q_0, u) \vdash_A^\star (p, \varepsilon) \end{array}$$

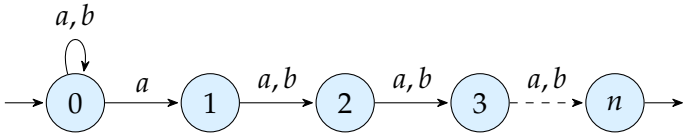
We use induction on the length of u. If $u = \varepsilon$ the claim is true by definition of the initial state of A'. Suppose the result holds for every word of length strictly less than $|u|$, and consider a word $u = va$. let $(q_0, va) \vdash_A^\star (p, a) \vdash_A (q, \varepsilon)$ be a computation in A. By the inductive hypothesis, there exists a computation in A' such that $(\{I\}, v) \vdash_{A'}^\star (G, \varepsilon)$, with $p \in G$. This implies, by virtue of the same definition of δ', that q belongs to $H = \delta'(G, a)$, and thus that $(\{I\}, u = va) \vdash_{A'}^\star (H, \varepsilon)$, with $q \in H$.

Conversely, let $(\{I\}, u = va) \vdash_{A'}^\star (G, a) \vdash_{A'} (H, \varepsilon)$. For every $p \in G$ there exists a computation on A such that $(q_0, v) \vdash_A^\star (p, \varepsilon)$. G begin a transition labeled a, there exists in G a state p such that $\delta(p, a) = q$, with $q \in H$. So $(q_0, u = va) \vdash_A^\star (q, \varepsilon)$, with $q \in H$.

We conclude then that we have $(q_0, u = va) \vdash_A^\star (q, \varepsilon)$, with $q \in F$ if and only if $(\{I\}, u) \vdash_{A'}^\star (G, \varepsilon)$ with $q \in G$, thus with $F \cap G \neq \emptyset$, thus $G \in F'$. It follows directly that $L(A) = L(A')$. □

The construction used for constructing a DFA equivalent to an NFA is called *construction by subset*. It is translated directly from an algorithm which allows the construction of the determinization of general automata. We note that his construction could be organized in such a manner to consider only *the accessible states* of the determinization. For such, it suffices to construct step by step the states starting with $\{I\}$ leading to the states accessible from states thus constructed. This method results in general in complete automata having less than 2^n states.

However, there are those automata for which a combinatoric explosion occurs, such as automate 4.10. Can you explain where the difficulty originates? Which language is recognized by this automaton?



Automaton 4.10 – An automaton difficult to determinize

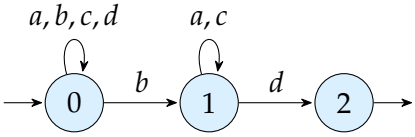
Being that it does not deliver any gain in expressivity, is reasonable to ask what is the value of the non-deterministic automata . There are at least two things: they are easier to construct given certain representations of languages, and thus they are useful for certain proofs (seen later) or algorithms. The can provide more "compact" (in some cases exponentially more compact) machines—not a negligible advantage.

To determine an automaton, use '`automaton.determinize`'. Notice that the states are labeled by the states of the original automaton. Use '`automaton.strip`' to eliminate the decorations.

The routine '`context.de_bruijn`' generates automata of the family of 'automate 4.10.

Launch '`vcsn.context('lal_char(abc), b').de_bruijn(3).determinize().strip()`', then increase the argument of '`de_bruijn`'.

Exercise 4.10 (BCI-0203). *We consider* $A = (\Sigma, Q, 0, \{2\}, \delta)$, *automate 4.11.*



Automaton 4.11 – An automaton, potentially to determinize

1. *Is A deterministic? Justify your answer.*
2. *Which language is recognized by A?*
3. *Give a graphical representation of a deterministic automaton, A', equivalent to A. Justify your construction.*

4.1.5 Spontaneous Transitions

It is common, in practice, to use a more flexible definition of the notion of finite automaton, by allowing transitions labeled by the empty word. These are called *transitions* spontaneous.

Definition 4.11 (Finite Automaton with spontaneous transitions)**.** *A (non-deterministic) finite automaton* with spontaneous transitions *(*ε-NFA*, non-deterministic finite automaton with* ε *transitions) is defined by a structure* $A = (\Sigma, Q, I, F, \delta)$*, where:*

— Σ *is a finite set of symbols (the alphabet)*

— Q *is a finite set of states*

— $I \subset Q$ *is the set of* initial *states*

— $F \subset Q$ *is the set of* final *states*

— $\delta \subset Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ *is the transition relation.*

Again, we could have required a unique initial state and defined δ as a function (partial) from $Q \times (\Sigma \cup \{\varepsilon\})$ to 2^Q.

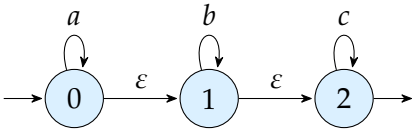
Formally, a non-deterministic automaton with spontaneous transitions is defined as an NFA but with the difference being that δ allows transitions labeled by ε. The spontaneous transitions convey the notion of: $(q, u) \vdash_A (p, v)$ if either (i) $u = av$ and $(q, a, p) \in \delta$ or rather (ii) $u = v$ and $(q, \varepsilon, p) \in \delta$. In other words, in an ε-NFA, it is possible to change state without consuming a symbol, by using a transition labeled by the empty word. The language recognized par an ε-NFA, A, is, as before, defined by:

$$L(A) = \{u \in \Sigma^\star \mid (q_0, u) \vdash_A^\star (q, \varepsilon) \text{ with } q_0 \in I, q \in F\}$$

automate 4.12 shows an example of ε-NFA.

This new extension adds no expressivity to the model, since it is possible to transform any ε-NFA, A, into an equivalent NFA. Towit, we first introduce the notion of ε-closure of a state q, corresponding to all the states accessible from q by one or more spontaneous transitions. Formally

Definition 4.12 (ε-closure of a state)**.** *Let* $q \in Q$*. We mean by* ε-closure *(in French* ε*-fermeture) of* q*, the set* ε*-closure*$(q) = \{p \in Q \mid (q, \varepsilon) \vdash_A^\star (p, \varepsilon)\}$*. By construction,* $q \in \varepsilon$*-closure*(q)*.*



Automaton 4.12 – An automaton with spontaneous transitions corresponding à $a^\star b^\star c^\star$

Intuitively, the closure of a state q contains all the states which are possible to reach starting at q without consuming any symbols from the input. For example, the ε-closure of state 0 of automate 4.12 is equal to $\{0, 1, 2\}$.

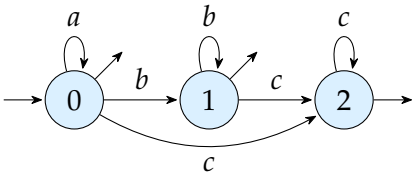
Theorem 4.13. *For every ε-NFA, A, there exists an NFA, A', such that $L(A) = L(A')$.*

Proof. Let $A = (\Sigma, Q, q_0, F, \delta)$. We define A' to be $A' = (\Sigma, Q, q_0, F', \delta')$ with:

$$F' = \{q \in Q \mid \varepsilon\text{-}closure(q) \cap F \neq \emptyset\} \qquad \delta'(q, a) = \bigcup_{p \in \varepsilon\text{-}closure(q)} \delta(p, a)$$

By induction similar to the previous one, we shoe then that any computation $(q_0, u) \vdash_A^\star p$ is equivalent to a computation $(q_0, u) \vdash_{A'}^\star p'$, with $p \in \varepsilon\text{-}closure(p')$, and then that $L(A) = L(A')$. □

We directly deduce an constructive algorithm for removing, the spontaneous transitions, but maintaining the same number of states—a state may neither be deleted nor added. Apply the algorithm to automate 4.12 to construct automate 4.13.



Automaton 4.13 – automate 4.12 having removed its spontaneous transitions

In the case of the previous proof we speak of *backward ε-closure of an automaton* (you might say "upstream"): the function δ "starts" (upstream) to change spontaneous transitions which are encountered, then connects the transition with a letter. Any (upstream) state from which we arrive at a final state via an ε-transition also becomes a final state.

We can equally introduce the *forward ε-closure of an ε-NFA* (you might say "downstream"): we follow every transition which is not spontaneous with every spontaneous transition (upstream). Every state which is reachable by an ε-transition directly from an initial state becomes initial, i.e., all the states upstream (by one or more ε-transitions) from an initial state.

We prefer in general the backward closure because it does not introduce new initial states.

Spontaneous transitions introduce an additional flexibility to the automaton object. For example, it is easy to see that we can transform any finite automaton into an automaton having a unique final state by making use of spontaneous transitions.


The empty word is designated '\e' in Vcsn. To use ε-NFA, you must specify that the empty word is a valid label The context is '`lan_char, b`' in stead of '`lal_char, b`'. Here is how to write automate 4.12. Notice the '`r`' before the Python string:

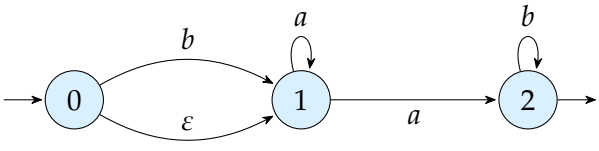
```
>>> a = vcsn.automaton(r'''
context = "lan_char, b"
$ -> 0
0 -> 0 a
0 -> 1 \e
1 -> 1 b
1 -> 2 \e
2 -> 2 c
2 -> $
''')
```

To eliminate the spontaneous transitions, use '`automaton.proper`'.

Exercise 4.14 (BCI-0304). *Primarily in French, but also occasionally in English, adjectives may be placed before or after the nouns they modify. Consider that we may place adjectives either before the noun or after but not both, a linguist proposes to represent the phrases of nouns and adjectives as the rational expression:* $a^\star n + na^\star$.

1. *By systematic application of the Thompson method described in section 4.2.2, construct a finite automaton corresponding to this rational expression:*
2. *Construct the ε-closure of each of the states in the automaton constructed in question 1.*
3. *Deduce an automaton without ε-transition pour the rational expression of question 1. You should take care to identify and eliminate the useless states.*
4. *Finally, use the construction studied in the lecture to derive an equivalent* deterministic *automaton for the previous question.*

Exercise 4.15 (BCI-0405-1). *Consider A to be the non-deterministic automate 4.14.*



Automaton 4.14 – Non-deterministic Automaton, A

1. *Use the method from the lecture to construct the deterministic automaton A' equivalent to A.*

4.2 Recognizables

In section 4.1, we have defined the recognizable languages as being those languages recognized by a deterministic finite automaton. The previous sections have shown than we could have equally well defined them as the languages recognized by a non-deterministic finite automaton or even by a non-deterministic finite automaton with spontaneous transitions.

In this section, we will first show that the set of recognizable languages is closed for all "classical" operations by showing constructions applying directly to the automata. We will, thereafter, show that the recognizables are exactly the languages rationals (as presented in section 3.1.1). Finally we will present a set of classic results which makes it possible to characterize the recognizable languages .

4.2.1 Operations on recognizables

Theorem 4.16 (Closed under complement)**.** *The set of recognizable languages is closed under complement.*

Proof. Let L be a recognizable and A be a *complete* DFA recognizing L. We construct an automaton A' for $\overline{L}$ by taking $A' = (\Sigma, Q, q_0, F', \delta)$, with $F' = Q \backslash F$. Any successful computation of A finishes in a state of F, causing its failure in A'. Conversely, any failing computation A ends up in a non-final state of A, which implies that it succeeds in A'. □

The following theorem is a tautology, since by definition the rational languages are closed under union. However, the proof of this theorem provides the construction of a deterministic automaton which accepts the union of the languages of the two given deterministic automata.

Theorem 4.17 (Closure under union)**.** *The set of recognizable languages is closed under union.*

Proof. Let L^1 and L^2 be two recognizables, recognized respectively by A^1 et A^2, two *complete* DFA [2]. We construct an automaton $A = (\Sigma, Q = Q^1 \times Q^2, q_0, F, \delta)$ for $L^1 \cup L^2$ in the following way:

— $q_0 = (q_0^1, q_0^2)$

— $F = (F^1 \times Q^2) \cup (Q^1 \times F^2)$

— $\delta((q^1, q^2), a) = (\delta^1(q^1, a), \delta^2(q^2, a))$

The construction of A is intended to function as A^1 and A^2 "in parallel": for any symbol, a, at the entry, we transit according to δ to the pair of states resulting from one transition in A^1 and one transition in A^2. A successful computation in A corresponds to a successful computation in A^1 (stopping in a state of $F^1 \times Q^2$) and in A^2 (stopping in a state of $Q^1 \times F^2$). □

2. In this proof and in the following section, when we use the notation A^n, we do not mean the concatenation $A \cdot A^{n-1}$. The notation A^1 and A^2 as well as L^1 and L^2 we do not mean self concatenation. The superscripts are simply used for indexing purposes, not for exponentiation.

We will see another, simpler, construction a bit later (automate 4.18) for this operation, but which, on the contrary to the preceding one, does not preserve determinism of the machine implementing the union.

Theorem 4.18 (Closure under intersection)**.** *The set of recognizable languages is closed under intersection.*

We present a constructive proof, but notice that the result also follows directly from two preceding results and de Morgan's law: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.

Proof. Let L^1 and L^2 be two recognizables, recognized respectively by A^1 and A^2, two *complete* DFA. We construct an automaton $A = (\Sigma, Q = Q^1 \times Q^2, q_0, F, \delta)$ for $L^1 \cap L^2$ in the following way:

— $q_0 = (q_0^1, q_0^2)$

— $F = F^1 \times F^2$

— $\delta((q^1, q^2), a) = (\delta^1(q^1, a), \delta^2(q^2, a))$

The construction of the intersection automaton is identical to the automaton realizing the union except that a computation which succeeds A must simultaneously both in A^1 and A^2. This is explained in the new definition of the set F of final states: $F = F^1 \times F^2$. □

Exercise 4.19 (AST-0506)**.** *Intersection is not considered a rational operation. However, the intersection of any two languages rational languages is a rational language. If the language L_1 is recognized by the complete deterministic automaton, $A_1 = (\Sigma, Q^1, q_0^1, F^1, \delta^1)$, and L_2 by the complete deterministic automaton, $A_2 = (\Sigma, Q^2, q_0^2, F^2, \delta^2)$, then $L = L_1 \cap L_2$ is recognized by the automaton $A = (\Sigma, Q^1 \times Q^2, q_0, F, \delta)$. Teh states of A are the pairs of states of Q^1 and Q^2, and:*

— $q_0 = (q_0^1, q_0^2)$

— $F = F^1 \times F^2$

— $\delta((q_1, q_2), a) = (\delta^1(q^1, a), \delta^2(q^2, a))$

A simulates, in fact, the computations in A_1 and A_2 in parallel, et terminates a successful computation when it simultaneously *attains a final state in A_1 and A_2.*

In the rest of this exercise, we consider $\Sigma = \{a, b\}$.

1. *Let L_1 be the language of words beginning with an a, and let L_2 be the language of words whose length is a multiple of* 3 *Propose a complete deterministic finite automaton for each of these languages.*
2. *Construct $L_1 \cap L_2$ using the directions given above.*
3. *We will no recompute the same construction* in the general case *by using the De Morgan's law: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. We remember that if L is rational et recognized by A (which is complete and deterministic), then $\overline{L}$ is rational and is recognized by $\overline{A}$, which can be deduced from A by rendering the final states of A as non-final in $\overline{A}$, and likewise rendering the non-final states of A as final of $\overline{A}$. $\overline{A}$ is likewise deterministic and complete.*

a. *As a first step we are interested in computing a deterministic automaton $\overline{A}$ for $\overline{L_1} \cup \overline{L_2}$.*
   The automaton for the union of two rational languages recognized respectively by $\overline{A_1}$ and $\overline{A_2}$ is constructed by adjoining a new initial state q_0 et two ε-transitions both with origin q_0 and with destinations q_0^1 and q_0^2 respectively. This set of states is thus $Q = Q^1 \cup Q^2 \cup \{q_0\}$.
   We denote as $\overline{A'}$ the automaton obtained by eliminating the ε-transition.
   Prove that, if $\overline{A_1}$ and $\overline{A_2}$ are deterministic, then $\overline{A'}$ only has one non-deterministic state[3].

b. *The determinization of $\overline{A'}$ is an automaton, $\overline{A}$, whose set of states corresponds to the subsets of Q. Prove, by induction, that if $\overline{A_1}$ and $\overline{A_2}$ are deterministic and complete, only the doubleton subsets $\{q^1, q^2\}, q^1 \in Q^1, q^2 \in Q^2$ appear in the set of useful states* useful *of the determinization. Show that, therefore, the transition function of $\overline{A}$ conforms to the construction presented directly above.*

c. *What are the final states of $\overline{A'}$? What can you deduce from this about the automaton representing the complement of the language of $\overline{A}$? Conclude by justifying the direct construction given above.*

Exercise 4.20 (BCI-0203-2-1)**.** *Construct a non-deterministic automaton, A, recognizing the language L over the alphabet $\Sigma = \{a, b\}$, such that every word of L simultaneously satisfies the following two conditions:*

— *every word in L has length divisible by 3.*

— *every word in L starts with letter a and ends with the letter b*

Justify your construction.

Construct the equivalent deterministic automaton by the method of subsets.

Theorem 4.21 (Closure under mirroring)**.** *The set of recognizable languages is closed under mirroring.*

Proof. Let L be a recognizable language, recognized by $A = (\Sigma, Q, q_0, F, \delta)$, and having a unique final state, denoted q_F. A', defined by $A' = (\Sigma, Q, q_F, \{q_0\}, \delta')$, where $\delta'(q, a) = p$ if and only if $\delta(p, a) = q$ recognizes exactly the mirror language of L. A' is in fact obtained by reversing the direction of the edges of the graph of A: any computation in A which concatenates the symbols of u from q_0 to q_F corresponds to a computation in A' which concatenates the symbols of u^R from q_F to q_0. We note that even if A is deterministic, this proposed construction does not necessarily result in a deterministic automaton for the mirror language. □

Theorem 4.22 (Closure under concatenation)**.** *The set of recognizable languages is closed under concatenation.*

Proof. Let L^1 and L^2 be two languages recognized respectively by A^1 and A^2. We suppose that the sets of states Q^1 and Q^2 are disjoint and that A^1 as a unique final state with no outgoing transition. We construct the automaton A with $L^1 \cdot L^2$ by identifying the final state of A^1 with the initial state of A^2. Formally, we have $A = (\Sigma, Q^1 \cup Q^2 \setminus \{q_0^2\}, q_0^1, F^2, \delta)$, where δ is defined by:

3. A state, q, is non-deterministic if there are multiple transitions with a as origin, all having the same label.

— $\forall q \in Q^1, q \neq q_f, a \in \Sigma, \delta(q,a) = \delta^1(q,a)$

— $\delta(q,a) = \delta^2(q,a)$ if $q \in Q^2$.

— $\delta(q_F^1, a) = \delta^1(q_F, a) \cup \delta^2(q_0^2, a)$

Every successful computation in A must necessarily reach a final state of A^2, and must have passed through the state of A^1, which is the only point of passage between only point of passage from A^1 to A^2. After crossing through q_F^1 into the states of A_2 the computation never returns to the states of A_1. It decomposes, therefore, into a computation through both automata.

Conversely, a word of L^1L^2 factorizes into the form $u = vw$, with $v \in L^1$ and $w \in L^2$. Each factor corresponds to a successful computation respectively in A^1 and A^2, from which we immediately deduce a successful computation in A. □

Theorem 4.23 (Closure under star)**.** *The set of recognizable languages is closed by star.*

Proof. The construction of A' recognizing $L^\star$ given A recognizing L is as follows. First, we add a new initial state, q'_0, to A, with a transition spontaneous to q_0 of A. Next, we add one spontaneous transition per final state of A to the new initial state initial q'_0. These new transitions implement iteration in A' over the words of L. To finish the construction, we mark q'_0 as a final state to assure that $\varepsilon \in A'$. □

As an application of this section, you can show (by constructing the corresponding automata) that the set of recognizable languages is also closed under the operations of prefixing, suffixing, factors, and subsequences. TODO TODO it is not clear what the *operations* of prefixing, suffixing, factors, and subsequences are.

4.2.2 Recognizable and rational languages

The properties proven for recognizable languages (of union, concatenation, and star) in the previous section, plus the remark that all finite languages are recognizable, allows us to affirm that every rational language is recognizable. The set of all rational languages begin, indeed, the smallest set containing all the finite sets and closed under rational operations, is necessarily included in the set of recognizable languages. First, we will show how to exploit the constructions shown previously to construct, in a simple way, an automaton corresponding to a given rational expression. Thereafter, we will show the converse, to know that every recognizable language is also rational: the languages recognized by the finite automata are precisely those which are designated by rational expressions.

From rational expression to automaton

The constructions in the previous section have shown how to construct automata which realize elementary operations over the languages. Now we will inspire you by deriving an algorithm to convert a rational expression to a finite automaton recognizing the same language.

There exist several ways to construct such an automaton. The one will discuss is th Thompson "pure" construction which presents several simple properties simples:

— a unique initial state, q_0, having no incoming transition

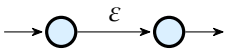
— a unique final state, q_F, having no outgoing transition

— exactly two times as many states as the number of symbols in the rational expression rational (without counting the parentheses nor the implicit concatenation operator).

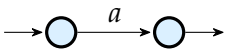
That last property gives us a simply way to check the result.

Since rational expressions are formally defined inductively (recursively), we begin by presenting the finite which form the "building blocks", they are $\emptyset$ (automate 4.15), ε (automate 4.16), and symbols Σ (automate 4.17).

Automaton 4.15 – Thompson Automaton for $\emptyset$

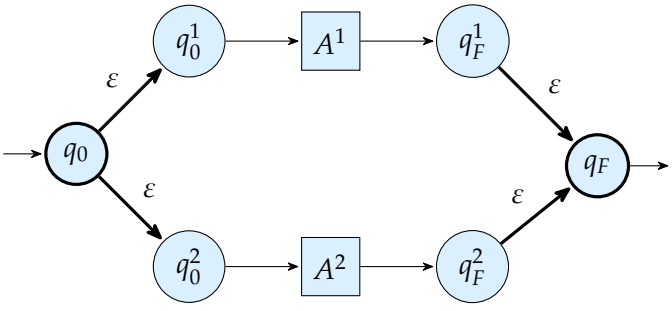


Automaton 4.16 – Thompson Automaton for ε



Automaton 4.17 – Thompson Automaton for a

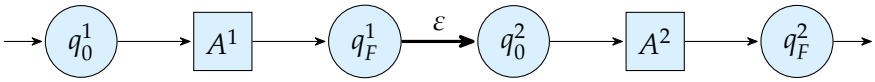
Using these elementary automata, we will iteratively construct automata for more complex rational expressions. If A_1 and A_2 are Thompson automata of e_1 and e_2, then automate 4.18 recognizes the language denoted by the expression $e_1 + e_2$.



Automaton 4.18 – Thompson Automaton for $e_1 + e_2$

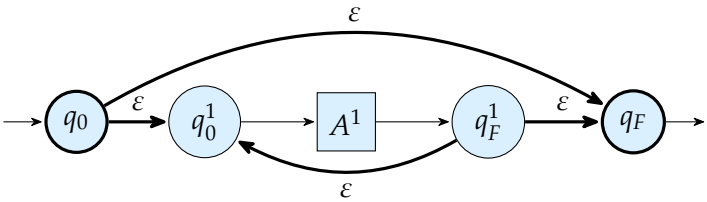
The union corresponds, thus, to parallel placement of A_1 and A_2: a computation in this machine is successful if and only if it is successful in either of the two machines A_1 or A_2 (or perhaps both).

The machine recognizing the language designated by concatenation of two expressions, e_1 and e_2, corresponds to a series placement of the two machines, A_1 and A_2, where the final state of A_1 is connected to the initial state of A_2 by a spontaneous transition as in 'automate 4.19.



Automaton 4.19 – Thompson Automaton for $e_1 e_2$

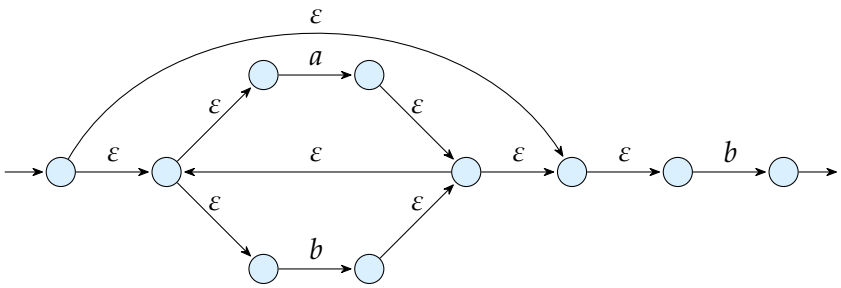
The machine implementing the star is, as previously, constructed by inserting a looping possibility from the final state to the initial state of the machine, and by make it possible to recognize ε, as in automate 4.20.



Automaton 4.20 – Thompson Automaton for $e_1{}^\star$

Using these simple constructions, is possible to derive an algorithm which constructs an automaton recognizing the language designated by any rational expression. It suffices to decompose the expression into its elementary components, then to apply the preceding steps to construct the target automaton. This algorithm is known by name as the *Thompson Algorithm* .

automate 4.21 illustrates this construction.



Automaton 4.21 – Thompson Automaton for $(a + b)^\star b$

This straightforward construction produces an automaton which has a $2n$ states for an expression formed by n rational operations other than concatenation (every other operations inserts exactly two additional states). Every state of the automaton has at most two exiting transitions. However, this construction has the disadvantage of producing a non-deterministic automaton, which contains several spontaneous transitions. Other procedure exist which are more efficient for expressions which do not explicitly contain the symbol ε (par example the *Glushkov construction*) or for directly constructing a deterministic automaton.


The Python method, '*expression*.thompson', produces the Thompson automaton.

```
>>> vcsn.B.expression('(a+b)*b').thompson()
```

From automaton to rational expression

The construction of a rational expression designating the language recognized by an automaton, A, requires the introduction of a new variety of automata, which we will call *generalized*. The automata generalized differ from finite automata in that their transitions are labeled by rational subsets of $\Sigma^\star$. In a generalized automaton, the label of a computation is constructed by concatenation of labels encountered along transitions. The language recognized par an automaton generalized is the union of the languages corresponding to the successful computations. The generalized automata recognize exactly the same languages as "standard" finite automata .

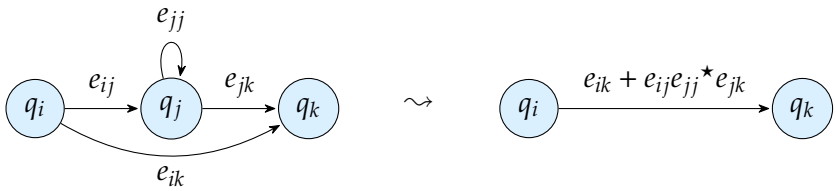
The general idea of the transformation which we will study consists of beginning with a standard finite automaton and deleting the states, one by one, each time assuring that that deletion does not modify the language recognized by the automaton. Each step constructs an element of a series of generalized automata which are all equivalent to the original automaton. The procedure terminates when there only remains one initial state and one final state. We read the transition label to arrive at a rational expression which designates the language of the original automaton.

To simplify the test we begin by introducing two new states, q_I and q_F, which play the role of unique initial and final states respectively. These new states are connected respectively to all the other initial and final states via spontaneous transitions. We thereafter assure that at the end of the algorithm, only one transition remains which has, moreover, q_I as origin and q_F as destination.

The crucial operation of this method consists of deleting the state q_j, where q_j is neither initial nor final. If there is more than one transition from one state to another, the labels can be combined with a union operation to form a single transition. Let e_{jj} denote the label of the transition from q_j to itself, if such exists, otherwise we say $e_{jj} = \varepsilon$.

The procedure to delete node q_j consists of the following steps:

— For each pair of states (q_i, q_k) with $i \neq j, k \neq j$, such that $q_i \xrightarrow{e_{ij}} q_j$ is a transition from q_i to q_j with label e_{ij} and $q_j \xrightarrow{e_{jk}} q_j$ is a transition from q_j to q_k with label e_{jk}, add a transition $q_i \xrightarrow{e_{ik}+e_{ij}e_{jj}^\star e_{jk}} q_k$ from q_i to q_k with label $e_{ik} + e_{ij}e_{jj}^\star e_{jk}$. Note that e_{ik} is the label of the transition from e_i to e_k if such exists, otherwise $e_{ik} = \emptyset$, meaning the new label is simply $e_{ij}e_{jj}^\star e_{jk}$.
This process of adding a transition must be done for each pair of states including $q_i = q_k$, before q_j may be deleted.
— Delete q_j including all the transitions which have q_j as source or destination.



Automaton 4.22 – Illustration of Brzozowski McCluskey: elimination of state q_j

This procedure is illustrated in automate 4.22.

The proof of correction of this algorithm relies on the fact that on each iteration, the language recognized by the generalize automaton is unchanged. This invariant can be easily verified. Each successful computation passing through q_j before q_j is deleted contains the sequence $q_i q_j q_k$. The label of this sub-computation is copied at the deletion of q_j through the transition $q_i \to q_k$. An equivalent computation exists in the reduced automaton. Conversely, any successful computation passing through the new transition after deletion of a state, or a transition on which the label changed, corresponds to a computation passing through q_j in the original automaton.

This method of construction of the expression equivalent to an automaton is called *method of state elimination*, and also known by the name *Brzozowski McCluskey algorithm*. You will notice that the result obtained depends on the order in which the states are examined.

As an application, it is recommended that you determine a rational expression rational corresponding to automates 4.8 and 4.9.

A fundamental corollary of this result is that for every recognizable language, there exists (at least) one rational expression which designates it: every recognizable language is rational.

Kleene's Theorem

We have shown how to construct an automaton corresponding to a rational expression et how to derive a rational expression designating an language equivalent to it recognized by any finite automaton. These two results make it possible to state one of the major results of this chapter.

Theorem 4.24 (Kleene's Theorem)**.** *A language is recognizable (recognized by a finite automaton) if et only if it is rational (designated by a rational expression).*

This result makes it possible to draw some consequences not yet foreseen. For example, the set of rational languages is closed under complement and finite intersection. This result, is far from evident from looking at notions rational operations alone. However, it simply falls from heaven if we make use of the equivalence between rational and recognizable languages.

In a similar way, the methods of transforming one representation to another are very practical. It is immediate to derive from automate 4.3 a rational expression for a language containing a number of a's which is a multiple of 3. It is much easier to write the rational expression corresponding to automate 4.3. In particular, passing the representation through the form of

automaton makes it possible to derive a method for verifying whether two rational expressions are equivalent: construct the two automata, and verify that they are equivalent. We already know how to construct the automata. A procedure for testing equivalence of two automata is presented in section 4.3.2.

4.3 Some properties of recognizable languages

The equivalence of rational languages to recognizable languages strengthens our toolbox concerning rationals languages: to show that a language is rational, we can choose to exhibit a finite automaton for the language or rather exhibit a rational expression. To show that a language *is not rational*, it is useful to make use of the property presented in the section 4.3.1, which is known by the name *pumping lemma*.

4.3.1 Pumping lemma

Intuitively, the *pumping lemma* (ou *start lemma*, (in French *lemma de pompage* or *lemme de l'étoile*) establishes certain intrinsic limitations on the diversity of words which being to an infinite rational language: beyond a certain length, the words of a rational language are in fact constructed by iteration of patterns which appear in shorter words.

Theorem 4.25 (Pumping lemma)**.** *Let L be a rational language. There exists an integer, k, such that any word, $x \in L$, whose length is larger than k, can be factored into $x = uvw$, with (prop. i) $|v| > 0$ (prop. ii) $|uv| \leq k$ et (prop. iii) pour tout $i \geq 0$, $uv^i w \in L$.*

The thing this lemma fundamentally signifies is that the only way for a rational language to be infinite, is by iteration of patterns: $L(uv^\star w) \subset L$ — hence the name "star lemma".

Proof. In the case that the language is finite, the lemma is trivially true. We may choose a k larger than the largest length of a word of the language.

Otherwise, let A be a DFA of k states recognizing L, and take $x \in L$, such that $|x| \geq k$ (the language being infinite, such an x always exists). The recognition of x in A corresponds to a computation $q_0 \dots q_n$ implying $|x|+1$ states. A only has k states. The prefix of this sequence of length $k+1$ necessarily contains the same state q at least twice: let $q_i = q_u$ with $0 \leq i < j \leq k$. Let u be the prefix of x such that $\delta^\star(q_0, u) = q$, and let v be the factor such that $\delta^\star(q, v) = q$. We have (prop. i) because at least one symbol is consumed along the cycle $q_i \dots q_j = q' \dots q$. We have (prop. ii) because $j \leq k$. Finally, we have (prop. iii) by short circuiting or iterating the path around the loop $q_i \dots q_j = q \dots q$. □

Attention: some non-rational languages also obey the pumping lemma. The Pumping lemma is only a necessary condition; it is not a sufficient condition for rationality Consider example $\{a^n b^n \mid n \in \mathbb{N}\} \cup \Sigma^\star\{ba\}\Sigma^\star$.

TODO TODO TODO the following seems suspicious to me. —— The lemma makes it possible, for example, to prove that the language of perfect squares $L = \{u \in \Sigma^\star, \exists v \text{ tel que } u = v^2\}$, is

not rational. Indeed, let k be the integer specified by the pumping lemma, and x be a word longer than $2k$: $x = yy$ with $|y| \geq k$. It is therefore possible to write $x = uvw$, with $|uv| \leq k$. This implies that uv is a prefix of y, and y is a suffix of w. However, $uv^i w$ must be in L, so that only one of the y's is effected by the iteration: This is impossible.

You likewise show that language which contains only words whose length is a perfect square and the language of words whose lengths are each a prime number both fail to be recognizable languages.

One simple way to express this intrinsic, fundamental limitation of languages rationals is with the following observation. Within an automaton, the choice of successor of a state q depends only on q itself, and not on the way the computation arrived at q. Consequently, *a finite automaton can only control a finite number of different configurations*—in other words *they possess bounded memory*

Bounded memory is not sufficient for a language such as the language of perfect squares. That language demands that the action after a prefix u depend on all of u. Recognizing such a language would require infinitely many states.

4.3.2 Some consequences

In this section, we establish some complementary results relating to decidability, i.e., toward the existence of algorithms which make it possible to resolve some classical problems related to rational languages. We are already aware of an algorithm for deciding whether a word belongs to a ration language (l'algorithm 4.2). This section shows that the majority of classical problems of rational languages have algorithmic solutions.

Theorem 4.26. *If A is a finite automaton having k states:*

(i) $L(A)$ is inhabited (non-empty) if and only if A recognizes a word of length strictly less than k.

(ii) $L(A)$ is infinite if and only if A recognizes a word u for which $k \leq |u| < 2k$.

Proof. item i: one direction of the implication is trivial. If A recognizes a word longer than k, then $L(A)$ in inhabited. Suppose that $L(A)$ is inhabited and let u be a word of $L(A)$ having smallest length. Suppose that $|(|u) > k$ (strictly greater). The computation $(q_0, u) \vdash_A^\star (q, \varepsilon)$ contains at most k steps, implying that some state is visited at least twice and thus implies that the computation contains a loop; call it C. If we omit C from the computation, $(q_0, u) \vdash_A^\star (q, \varepsilon)$, (short circuit) then we have a word of $L(A)$ whose length is strictly less than $|(|u)$, which contradictions the assumption that u has shortest length. By contradiction we have established: $|u| < k$.

item ii: Analogous reasoning to that used to prove the pumping lemma shows one direction of this implication. Now if $L(A)$ is infinite, it must at least contain a word longer than k. Let u a word of length at least k with the smallest possible length. Either $|(|u) < 2k$ (strictly less) in which case the result is proven, or $|(|u) \geq 2k$. By the pumping lemma we may short-circuit a factor of size k and have a word u' with $|(|u') \leq k$. The smallest word whose length is less than k has length less than $2k$. □

Theorem 4.27. *Let A be a finite automaton. There exists an algorithm making it possible to decide if:*

— $L(A)$ *is empty*

— $L(A)$ *is finite/infinite.*

The result flows directly as the previous ones. There exists, indeed, an algorithm for deciding whether a word u is recognized by A. The previous result assures us that it suffices to test $|\Sigma|^k$-many words for deciding whether the language of an automaton A is empty. In the same way, $|\Sigma|^{2k} - |\Sigma|^k$ verifications suffice for proving that an automaton recognizes an infinite language. For these two problems, more efficient algorithm than these exist, which rest on a walk through the graph of the automaton.

Form this we deduce a result concerning equivalence:

Theorem 4.28. *Let A_1 and A_2 be two finite automata. There exists a procedure making it possible to decode whether A_1 and A_2 are equivalent.*

Proof. It suffices to form the automata recognizing $(L(A_1) \cap \overline{L(A_2)}) \cup (\overline{L(A_1)} \cap L(A_2))$, (for example by using the procedures in section 4.2.1), and test whether the language is empty. If that is the case, then two automata are thus equivalent. □

4.4 The canonical automaton

In this section we give a new characterization of recognizable languages, from which we will introduce the notion of the *canonical automaton of a language*. We present thereafter, an algorithm to construct the canonical automaton of the recognizable language represented by any DFA.

4.4.1 A new characterization of recognizable languages

We begin with a new definition: of indistinguishability.

Definition 4.29 (Indistinguishable words of a language)**.** *Let $L \in \Sigma^\star$. Two words u et v are said to be* indistinguishable *in L if pour any $w \in \Sigma^\star$, either uw and vw are both L, or uw and vw are both in $\overline{L}$. We use $\equiv_L$ to denote the indistinguishability relation in L.*

In other words, two words u and v are *distinguishable in L* if there exists a word $w \in \Sigma^\star$ such that either $uw \in L$ and $vw \notin L$, or the contrary. The indistinguishability relation in L is a reflexive, symmetric, and transitive relation: thus an equivalence relation.

Let's consider, as an illustration, the language $L = a(a + b)(bb)^\star$. For this language, $u = aab$ and $v = abb$ are indistinguishable. For every word $x = uw$ of L, $y = vw$ is indeed another word of L. On the contrary, $u = a$ and $v = aa$ are distinguishable. By concatenating $w = abb$ with u, we obey $aabb$ which is in L; on the contrary, $aaabb$ is not in L.

In a similar manner, we define the notation of indistinguishability in a deterministic automaton.

Definition 4.30 (Words indistinguishable in an automaton)**.** *Let $A = (\Sigma, Q, q_0, F, \delta)$ be a* deterministic *finite automaton. Two words, u and v, are said to be* indistinguishable in A *if $\delta^\star(q_0, u) = \delta^\star(q_0, v)$. We use $\equiv_A$ to denote the indistinguishability relation in A.*

To say it differently, two words u and v are indistinguishable in A, if the computation of u in A starting at q_0, finishes in the same state q as the computation of v. This notion is lined with the preceding notation. For two words, u and v, indistinguishable in $L(A)$, each word w such that $\delta^\star(q, w)$ finishes in a final state is a valid continuation of both u and v in L. Conversely, every word failing after q is an invalid continuation both of u and of v. On the contrary, the converse is false, two words which indistinguishable in $L(A)$ may be distinguishable in A.

Recall the notion of right-congruence introduced in section 2.4.3:

Definition 4.31 (Right invariance)**.** *An equivalence relation $\mathcal{R}$ over $\Sigma^\star$ is said to be a* right invariant *if $u \mathcal{R} v \Rightarrow \forall w, uw \mathcal{R} vw$. A right equivalence invariant relation invariant is called a* right congruence*.*

By definition, the two relations of indistinguishability defined here are right invariants.

Now we are ready for the principle result of this section.

Theorem 4.32 (Myhill-Nerode)**.** *Let L be a language over Σ. The three following assertions are equivalent:*

(i) *L is a rational language*

(ii) *There exists an equivalence relation $\equiv$ over $\Sigma^\star$, which is right invariant, having finitely many equivalence classes, and such that L is equal to the union of these equivalence classes.*

(iii) *$\equiv_L$ has a finite number of equivalence classes.*

Proof. item i $\Rightarrow$ item ii: A being rational, there exists a, A, which recognizes it. The equivalence relation, $\equiv_A$, having as many equivalence classes as states. This number is necessarily finite. This relation is a right invariant, and L, defined as $\{u \in \Sigma^\star \mid \delta^\star(q_0, u) \in F\}$, is simply the union of equivalence classes associated with the final states of A.

item ii $\Rightarrow$ item iii. Let $\equiv$ be the relation which satisfies the property item ii, and let u and v be such that $u \equiv v$. By the property of right invariance, we have for every word $w \in \Sigma^\star$, $uw \equiv vw$. This leads either to that uw and vw are simultaneously in L (if their common equivalence class is a superset L), or that both are outside of L (in the other case). It follows that $u \equiv_L v$: every equivalence class for $\equiv$ is included in the equivalence class of $\equiv_L$; There are, therefore, fewer equivalence classes for $\equiv_L$ than for $\equiv$. Thus the number of equivalence classes of $\equiv_L$ is finite.

item iii $\Rightarrow$ item i: Let us construct the automaton $A = (\Sigma, Q, q_0, F, \delta)$ in the following way:

— Every state of Q corresponds to an equivalence class $[u]_L$ of $\equiv_L$; from which we know Q has finite cardinality.

— $q_0 = [\varepsilon]_L$, equivalence class of ε.

— $F = \{[u]_L, u \in L\}$.

— $\delta([u]_L, a) = [ua]_L$. This definition of δ is independent of the choice of representative element of $[u]_L$. If u and v are in the same equivalence class for $\equiv_L$, by right invariance of $\equiv_L$, the same will hold for ua and va.

A, defined this way, is a complete deterministic finite automaton. We now show that A recognizes L and in doing so, we show by induction that $(q_0, u) \vdash_A^\star [u]_L$. The property is true for $u = \varepsilon$. Suppose it is true for every word of size less than k. Let $u = va$ of size $k + 1$. We have $(q_0, ua) \vdash_A^\star (p, a) \vdash_A (q, \varepsilon)$. Thus, by the induction hypothesis we know that $p = [u]_L$; and because $q = \delta([u]_L, a)$, then $q = [ua]_L$. This is the needed result.

We deduce that if $u \in L(A)$, a computation with u as input finishes in a final state of A, and thus that $u \in L$. Conversely, if $u \in L$, a computation with u as input finishes in a state $[u]_L$, which is final, by definition of A. □

This result provides a new characterization of recognizable languages and may thus be used to show that a language is recognizable, or fails to be. As such, for example, $L = \{u \in \Sigma^\star \mid \exists a \in \Sigma, i \in \mathbb{N} \text{ such that } u = a^{2^i}\}$ is not recognizable. Indeed, for any i, j, a^{2^i} and a^{2^j} are distinguishable from a^{2^i}. There is, thus, not a finite number of equivalence classes for $\equiv_L$, and consequently L cannot be recognized by a finite automaton. L is, thus, not a rational language.

4.4.2 Canonical automaton

The main consequence of the previous result concerns the existence of a unique (except for a renumbering of the states) and minimal (in terms of number of states) representative among the equivalence relation of finite automata.

Theorem 4.33 (Canonical automaton)**.** *The Automaton, A_L, based on the equivalence relation, $\equiv_L$, is a smallest complete deterministic automaton which recognizes L. This automaton is unique (except for the renumbering of states) and is called the* canonical automaton *of L.*

Proof. Let A be a deterministic finite automaton recognizing L. $\equiv_A$ defines an equivalence relation satisfying the conditions of item ii of the proof of theorem 4.32. We have shown that every state of A corresponds to an equivalence class (pour $\equiv_A$) which is included in the of equivalence for $\equiv_L$. The number of states of A is thus necessarily greater than that of A_L. The case where A and A_L have the same number of states corresponds to the case where the equivalence classes are all alike, making it possible to define a 1-to-1 mapping between the states of two machines. □

The existence of A_L being guaranteed, a separate exercise is how to construct it. The direct construction strategy for equivalence classes of $\equiv_L$ it not necessarily obvious. We will present an algorithm which enables the construction of A_L starting with any deterministic automaton recognizing L. As discussed previously, we will define a third indistinguishability relation, but this time on the states.

Definition 4.34 (State indistinguishable)**.** *Two state q and p of a deterministic finite automaton, A, are* distinguishable *if there exists a word, w, such that the computation (q, w) terminates*

on a final state, then the computation (p, w) *fails. If two states are not distinguishable, they are* indistinguishable.

Like the relations of indistinguishability discussed previously, this relation is an equivalence relation, denoted $\equiv_\nu$ over the states of Q. The set of equivalence classes, $[q]_\nu$, is denoted Q_ν. Pour a deterministic finite automaton, $A = (\Sigma, Q, q_0, F, \delta)$, defines the finite automaton, A_ν, by: $A_\nu = (\Sigma, Q_\nu, [q_0]_\nu, F_\nu, \delta_\nu)$, with: $\delta_\nu([q]_\nu, a) = [\delta(q, a)]_\nu$; and $F_\nu = [q]_\nu$, with $q \in F$. The function, δ_ν, is correctly defined in the sense that if p and q are indistinguishable, then necessarily $\delta(q, a) \equiv_\nu \delta(p, a)$.

We will first show that this new automaton is completely identical to the canonical automaton, A_L.

We define a function ϕ which associates a state of A_ν with a state of A_L in the following way.

$$\phi([q]_\nu) = [u]_L \text{ if there exists } u \text{ such that } \delta^\star(q_0, u) = q$$

We note that ϕ is a function: if u and v in $\Sigma^\star$ are both mapped to indistinguishable states in A, then it is clear that $u$and v are also indistinguishable, and are thus in the same equivalence class for $\equiv_L$: The result of ϕ does not depend on the choice of the particular u.

We now show that ϕ is a bijection. This follows from the following sequence of equivalences.

$$\begin{aligned}
\phi([q]_\nu) = \phi([p]_\nu) &\Leftrightarrow \exists u, v \in \Sigma^\star, \delta^\star(q_0, u) = q, \delta^\star(q_0, u) = p, \text{ and } u \equiv_L v && (4.1)\\
&\Leftrightarrow \delta^\star(q_0, u) \equiv_\nu \delta^\star(q_0, u) && (4.2)\\
&\Leftrightarrow [q]_\nu = [p]_\nu && (4.3)
\end{aligned}$$

We finally show that computations in A_ν are in bijection with ϕ with computations in A_L. We have in effect:

- $\phi([q_0]_\nu) = [\varepsilon]_L$, for $\delta^\star(q_0, \varepsilon) = q_0 \in [q_0]_\nu$
- $\phi(\delta_\nu([q]_\nu, a) = \delta_L(\phi([q]_\nu), a)$ for let u be such that $\delta^\star(q_0, u) \in [q]_\nu$, then: (i) $\delta(\delta^\star(q_0, u), a) \in \delta_\nu([q]_\nu, a)$ (cf. the definition of δ_ν) et $[ua]_L = \phi(\delta_\nu([q]_\nu, a)) = \delta_L([u], a)$ (cf. the definition of δ_L), which we would need to prove.
- if $[q]_\nu$ is final in A_ν, then there exists u such that $\delta^\star(q_0, u)$ is a final state de A, implying that $u \in L$, and thus that $[u]_L$ is a final state of the canonical automaton.

It follows that every computation in A_ν is isomorphic (by ϕ) to a computation in A_L, and also that, these two automata having the same initial and final states, recognize the same language.

4.4.3 Minimization

The idea of the algorithm minimization of a deterministic automaton, $A = (\Sigma, Q, q_0, F, \delta)$, consists of trying to identify the equivalence classes for $\equiv_\nu$,and deriving the automaton, A_ν (alias A_L). That Q is finite guarantees us that the algorithm exists for computing these equivalence classes. The iterative procedure described below sketches a naïve implementation of

the algorithm, which constructs the partition corresponding to the equivalence classes by refinement of an initial partition, Π_0, which only distinguishes final vs non-final states. The algorithm basically proceeds as follows:

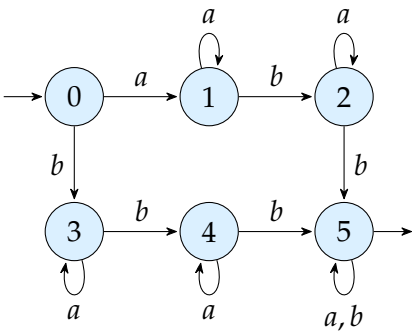
— Initialize with two equivalence classes: F and $Q \setminus F$

— Iterate until stable:

  — For each pair of states, q and p, in the same class of the partition, Π_k, if there exists $a \in \Sigma$ such that $\delta(q, a)$ and $\delta(p, a)$ are not in the same class for Π_k, then they are in two different classes of Π_{k+1}.

We verify that when this procedure stops (after a finite number of steps), two states are in the same class if and only if they are indistinguishable. This procedure is known by the name *algorithm of Moore*. A brute force implement has quadratic complexity (because we must compare every pair of states). We may use other means to reduce the complexity to $n \log(n)$, with n being the number of states.

The following illustrates the procedure automate 4.23:



Automaton 4.23 – A DFA to minimize

The successive iterations of the construction algorithm for the equivalence classes for $\equiv_\nu$ proceed as follows:

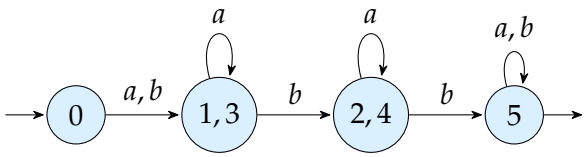
— $\Pi_0 = \{\{0, 1, 2, 3, 4\}, \{5\}\}$ (because 5 is the only final state)

— $\Pi_1 = \{\{0, 1, 3\}, \{2, 4\}, \{5\}\}$ (because 2 and 4, on the letter b, transition to 5).

— $\Pi_2 = \{\{0\}, \{1, 3\}, \{2, 4\}, \{5\}\}$ (because 1 and 3, on the letter b, advance respectively to 2 and 4).

— $\Pi_3 = \Pi_2$ end of the procedure

The minimal automaton minimal resulting from this procedure is l'automate 4.24.

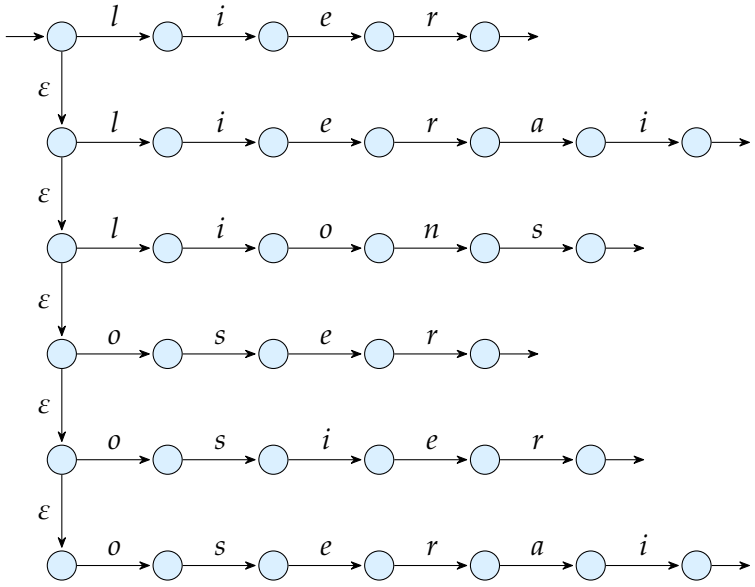


Automaton 4.24 – The minimal automaton of $(a + b)a^\star ba^\star b(a + b)^\star$


For minimizing an automaton, use '`automaton.minimize`'. The states are labeled by the states of the original automaton.

Exercise 4.35 (BCI-0607-2-1). *In this exercise, we are interested in a representation of large dictionaries for the finite automata. We consider here that dictionary is a finite list of words. There therefore exists a finite automaton which represents it. One such representation guarantees in particular that the sort for a word has linear complexity as a function of the length of the word in question.*

Consider the list of (French) words $L = \{lier, lierai, lions, oser, osier, oserai\}$*, which corresponds to the language of l'automate 4.25 (in which the state names are omitted).*



Automaton 4.25 – A small dictionary

1. *Use a method seen in the lecture to construct an* deterministic *automaton pour the language, L. You should proceed in two steps, taking care to explain each step: removing spontaneous transitions, and then determinization of the automaton having no spontaneous transitions.*
2. *We say that two states, p and q, of a deterministic automaton are* indistinguishable *if every word* $u \in \Sigma^*$, $\delta^*(p,u) \in F \Leftrightarrow \delta^*(q,u) \in F$.

   That is to say, two states, p and q, are indistinguishable if and only if every word which leads to a successful computation starting at p also leads to a successful computation starting at q.

   (a) *Identify a pair of indistinguishable states in the automaton constructed in question 1; and a pair of distinguishable states (non-indistinguishable).*

   (b) *The indistinguishability relation is a binary relation ofer the set of states of an automaton. Prove that it is an* equivalence relation *(i.e. that it is reflexive, symmetric, and transitive).*

3. *Let* $A = (\Sigma, Q, q_0, F, \delta)$ *be a deterministic finite automaton with only useful states, the* merging *of two indistinguishable states, p et q, produce an automaton, A', identical to A, ex-*

cept that p and q are replaced by a single state r, which "inherits" all the properties (being initial or final) and transitions of the two original states. Formally, A' is defined by $A' = (\Sigma, Q \setminus \{p,q\} \cup \{r\}, q'_0, F', \delta')$ with:

- *$q'_0 = r$, if $q = q_0$ or $p = q_0$, $q'_0 = q_0$; else*
- *$F' = F \setminus \{p,q\} \cup \{r\}$, if $p \in F$, $F' = F$; else*
- *$\forall a \in \Sigma, \forall e \in Q \setminus \{p,q\}, \delta'(e,a) = r$, if $\delta(e,a) = p$ or $\delta(e,a) = q$; $\delta'(e,a) = \delta(e,a)$, if $\delta(e,a)$ exists; $\delta'(e,a)$ is undefined otherwise.*
- *$\forall a \in \Sigma, \delta'(r,a) = \delta(p,a)$, if $\delta(p,a)$ exists, $\delta'(r,a)$ is undefined otherwise.*

Starting with the automaton you constructed in question 1, construct a new automaton by combining two states which you have identified as indistinguishable.

4. *prove that if A is a deterministic finite automaton, and A' is derived from A by combining two indistinguishable states, then $L(A) = L(A')$ (the automata are equivalent).*
5. *The computation of the indistinguishability relation is done by determining which pairs of states are distinguishable; those which are not distinguishable are indistinguishable. The algorithm depends on the following two properties:*

   [initialization] *if p is final and q is not final, then p and q are distinguishable;*

   [iteration] *if $\exists a \in \Sigma$ and (p', q') distinguishable such that $\delta(q,a) = q'$ and $\delta(p,a) = p'$, then (p,q) are distinguishable.*

   Formally, we initialize the set of distinguishable pairs by using the property [initialization]; then we consider repeatedly all the pairs of states which are not already known to be distinguishable by applying the step [iteration], until the set of distinguishable couples stabilizes.
   Apply this algorithm to the deterministic automaton you constructed in question 1.
6. *Repeatedly apply the operation of combination to all the states which are indistinguishable in automate 4.25.*

The computation of the indistinguishability relation is fundamentally the operation of minimization *of deterministic automata. The operations makes it possible to representer the dictionaries — even very large — in the same way; it also makes it possible to derive an very simple algorithm for testing whether two automata are equivalent.*

Chapter 5

Grammaires syntagmatiques

Dans cette partie, nous présentons de manière générale les grammaires syntagmatiques, ainsi que les principaux concepts afférents. Nous montrons en particulier qu'au-delà de la classe des langages rationnels, il existe bien d'autres familles de langages, qui valent également la peine d'être explorées.

5.1 Grammaires

Definition 5.1 (Grammaire)**.** *Une* grammaire syntagmatique *G est définie par un quadruplet (N, Σ, P, S), où N, Σ et P désignent respectivement des ensembles de* non-terminaux *(ou* variables*), de* terminaux*, et de* productions *(ou règles de production). N et Σ sont des alphabets disjoints. Les productions sont des éléments de $(N \cup \Sigma)^\star \times (N \cup \Sigma)^\star$, que l'on note sous la forme $\alpha \to \beta$. S est un élément distingué de N, appelé le* symbole initial *ou encore l'*axiome *de la grammaire.*

La terminologie de langue anglaise équivalente à grammaire syntagmatique est *Phrase Structure Grammar*, plutôt utilisée par les linguistes, qui connaissent bien d'autres sortes de grammaires. Les informaticiens disent plus simplement *rules* (pour eux, il n'y a pas d'ambiguïté sur le terme !).

Il est fréquent d'utiliser T pour désigner Σ, l'ensemble des terminaux. Il est également utile d'introduire $V = T \cup N$, le *vocabulaire* de la grammaire. Dans la suite, nous utiliserons les conventions suivantes pour noter les éléments de la grammaire: les non-terminaux seront notées par des symboles en majuscule latine; les terminaux par des symboles en minuscules; les mots de $V^\star = (T \cup N)^\star$ par des lettres minuscules grecques.

Illustrons ces premières définitions en examinant la grammaire G_1 (grammar 5.1). Cette grammaire contient une seule variable, S, qui est également l'axiome; deux éléments terminaux a et b, et deux règles de production, p_1 et p_2.

Si $\alpha \to \beta$ est une production d'une grammaire G, on dit que α est la *partie gauche* et β la *partie droite* de la production.

L'unique opération autorisée, dans les grammaires syntagmatiques, est la récriture d'une séquence de symboles par application d'une production. Formellement,

$$p_1 = S \rightarrow aSb$$
$$p_2 = S \rightarrow ab$$

Grammar 5.1 – G_1, une grammaire pour $a^n b^n$

Definition 5.2 (Dérivation immédiate)**.** *On définit la relation* $\underset{G}{\Rightarrow}$ *(lire:* dérive immédiatement*) sur l'ensemble* $V^\star \times V^\star$ *par* $\gamma\alpha\delta \underset{G}{\Rightarrow} \gamma\beta\delta$ *si et seulement si* $\alpha \rightarrow \beta$ *est une production de G.*

La notion de dérivation immédiate se généralise à la dérivation en un nombre quelconque d'étapes. Il suffit pour cela de considérer la fermeture transitive et réflexive de la relation $\underset{G}{\Rightarrow}$, que l'on note $\overset{\star}{\underset{G}{\Rightarrow}}$:

Definition 5.3 (Dérivation)**.** *On définit la relation* $\overset{\star}{\underset{G}{\Rightarrow}}$ *sur l'ensemble* $V^\star \times V^\star$ *par* $\alpha_1 \overset{\star}{\underset{G}{\Rightarrow}} \alpha_m$ *si et seulement si il existe* $\alpha_2, ..., \alpha_{m-1}$ *dans* $V^\star$ *tels que* $\alpha_1 \underset{G}{\Rightarrow} \alpha_2 \underset{G}{\Rightarrow} ... \underset{G}{\Rightarrow} \alpha_{m-1} \underset{G}{\Rightarrow} \alpha_m$.

Ainsi, par exemple, la dérivation suivante est une dérivation pour la grammar 5.1:

$$S \underset{G_1}{\Rightarrow} aSb \underset{G_1}{\Rightarrow} aaSbb \underset{G_1}{\Rightarrow} aaaSbbb \underset{G_1}{\Rightarrow} aaaaSbbbb$$

permettant de déduire que $S \overset{\star}{\underset{G_1}{\Rightarrow}} aaaaSbbbb$.

Ces définitions préliminaires étant posées, il est maintenant possible d'exprimer formellement le lien entre grammaires et langages.

Definition 5.4 (Langage engendré par une grammaire)**.** *On appelle* langage engendré *par G, noté* $L(G)$*, le sous-ensemble de* $\Sigma^\star$ *défini par* $L(G) = \{w \in \Sigma^\star \mid S \overset{\star}{\underset{G}{\Rightarrow}} w\}$.

$L(G)$ est donc un sous-ensemble de $\Sigma^\star$, contenant précisément les mots qui se dérivent par $\overset{\star}{\underset{G}{\Rightarrow}}$ depuis S. Lorsque α, contenant des non-terminaux, se dérive de S, on dit qu'α est un *proto-mot* (en anglais *sentential form*). Pour produire un mot[1] du langage, il faudra alors habilement utiliser les productions, de manière à se débarrasser, par récritures successives depuis l'axiome, de tous les non-terminaux du proto-mot courant. Vous vérifierez ainsi que le langage engendré par la grammar 5.1 est l'ensemble des mots formés en concaténant n fois le symbole a, suivi de n fois le symbole b (soit le langage $\{a^n b^n, n \geq 1\}$).

Si toute grammaire engendre un langage unique, la réciproque n'est pas vraie. Un langage L peut être engendré par de multiples grammaires, comme on peut s'en persuader en rajoutant

1. La terminologie est ainsi faite que lorsque l'on parle de grammaires, il est d'usage d'appeler *phrases* les séquences d'éléments terminaux, que nous appelions précédemment *mots*. Il s'ensuit parfois (et c'est fort fâcheux), que le terme de *mot* est parfois employé pour dénoter les éléments de l'alphabet Σ (et non plus les éléments de $\Sigma^\star$). Nous essaierons d'éviter cette confusion, et nous continuerons de parler de mots pour désigner les éléments d'un langage, même lorsque ces mots correspondront à ce qu'il est commun d'appeler phrase dans le langage courant.

à volonté des non-terminaux ou des terminaux inutiles. Il existe plusieurs manières pour un non-terminal d'être inutile: par exemple en n'apparaissant dans aucune partie droite (ce qui fait qu'il n'apparaîtra dans aucun proto-mot); ou bien en n'apparaissant dans aucune partie gauche (il ne pourra jamais être éliminé d'un proto-mot, ni donc être utilisé dans la dérivation d'une phrase du langage...). Nous reviendrons sur cette notion d'utilité des éléments de la grammaire à la section 9.1.2.

Comme nous l'avons fait pour les expressions rationnelles (à la section 3.1.3) et pour les automates finis (voir la section 4.1), il est en revanche possible de définir des classes d'équivalence de grammaires qui engendrent le même langage.

Definition 5.5 (Équivalence entre grammaires)**.** *Deux grammaires G_1 et G_2 sont* équivalentes *si et seulement si elles engendrent le même langage.*

Les grammaires syntagmatiques décrivent donc des systèmes permettant d'engendrer, par récritures successives, les mots d'un langage: pour cette raison, elles sont parfois également appelées *grammaires génératives*. De ces grammaires se déduisent (de manière plus ou moins directe) des algorithmes permettant de *reconnaître* les mots d'un langage, voire, pour les sous-classes les plus simples, de *décider* si un mot appartient ou non à un langage.

Le processus de construction itérative d'un mot par récriture d'une suite de proto-mots permet de tracer les étapes de la construction et apporte une information indispensable pour *interpréter* le mot. Les grammaires syntagmatiques permettent donc également d'associer des structures aux mots d'un langage, à partir desquelles il est possible de calculer leur signification. Ces notions sont formalisées à la section 6.2.

5.2 La hiérarchie de Chomsky

Chomsky identifie une hiérarchie de familles de grammaires de complexité croissante, chaque famille correspondant à une contrainte particulière sur la forme des règles de récriture. Cette hiérarchie, à laquelle correspond une hiérarchie [2] de langages, est présentée dans les sections suivantes.

5.2.1 Grammaires de type 0

Definition 5.6 (Type 0)**.** *On appelle grammaire de type 0 une grammaire syntagmatique dans laquelle la forme des productions est non-contrainte.*

Le type 0 est le type le plus général. Toute grammaire syntagmatique est donc de type 0; toutefois, au fur et à mesure que les autres types seront introduits, on prendra pour convention d'appeler *type d'une grammaire* le type le plus spécifique auquel elle appartient.

Le principal résultat à retenir pour les grammaires de type 0 est le suivant, que nous ne démontrerons pas.

2. Les développements des travaux sur les grammaires formelles ont conduit à largement raffiner cette hiérarchie. Il existe ainsi, par exemple, de multiples sous-classes des langages algébriques, dont certaines seront présentées dans la suite.

Theorem 5.7. *Les langages récursivement énumérables sont les langages engendrés par une grammaire de type 0.*

L'ensemble des langages récursivement énumérables est noté $\mathcal{RE}$. Rappelons qu'il a été introduit à la section 2.2.2.

Ce qui signifie qu'en dépit de leur apparente simplicité, les grammaires syntagmatiques permettent de décrire (pas toujours avec élégance, mais c'est une autre question) exactement les langages que l'on sait reconnaître (ou énumérer) avec une machine de Turing. Les grammaires syntagmatiques de type 0 ont donc une expressivité maximale.

Ce résultat est théoriquement satisfaisant, mais ne nous en dit guère sur la véritable nature de ces langages. Dans la pratique, il est extrêmement rare de rencontrer un langage de type 0 qui ne se ramène pas à un type plus simple.

5.2.2 Grammaires contextuelles (type 1)

Les grammaires *monotones* introduisent une première restriction sur la forme des règles, en imposant que la partie droite de chaque production soit nécessairement plus longue que la partie gauche. Formellement:

Definition 5.8 (Grammaire monotone)**.** *On appelle* grammaire monotone *une grammaire syntagmatique dans laquelle toute production $\alpha \to \beta$ est telle que $|\alpha| \leq |\beta|$.*

Cette définition impose qu'au cours d'une dérivation, les proto-mots d'une grammaire monotone s'étendent de manière monotone. Cette contrainte interdit en particulier d'engendrer le mot vide ε. Nous reviendrons en détail sur cette question à la section 5.2.6.

Les langages engendrés par les grammaires monotones sont également obtenus en posant une contrainte alternative sur la forme des règles, conduisant à la notion de grammaire contextuelle:

Definition 5.9 (Grammaire contextuelle)**.** *On appelle* grammaire contextuelle*, ou* grammaire sensible au contexte*, en abrégé* grammaire CS *(en anglais Context-Sensitive) une grammaire telle que toute production de G est de la forme $\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$, avec $\alpha_1, \alpha_2, \beta$ dans $V^\star$, $\beta \neq \varepsilon$ et A est dans N.*

Les grammaires contextuelles sont donc telles que chaque production ne récrit qu'un symbole non-terminal à la fois, le contexte (c'est-à-dire les symboles encadrant ce non-terminal) restant inchangé: d'où la qualification de *contextuelles* pour ces grammaires. Par construction, toute grammaire contextuelle est monotone (car $\beta \neq \varepsilon$). Le théorème suivant nous dit plus: grammaires contextuelles et monotones engendrent exactement les mêmes langages.

Theorem 5.10. *Pour tout langage L engendré par une grammaire monotone G, il existe une grammaire contextuelle qui engendre L.*

Proof. Soit L engendré par la grammaire monotone $G = (N, \Sigma, S, P)$. Sans perte de généralité, nous supposons que les terminaux n'apparaissent que dans des productions de la forme

$A \to a$. Si cela n'est pas le cas, il suffit d'introduire un nouveau non-terminal X_a pour chaque terminal, de remplacer a par X_a dans toutes les productions, et d'ajouter à P la production $X_a \to a$. Nous allons construire une grammaire contextuelle équivalente à G et pour cela nous allons démontrer deux résultats intermédiaires. □

Lemma 5.11. *Si G est une grammaire monotone, il existe une grammaire monotone équivalente dans laquelle toutes les parties droites ont une longueur inférieure ou égale à* 2.

Proof. En procédant par récurrence sur le nombre de productions de G qui ne satisfont pas la condition, il suffit d'illustrer comment récrire une seule telle production. Ainsi, soit $\alpha = \alpha_1 \dots \alpha_m \to \beta = \beta_1 \dots \beta_n$ une production de G. G étant monotone, $m \leqslant n$: complétons alors α avec des ε de manière à écrire $\alpha = \alpha_1 \dots \alpha_n$, où les α_i sont dans $\Sigma \cup N \cup \{\varepsilon\}$. Construisons ensuite G', déduite de G en introduisant les $n-1$ nouveaux non-terminaux $X_1 \dots X_{n-1}$, et en remplaçant $\alpha \to \beta$ par les n les productions suivantes:

$$\begin{array}{rcll} \alpha_1\alpha_2 & \to & \beta_1 X_1 & \\ X_{i-1}\alpha_{i+1} & \to & \beta_i X_i & \forall i, 1 < i < n \\ X_{n-1} & \to & \beta_n & \end{array}$$

Il est clair que G' est monotone et que l'application successive de ces nouvelles règles a bien pour effet global de récrire de proche en proche α en β par:

$$\begin{array}{rcl} \alpha_1\alpha_2 \dots \alpha_m & \underset{G'}{\Rightarrow} & \beta_1 X_1 \alpha_3 \dots \alpha_m \\ & \underset{G'}{\Rightarrow} & \beta_1\beta_2 X_2 \alpha_4 \dots \alpha_n \\ & \underset{G'}{\Rightarrow} & \dots \\ & \underset{G'}{\Rightarrow} & \beta_1\beta_2 \dots \beta_{n-1} X_{n-1} \\ & \underset{G'}{\Rightarrow} & \beta_1\beta_2 \dots \beta_{n-1}\beta_n \end{array}$$

Soit alors u dans $L(G)$: si u se dérive de S sans utiliser $\alpha \to \beta$, alors u se dérive pareillement de S dans G'. Si u se dérive dans G par $S \overset{\star}{\underset{G}{\Rightarrow}} v\alpha w \underset{G}{\Rightarrow} v\beta w \overset{\star}{\underset{G}{\Rightarrow}} u$, alors u se dérive dans G' en remplaçant l'étape de dérivation $v\alpha w \underset{G}{\Rightarrow} v\beta w$ par les étapes de dérivation détaillées ci-dessus. Inversement, si u se dérive dans G' sans utiliser aucune variable X_i, elle se dérive à l'identique dans G. Si X_1 apparaît dans une étape de la dérivation, son élimination par application successive des n règles précédentes implique la présence du facteur α dans le proto-mot dérivant u: u se dérive alors dans G en utilisant $\alpha \to \beta$. □

Par itération de cette transformation, il est possible de transformer toute grammaire monotone en une grammaire monotone équivalente dont toutes les productions sont telles que leur partie droite (et, par conséquent, leur partie gauche, par monotonicité de G) ont une longueur inférieure ou égale à 2. Le lemme suivant nous permet d'arriver à la forme désirée pour les parties gauches (les productions de la forme $A \to a$ pouvant être gardées telles quelles).

Lemma 5.12. *Si G est une grammaire monotone ne contenant que des productions de type $AB \to CD$, avec A, B, C et D dans $N \cup \{\varepsilon\}$, alors il existe une grammaire équivalente satisfaisant la propriété du theorem 5.10.*

Proof. Cette démonstration utilise le même principe que la démonstration précédente, en introduisant pour chaque production $R = AB \rightarrow CD$ le nouveau non-terminal X_{AB} et en remplaçant R par les trois productions suivantes:

— $AB \rightarrow X_{AB}B$

— $X_{AB}B \rightarrow X_{AB}D$

— $X_{AB}D \rightarrow CD$

□

On déduit directement le résultat qui nous intéresse: toute grammaire monotone est équivalente à une grammaire dans laquelle chaque production ne récrit qu'un seul et unique symbole.

Ces résultats permettent finalement d'introduire la notion de langage contextuel.

Definition 5.13 (Langage contextuel)**.** *On appelle* langage contextuel *(ou langage* sensible au contexte*, en abrégé* langage CS*) un langage engendré par une grammaire contextuelle (ou par une grammaire monotone).*

Les langages contextuels constituent une classe importante de langages, que l'on rencontre effectivement (en particulier en traitement automatique des langues). Un représentant notable de ces langages est le langage $\{a^n b^n c^n, n \geq 1\}$, qui est engendré par la grammar 5.2 d'axiome S.

$$\begin{array}{ll} p_1 = S & \rightarrow aSQ \\ p_2 = S & \rightarrow abc \\ p_3 = cQ & \rightarrow Qc \\ p_4 = bQc & \rightarrow bbcc \end{array}$$

Grammar 5.2 – Une grammaire pour $a^n b^n c^n$

La grammar 5.2 est monotone. Dans cette grammaire, on observe par exemple les dérivations listées dans le table 5.3.

Il est possible de montrer qu'en fait les seules dérivations qui réussissent dans cette grammaire produisent les mots (et tous les mots) du langage $\{a^n b^n c^n, n \geq 1\}$, ce qui est d'ailleurs loin d'être évident lorsque l'on examine les productions de la grammaire.

Un dernier résultat important concernant les langages contextuels est le suivant:

Theorem 5.14. *Tout langage contextuel est récursif, soit en notant $\mathcal{RC}$ l'ensemble des langages récursifs, et $\mathcal{CS}$ l'ensemble des langages contextuels: $\mathcal{CS} \subset \mathcal{RC}$.*

Ce que dit ce résultat, c'est qu'il existe un algorithme capable de décider si un mot appartient ou pas au langage engendré par une grammaire contextuelle. Pour s'en convaincre, esquissons le raisonnement suivant: soit u le mot à décider, de taille $|u|$. Tout proto-mot impliqué dans la dérivation de u est au plus aussi long que u, à cause de la propriété de monotonie.

S	$\underset{G}{\overset{p_2}{\Rightarrow}}$	abc
S	$\underset{G}{\overset{p_1}{\Rightarrow}}$	aSQ
	$\underset{G}{\overset{p_2}{\Rightarrow}}$	$aabcQ$
	$\underset{G}{\overset{p_3}{\Rightarrow}}$	$aabQc$
	$\underset{G}{\overset{p_4}{\Rightarrow}}$	$aabbcc$
S	$\underset{G}{\overset{p_1}{\Rightarrow}}$	aSQ
	$\underset{G}{\overset{p_1}{\Rightarrow}}$	$aaSQQ$
	$\underset{G}{\overset{p_2}{\Rightarrow}}$	$aaabcQQ$
	$\underset{G}{\overset{p_3}{\Rightarrow}}$	$aaabQcQ$
	$\underset{G}{\overset{p_4}{\Rightarrow}}$	$aaabbccQ$
	$\underset{G}{\overset{p_3}{\Rightarrow}}$	$aaabbcQc$
	$\underset{G}{\overset{p_3}{\Rightarrow}}$	$aaabbQcc$
	$\underset{G}{\overset{p_4}{\Rightarrow}}$	$aaabbbccc$

Table 5.3 – Des dérivations pour $a^n b^n c^n$

Il "suffit" donc, pour décider u, de construire de proche en proche l'ensemble D de tous les proto-mots qu'il est possible d'obtenir en "inversant les productions" de la grammaire. D contient un nombre fini de proto-mots; il est possible de construire de proche en proche les éléments de cet ensemble. Au terme de processus, si S est trouvé dans D, alors u appartient à $L(G)$, sinon, u n'appartient pas à $L(G)$.

Ce bref raisonnement ne dit rien de la complexité de cet algorithme, c'est-à-dire du temps qu'il mettra à se terminer. Dans la pratique, tous les algorithmes généraux pour les grammaires CS sont exponentiels. Quelques sous-classes particulières admettent toutefois des algorithmes de décision polynomiaux.

Pour finir, notons qu'il est possible de montrer que la réciproque n'est pas vraie, donc qu'il existe des langages récursifs *qui ne peuvent être décrits par aucune grammaire contextuelle*.

5.2.3 Grammaires hors-contexte (type 2)

Une contrainte supplémentaire par rapport à celle imposée pour les grammaires contextuelles consiste à exiger que toutes les productions contextuelles aient un contexte vide. Ceci conduit à la définition suivante.

Definition 5.15 (Grammaire hors-contexte)**.** *On appelle grammaire de type 2 (on dit également* grammaire hors-contexte*, en anglais* Context-free*, ou* grammaire algébrique*, en abrégé* CFG*) une grammaire syntagmatique dans laquelle toute production $\alpha \to \beta$ est de la forme: $A \to \beta$, avec A*

dans N et β dans V^+.

Le sens de cette nouvelle contrainte est le suivant: chaque fois qu'une variable A figure dans un proto-mot, elle peut être récrite *indépendamment du contexte* dans lequel elle apparaît. Par définition, toute grammaire hors-contexte est un cas particulier de grammaire contextuelle.

Cette nouvelle restriction permet d'introduire la notion de A-production:

Definition 5.16 (A-production)**.** *On appelle A-production une production dont la partie gauche est réduite au symbole A.*

Les langages hors-contexte se définissent alors par:

Definition 5.17 (Langage hors-contexte)**.** *On appelle* langage hors-contexte *(ou langage* algébrique*, en abrégé* langage CF*) un langage engendré par une grammaire hors-contexte.*

Attention: il est aisé de faire des définitions complexes d'entités simples. Une grammaire compliquée, par exemple sensible au contexte, peut définir un langage simple, par exemple hors-contexte, rationnel, voire fini.

$$\begin{aligned} p_1 &= S \quad \rightarrow aSb \\ p_2 &= aSb \rightarrow aaSbb \\ p_3 &= S \quad \rightarrow ab \end{aligned}$$

Grammar 5.4 – Une grammaire contextuelle pour $a^n b^n$

Par exemple, la grammar 5.4, contextuelle, engendre le langage hors-contexte $a^n b^n$. Ce langage étant un langage CF, on peut trouver une grammaire CF pour ce langage (par exemple la grammar 5.1).

On notera $\mathcal{CF}$ l'ensemble des langages hors-contexte. Ces langages constituent probablement la classe de langages la plus étudiée et la plus utilisée dans les applications pratiques. Nous aurons l'occasion de revenir en détail sur ces langages dans les chapitres qui suivent et d'en étudier de nombreux exemples, en particulier dans le chapter 6.

Notons simplement, pour l'instant, que cette classe contient des langages non-triviaux, par exemple $a^n b^n$, ou encore le langage des palindromes (dont l'écriture d'une grammaire est laissée en exercice).

5.2.4 Grammaires régulières (type 3)

Régularité

Les grammaires de type 3 réduisent un peu plus la forme des règles autorisées, définissant une classe de langages encore plus simple que la classe des langages hors-contexte.

Definition 5.18 (Grammaire de type 3)**.** *On appelle grammaire de type 3 (on dit également* grammaire régulière*, en abrégé* grammaire RG*) (en anglais* regular*) une grammaire syntagmatique*

dans laquelle toute production $\alpha \to \beta$ est soit de la forme $A \to aB$, avec a dans Σ et A, B dans N, soit de la forme $A \to a$.

Par définition, toute grammaire régulière est hors-contexte. Le trait spécifique des grammaires régulières est que chaque règle produit un symbole terminal et au plus un non-terminal à sa droite. Les dérivations d'une grammaire régulière ont donc une forme très simple, puisque tout proto-mot produit par n étapes successives de dérivation contient en tête n symboles terminaux, suivis d'au plus un non-terminal. La dérivation se termine par la production d'un mot du langage de la grammaire, lorsque le dernier terminal est éliminé par une récriture de type $A \to a$.

La notion de langage régulier se définit comme précédemment par:

Definition 5.19 (Langage régulier)**.** *Un langage est* régulier *si et seulement si il existe une grammaire régulière qui l'engendre.*

On notera $\mathcal{RG}$ l'ensemble des langages réguliers.

Les réguliers sont rationnels

Le résultat principal concernant les langages réguliers est énoncé dans le théorème suivant:

Theorem 5.20. *Si L est un langage régulier, il existe un automate fini A qui reconnaît L. Réciproquement, si L est un langage reconnaissable, avec $\varepsilon \notin L$, alors il existe une grammaire régulière qui engendre L.*

Ainsi les langages réguliers, à un détail près (que nous discutons à la section 5.2.6), ne sont rien d'autre que les langages rationnels, aussi connus sous le nom de reconnaissables. Leurs propriétés principales ont été détaillées par ailleurs (notamment à la section 4.2); nous y renvoyons le lecteur.

Pour avoir l'intuition de ce théorème, considérons la grammaire engendrant le langage $aa(a+b)^\star a$ et l'automate reconnaissant ce langage, tous deux reproduits dans le table 5.5.

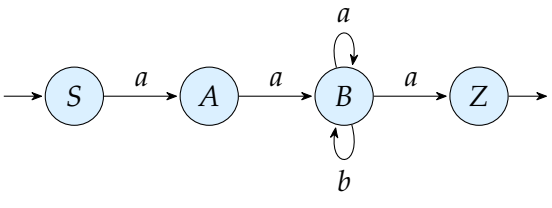
G	A
$S \to aA$ $A \to aB$ $B \to aB \mid bB$ $B \to a$	

Table 5.5 – Grammaire et automate pour $aa(a+b)^\star a$

Dans G, la dérivation de l'entrée $aaba$ est $S \underset{G}{\Rightarrow} aA \underset{G}{\Rightarrow} aaB \underset{G}{\Rightarrow} aabB \underset{G}{\Rightarrow} aaba$, correspondant au calcul: $(S, aabb) \vdash_A (A, abb) \vdash_A (B, bb) \vdash_A (B, a) \vdash_A (F, \varepsilon)$. Cet exemple illustre la symétrie du rôle joué par les variables de la grammaire et les états de l'automate. Cette symétrie est formalisée

dans la construction suivante, qui vise à fournir une démonstration de l'équivalence entre rationnels et réguliers.

Proof du theorem 5.20. Soient $G = (N, \Sigma, P, S)$ une grammaire régulière et A l'automate dérivé de G suivant $A = (\Sigma, N \cup \{Z\}, S, \{Z\}, \delta)$, avec la fonction δ définie selon:

— $\delta(A, a) = B \Leftrightarrow (A \to aB) \in P$

— $\delta(A, a) = Z \Leftrightarrow (A \to a) \in P$

Montrons maintenant que $L(G) = L(A)$ et pour cela, montrons par induction l'équivalence suivante: $S \overset{\star}{\underset{G}{\Rightarrow}} uB$ si et seulement si $(S, u) \vdash_A^\star (B, \varepsilon)$. Cette équivalence est vraie par définition si u est de longueur 1. Supposons-la vraie jusqu'à une longueur n, et considérons: $u = avb$ tel que $S \overset{\star}{\underset{G}{\Rightarrow}} avA \underset{G}{\Rightarrow} avbB$. Par l'hypothèse de récurrence il s'ensuit que $(S, av) \vdash_A^\star (A, \varepsilon)$, et donc que $(S, avb) \vdash_A^\star (B, \varepsilon)$ par construction de δ. Pour conclure, reste à examiner comment une dérivation se termine: il faut nécessairement éliminer la dernière variable par une production de type $A \to a$; ceci correspond à une transition dans l'état Z, unique état final de A. Inversement, un calcul réussi dans A correspond à une dérivation "éliminant" toutes les variables et donc à un mot du langage $L(G)$.

La construction réciproque, permettant de construire une grammaire équivalente à un automate fini quelconque, est exactement inverse: il suffit d'associer une variable à chaque état de l'automate (en prenant l'état initial pour axiome) et de rajouter une production par transition. Cette construction est achevée en rajoutant une nouvelle production $A \to a$ pour toute transition $\delta(A, a)$ aboutissant dans un état final. □

Une conséquence de cette équivalence est que si tout langage régulier est par définition hors contexte, le contraire n'est pas vrai. Nous avons rencontré plus haut (à la section 5.2.3) une grammaire engendrant $\{a^n b^n, n \geq 1\}$, langage dont on montre aisément qu'en vertu du lemme de pompage pour les langages rationnels (cf. la section 4.3.1), il ne peut être reconnu par un automate fini. La distinction introduite entre langages réguliers et langages hors-contexte n'est donc pas de pure forme: il existe des langages CF qui ne sont pas rationnels.

Exercise 5.21 (BCI-0405-1, suite (exercise 4.15))**.** *2 Construisez, par une méthode du cours, une grammaire régulière G telle que $L(G) = L(A)$, où A est l'automate 4.14.*

Variantes

Il est possible de définir de manière un peu plus libérale les grammaires de type 3: la limitation essentielle concerne en fait le nombre de non-terminaux en partie droite et leur positionnement systématique à droite de tous les terminaux.

Definition 5.22 (Grammaire de type 3)**.** *On appelle grammaire de type 3 (ici* grammaire linéaire à droite*) (en anglais* right linear*) une grammaire syntagmatique dans laquelle toute production $\alpha \to \beta$ est soit de la forme: $A \to uB$, avec u dans $\Sigma^\star$ et A, B dans N, soit de la forme $A \to u$, avec u dans Σ^+.*

Cette définition est donc plus générale que la définition des grammaires régulières. Il est pourtant simple de montrer que les langages engendrés sont les mêmes: chaque grammaire linéaire à droite admet une grammaire régulière équivalente. La démonstration est laissée en exercice. Les grammaires linéaires à droite engendrent donc exactement la classe des langages rationnels.

On montre, de même, que les *grammaires linéaires à gauche* (au plus un non-terminal dans chaque partie droite, toujours positionné en première position) engendrent les langages rationnels. Ces grammaires sont donc aussi des grammaires de type 3.

Attention: ces résultats ne s'appliquent plus si l'on considère les *grammaires linéaires* quelconques (définies comme étant les grammaires telles que toute partie droite contient au plus un non-terminal): *les grammaires linéaires peuvent engendrer des langages hors-contexte non rationnels*. Pour preuve, il suffit de considérer de nouveau la grammar 5.1.

On ne peut pas non plus "généraliser" en acceptant les grammaires dont les règles panachent linéarité à gauche *et* à droite. Il est facile de reconnaître la grammar 5.1 dans le déguisement linéaire suivant.

$$S \to aX \mid ab$$
$$X \to Sb$$

5.2.5 Grammaires à choix finis (type 4)

Il existe des grammaires encore plus simples que les grammaires régulières. En particulier les grammaires à choix finis sont telles que tout non-terminal ne récrit que des terminaux. On appelle de telles grammaires les grammaires de type 4:

Definition 5.23 (Grammaire de type 4)**.** *On appelle grammaire de type 4 (on dit également* grammaire à choix finis*, en abrégé* grammaire FC*) une grammaire syntagmatique dans laquelle toute production est de la forme: $A \to u$, avec A dans N et u dans Σ^+.*

Les grammaires de type 4, on s'en convaincra aisément, n'engendrent que des langages finis, mais engendrent tous les langages finis qui ne contiennent pas le mot vide.

5.2.6 Les productions ε

La définition que nous avons donnée des grammaires de type 1 implique un accroissement monotone de la longueur des proto-mots au cours d'une dérivation, ce qui interdit la production du mot vide. Cette "limitation" théorique de l'expressivité des grammaires contextuelles a conduit à énoncer une version "faible" (c'est-à-dire ne portant que sur les reconnaissables ne contenant pas le mot vide) de l'équivalence entre langages réguliers et langages rationnels.

Pour résoudre cette limitation, on admettra qu'une grammaire contextuelle (ou hors-contexte, ou régulière) puisse contenir des règles de type $A \to \varepsilon$ et on acceptera la possibilité qu'un langage contextuel contienne le mot vide. Pour ce qui concerne les grammaires CF, nous reviendrons sur ce point en étudiant les algorithmes de normalisation de grammaires (à la section 9.1.4).

Pour ce qui concerne les grammaires régulières, notons simplement que si l'on accepte des productions de type $A \rightarrow \varepsilon$, alors on peut montrer que les langages réguliers sont exactement les langages reconnaissables. Il suffit pour cela de compléter la construction décrite à la section 5.2.4 en marquant comme états finaux de A tous les états correspondant à une variable pour laquelle il existe une règle $A \rightarrow \varepsilon$. La construction inverse en est également simplifiée: pour chaque état final A de l'automate, on rajoute la production $A \rightarrow \varepsilon$.

Pour différentier les résultats obtenus avec et sans ε, on utilisera le qualificatif de *strict* pour faire référence aux classes de grammaires et de langages présentées dans les sections précédentes: ainsi on dira qu'une grammaire est *strictement hors-contexte* si elle ne contient pas de production de type $A \rightarrow \varepsilon$; qu'elle est simplement hors-contexte sinon. De même on parlera de langage strictement hors-contexte pour un langage CF ne contenant pas ε et de langage hors-contexte sinon.

5.2.7 Conclusion

Le titre de la section introduisait le terme de hiérarchie: quelle est-elle finalement? Nous avons en fait montré dans cette section la série d'inclusions suivante:

$$\mathcal{FC} \subset \mathcal{RG} \subset \mathcal{CF} \subset \mathcal{CS} \subset \mathcal{RC} \subset \mathcal{RE}$$

Il est important de bien comprendre le sens de cette hiérarchie : les grammaires les plus complexes ne décrivent pas des langages plus grands, mais permettent d'exprimer des distinctions plus subtiles entre les mots du langage de la grammaire et ceux qui sont ne sont pas grammaticaux.

Un autre point de vue, sans doute plus intéressant, consiste à dire que les grammaires plus complexes permettent de décrire des "lois" plus générales que celles exprimables par des grammaires plus simples. Prenons un exemple en traitement des langues naturelles: en français, le bon usage impose que le sujet d'une phrase affirmative s'accorde avec son verbe en nombre et personne. Cela signifie, par exemple, que si le sujet est un pronom singulier à la première personne, le verbe sera également à la première personne du singulier. Si l'on appelle L l'ensemble des phrases respectant cette règle, on a:

— la phrase *l'enfant mange* est dans L

— la phrase *l'enfant manges* n'est pas dans L

Une description de la syntaxe du français par une grammaire syntagmatique qui voudrait exprimer cette règle avec une grammaire régulière est peut-être possible, mais serait certainement très fastidieuse, et sans aucune vertu pour les linguistes, ou pour les enfants qui apprennent cette règle. Pourquoi ? Parce qu'en français, le sujet peut être séparé du verbe par un nombre arbitraire de mots, comme dans *l'enfant de Jean mange*, *l'enfant du fils de Jean mange*... Il faut donc implanter dans les règles de la grammaire un dispositif de mémorisation du nombre et de la personne du sujet qui "retienne" ces valeurs jusqu'à ce que le verbe soit rencontré, c'est-à-dire pendant un nombre arbitraire d'étapes de dérivation. La seule solution, avec une grammaire régulière, consiste à envisager à l'avance toutes les configurations possibles, et de les encoder dans la topologie de l'automate correspondant, puisque la mémorisation "fournie" par la grammaire est de taille 1 (le passé de la dérivation

est immédiatement oublié). On peut montrer qu'une telle contrainte s'exprime simplement et sous une forme bien plus naturelle pour les linguistes, lorsque l'on utilise une grammaire CF.

La prix à payer pour utiliser une grammaire plus fine est que le problème algorithmique de la reconnaissance d'une phrase par une grammaire devient de plus en plus complexe. Vous savez déjà que ce problème est solvable en temps linéaire pour les langages rationnels (réguliers), et non-solvable (indécidable) pour les langages de type 0. On peut montrer qu'il existe des automates généralisant le modèle d'automate fini pour les langages CF et CS, d'où se déduisent des algorithmes (polynomiaux pour les langages CF, exponentiels en général pour les langages CS) permettant également de décider ces langages. En particulier, le problème de la reconnaissance pour les grammaires CF sera étudié en détail dans les chapters 7 and 8.

Chapter 6

Langages et grammaires hors-contexte

Dans ce chapitre, nous nous intéressons plus particulièrement aux grammaires et aux langages hors-contexte. Rappelons que nous avons caractérisé ces grammaires à la section 5.2.3 comme étant des grammaires syntagmatiques dont toutes les productions sont de la forme $A \to u$, avec A un non-terminal et u une séquence quelconque de terminaux et non-terminaux. Autrement dit, une grammaire algébrique est une grammaire pour laquelle il est toujours possible de récrire un non-terminal en cours de dérivation, quel que soit le contexte (les symboles adjacents dans le proto-mot) dans lequel il apparaît.

Ce chapitre débute par la présentation, à la section 6.1, de quelques grammaires CF exemplaires. Cette section nous permettra également d'introduire les systèmes de notation classiques pour ces grammaires. Nous définissons ensuite la notion de dérivation gauche et d'arbre de dérivation, puis discutons le problème de l'équivalence entre grammaires et de l'ambiguïté (section 6.2). La section 6.3 introduit enfin un certain nombre de propriétés élémentaires des langages CF, qui nous permettent de mieux cerner la richesse (et la complexité) de cette classe de langages. L'étude des grammaires CF, en particulier des algorithmes permettant de traiter le problème de la reconnaissance d'un mot par une grammaire, sera poursuivie dans les chapitres suivants, en particulier au chapter 7.

6.1 Quelques exemples

6.1.1 La grammaire des déjeuners du dimanche

La grammar 6.1 est un exemple de grammaire hors-contexte correspondant à une illustration très simplifiée de l'utilisation de ces grammaires pour des applications de traitement du langage naturel. Cette grammaire engendre un certain nombre d'énoncés du français. Dans cette grammaire, les non-terminaux sont en majuscules, les terminaux sont des mots du vocabulaire usuel. On utilise également dans cette grammaire le symbole "|" pour exprimer une alternative: $A \to u \mid v$ vaut pour les deux règles $A \to u$ et $A \to v$.

Première remarque sur la grammar 6.1: elle contient de nombreuses règles de type $A \to a$, qui servent simplement à introduire les symboles terminaux de la grammaire: les symboles

p_1	S	$\rightarrow GN\ GV$	p_{15}	V	$\rightarrow mange \mid sert$
p_2	GN	$\rightarrow DET\ N$	p_{16}	V	$\rightarrow donne$
p_3	GN	$\rightarrow GN\ GNP$	p_{17}	V	$\rightarrow boude \mid s'ennuie$
p_4	GN	$\rightarrow NP$	p_{18}	V	$\rightarrow parle$
p_5	GV	$\rightarrow V$	p_{19}	V	$\rightarrow coupe \mid avale$
p_6	GV	$\rightarrow V\ GN$	p_{20}	V	$\rightarrow discute \mid gronde$
p_7	GV	$\rightarrow V\ GNP$	p_{21}	NP	$\rightarrow Louis \mid Paul$
p_8	GV	$\rightarrow V\ GN\ GNP$	p_{22}	NP	$\rightarrow Marie \mid Sophie$
p_9	GV	$\rightarrow V\ GNP\ GNP$	p_{23}	N	$\rightarrow fille \mid maman$
p_{10}	GNP	$\rightarrow PP\ GN$	p_{24}	N	$\rightarrow paternel \mid fils$
p_{11}	PP	$\rightarrow de \mid à$	p_{25}	N	$\rightarrow viande \mid soupe \mid salade$
p_{12}	DET	$\rightarrow la \mid le$	p_{26}	N	$\rightarrow dessert \mid fromage \mid pain$
p_{13}	DET	$\rightarrow sa \mid son$	p_{27}	ADJ	$\rightarrow petit \mid gentil$
p_{14}	DET	$\rightarrow un \mid une$	p_{28}	ADJ	$\rightarrow petite \mid gentille$

Grammar 6.1 – La grammaire G_D des repas dominicaux

apparaissant en partie gauche de ces productions sont appelés *pré-terminaux*. En changeant le vocabulaire utilisé dans ces productions, on pourrait facilement obtenir une grammaire permettant d'engendrer des énoncés décrivant d'autres aspects de la vie quotidienne. Il est important de réaliser que, du point de vue de leur construction (de leur structure interne), les énoncés ainsi obtenus resteraient identiques à ceux de $L(G_D)$.

En utilisant la grammar 6.1, on construit une première dérivation pour l'énoncé *Louis boude*, représentée dans le table 6.2:

$S \underset{G_D}{\Rightarrow} GN\ GV$	(par p_1)
$S \underset{G_D}{\Rightarrow} NP\ GV$	(par p_4)
$\underset{G_D}{\Rightarrow} Louis\ GV$	(par p_{21})
$\underset{G_D}{\Rightarrow} Louis\ V$	(par p_5)
$\underset{G_D}{\Rightarrow} Louis\ boude$	(par p_{17})

Table 6.2 – Louis boude

Il existe d'autres dérivations de *Louis boude*, consistant à utiliser les productions dans un ordre différent: par exemple p_5 avant p_4, selon: $S \underset{G_D}{\Rightarrow} GN\ GV \underset{G_D}{\Rightarrow} GN\ V \underset{G_D}{\Rightarrow} GN\ boude \underset{G_D}{\Rightarrow} NP\ boude \underset{G_D}{\Rightarrow} Louis\ boude$. Ceci illustre une première propriété importante des grammaires hors contexte: lors d'une dérivation, il est possible d'appliquer les productions dans un ordre arbitraire. Notons également qu'à la place de *Louis*, on aurait pu utiliser *Marie* ou *Paul*, ou même *le fils boude* et obtenir un nouveau mot du langage: à nouveau, c'est la propriété d'indépendance au contexte qui s'exprime. Ces exemples éclairent un peu la signification des noms de variables: *GN* désigne les groupes nominaux, *GV* les groupes verbaux... La

première production de G_D dit simplement qu'un énoncé bien formé est composé d'un groupe nominal (qui, le plus souvent, est le sujet) et d'un groupe verbal (le verbe principal) de la phrase.

Cette grammaire permet également d'engendrer des énoncés plus complexes, comme par exemple:

le paternel sert le fromage

qui utilise une autre production (p_6) pour dériver un groupe verbal (GV) contenant un verbe et son complément d'objet direct.

La production p_3 est particulière, puisqu'elle contient le même symbole dans sa partie gauche et dans sa partie droite. On dit d'une production possédant cette propriété qu'elle est *récursive*. Pour être plus précis p_3 est récursive à gauche: le non-terminal en partie gauche de la production figure également en tête de la partie droite. Comme c'est l'élément le plus à gauche de la partie droite, on parle parfois de *coin gauche* de la production.

Cette propriété de p_3 implique immédiatement que le langage engendré par G_D est infini, puisqu'on peut créer des énoncés de longueur arbitraire par application itérée de cette règle, engendrant par exemple:

— *le fils de Paul mange*
— *le fils de la fille de Paul mange*
— *le fils de la fille de la fille de Paul mange*
— *le fils de la fille de la fille ... de Paul mange*

Il n'est pas immédiatement évident que cette propriété, à savoir qu'une grammaire contenant une règle récursive engendre un langage infini, soit toujours vraie... Pensez, par exemple, à ce qui se passerait si l'on avait une production récursive de type $A \rightarrow A$. En revanche, il est clair qu'une telle production risque de poser problème pour engendrer de manière systématique les mots de la grammaire. Le problème est d'éviter de tomber dans des dérivations interminables telles que: $GN \underset{G_D}{\Rightarrow} GN\ GNP \underset{G_D}{\Rightarrow} GN\ GNP\ GNP \underset{G_D}{\Rightarrow} GN\ GNP\ GNP\ GNP \ldots$.

Une autre remarque sur G_D: cette grammaire engendre des énoncés qui ne sont pas corrects en français. Par exemple, *la fils avale à le petite dessert*.

Dernière remarque concernant G_D: cette grammaire engendre des énoncés qui sont (du point de vue du sens) ambigus. Ainsi *Louis parle à la fille de Paul*, qui peut exprimer soit une conversation entre *Louis* et *la fille de Paul*, soit un échange concernant *Paul* entre *Louis* et *la fille*. En écrivant les diverses dérivations de ces deux énoncés, vous pourrez constater que les deux sens correspondent à deux dérivations employant des productions différentes pour construire un groupe verbal: la première utilise p_7, la seconde p_9.

6.1.2 Une grammaire pour le shell

Dans cette section, nous présentons des fragments [1] d'une autre grammaire, celle qui décrit la syntaxe des programmes pour l'interpréteur `bash`. Nous ferons, dans la suite, référence à cette grammaire sous le nom de G_B.

1. Ces fragments sont extraits de *Learning the Bash Shell*, de C. Newham et B. Rosenblatt, O'Reilly & associates, 1998, consultable en ligne à l'adresse http://safari.oreilly.com.

$$
\begin{aligned}
\langle \texttt{number} \rangle &\underset{G_B}{\Rightarrow} \langle \texttt{number} \rangle\ \texttt{0} \\
&\underset{G_B}{\Rightarrow} \langle \texttt{number} \rangle\ \texttt{10} \\
&\underset{G_B}{\Rightarrow} \langle \texttt{number} \rangle\ \texttt{510} \\
&\underset{G_B}{\Rightarrow} \texttt{3510}
\end{aligned}
$$

Table 6.4 – Dérivation du nombre `3510`

Un mot tout d'abord sur les notations: comme il est d'usage pour les langages informatiques, cette grammaire est exprimée en respectant les conventions initialement proposées par Backus et Naur pour décrire le langage ALGOL 60. Ce système de notation des grammaires est connu sous le nom de *Backus-Naur Form* ou en abrégé BNF. Dans les règles de la grammar 6.3, les non-terminaux figurent entre chevrons (⟨⟩); les terminaux sont les autres chaînes de caractères, certains caractères spéciaux apparaissant entre apostrophes; les productions sont marquées par l'opérateur ::=; enfin les productions alternatives ayant même partie gauche sont séparées par le symbole |, les alternatives pouvant figurer sur des lignes distinctes.

Les éléments de base de la syntaxe sont les mots et les chiffres, définis dans la grammar 6.3.

```
⟨letter⟩ ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
           | A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9

⟨number⟩ ::= ⟨number⟩ ⟨digit⟩
           | ⟨digit⟩

⟨word⟩ ::= ⟨word⟩ ⟨letter⟩
         | ⟨word⟩ '_'
         | ⟨letter⟩

⟨word_list⟩ ::= ⟨word_list⟩ ⟨word⟩
              |  ⟨word⟩
```

Grammar 6.3 – Constructions élémentaires de Bash

Les deux premières définitions sont des simples énumérations de terminaux définissant les lettres (⟨`letter`⟩), puis les chiffres (⟨`digit`⟩). Le troisième non-terminal, ⟨`number`⟩, introduit une figure récurrente dans les grammaires informatiques: celle de la liste, ici une liste de chiffres. Une telle construction nécessite deux productions:

— une production récursive (ici récursive à gauche) spécifie une liste comme une liste suivie d'un nouvel élément;

— la seconde alternative achève la récursion, en définissant la liste composée d'un unique chiffre.

Ce couple de règles permet de dériver des séquences de longueur arbitraire, selon des dérivations similaires à celle du table 6.4. On reconnaît dans celle-ci une dérivation régulière: le fragment de G_B réduit aux trois productions relatives aux nombres et d'axiome ⟨`number`⟩ définit une grammaire régulière et pourrait être représenté sous la forme d'un automate fini. Le même principe est à l'œuvre pour décrire les mots, ⟨`word`⟩, par des listes de lettres; ou encore les listes de mots (⟨`word_list`⟩).

Une seconde figure remarquable et typique des langages de programmation apparaît dans les productions du table 6.5.

```
⟨group_command⟩ ::=  '{' ⟨list⟩ '}'

⟨if_command⟩ ::= if ⟨compound_list⟩ then ⟨compound_list⟩ fi
               | if ⟨compound_list⟩ then ⟨compound_list⟩ else ⟨compound_list⟩ fi
               | if ⟨compound_list⟩ then ⟨compound_list⟩ ⟨elif_clause⟩ fi

⟨elif_clause⟩ ::= elif ⟨compound_list⟩ then ⟨compound_list⟩
                | elif ⟨compound_list⟩ then ⟨compound_list⟩ else ⟨compound_list⟩
                | elif ⟨compound_list⟩ then ⟨compound_list⟩ ⟨elif_clause⟩
```

Table 6.5 – Constructions parenthésées

La définition du non-terminal ⟨`group_command`⟩ fait apparaître deux caractères spéciaux *appariés* { (ouvrant) et } (fermant): dans la mesure où cette production est la seule qui introduit ces symboles, il est garanti qu'à chaque ouverture de { correspondra une fermeture de }, et ceci indépendamment du nombre et du degré d'imbrication de ces symboles. Inversement, si cette condition n'est pas satisfaite dans un programme, une erreur de syntaxe sera détectée.

Le cas des constructions conditionnelles est similaire: trois constructions sont autorisées, correspondant respectivement à la construction sans alternative (simple `if-then`), construction alternative unique (`if-then-else`), ou construction avec alternatives multiples, (`if-then-elif`...). Dans tous les cas toutefois, chaque mot-clé `if` (ouvrant) doit s'apparier avec une occurrence mot-clé `fi` (fermant), quel que soit le contenu délimité par le non-terminal ⟨`compound_list`⟩. Les constructions parenthésées apparaissent, sous de multiples formes, dans tous les langages de programmation. Sous leur forme la plus épurée, elles correspondent à des langages de la forme $a^n b^n$, qui sont des langages hors-contexte, mais pas réguliers (cf. la discussion de la section 5.2.4).

6.2 Dérivations

Nous l'avons vu, une première caractéristique des grammaires CF est la multiplicité des dérivations possibles pour un même mot. Pour "neutraliser" cette source de variabilité, nous allons poser une convention permettant d'ordonner l'application des productions. Cette convention posée, nous introduisons une représentation graphique des dérivations et définissons les notions d'ambiguïté et d'équivalence entre grammaires.

6.2.1 Dérivation gauche

Definition 6.1 (Dérivation gauche)**.** *On appelle* dérivation gauche *d'une grammaire hors-contexte G une dérivation dans laquelle chaque étape de dérivation récrit le non-terminal le plus à gauche du proto-mot courant.*

En guise d'illustration, considérons de nouveau la dérivation de *Louis boude* présentée dans le table 6.2: chaque étape récrivant le non-terminal le plus à gauche, cette dérivation est bien une dérivation gauche.

Un effort supplémentaire est toutefois requis pour rendre la notion de dérivation gauche complètement opératoire. Imaginez, par exemple, que la grammaire contienne une règle de type $A \to A$. Cette règle engendre une infinité de dérivations gauches équivalentes pour toute dérivation gauche contenant A: chaque occurrence de cette production peut être répétée à volonté sans modifier le mot engendré. Pour achever de spécifier la notion de dérivation gauche, on ne considérera dans la suite que des dérivations gauches *minimales*, c'est-à-dire qui sont telles que chaque étape de la dérivation produit un proto-mot différent de ceux produits aux étapes précédentes.

On notera $A \underset{G}{\overset{L}{\Rightarrow}} u$ lorsque u dérive de A par une dérivation gauche.

De manière duale, on définit la notion de dérivation droite:

Definition 6.2 (Dérivation droite)**.** *On appelle* dérivation droite *d'une grammaire hors-contexte G une dérivation dans laquelle chaque étape de la dérivation récrit le terminal le plus à droite du proto-mot courant.*

Un premier résultat est alors énoncé dans le théorème suivant, qui nous assure qu'il est justifié de ne s'intéresser qu'aux seules dérivations gauches d'une grammaire.

Theorem 6.3. *Soit G une grammaire CF d'axiome S, et u dans* $\Sigma^\star$*:* $S \underset{G}{\overset{\star}{\Rightarrow}} u$ *si et seulement si* $S \underset{G}{\overset{L}{\Rightarrow}} u$ *(idem pour* $S \underset{G}{\overset{R}{\Rightarrow}} u$*).*

Proof. Un sens de l'implication est immédiat: si un mot u se dérive par une dérivation gauche depuis S, alors $u \in L(G)$. Réciproquement, soit $u = u_1 \dots u_n$ se dérivant de S par une dérivation non-gauche, soit B le premier non-terminal non-gauche récrit dans la dérivation de u (par $B \to \beta$) et soit A le non-terminal le plus à gauche de ce proto-mot. La dérivation de u s'écrit:

$$S \underset{G}{\overset{\star}{\Rightarrow}} u_1 \dots u_i A u_j \dots u_k B \gamma \underset{G}{\Rightarrow} u_1 \dots u_i A u_j \dots u_k \beta \gamma \underset{G}{\overset{\star}{\Rightarrow}} u$$

La génération de u implique qu'à une étape ultérieure de la dérivation, le non-terminal A se récrive (éventuellement en plusieurs étapes) en: $u_{i+1} \dots u_{j-1}$. Soit alors $A \to \alpha$ la première production impliquée dans cette dérivation: il apparaît qu'en appliquant cette production juste avant $B \to \beta$, on obtient une nouvelle dérivation de u depuis S, dans laquelle A, qui est plus à gauche que B, est récrit avant lui. En itérant ce procédé, on montre que toute dérivation non-gauche de u peut se transformer de proche en proche en dérivation gauche de u, précisément ce qu'il fallait démontrer. □

Attention: si l'on peut dériver par dérivation gauche tous les mots d'un langage, on ne peut pas en dire autant des proto-mots. Il peut exister des proto-mots qui ne sont pas accessibles par une dérivation gauche: ce n'est pas gênant, dans la mesure où ces proto-mots ne permettent pas de dériver davantage de mots de la grammaire.

En utilisant ce résultat, il est possible de définir une relation d'équivalence sur l'ensemble des dérivations de G: deux dérivations sont équivalentes si elles se transforment en une même dérivation gauche. Il apparaît alors que ce qui caractérise une dérivation, ou plutôt une classe de dérivations, est partiellement [2] indépendant de l'ordre d'application des productions, et peut donc simplement être résumé par l'ensemble [3] des productions utilisées. Cet ensemble partiellement ordonné admet une expression mathématique (et visuelle): *l'arbre de dérivation*.

6.2.2 Arbre de dérivation

Definition 6.4 (Arbre de dérivation)**.** *Un* arbre de dérivation *dans G est un arbre A tel que:*

- *tous les nœuds de A sont étiquetés par un symbole de $V = T \cup N$,*
- *la racine est étiquetée par S,*
- *si un nœud n n'est pas une feuille et porte l'étiquette X, alors $X \in N$,*
- *si $n_1, n_2, \ldots, n_k$ sont les fils de n dans A, d'étiquettes respectives $X_1, X_2, \ldots, X_k$, alors $X \to X_1 X_2 \cdots X_k$ est une production de G.*

Notons que cette définition n'impose pas que toutes les feuilles de l'arbre soient étiquetées par des terminaux: un arbre peut très bien décrire un proto-mot en cours de dérivation.

Pour passer de l'arbre de dérivation à la dérivation proprement dite, il suffit de parcourir l'arbre de dérivation en lisant les productions appliquées. La dérivation gauche s'obtient en effectuant un parcours *préfixe* de l'arbre, c'est-à-dire en visitant d'abord un nœud père, puis tous ses fils de gauche à droite. D'autres parcours fourniront d'autres dérivations équivalentes.

Un arbre de dérivation (on parle aussi d'*arbre d'analyse* d'un mot) correspondant à la production de l'énoncé *Paul mange son fromage* dans la grammaire des "dimanches" est représenté à la fig. 6.6. Les numéros des nœuds sont portés en indice des étiquettes correspondantes; la numérotation adoptée correspond à un parcours préfixe de l'arbre. Sur cette figure, on voit en particulier qu'il existe deux nœuds étiquetés *GN*, portant les indices 2 et 8.

L'arbre de dérivation représente graphiquement la *structure* associée à un mot du langage, de laquelle se déduira *l'interprétation* à lui donner.

On appelle *parsage* (en anglais *parsing*) d'un mot dans la grammaire le processus de construction du ou des arbres de dérivation pour ce mot, lorsqu'ils existent.

L'interprétation correspond à la signification d'un énoncé pour la grammaire des dimanches, à la valeur du calcul dénoté par une expression arithmétique pour une grammaire de calculs,

2. Partiellement seulement: pour qu'un non-terminal A soit dérivé, il faut qu'il est été préalablement produit par une production qui le contient dans sa partie droite.

3. En réalité un multi-ensemble, pas un ensemble au sens traditionnel: le nombre d'applications de chaque production importe.

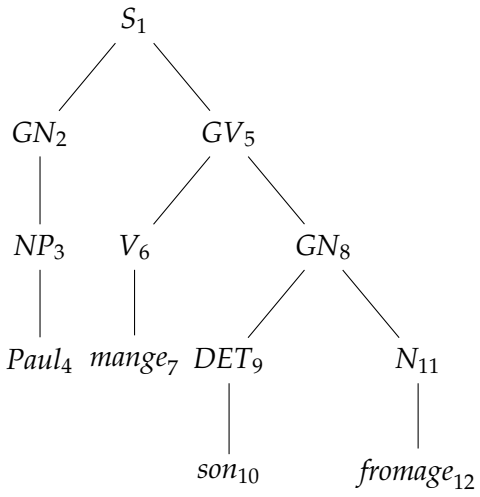


Figure 6.6 – L'arbre de dérivation de *Paul mange son fromage*

ou encore à la séquence d'instructions élémentaires à effectuer dans le cadre d'une grammaire représentant un langage informatique. On appelle *sémantique* d'un mot le résultat de son interprétation, et par extension *la sémantique* le domaine (formel) traitant les problèmes d'interprétation. *Sémantique* s'oppose ainsi à *syntaxique* aussi bien lorsque l'on parle de langage informatique ou de langage humain. On notera que dans les deux cas, il existe des phrases syntaxiques qui n'ont pas de sens, comme $0 = 1$, ou encore *le dessert s'ennuie Marie*.

Notons que la structure associée à un mot n'est pas nécessairement unique, comme le montre l'exemple suivant. Soit G_E une grammaire d'axiome *Sum* définissant des expressions arithmétiques simples en utilisant les productions de la grammar 6.7.

$$\begin{aligned} \mathit{Sum} &\rightarrow \mathit{Sum} + \mathit{Sum} \mid \mathit{Number} \\ \mathit{Number} &\rightarrow \mathit{Number}\ \mathit{Digit} \mid \mathit{Digit} \\ \mathit{Digit} &\rightarrow 0 \mid 1 \mid 2 \ldots \mid 9 \end{aligned}$$

Grammar 6.7 – Une grammaire pour les sommes

Pour cette grammaire, l'expression $3 + 5 + 1$ correspond à deux arbres d'analyse, représentés à la fig. 6.8.

En remplaçant l'opérateur $+$, qui est associatif, par $-$, qui ne l'est pas, on obtiendrait non seulement deux analyses syntaxiques différentes du mot, mais également deux interprétations (résultats) différents: -3 dans un cas, -1 dans l'autre. Ceci est fâcheux.

6.2.3 Ambiguïté

Definition 6.5 (Ambiguïté d'une grammaire)**.** *Une grammaire est* ambiguë *s'il existe un mot admettant plusieurs dérivations gauches dans la grammaire.*

De manière équivalente, une grammaire est ambiguë s'il existe un mot qui admet plusieurs arbres de dérivation.

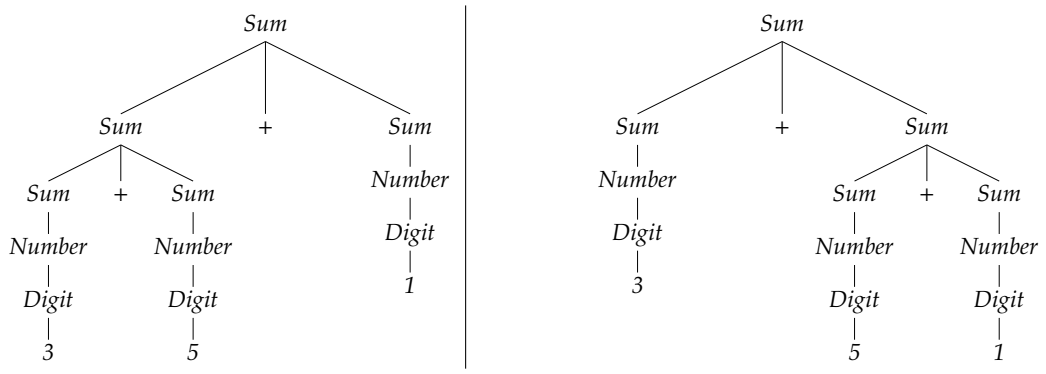


Figure 6.8 – Deux arbres de dérivation d'un même calcul

La grammar 6.9 est un exemple de grammaire ambiguë.

$$\begin{aligned} S &\rightarrow ASB \mid AB \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid b \end{aligned}$$

Grammar 6.9 – Une grammaire ambiguë

Le mot *aabb* admet ainsi plusieurs analyses, selon que l'on utilise ou pas la première production de S. Cette grammaire engendrant un langage régulier (lequel ?), il est toutefois élémentaire, en utilisant le procédé de transformation d'un automate en grammaire régulière, de construire une grammaire non-ambiguë qui reconnaît ce langage. L'ambiguïté est une propriété des grammaires.

Cette propriété contamine pourtant parfois même les langages :

Definition 6.6 (Ambiguïté d'un langage)**.** *Un langage hors-contexte est (intrinsèquement)* ambigu *si toutes les grammaires hors-contexte qui l'engendrent sont ambiguës.*

Un langage intrinsèquement ambigu ne peut donc être décrit par une grammaire non-ambiguë. Un exemple fameux est $L = L_1 \cup L_2 = \{a^n b^n c^m\} \cup \{a^m b^n c^n\}$, soit $\{a^n b^m c^p$, avec $m = n$ ou $m = p\}$. Décrire L_1 demande d'introduire une récursion centrale, pour apparier les a et les b; L_2 demande également une règle exhibant une telle récursion, pour apparier les b et les c. On peut montrer que pour toute grammaire hors-contexte engendrant ce langage, tout mot de la forme $a^n b^n c^n$ aura une double interprétation, selon que l'on utilise le mécanisme d'appariement (de comptage) des a et des b ou celui qui contrôle l'appariement des b et des c.

Fort heureusement, les langages (informatiques) intrinsèquement ambigus ne courent pas les rues: lorsque l'on conçoit un nouveau langage informatique, il suffit de se prémunir contre les ambiguïtés qui peuvent conduire à des conflits d'interprétation. Ainsi, les grammaires formelles utilisées pour les langages de programmation sont-elles explicitement conçues pour limiter l'ambiguïté d'analyse.

En revanche, lorsqu'on s'intéresse au langage naturel, l'ambiguïté lexicale (un mot ayant

plusieurs catégories ou plusieurs sens) et syntaxique (un énoncé avec plusieurs interprétations) sont des phénomènes massifs et incontournables, qu'il faut donc savoir affronter avec les outils adéquats.

6.2.4 Équivalence

À travers la notion de dérivation gauche et d'arbre de dérivation, nous sommes en mesure de préciser les différentes notions d'équivalence pouvant exister entre grammaires.

Definition 6.7 (Équivalence)**.** *Deux grammaires G_1 et G_2 sont* équivalentes *si et seulement si elles engendrent le même langage (definition 5.5).*

Si, de plus, pour tout mot du langage, les arbres de dérivation dans G_1 et dans G_2 sont identiques, on dit que G_1 et G_2 sont fortement équivalentes*. Dans le cas contraire, on dit que G_1 et G_2 sont* faiblement équivalentes*.*

Ces notions sont importantes puisque, on l'a vu, c'est l'arbre de dérivation qui permet d'interpréter le sens d'un énoncé: transformer une grammaire G_1 en une autre grammaire G_2 qui reconnaît le même langage est utile, mais il est encore plus utile de pouvoir le faire *sans avoir à changer les interprétations*. À défaut, il faudra se résoudre à utiliser des mécanismes permettant de reconstruire les arbres de dérivation de la grammaire initiale à partir de ceux de la grammaire transformée.

6.3 Les langages hors-contexte

6.3.1 Le lemme de pompage

Bien que se prêtant à de multiples applications pratiques, les grammaires algébriques sont intrinsèquement limitées dans la structure des mots qu'elles engendrent. Pour mieux appréhender la nature de cette limitation, nous allons introduire quelques notions nouvelles. Si a est un symbole terminal d'un arbre de dérivation d'une grammaire G, on appelle *lignée* de a la séquence de règles utilisée pour produire a à partir de S. Chaque élément de la lignée est une paire (P, i), où P est une production, et i l'indice dans la partie droite de P de l'ancêtre de a. Considérant de nouveau la grammar 5.1 pour le langage $a^n b^n$ et une dérivation de *aaabbb* dans cette grammaire, la lignée du second a de *aaabbb* correspond à la séquence $(S \to aSb, 2)$, $(S \to aSb, 1)$.

On dit alors qu'un symbole est *original* si tous les couples (P, i) qui constituent sa lignée sont différents. Contrairement au premier et au second a de *aaabbb*, le troisième a n'est pas original, puisque sa lignée est $(S \to aSb, 2)$, $(S \to aSb, 2)$, $(S \to ab, 1)$. Par extension, un mot est dit *original* si tous les symboles qui le composent sont originaux.

Le résultat intéressant est alors qu'une grammaire algébrique, même lorsqu'elle engendre un nombre infini de mots, ne peut produire *qu'un nombre fini de mots originaux*. En effet, puisqu'il n'y a qu'un nombre fini de productions, chacune contenant un nombre fini de symboles dans sa partie droite, chaque symbole terminal ne peut avoir qu'un nombre fini de

lignées différentes. Les symboles étant en nombre fini, il existe donc une longueur maximale pour un mot original et donc un nombre fini de mots originaux.

À quoi ressemblent alors les mots non-originaux ? Soit s un tel mot, il contient nécessairement un symbole non-original a, dont la lignée contient donc deux fois le même ancêtre A. La dérivation complète de s pourra donc s'écrire:

$$S \underset{G}{\overset{\star}{\Rightarrow}} uAy \underset{G}{\overset{\star}{\Rightarrow}} uvAxy \underset{G}{\overset{\star}{\Rightarrow}} uvwxy$$

où u, v, w, x, y sont des séquences de terminaux, la séquence w contenant le symbole a. Il est alors facile de déduire de nouveaux mots engendrés par la grammaire, en remplaçant w (qui est une dérivation possible de A) par vwx qui est une autre dérivation possible de A. Ce processus peut même être itéré, permettant de construire un nombre infini de nouveaux mots, tous non-originaux, et qui sont de la forme: $uv^n wx^n z$. Ces considérations nous amènent à un théorème caractérisant de manière précise cette limitation des langages algébriques.

Theorem 6.8 (Lemme de pompage pour les CFL)**.** *Si L est un langage CF, alors il existe un entier k tel que tout mot de L de longueur supérieure à k se décompose en cinq facteurs u, v, w, x, y, avec $vx \neq \varepsilon$, $|vwx| < k$ et tels que pour tout n, $uv^n wx^n y$ est également dans L.*

Proof. Une partie de la démonstration de ce résultat découle des observations précédentes. En effet, si L est vide ou fini, on peut prendre pour k un majorant de la longueur d'un mot de L. Comme aucun mot de L n'est plus long que k, il est vrai que tout mot de L plus long que k satisfait le theorem 6.8.

Supposons que L est effectivement infini, alors il existe nécessairement un mot z dans $L(G)$ plus long que le plus long mot original. Ce mot étant non-original, la décomposition précédente en cinq facteurs, dont deux sont simultanément itérables, s'applique immédiatement. La première condition supplémentaire, $vx \neq \varepsilon$, dérive de la possibilité de choisir G telle qu'aucun terminal autre que S ne dérive ε (voir chapter 9): en considérant une dérivation minimale, on s'assure ainsi qu'au moins un terminal est produit durant la dérivation $A \overset{\star}{\Rightarrow} vAx$. La seconde condition, $|vwx| < k$, dérive de la possibilité de choisir pour vwx un facteur original et donc de taille strictement inférieure à k. □

Ce résultat est utile pour prouver qu'un langage n'est pas hors-contexte. Montrons, à titre d'illustration, que $\{a^n b^n c^n, n \geq 1\}$ n'est pas hors-contexte. Supposons qu'il le soit et considérons un mot z suffisamment long de ce langage. Décomposons z en $uvwxy$, et notons $z_n = uv^n wx^n y$. Les mots v et x ne peuvent chacun contenir qu'un seul des trois symboles de l'alphabet, sans quoi leur répétition aboutirait à des mots ne respectant pas le séquencement imposé: tous les a avant tous les b avant tous les c. Pourtant, en faisant croître n, on augmente simultanément, dans z_n, le nombre de a, de b et de c dans des proportions identiques: ceci n'est pas possible puisque seuls deux des trois symboles sont concernés par l'exponentiation de v et x. Cette contradiction prouve que ce langage n'est pas hors-contexte.

6.3.2 Opérations sur les langages hors-contexte

Dans cette section, nous étudions les propriétés de clôture pour les langages hors-contexte, d'une manière similaire à celle conduite pour les reconnaissables à la section 4.2.1.

Une première série de résultats est établie par le théorème suivant:

Theorem 6.9 (Clôture)**.** *Les langages hors-contexte sont clos pour les opérations rationnelles.*

Proof. Pour les trois opérations, une construction simple permet d'établir ce résultat. Si, en effet, G_1 et G_2 sont définies par: $G_1 = (N_1, \Sigma_1, P_1, S_1)$ et $G_2 = (N_2, \Sigma_2, P_2, S_2)$, on vérifie simplement que:

— $G = (N_1 \cup N_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}, S)$ engendre exactement $L_1 \cup L_2$.

— $G = (N_1 \cup N_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S)$ engendre exactement $L_1 L_2$.

— $G = (N_1 \cup \{S\}, \Sigma_1, P_1 \cup \{S \to SS_1, S \to \varepsilon\}, S)$ engendre exactement ${L_1}^\star$.

□

En revanche, contrairement aux langages rationnels/reconnaissables, les langages algébriques ne sont pas clos pour l'intersection. Soient en effet $L_1 = \{a^n b^n c^m\}$ et $L_2 = \{a^m b^n c^n\}$: ce sont clairement deux langages hors-contexte dont nous avons montré plus haut que l'intersection, $L = L_1 \cap L_2 = \{a^n b^n c^n\}$, n'est pas un langage hors contexte. Un corollaire (déduit par application directe de la loi de De Morgan [4]) est que les langages hors-contexte ne sont pas clos par complémentation.

Pour conclure cette section, ajoutons un résultat que nous ne démontrons pas ici :

Theorem 6.10. *L'intersection d'un langage régulier et d'un langage hors-contexte est un langage hors-contexte.*

6.3.3 Problèmes décidables et indécidables

Dans cette section, nous présentons sommairement les principaux résultats de décidabilité concernant les grammaires et langages hors-contexte. Cette panoplie de nouveaux résultats vient enrichir le seul dont nous disposons pour l'instant, à savoir que les langages hors-contexte sont récursifs et qu'en conséquence, il existe des algorithmes permettant de décider si un mot u de $\Sigma^\star$ est, ou non, engendré par une grammaire G. Nous aurons l'occasion de revenir longuement sur ces algorithmes, notamment au chapter 7.

Nous commençons par deux résultats positifs, qui s'énonce comme:

Theorem 6.11. *Il existe un algorithme permettant de déterminer si le langage engendré par une grammaire hors-contexte est vide.*

Proof. La preuve exploite en fait un argument analogue à celui utilisé dans notre démonstration du lemme de pompage (theorem 6.8): on considère un arbre de dérivation hypothétique quelconque de la grammaire, engendrant w. Supposons qu'un chemin contienne plusieurs fois le même non-terminal A (aux nœuds n_1 et n_2, le premier dominant le second). n_1 domine le facteur w_1, n_2 domine w_2; on peut remplacer dans w la partie correspondant à w_1 par celle correspondant à w_2. Donc, s'il existe un mot dans le langage engendré, il en existera également un qui soit tel que le même non-terminal n'apparaît jamais deux fois dans un

4. Rappelons: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.

chemin. Dans la mesure où les non-terminaux sont en nombre fini, il existe un nombre fini de tels arbres. Il suffit alors de les énumérer et de vérifier s'il en existe un dont la frontière ne contient que des terminaux: si la réponse est oui, alors $L(G)$ est non-vide, sinon, $L(G)$ est vide. □

Theorem 6.12. *Il existe un algorithme permettant de déterminer si le langage engendré par une grammaire hors-contexte est infini.*

Proof. L'idée de la démonstration repose sur l'observation suivante: après élimination des productions inutiles, des productions epsilon et des cycles (voir la section 9.1), il est possible d'énumérer tous les arbres de profondeur bornée: si on ne trouve jamais deux fois le même non terminal sur une branche, le langage est fini; sinon il est infini. □

Ces résultats sont les seuls résultats positifs pour les grammaires CF, puisqu'il est possible de prouver les résultats négatifs suivants:

- il n'existe pas d'algorithme pour décider si deux grammaires sont équivalentes (rappelons: la preuve du résultat (positif) obtenu pour les langages rationnels utilisait la clôture par intersection de ces langages);
- il n'existe pas d'algorithme pour décider si le langage engendré par une grammaire CF est inclus dans le langage engendré par une autre grammaire CF;
- il n'existe pas d'algorithme pour décider si une grammaire CF engendre en fait un langage régulier;
- il n'existe pas d'algorithme pour décider si une grammaire est ambiguë.

Munis de ces résultats, nous pouvons maintenant aborder le problème de la reconnaissance des mots engendrés par une grammaire CF, qui fait l'objet des chapitres suivants.

Chapter 7

Introduction au parsage de grammaires hors-contexte

Dans ce chapitre, nous présentons les principales difficultés algorithmiques que pose l'analyse de grammaires hors-contexte. Il existe, en fait, trois tâches distinctes que l'on souhaiterait effectuer à l'aide d'une grammaire: la *reconnaissance*, qui correspond au calcul de l'appartenance d'un mot à un langage; l'analyse (ou le *parsage*), qui correspond au calcul de tous les arbres d'analyse possibles pour un énoncé; la *génération*, qui correspond à la production de tous les mots du langage décrit par une grammaire. Dans la suite de ce chapitre, nous ne nous intéresserons qu'aux deux premières de ces tâches.

Les langages hors-contexte étant une sous classe des langages récursifs, nous savons qu'il existe des algorithmes permettant d'accomplir la tâche de reconnaissance. En fait, il en existe de multiples, qui, essentiellement se ramènent tous à deux grands types: les analyseurs *ascendants* et les analyseurs *descendants*. Comme expliqué dans la suite, toute la difficulté pour les analyseurs consiste à affronter le non-déterminisme inhérent aux langages hors-contexte de manière à:

— éviter de perdre son temps dans des impasses;

— éviter de refaire deux fois les mêmes calculs.

Ce chapitre est organisé comme suit: dans la section 7.1, nous présentons l'espace de recherche du parsage; nous présentons ensuite les stratégies ascendantes dans la section 7.2 et descendantes en section 7.3, ainsi que les problèmes que pose la mise en œuvre de telles stratégies. Cette étude des analyseurs se poursuit au chapter 8, où nous présentons des analyseurs déterministes, utilisables pour certains types de grammaires; ainsi qu'au chapter 9, où nous étudions des techniques de normalisation des grammaires, visant à simplifier l'analyse, ou à se prémunir contre des configurations indésirables.

Une référence extrêmement riche et complète pour aborder les questions de parsage est Grune et Jacob (1990).

7.1 Graphe de recherche

Un point de vue général sur la question de la reconnaissance est donné par la considération suivante. Comme toute relation binaire, la relation binaire $\underset{G}{\Rightarrow}$ sur les séquences de $(N \cup \Sigma)^\star$ se représente par un graphe dont les sommets sont les séquences de $(N \cup \Sigma)^\star$ et dans lequel un arc de α vers β indique que $\alpha \underset{G}{\Rightarrow} \beta$. On l'appelle le *graphe de la grammaire* G. Ce graphe est bien sûr infini dès lors que $L(G)$ est infini; il est en revanche *localement fini*, signifiant que tout sommet a un nombre fini de voisins.

La question à laquelle un algorithme de reconnaissance doit répondre est alors la suivante: existe-t-il dans ce graphe un chemin du nœud S vers le nœud u ? Un premier indice: s'il en existe un, il en existe nécessairement plusieurs, correspondant à plusieurs dérivations différentes réductibles à une même dérivation gauche. Il n'est donc pas nécessaire d'explorer tout le graphe.

Pour répondre plus complètement à cette question, il existe deux grandes manières de procéder: construire et explorer de proche en proche les voisins de S en espérant rencontrer u: ce sont les approches dites *descendantes*; ou bien inversement construire et explorer les voisins [1] de u, en essayant de "remonter" vers S: on appelle ces approches *ascendantes*. Dans les deux cas, si l'on prend soin, au cours de l'exploration, de se rappeler des différentes étapes du chemin, on sera alors à même de reconstruire un arbre de dérivation de u.

Ces deux groupes d'approches, ascendantes et descendantes, sont considérés dans les sections qui suivent, dans lesquelles on essayera également de suggérer que ce problème admet une implantation algorithmique polynomiale: dans tous les cas, la reconnaissance de u demande un nombre d'étapes borné par $k|u|^p$ pour un certain p fixé qui est indépendant de l'entrée.

Le second problème intéressant est celui de l'analyse, c'est-à-dire de la construction de tous les arbres de dérivation de u dans G. Pour le résoudre, il importe non seulement de déterminer non pas un chemin, mais tous les chemins [2] et les arbres de dérivation correspondants. Nous le verrons, cette recherche exhaustive peut demander, dans les cas où la grammaire est ambiguë, un temps de traitement exponentiel par rapport à la longueur de u. En effet, dans une grammaire ambiguë, il peut exister un nombre exponentiel d'arbres de dérivation, dont l'énumération (explicite) demandera nécessairement un temps de traitement exponentiel.

7.2 Reconnaissance ascendante

Supposons qu'étant donnée la grammar 6.1 des repas dominicaux, nous soyons confrontés à l'énoncé: *le fils mange sa soupe*. Comment faire alors pour:

— vérifier que ce "mot" appartient au langage engendré par la grammaire ?

— lui assigner, dans le cas où la réponse est positive, une (ou plusieurs) structure(s) arborée(s) ?

1. En renversant l'orientation des arcs.
2. En fait, seulement ceux qui correspondent à une dérivation gauche.

Une première idée d'exploration nous est donnée par l'algorithme suivant, qui, partant des terminaux du mot d'entrée, va chercher à "inverser" les règles de production pour parvenir à récrire le symbole initial de la grammaire: chaque étape de l'algorithme vise à réduire la taille du proto-mot courant, en substituant la partie droite d'une production par sa partie gauche. Dans l'algorithme présenté ci-dessous, cette réduction se fait par la gauche: on cherche à récrire le proto-mot courant en commençant par les symboles qui se trouvent le plus à gauche: à chaque étape, il s'agit donc d'identifier, en commençant par la gauche, une partie droite de règle. Lorsqu'une telle partie droite est trouvée, on la remplace par la partie gauche correspondante et on continue. Si on ne peut trouver une telle partie droite, il faut remettre en cause une règle précédemment appliquée (par retour en arrière) et reprendre la recherche en explorant un autre chemin. Cette recherche s'arrête lorsque le proto-mot courante ne contient plus, comme unique symbole, que l'axiome S.

Une trace de l'exécution de cet algorithme est donnée ci-dessous, après que le proto-mot courant α a été initialisé avec *le fils mange sa soupe*:

1. on remplace *le* par la partie gauche de la règle $le \rightarrow DET$. α = *DET fils mange sa soupe*.
2. on essaie de remplacer *DET* en le trouvant comme partie droite. Échec. Idem pour *DET fils*, *DET fils mange*...
3. on récrit *fils*: N. α = *DET N mange sa soupe*.
4. on essaie de remplacer *DET* en le trouvant comme partie droite. Échec.
5. on récrit *DET N*: GN. α = *GN mange sa soupe*.
6. on essaie de remplacer *GN* en le trouvant comme partie droite. Échec. Idem pour *GN mange*, *GN mange sa*...
7. on récrit *mange*: V. α = *GN V sa soupe*.
8. on essaie de remplacer *GN* en le trouvant comme partie droite. Échec. Idem pour *GN V*, *GN V sa*...
9. on récrit V: GV. α = *GN GV sa soupe*.
10. on essaie de remplacer *GN* en le trouvant comme partie droite. Échec.
11. on récrit *GN GV*: S. α = *S sa soupe*.
12. on essaie de remplacer *S* en le trouvant comme partie droite. Échec. Idem pour *S sa*, *S sa soupe*.
13. on récrit *sa*: *DET*. Résultat: *S DET soupe*.
14. on essaie de remplacer *S* en le trouvant comme partie droite. Échec. Idem pour *S DET*, *S DET soupe*.
15. on essaie de remplacer *DET* en le trouvant comme partie droite. Échec. Idem pour *DET soupe*.
16. on remplace *soupe* par N. α = *S DET N*.
17. on essaie de remplacer *S* en le trouvant comme partie droite. Échec. Idem pour *S DET*, *S DET GN*.
18. on essaie de remplacer *DET* en le trouvant comme partie droite. Échec.
19. on remplace *DET N* par GN. α = *S GN*.
20. on essaie de remplacer *S* en le trouvant comme partie droite. Échec.

21. on essaie de remplacer *S GN* en le trouvant comme partie droite. Échec. On est bloqué: on fait un retour arrière jusqu'en 11. α = *GN GV sa soupe*.

... on va d'abord parcourir entièrement le choix après *GN* jusqu'à *GN GV GN*. Nouvelle impasse. Donc on revient en arrière en 8. Et on recommence...

... tôt ou tard, on devrait finalement parvenir à une bonne solution.

Cette stratégie est mise en œuvre par l'algorithm 7.1.

```
// La fonction principale est bulrp:  bottom-up left-right parsing.
// α contient le proto-mot courant.
Function bulrp(α): bool is
    if α = S then  return true ;
    for i := 1 to |α| do
        for j := i to |α| do
            if ∃A → α_i ... α_j then
                if bulrp(α_1 ... α_{i-1} A α_{j+1} ... α_n) = true then
                    return true
    return false ;
```

Algorithm 7.1– Parsage ascendant en profondeur d'abord

La procédure détaillée dans l'algorithm 7.1 correspond à la mise en œuvre d'une stratégie *ascendante* (en anglais *bottom-up*), signifiant que l'on cherche à construire l'arbre de dérivation depuis le bas (les feuilles) vers le haut (la racine de l'arbre), donc depuis les symboles de Σ vers S. La stratégie mise en œuvre par l'algorithm 7.1 consiste à explorer le graphe de la grammaire *en profondeur d'abord*. Chaque branche est explorée de manière successive; à chaque échec (lorsque toutes les parties droites possibles ont été successivement envisagées), les choix précédents sont remis en cause et des chemins alternatifs sont visités. Seules quelques branches de cet arbre mèneront finalement à une solution. Notez qu'il est possible d'améliorer un peu `bulrp` pour ne considérer que des dérivations droites. Comment faudrait-il modifier cette fonction pour opérer une telle optimisation ?

Comme en témoigne l'aspect un peu répétitif de la simulation précédente, cette stratégie conduit à répéter de nombreuses fois les mêmes tests, et à reconstruire en de multiples occasions les mêmes constituants (voir la fig. 7.2).

L'activité principale de ce type d'algorithme consiste à chercher, dans le proto-mot courant, une séquence pouvant correspondre à la partie droite d'une règle. L'alternative que cette stratégie d'analyse conduit à évaluer consiste à choisir entre remplacer une partie droite identifiée par le non-terminal correspondant (étape de *réduction*, en anglais *reduce*), ou bien différer la réduction et essayer d'étendre la partie droite en considérant des symboles supplémentaires (étape *d'extension*, on dit aussi *décalage*, en anglais *shift*). Ainsi les étapes de réduction 9 et 11 dans l'exemple précédent conduisent-elles à une impasse, amenant à les remplacer par des étapes d'extension, qui vont permettre dans notre cas de construire l'objet direct du verbe avant d'entreprendre les bonnes réductions. Comme on peut le voir dans cet exemple, l'efficacité de l'approche repose sur la capacité de l'analyseur à prendre les bonnes décisions (*shift* ou *reduce*) au bon moment. À cet effet, il est possible de pré-calculer un certain nombre de tables auxiliaires, qui permettront de mémoriser le fait que par exemple, *mange* attend un complément, et que donc il ne faut pas réduire V avant d'avoir trouvé ce complément. Cette intuition sera formalisée dans le chapitre suivant, à la section 8.2.

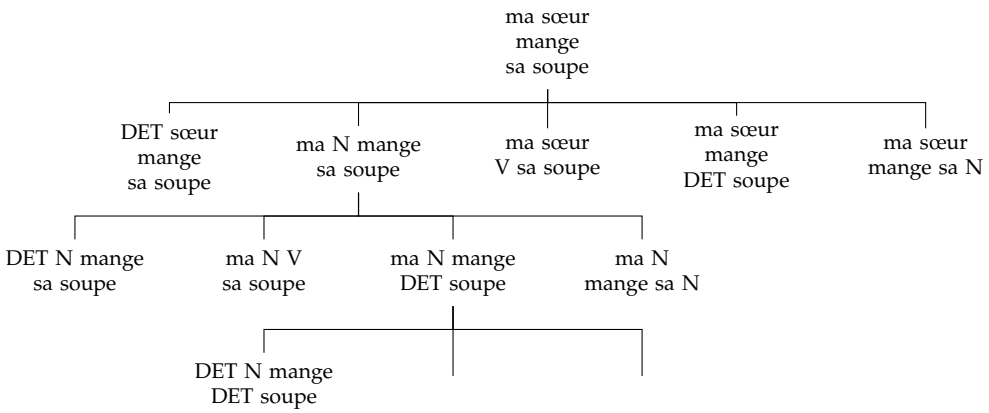


Figure 7.2 – Recherche ascendante

La stratégie naïve conduit à une analyse exponentielle en fonction de la longueur de l'entrée. Fort heureusement, il existe des techniques éprouvées, consistant à stocker les résultats intermédiaires dans des tableaux, qui permettent d'aboutir à une complexité polynomiale.

Notez, pour conclure, que dans notre exemple, la stratégie mise en œuvre termine, ce qui ne serait pas le cas si notre grammaire était moins "propre". L'analyse ascendante est prise en défaut par des règles du type $X \to X$, ou par des cycles de production de type $X \to Y, Y \to X$, qui conduisent à effectuer (indéfiniment !) des séries de réductions "stériles", c'est-à-dire des réductions qui laissent inchangée la longueur du proto-mot courant. Des problèmes sérieux se posent également avec les productions ε, qui sont susceptibles de s'appliquer à toutes les étapes de l'algorithme. La "grammaire des dimanches" n'en comporte pas, mais de telles règles ne sont pas rares dans les grammaires réelles.

Avant de montrer qu'il est, en théorie, possible de se prémunir contre de telles horreurs (voir la section 9.1), il est important de noter que cette stratégie s'accompagne de multiples variantes, consistant par exemple à effectuer l'exploration *en largeur d'abord*, ou de droite à gauche, ou simultanément dans les deux sens...

Donnons l'intuition de ce que donnerait une exploration en largeur d'abord: d'une manière générale, ce type de recherche correspond à la poursuite en parallèle d'un ensemble de chemins possibles. L'algorithme est initialisé avec un seul proto-mot, qui est l'énoncé à analyser. À chaque étape, un nouvel ensemble de proto-mots est considéré, auxquels sont appliquées toutes les réductions possibles, donnant lieu à un nouvel ensemble de proto-mots. Si cet ensemble contient S, on arrête; sinon la procédure est répétée. En guise d'application, écrivez sur le modèle de l'algorithme précédent une procédure mettant en application cette idée.

7.3 Reconnaissance descendante

Une seconde stratégie de recherche consiste à procéder de manière *descendante*, c'est-à-dire à partir de l'axiome de la grammaire pour tenter d'engendrer l'énoncé à analyser. Les

impasses de cette recherche sont détectées en confrontant des préfixes terminaux du proto-mot courant avec les terminaux de l'énoncé à analyser. Comme précédemment, dans une recherche en profondeur d'abord, les situations d'échecs conduisent à remettre en cause les choix précédemment effectués; une recherche en largeur d'abord conduit à développer simultanément plusieurs proto-mots. La recherche s'arrête lorsque le proto-mot dérivé depuis S est identique à la séquence qu'il fallait analyser.

Simulons, par exemple, le fonctionnement d'un analyseur en profondeur d'abord:

1. S
2. *GN GV*
3. *DET N GV*
4. *la N GV*. Échec : l'entrée commence par *le*, pas par *la*. On revient à *DET N GV*
5. *le N GV*.
6. *le fille GV*. Nouvel échec α = *le N GV*

...

7. *le fils GV*.

... *le fils V*.

... *le fils boude*. Échec α = *le fils V*

... *le fils s'ennuie*. Échec α = *le fils V*

... *le fils mange*. Il reste des terminaux non-analysés dans l'entrée, mais le proto-mot ne contient plus de non-terminaux. Échec et retour en 7.

...

Algorithmiquement, cette stratégie d'analyse correspond à une boucle dans laquelle l'analyseur examine le non-terminal le plus à gauche du proto-mot courant et essaie de le dériver tout en restant compatible avec l'énoncé d'entrée. Cet analyseur construit donc des dérivations gauches. Son activité alterne des étapes de *prédiction* (en anglais *prediction*) d'un symbole terminal ou non-terminal et des étapes *d'appariement* (en anglais *matching*) des terminaux prédits avec les mots de l'entrée. Pour cette raison, le parsage descendant est souvent qualifié de *prédictif*. Le pseudo-code d'un tel analyseur est donné dans l'algorithm 7.3.

```
// La fonction principale est tdlrp: top-down left-right parsing.
// α est le proto-mot courant, u l'entrée à reconnaître.
Function tdlrp(α, u): bool is
    if α = u then return true ;
    α = u_1 ... u_k A γ ;
    while ∃A → u_{k+1}...u_{k+l} δ avec δ = ε ou δ = A ... do
        if tdlrp(u_1 ... u_{k+l} δ γ) = true then return true ;
    return false
```

Algorithm 7.3– Parsage descendant en profondeur d'abord

Une implantation classique de cette stratégie représente α sous la forme d'une *pile* de laquelle est exclu le préfixe terminal $u_1 \dots u_k$: à chaque étape il s'agit de dépiler le non-terminal A en tête de la pile, et d'empiler à la place un suffixe de la partie droite correspondante (β),

après avoir vérifié que l'(éventuel) préfixe terminal de β ($u_{uk+1} \dots u_k + l$) était effectivement compatible avec la partie non-encore analysée de u.

Pour que cette stratégie soit efficace, il est possible de pré-calculer dans des tables l'ensemble des terminaux qui peuvent débuter l'expansion d'un non-terminal. En effet, considérons de nouveau la grammar 6.1: cette grammaire possède également les productions suivantes, $GN \rightarrow NP$, $NP \rightarrow Louis$, $NP \rightarrow Paul$, correspondant à des noms propres. À l'étape 2 du parcours précédent, l'algorithme pourrait être (inutilement) conduit à prédire un constituant de catégorie NP, alors même qu'aucun nom propre ne peut débuter par *le*, qui est le premier terminal de l'énoncé à analyser. Cette intuition est formalisée dans les analyseurs LL, qui sont présentés dans la section 8.1.

Dans l'optique d'une stratégie de reconnaissance descendante, notre grammaire possède un vice de forme manifeste, à savoir la production $GN \rightarrow GN \quad GNP$, qui est récursive à gauche: lorsque cette production est considérée, elle insère en tête du proto-mot courant un nouveau symbole GN, qui à son tour peut être remplacé par un nouveau GN, conduisant à un accroissement indéfini de α. Dans le cas présent, la récursivité gauche est relativement simple à éliminer. Il existe des configurations plus complexes, dans lesquelles cette récursivité gauche résulte non pas d'une règle unique mais d'une séquence de règles, comme dans: $S \rightarrow A\alpha$, $A \rightarrow S\beta$. On trouvera au chapter 9 une présentation des algorithmes permettant d'éliminer ce type de récursion.

Notons, pour finir, que nous avons esquissé ci-dessus les principales étapes d'une stratégie descendante en profondeur d'abord. Il est tout aussi possible d'envisager une stratégie en largeur d'abord, consistant à conduire de front l'examen de plusieurs proto-mots. L'écriture d'un algorithme mettant en œuvre cette stratégie est laissée en exercice.

7.4 Conclusion provisoire

Nous avons, dans ce chapitre, défini les premières notions nécessaires à l'étude des algorithmes d'analyse pour les grammaires algébriques. Les points de vue sur ces algorithmes diffèrent très sensiblement suivant les domaines d'application:

— dans le cas des langages informatiques, les mots (des programmes, des documents structurés) sont longs, voire très longs; l'ambiguïté est à proscrire, pouvant conduire à des conflits d'interprétation. Les algorithmes de vérification syntaxique doivent donc avoir une faible complexité (idéalement une complexité linéaire en fonction de la taille de l'entrée); il suffit par ailleurs en général de produire une analyse et une seule. Les algorithmes qui répondent à ce cahier des charges sont présentés au chapter 8.

— dans le cas des langues naturelles, l'ambiguïté est une des données du problème, qu'il faut savoir contrôler. Les mots étant courts, une complexité supérieure (polynomiale, de faible degré) pour l'analyse est supportable. On peut montrer qu'à l'aide de techniques de programmation dynamique, consistant à sauvegarder dans des tables les fragments d'analyse réalisés, il est possible de parvenir à une complexité en $O(n^3)$ pour construire d'un seul coup tous les arbres d'analyse. Leur énumération exhaustive conserve une complexité exponentielle.

Quel que soit le contexte, les algorithmes de parsage bénéficient toujours d'une étape de

prétraitement et de normalisation de la grammaire, visant en particulier à éviter les configurations problématiques (cycles et productions ε pour les analyseurs ascendants; récursions gauches pour les analyseurs descendants). Ces prétraitements sont présentés dans le chapter 9.

Chapter 8

Introduction aux analyseurs déterministes

Dans ce chapitre, nous présentons deux stratégies d'analyse pour les grammaires hors-contexte, qui toutes les deux visent à produire des analyseurs déterministes. Comme présenté au chapter 7, le parsage peut être vu comme l'exploration d'un graphe de recherche. L'exploration est rendue coûteuse par l'existence de ramifications dans le graphe, correspondant à des chemins alternatifs: ceci se produit, pour les analyseurs descendants, lorsque le terminal le plus à gauche du proto-mot courant se récrit de plusieurs manières différentes (voir la section 7.1). De telles alternatives nécessitent de mettre en œuvre des stratégies de retour arrière (recherche en profondeur d'abord) ou de pouvoir explorer plusieurs chemins en parallèle (recherche en largeur d'abord).

Les stratégies d'analyse présentées dans ce chapitre visent à éviter au maximum les circonstances dans lesquelles l'analyseur explore inutilement une branche de l'arbre de recherche (en profondeur d'abord) ou dans lesquelles il conserve inutilement une analyse dans la liste des analyses alternatives (en largeur d'abord). À cet effet, ces stratégies mettent en œuvre des techniques de pré-traitement de la grammaire, cherchant à identifier par avance les productions qui conduiront à des chemins alternatifs; ainsi que des contrôles permettant de choisir sans hésiter la bonne ramification. Lorsque de tels contrôles existent, il devient possible de construire des analyseurs *déterministes*, c'est-à-dire des analyseurs capables de toujours faire le bon choix. Ces analyseurs sont algorithmiquement efficaces, en ce sens qu'ils conduisent à une complexité d'analyse *linéaire* par rapport à la longueur de l'entrée.

La section 8.1 présente la mise en application de ce programme pour les stratégies d'analyse descendantes, conduisant à la famille d'analyseurs prédictifs LL. La section 8.2 s'intéresse à la construction d'analyseurs ascendants, connus sous le nom d'analyseurs LR, et présente les analyseurs les plus simples de cette famille.

8.1 Analyseurs LL

Nous commençons par étudier les grammaires qui se prêtent le mieux à des analyses descendantes et dérivons un premier algorithme d'analyse pour les grammaires SLL(1). Nous généralisons ensuite cette approche en introduisant les grammaires LL(1), ainsi que les algorithmes d'analyse correspondants.

8.1.1 Une intuition simple

Comme expliqué à la section 7.3, les analyseurs descendants essayent de construire de proche en proche une dérivation gauche du mot u à analyser: partant de S, il s'agit d'aboutir, par des récritures successives du (ou des) proto-mots(s) courant(s), à u. À chaque étape de l'algorithme, le non-terminal A le plus à gauche est récrit par $A \to \alpha$, conduisant à l'exploration d'une nouvelle branche dans l'arbre de recherche. Deux cas de figure sont alors possibles:

(i) soit α débute par au moins un terminal: dans ce cas on peut *immédiatement* vérifier si le proto-mot courant est compatible avec le mot à analyser et, le cas échéant, revenir sur le choix de la production $A \to \alpha$. C'est ce que nous avons appelé la phase d'*appariement*.

(ii) soit α débute par un non-terminal (ou est vide): dans ce cas, il faudra continuer de développer le proto-mot courant pour valider la production choisie et donc *différer* la validation de ce choix.

Une première manière de simplifier la tâche des analyseurs consiste à éviter d'utiliser dans la grammaire les productions de type item ii: ainsi l'analyseur pourra toujours immédiatement vérifier le bien-fondé des choix effectués, lui évitant de partir dans l'exploration d'un cul-de-sac.

Ceci n'élimine toutefois pas toute source de non-déterminisme: s'il existe un non-terminal A apparaissant dans deux productions de type item i $A \to a\alpha$ et $A \to a\beta$, alors il faudra quand même considérer (en série ou en parallèle) plusieurs chemins alternatifs.

Considérons, à titre d'exemple, le fragment présenté par la grammar 8.1. Elle présente la particularité de ne contenir que des productions de type item i; de plus, les coins gauches des productions associées à ce terminal sont toutes différentes.

$$
\begin{aligned}
S &\to \mathit{if}\ (\ B\ )\ \mathit{then}\ \{\ I\ \} \\
S &\to \mathit{while}\ (\ B\ )\ \{\ I\ \} \\
S &\to \mathit{do}\ \{\ I\ \}\ \mathit{until}\ (\ B\ ) \\
B &\to \mathit{false} \mid \mathit{true} \\
I &\to \ldots
\end{aligned}
$$

Grammar 8.1 – Fragments d'un langage de commande

Il est clair qu'un analyseur très simple peut être envisagé, au moins pour explorer sans hésitation (on dit aussi: *déterministiquement*) les dérivations de S: soit le premier symbole à apparier est le mot-clé *if*, et il faut appliquer la première production; soit c'est *while* et il faut appliquer la seconde; soit c'est *do* et il faut appliquer la troisième. Tous les autres cas de figure correspondent à des situations d'erreur.

Si l'on suppose que tous les non-terminaux de la grammaire possèdent la même propriété que S dans la grammar 8.1, alors c'est l'intégralité de l'analyse qui pourra être conduite de manière déterministe par l'algorithme suivant: on initialise le proto-mot courant avec S et à chaque étape, on consulte les productions du non-terminal A le plus à gauche, en examinant s'il en existe une dont le coin gauche est le premier symbole u_i non encore apparié. Les conditions précédentes nous assurant qu'il existe au plus une telle production, deux configurations sont possibles:

— il existe une production $A \to u_i\alpha$, et on l'utilise; le prochain symbole à apparier devient u_{i+1};

— il n'en existe pas: le mot à analyser n'est pas engendré par la grammaire.

Cet algorithme se termine lorsque l'on a éliminé tous les non-terminaux du proto-mot: on vérifie alors que les terminaux non-encore appariés dans l'entrée correspondent bien au suffixe du mot engendré: si c'est le cas, l'analyse a réussi. Illustrons cet algorithme en utilisant la grammar 8.2; le table 8.3 décrit pas-à-pas les étapes de l'analyse de *aacddcbb* par cette grammaire.

$$
\begin{aligned}
S &\to aSb \\
S &\to cC \\
C &\to dC \\
C &\to c
\end{aligned}
$$

Grammar 8.2 – Une grammaire LL(1) simple

itération	apparié	à apparier	prédit	règle
0	ε	*aacddcbb*	S	$S \to aSb$
1	*a*	*acddcbb*	Sb	$S \to aSb$
2	*aa*	*cddcbb*	Sbb	$S \to cC$
3	*aac*	*ddcbb*	Cbb	$C \to dC$
4	*aacd*	*dcbb*	Cbb	$C \to dC$
5	*aacdd*	*cbb*	Cbb	$C \to c$
6	*aacddc*	*bb*	*bb*	

À l'itération 3 de l'algorithme, le proto-mot courant commande la récriture de C: le symbole à apparier étant un d, le seul choix possible est d'appliquer la production $C \to dC$; ceci a pour effet de produire un nouveau proto-mot et de décaler vers la droite le symbole à apparier.

Table 8.3 – Étapes de l'analyse de aacddcbb

L'application de l'algorithme d'analyse précédent demande des consultations répétitives de la grammaire pour trouver quelle règle appliquer. Il est possible de construire à l'avance une table enregistrant les productions possibles. Cette table, dite *table d'analyse prédictive*, contient pour chaque non-terminal A et pour chaque symbole d'entrée a l'indice de la production à utiliser lorsque A est le plus à gauche dans le proto-mot courant, alors que l'on veut apparier a. Cette table est en fait exactement analogue à la table de transition d'un automate déterministe: pour chaque non-terminal (état) et chaque symbole d'entrée, elle indique la production (la transition) à utiliser. Pour la grammar 8.2, cette matrice est reproduite dans le table 8.4.

	a	b	c	d
S	$a\ S\ b$		$c\ C$	
C			c	$d\ C$

Si S est prédit et a le premier terminal non apparié, alors récrire S par aSb. Les cases vides sont des situations d'échec.

Table 8.4 – Table d'analyse prédictive

Cette analogie[1] suggère que la stratégie d'analyse appliquée ci-dessus a une complexité linéaire: un mot de longueur k ne demandera jamais un nombre d'étapes supérieur à k: le déterminisme rend l'analyse algorithmiquement efficace, ce qui justifie les efforts pour rechercher des algorithmes déterministes.

Les grammaires présentant les deux propriétés précédentes sont appelées des grammaires LL(1) *simples* et sont définies formellement par:

Definition 8.1 (Grammaire SLL(1))**.** *Une grammaire* $G = (N, \Sigma, S, P)$ *est une grammaire* SLL(1) *si et seulement si:*

(i) $\forall (A \to \alpha) \in P, \exists a \in \Sigma, \alpha = a\beta$

(ii) $\forall A \in N, (A \to a_1\alpha_1) \in P$ *et* $(A \to a_2\alpha_2) \in P$ *et* $a_1\alpha_1 \neq a_2\alpha_2 \Rightarrow a_1 \neq a_2$

Pourquoi cette terminologie ? Parce qu'elles permettent directement de mettre en œuvre des analyseurs construisant de manière déterministe de gauche à droite (*Left-to-right*) des dérivations gauches (*Leftmost derivation*) avec un regard avant (en anglais *Lookahead*) de 1 seul symbole. Dans la mesure où la forme de la grammaire rend immédiate cette construction, ces grammaires sont additionnellement qualifiées de *simples*, d'où la terminologie *Simple LL*.

La question qui se pose alors est la suivante: tout langage hors-contexte est-il susceptible d'être analysé par une telle technique ? En d'autres termes, est-il possible de construire une grammaire SLL(1) pour tout langage algébrique ? La réponse est malheureusement non: il existe des langages intrinsèquement ambigus (voir la section 6.2.3) pour lesquels il est impossible de construire un analyseur déterministe. Il est, en revanche, possible de transformer toute grammaire en une grammaire satisfaisant la propriété (i) par une suite de transformations aboutissant à la forme normale de Greibach (voir la section 9.2.2). Cette transformation conduisant toutefois à des grammaires (et donc à des dérivations) très éloignées de la grammaire initiale, il est tentant de chercher à généraliser les stratégies développées dans cette section, de manière à pouvoir les appliquer à des grammaires moins contraintes. Cette démarche est poursuivie dans la suite de ce chapitre avec l'introduction des grammaires LL.

8.1.2 Grammaires LL(1)

La clé du succès des analyseurs pour les grammaires SLL(1) est la possibilité de contrôler sans attendre la validité d'une prédiction, rendue possible par la propriété que tout non-terminal

1. C'est un peu plus qu'une analogie: une grammaire régulière, par définition, vérifie la propriété (i) ci-dessus; si elle vérifie de plus la propriété (ii), alors vous vérifierez que l'algorithme de transformation de la grammaire en automate décrit à la section 5.2.4 aboutit en fait à un automate *déterministe*.

récrit comme premier symbole un symbole terminal.

Considérons alors une grammaire qui n'a pas cette propriété, telle que celle de la grammar 8.5, qui engendre des formules arithmétiques.

$$\begin{aligned} S &\rightarrow S + F \mid F \\ F &\rightarrow F * T \mid T \\ T &\rightarrow (S) \mid D \\ D &\rightarrow 0 \mid \cdots \mid 9 \mid 0D \mid \cdots \mid 9D \end{aligned}$$

Grammar 8.5 – Une grammaire pour les expressions arithmétiques

Le non-terminal F de la grammar 8.5, par exemple, se récrit toujours en un non-terminal. En examinant les dérivations de ce terminal, on constate toutefois que son élimination d'une dérivation gauche conduit toujours à avoir comme nouveau non-terminal le plus à gauche un T, par des dérivations de la forme:

$$F \underset{A}{\Rightarrow} F * T \overset{\star}{\underset{A}{\Rightarrow}} F * T * T * \cdots * T \underset{A}{\Rightarrow} T * T * \ldots * T$$

Examinons alors les productions de T: l'une commence par récrire le terminal '(': cela signifie donc que T, et donc F aussi, dérive des proto-mots de type $(\alpha$. T peut également se récrire D; ce non-terminal respecte la contrainte précédente, nous garantissant qu'il récrit toujours en premier un chiffre entre 0 et 9. On en déduit que les dérivations de T débutent soit par '(', soit par un chiffre; il en va alors de même pour les dérivations de F. À quoi nous sert cette information ? D'une part, à détecter une erreur dès que F apparaît en tête d'un proto-mot alors qu'un autre terminal (par exemple + ou $*$) doit être apparié. Mais on peut faire bien mieux encore: supposons en effet que l'on cherche à dériver T. Deux productions sont disponibles: le calcul précédent nous fournit un moyen infaillible de choisir entre elles: si le symbole à apparier est '(', il faut choisir $T \rightarrow (S)$; sinon si c'est un chiffre, il faut choisir $T \rightarrow D$.

En d'autres termes, le calcul des terminaux qui sont susceptibles d'initier une dérivation gauche permet de sélectionner au plus tôt entre productions ayant une même partie gauche, ainsi que d'anticiper sur l'application erronée de productions. En fait, ces symboles jouent le même rôle que les terminaux apparaissant en coin gauche des productions de grammaire SLL(1) et leur calcul est donc essentiel pour anticiper sur les bonnes prédictions.

L'exemple précédent suggère une approche récursive pour effectuer ce calcul, consistant à examiner récursivement les coins gauches des productions de la grammaire jusqu'à tomber sur un coin gauche terminal. Formalisons maintenant cette intuition.

8.1.3 NULL, FIRST et FOLLOW

FIRST Pour commencer, définissons l'ensemble FIRST(A) des symboles terminaux pouvant apparaître en tête d'une dérivation gauche de A, soit:

Definition 8.2 (FIRST). *Soit $G = (N, \Sigma, S, P)$ une grammaire CF et A un élément de N. On appelle* FIRST(A) *le sous-ensemble de Σ défini par:*

$$\text{FIRST}(A) = \{a \in \Sigma, \exists \alpha \in (N \cup \Sigma), A \overset{\star}{\underset{G}{\Rightarrow}} a\alpha\}$$

Ainsi, dans la grammar 8.5, on a:

$$\text{FIRST}(F) = \text{FIRST}(T) = \{(, 0, 1, \ldots, 9\}$$

Comment construire automatiquement cet ensemble ? Une approche naïve consisterait à implanter une formule récursive du type:

$$\text{FIRST}(A) = \bigcup_{X, A \to X\alpha \in P} \text{FIRST}(X)$$

Cette approche se heurte toutefois à deux difficultés:

— les productions récursives à gauche, qui sont du type du type $A \to A\alpha$; ces productions sont hautement nocives pour les analyseurs descendants (voir la section 7.3) et il faudra dans tous les cas s'en débarrasser. Des techniques idoines pour ce faire sont présentées à la section 9.2.2; dans la suite de l'exposé on considérera qu'il n'y a plus de production (ni de chaîne de productions) récursive à gauche;

— les productions $A \to X\alpha$ pour lesquelles le non-terminal X est tel que $X \overset{\star}{\underset{G}{\Rightarrow}} \varepsilon$: dans ce cas, il faudra, pour calculer correctement FIRST(A), tenir compte du fait que le coin gauche X des A-productions peut dériver le mot vide.
Ces non-terminaux soulèvent toutefois un problème nouveau: supposons en effet que P contienne une règle de type $A \to \varepsilon$, et considérons l'état d'un analyseur descendant tentant de faire des prédictions à partir d'un proto-mot de la forme $uA\alpha$. Le non-terminal A pouvant ne dériver aucun symbole, il paraît difficile de contrôler les applications erronées de la production $A \to \varepsilon$ en regardant simplement FIRST(A) et le symbole à apparier. Notons qu'il en irait de même si l'on avait $A \overset{\star}{\underset{G}{\Rightarrow}} \varepsilon$. Comment alors anticiper sur les dérivations erronées de tels non-terminaux et préserver le déterminisme de la recherche ?

Pour résoudre les problèmes causés par l'existence de non-terminaux engendrant le mot vide, introduisons deux nouveaux ensembles: NULL et FOLLOW.

NULL NULL est l'ensemble des non-terminaux dérivant le mot ε. Il est défini par:

Definition 8.3 (NULL). *Soit $G = (N, \Sigma, S, P)$ une grammaire CF. On appelle NULL le sous-ensemble de N défini par:*

$$\text{NULL} = \{A \in N, A \overset{\star}{\underset{G}{\Rightarrow}} \varepsilon\}$$

Cet ensemble se déduit très simplement de la grammaire G par la procédure consistant à initialiser NULL avec $\emptyset$ et à ajouter itérativement dans NULL tous les non-terminaux A tels qu'il existe une production $A \to \alpha$ et que tous les symboles de α sont soit déjà dans NULL, soit égaux à ε. Cette procédure s'achève lorsqu'un examen de toutes les productions de P n'entraîne aucun nouvel ajout dans NULL. Il est facile de voir que cet algorithme termine en un nombre fini d'étapes (au plus $|N|$). Illustrons son fonctionnement sur la grammar 8.6.

Un premier examen de l'ensemble des productions conduit à insérer B dans NULL (à cause de $B \to \varepsilon$); la seconde itération conduit à ajouter A, puisque $A \to B$. Une troisième et

$$
\begin{aligned}
S &\rightarrow c \mid ABS \\
A &\rightarrow B \mid a \\
B &\rightarrow b \mid \varepsilon
\end{aligned}
$$

Grammar 8.6 – Une grammaire (ambiguë) pour $(a \mid b)^\star c$

```
// Initialisation.
foreach A ∈ N do FIRST^0(A) := ∅;
k := 0;
cont := true;
while cont = true do
    k := k + 1;
    foreach A ∈ N do
        FIRST^k(A) := FIRST^{k-1}(A)
    foreach (A → X_1 ... X_k) ∈ P do
        j:= 0;
        repeat
            j := j+1;
            // Par convention, si X_j est terminal, FIRST(X_j) = {X_j}
            FIRST^k(A) := FIRST^k(A) ∪ FIRST^k(X_j)
        until X_j ∉ NULL;
    // Vérifie si un des FIRST a changé; sinon stop
    cont := false;
    foreach A ∈ N do
        if FIRST^k(A) ≠ FIRST^{k-1}(A) then
            cont := true
```

Algorithm 8.7– Calcul de FIRST

dernière itération n'ajoute aucun autre élément dans NULL: en particulier S ne dérive pas ε puisque tout mot du langage engendré par cette grammaire contient au moins un c en dernière position.

Calculer FIRST Nous savons maintenant calculer NULL: il est alors possible d'écrire une procédure pour calculer FIRST(A) pour tout A. Le pseudo-code de cette procédure est donné par l'algorithm 8.7.

Illustrons le fonctionnement de cet algorithme sur la grammar 8.6: l'initialisation conduit à faire FIRST(S) = FIRST(A) = FIRST(B) = $\emptyset$. La première itération ajoute c dans FIRST(S), puis a dans FIRST(A) et b dans FIRST(B); la seconde itération conduit à augmenter FIRST(S) avec a (qui est dans FIRST(A)), puis avec b (qui n'est pas dans FIRST(A), mais comme A est dans NULL, il faut aussi considérer les éléments de FIRST(B)); on ajoute également durant cette itération b dans FIRST(A). Une troisième itération ne change pas ces ensembles, conduisant

finalement aux valeurs suivantes:

$$\begin{aligned} FIRST(S) &= \{a, b, c\} \\ FIRST(A) &= \{a, b\} \\ FIRST(B) &= \{b\} \end{aligned}$$

FOLLOW FOLLOW est nécessaire pour contrôler par anticipation la validité de prédictions de la forme $A \to \varepsilon$, ou, plus généralement $A \overset{\star}{\underset{G}{\Rightarrow}} \varepsilon$: pour valider un tel choix, il faut en effet connaître les terminaux qui peuvent apparaître après A dans un proto-mot. Une fois ces terminaux connus, il devient possible de valider l'application de $A \to \varepsilon$ en vérifiant que le symbole à apparier fait bien partie de cet ensemble; si ce n'est pas le cas, alors l'utilisation de cette production aboutira nécessairement à un échec.

Formellement, on définit:

Definition 8.4 (FOLLOW)**.** *Soit $G = (N, \Sigma, S, P)$ une grammaire CF et A un élément de N. On appelle* FOLLOW(A) *le sous-ensemble de Σ défini par:*

$$\text{FOLLOW}(A) = \{a \in \Sigma, \exists u \in \Sigma^\star, \alpha, \beta \in (N \cup \Sigma)^\star, S \overset{\star}{\Rightarrow} uA\alpha \Rightarrow uAa\beta\}$$

Comment calculer ces ensembles ? En fait, la situation n'est guère plus compliquée que pour le calcul de FIRST: la base de la récursion est que si $A \to X_1 \dots X_n$ est une production de G, alors tout symbole apparaissant après A peut apparaître après X_n. La prise en compte des non-terminaux pouvant dériver ε complique un peu le calcul, et requiert d'avoir au préalable calculé NULL et FIRST. Ce calcul se formalise par l'algorithm 8.8.

Illustrons, de nouveau, le fonctionnement de cette procédure sur la grammar 8.6. La première itération de cet algorithme conduit à examiner la production $S \to ABS$: cette règle présente une configuration traitée dans la première boucle `for` avec $X_j = A$, $l = 0$: les éléments de FIRST(B), soit b, sont ajoutés à FOLLOW$^1(A)$. Comme B est dans NULL, on obtient aussi que les éléments de FIRST(S) sont dans FOLLOW$^1(A)$, qui vaut alors $\{a, b, c\}$ ($j = 1, l = 1$). En considérant, toujours pour cette production, le cas $j = 2, l = 0$, il s'avère que les éléments de FIRST(S) doivent être insérés également dans FOLLOW$^1(B)$. La suite du déroulement de l'algorithme n'apportant aucun nouveau changement, on en reste donc à:

$$\begin{aligned} FOLLOW(S) &= \emptyset \\ FOLLOW(A) &= \{a, b, c\} \\ FOLLOW(B) &= \{a, b, c\} \end{aligned}$$

On notera que FOLLOW(S) est vide: aucun terminal ne peut apparaître à la droite de S dans un proto-mot. Ceci est conforme à ce que l'on constate en observant quelques dérivations: S figure, en effet, toujours en dernière position des proto-mots qui le contiennent.

8.1.4 La table de prédiction

Nous avons montré à la section précédente comment calculer les ensembles FIRST() et FOLLOW(). Nous étudions, dans cette section, comment les utiliser pour construire des

```
// Initialisation
foreach A ∈ N do FOLLOW^0(A) := ∅ ;
foreach (A → X_1 ... X_n) ∈ P do
    for j = 1 to n − 1 do
        if X_j ∈ N then
            for l = 0 to n − j do
                // Inclut le cas où X_{j+1} ... X_{j+l+1} = ε
                if X_{j+1} ... X_{j+l} ∈ NULL* then
                    FOLLOW^0(X_j) = FOLLOW^0(X_j) ∪ FIRST(X_{j+l+1})
k := 0;
cont := true;
while cont = true do
    k := k + 1;
    foreach A ∈ N do
        FOLLOW^k(A) := FOLLOW^{k−1}(A)
    foreach (A → X_1 ... X_n) ∈ P do
        j:= n+1;
        repeat
            j := j-1;
            FOLLOW^k(X_j) := FOLLOW^k(X_j) ∪ FOLLOW^k(A)
        until X_j ∉ NULL;
    // Vérifie si un des FOLLOW a changé; sinon stop
    cont:= false;
    foreach A ∈ N do
        if FOLLOW^k(A) ≠ FOLLOW^{k−1}(A) then cont := true ;
```

Algorithm 8.8– Calcul de FOLLOW

analyseurs efficaces. L'idée que nous allons développer consiste à déduire de ces ensembles une table M (dans $N \times \Sigma$) permettant de déterminer à coup sûr les bons choix à effectuer pour construire déterministiquement une dérivation gauche.

Comme préalable, généralisons la notion de FIRST à des séquences quelconques de $(N \cup \Sigma)^\star$ de la manière suivante [2].

$$\text{FIRST}(\alpha = X_1 \cdots X_k) = \begin{cases} FIRST(X_1) \text{ si } X_1 \notin \text{NULL} \\ FIRST(X_1) \cup \text{ FIRST}(X_2 \cdots X_k) \text{ sinon} \end{cases}$$

Revenons maintenant sur l'intuition de l'analyseur esquissé pour les grammaires SLL(1): pour ces grammaires, une production de type $A \to a\alpha$ est sélectionnée à coup sûr dès que A est le plus à gauche du proto-mot courant et que a est le symbole à apparier dans le mot en entrée.

Dans notre cas, c'est l'examen de l'ensemble des éléments pouvant apparaître en tête de la dérivation de la partie droite d'une production qui va jouer le rôle de sélecteur de la "bonne" production. Formellement, on commence à remplir M en appliquant le principe suivant:

2. Rappelons que nous avons déjà étendu la notion de FIRST (et de FOLLOW) pour des terminaux quelconques par FIRST(a) = FOLLOW(a) = a.

> Si $A \to \alpha$, avec $\alpha \neq \varepsilon$, est une production de G et a est un élément de FIRST(α), alors on insère α dans $M(A, a)$

Ceci signifie simplement que lorsque l'analyseur descendant voit un A en tête du proto-mot courant et qu'il cherche à apparier un a, alors il est licite de prédire α, dont la dérivation peut effectivement débuter par un a.

Reste à prendre en compte le cas des productions de type $A \to \varepsilon$: l'idée est que ces productions peuvent être appliquées lorsque A est le plus à gauche du proto-mot courant, et que le symbole à apparier *peut suivre* A. Ce principe doit en fait être généralisé à toutes les productions $A \to \alpha$ où α ne contient que des symboles dans NULL ($\alpha \in \text{NULL}^\star$). Ceci conduit à la seconde règle de remplissage de M:

> Si $A \to \alpha$, avec $\alpha \in \text{NULL}^\star$, est une production de G et a est un élément de FOLLOW(A), alors insérer α dans $M(A, a)$.

Considérons alors une dernière fois la grammar 8.6. L'application du premier principe conduit aux opérations de remplissage suivantes:

— par le premier principe on insérera c dans $M(S, c)$, puis ABS dans les trois cases $M(S, a)$, $M(S, b)$, $M(S, c)$. Ce principe conduit également à placer a dans $M(A, a)$, B dans $M(A, b)$ puis b dans $M(B, b)$.

— reste à étudier les deux productions concernées par le second principe de remplissage. Commençons par la production $B \to \varepsilon$: elle est insérée dans $M(B, x)$ pour tout symbole dans FOLLOW(B), soit dans les trois cases $M(B, a)$, $M(B, b)$, $M(B, c)$. De même, la partie droite de $A \to B$ doit être insérée dans toutes les cases $M(A, x)$, avec x dans FOLLOW(A), soit dans les trois cases: $M(A, a)$, $M(A, b)$, $M(A, c)$.

On parvient donc à la configuration du table 8.9.

	a	b	c
S	ABS	ABS	ABS c
A	a B	B	B
B	ε	ε b	ε

Table 8.9 – Table d'analyse prédictive pour la grammar 8.6

L'examen du table 8.9 se révèle instructif pour comprendre comment analyser les mots avec cette grammaire. Supposons, en effet, que l'on souhaite analyser le mot bc. Le proto-mot courant étant initialisé avec un S, nous consultons la case $M(S, b)$ du table 8.9, qui nous prescrit de récrire S en ABS. Comme nous n'avons vérifié aucun symbole dans l'opération, nous consultons maintenant la case $M(A, b)$. De nouveau la réponse est sans ambiguïté: appliquer $A \to B$. À ce stade, les choses se compliquent: l'opération suivante demande de consulter la case $M(B, b)$, qui contient deux productions possibles: $B \to b$ et $B \to \varepsilon$. Si, toutefois, on choisit la première, on aboutit rapidement à un succès de l'analyse: b étant

apparié, il s'agit maintenant d'apparier le c, à partir du proto-mot courant: BS. Après consultation de la case $M(B, c)$, on élimine le B; choisir $S \to c$ dans la case $M(S, c)$ achève l'analyse.

Pour aboutir à ce résultat, il a toutefois fallu faire des choix, car le table 8.9 ne permet pas de mettre en œuvre une analyse déterministe. Ceci est dû à l'ambiguïté de la grammar 8.6, dans laquelle plusieurs (en fait une infinité de) dérivations gauches différentes sont possibles pour le mot c (comme pourra s'en persuader le lecteur en en listant quelques-unes).

8.1.5 Analyseurs LL(1)

Nous sommes finalement en mesure de définir les grammaires LL(1) et de donner un algorithme pour les analyser.

Definition 8.5 (Grammaire LL(1))**.** *Une grammaire hors-contexte est* LL(1) *si et seulement si sa table d'analyse prédictive* $M()$ *contient au plus une séquence de* $(N \cup \Sigma)^\star$ *pour chaque valeur* (A, a) *de* $N \times \Sigma$.

Toutes les grammaires ne sont pas LL(1), comme nous l'avons vu à la section précédente en étudiant une grammaire ambiguë. Il est, en revanche, vrai que toute grammaire LL(1) est non ambiguë. La démonstration est laissée en exercice.

Une grammaire LL(1) est susceptible d'être analysée par algorithm 8.10 (on suppose que la construction de M est une donnée de l'algorithme).

8.1.6 LL(1)-isation

Les grammaires LL(1) sont particulièrement sympathiques, puisqu'elles se prêtent à une analyse déterministe fondée sur l'exploitation d'une table de prédiction. Bien que toutes les grammaires ne soient pas aussi accommodantes, il est instructif d'étudier des transformations simples qui permettent de se rapprocher du cas LL(1). Un premier procédé de transformation consiste à supprimer les récursions gauches (directes et indirectes) de la grammaire: cette transformation est implicite dans le processus de mise sous forme normale de Greibach et est décrit à la section 9.2.2. Une seconde transformation bien utile consiste à *factoriser à gauche* la grammaire.

Pour mesurer l'utilité de cette démarche, considérons la grammar 8.11.

$$
\begin{aligned}
S &\to \mathit{if}\ B\ \mathit{then}\ S \\
S &\to \mathit{if}\ B\ \mathit{then}\ S\ \mathit{else}\ S \\
&\dots
\end{aligned}
$$

Grammar 8.11 – Une grammaire non-LL(1) pour *if-then-else*

Ce fragment de grammaire n'est pas conforme à la définition donnée pour les grammaires LL(1), puisqu'il apparaît clairement que le mot-clé *if* sélectionne deux productions possibles pour S. Il est toutefois possible de transformer la grammaire pour se débarrasser de cette

```
// le mot en entrée est:  u = u_1 ... u_n
// initialisation
matched := ε;
tomatch := u_1;
left := S;
i := 1;
while left ≠ ε ∧ i ≤ |u| do
    left = Xβ;
    if X ∈ N then
        if undefined(M(X, tomatch)) then return false ;
        α := M(X, tomatch);
        // Remplace A par α
        left := αβ
    else
        // le symbole de tête est terminal:  on apparie simplement
        if X ≠ u_i then return false ;
    matched := matched · u_i;
    tomatch := u_{i+1};
    i := i + 1
if left = ε ∧ tomatch = ε then
    return true
else
    return false
```

Algorithm 8.10– Analyseur pour grammaire LL(1)

configuration: il suffit ici de factoriser le plus long préfixe commun aux deux parties droites, et d'introduire un nouveau terminal S'. Ceci conduit à la grammar 8.12.

$$
\begin{aligned}
S &\rightarrow \textit{if } B \textit{ then } S\ S' \\
S' &\rightarrow \textit{else } S \mid \varepsilon \\
&\dots
\end{aligned}
$$

Grammar 8.12 – Une grammaire factorisée à gauche pour *if-then-else*

Le mot clé *if* sélectionne maintenant une production unique; c'est aussi vrai pour *else*. Le fragment décrit dans la grammar 8.12 devient alors susceptible d'être analysé déterministiquement (vérifiez-le en examinant comment appliquer à coup sûr $S' \rightarrow \varepsilon$).

Ce procédé se généralise au travers de la construction utilisée pour démontrer le théorème suivant:

Theorem 8.6 (Factorisation gauche)**.** *Soit $G = (N, \Sigma, S, P)$ une grammaire hors-contexte, alors il existe une grammaire équivalente G' telle que si $A \rightarrow X_1 \cdots X_k$ et $A \rightarrow Y_1 \cdots Y_l$ sont deux A-productions de G', alors $X_1 \neq Y_1$.*

Proof. La preuve repose sur le procédé de transformation suivant. Supposons qu'il existe une série de A-productions dont les parties droites débutent toutes par le même symbole X. En remplaçant toutes les productions $A \rightarrow X\alpha_i$ par l'ensemble $\{A \rightarrow XA' , A' \rightarrow \alpha_i\}$, où A' est

un nouveau symbole non-terminal introduit pour la circonstance, on obtient une grammaire équivalente à G. Si, à l'issue de cette transformation, il reste des A'-productions partageant un préfixe commun, il est possible de répéter cette procédure, et de l'itérer jusqu'à ce qu'une telle configuration n'existe plus. Ceci arrivera au bout d'un nombre fini d'itérations, puisque chaque transformation a le double effet de réduire le nombre de productions potentiellement problématiques, ainsi que de réduire la longueur des parties droites. □

8.1.7 Quelques compléments

Détection des erreurs

Le rattrapage sur erreur La détection d'erreur, dans un analyseur LL(1), correspond à une configuration (non-terminal A le plus à gauche, symbole a à apparier) pour laquelle la table d'analyse ne prescrit aucune action. Le message à donner à l'utilisateur est alors clair:

> Arrivé au symbole numéro X, j'ai rencontré un a alors que j'aurais dû avoir . . . (suit l'énumération des symboles tels que $M(A, .)$ est non-vide).

Stopper là l'analyse est toutefois un peu brutal pour l'utilisateur, qui, en général, souhaite que l'on détecte en une seule passe toutes les erreurs de syntaxe. Deux options sont alors possibles pour continuer l'analyse:

— s'il n'existe qu'un seul b tel que $M(A, b)$ est non-vide, on peut faire comme si on venait de voir un b, et continuer. Cette stratégie de récupération d'erreur par insertion comporte toutefois un risque: celui de déclencher une cascade d'insertions, qui pourraient empêcher l'analyseur de terminer correctement;

— l'alternative consisterait à détruire le a et tous les symboles qui le suivent jusqu'à trouver un symbole pour lequel une action est possible. Cette méthode est de loin préférable, puisqu'elle conduit à une procédure qui est assurée de se terminer. Pour obtenir un dispositif de rattrapage plus robuste, il peut être souhaitable d'abandonner complètement tout espoir d'étendre le A et de le faire disparaître: l'analyse reprend alors lorsque l'on trouve, dans le flux d'entrée, un symbole dans FOLLOW(A).

Les grammaires LL(k) Nous l'avons vu, toutes les grammaires ne sont pas LL(1), même après élimination des configurations les plus problématiques (récursions gauches...). Ceci signifie que, pour au moins un couple (A, a), $M(A, a)$ contient plus d'une entrée, indiquant que plusieurs prédictions sont en concurrence. Une idée simple pour lever l'indétermination concernant la "bonne" expansion de A consiste à augmenter le regard avant. Cette intuition se formalise à travers la notion de grammaire *LL(k)* et d'analyseur LL(k): pour ces grammaires, il suffit d'un regard avant de k symboles pour choisir sans hésitation la A-production à appliquer; les tables d'analyse correspondantes croisent alors des non-terminaux (en ligne) avec des mots de longueur k (en colonne). Il est important de réaliser que ce procédé ne permet pas de traiter toutes les grammaires: c'est évidemment vrai pour les grammaires ambiguës; mais il existe également des grammaires non-ambiguës, pour lesquelles aucun regard avant de taille borné ne permettra une analyse descendante déterministe.

Ce procédé de généralisation des analyseurs descendants, bien que fournissant des résultats théoriques importants, reste d'une donc utilité pratique modeste, surtout si on compare cette famille d'analyseurs à l'autre grande famille d'analyseurs déterministes, les analyseurs de type LR, qui font l'objet de la section suivante.

8.1.8 Un exemple complet commenté

Nous détaillons dans cette section les constructions d'un analyseur LL pour la grammar 8.5, qui est rappelée dans la grammar 8.13, sous une forme légèrement modifiée. L'axiome est la variable Z.

$$\begin{aligned} Z &\rightarrow S\# \\ S &\rightarrow S + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (S) \mid D \\ D &\rightarrow 1 \mid 2 \end{aligned}$$

Grammar 8.13 – Une grammaire (simplifiée) pour les expressions arithmétiques

La première étape de la construction consiste à se débarrasser des productions récursives à gauche, puis à factoriser à gauche la grammaire résultante. Au terme de ces deux étapes, on aboutit à la grammar 8.14, qui comprend deux nouvelles variables récursives à droite, S' et T', et dont nous allons montrer qu'elle est effectivement LL(1).

$$\begin{aligned} Z &\rightarrow S\# \\ S &\rightarrow TS' \\ S' &\rightarrow +TS' \mid \varepsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \varepsilon \\ F &\rightarrow (S) \mid D \\ D &\rightarrow 1 \mid 2 \end{aligned}$$

Grammar 8.14 – Une grammaire LL pour les expressions arithmétiques

Le calcul de NULL permet d'identifier S' et T' comme les seules variables dérivant le mot vide. Les étapes du calcul des ensembles FIRST et FOLLOW sont détaillées dans le table 8.15.

Il est alors possible de déterminer la table d'analyse prédictive, représentée dans le table 8.16. Cette table est sans conflit, puisque chaque case contient au plus une production.

Cette table permet effectivement d'analyser déterministiquement des expressions arithmétiques simples, comme le montre la trace d'exécution du table 8.17.

	FIRST				
It.	1	2	3	4	5
Z				(	(12
S			(	(12	(12
S'	+	+	+	+	+
T		(	(12	(12	(12
T'	*	*	*	*	*
F	(	(12	(12	(12	(12
D	12	12	12	12	12

	FOLLOW		
It.	0	1	2
Z			
S	#)	#)	#)
S'		#)	#)
T	+	+#)	+#)
T'		+#)	+#)
F	*	*+#)	*+#)
D		*+#)	*+#)

Table 8.15 – Calcul de FIRST et FOLLOW

	#	+	*	(	)	1	2
Z				$Z \to S\#$		$Z \to S\#$	$Z \to S\#$
S				$S \to TS'$		$S \to TS'$	$S \to TS'$
S'	$S' \to \varepsilon$	$S' \to +TS'$			$S' \to \varepsilon$		
T				$T \to FT'$		$T \to FT'$	$T \to FT'$
T'	$T' \to \varepsilon$	$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$		
F				$T \to (S)$		$T \to D$	$T \to D$
D						$D \to 1$	$D \to 2$

Table 8.16 – Table d'analyse prédictive pour la grammaire de l'arithmétique

8.2 Analyseurs LR

8.2.1 Concepts

Comme évoqué à la section 7.2, les analyseurs ascendants cherchent à récrire le mot à analyser afin de se ramener, par des réductions successives, à l'axiome de la grammaire. Les bifurcations dans le graphe de recherche correspondent alors aux alternatives suivantes:

(i) la partie droite α d'une production $A \to \alpha$ est trouvée dans le proto-mot courant; on choisit de remplacer α par A pour construire un nouveau proto-mot et donc d'appliquer une *réduction*.

(ii) on poursuit l'examen du proto-mot courant en considérant un autre facteur α', obtenu, par exemple, en étendant α par la droite. Cette action correspond à un *décalage* de l'entrée.

Afin de rationaliser l'exploration du graphe de recherche correspondant à la mise en œuvre de cette démarche, commençons par décider d'une stratégie d'examen du proto-mot courant: à l'instar de ce que nous avons mis en œuvre dans l'algorithm 7.1, nous l'examinerons toujours depuis la gauche vers la droite. Si l'on note α le proto-mot courant, factorisé en $\alpha = \beta\gamma$, où β est la partie déjà examinée, l'alternative précédente se récrit selon:

— β possède un suffixe δ correspondant à la partie droite d'une règle: réduction et développement d'un nouveau proto-mot.

— β est étendu par la droite par décalage d'un nouveau terminal.

Pile	Symbole
Z	2
S#	2
TS'#	2
FT'S'#	2
DT'S'#	2
2T'S'#	*
2*FT'S'#	(
2*(S)T'S'#	1
2*(TS')T'S'#	1
2*(FT'S')T'S'#	1
2*(DT'S')T'S'#	1

Pile (suite)	Symbole
2*(1T'S')T'S'#	+
2*(1S')T'S'#	+
2*(1+TS')T'S'#	+
2*(1+FT'S')T'S'#	+
2*(1+DT'S')T'S'#	+
2*(1+2T'S')T'S'#	)
2*(1+2S')T'S'#	)
2*(1+2)T'S'#	#
2*(1+2)S'#	#
2*(1+2)#	#
succès	

Table 8.17 – Trace de l'analyse déterministe de $2 * (1 + 2)$

Cette stratégie conduit à la construction de dérivations *droites* de l'entrée courante. Pour vous en convaincre, remarquez que le suffixe γ du proto-mot courant n'est jamais modifié: il n'y a toujours que des terminaux à droite du symbole récrit, ce qui est conforme à la définition d'une dérivation droite.

Illustrons ce fait en considérant la grammar 8.18.

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA \mid b \\ B &\rightarrow bB \mid a \end{aligned}$$

Grammar 8.18 – Une grammaire pour $a^\star bb^\star a$

Soit alors $u = abba$; une analyse ascendante de ce mot est reproduite au table 8.19.

En considérant les réductions opérées, on obtient la dérivation: $S \underset{G}{\Rightarrow} AB \underset{G}{\Rightarrow} AbB \underset{G}{\Rightarrow} Aba \underset{G}{\Rightarrow} aAba \underset{G}{\Rightarrow} abba$, qui est effectivement une dérivation droite. On note également qu'on a introduit un troisième type d'action de l'analyseur, consistant à *accepter* l'entrée comme un mot de la grammaire. Il existe enfin un quatrième type d'action, non représenté ici, consistant à diagnostiquer une situation d'échec de l'analyse.

Dernière remarque concernant cette trace: les différentes transformations possibles de α via les actions de réduction et de décalage n'agissent que sur les suffixes de β. Ceci suggère d'implanter β sous la forme d'une pile, sur laquelle s'accumulent (puis se réduisent) progressivement les symboles de u. Si l'on adopte ce point de vue, une pile correspondant à la trace du table 8.19 passerait par les états successifs suivants: a, ab, aA, A, Ab, Aba, AbB, AB, S. Bien naturellement, cette implantation demanderait également de conserver un pointeur vers la position courante dans u, afin de savoir quel symbole empiler. Dans la suite, nous ferons l'hypothèse que β est effectivement implanté sous la forme d'une pile.

Comment procéder pour rendre ce processus déterministe ? Cherchons des éléments de réponse dans la trace d'analyse présentée dans le table 8.19. Ce processus contient quelques

α	β	γ	Action
$abba$	ε	$abba$	décaler
$abba$	a	bba	décaler
$abba$	ab	ba	réduire par $A \to b$
$aAba$	aA	ba	réduire par $A \to aA$
Aba	A	ba	décaler
Aba	Ab	a	décaler
Aba	Aba	ε	réduire par $B \to a$
AbB	AbB	ε	réduire par $B \to bB$
AB	AB	ε	réduire par $S \to AB$
S			fin: accepter l'entrée

Table 8.19 – Analyse ascendante de $u = abba$

actions déterministes, comme la première opération de décalage: lorsqu'en effet β (la pile) ne contient aucun suffixe correspondant à une partie droite de production, décaler est l'unique action possible.

De manière duale, lorsque γ est vide, décaler est impossible: il faut nécessairement réduire, lorsque cela est encore possible: c'est, par exemple, ce qui est fait durant la dernière réduction.

Un premier type de choix correspond au second décalage: $\beta = a$ est à ce moment égal à une partie droite de production ($B \to a$), pourtant on choisit ici de décaler (ce choix est presque toujours possible) plutôt que de réduire. Notons que la décision prise ici est la (seule) bonne décision: réduire prématurément aurait conduit à positionner un B en tête de β, conduisant l'analyse dans une impasse: B n'apparaissant en tête d'aucune partie droite, il aurait été impossible de le réduire ultérieurement. Un second type de configuration (non représentée ici) est susceptible de produire du non-déterminisme: il correspond au cas où l'on trouve en queue de β deux parties droites de productions: dans ce cas, il faudra choisir entre deux réductions concurrentes.

Résumons-nous: nous voudrions, de proche en proche, et par simple consultation du sommet de la pile, être en mesure de décider déterministiquement si l'action à effectuer est un décalage ou bien une réduction (et dans ce cas quelle est la production à utiliser), ou bien encore si l'on se trouve dans une situation d'erreur. Une condition suffisante serait que, pour chaque production p, on puisse décrire l'ensemble L_p des configurations de la pile pour lesquelles une réduction par p est requise; et que, pour deux productions p_1 et p_2 telles que $p_1 \neq p_2$, L_{p_1} et L_{p_2} soient toujours disjoints. Voyons à quelle(s) condition(s) cela est possible.

8.2.2 Analyseurs LR(0)

Pour débuter, formalisons la notion de contexte LR(0) d'une production.

Definition 8.7 (Contexte LR(0))**.** *Soit G une grammaire hors-contexte, et $p = (A \to \alpha)$ une*

production de G, on appelle contexte LR(0) *de p le langage* $L_{A\to\alpha}$ *défini par:*

$$L_{A\to\alpha} = \{\gamma = \beta\alpha \in (\Sigma \cup N)^\star \ tq.\ \exists v \in \Sigma^\star, S \overset{\star}{\underset{D}{\Rightarrow}} \beta A v \underset{D}{\Rightarrow} \beta\alpha v\}$$

En d'autres termes, tout mot γ de $L_{A\to\alpha}$ contient un suffixe α et est tel qu'il existe une réduction d'un certain γv en S qui débute par la réduction de α en A. Chaque fois qu'un tel mot γ apparaît dans la pile d'un analyseur ascendant gauche-droit, il est utile d'opérer la réduction $A \to \alpha$; à l'inverse, si l'on trouve dans la pile un mot absent de $L_{A\to\alpha}$, alors cette réduction ne doit pas être considérée, même si ce mot se termine par α.

Examinons maintenant de nouveau la grammar 8.18 et essayons de calculer les langages L_p pour chacune des productions. Le cas de la première production est clair: il faut impérativement réduire lorsque la pile contient AB, et c'est là le seul cas possible. On déduit directement que $L_{S\to AB} = \{AB\}$. Considérons maintenant $A \to aA$: la réduction peut survenir quel que soit le nombre de a présents dans la pile. En revanche, si la pile contient un symbole différent de a, c'est qu'une erreur aura été commise. En effet:

— une pile contenant une séquence baA ne pourra que se réduire en AA, dont on ne sait plus que faire; ceci proscrit également les piles contenant plus d'un A, qui aboutissent pareillement à des configurations d'échec;

— une pile contenant une séquence $B \ldots A$ ne pourra que se réduire en BA, dont on ne sait non plus comment le transformer.

En conséquence, on a: $L_{A\to aA} = aa^\star A$. Des considérations similaires nous amènent à conclure que:

— $L_{A\to b} = a^\star b$;

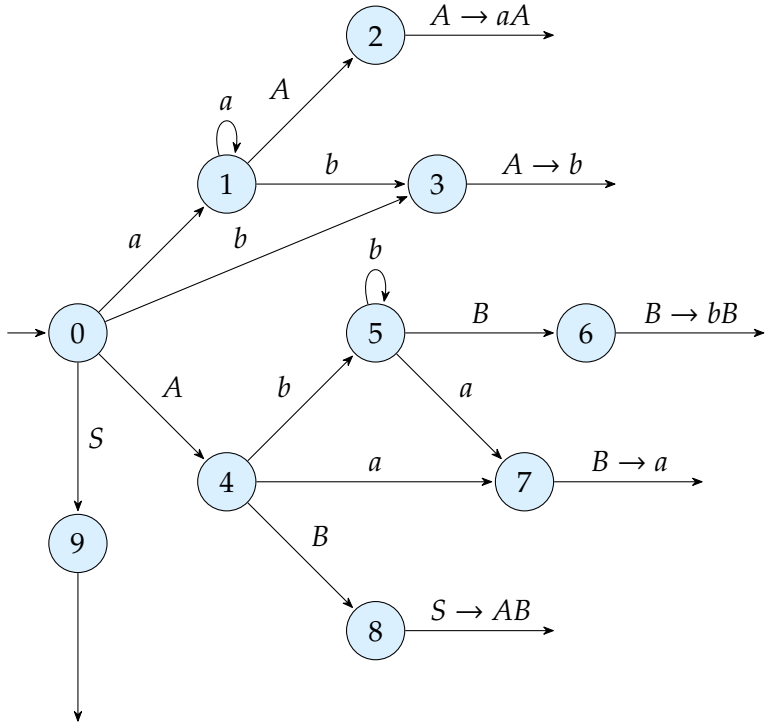
— $L_{B\to bB} = Abb^\star B$;

— $L_{B\to a} = Ab^\star a$;

Chacun des ensembles L_p se décrivant par une expression rationnelle, on déduit que l'ensemble des L_p se représente sur l'automate 8.20, qui réalise l'union des langages L_p. Vous noterez de plus que (i) l'alphabet d'entrée de cet automate contient à la fois des symboles terminaux et non-terminaux de la grammaire; (ii) les états finaux de l'automate 8.20 sont associés aux productions correspondantes; (iii) toute situation d'échec dans l'automate correspond à un échec de l'analyse: en effet, ces configurations sont celles où la pile contient un mot dont l'extension ne peut aboutir à aucune réduction: il est alors vain de continuer l'analyse.

Comment utiliser cet automate ? Une approche naïve consiste à mettre en œuvre la procédure suivante: partant de l'état initial $q_i = q_0$ et d'une pile vide on applique des décalages jusqu'à atteindre un état final q. On réduit ensuite la pile selon la production associée à q, donnant lieu à une nouvelle pile β qui induit un repositionnement dans l'état $q_i = \delta^\star(q_0, \beta)$ de l'automate. La procédure est itérée jusqu'à épuisement simultané de la pile et de u. Cette procédure est formalisée à travers l'algorithm 8.21.

L'algorithm 8.21 est inefficace: en effet, à chaque réduction, on se repositionne à l'état initial de l'automate, perdant ainsi le bénéfice de l'analyse des symboles en tête de la pile. Une meilleure implantation consiste à mémoriser, pour chaque symbole de la pile, l'état q atteint dans A. À la suite d'une réduction, on peut alors directement se positionner sur q pour poursuivre l'analyse. Cette amélioration est mise en œuvre dans l'algorithm 8.22.



Automaton 8.20 – L'automate des réductions licites

```
// Initialisation.
β := ε // Initialisation de la pile.
q := q0 // Se positionner dans l'état initial.
i := j := 0;
while i ≤ |u| do
    while j ≤ |β| do
        j := j + 1;
        q := δ(q, βj)
    while q ∉ F do
        β := βui // Empilage de ui
        if δ(q, ui) existe then q := δ(q, ui) else return false ;
        i := i + 1
    // Réduction de p = A → γ.
    β := βγ^−1 A ;
    j := 0;
    q := q0
if β = S ∧ q ∈ F then return true else return false;
```

Algorithm 8.21– Reconnaissance ascendante guidée par un automate $A = (\Sigma \cup N, Q, q_0, F, \delta)$. Version 1

```
// Initialisations.
β := (ε, q_0) // Initialisation de la pile.
q := q_0 // Se positionner dans l'état initial.
i := 0;
while i ≤ |u| do
    while q ∉ F do
        if δ(q, u_i) existe then
            q := δ(q, u_i) // Progression dans A.
            push(β, (u_i, q)) // Empilage de (u_i, q).
            i := i + 1
        else
            return false
    // On atteint un état final: réduction de p = A → α.
    j := 0;
    // Dépilage de α.
    while j < |α| do
        pop(β);
        j := j + 1
    // (x, q) est sur le sommet de la pile: repositionnement.
    push(β, (A, δ(q, A)));
    q := δ(q, A);
if β = (S, q) ∧ q ∈ F then return true else return false ;
```

Algorithm 8.22– Reconnaissance ascendante guidée par un automate $A = (\Sigma \cup N, Q, q_0, F, \delta)$. Version 2

Notez, pour finir, que l'on effectue en fait deux sortes de transitions dans A: celles qui sont effectuées directement à l'issue d'un décalage et qui impliquent une extension de la pile; et celles qui sont effectuées à l'issue d'une réduction et qui consistent simplement à un repositionnement ne nécessitant pas de nouveau décalage. Ces deux actions sont parfois distinguées, l'une sous le nom de décalage (*shift*), l'autre sous le nom de *goto*.

De l'automate précédent, se déduit mécaniquement une table d'analyse (dite LR(0)), donnée pour l'automate 8.20 au table 8.23. Cette table résume les différentes actions à effectuer en fonction de l'état courant de l'automate.

Le table 8.23 se consulte de la façon suivante. Pour chaque état, la colonne "Action" désigne l'unique type d'action à effectuer:

s
: effectuer un décalage. Lire le symbole en tête de pile, si c'est un x, consulter la colonne correspondante dans le groupe "Shift", et transiter dans l'état correspondant. *Attention*: le décalage a bien lieu de *manière inconditionnelle*, c'est-à-dire indépendamment de la valeur du symbole empilé; en revanche, cette valeur détermine la transition de l'automate à exercer.

r $A \to \alpha$
: réduire la pile selon $A \to \alpha$. À l'issue de la réduction, le A sera posé sur la pile. Transiter vers l'état correspondant à A dans le groupe "Goto" et à l'état au sommet de la pile après en avoir retiré α.

État	Action	Shift a	Shift b	Goto A	Goto B
0	s	1	3	4	
1	s	1	3	2	
2	r $A \to aA$				
3	r $A \to b$				
4	s	7	5		8
5	s	7	5		6
6	r $B \to bB$				
7	r $B \to a$				
8	r $S \to AB$				

Table 8.23 – Une table d'analyse LR(0)

Toutes les autres configurations (qui correspondent aux cases vides de la table) sont des situations d'erreur. Pour distinguer, dans la table, les situations où l'analyse se termine avec succès, il est courant d'utiliser la transformation suivante:

— on ajoute un nouveau symbole terminal représentant la fin du texte, par exemple #;

— on transforme G en G' en ajoutant un nouvel axiome Z et une nouvelle production $Z \to S\#$, où S est l'axiome de G.

— on transforme l'entrée à analyser en $u\#$;

— l'état (final) correspondant à la réduction $Z \to S\#$ est (le seul) état d'acceptation, puisqu'il correspond à la fois à la fin de l'examen de u (# est empilé) et à la présence du symbole S en tête de la pile.

Il apparaît finalement, qu'à l'aide du table 8.23, on saura analyser la grammar 8.18 de manière déterministe et donc avec une complexité linéaire. Vous pouvez vérifier cette affirmation en étudiant le fonctionnement de l'analyseur pour les entrées $u = ba$ (succès), $u = ab$ (échec), $u = abba$ (succès).

Deux questions se posent alors : (i) peut-on, pour toute grammaire, construire un tel automate ? (ii) comment construire l'automate à partir de la grammaire G ? La réponse à (i) est non: il existe des grammaires (en particulier les grammaires ambiguës) qui résistent à toute analyse déterministe. Il est toutefois possible de chercher à se rapprocher de cette situation, comme nous le verrons à la section 8.2.3. Dans l'intervalle, il est instructif de réfléchir à la manière de construire automatiquement l'automate d'analyse A.

L'intuition de la construction se fonde sur les remarques suivantes. Initialement, on dispose de u, non encore analysé, qu'on aimerait pouvoir réduire en S, par le biais d'une série non-encore déterminée de réductions, mais qui s'achèvera nécessairement par une réduction de type $S \to \alpha$.

Supposons, pour l'instant, qu'il n'existe qu'une seule S-production: $S \to X_1 \dots X_k$: le but original de l'analyse "aboutir à une pile dont S est l'unique symbole en ayant décalé tous les symboles de u" se reformule alors en: "aboutir à une pile égale à $X_1 \dots X_k$ en ayant décalé tous les symboles de u". Ce nouveau but se décompose naturellement en une série d'étapes

qui vont devoir être accomplies séquentiellement: d'abord parvenir à une configuration où X_1 est "au fond" de la pile, puis faire que X_2 soit empilé juste au-dessus de X_1...

Conserver la trace de cette série de buts suggère d'insérer dans l'automate d'analyse une branche correspondant à la production $S \to X_1 \dots X_k$, qui s'achèvera donc sur un état final correspondant à la réduction de $X_1 \dots X_k$ en S. Une telle branche est représentée à la fig. 8.24.

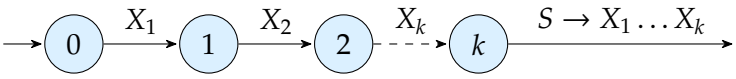


Figure 8.24 – Une branche de l'automate

Chaque état le long de cette branche correspond à la résolution d'un sous-but supplémentaire, la transition X_i annonçant l'empilage de X_i, et déclenchant la recherche d'un moyen d'empiler X_{i+1}. Pour formaliser cette idée, nous introduisons le concept de *production (ou règle) pointée* [3].

Definition 8.8 (Production pointée)**.** *Une* production pointée *d'une grammaire hors-contexte G est un triplet (A, α, β) de $N \times (N \cup \Sigma)^\star \times (N \cup \Sigma)^\star$, avec $A \to \gamma = \alpha\beta$ une production de G. Une production pointée est notée avec un point: $A \to \alpha \bullet \beta$.*

Une production pointée exprime la résolution partielle d'un but: $A \to \alpha \bullet \beta$ exprime que la résolution du but "empiler A en terminant par la réduction $A \to \alpha\beta$" a été partiellement accomplie, en particulier que α a déjà été empilé et qu'il reste encore à empiler les symboles de β. Chaque état de la branche de l'automate correspondant à la production $A \to \gamma$ s'identifie ainsi à une production pointée particulière, les états initiaux et finaux de cette branche correspondant respectivement à: $A \to \bullet\gamma$ et $A \to \gamma\bullet$.

Retournons à notre problème original, et considérons maintenant le premier des sous-buts: "empiler X_1 au fond de la pile". Deux cas de figure sont possibles:

— soit X_1 est symbole terminal: le seul moyen de l'empiler consiste à effectuer une opération de décalage, en cherchant un tel symbole en tête de la partie non encore analysée de l'entrée courante.

— soit X_1 est un non-terminal: son insertion dans la pile résulte nécessairement d'une série de réductions, dont la dernière étape concerne une X_1-production : $X_1 \to Y_1 \dots Y_n$. De nouveau, le but "observer X_1 au fond de la pile" se décompose en une série de sous-buts, donnant naissance à une nouvelle "branche" de l'automate pour le mot $Y_1 \dots Y_n$. Comment relier ces deux branches ? Tout simplement par une transition ε entre les deux états initiaux, indiquant que l'empilage de X_1 se résoudra en commençant l'empilage de Y_1 (voir la fig. 8.25). S'il existe plusieurs X_1-productions, on aura une branche (et une transition ε) par production.

  Ce procédé se généralise: chaque fois qu'une branche porte une transition $\delta(q, X) = r$, avec X un non-terminal, il faudra ajouter une transition entre q et tous les états initiaux des branches correspondants aux X-productions.

De ces remarques découle un procédé systématique pour construire un ε-NFA $(N \cup \Sigma, Q, q_0, F, \delta)$ permettant de guider une analyse ascendante à partir d'une grammaire $G = (\Sigma, N, Z, P)$, telle que Z est non-récursif et ne figure en partie gauche que dans l'unique règle $Z \to \alpha$:

3. On trouve également le terme d'*item* et en anglais de *dotted rule*.

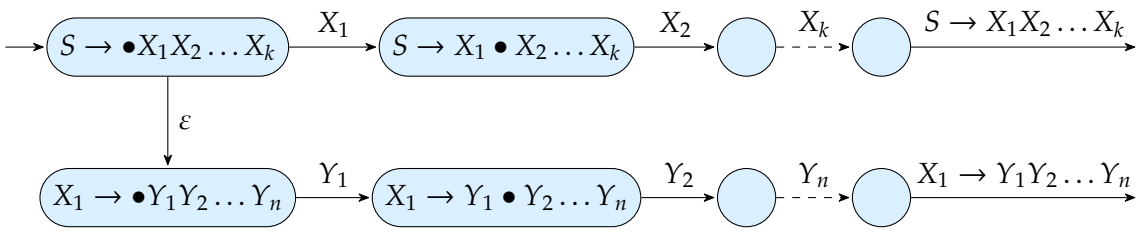


Figure 8.25 – Deux branches de l'automate

— $Q = \{[A \to \alpha \bullet \beta]$ avec $A \to \alpha\beta \in P\}$
— $q_0 = [Z \to \bullet\alpha]$
— $F = \{[A \to \alpha\bullet]$ avec $A \to \alpha \in P\}$; à chaque état final est associée une production $A \to \alpha$
— $\forall q = [A \to \alpha \bullet X\beta] \in Q, \delta(q, X) = [A \to \alpha X \bullet \beta]$
— $\forall q = [A \to \alpha \bullet B\beta] \in Q$ tq. $B \in N, \forall q' = [B \to \bullet\gamma], \delta(q, \varepsilon) = q'$.

Le théorème suivant, non démontré ici, garantit que ce procédé de construction permet effectivement d'identifier les contextes LR(0) des productions.

Theorem 8.9. *Soit A l'automate fini dérivé de G par la construction précédente, alors: $\delta^\star(q_0, \gamma) = [A \to \alpha \bullet \beta]$ si et seulement si:*

(i) $\exists \eta, \gamma = \eta\alpha$
(ii) $\gamma\beta \in L_{A \to \alpha\beta}$

Ce résultat assure, en particulier, que lorsque l'on atteint un état final de l'automate (pour $\beta = \varepsilon$), la pile contient effectivement un mot appartenant au contexte LR(0) de la production associée à l'état final atteint.

Il est naturellement possible de déterminiser cet automate, en appliquant les algorithmes de suppression des transitions spontanées (voir le theorem 4.13), puis la construction des sous-ensembles décrite dans le theorem 4.9; on prendra soin de propager lors de ces transformations l'information de réduction associée aux états finaux des branches. En clair, si un état q du déterminisé contient une production pointée originale de type $A \to \alpha\bullet$, alors q sera un état (final) dans lequel cette réduction est possible. Le lecteur est vivement encouragé à entreprendre cette démarche et vérifier qu'il retrouve bien, en partant de la grammar 8.18, un automate ressemblant fortement à l'automate 8.20.

Une construction directe de l'automate LR(0) consiste à construire à la volée les ensembles d'items qui définissent les états de l'automate déterminisé, ainsi que les transitions associées. Soit I un ensemble d'items, on définit tout d'abord la *clôture* de I comme:

Definition 8.10 (Clôture d'un ensemble d'items LR(0))**.** *La clôture $cl(I)$ d'un ensemble I d'items LR(0) est le plus petit ensemble vérifiant:*

— $I \subset cl(I)$
— *si* $[X \to \alpha \bullet Y\beta] \in cl(I)$, *alors* $\forall Y \to \gamma, [Y \to \bullet\gamma] \in cl(I)$

L'automate LR(0) est alors défini par:

— $q_0 = cl([z \rightarrow \bullet S\#]$ est l'état initial
— $\forall I, \forall a \in \Sigma, \delta(I, a) = cl\{[X \rightarrow \alpha a \bullet \beta]$, avec $[X \rightarrow \alpha \bullet a\beta] \in I\}$
— $\forall I, \forall X \in N, \delta(I, X) = cl\{[X \rightarrow \alpha X \bullet \beta]$, avec $[X \rightarrow \alpha \bullet X\beta] \in I\}$
— $F = \{I, \exists [X \rightarrow \alpha \bullet] \in I\}$

On déduit finalement de cet automate déterministe, par le procédé utilisé pour construire le table 8.23, la *table d'analyse LR(0)*.

Definition 8.11 (Grammaire LR(0))**.** *Une grammaire hors-contexte est* LR(0)[4] *si sa table* $T()$ *d'analyse LR(0) est telle que: pour toute ligne i (correspondant à un état de l'automate), soit il existe* $x \in N \cup \Sigma$ *tel que* $T(i, x)$ *est non-vide et* $T(i, *)$ *est vide; soit* $T(i, *)$ *est non-vide et contient une réduction unique.*

Une grammaire LR(0) peut être analysée de manière déterministe. La définition d'une grammaire LR(0) correspond à des contraintes qui sont souvent, dans la pratique, trop fortes pour des grammaires "réelles", pour lesquelles la construction de la table LR(0) aboutit à des conflits. Le premier type de configuration problématique correspond à une indétermination entre décaler et réduire (conflit *shift/reduce*); le second type correspond à une indétermination sur la réduction à appliquer; on parle alors de conflit *reduce/reduce*. Vous noterez, en revanche, qu'il n'y a jamais de conflit *shift/shift*. Pourquoi ?

Comme pour les analyseurs descendants, il est possible de lever certaines indéterminations en s'autorisant un regard en avant sur l'entrée courante; les actions de l'analyseur seront alors conditionnées non seulement par l'état courant de l'automate d'analyse, mais également par les symboles non-analysés. Ces analyseurs font l'objet de la section qui suit.

8.2.3 Analyseurs SLR(1), LR(1), LR(k)...

Regarder vers l'avant

Commençons par étudier le cas le plus simple, celui où les conflits peuvent être résolus avec un regard avant de 1. C'est, par exemple, le cas de la grammar 8.26:

$$\begin{array}{ll} S \rightarrow E\# & (1) \\ E \rightarrow T + E \mid T & (2), (3) \\ T \rightarrow x & (4) \end{array}$$

Grammar 8.26 – Une grammaire non-LR(0)

Considérons une entrée telle que $x + x + x$: sans avoir besoin de construire la table LR(0), il apparaît qu'après avoir empilé le premier x, puis l'avoir réduit en T, par $T \rightarrow x$, deux alternatives vont s'offrir à l'analyseur:

— soit immédiatement réduire T en E
— soit opérer un décalage et continuer de préparer une réduction par $E \rightarrow T + E$.

4. Un 'L' pour *left-to-right*, un 'R' pour *rightmost derivation*, un 0 pour *0 lookahead* ; en effet, les décisions (décalage vs. réduction) sont toujours prises étant uniquement donné l'état courant (le sommet de la pile).

Il apparaît pourtant que '+' ne peut jamais suivre E dans une dérivation réussie: vous pourrez le vérifier en construisant des dérivations droites de cette grammaire. Cette observation anticipée du prochain symbole à empiler suffit, dans le cas présent, à restaurer le déterminisme. Comment traduire cette intuition dans la procédure de construction de l'analyseur?

La manière la plus simple de procéder consiste à introduire des conditions supplémentaires aux opérations de réduction: dans les analyseurs LR(0), celles-ci s'appliquent de manière inconditionnelle, c'est-à-dire quel que soit le prochain symbole apparaissant dans l'entrée courante (le regard avant): à preuve, les actions de réduction apparaissent dans la table d'analyse dans une colonne séparée. Pourtant, partant d'une configuration dans laquelle la pile est $\beta\gamma$ et le regard avant est a, réduire par $X \to \gamma$ aboutit à une configuration dans laquelle la pile vaut βX. Si l'analyse devait se poursuivre avec succès, on aurait alors construit une dérivation droite $S \overset{\star}{\Rightarrow} \beta Xa... \Rightarrow \beta\gamma a$, dans laquelle la lettre a suit directement X. En conséquence, la réduction ne peut déboucher sur un succès que si a est un successeur de X, information qui est donnée dans l'ensemble FOLLOW(X) (cf. la section 8.1.3).

Ajouter cette contrainte supplémentaire à la procédure de construction de la table d'analyse consiste à conditionner les actions de réduction $X \to \gamma$ par l'existence d'un regard avant appartenant à l'ensemble FOLLOW(X): elles figureront dans la table d'analyse dans les colonnes dont l'en-tête est un terminal vérifiant cette condition.

Si cette restriction permet d'aboutir à une table ne contant qu'une action par cellule, alors on dit que la grammaire est *SLR(1)* (pour *simple LR*, quand on s'autorise un regard avant de 1).

Illustrons cette nouvelle procédure de la construction de la table d'analyse. Les actions de l'automate LR(0) sont données dans le table 8.27a, dans laquelle on observe un conflit décaler/réduire dans l'état 2. Le calcul des ensembles FOLLOW aboutit à FOLLOW(S) = $\emptyset$, FOLLOW(E) = {#}, FOLLOW(T) = {#, +}, ce qui permet de préciser les conditions d'application des réductions, donnant lieu à la table d'analyse reproduite dans le table 8.27b. La grammaire est donc SLR(1).

En guise d'application, montrez que la grammar 8.5 est SLR(1). On introduira pour l'occasion une nouvelle production $Z \to S\#$; vous pourrez également considérer D comme un terminal valant pour n'importe quel nombre entier.

LR(1)

Si cette procédure échoue, l'étape suivante consiste à modifier la méthode de construction de l'automate LR(0) en choisissant comme ensemble d'états toutes les paires [5] constituées d'une production pointée et d'un terminal. On note ces états $[A \to \beta \bullet \gamma, a]$. La construction de l'automate d'analyse LR (ici LR(1)) $(N \cup \Sigma, Q, q_0, F, \delta)$ se déroule alors comme suit (on suppose toujours que l'axiome Z n'est pas récursif et n'apparaît que dans une seule règle):

— $Q = \{[A \to \alpha \bullet \beta, a]$ avec $A \to \alpha\beta \in P$ et $a \in \Sigma\}$

— $q_0 = [Z \to \bullet\alpha, ?]$, où ? est un symbole "joker" qui vaut pour n'importe quel symbole terminal;

5. En fait, les prendre *toutes* est un peu excessif, comme il apparaîtra bientôt.

		Shift			Goto	
État	Action	x	+	#	E	T
0	s	3			1	2
1	s			6		
2	s/r3		4			
3	r4					
4	s	3			5	2
5	r2					
6	acc					

(a) Table d'analyse LR(0) avec conflit (état 2)

	Action			Goto	
État	x	+	#	E	T
0	s3			g1	g2
1			s6		
2	s4		r3		
3		r4	r4		
4	s3			g5	g2
5			r2		
6			acc		

(b) Table d'analyse SLR(1) sans conflit

Table 8.27 – Tables d'analyse LR(0) et SLR(1) de la grammar 8.26

Les numéros de règles associés aux réductions correspondent à l'ordre dans lesquelles elles apparaissent dans la table ci-dessus (1 pour $S \rightarrow E\#...$). On notera que la signification des en-têtes de colonne change légèrement entre les deux représentations: dans la table LR(0), les actions sont effectuées sans regard avant; les labels de colonnes indiquent des transitions; dans la table SLR, les actions "réduire" et "décaler" sont effectuées *après* consultation du regard avant. L'état 6 est un état d'acceptation; par la convention selon laquelle un fichier se termine par une suite infinie de #, cette action est reportée en colonne #: en quelque sorte FOLLOW(S) = {#}. Par conséquent, lire un # dans l'état 1 de la table SLR conduira bien à accepter l'entrée (a).

— $F = \{[A \rightarrow \alpha\bullet, a],$ avec $A \rightarrow \alpha \in P\}$; à chaque état final est associée la production $A \rightarrow \alpha$, correspondant à la réduction à opérer;

— $\forall q = [A \rightarrow \alpha \bullet X\beta, a] \in Q, \delta(q, X) = [A \rightarrow \alpha X \bullet \beta, a]$; comme précédemment, ces transitions signifient la progression de la résolution du but courant par empilement de X;

— $\forall q = [A \rightarrow \alpha \bullet B\beta, a] \in Q$ tq. $B \in N, \forall q' = [B \rightarrow \bullet\gamma, b]$ tq. $b \in$ FIRST$(\beta a), \delta(q, \varepsilon) = q'$.

La condition supplémentaire $b \in$ FIRST(βa) (voir la section 8.1.3) introduit une information de désambiguïsation qui permet de mieux caractériser les réductions à opérer. En particulier, dans le cas LR(1), on espère qu'en prenant en compte la valeur du premier symbole dérivable depuis βa , il sera possible de sélectionner la bonne action à effectuer en cas de conflit sur B.

Pour illustrer ce point, considérons de nouveau la grammar 8.26. L'état initial de l'automate LR(1) correspondant est $[S \rightarrow \bullet E\#]$; lequel a deux transitions ε, correspondant aux deux façons d'empiler un E, atteignant respectivement les états $q_1 = [E \rightarrow \bullet T + E, \#]$ et $q_2 = [E \rightarrow \bullet T, \#]$. Ces deux états se distinguent par leurs transitions ε sortantes: dans le cas de q_1, vers $[T \rightarrow \bullet x, +]$; dans le cas de q_2, vers $[T \rightarrow \bullet x, \#]$. À l'issue de la réduction d'un x en T, le regard avant permet de décider sans ambiguïté de l'action: si c'est un '+', il faut continuer de chercher une réduction $E \rightarrow T + E$; si c'est un '#', il faut réduire selon $E \rightarrow T$.

Construction pas-à-pas de l'automate d'analyse

La construction de la table d'analyse à partir de l'automate se déroule comme pour la table LR(0). Détaillons-en les principales étapes: après construction et déterminisation de l'automate LR(1), on obtient un automate fini dont les états finaux sont associés à des productions de G. On procède alors comme suit:

— pour chaque transition de q vers r étiquetée par un terminal a, la case $T(q, a)$ contient la séquence d'actions (décaler, consommer a en tête de la pile, aller en r);

— pour chaque transition de q vers r étiquetée par un non-terminal A, la case $T(q, A)$ contient la séquence d'actions (consommer A en tête de la pile, aller en r);

— pour chaque état final $q = [A \to \alpha\bullet, a]$, la case $T(q, a)$ contient l'unique action (réduire la pile selon $A \to \alpha$): la décision de réduction (ainsi que la production à appliquer) est maintenant conditionnée par la valeur du regard avant associé à q.

Lorsque $T()$ ne contient pas de conflit, la grammaire est dite *LR(1)*. Il existe des grammaires LR(1) ou "presque[6]" LR(1) pour la majorité des langages informatiques utilisés dans la pratique.

En revanche, lorsque la table de l'analyseur LR(1) contient des conflits, il est de nouveau possible de chercher à augmenter le regard avant pour résoudre les conflits restants. Dans la pratique[7], toutefois, pour éviter la manipulation de tables trop volumineuses, on préférera chercher des moyens ad-hoc de résoudre les conflits dans les tables LR(1) plutôt que d'envisager de construire des tables LR(2) ou plus. Une manière courante de résoudre les conflits consiste à imposer des priorités via des règles du type: "en présence d'un conflit shift/reduce, toujours choisir de décaler[8]"...

En guise d'application, le lecteur est invité à s'attaquer à la construction de la table LR(1) pour la grammar 8.26 et d'en déduire un analyseur déterministe pour cette grammaire. Idem pour la grammar 8.28, qui engendre des mots tels que $x = **x$.

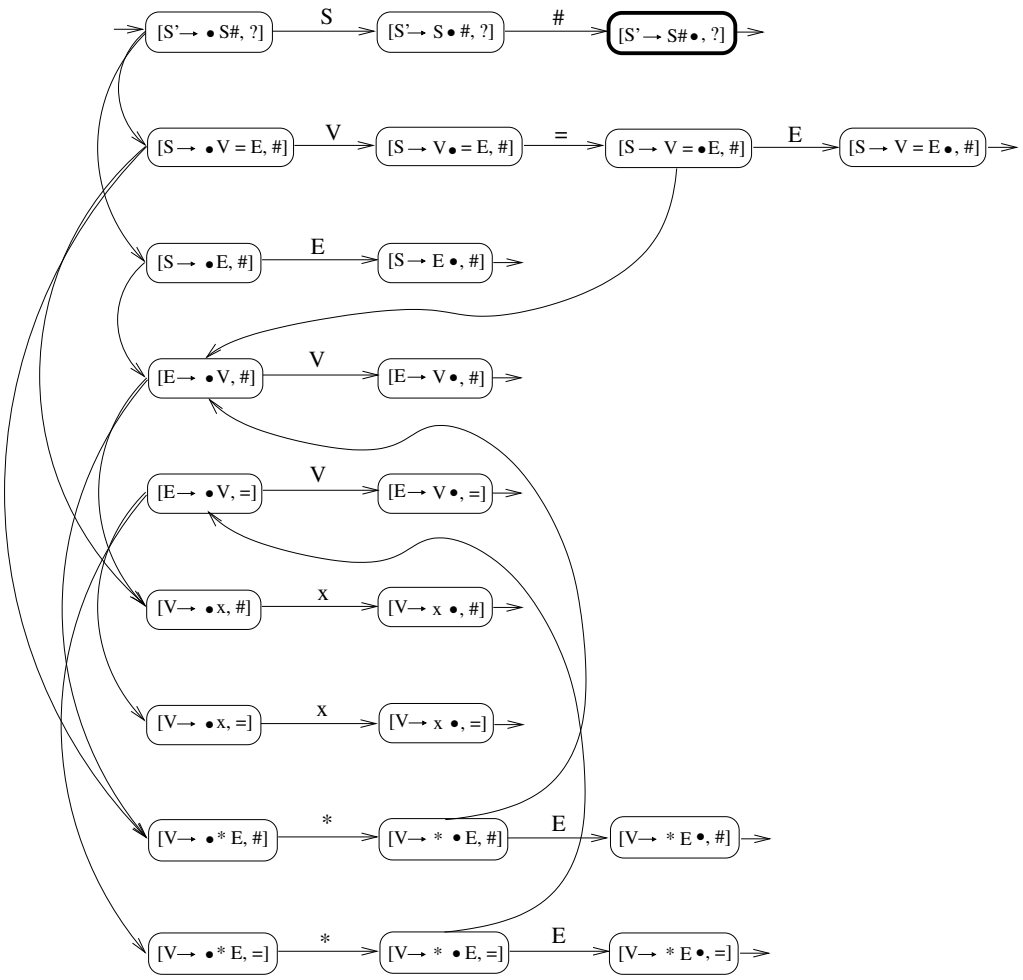
$$
\begin{aligned}
S' &\to S\# \\
S &\to V = E \mid E \\
E &\to V \\
V &\to *E \mid x
\end{aligned}
$$

Grammar 8.28 – Une grammaire pour les manipulations de pointeurs

6. C'est-à-dire que les tables correspondantes sont presque sans conflit.

7. Il existe une autre raison, théorique celle-là, qui justifie qu'on se limite aux grammaires LR(1): les grammaires LR(1) engendrent tous les langages hors-contexte susceptibles d'être analysés par une procédure déterministe ! En d'autres termes, l'augmentation du regard avant peut conduire à des grammaires plus simples à analyser; mais ne change rien à l'expressivité des grammaires. La situation diffère donc ici de ce qu'on observe pour la famille des grammaires LL(k), qui induit une hiérarchie stricte de langages.

8. Ce choix de privilégier le décalage sur la réduction n'est pas innocent : en cas d'imbrication de structures concurrentes, il permet de privilégier la structure la plus intérieure, ce qui correspond bien aux attentes des humains. C'est ainsi que le mot `if cond1 then if cond2 then inst1 else inst2` sera plus naturellement interprété `if cond1 then (if cond2 then inst1 else inst2)` que `if cond1 then (if cond2 then inst1) else inst2` en laissant simplement la priorité au décalage du `else` plutôt qu'à la réduction de `if cond2 then inst1`.



Pour améliorer la lisibilité, les ε ne sont pas représentés. L'état d'acceptation est représenté en gras.

Automaton 8.29 – Les réductions licites pour la grammar 8.28

Pour vous aider dans votre construction, l'automate 8.29 permet de visualiser l'effet de l'ajout du regard avant sur la construction de l'automate d'analyse.

Une construction directe

Construire manuellement la table d'analyse LR est fastidieux, en particulier à cause du passage par un automate non-déterministe, qui implique d'utiliser successivement des procédures de suppression des transitions spontanées, puis de déterminisation. Dans la mesure où la forme de l'automate est très stéréotypée, il est possible d'envisager la construction directe de l'automate déterminisé à partir de la grammaire, en s'aidant des remarques suivantes (cf. la construction directe de l'analyseur LR(0) supra):

— chaque état du déterminisé est un ensemble d'éléments de la forme [production pointée,

terminal]: ceci du fait de l'application de la construction des sous-ensembles (cf. la section 4.1.4)

— la suppression des transitions spontanées induit la notion de *fermeture*: la ε-fermeture d'un état étant l'ensemble des états atteints par une ou plusieurs transitions ε.

Cette procédure de construction du DFA A est détaillée dans l'algorithm 8.30, qui utilise deux fonctions auxiliaires pour effectuer directement la déterminisation.

```
Function LR0 is                                          // Programme principal.
    q_0 = Closure([S' → •S#, ?]) // L'état initial de A.
    Q := {q_0} // Les états de A.
    T := ∅ // Les transitions de A.
    while true do
        Q_i := Q;
        T_i := T;
        foreach q ∈ Q do                          // q est lui-même un ensemble !
            foreach [A → β • Xγ, a] ∈ q do
                r := Successor(q, X);
                Q := Q ∪ {r};
                T := T ∪ {(q, X, r)};
        // Test de stabilisation.
        if Q = Q_i ∧ T = T_i then break ;
// Procédures auxiliaires.
Function Closure(q) is          // Construction directe de la ε-fermeture.
    while true do
        q_i := q;
        foreach [A → β • Xγ, a] ∈ q do
            foreach (X → α) ∈ P do
                foreach b ∈ FIRST(γa) do
                    q := q ∪ [X → •α, b]
        // Test de stabilisation.
        if q = q_i then break ;
    return q
Function Successor(q, X) is                    // Développement des branches.
    r := ∅;
    foreach [A → β • Xγ, a] ∈ q do
        r := r ∪ [A → βX • γ, a]
    return Closure(r)
```

Algorithm 8.30– Construction de l'automate d'analyse LR(1)

En guise d'application, vous vérifierez que la mise en œuvre de l'algorithm 8.30 construit directement le déterminisé de l'automate 8.29.

8.2.4 Compléments

LR et LL La famille des analyseurs *LR(k)* permet d'analyser tous les langages LL(k) et bien d'autres langages non-ambigus. La raison de cette plus grande généricité des analyseurs LR est au fond leur plus grande "prudence": alors qu'un analyseur LL(k) doit pouvoir sélectionner sans erreur une production $A \to \alpha$ sur la seule base des k symboles terminaux non encore appariés (dont tout ou partie peut être dérivé de A), un analyseur LR(k) fonde sa décision d'appliquer une réduction $A \to \alpha$ sur (i) la connaissance de l'intégralité de la partie droite α et (ii) la connaissance des k symboles terminaux à droite de A. Pour k fixé, il est alors normal qu'un analyseur LR(k), ayant plus d'information à sa disposition qu'un analyseur LL(k), fasse des choix plus éclairés, faisant ainsi porter moins de contraintes sur la forme de la grammaire.

Génération d'analyseurs Lorsque l'on s'intéresse à des grammaires réelles, la construction de l'automate et de la table d'analyse LR peut rapidement conduire à de très gros automates: il est en fait nécessaire de déterminiser un automate dont le nombre d'états est proportionnel à la somme des longueurs des parties droites des productions de G; étape qui peut conduire (cf. la section 4.1.4) à un automate déterministe ayant exponentiellement plus d'états que le non-déterministe d'origine. Il devient alors intéressant de recourir à des programmes capables de construire automatiquement un analyseur pour une grammaire LR: il en existe de nombreux, dont le plus fameux, `yacc` est disponible et documenté (dans sa version libre, connue sous le nom de `bison`) à l'adresse suivante: `http://www.gnu.org/software/bison/bison.html`.

LR et LALR et ... Utiliser un générateur d'analyseurs tel que `bison` ne fait que reporter sur la machine la charge de construire (et manipuler) un gros automate; ce qui, en dépit de l'indéniable bonne volonté générale des machines, peut malgré tout poser problème. Le remède le plus connu est d'essayer de compresser *avec perte*, lorsque cela est possible (et c'est le cas général) les tables d'analyse LR(1), donnant lieu à la classe d'analyseurs LALR(1), qui sont ceux que construit `yacc`. Il existe de nombreuses autres variantes des analyseurs LR visant à fournir des solutions pour sauver le déterminisme de l'analyse, tout en maintenant les tables dans des proportions raisonnables.

Notes historiques L'invention de LL est due à Conway (1963). Sa formalisation fut faite par Lewis et Stearns (1968). On doit la découverte de la famille LR à Donald Knuth en 1965 (Knuth, 1965). Il est également à l'origine de l'appellation LL.

Chapter 9

Normalisation des grammaires CF

Dans ce chapitre, nous étudions quelques résultats complémentaires concernant les grammaires hors-contexte, résultats qui garantissent d'une part que les cas de règles potentiellement problématiques pour l'analyse (productions ε, récursions gauches...) identifiées dans les discussions du chapter 7 possèdent des remèdes bien identifiés; d'autre part, qu'il est possible d'imposer *a priori* des contraintes sur la forme de la grammaire, en utilisant des transformations permettant de mettre les productions sous une forme standardisée. On parle, dans ce cas, de grammaires sous *forme normale*. Nous présentons dans ce chapitre les deux formes normales les plus utiles, la forme normale de Chomsky et la forme normale de Greibach.

9.1 Simplification des grammaires CF

Dans cette section, nous nous intéressons tout d'abord aux procédures qui permettent de simplifier les grammaires CF, en particulier pour faire disparaître un certain nombre de configurations potentiellement embarrassantes pour les procédures d'analyse.

9.1.1 Quelques préliminaires

Commençons par deux résultats élémentaires, que nous avons utilisés sans les formaliser à différentes reprises.

Une première notion utile est celle du langage engendré par un non-terminal A. Formellement,

Definition 9.1 (Langage engendré par un non-terminal)**.** *Soit $G = (N, \Sigma, S, P)$ une grammaire CF. On appelle* langage engendré par le non-terminal A*, noté $L_A(G)$, le langage engendré par la grammaire $G' = (N, \Sigma, A, P)$.*

Le langage engendré par un non-terminal est donc le langage obtenu en choisissant ce non-terminal comme axiome. Le langage engendré par la grammaire G est alors simplement le langage $L(G) = L_S(G)$.

Introduisons maintenant la notion de sous-grammaire:

Definition 9.2 (Sous-grammaire)**.** *Soit $G = (N, \Sigma, S, P)$ une grammaire CF. On appelle* sous-grammaire *toute grammaire $G' = (N, \Sigma, S, P')$, avec $P' \subset P$. Si G' est une sous-grammaire de G, alors $L(G') \subset L(G)$.*

Une sous-grammaire de G n'utilise qu'un sous-ensemble des productions de G; le langage engendré est donc un sous-ensemble du langage engendré par G.

Le procédé suivant nous fournit un premier moyen pour construire systématiquement des grammaires équivalentes à une grammaire G. De manière informelle, ce procédé consiste à court-circuiter des étapes de dérivation en les remplaçant par une dérivation en une étape. Cette opération peut être effectuée sans changer le langage engendré, comme l'énonce le résultat suivant.

Lemma 9.3 ("Court-circuitage" des dérivations)**.** *Soit $G = (N, \Sigma, S, P)$ une grammaire CF; soient A un non-terminal et α une séquence de terminaux et non-terminaux tels que $A \overset{\star}{\underset{G}{\Rightarrow}} \alpha$. Alors $G' = (N, \Sigma, S, P \cup \{A \to \alpha\})$ est faiblement équivalente à G.*

Proof. Le lemme précédent concernant les sous-grammaires nous assure que $L(G) \subset L(G')$. Inversement, soit u dans $L(G')$: si sa dérivation n'utilise pas la production $A \to \alpha$, alors la même dérivation existe dans G; sinon si sa dérivation utilise $A \to \alpha$, alors il existe une dérivation dans G utilisant $A \overset{\star}{\underset{G}{\Rightarrow}} \alpha$. □

Nous terminons cette section par un second procédé permettant de construire des grammaires équivalentes, qui utilise en quelque sorte la distributivité des productions.

Lemma 9.4 ("Distributivité" des dérivations)**.** *Soit G une CFG, $A \to \alpha_1 B \alpha_2$ une A-production de G et $\{B \to \beta_1, \ldots, B \to \beta_n\}$ l'ensemble des B-productions. Alors, la grammaire G' dérivée de G en supprimant $A \to \alpha_1 B \alpha_2$ et en ajoutant aux productions de G l'ensemble $\{A \to \alpha_1\beta_1\alpha_2, \ldots, A \to \alpha_1\beta_n\alpha_2\}$ est faiblement équivalente à G.*

La démonstration de ce second résultat est laissée en exercice.

9.1.2 Non-terminaux inutiles

La seule contrainte définitoire posée sur les productions des CFG est que leur partie gauche soit réduite à un non-terminal. Toutefois, un certain nombre de configurations posent des problèmes pratiques, notamment lorsqu'il s'agit de mettre en œuvre des algorithmes d'analyse. Nous nous intéressons, dans cette section, aux configurations qui, sans introduire de difficultés majeures, sont source d'inefficacité.

Une première source d'inefficacité provient de l'existence de non-terminaux n'apparaissant que dans des parties droites de règles. En fait, ces non-terminaux ne dérivent aucun mot et peuvent être supprimés, ainsi que les règles qui y font référence, sans altérer le langage engendré par la grammaire.

Une configuration voisine est fournie par des non-terminaux (différents de l'axiome) qui n'apparaîtraient dans aucune partie droite. Ces non-terminaux ne pouvant être dérivés de l'axiome, ils sont également inutiles et peuvent être supprimés.

Enfin, les non-terminaux improductifs sont ceux qui, bien que dérivables depuis l'axiome, n'apparaissent dans aucune dérivation réussie, parce qu'ils sont impossibles à éliminer. Pensez, par exemple, à un non-terminal X qui n'apparaîtrait (en partie gauche) que dans une seule règle de la forme $X \to aX$.

Formalisons maintenant ces notions pour construire un algorithme permettant de se débarrasser des non-terminaux et des productions inutiles.

Definition 9.5 (Utilité d'une production et d'un non-terminal)**.** *Soit G une CFG, on dit qu'une production $P = A \to \alpha$ de G est* utile *si et seulement s'il existe un mot $w \in \Sigma^*$ tel que $S \overset{\star}{\underset{G}{\Rightarrow}} xAy \overset{P}{\underset{G}{\Rightarrow}} x\alpha y \overset{\star}{\underset{G}{\Rightarrow}} w$. Sinon, on dit que P est* inutile*.*

*De même, on qualifie d'*utiles *les non-terminaux qui figurent en partie gauche des règles utiles.*

L'identification des productions et non-terminaux utiles se fait en appliquant les deux procédures suivantes :

— La première étape consiste à étudier successivement toutes les grammaires $G_A = (N, \Sigma, A, P)$ pour $A \in N$. Le langage $L(G_A)$ contient donc l'ensemble des mots qui se dérivent depuis le non-terminal A. Il existe un algorithme (cf. la section 6.3) permettant de déterminer si $L(G_A)$ est vide. On construit alors G' en supprimant de G tous les non-terminaux A pour lesquels $L(G_A) = \emptyset$, ainsi que les productions dans lesquels ils apparaissent [1]. La grammaire G' ainsi construite est *fortement* équivalente à G. En effet:

  — $L(G') \subset L(G)$, par le simple fait que G' est une sous-grammaire de G; de plus, les dérivations gauches de G' sont identiques à celles de G.

  — $L(G) \subset L(G')$: s'il existe un mot $u \in \Sigma^\star$ tel que $S \overset{\star}{\underset{G}{\Rightarrow}} u$ mais pas $S \overset{\star}{\underset{G'}{\Rightarrow}} u$, alors nécessairement la dérivation de u contient un des non-terminaux éliminés de G. Ce non-terminal dérive un des facteurs de u, donc engendre un langage non-vide, ce qui contredit l'hypothèse.

  Cette procédure n'est toutefois pas suffisante pour garantir qu'un non-terminal est utile: il faut, de plus, vérifier qu'il peut être dérivé depuis S. C'est l'objet de la seconde phase de l'algorithme.

— Dans une seconde étape, on construit récursivement les ensembles N_U et P_U contenant respectivement des non-terminaux et des productions. Ces ensembles contiennent initialement respectivement S et toutes les productions de partie gauche S. Si, à une étape donnée de la récursion, N_U contient A, alors on ajoute à P_U toutes les règles dont A est partie gauche, et à N_U tous les non-terminaux figurant dans les parties droites de ces règles. Cette procédure s'arrête après un nombre fini d'itérations, quand plus

1. La procédure esquissée ici est mathématiquement suffisante, mais algorithmiquement naïve. Une implémentation plus efficace consisterait à déterminer de proche en proche l'ensemble des non-terminaux engendrant un langage non-vide, par une procédure similaire à celle décrite plus loin. L'écriture d'un tel algorithme est laissée en exercice.

aucun non-terminal ne peut être ajouté à N_U. Par construction, pour toute production $A \to \alpha$ de P_U, il existe α_1 et α_2 tels que $S \overset{\star}{\underset{G}{\Rightarrow}} \alpha_1 A \alpha_2 \underset{G}{\Rightarrow} \alpha_1 \alpha \alpha_2$.

En supprimant de G tous les terminaux qui ne sont pas dans N_U, ainsi que les productions P_U correspondantes, on construit, à partir de G, une nouvelle sous-grammaire G', qui par construction ne contient que des terminaux et des productions utiles. Par un argument similaire au précédent, on vérifie que G' est fortement équivalente à G.

Attention: ces deux procédures doivent être appliquées dans un ordre précis. En particulier, il faut commencer par supprimer les variables ne générant aucun mot, puis éliminer celles qui n'apparaissent dans aucune dérivation. Vous pourrez vous en convaincre en examinant la grammar 9.1.

$$\begin{aligned} S &\to a \mid AB \\ A &\to b \end{aligned}$$

Grammar 9.1 – Élimination des productions inutiles : l'ordre importe

9.1.3 Cycles et productions non-génératives

Les cycles correspondent à des configurations mettant en jeu des productions "improductives" de la forme $A \to B$. Ces productions, que l'on appelle *non-génératives*, effectuent un simple renommage de variable, sans réellement entraîner la génération (immédiate ou indirecte) de symboles terminaux.

Ces productions sont potentiellement nuisibles pour les algorithmes de génération ou encore d'analyse ascendante, qui peuvent être conduits dans des boucles sans fin. Ceci est évident dans le cas de productions de type $A \to A$, mais apparaît également lorsque l'on a des cycles de productions non-génératives comme dans: $A \to B$, $B \to C$, $C \to A$. Ces cycles doivent donc faire l'objet d'un traitement particulier. Fort heureusement, pour chaque mot dont la dérivation contient un cycle, il existe également une dérivation sans cycle, suggérant qu'il est possible de se débarrasser des cycles sans changer le langage reconnu. C'est précisément ce qu'affirme le théorème suivant :

Theorem 9.6 (Élimination des cycles)**.** *Soit G une CFG. On peut contruire une CFG G', faiblement équivalente à G, qui ne contient aucune production de la forme $A \to B$, où A et B sont des non-terminaux.*

Avant de rentrer dans les détails techniques, donnons l'intuition de la construction qui va être développée dans la suite: pour se débarrasser d'une règle $A \to B$ sans perdre de dérivation, il "suffit" de rajouter à la grammaire une règle $A \to \beta$ pour chaque règle $B \to \beta$: ceci permet effectivement bien de court-circuiter la production non-générative. Reste un problème à résoudre: que faire des productions $B \to C$? En d'autres termes, comment faire pour s'assurer qu'en se débarrassant d'une production non-générative, on n'en a pas ajouté une autre ? L'idée est de construire en quelque sorte la clôture transitive de ces productions non-génératives, afin de détecter (et de supprimer) les cycles impliquant de telles productions.

Definition 9.7 (Dérivation non-générative)**.** *$B \in N$ dérive immédiatement non-génèrativement de A dans G si et seulement si $A \to B$ est une production de G. On notera $A \underset{G}{\mapsto} B$.*

$S \to A$	$A \to B$	$B \to bb$	$C \to B$
$S \to B$	$A \to B$	$B \to C$	$C \to Aa$
	$A \to aB$		$C \to aAa$

Grammar 9.2 – Une grammaire contenant des productions non-génératives

B dérive non-génèrativement de A dans G, noté $A \overset{\star}{\underset{G}{\mapsto}} B$ *si et seulement si* $\exists X_1 \dots X_n$ *dans N tels que* $A \underset{G}{\mapsto} X_1 \underset{G}{\mapsto} \cdots \underset{G}{\mapsto} X_n \underset{G}{\mapsto} B$.

Proof du theorem 9.6. Un algorithme de parcours du graphe de la relation $\underset{G}{\mapsto}$ permet de déterminer $C_A = \{A\} \cup \{X \in N, A \overset{\star}{\underset{G}{\mapsto}} X\}$ de proche en proche pour chaque non-terminal A.

Construisons alors G' selon:

— G' a le même axiome, les mêmes terminaux et non-terminaux que G;

— $A \to \alpha$ est une production de G' si et seulement s'il existe dans G une production $X \to \alpha$, avec $X \in C_A$ et $\alpha \notin N$. Cette condition assure en particulier que G' est bien sans production non-générative.

En d'autres termes, on remplace les productions $X \to \alpha$ de G en court-circuitant (de toutes les manières possibles) X.

Montrons alors que G' est bien équivalente à G. Soit en effet D une dérivation gauche minimale dans G, contenant une séquence (nécessairement sans cycle) maximale de productions non-génératives de $X_1, \dots, X_k$ suivie d'une production générative $X_k \to \alpha$. Cette séquence peut être remplacée par $X_1 \to \alpha$, qui par construction existe dans G'. Inversement, toute dérivation dans G' ou bien n'inclut que des productions de G, ou bien inclut au moins une production $A \to \alpha$ qui n'est pas dans G. Mais alors il existe dans G une séquence de règles non-génératives $A \to \cdots \to X \to \alpha$ et la dérivation D existe également dans G. On notera que contrairement à l'algorithme d'élimination des variables inutiles, cette transformation a pour effet de modifier (en fait d'aplatir) les arbres de dérivation: G et G' ne sont que faiblement équivalentes. □

Pour illustrer le fonctionnement de cet algorithme, considérons la grammar 9.2. Le calcul de la clôture transitive de $\underset{G}{\mapsto}$ conduit aux ensembles suivants:

$$\begin{aligned} C_S &= \{S, A, B, C\} \\ C_A &= \{A, B, C\} \\ C_B &= \{B, C\} \\ C_C &= \{B, C\} \end{aligned}$$

La grammar 9.3, G', contient alors les productions qui correspondent aux quatre seules productions génératives: $a \to aB$, $B \to bb$, $C \to aAa$, $C \to Aa$.

$S \to aB$	$B \to bb$	$A \to aB$	$C \to aAa$
$S \to bb$	$B \to aAa$	$A \to bb$	$C \to Aa$
$S \to aAa$	$B \to Aa$	$A \to aAa$	$C \to bb$
$S \to Aa$		$A \to Aa$	

Grammar 9.3 – Une grammaire débarrassée de ses productions non-génératives

9.1.4 Productions ε

Si l'on accepte la version libérale de la définition des grammaires CF (cf. la discussion de la section 5.2.6), un cas particulier de règle licite correspond au cas où la partie droite d'une production est vide: $A \to \varepsilon$. Ceci n'est pas gênant en génération et signifie simplement que le non-terminal introduisant ε peut être supprimé de la dérivation. Pour les algorithmes d'analyse, en revanche, ces productions particulières peuvent singulièrement compliquer le travail, puisque l'analyseur devra à tout moment examiner la possibilité d'insérer un non-terminal. Il peut donc être préférable de chercher à se débarrasser de ces productions avant d'envisager d'opérer une analyse: le résultat suivant nous dit qu'il est possible d'opérer une telle transformation et nous montre comment la mettre en œuvre.

Theorem 9.8 (Suppression des productions ε)**.** *Si L est un langage engendré par G, une grammaire CF telle que toute production de G est de la forme: $A \to \alpha$, avec α éventuellement vide, alors L peut être engendré par une grammaire $G' = (N \cup \{S'\}, \Sigma, S', P')$, dont les productions sont soit de la forme $A \to \alpha$, avec α non-vide, soit $S' \to \varepsilon$ et S' n'apparaît dans la partie droite d'aucune règle.*

Ce résultat dit deux choses : d'une part que l'on peut éliminer toutes les productions ε sauf peut-être une (si $\varepsilon \in L(G)$), dont la partie gauche est alors l'axiome; d'autre part que l'axiome lui-même peut être rendu non-récursif (i.e. ne figurer dans aucune partie droite). L'intuition du premier de ces deux résultats s'exprime comme suit: si S dérive ε, ce ne peut être qu'au terme d'un enchaînement de productions n'impliquant que des terminaux qui engendrent ε. En propageant de manière ascendante la propriété de dériver ε, on se ramène à une grammaire équivalente dans laquelle seul l'axiome dérive (directement) ε.

Proof. Commençons par le second résultat en considérant le cas d'une grammaire G admettant un axiome récursif S. Pour obtenir une grammaire G' (faiblement) équivalente, il suffit d'introduire un nouveau non-terminal S', qui sera le nouvel axiome de G' et une nouvelle production: $S' \to S$. Cette transformation n'affecte pas le langage engendré par la grammaire G.

Supposons alors, sans perte de généralité, que l'axiome de G est non-récursif et intéressons-nous à l'ensemble N_ε des variables A de G telles que le langage engendré par A, $L_A(G)$, contient ε. Quelles sont-elles ? Un cas évident correspond aux productions $A \to \varepsilon$. Mais ε peut également se déduire depuis A par plusieurs règles, $A \to \alpha \overset{\star}{\underset{G}{\Rightarrow}} \varepsilon$, à condition que tous les symboles de α soient eux-mêmes dans N_ε. On reconnaît sous cette formulation le problème du calcul de l'ensemble NULL que nous avons déjà étudié lors de la présentation des analyseurs LL (voir en particulier la section 8.1.3).

Une fois N_ε calculé, la transformation suivante de G conduit à une grammaire G' faiblement équivalente: G' contient les mêmes non-terminaux, terminaux et axiome que G. De surcroît,

Productions de G	Productions de G'
$S \to ASB \mid c$	$S \to ASB \mid AS \mid SB \mid S \mid c$
$A \to aA \mid B$	$A \to aA \mid a \mid B$
$B \to b \mid \varepsilon$	$B \to b$

Grammar 9.4 – Une grammaire avant et après élimination des productions ε

G' contient toutes les productions de G n'impliquant aucune variable de N_ε. Si maintenant G contient une production $A \to \alpha$ et α inclut des éléments de N_ε, alors G' contient *toutes* les productions de type $A \to \beta$, où β s'obtient depuis α en supprimant une ou plusieurs variables de N_ε. Finalement, si S est dans N_ε, alors G' contient $S \to \varepsilon$. G' ainsi construite est équivalente à G: notons que toute dérivation de G qui n'inclut aucun symbole de N_ε se déroule à l'identique dans G'. Soit maintenant une dérivation impliquant un symbole de N_ε: soit il s'agit de $S \overset{\star}{\underset{G}{\Rightarrow}} \varepsilon$ et la production $S' \to \varepsilon$ permet une dérivation équivalente dans G'; soit il s'agit d'une dérivation $S \overset{\star}{\underset{G}{\Rightarrow}} \alpha \underset{G}{\Rightarrow} \beta \overset{\star}{\underset{G}{\Rightarrow}} u$, avec $u \neq \varepsilon$ et β contient au moins un symbole X de N_ε, mais pas α. Mais pour chaque X de β, soit X engendre un facteur vide de u, et il existe une production de G' qui se dispense d'introduire ce non-terminal dans l'étape $\alpha \underset{G}{\Rightarrow} \beta$; soit au contraire X n'engendre pas un facteur vide et la même dérivation existe dans G'. On conclut donc que $L(G) = L(G')$. □

Cette procédure est illustrée par la grammar 9.4: G contenant deux terminaux qui dérivent le mot vide ($N_\varepsilon = \{A, B\}$), les productions de G' se déduisent de celles de G en considérant toutes les manières possibles d'éviter d'avoir à utiliser ces terminaux.

9.1.5 Élimination des récursions gauches directes

On appelle *directement récursifs* les non-terminaux A d'une grammaire G qui sont tels que $A \overset{\star}{\Rightarrow} A\alpha$ (récursion gauche) ou $A \overset{\star}{\Rightarrow} \alpha A$ (récursion droite). Les productions impliquant des récursions gauches directes posent des problèmes aux analyseurs descendants, qui peuvent être entraînés dans des boucles sans fin (cf. les sections 7.3 and 8.1). Pour utiliser de tels analyseurs, il importe donc de savoir se débarrasser de telles productions. Nous nous attaquons ici aux récursions gauches directes; les récursions gauches indirectes seront traitées plus loin (à la section 9.2.2).

Il existe un procédé mécanique permettant d'éliminer ces productions, tout en préservant le langage reconnu. L'intuition de ce procédé est la suivante: de manière générique, un terminal récursif à gauche est impliqué dans la partie gauche de deux types de production: celles qui sont effectivement récursives à gauche et qui sont de la forme:

$$A \to A\alpha_1 \mid A\alpha_2 \ldots \mid A\alpha_n$$

et celles qui permettent d'éliminer ce non-terminal, et qui sont de la forme:

$$A \to \beta_1 \mid \beta_2 \ldots \mid \beta_m$$

où le premier symbole x_i de β_i n'est pas un A.

L'effet net de l'utilisation de ces productions conduit donc à des dérivations gauches de A dans lesquelles on "accumule" à droite de A un nombre arbitraire de α_i; l'élimination de A introduisant en tête du proto-mot le symbole (terminal ou non-terminal) x_j. Formellement:

$$A \underset{G}{\Rightarrow} A\alpha_{i_1} \underset{G}{\Rightarrow} A\alpha_{i_2}\alpha_{i_1} \ldots \overset{\star}{\underset{G}{\Rightarrow}} A\alpha_{i_n} \ldots \alpha_{i_1}$$

L'élimination de A par la production $A \to \beta_j$ conduit à un proto-mot

$$\beta_j \alpha_{i_n} \ldots \alpha_{i_1}$$

dont le symbole initial est donc β_j. Le principe de la transformation consiste à produire β_j sans délai et à simultanément transformer la récursion gauche (qui "accumule" simplement les α_i) en une récursion droite. Le premier de ces buts est servi par l'introduction d'un nouveau non-terminal R dans des productions de la forme:

$$A \to \beta_1 \mid \beta_2 \ldots \mid \beta_m \mid \beta_1 R \ldots \mid \beta_m R$$

La récursivité du terminal A est préservée par la nouvelle série de productions:

$$R \to \alpha_1 R \mid \alpha_2 R \ldots \alpha_n R \mid \alpha_1 \mid \alpha_2 \ldots \mid \alpha_n$$

On vérifie, par les techniques habituelles de preuve (e.g. par induction sur la longueur des dérivations) que cette transformation produit bien une grammaire (faiblement) équivalente à la grammaire de départ.

En guise d'illustration, reprenons la grammar 8.5 des expressions arithmétiques, entrevue lors de notre présentation des analyseurs LL(1). La grammar 9.5, qui rappelle cette grammaire [2], contient deux terminaux directement récursifs à gauche.

$$\begin{aligned} S &\to S - F \mid F \\ F &\to F/T \mid T \\ T &\to (S) \mid D \\ D &\to 0 \mid \ldots \mid 9 \mid 0D \mid \ldots \mid 9D \end{aligned}$$

Grammar 9.5 – Une grammaire pour les expressions arithmétiques

Le premier non-terminal à traiter est S, qui contient une règle directement récursive et une règle qui ne l'est pas. On introduit donc le nouveau symbole S', ainsi que les deux productions: $S \to FS' \mid F$ et $S' \to -FS' \mid -F$. Le traitement du symbole F se déroule de manière exactement analogue.

9.2 Formes normales

La notion de *forme normale* d'une grammaire répond à la nécessité, pour un certain nombre d'algorithmes de parsage, de disposer d'une connaissance *a priori* sur la forme des productions de la grammaire. Cette connaissance est exploitée pour simplifier la programmation

2. À une différence près, et de taille: nous utilisons ici les opérateurs $-$ et $/$, qui sont ceux qui sont associatifs par la gauche et qui justifient l'écriture d'une grammaire avec une récursion gauche. Si l'on avait que $+$ et $*$, il suffirait d'écrire les règles avec une récursion droite et le tour serait joué !

d'un algorithme de parsage, ou encore pour accélérer l'analyse. Les principales formes normales (de Chomsky et de Greibach) sont décrites dans les sections qui suivent. On verra que les algorithmes de mise sous forme normale construisent des grammaires faiblement équivalentes à la grammaire d'origine: les arbres de dérivation de la grammaire normalisée devront donc être transformés pour reconstruire les dérivations (et les interprétations) de la grammaire originale.

9.2.1 Forme normale de Chomsky

Theorem 9.9 (Forme normale de Chomsky)**.** *Toute grammaire hors-contexte admet une grammaire faiblement équivalente dans laquelle toutes les productions sont soit de la forme $A \to BC$, soit de la forme $A \to a$, avec A, B, C des non-terminaux et a un terminal. Si, de surcroît, $S \overset{\star}{\underset{G}{\Rightarrow}} \varepsilon$, alors la forme normale contient également $S \to \varepsilon$. Cette forme est appelée* forme normale de Chomsky*, abrégée en CNF conformément à la terminologie anglaise (Chomsky Normal Form).*

Proof. Les résultats de simplification obtenus à la section 9.1 nous permettent de faire en toute généralité l'hypothèse que G ne contient pas de production ε autre que éventuellement $S \to \varepsilon$ et que si $A \to X$ est une production de G, alors X est nécessairement terminal (il n'y a plus de production non-générative).

La réduction des autres productions procède en deux étapes : elle généralise tout d'abord l'introduction des terminaux par des règles ne contenant que ce seul élément en partie droite; puis elle ramène toutes les autres productions à la forme normale correspondant à deux non-terminaux en partie droite.

Première étape, construisons $G' = (N', \Sigma, S, P')$ selon:

- N' contient tous les symboles de N;
- pour tout symbole a de Σ, on ajoute une nouvelle variable A_a et une nouvelle production $A_a \to a$.
- toute production de P de la forme $A \to a$ est copiée dans P'
- toute production $A \to X_1 \cdots X_k$, avec tous les X_i dans N est copiée dans P';
- soit $A \to X_1 \cdots X_m$ contenant au moins un terminal: cette production est transformée en remplaçant chaque occurrence d'un terminal a par la variable A_a correspondante.

Par des simples raisonnements inductifs sur la longueur des dérivations, on montre que pour toute variable de G, $A \overset{\star}{\underset{G}{\Rightarrow}} u$ si et seulement si $A \overset{\star}{\underset{G'}{\Rightarrow}} u$, puis l'égalité des langages engendrés par ces deux grammaires. En effet, si $A \to u = u_1 \cdots u_l$ est une production de G, on aura dans G':

$$A \underset{G'}{\Rightarrow} A_{u_1} \cdots A_{u_l} \underset{G'}{\Rightarrow} u_1 A_{u_2} \cdots A_{u_l} \cdots \overset{\star}{\underset{G'}{\Rightarrow}} u$$

Supposons que cette propriété soit vraie pour toutes les dérivations de longueur n et soit A et u tels que $A \overset{\star}{\Rightarrow} u$ en $n+1$ étapes. La première production est de la forme: $A \to x_1 A_1 x_2 A_2 \cdots A_k$, où chaque x_i ne contient que des terminaux. Par hypothèse de récurrence, les portions de u engendrées dans G par les variables A_i se dérivent également dans G'; par construction

chaque x_i se dérive dans G' en utilisant les nouvelles variables A_a, donc $A \overset{\star}{\underset{G'}{\Rightarrow}} u$. La réciproque se montre de manière similaire.

Seconde phase de la procédure: les seules productions de G' dans lesquelles apparaissent des terminaux sont du type recherché $A \to a$; il s'agit maintenant de transformer G' en G'' de manière que toutes les productions qui contiennent des non-terminaux en partie droite en contiennent exactement 2. Pour aboutir à ce résultat, il suffit de changer toutes les productions de type $A \to B_1 ... B_m, m \geq 3$ dans G' par l'ensemble des productions suivantes (les non-terminaux D_i sont créés pour l'occasion) : $\{A \to B_1\ D_1, D_1 \to B_2\ D_2, \ldots\ , D_{m-2} \to B_{m-1}\ B_m\}$. De nouveau, il est relativement direct de montrer que le langage engendré par G'' est le même que celui engendré par G. □

Les arbres de dérivation des grammaires CNF ont une forme extrêmement caractéristique, puisque chaque nœud interne de l'arbre a soit un fils unique et ce fils est une feuille portant un symbole terminal; soit exactement deux fils et ces deux fils sont des variables.

Comme exemple d'application, entreprenez la mise sous forme normale de Chomsky de la grammar 9.6. Le résultat auquel vous devez aboutir est la grammar 9.7.

$$\begin{array}{lll} S \to bA & A \to a & B \to b \\ S \to aB & A \to aS & B \to bS \\ & A \to bAA & B \to aBB \end{array}$$

Grammar 9.6 – Une grammaire à Chomsky-normaliser

$$\begin{array}{llll} S \to A_bA & A \to a & B \to b & A_a \to a \\ S \to A_aB & A \to A_aS & B \to A_bS & A_b \to b \\ & A \to A_bD_1 & B \to A_aD_2 & D_1 \to AA \\ & & & D_2 \to BB \end{array}$$

Grammar 9.7 – Une grammaire Chomsky-normalisée

En plus de son importance théorique, cette forme normale présente de nombreux avantages:

— un premier avantage est que les terminaux sont introduits par des productions dédiées, de type $A \to a$ (au moins une par terminal). Ceci s'avère particulièrement bienvenu pour les algorithmes de parsage descendant.

— La mise sous CNF facilite également les étapes de réduction des analyseurs ascendants, puisqu'il suffit simplement de regarder les couples de non-terminaux successifs pour juger si une telle opération est possible.

En revanche, cette transformation conduit en général à une augmentation sensible du nombre de productions dans la grammaire. Cette augmentation joue dans le sens d'une pénalisation générale des performances des analyseurs.

9.2.2 Forme normale de Greibach

La transformation d'une grammaire en une grammaire sous forme normale de Greibach généralise en un sens le procédé d'élimination des récursions gauches directes (cf. la sec-

tion 9.1.5), dans la mesure où elle impose une contrainte qui garantit que chaque production augmente de manière strictement monotone le préfixe terminal du proto-mot en cours de dérivation. À la différence de la forme normale de Chomsky, l'existence de la forme normale de Greibach est plus utilisée pour démontrer divers résultats théoriques que pour fonder des algorithmes d'analyse.

La forme normale de Greibach est définie dans le théorème suivant:

Theorem 9.10 (Forme normale de Greibach)**.** *Tout langage hors-contexte L ne contenant pas ε peut être engendré par une grammaire dont toutes les productions sont de la forme $A \to a\alpha$, avec a un terminal et α est une séquence de non-terminaux (éventuellement vide). Cette grammaire est appelée* forme normale de Greibach*, en abrégé GNF, conformément à la terminologie anglaise.*

Si L contient ε, alors ce résultat reste valide, en ajoutant toutefois une règle $S \to \varepsilon$, qui dérive ε depuis l'axiome, comme démontré à la section 9.1.4.

Proof. La preuve de l'existence d'une GNF repose, comme précédemment, sur un algorithme permettant de construire explicitement la forme normale de Greibach à partir d'une grammaire CF quelconque. Cet algorithme utilise deux procédés déjà présentés, qui transforment une grammaire CF en une grammaire CF faiblement équivalente.

Le premier procédé est celui décrit au lemma 9.3 et consiste à ajouter une production $A \to \alpha$ à G si l'on observe dans G la dérivation $A \overset{\star}{\underset{G}{\Rightarrow}} \alpha$. Le second procédé est décrit dans la section 9.1.5 et consiste à transformer une récursion gauche en récursion droite par ajout d'un nouveau non-terminal.

L'algorithme de construction repose sur une numérotation arbitraire des variables de la grammaire $N = \{A_0, \ldots, A_n\}$, l'axiome recevant conventionnellement le numéro 0. On montre alors que:

Lemma 9.11. *Soit G une grammaire CF. Il existe une grammaire équivalente G' dont toutes les productions sont de la forme:*

- $A_i \to a\alpha$, *avec* $a \in \Sigma$
- $A_i \to A_j\beta$, *et A_j est classé* strictement après *A_i dans l'ordonnancement des non-terminaux ($j > i$).*

La procédure de construction de G' est itérative et traite les terminaux dans l'ordre dans lequel ils sont ordonnés. On note, pour débuter, que si l'on a pris soin de se ramener à une grammaire dont l'axiome est non récursif, ce que l'on sait toujours faire (cf. le résultat de la section 9.1.4), alors toutes les productions dont $S = A_0$ est partie gauche satisfont par avance la propriété du lemma 9.11. Supposons que l'on a déjà traité les non-terminaux numérotés de 0 à $i-1$, et considérons le non-terminal A_i. Soit $A_i \to \alpha$ une production dont A_i est partie gauche, telle que α débute par une variable A_j. Trois cas de figure sont possibles:

- (a) $j < i$: A_j est classé avant A_i;
- (b) $j = i$: A_j est égale à A_i;
- (c) $j > i$: A_j est classé après A_i;

Pour les productions de type (a), on applique itérativement le résultat du lemma 9.4 en traitant les symboles selon leur classement: chaque occurrence d'un terminal $A_j, j < i$ est remplacée par l'expansion de toutes les parties droites correspondantes. Par l'hypothèse de récurrence, tous les non-terminaux ainsi introduits ont nécessairement un indice strictement supérieur à i. Ceci implique qu'à l'issue de cette phase, toutes les productions déduites de $A_i \rightarrow \alpha$ sont telles que leur partie droite ne contient que des variables dont l'indice est au moins i. Toute production dont le coin gauche n'est pas A_i satisfait par construction la propriété du lemma 9.11: laissons-les en l'état. Les productions dont le coin gauche est A_i sont de la forme $A_i \rightarrow A_i\beta$, sont donc directement récursives à gauche. Il est possible de transformer les A_i- productions selon le procédé de la section 9.1.5, sans introduire en coin gauche de variable précédant A_i dans le classement. En revanche, cette procédure conduit à introduire de nouveaux non-terminaux, qui sont conventionnellement numérotés depuis $n+1$ dans l'ordre dans lequel ils sont introduits. Ces nouveaux terminaux devront subir le même traitement que les autres, traitement qui est toutefois simplifié par la forme des règles (récursions droites) dans lesquels ils apparaissent. Au terme de cette transformation, tous les non-terminaux d'indice supérieur à $i+1$ satisfont la propriété du lemma 9.11, permettant à la récurrence de se poursuivre.

Notons qu'à l'issue de cette phase de l'algorithme, c'est-à-dire lorsque le terminal d'indice maximal $n+p$ a été traité, G' est débarrassée de toutes les chaînes de récursions gauches: on peut en particulier envisager de mettre en œuvre des analyses descendantes de type LL et s'atteler à la construction de la table d'analyse prédictive correspondante (voir la section 8.1).

Cette première partie de la construction est illustrée par le table 9.8.

Considérons maintenant ce qu'il advient après que l'on achève de traiter le dernier non-terminal A_{n+p}. Comme les autres, il satisfait la propriété que le coin gauche de toutes les A_{n+p}-productions est soit un terminal, soit un non-terminal d'indice strictement plus grand. Comme ce second cas de figure n'est pas possible, c'est donc que toutes les A_{n+p}-productions sont de la forme: $A_{n+p} \rightarrow a\alpha$, avec $a \in \Sigma$, qui est la forme recherchée pour la GNF. Considérons alors les A_{n+p-1} productions: soit elles ont un coin gauche terminal, et sont déjà conformes à la GNF; soit elles ont A_{n+p} comme coin gauche. Dans ce second cas, en utilisant de nouveau le procédé du lemma 9.4, il est possible de remplacer A_{n+p} par les parties droites des A_{n+p}-productions et faire ainsi émerger un symbole terminal en coin gauche. En itérant ce processus pour $i = n+p$ à $i = 0$, on aboutit, de proche en proche, à une grammaire équivalente à celle de départ et qui, de plus, est telle que chaque partie droite de production débute par un terminal. Il reste encore à éliminer les terminaux qui apparaîtraient dans les queues de partie droite (cf. la mise sous CNF) pour obtenir une grammaire qui respecte les contraintes énoncées ci-dessus. □

Pour compléter cette section, notons qu'il existe des variantes plus contraintes de la GNF, qui constituent pourtant également des formes normales. Il est en particulier possible d'imposer, dans la définition du theorem 9.10, que toutes les parties droites de production contiennent au plus deux variables: on parle alors de forme normale de Greibach *quadratique* (Salomaa, 1973, p.202).

État initial	$S \to A_2A_2$ $A_1 \to A_1A_1 \mid a$ $A_2 \to A_1A_1 \mid A_2A_1 \mid b$	
Traitement de A_1	$S \to A_2A_2$ $A_1 \to a \mid aR_1$ $A_2 \to A_1A_1 \mid A_2A_1 \mid b$ $R_1 \to A_1 \mid A_1R_1$	 A_3 est nouveau récursion droite
Traitement de A_2 (première étape)	$S \to A_2A_2$ $A_1 \to aA_1 \mid aA_1R_1$ $A_2 \to aA_1A_1 \mid aA_1R_1A_1 \mid A_2A_1 \mid b$ $R_1 \to A_1 \mid A_1R_1$	
Traitement de A_2 (seconde étape)	$S \to A_2A_2$ $A_1 \to aA_1 \mid aA_1A_3$ $A_2 \to aA_1A_1 \mid aA_1A_3A_1 \mid b \mid aA_1A_1A_4 \mid aA_1A_3A_1A_4 \mid bA_4$ $A_3 \to A_1 \mid A_1A_3$ $A_4 \to A_1 \mid A_1A_4$	
Traitement de A_3	$S \to A_2A_2$ $A_1 \to aA_1 \mid aA_1A_3$ $A_2 \to aA_1A_1 \mid aA_1A_3A_1 \mid b \mid aA_1A_1A_4 \mid aA_1A_3A_1A_4 \mid bA_4$ $A_3 \to aA_1 \mid aA_1A_3 \mid aA_1A_3A_3$ $A_4 \to A_1 \mid A_1A_4$	 A_4 est nouveau
Traitement de A_4	$S \to A_2A_2$ $A_1 \to aA_1 \mid aA_1A_3$ $A_2 \to aA_1A_1 \mid aA_1A_3A_1 \mid b \mid aA_1A_1A_4 \mid aA_1A_3A_1A_4 \mid bA_4$ $A_3 \to aA_1 \mid aA_1A_3 \mid aA_1A_3A_3$ $A_4 \to aA_1 \mid aA_1A_4 \mid aA_1A_3 \mid aA_1A_3A_4$	

Table 9.8 – Une grammaire en cours de Greibach-normalisation

Chapter 10

Notions de calculabilité

Ce chapitre porte sur la notion de langage décidable et semi-décidable. Nous resterons à un niveau assez informel, mais il est possible, en particulier en se fixant un modèle de calcul comme les machines de Turing, de montrer rigoureusement tous les résultats vus ici.

10.1 Décidabilité et semi-décidabilité

Nous définissions la notion de langage décidable et semi-décidable en reprenant et développant les définitions du chapter 2, plus particulièrement de la section 2.2.2. Les définitions de cette partie sont exprimées en utilisant le terme vague d'*algorithme*: ce terme peut être interprété comme "n'importe quel programme d'un langage de programmation classique (p. ex., C ou Java) qui s'exécuterait dans un environnement sans limitation matérielle (taille mémoire, espace de stockage)". À la fin de ce chapitre (section 10.4), nous expliquerons comment nous pouvons rendre la notion d'algorithme plus formelle et plus précise, en se donnant un *modèle de calcul*, mécanisme formel pour décrire un processus automatique prenant éventuellement des données en entrée et produisant des données en sortie. On dit qu'un algorithme *termine* (sur l'entrée x) si l'exécution de cet algorithme s'arrête après un nombre fini d'étapes (sur l'entrée x).

On se donne un alphabet Σ.

On commence par introduire les langages semi-décidables comme ceux dont les mots sont énumérés un par un par un algorithme ne prenant aucune entrée:

Definition 10.1. *Un langage $L \subset \Sigma^*$ est* récursivement énumérable (r.e., *ou* semi-décidable, *ou encore* semi-calculable*) s'il existe un algorithme A qui énumère les mots de L.*

Si le langage est infini, un tel algorithme d'énumération ne termine donc pas. On peut donner une définition équivalente des langages semi-décidables, en termes d'algorithmes prenant un mot en entrée et répondant Vrai si le mot appartient au langage.

Proposition 10.2. *Un langage $L \subset \Sigma^*$ est semi-décidable si et seulement s'il existe un algorithme A tel que pour tout mot $u \in \Sigma^*$:*

Chapitre rédigé par Pierre Senellart.

— *si $u \in L$, alors A termine sur u en produisant la sortie* VRAI*;*
— *si $u \notin L$, alors soit A termine sur u en produisant la sortie* FAUX*, soit A ne termine pas sur u.*

Proof. Pour montrer cette équivalence, fixons-nous un langage L semi-décidable et soit A un algorithme qui énumère L. On construit un algorithme A' ayant les propriétés recherchées de la manière suivante. On modifie A de manière à ce que, à chaque fois que celui-ci produit un mot en sortie, il compare ce mot avec le mot donné en entrée. Si les deux sont différents, on continue; sinon, on termine en produisant VRAI en sortie.

Réciproquement, soit A un algorithme qui a les propriétés de la proposition. On considère également un algorithme B qui considère l'ensemble des mots de Σ^* et énumère ceux qui appartiennent à L. La principale difficulté est que le nombre de mots de Σ^* est infini, et le nombre d'étapes de calcul à exécuter sur chaque mot n'est pas borné (il peut être infini pour les mots qui ne sont pas dans L, sur lesquels A peut ne pas terminer). On procède en effectuant tous les calculs en parallèle suivant la technique du "déployeur universel" (en anglais, *dovetailing*). La technique est comme suit:

1. On énumère un mot u de Σ^* en utilisant B.
2. On lance la première étape du calcul de A sur u. Pour tous les mots v énumérés jusqu'à présent et dont le calcul de A n'a pas terminé, on lance une étape supplémentaire du calcul de A sur v.
3. Si l'un des calculs de A sur un v s'est terminé en produisant VRAI en sortie, on produit v en sortie.
4. On revient à l'étape 1. □

Les langages semi-décidables sont ainsi ceux qui sont *reconnus* par un algorithme, sans présager du comportement de cet algorithme sur les mots qui ne sont pas dans le langage. Les langages décidables, par contre, ont un algorithme qui décide si oui ou non un mot appartient au langage:

Definition 10.3. *Un langage $L \subset \Sigma^*$ est* récursif *(ou* décidable*, ou* calculable*) s'il existe un algorithme A qui, prenant un mot u de Σ^* en entrée, termine et produit* VRAI *si u est dans L et* FAUX *sinon. On dit alors que l'algorithme A* décide *le langage L.*

Évidemment, tout langage décidable est également semi-décidable; dans ce qui suit, nous allons voir que tout langage n'est pas nécessairement semi-décidable, et que tout langage semi-décidable n'est pas nécessairement décidable.

10.2 Langages non semi-décidables

Dans cette partie, nous allons montrer qu'il existe des langages non semi-décidables. Ce résultat est assez simple à démontrer et il repose sur des arguments de *cardinalité*, c'est-à-dire, du "nombre" de langages semi-décidables par rapport au "nombre" de langages.

Nous commençons par quelques rappels de mathématiques élémentaires. Étant donné un ensemble E, l'*ensemble des parties* de E, noté 2^E, est l'ensemble de tous les sous-ensembles de E ($X \subset E$ et $X \in 2^E$ sont ainsi synonymes). Étant donnés deux ensembles A et B, on

dit qu'une fonction $f : A \to B$ est *injective* si $\forall (x, y) \in A^2, f(x) = f(y) \Rightarrow x = y$, *surjective* si $\forall x' \in B, \exists x \in A, f(x) = x'$ et *bijective* si elle est à la fois injective et surjective. À une injection $f : A \to B$ on peut naturellement associer une bijection $f^{-1} : f(A) \to A$ définie par $f^{-1}(x') = x \iff f(x) = x'$. S'il existe une injection de A vers B, il existe une surjection de B vers A et vice-versa. La composition de deux injections (respectivement, surjections) est une injection (respectivement, surjection). Un ensemble infini A est dit *infini dénombrable* s'il existe une surjection de $\mathbb{N}$ (l'ensemble des entiers naturels) vers A.

Nous aurons également besoin d'un résultat remarquable sur la relation entre un ensemble et l'ensemble de ses parties, connu sous le nom de théorème de Cantor. Sa preuve utilise un principe *diagonal* (dans l'esprit du *paradoxe du barbier*: le barbier d'une ville rase tous les gens qui ne se rasent pas eux-mêmes; se rase-t-il lui-même?).

Theorem 10.4 (Cantor)**.** *Soit E un ensemble quelconque. Il n'existe pas de surjection de E vers 2^E.*

Proof. Supposons par l'absurde l'existence d'un ensemble E et d'une surjection f de E vers 2^E. On pose $X = \{ e \in E \mid e \notin f(e) \}$. X est une partie de E; f étant surjective, il existe $x \in E$ telle que $f(x) = X$. De deux choses l'une:

— soit $x \in X$: par définition de X, $x \notin f(x) = X$, ce qui est une contradiction;
— soit $x \notin X$, c'est-à-dire, $x \notin f(x)$: par définition de X, $x \in X$, ce qui est une contradiction également. □

Nous sommes maintenant armés pour prouver l'existence de langages non semi-décidables:

Theorem 10.5. *Soit Σ un alphabet. Il existe un langage $L \subset \Sigma^*$ tel que L n'est pas semi-décidable.*

L'idée de la démonstration est de montrer que l'ensemble des langages semi-décidables est infini dénombrable (car ils sont indexés par les algorithmes) tandis que l'ensemble des langages est infini non dénombrable (comme ensemble des parties d'un ensemble dénombrable).

Proof. Soit $\mathcal{A}$ l'ensemble de tous les algorithmes énumérant des langages sur Σ. Pour $A \in \mathcal{A}$, on note $f(A)$ le langage énuméré par A. Par définition de la semi-décidabilité, f est une surjection de $\mathcal{A}$ vers l'ensemble des langages semi-décidables. Chaque algorithme de $\mathcal{A}$ a une description finie, par exemple comme une suite d'instructions d'un langage de programmation, ou comme une suite de bits. Autrement dit, chaque algorithme de $\mathcal{A}$ est décrit par un mot d'un certain langage, sur un certain alphabet Σ' (disons, $\Sigma' = \{0, 1\}$ pour une description par suite de bits). Donc $\mathcal{A}$ peut être vu comme un sous-ensemble de Σ'^*: il y a une injection de $\mathcal{A}$ vers Σ'^* et donc une surjection g de Σ'^* vers $\mathcal{A}$. Par ailleurs, il est facile de construire une bijection h de $\mathbb{N}$ vers Σ'^*: on énumère les mots un à un par taille croissante (ϵ, 0, 1, 00, 01, 10, 11, 001...) et on affecte à chaque entier $n \in \mathbb{N}$ le $n+1$-ème mot de cette énumération. Au final, $f \circ g \circ h$ est une surjection de $\mathbb{N}$ vers l'ensemble des langages semi-décidables et ce dernier est infini dénombrable.

Considérons maintenant l'ensemble de tous les langages, 2^{Σ^*}. Supposons par l'absurde que tous les langages soient semi-décidables. Alors d'après la remarque précédente, il existe une surjection $f \circ g \circ h$ de $\mathbb{N}$ vers 2^{Σ^*}. Mais on vient de voir qu'il était possible de construire une bijection φ de $\mathbb{N}$ vers Σ^*. Alors $f \circ g \circ h \circ \varphi^{-1}$ est une surjection de Σ^* vers 2^{Σ^*}, ce qui est absurde d'après le théorème de Cantor. □

Cette preuve montre en fait qu'il existe beaucoup plus de langages non semi-décidables (une infinité non dénombrable) que de langages semi-décidables (une infinité dénombrable). Nous donnerons à la fin de la partie suivante quelques exemples de langages non semi-décidables, mais la plupart des langages non semi-décidables n'ont en fait pas de description en français puisqu'il n'y a qu'un "nombre" infini dénombrable de telles descriptions!

10.3 Langages non décidables

Les langages non semi-décidables ont peu d'intérêt en pratique: il n'existe pas d'algorithme d'énumération pour ces langages donc il est rare de s'y intéresser. Un langage qui serait semi-décidable mais non décidable est par contre beaucoup plus intéressant: on peut énumérer l'ensemble de ses mots, mais on ne peut pas en fournir d'algorithme de reconnaissance. Le mathématicien David Hilbert avait fixé comme l'un des grands défis des mathématiques du 20e siècle le *problème de décision* (*Entscheidungsproblem*): parvenir à une méthode automatique de calcul permettant de prouver si un résultat mathématique est vrai ou faux. L'existence de langages semi-décidables (donc formellement descriptibles) mais non décidables (donc pour lequel il est impossible de décider de l'appartenance d'un mot) a pour conséquence l'impossibilité de résoudre le problème de décision de Hilbert. Certains langages correspondant à des concepts très utiles en pratique se trouvent être semi-décidables mais non décidables. Nous allons montrer que c'est le cas du langage de l'arrêt.

Le *langage de l'arrêt* est l'ensemble H des mots sur un alphabet donné décrivant les couples d'algorithmes et entrées tels que l'algorithme termine sur cette entrée. Formellement, soit $\Sigma = \{0, 1\}$. Soit $\mathcal{A}$ l'ensemble des algorithmes prenant en entrée un mot u de Σ^*. Quitte à réencoder en binaire les descriptions des algorithmes, on peut voir tout algorithme A de $\mathcal{A}$ comme un mot de Σ^*. Quitte également à ajouter un codage spécial pour les paires, on peut voir toute paire formée d'un algorithme $A \in \mathcal{A}$ et d'un mot $u \in \Sigma^*$ comme un mot $f(A, u)$ de Σ^*, f étant une injection de $\mathcal{A} \times \Sigma^*$ vers Σ^* (on peut faire en sorte que f et f^{-1} soient exprimables par un algorithme de $\mathcal{A}$). Le langage de l'arrêt est l'ensemble $H \subset \Sigma^*$ tel que $x \in H$ si et seulement si il existe $A \in \mathcal{A}$ et $u \in \Sigma^*$ avec $x = f(A, u)$ et *A s'arrêtant sur u*.

La preuve de non-décidabilité du langage de l'arrêt utilise encore une fois un argument diagonal.

Theorem 10.6 (Turing)**.** *Le langage de l'arrêt est semi-décidable mais non décidable.*

Proof. Montrons tout d'abord que H, le langage de l'arrêt, est semi-décidable. On construit un algorithme B qui prend en entrée un mot x de Σ^* et procède comme suit:

1. si $x \notin f(\mathcal{A}, \Sigma^*)$, l'algorithme retourne Faux;
2. sinon, l'algorithme calcule $f^{-1}(x) = (A, u)$ et applique l'algorithme A à l'entrée u; si ce calcul termine, l'algorithme retourne Vrai.

On a choisi B tel que si $x \in H$, B retourne Vrai sur x (et sinon, B peut soit retourner Faux, soit ne pas terminer) donc H est semi-décidable.

Montrons maintenant que H n'est pas décidable. On procède par l'absurde: soit B un algorithme qui décide H. $\mathcal{A}$ et Σ^* sont deux langages dénombrables (cf. section précédente)

donc on peut numéroter les algorithmes de $\mathcal{A}$ et les mots de Σ^* par A_0, $A_1\ldots$ et u_0, $u_1\ldots$, respectivement. On construit un algorithme B' prenant en entrée un mot u_i de Σ^* de la manière suivante: on commence par appliquer B à $f(A_i, u_i)$. Si B renvoie Faux, on renvoie Faux; sinon, on part dans une boucle infinie. B' étant un algorithme de $\mathcal{A}$, il existe un entier k tel que $B' = A_k$. Appliquons maintenant l'algorithme de décision B à $f(B', u_k)$. De deux choses l'une:

— B retourne Vrai sur $f(B', u_k)$. Ceci signifie que B' s'arrête sur l'entrée u_k, et donc, par définition de B', que B retourne Faux sur $f(A_k, u_k) = f(B', u_k)$, ce qui est une contradiction.
— B retourne Faux sur $f(B', u_k)$. Donc B' ne s'arrête pas sur l'entrée u_k et donc, comme B s'arrête sur toute entrée, B renvoie Vrai avec l'entrée $f(A_k, u_k) = f(B', u_k)$. Nous aboutissons de nouveau à une contradiction. □

L'argument diagonal, l'idée de coder les algorithmes par des mots du langage et le résultat lui-même rappellent la preuve du théorème d'incomplétude de Gödel, qui est un analogue pour la logique mathématique du théorème de l'arrêt de Turing en logique informatique. Le théorème de Gödel énonce que dans tout système formel permettant d'exprimer l'arithmétique, il existe des énoncés mathématiques que l'on ne peut ni prouver ni infirmer. Le théorème de l'arrêt montre que quand on se fixe un modèle de calcul suffisamment puissant, il existe des propriétés mathématiques de ce modèle de calcul qui ne peuvent être calculées.

Beaucoup de problèmes concrets dans de nombreux domaines de l'informatique sont ainsi décrits par des langages semi-décidables mais non décidables. Ici, le problème "déterminer si $x \in X$ a la propriété P" est décrit par le langage "l'ensemble des $x \in X$ qui ont la propriété P". Dans tous les exemples ci-dessous, "l'ensemble des $x \in X$" lui-même est toujours décidable.

— Déterminer si une grammaire hors-contexte est ambiguë.

— Déterminer si la fonction calculée par un programme d'un langage de programmation classique a n'importe quelle propriété non triviale (par exemple, elle ne lève jamais d'exceptions). Ce résultat est le *théorème de Rice*.

— Déterminer si une requête écrite dans le langage d'interrogation de bases de données SQL renvoie un résultat sur au moins une base de données (*théorème de Trakhtenbrot*).

— Étant donnés deux ensembles de mots, déterminer si une séquence de mots du premier ensemble peut être identique à une séquence de mots du second ensemble (problème de *correspondance de Post*).

— Étant données des règles de calcul dans un groupe, déterminer si deux expressions sont égales.

Un exemple de langage non semi-décidable est le complément du langage de l'arrêt. Un tel langage, dont le complément est récursivement énumérable, est dit *co-récursivement énumérable* (noté co-r.e.). On peut construire un langage qui n'est ni r.e. ni co-r.e. en combinant deux langages possédant ces propriétés: par exemple, l'ensemble des couples d'algorithmes dont le premier termine et le second ne termine pas sur une entrée donnée.

10.4 Modèles de calculs

Nous précisons maintenant ce que nous entendons par un *algorithme* dans les parties qui précèdent. Rappelons qu'un algorithme est un procédé de calcul automatique prenant en entrée des données éventuelles (un mot sur un certain alphabet) et produisant en sortie des données (un mot sur un certain alphabet). Une des méthodes les plus pratiques de formaliser cette notion d'algorithme est par le biais des *machines de Turing*, qui peuvent être vues comme des *automates finis à bande*. Nous définissons les machines de Turing dans le cas simple où la sortie est soit VRAI soit FAUX; on peut facilement étendre la définition au cas où la machine produit une sortie non booléenne, par exemple en rajoutant une seconde bande.

Definition 10.7. *Une* machine de Turing *(déterministe) est la donnée de:*

(i) *un ensemble fini Q d'états;*
(ii) *un alphabet de travail Γ;*
(iii) *$b \in \Gamma$ est un symbole spécial "blanc";*
(iv) *$\Sigma \subset \Gamma \backslash \{b\}$ est l'alphabet d'entrée/sortie;*
(v) *q_0 est un état initial;*
(vi) *$F \subset Q$ est un ensemble d'états finaux;*
(vii) *$\delta : Q \backslash F \times \Gamma \to Q \times \Gamma \times \{L, R\}$ est une fonction de transition.*

Étant donnée une machine de Turing $(Q, \Gamma, b, \Sigma, q_0, F, \delta)$, un calcul *sur l'entrée $u \in \Sigma^*$ est une séquence de configurations $\{(q_k, \varphi_k, i_k)\}_{1 \leqslant k \leqslant n}$ où $q_k \in Q$ est l'état courant, $\varphi_k : \mathbb{Z} \to \Gamma$ est une fonction associant un symbole à chaque position d'une bande virtuelle infinie, et $i_k \in \mathbb{Z}$ est la position actuelle sur la bande. On impose que la séquence de configurations respecte les propriétés suivantes:*

— *$q_1 = q_0$, $i_1 = 0$, $\varphi_1(x) = u_x$ pour $1 \leqslant x \leqslant |u|$ et $\varphi_1(x) = b$ pour $x \notin [1, |u|]$.*
— *Pour $1 \leqslant k \leqslant n-1$, $\delta(q_k, \alpha_k) = (q_{k+1}, \alpha_{k+1}, \beta)$ avec:*
  — *$i_{k+1} = i_k + 1$ si $\beta = R$, $i_{k+1} = i_k - 1$ sinon;*
  — *Pour $x \neq i_k$, $\varphi_k(x) = \varphi_{k+1}(x)$;*
  — *Pour $x = i_k$, $\varphi_k(x) = \alpha_k$ et $\varphi_{k+1}(x) = \alpha_{k+1}$.*

Un calcul d'une machine de Turing est dit acceptant *(ou* produisant VRAI*) s'il conduit à un état final,* échouant *(ou* produisant FAUX*) s'il conduit à un état non final depuis lequel il n'existe pas de transition possible.*

De même qu'un automate fini déterministe, une machine de Turing effectue un calcul sur un mot d'entrée pour déterminer si le mot est accepté ou non. Ici, le calcul est plus complexe que pour un automate car les transitions peuvent également dépendre des symboles écrits sur une bande de papier de taille non bornée (décrite par les φ_k); noter cependant qu'un calcul acceptant ou échouant donné n'utilisera qu'une partie finie de cette bande.

Pour un exemple de machine de Turing et de son exécution, voir http://ironphoenix.org/tril/tm/ (Palindrome Detector).

Les machines de Turing peuvent être utilisées comme modèle de calcul pour les notions de semi-calculabilité et calculabilité, ce qui donne un cadre formel aux résultats présentés dans les parties précédentes. Elles sont également utilisées pour formaliser les notions de *complexité d'un problème*: la complexité est la longueur du calcul que doit faire une machine de Turing pour résoudre ce problème en fonction de la taille de l'entrée.

De manière très intéressante, un grand nombre d'autres formalismes ont le même *pouvoir d'expression* que les machines de Turing, c'est-à-dire qu'ils permettent de décider exactement les mêmes langages.

λ-calcul. Introduit par Church parallèlement à Turing, ce formalisme, à la base des langages de programmation fonctionnels, est basé sur la définition et l'application de fonctions récursives, décrites dans un langage logique muni de règles de *réduction* permettant de réécrire ces formules logiques.

Fonctions récursives. La théorie des fonctions récursives a été élaborée par Gödel et Kleene comme modèle de fonction calculable basé sur un ensemble de fonctions et primitives de base: fonctions constantes, composition, récursion, etc.

Machines à registres. Une alternative aux machines de Turing plus proche des ordinateurs moderne est celle des machines à registres: ici, la mémoire n'est plus représentée comme une bande infinie, mais comme un ensemble de registres mémoire (ou *variables*) qui peuvent être lus et écrits dans un ordre quelconque. L'*architecture de von Neumann*, utilisée pour concevoir les premiers ordinateurs, est un exemple de ces machines à registres.

Langages de programmation Turing-complets. La plupart des langages utilisés pour développer des logiciels [1] sont *Turing-complets*: si on suppose qu'ils sont exécutés dans un environnement sans limite de mémoire, ils permettent de décider n'importe quel langage décidable par une machine de Turing.

Les grammaires génératives de type 0, introduites par Chomsky, ne sont pas un modèle de calcul à proprement parler, mais engendrent exactement les langages semi-décidables.

Cette remarquable équivalence de modèles formels aussi divers de ce que peut être un processus automatique de calcul a conduit Church et Turing à formuler ce qu'on appelle la *thèse de Church–Turing*: ce qui est intuitivement calculable (ou ce qui est *humainement* calculable), c'est ce qui est calculable par une machine de Turing (ou par le λ-calcul, etc.).

1. Les exceptions sont quelques rares langages qui sont utilisés pour des buts spécifiques, comme SQL sans les fonctions utilisateurs pour l'interrogation de bases de données relationnelles, ou XPath pour la navigation dans les arbres XML.

Part II

Annexes

Chapter 11

Compléments historiques

Ce chapitre porte sur l'histoire de l'informatique. Le but est de donner un contexte historique aux grandes notions et résultats abordés dans le cours, en expliquant brièvement qui ont été les logiciens et informaticiens qui ont laissé leurs noms aux notions de calculabilité et de langages formels.

11.1 Brèves biographies

John Backus (Américain, 1924 – 2007)

est le concepteur du premier langage de programmation de haut niveau compilé, Fortran. Co-auteur avec Peter Naur d'une notation standard pour les grammaires hors-contexte (*BNF* ou *Backus–Naur Form*), il a obtenu le prix Turing en 1977 pour ses travaux sur les langages de programmation.

Janusz Brzozowski (Polono-Canadien, né en 1935)

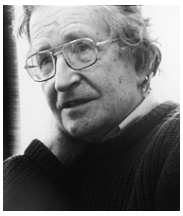
est un chercheur en théorie des automates, qui fut l'étudiant de Edward J. McCluskey. Il a apporté de nombreuses contributions dans ce domaine, et est en particulier connu pour l'*algorithme de minimisation de Brzozowski* et pour l'algorithme de Brzozowski et McCluskey (section 4.2.2).

Chapitre rédigé par Pierre Senellart.

Georg Cantor (Allemand, 1845 – 1918)

est l'inventeur de la théorie des ensembles et des notions d'ordinal et de cardinal d'un ensemble infini. Le nom de Cantor est donné à un ensemble dont il a imaginé la construction; cet ensemble, parmi les premiers exemples de fractales, a des caractéristiques topologiques et analytiques très inhabituelles. Le *théorème de Cantor* énonce qu'un ensemble n'est jamais en bijection avec l'ensemble de ses parties, ce qui a pour conséquence que l'ensemble $\mathbb{R}$ des nombres réels n'est pas en bijection avec l'ensemble $\mathbb{N}$ des entiers naturels. Ce résultat, et l'argument de *diagonalisation* utilisé dans sa preuve, fut très mal accueilli par certains mathématiciens de l'époque, y compris de très grands comme Henri Poincaré. La fin de la vie de Georg Cantor fut marquée par une longue période de dépression et de récurrents séjours en hôpital psychiatrique.

Noam Chomsky (Américain, né 1928)



est le fondateur de la linguistique moderne. Pour décrire le langage naturel, il a développé la notion de grammaire générative (en particulier, de grammaire hors-contexte) et a étudié les pouvoirs d'expression de différentes classes de grammaire au sein de la *hiérarchie de Chomsky*. Ces travaux s'appuient sur l'hypothèse d'une *grammaire universelle*, qui suppose que certains mécanismes du langage, universels et indépendants de la langue, sont pré-câblés dans le cerveau. Ses travaux ont eu un impact majeur en linguistique, en traitement du langage naturel, et en informatique théorique. Noam Chomsky est également connu pour son activisme politique très critique de la politique étrangère des États-Unis.

Alonzo Church (Américain, 1903 – 1995)

était un pionnier de la logique informatique. Stephen C. Kleene et Alan Turing furent ses étudiants. Il proposa le λ-calcul (ancêtre des langages de programmation fonctionnels comme Lisp) comme modèle de calculabilité et démontra, indépendamment de Turing, l'impossibilité de résoudre le problème de décision de Hilbert (*théorème de Church–Turing*). Il est également connu pour la *thèse de Church–Turing*, une hypothèse qui postule que les différents modèles de calculabilité proposés (λ-calcul, machines de Turing, fonctions récursives générales, machines de von Neumann), tous démontrés équivalents, correspondent à la notion intuitive de calculabilité; dans sa version la plus générale, cette thèse affirme que tout ce qui peut être calculé par la nature, et donc par le cerveau humain, peut l'être par une machine de Turing.

Augustus De Morgan (Britannique, 1806 – 1871)

fut l'un des premiers à tenter de baser les mathématiques sur un raisonnement logique formel. Les *lois de De Morgan* expriment que la négation d'une disjonction est la conjonction des négations, et vice-versa.

Victor Glushkov (Soviétique, 1923 – 1982)

fut un pionnier de la théorie de l'information, de la théorie des automates et de la conception des premiers ordinateurs en Union soviétique. Il a donné son nom à la construction de Glushkov d'un automate à partir d'une expression rationnelle, alternative à la construction de Thompson.

Kurt Gödel (Autrichien, 1906 – 1978)

est connu pour ses travaux sur la théorie des fonctions récursives, un modèle de calculabilité, mais surtout pour deux résultats fondamentaux de logique mathématique: le théorème de *complétude* de Gödel montre que tout résultat vrai dans tout modèle de la logique du premier ordre est démontrable; le théorème d'*incomplétude* de Gödel montre que dans tout système formel permettant d'exprimer l'arithmétique, il existe des énoncés mathématiques que l'on ne peut ni prouver ni infirmer. Il mourut dans des circonstances tragiques: victime de paranoïa, il était convaincu qu'on cherchait à l'empoisonner, et refusa petit à petit de s'alimenter.

Sheila Greibach (Américaine, née 1939)

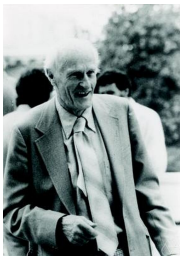
est une chercheuse en langages formels. Elle est en particulier connue pour la *forme normale de Greibach* d'une grammaire hors-contexte et ses conséquences sur l'équivalence entre grammaires hors-contexte et automates à pile.

David Hilbert (Allemand, 1862 – 1943)

est reconnu comme l'un des plus grands mathématiciens ayant existé. Il a apporté des contributions majeures dans de nombreux domaines, comme la géométrie, l'analyse (p. ex., *espaces de Hilbert*), la physique mathématique et la logique. En 1900, il énonce 23 problèmes ouverts importants des mathématiques. Cette liste a eu une très grande influence sur la recherche en mathématique au 20^{e} siècle. L'un de ces problèmes, le problème de décision (ou *Entscheidungsproblem*), consistait à obtenir une procédure automatique permettant de décider si un énoncé mathématique est vrai ou faux. En 1920, Hilbert énonce un programme pour l'axiomatisation des mathématiques, par lequel tout énoncé mathématique pourrait être prouvé ou infirmé à partir des axiomes. Kurt Gödel a montré que cela était impossible; Alonzo Church et Alan Turing en ont indépendamment déduit l'impossibilité de résoudre le problème de décision. L'épitaphe de David Hilbert résume sa vision de la science:

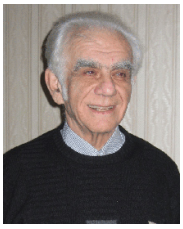
Wir müssen wissen. *(Nous devons savoir.)*
Wir werden wissen. *(Nous saurons.)*

Stephen C. Kleene (Américain, 1909 – 1994)



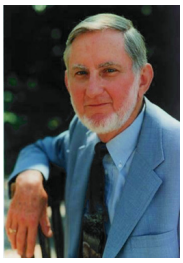
est l'inventeur des expressions rationnelles et a apporté des contributions majeures à la théorie des langages récursifs, un modèle de calculabilité. L'opérateur d'*étoile*, caractéristique des expressions rationnelles, porte son nom, de même que le théorème d'équivalence entre expressions rationnelles et automates finis (section 4.2.2).

Vladimir Levenshtein (Russe, né en 1935)



est un chercheur dans les domaines de la théorie de l'information et des codes correcteurs d'erreur. Il a introduit dans ce cadre la distance d'édition qui porte son nom.

Edward J. McCluskey (Américain, né en 1929)



est connu pour ses travaux sur la conception logique des circuits électroniques. Il est en particulier l'auteur avec W. V. Quine de la *méthode de Quine–McCluskey* de minimisation de fonctions booléennes en utilisant la notion d'impliquants premiers.

Edward F. Moore (Américain, 1925 – 2003)

est l'un des fondateurs de la théorie des automates. Il introduit l'algorithme de minimisation qui porte son nom, et la notion d'automate avec sortie (introduite également indépendamment par George H. Mealy).

Peter Naur (Danois, 1928 – 2016)

est un chercheur en informatique qui a travaillé en particulier sur la conception des langages de programmation. Co-auteur avec John Backus d'une notation standard pour les grammaires hors-contexte (*BNF* ou *Backus–Naur Form*), il a obtenu le prix Turing en 2005 pour ses travaux sur le langage Algol.

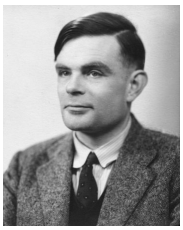
John von Neumann (Hongro-Américain, 1903 – 1957)

, en plus d'être un des pères fondateurs de l'informatique, a fourni des contributions majeures dans de nombreux domaines scientifiques (en particulier, théorie des ensembles, mécanique quantique, théorie des jeux). En informatique, il est connu pour les automates cellulaires, le tri fusion, et bien sûr l'architecture de von Neumann, le modèle formel de calcul qui se rapproche le plus des ordinateurs actuels. Ce modèle a été élaboré lors de ses travaux sur la construction des premiers ordinateurs (ENIAC, EDVAC). Durant la deuxième guerre mondiale, il a travaillé dans le projet américain Manhattan de conception de la bombe atomique, et a fait partie de la commission américaine à l'énergie atomique pendant la guerre froide. L'organisation IEEE décerne chaque année une très prestigieuse médaille John von Neumann à un chercheur en informatique.

Ken Thompson (Américain, né en 1943)

est un informaticien notable pour ses contributions à la fois théoriques et appliquées. Il a popularisé les expressions rationnelles en créant notamment l'utilitaire Unix `grep`, basé sur la construction (*de Thompson*, section 4.2.2) d'un automate décrivant l'expression rationnelle. Il a travaillé sur la conception du langage B avec Dennis Ritchie, ancêtre du langage C. Toujours avec Ritchie, il est à l'origine du système d'exploitation Unix. Il a également élaboré avec Rob Pike l'encodage de caractères UTF-8. Il obtient le prix Turing en 1983 avec Dennis Ritchie pour leurs recherches sur les systèmes d'exploitation et l'implémentation d'Unix.

Alan Turing (Britannique, 1912 – 1954)



est souvent considéré comme le premier informaticien. Ses travaux sur le problème de décision le conduisent à introduire le modèle de calculabilité connu aujourd'hui sous le nom de machine de Turing. En se basant sur les travaux de Kurt Gödel, il prouve l'indécidabilité de l'arrêt d'une machine de Turing, ce qui montre l'impossibilité d'obtenir une solution générale au problème de décision (*théorème de Church–Turing*). Durant la seconde guerre mondiale, il fut l'un des principaux cryptanalystes qui déchiffrèrent les codes de communication de l'armée allemande. Après la guerre, il travailla sur la conception des premiers ordinateurs. Alan Turing est également connu pour la notion de *test de Turing*, une expérience de pensée permettant de définir la notion d'intelligence artificielle. En 1952, Alan Turing fut condamné pour homosexualité à une castration chimique. Il meurt mystérieusement en 1954 (empoisonnement par une pomme enrobée de cyanure, à l'image du film de Walt Disney *Blanche-Neige*, qu'il affectionnait particulièrement), probablement un suicide. La société savante ACM décerne chaque année un prix Turing, considéré comme la plus haute distinction que peut recevoir un chercheur en informatique.

11.2 Pour aller plus loin

Pour approfondir l'histoire de la théorie des langages, voir les auteurs suivants: Hodges (1992), Doxiadis et Papadimitriou (2010), Dowek (2007), Turing (1936), Barsky (1997), Perrin (1995).

Attributions
Certaines photographies sont reproduites en vertu de leur licence d'utilisation Creative Commons, et attribuées à:

— John Backus: Pierre Lescanne
— Stephen C. Kleene: Konrad Jacobs (Mathematisches Forschungsinstitut Oberwolfach)
— Peter Naur: Eriktj (Wikipedia EN)
— Kenneth Thompson: Bojars (Wikimedia)

Les photographies suivantes sont reproduites en vertu du droit de citation:

— Janusz Brzozowski
— Alonzo Church
— Sheila Greibach
— Vladimir Levenshtein
— Edward F. McCluskey

Les autres photos sont dans le domaine public.

Chapter 12

Correction des exercices

12.1 Correction de l'exercise 4.5

1. For L_1, we verify that the choice of $I_1 = \{a\}$, $F_1 = \{a\}$, and $T_1 = \{ab, ba, bb\}$ guarantees $L_1 = (I_1\Sigma^\star \cap \Sigma^\star F_1) \setminus \Sigma^\star T_1 \Sigma^\star$.
   For L_2, it works the same for $I_2 = \{a\}$, $F_2 = \{b\}$, and $T_2 = \{aa, bb\}$.

   Note: For L_1, it suffices to take $T_1 = \{ab\}$ to forbid occurrences of b in L_1.

2. If we wish to accept the words of L_1 and L_2, it is necessary to have $a \in I$, $a, b \in F$, and T contain at most bb. Two possibilities: either $T = \emptyset$, but then the corresponding language does not contain a; or $T = \{bb\}$, in which case the word such as aab is in the local language local defined by I, F, et T, but not in the language $aa^\star + ab(ab)^\star$.

3. For the intersection, it is sufficient to use the fact that $A \setminus B = A \cap \overline{B}$. $L_1 \cap L_2$ can be expressed as an intersection of six terms; using the fact that intersection is associative and commutative, and that concatenation is distributive with respect to intersection, we have:
   $$L_1 \cap L_2 = ((I_1 \cap I_2)\Sigma^\star \cap \Sigma^\star(F_1 \cap F_2)) \setminus (\Sigma^\star(T_1 \cup T_2)\Sigma^\star)$$
   This proves that the intersection of two local languages is a local language.
   For the union, the example in question 2 serves as a direct counterexample.

4. I, F, and T are finite, thus rational, $\Sigma^\star$ as well; the rational are stable by concatenation, intersection, and complement. This suffices to conclude that any language of the form $(I\Sigma^\star \cap \Sigma^\star F) \setminus \Sigma^\star T \Sigma^\star$ is rational.

5. A local language recognizing the words of C must necessarily have $a, b \in I$, $a, b, c \in F$. We seek the le smallest language possible, which leads us to forbid the factors of length 2 which are not in at lest one word of c: $T = \{ac, bb, ba, cb, cc\}$.
   The automaton corresponding to an initial state q_0, which has transitions existing over the letters in I, and one state per letter: q_a, q_b, q_c. These states carry the "memory" of the most recent letter read (after an a, we go to q_a...). The exiting transitions of these states are defined in a way to forbid the factors of T: after an a, we may have a or b but not c, of where the transitions $\delta(q_a, a) = q_a$, $\delta(q_a, b) = q_b$. Likewise, we have: $\delta(q_b, c) = q_c$, $\delta(q_c, a) = q_a$. Finally, a state q_x is final if and only if $x \in F$. Here, $q_a, q_b, q_c \in F$.

Note: Many students correctly identify I, F, and T but construct an automaton which only recognizes the finite set of words, of which are not possible here.

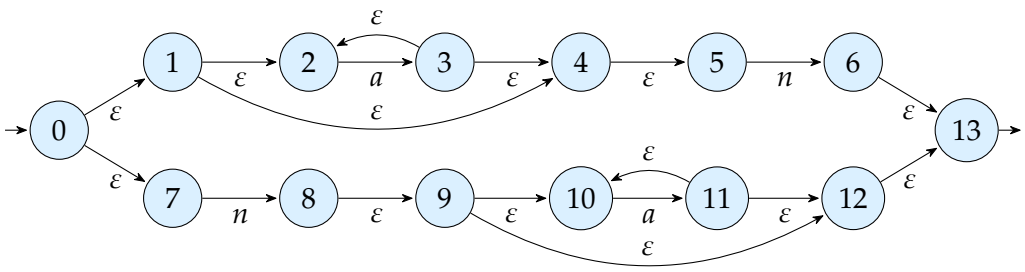
6. The intuition of this beautiful result is that a finite automaton, A, is defined by its initial state, its final states, and its transitions. Transposed into the set of walks in A, The initial state is characterized by the prefixes of length 1 of walks; the final states by the suffixes of length 1, and the transitions by the factors of length 2.
   Formally, let L be rational and $A = (\Sigma, Q, q_o, F, \delta)$ be a deterministic finite automaton recognizing L. We define L' over the alphabet $\Sigma' = Q \times \Sigma \times Q$ by these three sets:
   — $I = \{(q_0, a, \delta(q_o, a)), a \in \Sigma\}$
   — $F = \{(q, a, r), q \in Q, a\Sigma, r \in \delta(a, q) \cap F\}$
   — $\Sigma' \times \Sigma' \setminus \{(q, a, r)(r, b, p), q \in Q, a, b\Sigma, r \in \delta(a, q), b \in \delta(r, b)\}$.
   Now, let $u \in L$. Its computation in A is, by construction, a $w \in L'$ for which at least $\phi(w) = u$, thus $L \subset \phi(L')$. Conversely, if $u = \phi(w)$, with $w \in L'$, then we deduce immediately a computation for $u \in A$, and thus $u \in L$.

12.2 Correction de l'exercise 4.10

1. A is not deterministic. There are two transitions each of whose origin is 0 and whose label is b– one with destination 0 and one with destination 1.
2. The language recognized by A is the set of words with suffix $b(c + a)^\star d$; that is $\Sigma^\star b(c + a)^\star d$.
3. To do...

12.3 Correction de l'exercise 4.14

1. La construction de Thompson conduit à l'automate 12.1.



Automaton 12.1 – Automaton d'origine pour $(a^\star n \mid na^\star)$

2. Le computation de l'ε-closure conduit aux résultats du table 12.2.
3. La suppression des transitions ε s'effectue en court-circuitant ces transitions. La suppression *backward* des transitions spontaneous consiste à agréger les spontaneous transitions with les transitions non spontaneous qui sont à leur aval. En d'autres termes,

État	Closure
0	0, 1, 2, 4, 5, 7
1	1, 2, 4, 5
2	2
3	2, 3, 4, 5
4	4, 5
5	5
6	6, 13
7	7
8	8, 9, 10, 12, 13
9	9, 10, 12, 13
10	10
11	10, 11, 12, 13
12	12, 13
13	13

Table 12.2 – ε-closures (avant) des états de l'automate 12.1

Pour chaque état, l'ensemble des états accessibles par transitions spontaneous.

pour toute transition non spontanée $\delta(q, a) = q'$, on ajoute une transition partant de chaque state en amont de q, sur l'étiquette a, arrivant en q'. De même, tout state en amont d'un final state devient final.

En utilisant le tableau des clôtures avant (table 12.2), ajouter une transition $\delta(p, a) = q'$ pour tout state p tel que q appartienne à l'ε-closure avant de p. Marquer ensuite comme final tous les états dont la closure avant contient un final state.

État	Closure
0	0
1	0, 1
2	0, 1, 2, 3
3	3
4	0, 1, 3, 4
5	0, 1, 3, 4, 5
6	6
7	0, 7
8	8
9	8, 9
10	8, 9, 10, 11
11	11
12	8, 9, 11, 12
13	6, 8, 9, 11, 12, 13

Table 12.3 – ε-closures backwards des états de l'automate 12.1

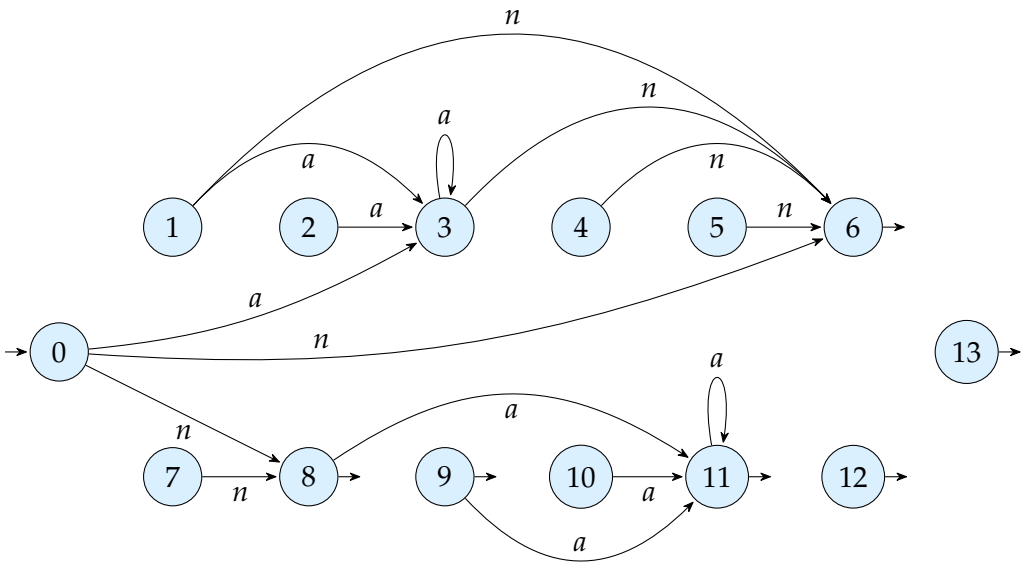
Pour chaque état, l'ensemble des états co-accessibles par spontaneous transitions.

Alternativement, en utilisant le tableau des clôtures backwards (table 12.3), il faut donc ajouter une transition $\delta(p, a) = q'$ pour tout state p dans l'backward ε-closure de q. On marque ensuite comme final tous les états dans la closure backward des états final.

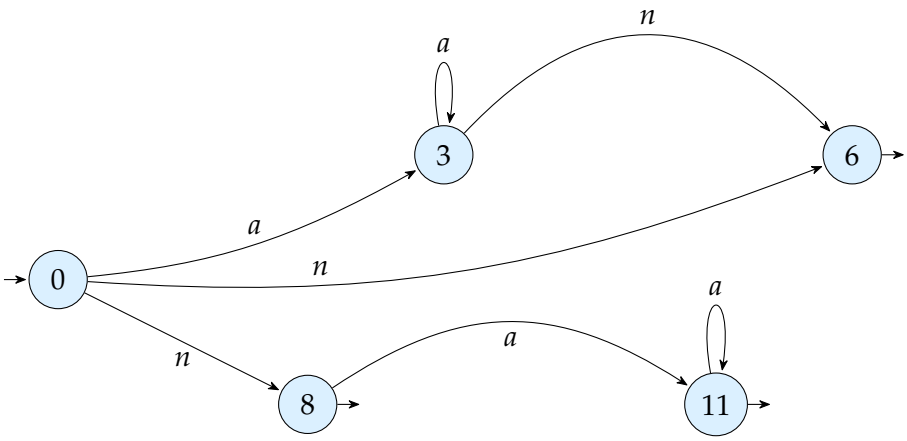
Cette procédure aboutit à l'automate 12.4.

Les états 1, 2, 4, 5, 7, 9, 10, 12 et 13 sont devenus non-useful dans l'automate 12.4, si on les élimine, on obtient l'automate 12.5. Cet automaton n'est pas deterministic, l'état 0 ayant deux transitions sortantes pour le symbol n.

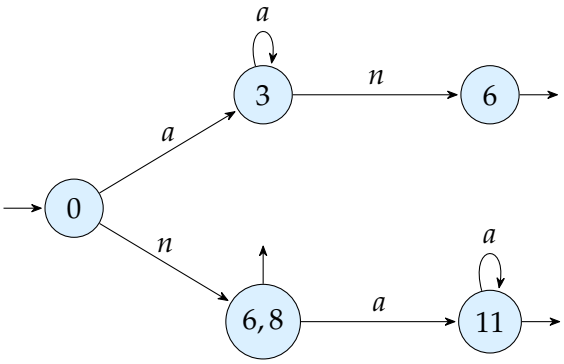
4. Après déterminisation par la méthode des sous-ensembles, on aboutit finalement à l'automate 12.6.



Automaton 12.4 – Automaton sans transition spontanée pour $(a^\star n \mid na^\star)$



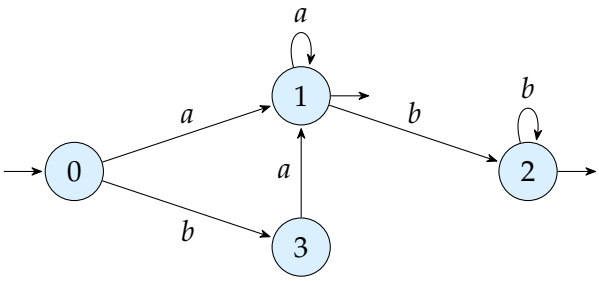
Automaton 12.5 – L'automate 12.4, émondé



Automaton 12.6 – Automaton deterministic pour $(a^\star n \mid na^\star)$

12.4 Correction de l'exercise 4.15

1. Après élimination de la transition spontanée, la méthode de construction des sous-ensembles permet de construire l'automate 12.7.



Automaton 12.7 – Déterminisation de l'automate 4.14

12.5 Correction de l'exercise 4.19

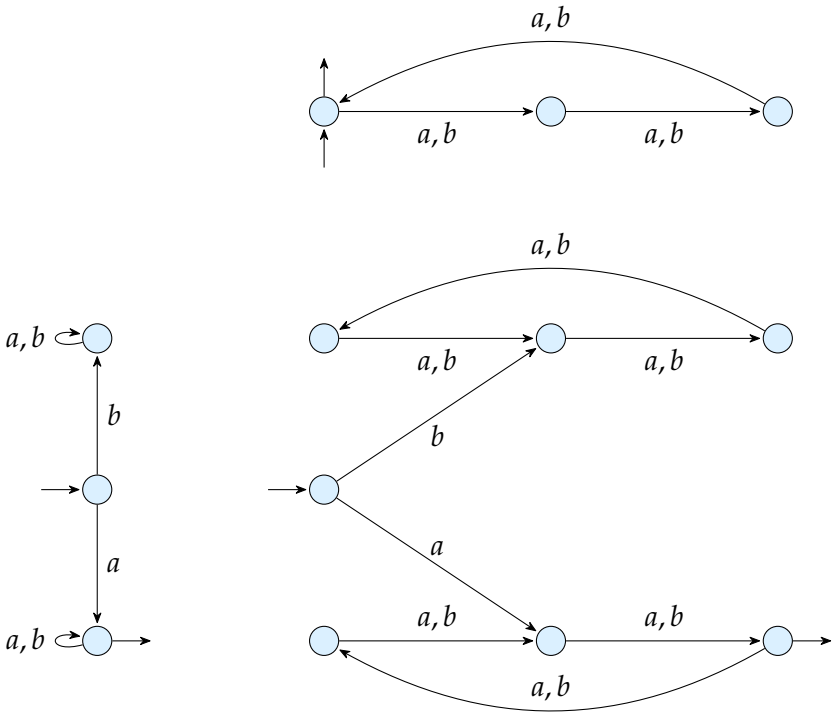
1. Les automata A_1 et A_2 sont représentés respectively verticalement et horizontalement dans la représentation de l'automate 12.8.
2. L'application de l'algorithme construisant l'intersection conduit à l'automate 12.8.
3. a. L'algorithme d'élimination des transitions ε demande de compute dans un premier temps l'ε-closure de chaque état. Puisque les seules transitions ε sont celles ajoutées par la construction de l'union, on a:
   — $\varepsilon - closure(q_0) = \{q_0^1, q_0^2\}$
   — $\forall q \in Q^1 \cup Q^2, \varepsilon - closure(q) = \{q\}$

   L'élimination des deux transitions ε conduit donc à ajouter les transitions suivantes: $\forall a \in \Sigma, \delta(q_0, a) = \{\delta(q_0^1, a), \delta(q_0^2, a)\}$. q_0 est bien non-deterministic et c'est le seul state dans ce cas, puisqu'aucune autre transition n'est rajoutée.

   b. L'algorithme de construction de la partie utile du déterminisé construct de proche en proche les transitions des états accessibles depuis q_0. On l'a vu, le traitement de q_0 ne construct que des états useful de la forme $\{\delta(q_0^1, a), \delta(q_0^2, a)\}$, qui correspondent à des paires de $Q^1 \times Q^2$, car A_1 et A_2 sont deterministics. Supposons qu'à l'étape n de l'algorithme, on traite un state $q = \{q^1, q^2\}$, with $q^1 \in Q^1$ et $q^2 \in Q^2$. Par définition du déterminisé on a:

$$\forall a \in \Sigma, \delta(q, a) = \delta^1(q^1, a) \cup \delta^2(q^2, a)$$

   Les A_i étant deterministics, chacun des $\delta^i(q^i, a)$ est un singleton de Q^i (pour $i = 1, 2$), les nouveaux états useful créés lors de cette étape correspondent bien des doubletons de $Q^1 \times Q^2$.
   On note également que, par construction, les transitions de $\overline{A}$ sont identiques aux transitions de la construction directe de l'intersection.



Automaton 12.8 – Les automata pour A_1 (à gauche), A_2 (en haut) et leur intersection (au milieu)

c. Les états final du déterminisé sont ceux qui contiennent un final state de $\overline{A^1}$ ou de $\overline{A^2}$. Les seuls non-final sont donc de la forme: $\{q^1, q^2\}$, with à la fois q^1 non-final dans $\overline{A_1}$ et q^2 non-final dans $\overline{A^2}$. Puisque les états non-final de $\overline{A^1}$ sont précisément les états finals de A^1 (et idem pour $\overline{A_2}$), les états finals de A correspondent à des doubletons $\{q^1, q^2\}$, with q^1 dans F^1 et q^2 dans F^2.

On retrouve alors le même ensemble d'états finals que dans la construction directe.

La dernière vérification est un peu plus fastidieuse et concerne l'état initial q_0. Notons tout d'abord que q_0 n'a aucune transition entrante (cf. le point [b]). Notons également que, suite à l'élimination des transitions ε, q_0 possède les mêmes transitions sortantes que celles qu'aurait l'état $\{q_0^1, q_0^2\}$ du déterminisé. Deux cas sont à envisager:

— soit la partie $\{q_0^1, q_0^2\}$ n'est pas constructe par l'algorithme de déterminisation et on peut assimiler q_0 à cette partie;

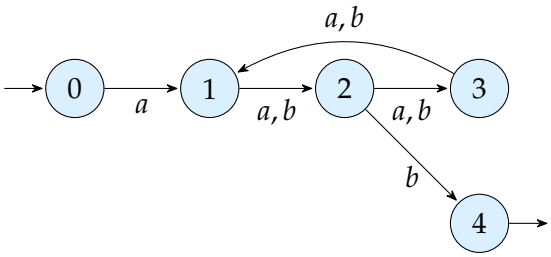
— soit elle est constructe et on peut alors rendre cette partie comme state initial et supprimer q_0: tous les computations réussis depuis q_0 seront des computations réussis depuis $\{q_0^1, q_2^0\}$, et réciproquement, puisque ces deux états ont les mêmes transitions sortantes.

12.6 Correction de l'exercise 4.20

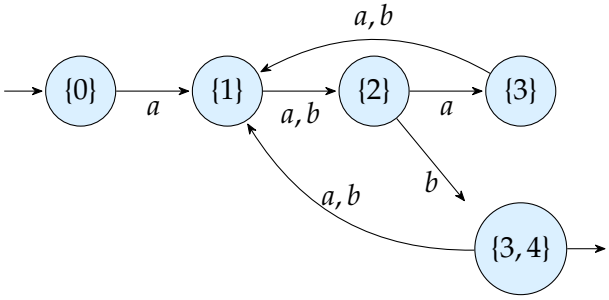
the language contenant les mots dont la longueur est divisible par 3 est recognized par an automaton à trois états, chaque state corresponding à un reste de la division par 3. De 0 on transite vers 1 (indépendamment de l'entrée); de 1 on transite vers 2, et de 2 vers 0, qui est à la fois initial et final.

the language $a\Sigma^*b$ est recognized par an automaton à 3 états. L'état initial a une unique transition sortante sur le symbol d'entrée a vers l'état 1, dans lequel on peut soit boucler (sur a ou b) ou bien transiter dans 2 (sur b). 2 est le seul final state.

L'intersection de ces deux machines conduit à A_1, l'automate 12.9 non-deterministic, qui correspond formellement au produit des automata intersectés. L'application de la méthode des sous-ensembles conduit à A_2, l'automate 12.10, deterministic.



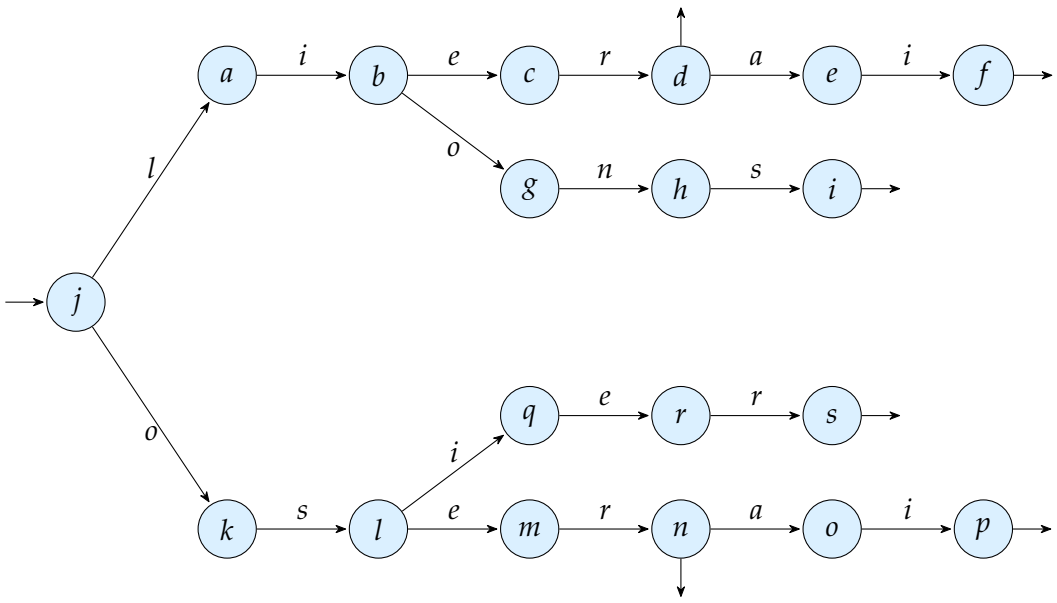
Automaton 12.9 – The Automaton A_1 de la question 1



Automaton 12.10 – The Automaton A_2 de la question 1

12.7 Correction de l'exercise 4.35

1. En utilisant les méthodes du cours, on obtient l'automate 12.11, déterminisé. Cet automaton est appelé *l'arbre accepteur des préfixes* (il comprend exactly un state par préfixe dans le language).
2. (a) Les états (f) and (p) sont two états final sans transition sortante: ils sont indistinguishable; l'état (f) est final quand (e) est non-final: le mot ε permet de les distinguer (i.e. qu'il conduit à un successful computation depuis (f) mais pas depuis (e)).



Automaton 12.11 – Le dictionary déterminisé

(b) Notons $\mathcal{I}$ la indistinguishability relation: cette relation est trivialement réflexive and symétrique. Soient p and q indistinguishable and q and r indistinguishable: supposons que p et r soient distingués par le mot v. On a alors un successful computation depuis p qui échoue depuis r ; ceci est absurde: s'il réussit depuis p il réussira depuis q, and réussira donc également depuis r.

3. La combinaison de (f) and (p) conduit à an automaton dans lequel ces two états sont remplacés par un unique state (fp), qui est final. (fp) possède two transitions entrantes, toutes two étiquetées par i; (fp) n'a aucune transition sortante.

4. Supposons que A' résulte de la fusion de p and q indistinguishable dans A, qui sont remplacés par r. Il est tout of abord clair que $L(A) \subset L(A')$: soit en effet u dans $L(A)$, il existe un successful computation pour u dans A et:

   — soit ce computation évite les états p and q and le même computation existe dans A'; with les notations de l'énoncé, il en va de même si ce computation évite l'état q.

   — soit ce computation utilise au moins une fois q: $(q_0, u) \vdash_A (q, v) \vdash_A (s, \varepsilon)$. Par construction de A' on a $(q_0, u) \vdash_{A'} (r, v)$; par ailleurs p and q étant indistinguable, il existe un computation $(p, v) \vdash_A (s', \varepsilon)$ with $s' \in F$ dans A qui continue of exister dans A' (au renommage de p en r près). En combinant ces two résultats, on exhibe un computation réussi pour u dans A'.

   Supposons maintenant que $L(A)$ soit strictement inclus dans $L(A')$ et que le mot v de $L(A')$ ne soit pas dans $L(A)$. Un successful computation de v dans A' utilise necessarily le nouvel state r, sinon ce computation existerait aussi dans A; on peut même supposer qu'il utilise une fois exactly r (s'il utilise plusieurs fois r, on peut court-circuiter les boucles).

On a donc: $(q'_0, v = v_1v_2) \vdash_{A'} (r, v_2) \vdash_{A'} (s, \varepsilon)$, with $s \in F'$. Dans A, ce computation devient soit $(q_0, v_1) \vdash_A p$ suivi de $(q, v_2) \vdash_A (s, \varepsilon)$; soit $(q_0, v_1) \vdash_A q$ suivi de $(p, v_2) \vdash_A (s, \varepsilon)$; dans la première de ces alternatives, il est de plus assuré que, comme v n'est pas dans A, il n'existe pas de computation (p, v_2) aboutissant dans un final state. Ceci contredit toutefois le fait que p and q sont indistinguishable.

5. Le computation de la indistinguishability relation demande de la méthode: dans la mesure où the automaton est sans cycle, ce computation peut s'effectuer en considérant la longueur du plus court chemin aboutissant dans un état final.

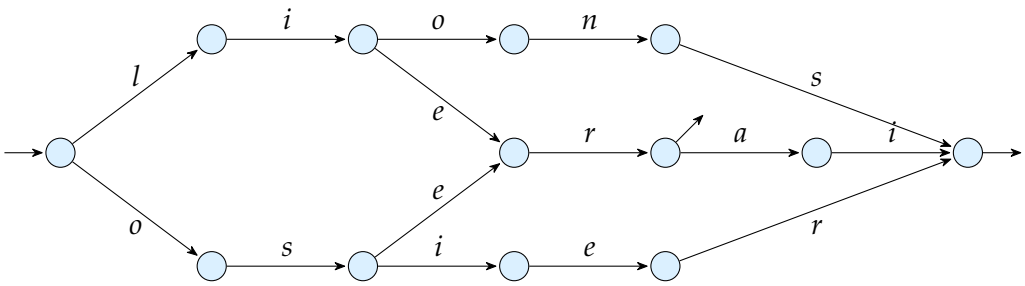
   Commençons par l'initialization: $\{d, f, i, n, s, p\}$ sont distingués de tous les autres états par leur "finalitude". On distingue two sous-sets: $\{f, i, s, p\}$ sont distingués de d and de n par le mot ai, mais sont indistinguishable between eux. À ce stade, on ne peut conclure sur d and n.

   Les états desquels débute un successful computation de longueur 1 sont: c, e, h, r, m, o: c and m sont distingués des autres par le mot rai, r and h sont distingués du couple (e, o) par respectively s et r, and sont aussi distingués between eux ; e and o sont indistinguishable.

   Les états depuis lesquels débutent un successful computation de longueur 2 sont d, g, q, n. g and q sont distingués des two autres, and distingués between eux. En revanche, d and n sont indistinguishable.

   En poursuivant ce raisonnement, on aboutit finalement aux classes of equivalence suivantes: $\{f, i, p, s\}$, $\{e, o\}$, $\{d, n\}$, $\{m, c\}$, $\{a\}$, $\{b\}$, $\{g\}$, $\{h\}$, $\{j\}$, $\{k\}$, $\{l\}$, $\{q\}$, $\{r\}$.

6. Après réalisation des fusions, on obtient l'automate 12.12.



On conçoit aisément que pour un dictionary du français, ce principe de "factorisation" des suffixes conduise à des réductions très sensibles du nombre of état de the automaton: chaque verbe du français (environ 10 000) possède une cinquantaine de formes différentes; parmi les verbes, les verbes du premier groupe sont de loin les plus nombreux and ont des suffixes très réguliers qui vont "naturellement" se factoriser.

Automaton 12.12 – Le dictionary déterminisé and minimisé

12.8 Correction de l'exercise 5.21

2 L'application de la méthode de transformation d'automate en grammaire présentée en section 5.2.4 conduit à la grammar 12.13, ou bien encore à la grammar 12.14 si l'on part de l'automate déterministe.

$$
\begin{aligned}
S &\rightarrow A \mid bA \\
A &\rightarrow aA \mid aB \\
B &\rightarrow bB \mid \varepsilon
\end{aligned}
$$

Grammar 12.13 – Grammaire régulière pour l'automate (non-déterministe)

$$
\begin{aligned}
S &\rightarrow aA \mid bB \\
A &\rightarrow aA \mid bC \\
B &\rightarrow aA \\
C &\rightarrow bC \mid \varepsilon
\end{aligned}
$$

Grammar 12.14 – Grammaire régulière pour l'automate (déterministe)

Part III

Références

List of Algorithms

Chapter 13

Liste des automates

Chapter 14

Liste des grammaires

Chapter 15

List of Tables

Chapter 16

List of Figures

Chapter 17

Bibliographie

Barsky, R. F. (1997). *Noam Chomsky: A Life of Dissent*. MIT Press. Biographie de Noam Chomsky, également disponible en ligne sur http://cognet.mit.edu/library/books/chomsky/chomsky/. 11.2

Conway, M. E. (1963). Design of a separable transition-diagram compiler. *Commun. ACM*, 6:396–408. 8.2.4

Dowek, G. (2007). *Les métamorphoses du calcul. Une étonnante histoire des mathématiques*. Le Pommier. L'histoire du calcul et du raisonnement, des mathématiciens grecs aux progès récents en preuve automatique. Grand prix de philosophie 2007 de l'Académie française. 11.2

Doxiadis, A. et Papadimitriou, C. H. (2010). *Logicomix. La folle quête de la vérité scientifique absolue*. Vuibert. Bande dessinée romançant les découvertes en logique mathématique du début du 20e siecle. 11.2

Grune, D. et Jacob, C. J. (1990). *Parsing Techniques: a practical Guide*. Ellis Horwood. 7

Hodges, A. (1992). *Alan Turing. The Enigma*. Random House. Biographie d'Alan Turing, traduit en français sous le nom de *Alan Turing ou l'énigme de l'intelligence*. 11.2

Hopcroft, J. E. et Ullman, J. D. (1979). *Introduction to automata theory, languages and computation*. Addison-Wesley. 4

Knuth, D. E. (1965). On the translation of languages from left to right. 8.2.4

Lewis, II, P. M. et Stearns, R. E. (1968). Syntax-directed transduction. *J. ACM*, 15:465–488. 8.2.4

Perrin, D. (1995). Les débuts de la théorie des automates. *Technique et science informatique*, 14(4). Un historique de l'émergence de la théorie des automates. 11.2

Sakarovitch, J. (2003). *Eléments de théorie des automates*. Vuibert, Paris. 4

Salomaa, A. (1973). *Formal Languages*. Academic Press. 9.2.2

Sudkamp, T. A. (1997). *Languages and Machines*. Addison-Wesley. 4

Turing, A. M. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42). L'article introduisant la notion de machine de Turing, un des articles fondateurs de la science informatique. Intéressant à la fois par son contenu et par son aspect historique. 11.2

Chapter 18

Index