Formal language and parsing

Lecture notes

SL5AE011

Timothée Bernard

Note to the reader

This document contains the lecture notes of the *Formal grammar and parsing* course taught at the M1 level of the LI program of Université Paris Cité[1] (code `SL5AE011` since 2021-22).

These notes draw heavily on the notes of Yvon and Demaille (2016).

[1]Previously 'Université de Paris'.

Contents

Introduction – Entering the matrix

- We all use language on a daily basis without even thinking about it. Learning a language as a child is a piece of cake; as an adult, it is clearly harder but still doable (given enough time, anyone can do it). Language is intuitively simple because the brain is an astonishing machine and because human beings are particularly well adapted for this type of communication.

- This simplicity hides a lot of complexity. Natural languages are complex in many ways, and we still understand quite little about them. But there are a few things that we do understand. With this course, we are about to enter the matrix in the sense that what we are about to study will show you some of the mathematics behind language. More precisely, we will study some of the computational aspects of syntax.

- Syntax concerns the structure of utterances. The goal of syntax is to explain which sequences of words, and why, are 'acceptable' in the sense that *The cat is chasing the mouse* is acceptable but neither *Mouse the chasing is cat the* nor *The cat is chasing the* are. (*The mouse is chasing the cat*, although unusual is some ways, is also acceptable in this sense.) Syntax tries to solve this problem in terms of structure. Only some sequences of words, in English, can be associated with a valid structure (*The cat is chasing the mouse*, for instance, is usually described as composed of a noun phrase, *The cat*, followed by a verb phrase, *is chasing the mouse*, which can be further decomposed), while others cannot.

- What we have just said about natural languages, such as English, is also true of other languages. Not all sequences of bits (0 or 1), for instance, form a valid MP3 file. Create a new text file, write anything inside, then rename it ending with the '.mp3' extension and try to open it with a music player. It is extremely unlikely that the music player be able to read your so-called MP3 file. Rename the file ending with the '.zip' extension and try to unzip it. Or rename it ending with the '.py' extension and try to execute it with a Python interpreter. These operations are likely to fail as well, because all of these types of file have a syntax, with which your file probably does not comply.

- The set of valid MP3 files, the set of valid zip files and the set of Python programs, are not natural languages but are examples of formal languages.

- This course mainly deals with two related topics:
  - what formal languages are and what their relation with natural languages is;
  - how to automatically analyse the syntactic structure of languages, formal or natural.

- Formal languages are mathematical objects; they are defined as sets (possibly infinite) of finite sequences of elementary units.

- These units are usually called ‘letters’ forming sequences called ‘words’ but these are merely technical terms that can be applied to objects of very different nature. For examples:
  - The set of all English words, if defined mathematically, can be seen as a formal language. One way to do this is to see each English word as a sequence of letters of the English alphabet. The words of the formal language thus defined are the English words and its letters are the characters of the English alphabet.
  - The set of all possible English sentences, if defined mathematically, can be seen as a formal language. One way to do this is to see each sentence as a sequence of English words and punctuation marks.[2] The words of the formal language thus defined are the possible English sentences and its letters are the English words and the punctuation marks.
  - The set of all possible MP3 files, if defined mathematically, can be seen as a formal language. One way to do this is to see each possible MP3 file as a sequence of bits (0 or 1). The words of the formal language thus defined are the possible MP3 files and its letters are the two bits (0 and 1).
- What is important in the context of a formal language, is what counts as a valid sequence of elementary units or not — and not what these units are.
- Is it possible to mathematically define the set of all possible English sentences? Defining such a set amounts to determining what counts as an English sentence. This is actually quite hard.
- While some sentences are fairly uncontroversial (as already mentioned, ‘The cat is chasing the mouse’ is an English sentence; ‘Mouse the chasing is cat the’ is not), some sentences are grammatical for some native speakers but not for others (I think that not everyone agrees on whether ‘The band are playing a song.’ is correct or not). If language is defined not at the level of the individual speaker but at the level of the community of speakers, then grammaticality does not seem to be categorical.
- In fact, even for a single native speaker, there exists a gradience in their grammaticality judgments — not in the sense that a native speaker might have trouble judging whether a finite sequence of words is grammatical or not (even though that might also be the case), but in the sense that they can judge it to be neither clearly correct nor clearly incorrect.
- It appears that in nature, grammaticality is a continuous concept, while the theory of formal languages is based on the notion of set, built around a binary relationship: A given sequence is either a valid word of the language or it is not. This means that one cannot apply the theory of formal languages to natural languages directly.
- It thus happens that we have to make some approximation in order to be able to apply our formal concepts to the natural or social phenomena that we are interested in studying. This is in fact always the case in science.[3] In science, one usually tries to make the approximation that leads to the most fruitful results. The quality of these results (in terms of explanation and/or prediction) is what justifies the approximation.

[2]‘English word’ in this text applies to any word that can be used in an English sentence. Because the elementary units that serve as letters of a formal language are, by definition, in finite number, we here need to assume that there is a finite number of English words and in particular a finite number of proper nouns.

[3]At least in the natural and social sciences.

- Let us assume that grammaticality can be binarised. A natural language such as English can now be thought of as a formal language. One of the goals of this course is to introduce you to the most interesting results that spring from this approximation. In particular, we will see how the theory of formal languages can characterise the syntactic complexity of human (written/spoken) language.

- The syntactic complexity of a language expresses how hard it is to describe this language. We will see that there are two main computational ways to describe a language: using an automaton and using a formal grammar. In terms of automata, syntactic complexity pertains to the type of memory that a machine must possess in order to be able to determine whether a sequence of symbols belongs to the language or not. This is quite different from the more common notions of worst-case time and space complexity applied to the same problem (determining whether a sequence of symbols belongs to the language or not). In terms of formal grammars, syntactic complexity pertains to the type of syntactic rules that a grammar must include in order to generate all and only the sequences of symbols that belong to the language.

- The other major goal of this course is to present the theoretical basis of the syntactic parsing of formal and natural languages. In other words, we are interested in algorithms able to compute whether a sentence is valid and if so, able to reconstruct the syntactic structure(s) of this sentence.

- To execute a piece of code written in any currently defined programming language (whether compiled or interpreted), a computer first needs to tokenise it (i.e. turn it into a sequence of tokens such as `def`, `if`, `my_var`, rather than a sequence of characters) and then to parse it, before compiling or interpreting the resulting parse tree. Parsing — and, usually, parsing efficiently — has thus been crucial to computer programming, until now and probably still for a very long time as the foundation of computer programming is unlikely to change in this respect.

- Syntactic parsing also matters to natural language processing (NLP). Historically, syntactic parsing has been considered one of the key steps for most advanced NLP tasks, such as coreference resolution and machine translation. While a large part of the research in NLP has shifted focus during the mid-2010s towards the development of purely statistical systems that do not use any intermediate symbolic linguistic representations, traditional models based on symbolic methods are still relevant and widely used in the industry. Indeed, such models tend to be more interpretable, easier to debug, improve and customise, and can be much less computationally expensive. Furthermore, in many situations (for example involving text with relatively limited variability, that depends on fine linguistic phenomena, or in settings with limited data available), symbolic, linguistically-informed, processing can still be the most effective option.

- Finally, formal language theory and syntactic parsing also play a role in the broader field of linguistic, well beyond the question of the syntactic complexity of natural language. Formal grammars for various natural languages are being developed in order to validate theoretical models, and the ability to parse text according to these grammars allows researcher to investigate and test linguistic hypotheses using an approach that is both theoretically and empirically grounded.

- While we will mainly study *grammar-based* parsing algorithms, which require the explicit definition of a formal grammar (that can be built manually or inferred algorithmically from corpora of syntactically annotated data), we will also mention alternative paradigms now popular in NLP: the paradigms of *classifier-based* and *graph-based* parsers.

Relevant reading

- The introduction of *The Formal Complexity of Natural Language* by Savitch et al. (1987).
- A short blog post entitled ‘Phases and Phrases: Some thoughts on Weak Equivalence, Strong Equivalence, and Empirical Coverage’ by Blix and Williams (2022).

Chapter 1

Fundamentals

1.1 Sets

Remark 1.1 (About sets) *Formally defining what a set is requires stating set theory.*[1] *Doing so is beyond the scope of this text. However, an informal yet precise description of the notion of a set will suffice.*

Intuition 1.1 (Set) *A* set *is a mathematical entity that is fully characterised by its* members *(or 'elements').*

Example(s) 1.1 (Empty set) *Because sets are fully characterised by their members, there can only be at most one set with no member. There is indeed a set with no member, the* empty set*, written '$\emptyset$' (or, alternatively, '$\{\}$').*

Remark 1.2 *Sets can be finite (i.e. have a finite number of members) or infinite.*

Example(s) 1.2

1. *There is a set the members of which are exactly the natural numbers (0, 1, 2, etc.), written '$\mathbb{N}$'. This set is infinite.*
2. *There is a set the members of which are exactly the (strictly) positive natural numbers (1, 2, 3, etc.), written '$\mathbb{N}^*$'. This set is infinite.*
3. *There is a set the members of which are exactly the relative numbers (0, 1, -1, 2, -2, etc.), written '$\mathbb{Z}$'. This set is infinite.*

Notation 1.1 (Set membership) *The fact that e is a member of set S is written '$e \in S$'. The opposite fact is written '$e \notin S$'.*

Example(s) 1.3 $0 \in \mathbb{N}$, $123456 \in \mathbb{N}$, $-1 \notin \mathbb{N}$, $\mathbb{N} \notin \mathbb{N}$.

Notation 1.2 (Extensional notation of a set) *For any $n \in \mathbb{N}$ and any e_1, e_2, ..., e_n, the set the members of which are exactly e_1, e_2, ..., e_n can be written '$\{e_1, e_2 \cdots, e_n\}$'.*

Property 1.1 *The members of a set are not ordered in the set (as opposed to as, say, in a sequence).*

[1]The way sets are defined by set theory is, in a sense, indirect. Set theory contains axioms stating the properties of some entities called 'sets', and axioms stating the existence (in an abstract, mathematical sense) of some of these entities. These axioms taken together are exactly what defines the sets of set theory.

Example(s) 1.4 *For any x, y and z, $\{x, y, z\}$, $\{y, x, z\}$ and $\{y, z, x\}$ are the same set. (In other words, '$\{x, y, z\}$', '$\{y, x, z\}$' and '$\{y, z, x\}$' are three different notations for the same entity.)*

Property 1.2 *The members of a set have no multiplicity in the set (as opposed to as, say, in a sequence).*

Example(s) 1.5 $\{3, 5\}$, $\{3, 5, 5\}$ *and* $\{3, 5, 5, 3, 3\}$ *are the same set.*

Notation 1.3 (Intensional notation of a set) *For any set S and any property P, the set the members of which are exactly the members of S that have the property P can be written '$\{x \in S \mid P(x)\}$'.*

In a context in which S is obvious, this notation can be simplified as '$\{x \mid P(x)\}$'.

Example(s) 1.6

1. $\{x \in \mathbb{N} \mid x \textit{ is prime}\}$ *is the set of all prime numbers.*
2. $\{x \in \mathbb{N} \mid x \neq 0\}$ *is* $\mathbb{N}^*$.
3. $\{x \in \mathbb{Z} \mid x \geq 0\}$ *is* $\mathbb{N}$.
4. $\{x \in \mathbb{N} \mid x \notin \mathbb{N}\}$ *is* $\emptyset$.
5. $\{x \in \{-2, -1, 0, 1, 2\} \mid x^2 = 1\}$ *is* $\{-1, 1\}$.
6. $\{x \in \{\textit{cat, dog, camel}\} \mid x \textit{ starts with a 'c'}\}$ *is* $\{\textit{cat, camel}\}$.

Remark 1.3 *The intensional notation of sets can be extended in a way that will not be formally defined in this text but that can be intuitively grasped thanks to the following examples:*

1. $\{2n \mid n \in \mathbb{N}\}$ *is* $\{k \in \mathbb{N} \mid \exists n \in \mathbb{N},\ k = 2n\}$, *the set of all even numbers.*
2. $\{3n + 2m \mid n, m \in \mathbb{N}\}$ *is* $\{k \in \mathbb{N} \mid \exists n, m \in \mathbb{N},\ k = 3n + 2m\}$, *a set that turns out to be the set of all natural numbers except 1.*

Definition 1.1 (Inclusion) *A set S_1 is* included *in a set S_2 (alternatively, S_1 is a* subset *of S_2), written '$S_1 \subseteq S_2$', if and only if ('iff') all members of S_1 are also members of S_2. The fact that set S_1 is not included in set S_2 is written '$S_1 \nsubseteq S_2$'.*

Example(s) 1.7

1. $\{2, 3, 5\} \subseteq \mathbb{N}$
2. *For any set S, $S \subseteq S$ and $\emptyset \subseteq S$.*
3. *The set of English words starting with 'fo' is a subset of the set of English words starting with 'f'.*
4. $\mathbb{N} \subseteq \mathbb{Z}$
5. $\mathbb{Z} \nsubseteq \mathbb{N}$

Remark 1.4 *A set S_1 might be a* member *of a set S_2 ($S_1 \in S_2$), which is very different from S_1 being a* subset *of it ($S_1 \subseteq S_2$).*

Example(s) 1.8

1. $\{0\} \in \{\{0\},\{0,1\},\{0,1,2\}\}$ *but* $\{0\} \not\subseteq \{\{0\},\{0,1\},\{0,1,2\}\}$ *(because* $0 \notin \{\{0\},\{0,1\},\{0,1,2\}\}$*).*
2. $\emptyset \notin \{a,b\}$ *but* $\emptyset \subseteq \{a,b\}$.

Remark 1.5 *Even though membership and inclusion are different relations, a set S_1 might both be a member and a subset of a set S_2.*

Example(s) 1.9

1. $\{0\} \in \{0,\{0\}\}$ *and* $\{0\} \subseteq \{0,\{0\}\}$.
2. $\emptyset \in \{\emptyset, a\}$ *and* $\emptyset \subseteq \{\emptyset, a\}$.

Definition 1.2 (Power set) *Given a set S, the* power set of S, *written '$\mathcal{P}(S)$', is the set the members of which are exactly the subsets of S.*

Example(s) 1.10

1. $\mathcal{P}(\{0,1\}) = \{\emptyset,\{0\},\{1\},\{0,1\}\}$
2. $\mathcal{P}(\{x,y,z\}) = \{\emptyset,\{x\},\{y\},\{z\},\{x,y\},\{x,z\},\{y,z\},\{x,y,z\}\}$
3. $\mathcal{P}(\emptyset) = \{\emptyset\}$

Property 1.3 *For any set S, $\emptyset$ and S are both members of $\mathcal{P}(S)$.*

Notation 1.4 (Interval of integers) *For any two numbers $m, n \in \mathbb{Z}$, $\{i \in \mathbb{Z} \mid m \leq i \leq n\}$, the set of all numbers that are both (i) larger or equal to m and (ii) smaller or equal to n, can be written '$[\![m,n]\!]$'.*

Example(s) 1.11

1. $[\![1,4]\!] = \{1,2,3,4\}$
2. $[\![-1,1]\!] = \{-1,0,1\}$
3. $[\![3,3]\!] = \{3\}$
4. $[\![4,1]\!] = \emptyset$

Intuition 1.2 (Cardinal of a set) *Each set S has a unique cardinal, written '$|S|$'.*

- *If S is finite, $|S|$ is its number of elements (it is thus a natural number).*
- *If S is infinite, $|S|$ is also an entity that can intuitively be interpreted as the (infinite) size of S.*

Cardinal numbers are a generalisation of the concept of quantity from natural numbers to infinite quantities.

Example(s) 1.12

1. $|\{2, 3, 5, 7\}| = 4$
2. $|\{12\}| = 1$
3. $|\emptyset| = 0$

4. *$|\mathbb{N}|$ is an infinite cardinal larger than any natural number.*

5. *For any finite set S, $|\mathcal{P}(S)| = 2^{|S|}$.*

Advanced remark 1.1 *$|\mathbb{N}|$ is often written '$\aleph_0$'. It is the smallest infinite cardinal, and this entails that any infinite subset of $\mathbb{N}$ is also of cardinality $\aleph_0$. For instance, $|\{n \in \mathbb{N} \mid n \text{ is prime}\}|$, $|\{n \in \mathbb{N} \mid n \text{ is odd}\}|$, $|\{n \in \mathbb{N} \mid n \text{ is even}\}|$, and $|\mathbb{N}|$, are all $\aleph_0$. $\mathbb{Z}$, which properly includes $\mathbb{N}$, is also of cardinality $\aleph_0$. For two sets S_1 and S_2, $S_1 \subseteq S_2$ and $S_1 \neq S_2$ together do not entail $|S_1| < |S_2|$ but only $|S_1| \leq |S_2|$. These facts can be derived from the formal definition of cardinality (not given here), which is not based on the notion of inclusion. It can be proven that a set S is of cardinality $\aleph_0$ iff there exists an exhaustive ordering of its elements by $\mathbb{N}$; i.e. iff $S = \{e_0, e_1, e_2, \dots\}$ (where, for any $i, j \in \mathbb{N}$ with $i \neq j$, $e_i \neq e_j$).*

Definition 1.3 (Union of sets) *Given two sets S_1 and S_2, the* union *of S_1 and S_2, written '$S_1 \cup S_2$', is the set that has for members all members of S_1, all members of S_2, and nothing else.*

This definition is naturally extended to any number of sets.

Example(s) 1.13 $\{2,3,5,7\} \cup \{0,2,4,6\}$ *is* $\{0,2,3,4,5,6,7\}$.

Remark 1.6 *The union of a collection of sets S_1, S_2, $\cdots$ is the smallest set (in the sense of inclusion) that includes each of them. In other words, there is only one set that both (i) includes each of S_1, S_2, $\cdots$ and (ii) is included in $S_1 \cup S_2 \cup \cdots$ (i.e. their union): this set is $S_1 \cup S_2 \cup \cdots$ itself.*

Remark 1.7 (Union and the empty set) *The empty set, $\emptyset$, is the neutral element of the union operation. In other words,*

1. *the union of any number of sets is the same as the union of only those sets that are non-empty;*

2. *the union of a single set is conventionally defined to be this set;*

3. *the union of zero set is conventionally defined to be the empty set.*

Example(s) 1.14

1. $\{11,7\} \cup \emptyset = \{11,7\}$.

2. $\{11,7\} \cup \emptyset \cup \emptyset \cup \{11,13\} \cup \{17\} \cup \emptyset = \{11,7,13,17\}$.

Definition 1.4 (Intersection of sets) *Given two sets S_1 and S_2, the* intersection *of S_1 and S_2, written '$S_1 \cap S_2$', is the set the members of which are exactly the members of S_1 which are also members are S_2.*

This definition is naturally extended to any number of sets.

Example(s) 1.15 $\{2,3,5,7\} \cap \{0,2,4,6\}$ *is* $\{2\}$.

Remark 1.8 *Given two sets S_1 and S_2, $S_1 \cap S_2 = \{x \in S_1 \mid x \in S_2\}$.*

Definition 1.5 (Set complementation) *Given two sets S_1 and S_2, the* complement *of S_1 in S_2, written '$S_2 \setminus S_1$', is the set the members of which are all members of S_2 that are not members of S_1.*

In a context in which S_2 is obvious, the following notation can be used instead: '$\overline{S_1}$'.

Example(s) 1.16 $\{2,3,5,7\} \setminus \{0,2,4,6\}$ *is* $\{3,5,7\}$.

Remark 1.9 *Given two sets* S_1 *and* S_2*,* $S_2 \setminus S_1 = \{x \in S_2 \mid x \notin S_1\}$.

Intuition 1.3 (Tuple) *A* tuple *is any finite sequence.*

Example(s) 1.17

1. $(3,4)$ *and* $(4,3)$ *are two distinct tuples of size two (*pairs*).*
2. $(3,3,4)$ *is a tuple of size three (a* triple*) and is distinct from* $(3,4)$.
3. $()$ *is the only tuple of size zero.*

Definition 1.6 (Cartesian product) *The* Cartesian product *of two sets* S_1 *and* S_2 *is the set of all pairs the first member of which is a member of* S_1 *and the second member of which is a member of* S_2*:* $S_1 \times S_2 = \{(x,y) \mid x \in S_1, y \in S_2\}$.

This definition is naturally extended to any finite number of sets.

Example(s) 1.18

1. $\{Sun, Moon, Earth\} \times \{0,1\} = \{(Sun,0),(Sun,1),(Moon,0),(Moon,1),(Earth,0),$ $(Earth,1)\}$
2. $\{0,1\} \times \{Sun, Moon, Earth\} = \{(0,Sun),(0,Moon),(0,Earth),(1,Sun),(1,Moon),$ $(1,Earth)\}$

Remark 1.10 *For any set* S*,* $S \times \emptyset = \emptyset \times S = \emptyset$.

Remark 1.11 (Non-commutativity of the Cartesian product) *As can be seen from example 1.18, the Cartesian product operation is not* commutative*, which means that there are two sets* S_1 *and* S_2 *such that* $S_1 \times S_2 \neq S_2 \times S_1$.

In fact, for any two non-empty and distinct sets S_1 *and* S_2*,* $S_1 \times S_2 \neq S_2 \times S_1$.

1.2 Functions

Intuition 1.4 (Function) *Given two sets* A *and* B*, a* function from A to B *is a mathematical entity that is fully characterised by the association of a member of* B *to each member* A.

Example(s) 1.19 *There is a function that is the function from* $\mathbb{N}$ *to* $\mathbb{N}$ *that, for each* $n \in \mathbb{N}$*, associates* $n+1$ *to* n.

Example(s) 1.20 *Let* W *be a set of English words, and* C *be the set of all characters used to write the words in* W*. There is a function that is the function from* W *to* C *that, for each* $w \in W$*, associates the first character in* w *to* w.

Notation 1.5 *Let* A *and* B *be two sets,* f *a function from* A *to* B*,* $a \in A$*, and* $b \in B$*. The entity associated by* f *to* a *can be written '*$f(a)$*'. Accordingly, the fact that* f *associates* b *to* a *can be written '*$f(a) = b$*'. When* $f(a) = b$*, it is also said that* f sends a *to* b.

Example(s) 1.21 *Let* f *be the function mentioned in example 1.19.* $f(0) = 1$ *(*f *sends* 0 *to* 1*),* $f(1) = 2$ *(*f *sends* 1 *to* 2*) and, more generally, for any* $n \in \mathbb{N}$*,* $f(n) = n+1$ *(*f *sends* n *to* $n+1$*).*

Remark 1.12 *A function from A to B may send two (or more) distinct members of A to the same member of B.*

Example(s) 1.22 *Let f be the function from $\mathbb{Z}$ to $\mathbb{N}$ that sends any relative number to its absolute value.*[2] *f sends, for instance, both 3 and -3 to the same natural number 3.*

Definition 1.7 (Domain and image) *Let f be a function from set A to set B, and $a \in A$. The* domain of f *is A and the* image of a by f *is $f(a)$.*

Example(s) 1.23

1. *Let f be the function mentioned in example 1.19. The domain of f is $\mathbb{N}$ and the image of 3 by f is 4.*
2. *Let f be the function mentioned in example 1.22. The domain of f is $\mathbb{Z}$ and the image of -3 by f is 3.*

Definition 1.8 (Injection) *Given two sets A and B, an* injection from A to B *is a function f from A to B such that no two distinct elements of A have the same image by f. In other words, an injection sends each element of A to a distinct element of B.*

Example(s) 1.24 *The function f mentioned in example 1.19 is an injection. Indeed, for any $n, m \in \mathbb{N}$ such that $n \neq m$, because $f(n) = n + 1$ and $f(m) = m + 1$, $f(n) \neq f(m)$.*

Notation 1.6 *Given two sets A and B, and a function f from A to B, the fact that f is a function from A to B can be written '$f : A \to B$'.*

Notation 1.7 *Given two sets A and B, and $f : A \to B$, f can also be written '$x \in A \mapsto f(x)$'. Accordingly, the fact that f sends any $x \in A$ to $f(x)$ can be written '$f = x \in A \mapsto f(x)$'.*

Example(s) 1.25

1. *Let f be the function mentioned in example 1.19. $f = n \in \mathbb{N} \mapsto n + 1$.*
2. *Let f be the function mentioned in example 1.22. $f = n \in \mathbb{Z} \mapsto \begin{cases} n \text{ if } n \geq 0 \\ -n \text{ otherwise} \end{cases}$.*

Remark 1.13 *Let A and B be two sets, and $f : A \to B$. While f sends each $a \in A$ to a* single *member of B (namely, $f(a)$), this value might be a set — if B contains sets as members.*

Example(s) 1.26 *Let $f = n \in \mathbb{N} \mapsto \{q \in \mathbb{N} \mid q \text{ is a divisor of } n\}$; f is a function from $\mathbb{N}$ to $\mathcal{P}(\mathbb{N})$. $f(9) = \{1, 3, 9\}$ and $f(12) = \{1, 2, 3, 4, 6, 12\}$.*

Definition 1.9 (Partial function) *Given two sets A and B, a* partial function from A to B *is a function from some subset of A to B.*

Example(s) 1.27 *The function defined on the set of square numbers (i.e. $\{n \in \mathbb{N} \mid \exists k \in \mathbb{N},\ n = k^2\}$) which associates its square root to any such number is a partial function from $\mathbb{N}$ to $\mathbb{N}$. This partial function can be written as the following: $n \in \mathbb{N} \mapsto \sqrt{n}$ if $\exists k \in \mathbb{N},\ n = k^2$. This function is undefined on all natural numbers that are not square numbers.*

Remark 1.14 *According to the above definitions, all functions from some set A to some set B are partial functions from A to B, while not all partial functions from some set A to some set B are functions from A to B.*

[2]Consider $n \in \mathbb{Z}$, if $n \geq 0$, then its *absolute value* is n itself, otherwise it is the opposite of n (i.e. $-n$).

1.3 Words

Definition 1.10 (Alphabet and letter) *In this text, an* alphabet *is any finite set. When considering an alphabet, a* letter *is any member of this set.*

Example(s) 1.28

1. $\{a, b, c\}$ *is an alphabet the letters of which are* a, b *and* c.
2. $\{., did, It, rain\}$ *is an alphabet the letters of which are ., did, It and rain.*

Definition 1.11 (Word) *A (finite) word on an alphabet* Σ *is any (finite) sequence of letters. If defined, the* n^{th} *letter of a word* w *is written '*w_n*'.*

Remark 1.15 (About words) *While words can be infinite, this course only deals with finite ones, i.e. finite sequences of letters. Unless otherwise stated (e.g. in 'infinite word'), in this text, 'word' should be understood as 'finite word'.*

Example(s) 1.29 *(Here and in many other examples, formal words are written with a space between adjacent letters.)*

1. $a\,a\,c\,a$ *is a word on the alphabet* $\{a, b, c\}$.
2. *It did rain .* *is a word on the alphabet* $\{., did, It, rain\}$. *In this alphabet, did is a letter.*

Remark 1.16 *The notions of an alphabet, a letter and a word are not really new concepts, but are used to define a level of granularity, i.e. to specify which are the objects that are considered atomic.*

Example(s) 1.30

1. *The same English word ('cats') can be seen as a word on the English alphabet (c a t s), but also as a word on the set of all morphemes used in English (cat s).*
2. *The same English sentence ('It did rain.') can be seen as a (formal) word on an alphabet made of all of English words and punctuation marks (It did rain .) or as a word on an alphabet made of all characters used in English, including the space (I t d i $\cdots$).*

Definition 1.12 (Empty word) *The* empty word*, written '*ϵ*', is the empty sequence.*

Notation 1.8 *The set of all words on an alphabet* Σ *is written '*$\Sigma^\star$*'.*

Example(s) 1.31 $\{0, 1\}^\star = \{\epsilon, 0, 1, 0\,0, 0\,1, 1\,0, \cdots\}$

Definition 1.13 (Length of a word) *The* length *of a word* w *is its length as a sequence of letters. It is written '*$|w|$*'.*

Example(s) 1.32

1. $|\epsilon| = 0$
2. $|a\,a\,c\,a| = 4$

Notation 1.9 *Given a word* w *of length* n*, for* $i \in [\![1, n]\!]$*, the* i^{th} *letter of* w *is written '*w_i*'.*

Example(s) 1.33 *Given* $w = a\,a\,c\,a$, $w_1 = w_2 = w_4 = a$ *and* $w_3 = c$.

Definition 1.14 (Span) *Given a word w of length n, for $i \in [\![1, n]\!]$ and $j \in [\![i, n]\!]$, the* span *(i, j) of w, written '$w_{i:j}$', is the word $w_i\,w_{i+1}\,\cdots\,w_j$. This definition is extended to $i \in [\![1, n+1]\!]$ and $j = i - 1$ with $w_{i:(i-1)} = \epsilon$.*

Example(s) 1.34 *Let $w = a\,b\,a\,c$.*

- $w_{1:0} = \epsilon$
- $w_{1:1} = a$
- $w_{1:2} = a\,b$
- $w_{1:3} = a\,b\,a$
- $w_{1:4} = a\,b\,a\,c$
- $w_{2:1} = \epsilon$
- $w_{2:2} = b$
- $w_{2:3} = b\,a$
- $w_{2:4} = b\,a\,c$

Notation 1.10 *Given a word w, the number of occurrences of any letter a in w is written '$|w|_a$'.*

Example(s) 1.35

1. $|\epsilon|_a = 0$
2. $|a\,a\,c\,a|_a = 3$
3. $|a\,a\,c\,a|_b = 0$
4. $|a\,a\,c\,a|_c = 1$

Remark 1.17 *Given a word w and a letter a, $|w|_a = |\{i \in [\![1, |w|]\!] \mid w_i = a\}|$.*

Definition 1.15 (Concatenation of words) *Given two words $u = u_1\,u_2\,\cdots u_{|u|}$ and $v = v_1\,v_2\,\cdots v_{|v|}$ on an alphabet Σ, the* concatenation of u and v *is the word w on Σ defined as $w = u_1\,u_2\,\cdots u_{|u|}\,v_1\,v_2\,\cdots v_{|v|}$. Its length is thus $|w| = |u| + |v|$.*

This definition is naturally extended to any finite number of words, and to an infinite number of words provided that only a finite number of them are non-empty.

Remark 1.18 (Concatenation and the empty word) *The empty word, ϵ, is the neutral element of the concatenation operation. In other words,*

1. *the concatenation of any number of words is the same as the concatenation of only those words that are non-empty;*
2. *the concatenation of a single word is conventionally defined to be this word;*
3. *the concatenation of zero word is conventionally defined to be the empty word.*

Example(s) 1.36

1. *The concatenation of $a\,a$ and ϵ is $a\,a$.*

2. *The concatenation of $a\,a$, ϵ, ϵ, b, a and ϵ is $a\,a\,b\,a$.*

Definition 1.16 (Power of a word) *Given a word w, the* powers *of w are defined inductively by:*

- *$w^0 = \epsilon$, is the empty word;*
- *for $n \in \mathbb{N}$, $w^{n+1} = w^n\,w$, is the concatenation of w^n and w.*

Example(s) 1.37 *Given two letters a and b,*

1. *$(a\,a\,b)^4 = a\,a\,b\,a\,a\,b\,a\,a\,b\,a\,a\,b$,*
2. *$a^2\,b^3\,a^0 = a\,a\,b\,b\,b$.*

Definition 1.17 (Subword, prefix and suffix) *Given a word u on an alphabet Σ, a* subword[3] *of u is any word v on Σ such that there are two words u_l and u_r on Σ such that $u = u_l\,v\,u_r$. In case $u_l = \epsilon$, v is a* prefix *of u. In case $u_r = \epsilon$, v is a* suffix *of u.*

Example(s) 1.38

1. *The word $a\,b\,a$ has 6 subwords ($\{\epsilon, a, b, a\,b, b\,a, a\,b\,a\}$), of which 4 are prefixes ($\{\epsilon, a, a\,b, a\,b\,a\}$) and 4 are suffixes ($\{\epsilon, a, b\,a, a\,b\,a\}$).*
2. *The word unreliably on the Latin alphabet has 55 subwords, of which 11 are prefixes and 11 are suffixes.*

1.4 Languages

Definition 1.18 (Formal language) *A* formal language *on an alphabet Σ is any set of words on Σ, i.e. any subset of $\Sigma^\star$.*

Remark 1.19 *Unless otherwise stated (e.g. in 'natural language'), in this text, 'language' should be understood as 'formal language'.*

Remark 1.20 *Languages can be finite (i.e. contain a finite number of words) or infinite. While some examples of finite languages are given, almost all languages mentioned in this text are infinite.*

Example(s) 1.39

1. *$\{a^n\,b^n \mid n \in \mathbb{N}\}$, i.e. the set of all words composed of any number of occurrences of the letter a followed by the same number of occurrences of the letter b, is a language on $\{a, b\}$.*
2. *The set of the first names of all students of a class is a language on the alphabet defined as the set of all symbols used in any of these names.*
3. *Given any alphabet Σ, $\emptyset$ and $\Sigma^\star$ are two languages on Σ.*
4. *The sets of all (SMS) text messages sent by a given individual during their entire lifetime is a language on the alphabet defined as the set of all symbols used in any of these messages.*

[3]The English 'subword' (given the present definition) and the French 'sous-mot' are two faux-amis: the latter can refer to any subsequence of the word under consideration. Any subword of a word is also one of its *sous-mots*, but the converse is not true. For example, $a\,c$ is a *sous-mot*, but not a subword, of $a\,b\,c\,d$.

Advanced remark 1.2 *Let Σ be some alphabet. A consequence of the fact that an alphabet is finite (by definition) is that $|\Sigma^\star| = \aleph_0$. A further consequence of this is that any infinite language on Σ is of cardinality $\aleph_0$.*

Remark 1.21 (Usual set operations applied to languages) *Because languages are sets, set union, set intersection and set complementation can be used to produce languages from other languages. Given two languages L_1 and L_2 on Σ:*

- $L_1 \cup L_2 = \{w \in \Sigma^\star \mid w \in L_1 \text{ or } w \in L_2\}$;
- $L_1 \cap L_2 = \{w \in \Sigma^\star \mid w \in L_1 \text{ and } w \in L_2\}$;
- $\overline{L_1} = \Sigma^\star \setminus L_1 = \{w \in \Sigma^\star \mid w \notin L_1\}$.

Definition 1.19 (Concatenation of languages) *Given two languages L_1 and L_2 on an alphabet Σ, the* concatenation of L_1 and L_2 *is the language on Σ defined as $L_1L_2 = \{w \in \Sigma^\star \mid \exists u_1 \in L_2,\ \exists u_2 \in L_2,\ w = u_1\, u_2\}$.*

This definition is naturally extended to any finite number of languages.

Example(s) 1.40 *Consider the two languages on $\Sigma = \{a, b\}$, $L_1 = \{a^n \mid n \in \mathbb{N}\}$ and $L_2 = \{b^n \mid n \in \mathbb{N}\}$, then $L_1L_2 = \{a^n\, b^m \mid n, m \in \mathbb{N}\}$.*

Remark 1.22 *Given two languages L_1 and L_2 on an alphabet Σ,*

- *if $\epsilon \in L_1$, then $L_2 \subseteq L_1L_2$;*
- *if $\epsilon \in L_2$, then $L_1 \subseteq L_1L_2$.*

Definition 1.20 (Power of a language) *Given a language L, the* powers *of L are defined inductively by:*

- *$L^0 = \{\epsilon\}$, is the language that contains only the empty word;*
- *for $n \in \mathbb{N}$, $L^{n+1} = L^n\, L$, is the concatenation of L^n and L.*

Remark 1.23 (L^1) *It follows from Definition 1.20 that for any language L, $L^1 = L$.*

Definition 1.21 (Kleene closure) Kleene closure *is the operation that, when applied to a language L on alphabet Σ, yields the language on Σ defined as $L^\star = \bigcup_{n \in \mathbb{N}} L^n$.*

Remark 1.24 (Alternative formulation) *Given L a language on alphabet Σ, $L^\star = \{w \in \Sigma^\star \mid \exists n \in \mathbb{N},\ \exists u_1, u_2, \cdots, u_n \in L,\ w = u_1\, u_2\, \cdots\, u_n\}$.*

Remark 1.25 *Given L a language on alphabet Σ, $\epsilon \in L^\star$ and $L \subseteq L^\star$.*

Remark 1.26 *Notation 1.8 above introduces the set of all words on Σ as '$\Sigma^\star$'. This notation is consistent with the notation of Kleene closure: if Σ is seen as a finite language of one letter words, then its Kleene closure is indeed the set of all words on Σ.*

Vocabulary

- a set: *un ensemble*
- a tuple: *un n-uplet*
- a word: *un mot*
- a subword: *un facteur*
- Kleene closure: *la fermeture de Kleene*

Chapter 2

Regular languages

Definition 2.1 (Regular language) *Given some alphabet Σ, the* regular languages *on Σ are defined inductively by:*

- *$\emptyset$ and $\{\epsilon\}$ are regular languages on Σ;*
- *for any $a \in \Sigma$, $\{a\}$ is a regular language on Σ;*
- *for any regular language L, $L^\star$ is a regular language on Σ;*
- *for any two regular languages L_1 and L_2, $L_1 \cup L_2$ and $L_1\, L_2$ are regular languages on Σ.*

In other words, a regular language on Σ is either $\emptyset$, $\{\epsilon\}$, $\{a\}$ for some $a \in \Sigma$, or is obtained from them by a finite number of applications of the Kleene closure, union and concatenation operations.

Remark 2.1 *It follows from the previous definition that the union of any finite number of regular languages on Σ and the concatenation of any finite number of regular languages on Σ are regular languages on Σ.*

Example(s) 2.1

1. *$\{a\,b\}$ is a regular language on $\{a, b\}$; it is obtained by concatenation of two regular languages ($\{a\,b\} = \{a\}\,\{b\}$).*
2. *$\{a, b, a\,b\}$ is a regular language on $\{a, b\}$; it is obtained by union of three regular languages ($\{a, b, a\,b\} = \{a\} \cup \{b\} \cup \{a\,b\}$).*
3. *$\{a^n \mid n \in \mathbb{N}\} \cup \{b^m \mid 0 \leq m \leq 5\}$ is a regular language on $\{a, b\}$.*
4. *For any alphabet Σ, Σ seen as a finite language of one letter words is a regular language.*

Theorem 2.1 (Regularity of finite languages) *Given some alphabet Σ, any finite language on Σ is regular.*

Proof 2.1 (Regularity of finite languages) *Let Σ be an alphabet and L a finite language on Σ.*
Let us prove that L is regular.

By definition, there exist $|L|$ words on Σ w_1, w_2, $\cdots$, $w_{|L|}$, such that $L = \{w_1, w_2, \cdots, w_{|L|}\}$.
$L = \{w_1\} \cup \{w_2\} \cup \cdots \cup \{w_{|L|}\}$.

Let us prove that for any $i \in [\![1, |L|]\!]$, $\{w_i\}$ is regular.

Let us consider $i \in [\![1, |L|]\!]$ and define $u = w_i$.
u is a word on Σ: $u = u_1 u_2 \cdots u_{|u|}$.
For any $j \in [\![1, |u|]\!]$, $\{u_j\}$ is a regular language.
Furthermore, $\{u_1 u_2 \cdots u_{|u|}\} = \{u_1\}\{u_2\} \cdots \{u_{|u|}\}$.
So, $\{u_1 u_2 \cdots u_{|u|}\}$ can be obtained by concatenation of a finite number of regular languages.
This proves that $\{w_i\}$ is regular.

So, L can be obtained by union of a finite number of regular languages.
This proves that L is regular.

2.1 Regular expressions

Definition 2.2 (Regular expression) *Given some alphabet Σ, the* regular expressions *on Σ are defined inductively by:*

- *$\varnothing$ and ε are regular expressions on Σ;*
- *for any $a \in \Sigma$, a is a regular expression on Σ;*
- *for any regular expression ϕ, $(\phi)^\star$ is a regular expression on Σ;*
- *for any two regular expressions ϕ_1 and ϕ_2, $(\phi_1 \,|\, \phi_2)$ and $(\phi_1\, \phi_2)$ are regular expressions on Σ.*

Example(s) 2.2 *$(a \,|\, \varepsilon)$, $((b\,b) \,|\, (a\,(a\,a)))^\star$ and $(((a\,a)\,b)^\star\, a)$ are three regular expressions on $\{a, b\}$.*

Remark 2.2

1. *In some texts, regular expressions of the form $(\phi_1 \,|\, \phi_2)$ are written as '$(\phi_1 + \phi_2)$' instead.*
2. *In some texts, regular expressions of the form $(\phi_1\, \phi_2)$ are written as '$(\phi_1 \cdot \phi_2)$' instead.*

Notation 2.1 *The notation of regular expressions are sometimes simplified according to the following conventions.*

- *(Left-associativity) Given three regular expressions ϕ_1, ϕ_2, ϕ_3,*
  - *'$\phi_1 \,|\, \phi_2 \,|\, \phi_3$' stands for '$(((\phi_1 \,|\, \phi_2) \,|\, \phi_3)$',*
  - *'$\phi_1\, \phi_2\, \phi_3$' stands for '$(((\phi_1\, \phi_2)\, \phi_3)$'.*
- *(Priorities) Given three regular expressions ϕ_1, ϕ_2, ϕ_3,*
  - *'$\phi_1 \,|\, \phi_2{}^\star$' and '$\phi_1\, \phi_2{}^\star$' stands for '$(\phi_1 \,|\, (\phi_2)^\star)$' and '$(\phi_1\, (\phi_2)^\star)$' respectively,*
  - *'$\phi_1\, \phi_2 \,|\, \phi_3$' and '$\phi_1 \,|\, \phi_2\, \phi_3$' stands for '$((\phi_1\, \phi_2) \,|\, \phi_3))$' and '$(\phi_1 \,|\, (\phi_2\, \phi_3))$' respectively.*

Example(s) 2.3

1. *'$b\,b \,|\, a\,(a\,a)$' stands for '$((b\,b) \,|\, (a\,(a\,a)))$'.*
2. *'$a\,a\,b\,a$' stands for '$(((a\,a)\,b)\,a)$'.*

Definition 2.3 (Denotation of regular expressions) *Given some alphabet Σ, the* denotation *of the regular expressions on Σ — written '$\mathcal{L}(\phi)$' for the regular expression ϕ — is defined inductively by:*

- *the denotation of $\varnothing$ and ε are $\emptyset$ and $\{\epsilon\}$ respectively (i.e. $\mathcal{L}(\varnothing) = \emptyset$ and $\mathcal{L}(\varepsilon) = \{\epsilon\}$);*
- *for any $a \in \Sigma$, the denotation of a is $\{a\}$ (i.e. $\mathcal{L}(a) = \{a\}$);*
- *for any regular expression ϕ of denotation L, the denotation of $\phi^\star$ is $L^\star$ (i.e. $\mathcal{L}(\phi^\star) = \mathcal{L}(\phi)^\star$);*
- *for any two regular expressions ϕ_1 and ϕ_2 of denotation L_1 and L_2 respectively, the denotation of $(\phi_1 \,|\, \phi_2)$ is $L_1 \cup L_2$ and the denotation of $(\phi_1\, \phi_2)$ is $L_1\, L_2$ (i.e. $\mathcal{L}(\phi_1 \,|\, \phi_2) = \mathcal{L}(\phi_1) \cup \mathcal{L}(\phi_2)$ and $\mathcal{L}(\phi_1\, \phi_2) = \mathcal{L}(\phi_1)\, \mathcal{L}(\phi_2)$).*

Example(s) 2.4

1. *$a^\star$ denotes $\{w \mid \forall i \in [\![1, |w|]\!],\ w_i = a\}$, also written '$\{a^n \mid n \in \mathbb{N}\}$'.*
2. *$a\, b$ denotes $\{a\, b\}$.*
3. *$a\, b\, c$ (i.e. $((a\, b)\, c)$, according to our conventions) denotes $\{a\, b\, c\}$.*
4. *$(a \,|\, b)^\star$ denotes $\{w \mid \forall i \in [\![1, |w|]\!],\ w_i = a$ or $w_i = b\}$.*

Theorem 2.2 (Regularity of the denotation of a regular expression) *The denotation of a regular expression on an alphabet Σ is a regular language on Σ.*

Proof 2.2 (Regularity of the denotation of a regular expression) *This proof consists of a very simple induction on the structure of regular expressions ϕ on alphabet Σ:*

- *(init-$\varnothing$) if $\phi = \varnothing$, then its denotation is $\emptyset$, which is a regular language on Σ;*
- *(init-ε) if $\phi = \varepsilon$, then its denotation is $\{\epsilon\}$, which is a regular language on Σ;*
- *(init-a) if $\phi = a$ for some $a \in \Sigma$, then its denotation is $\{a\}$, which is a regular language on Σ;*
- *(rec-star) if $\phi = \psi^\star$ for some regular expression ψ of denotation L, then its denotation is $L^\star$; assuming that L is a regular language on Σ, so is $L^\star$;*
- *(rec-union) if $\phi = (\phi_1 \,|\, \phi_2)$ for two regular expressions ϕ_1 and ϕ_2 of respective denotation L_1 and L_2, then its denotation is $L_1 \cup L_2$; assuming that L_1 and L_2 are regular languages on Σ, so is $L_1 \cup L_2$;*
- *(rec-concat) if $\phi = (\phi_1\, \phi_2)$ for two regular expressions ϕ_1 and ϕ_2 of respective denotation L_1 and L_2, then its denotation is $L_1\, L_2$; assuming that L_1 and L_2 are regular languages on Σ, so is $L_1\, L_2$;*

By the principle of induction, the denotation of any regular expression on Σ is a regular language on Σ.

Theorem 2.3 (Exhaustivity of regular expressions) *Any regular language on alphabet Σ is denoted by (at least) one regular expression on Σ.*

Proof 2.3 (Exhaustivity of regular expressions) *This proof consists of a very simple induction on the operations used to build the regular languages L on alphabet Σ:*

- *(init-$\emptyset$) if $L = \emptyset$, then it is denoted by the regular expression $\varnothing$;*
- *(init-$\{\epsilon\}$) if $l = \{\epsilon\}$, then it is denoted by the regular expression ε;*
- *(init-$\{a\}$) if $L = \{a\}$ for some $a \in \Sigma$, then it is denoted by the regular expression a;*
- *(rec-Kleene) if $L = L'^{\star}$ for some regular language L', then, assuming that L' is denoted by a regular expression ϕ', L is denoted by the regular expression $\phi'^{\star}$;*
- *(rec-union) if $L = L_1 \cup L_2$ for two regular languages L_1 and L_2, then, assuming that L_1 and L_2 are denoted by two regular expressions ϕ_1 and ϕ_2 respectively, L is denoted by the regular expression $(\phi_1 \,|\, \phi_2)$;*
- *(rec-concat) if $L = L_1\, L_2$ for two regular languages L_1 and L_2, then, assuming that L_1 and L_2 are denoted by two regular expressions ϕ_1 and ϕ_2 respectively, L is denoted by the regular expression $(\phi_1\, \phi_2)$;*

By the principle of induction, any regular language on Σ is denoted by a regular expression on Σ.

Theorem 2.4 (Equivalence between regular languages and regular expressions) *The denotations of regular expressions on alphabet Σ are exactly the regular languages on Σ.*

Proof 2.4 (Equivalence between regular languages and regular expressions) *This result directly follows from Theorem 2.2 and Theorem 2.3.*

Remark 2.3 *While a given regular expression on Σ denotes a single language on Σ, the same language might be denoted by two (or more) different regular expressions.*

Example(s) 2.5

1. *$((a\,b)\,c)$ and $(a\,(b\,c))$ both denote the same language (namely, $\{a\,b\,c\}$).*
2. *ε and $\varepsilon^{\star}$ both denote the same language (namely, $\{\epsilon\}$).*
3. *For any regular expression ϕ, $\varnothing$, $(\varnothing\,\phi)$ and $(\phi\,\varnothing)$ all denote the same language (namely, $\emptyset$).*
4. *For any regular expression ϕ, ϕ, $(\varepsilon\,\phi)$ and $(\phi\,\varepsilon)$ all denote the same language.*
5. *$(a\,|\,b)^{\star}$, $(b\,|\,a)^{\star}$ and $(a^{\star}\,b^{\star})^{\star}$ all denote the same language (namely, $\{a, b\}^{\star}$).*

Remark 2.4 (Common use of regular expressions) *Regular expressions (also called 'regex') are used as descriptions of search patterns in various software applications. An* occurrence *of such a pattern in a text is defined as any subword of the text that is also a member of the language denoted by the corresponding regular expression. Accordingly, one says that such a pattern* occurs *in a text iff it has at least one occurrence in the text.*

Programming languages usually provide functions to write search patterns with regular expressions and find their occurrences in texts. They also often define many notation conventions in order to simplify the writting of such patterns. Here are conventions that are similar to popular ones:

- *(zero or one) '$\phi?$' stands for '$(\phi\,|\,\epsilon)$';*
- *(one or more) '$\phi+$' stands for '$(\phi\,\phi^{\star})$';*

- *(set)* '$[a_1\,a_2\,\cdots\,a_n]$' *stands for* '$(a_1\,|\,a_2\,|\,\cdots\,|\,a_n)$'*;*
- *(excluded set)* '$[\wedge a_1\,a_2\,\cdots\,a_n]$' *stands for* '$(b_1\,|\,b_2\,|\,\cdots\,|\,b_m)$' *where the b_j·s are all the letters of the alphabet not among the a_i·s.*
- *(interval) for $i, j \in [\![0, 9]\!]$,* '$[i - j]$' *stands for* '$(i\,|\,i+1\,|\,\cdots\,|\,j)$'.

(These conventions will not be used in this text, except in the following examples.)

Example(s) 2.6 *With the previous conventions,*

1. $[0-2]\,[0-9]\ :\ [0-6]\,[0-9]$ *can be used to detect times in the* `hh:mm` *format, with the caveat that it also incorrectly matches some sequences of characters that do not correspond to any valid time, such as '11:64' or '26:12'. It is possible to define a 'perfect' regular expression for times in this format, but it would be quite long with standard notation conventions.*
2. $[0-9]\,[0-9]\,[0-9]\,[0-9]/[0\,1]\,[0-9]/[0-3]\,[0-9]$ *can be used to detect dates in the* `yyyy/mm/dd` *format, with the caveat that it also incorrectly matches some sequences of characters that do not correspond to any valid date, such as '1964/02/31' or '2055/19/00'. It is possible to define a 'perfect' regular expression for dates in this format, but it would be quite long with standard notation conventions.*

Exercise 2.1 *Write a Python script that expects one string as argument (given via the command line) and lists all dates in the* `yyyy/mm/dd` *format that occur in this string. (You might need to look at the documentation of Python's* `re` *library to do this exercise.)*

2.2 Finite-state automata

Definition 2.4 (Deterministic finite-state automaton; DFA) *A* deterministic finite-state automaton *(DFA) A is a quintuple $(\Sigma, Q, q_0, F, \delta)$ where:*

- *Σ is a finite set the elements of which are called 'symbols';*
- *Q is a finite set the elements of which are called 'states';*
- *q_0 is an element of Q called 'the initial state' (or 'start state');*
- *F is a subset of Q the elements of which are called 'final states' (or 'accepting states');*
- *δ is a function from $Q \times \Sigma$ to Q called 'the transition function'.*

Example(s) 2.7 *$A = (\Sigma, Q, q_0, F, \delta)$ with the following definitions is a DFA with three states among which a single final state (which happens to also be the initial state):*

- $\Sigma = \{a, b\}$,
- $Q = \{q_0, q_1, q_2\}$,
- $F = \{q_0\}$,
- $\delta = (q, x) \mapsto \begin{cases} q \text{ if } x = a \\ q_1 \text{ if } q = q_0 \text{ and } x = b \\ q_2 \text{ if } q = q_1 \text{ and } x = b \\ q_0 \text{ if } q = q_2 \text{ and } x = b \end{cases}$.

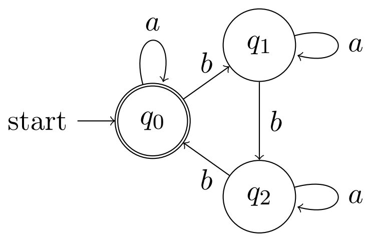


Figure 2.1: Graphical representation of the DFA from example 2.7.

Property 2.1 (DFAs as graphs) *A DFA $A = (\Sigma, Q, q_0, F, \delta)$ can been seen as a directed edge-labelled graph $G = (V, E, f)$ where:*

- *$V = Q$ (the nodes),*
- *$E = \{(q, q') \in Q \times Q \mid \exists a \in \Sigma,\ \delta((q, a)) = q'\}$ (the edges),*
- *$f = (q, q') \mapsto \{a \in \Sigma \mid \delta((q, a)) = q'\}$ (the edge-labelling; labels are non-empty sets of symbols),*

with, in addition, one node (q_0) designated as 'the initial state' and an arbitrary number of nodes (F) designated as 'final states'.

Remark 2.5 (Graphical representation of DFAs) *A DFA can be graphically represented as their directed edge-labelled graphs (see Property 2.1) with, in addition, an indication of the initial state and an indication of the final states. In this course, the inital state of a DFA is indicated with an incoming arrow labelled 'start', and the final states are indicated with a double circle (while non-final states are indicated with a single circle).*

Example(s) 2.8 *The DFA from example 2.7 is graphically represented in figure 2.1.*

Definition 2.5 (Transition matrix) *The* transition matrix *of a DFA $(\Sigma, Q, q_0, F, \delta)$ is the matrix T with rows labelled by Q and columns labelled by Σ, and such that, for $q \in Q$ and $a \in \Sigma$, the coefficient (q, a) of T is $\delta((q, a))$.*

Example(s) 2.9 *The transition matrix of the DFA from example 2.7 is the following:*

	a	b
q_0	q_0	q_1
q_1	q_1	q_2
q_2	q_2	q_0

Remark 2.6 (Matrices as transition functions) *Consider a matrix T with rows labelled by a set Q and columns labelled by a set Σ, and such that, for $q \in Q$ and $a \in \Sigma$, the coefficient (q, a) is an element of Q. Then T is the transition matrix of any DFA $(\Sigma, Q, q_0, F, \delta)$ where $q_0 \in Q$, $F \subseteq Q$, and, for $q \in Q$ and $a \in \Sigma$, $\delta((q, a))$ is the coefficient (q, a) of T.*

Definition 2.6 (Transition) *Given a DFA $A = (\Sigma, Q, q_0, F, \delta)$, a* transition *in A is a triple $t = (q, a, q') \in Q \times \Sigma \times Q$ such that $q' = \delta((q, a))$. q, a and q' are respectively the* source state, *the* label, *and the* target state, *of the transition.*

Definition 2.7 (Computation) *Given a DFA $A = (\Sigma, Q, q_0, F, \delta)$, a* computation *in A is a finite sequence of transitions in A such that for any two subsequent transitions, the target state of the first is the source state of the second.*

Example(s) 2.10 *With A the DFA from example 2.7:*

- $((q_1, b, q_2), (q_2, a, q_2), (q_2, a, q_2))$ *is a computation in A;*
- $((q_0, b, q_1), (q_1, b, q_2), (q_2, b, q_0), (q_0, b, q_1))$ *is a computation in A;*
- $()$, *the empty sequence of transition, is a computation in A.*

Definition 2.8 (Source and target of a non-empty computation) *Given a non-empty computation $(t_i)_{i \in [\![1,n]\!]}$, the* source *of $(t_i)_{i \in [\![1,n]\!]}$ is the source state of t_1, and its* target *is the end state of t_n.*

Remark 2.7 *A non-empty computation in a DFA is equivalent to a (finite) path of states starting from the source of the computation and, at each step, following one of the outward transitions from this state, ending at the target of the computation.*

Definition 2.9 (Input of a computation) *Given a DFA $A = (\Sigma, Q, q_0, F, \delta)$ and a computation $(t_i)_{i \in [\![1,n]\!]}$ in A, the* input *of this computation is the concatenation of the labels of the transitions of the computation (in the same order).*

Example(s) 2.11 *With A the DFA from example 2.7:*

- $b\,a\,a$ *is the input of* $((q_1, b, q_2), (q_2, a, q_2), (q_2, a, q_2))$;
- b^4 *(i.e. $b\,b\,b\,b$) is the input of* $((q_0, b, q_1), (q_1, b, q_2), (q_2, b, q_0), (q_0, b, q_1))$;
- ϵ *is the input of* $()$.

Notation 2.2 *Given a DFA $A = (\Sigma, Q, q_0, F, \delta)$, $q, q' \in Q$ and $w \in \Sigma^\star$, the notation '$q \xrightarrow{w}_A q'$' indicates that:*

- *either $q = q'$ and $w = \epsilon$,*
- *or there exists in A a non-empty computation of source q, target q' and input w.*

Example(s) 2.12 *With A the DFA from example 2.7:*

- $q_1 \xrightarrow{b\,a\,a}_A q_2$;
- $q_0 \xrightarrow{b^4}_A q_1$;
- $q_2 \xrightarrow{\epsilon}_A q_2$.

Definition 2.10 (Successful computation) *Given a DFA $A = (\Sigma, Q, q_0, F, \delta)$,*

- *if $q_0 \in F$ (i.e. the initial state is also a final state), then there is a single* trivial successful computation *in A, namely the empty computation $()$, otherwise, there is no trivial successful computation in A;*
- *a* non-trivial successful computation *in A is a non-empty computation in A such that:*
  - *the source state of the first transition is q_0, the initial state of A,*
  - *and the target state of the last transition is a final state of A;*
- *a* successful computation *in A is either a trivial or a non-trivial successful computation in A.*

Remark 2.8 *Successful computations are also referred to as 'accepting computations'.*

Example(s) 2.13 *With A the DFA from example 2.7:*

- $((q_0, b, q_1), (q_1, b, q_2), (q_2, b, q_0))$ *is a (non-trivial) successful computation in A;*
- $((q_0, b, q_1), (q_1, a, q_1), (q_1, b, q_2), (q_2, b, q_0))$ *is a (non-trivial) successful computation in A;*
- $()$*, the empty sequence of transition, is a (trivial) successful computation in A.*

Definition 2.11 (Acceptance and rejection by a DFA) *Given a DFA $A = (\Sigma, Q, q_0, F, \delta)$, a word $w \in \Sigma^\star$ is* accepted *by A if it is the input of a successful computation in A, otherwise, w is* rejected *by A.*

Example(s) 2.14 *$b\,b\,b$, $b\,a\,b\,b$, $a\,a$, b^6 and ϵ are five words accepted by A, the DFA from example 2.7. b and b^4 are two words rejected by this DFA.*

Remark 2.9 *Given a DFA $A = (\Sigma, Q, q_0, F, \delta)$, a word $w \in \Sigma^\star$ is accepted by A iff there is a $q_f \in F$ such that $q_0 \xrightarrow{w}_A q_f$.*

Definition 2.12 (Language recognised by a DFA) *Given a DFA $A = (\Sigma, Q, q_0, F, \delta)$, the* language recognised by *A, written '$\mathcal{L}(A)$', is the set of all words accepted by A: $\mathcal{L}(A) = \{w \in \Sigma^\star \mid w \text{ is accepted by } A\}$.*

Example(s) 2.15 *The language recognised by the DFA from example 2.7 is $\{w \in \{a, b\}^\star \mid \exists k \in \mathbb{N},\ |w|_b = 3k\}$, the set of words on $\{a, b\}$ that contain a multiple of 3 of occurrences of b (and any arbitrary number of occurrences of a).*

This (fairly intuitive) results can be proven by proving, by induction on $n \in \mathbb{N}$, that for any word $w \in \{a, b\}^n$, the computation in A of input w starting at q_0 ends in q_i where $i \in [\![0, 2]\!]$ is the remainder of the Euclidean division of $|w|_b$ by 3 (in other words, i is the (unique) $r \in [\![0, 2]\!]$ such that $\exists k \in \mathbb{N}$, $|w|_b = 3k + r$).

Remark 2.10 (The machine metaphor) *A DFA $A = (\Sigma, Q, q_0, F, \delta)$ can be thought of as a machine the purpose of which is to analyse words. In order to analyse a word w on Σ, the machine is presented a linear* tape *consisting of $|w| + 1$ cells, such that, for each $i \in [\![1, |w|]\!]$, the i^{th} cell contains the symbol w_i, and such that the last cell contains a special symbol $\$ \notin \Sigma$. The machine has as a part a* tape head*, which always points to some cell of the tape.*

- *The machine starts the analysis in its initial state q_0 and with its tape head pointing at the first cell of the tape.*
- *At each step of the analysis,*
  - *if the head points at the last cell of the tape (the one containing the special symbol $\$$), then the analysis ends with* `accept` *or* `reject` *as output depending on whether the current state is a final state (*`accept`*) or not (*`reject`*),*
  - *otherwise, the state of the machine is updated according to the transition function δ of the DFA and based on both its current state and the symbol currently under the tape head, and the tape head is then moved to the next following cell.*

Example(s) 2.16 *Figure 2.2 illustrates some of the steps of the execution of the machine running the DFA from example 2.7 on the word $b\,a\,b\,b$.*

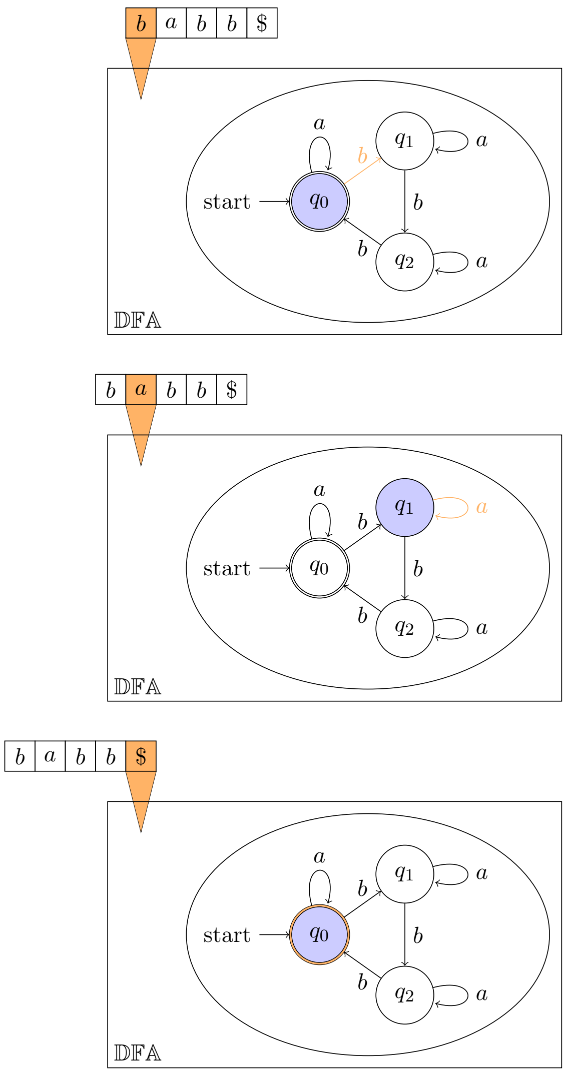


Figure 2.2: Illustration of the first, second and last (fifth) steps of the execution of the machine running the DFA from example 2.7 on the word $b\,a\,b\,b$ (from top to bottom). The input word, together with the special symbol $, is written on a tape. The tape head is represented as an orange triangle. The machine keeps track of the current state of the DFA (in blue).

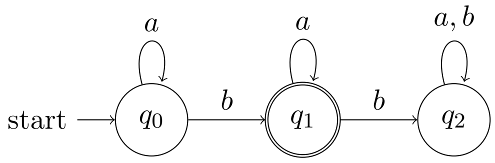


Figure 2.3: Graphical representation of the DFA from example 2.17, which contains a dead state (q_2).

Remark 2.11 *The tape on which is written the input of a DFA is more precisely a* buffer. *A buffer is a sequence of memory cells that can only be read in order.*

Definition 2.13 (Dead state) *Let $A = (\Sigma, Q, q_0, F, \delta)$ be a DFA. A* dead state *of A is a state $q \in Q$ such that $q \notin F$ and that for all $a \in \Sigma$, $\delta((q, a)) = q$.*

Example(s) 2.17 *Let $A = (\{a, b\}, \{q_0, q_1, q_2\}, q_0, \{q_1\}, \delta)$ where δ is the transition function represented by the following matrix:*

	a	b
q_0	q_0	q_1
q_1	q_1	q_2
q_2	q_2	q_2

A is a DFA, represented graphically in figure 2.3, and q_2 is a dead state of A.

Theorem 2.5 *Let $A = (\Sigma, Q, q_0, F, \delta)$ be a DFA and q be a dead state of A. No successful computation in A goes through q, meaning that no successful computation in A contains a transition the target state of which is q.*

Proof 2.5 *Let $A = (\Sigma, Q, q_0, F, \delta)$ be a DFA and q be a dead state of A.*
Let $(t_i)_{i \in [\![1,n]\!]}$ be a computation in A that goes through q.
Let us prove that $(t_i)_{i \in [\![1,n]\!]}$ is not successful.

Let us show that this computation ends in q, i.e. that the target state of t_n is q.

By assumption, there is an $i \in [\![1, n]\!]$ such that the target state of t_i is q.
If $i = n$, then the target state of t_n is q.
Otherwise, because for all $j \in [\![i, n-1]\!]$ the target state of t_j is the source state of t_{j+1} and because for all $a \in \Sigma$, $\delta((q, a)) = q$, a simple induction shows that for all $j \in [\![i, n]\!]$, the target state of t_j is q.
This proves that in any case, the target state of t_n is q.

By assumption, $q \notin F$, so the target state of the last transition of $(t_i)_{i \in [\![1,n]\!]}$ is not a final state.
This proves that $(t_i)_{i \in [\![1,n]\!]}$ is not successful.

Definition 2.14 (Product automaton) *Consider two DFAs $A_1 = (\Sigma, Q_1, q_{i,1}, F_1, \delta_1)$ and $A_2 = (\Sigma, Q_2, q_{i,2}, F_2, \delta_2)$. Without loss of generality, we can assume that $Q_1 \cap Q_2 = \emptyset$. The* product automaton *of A_1 and A_2 is the DFA $(\Sigma, Q, q_i, F, \delta)$ where*

- *$Q = Q_1 \times Q_2$,*

- $q_i = (q_{i,1}, q_{i,2})$,
- $F = F_1 \times F_2$,
- $\delta = ((q_1, q_2), x) \mapsto (\delta_1(q_1), \delta_2(q_2))$.

.

Example(s) 2.18 *TODO*

Theorem 2.6 *Let A_1 and A_2 be two DFAs. The product automaton of A_1 and A_2 recognises $\mathcal{L}(A_1) \times \mathcal{L}(A_2)$.*

Proof 2.6 *TODO*

Definition 2.15 (Complement automaton) *Consider a DFA $A = (\Sigma, Q, q_i, F, \delta)$. The* complement automaton *of A is the DFA $(\Sigma, Q, q_i, \overline{F}, \delta)$ where $\overline{F} = Q \setminus F$.*

Example(s) 2.19 *TODO*

Theorem 2.7 *Let A be a DFA. The complement automaton of A recognises $\overline{\mathcal{L}(A)}$ (i.e. $\Sigma^\star \setminus \mathcal{L}(A)$).*

Proof 2.7 *TODO*

Intuition 2.1 (Incomplete DFA) *The transition function of a DFA is complete in the sense that it defines a transition for all pairs of a state and a letter.* Incomplete DFAs *differ from DFAs in that their transition functions do not necessarily define a transition in all cases.*

Definition 2.16 (Incomplete DFA) *An* incomplete DFA *A is a quintuple $(\Sigma, Q, q_0, F, \delta)$ where:*

- *Σ is a finite set the elements of which are called 'symbols';*
- *Q is a finite set the elements of which are called 'states';*
- *q_0 is an element of Q called 'the initial state' (or 'start state');*
- *F is a subset of Q the elements of which are called 'final states' (or 'accepting states');*
- *δ is a partial function from $Q \times \Sigma$ to Q called 'the transition function'.*

Example(s) 2.20 *$A = (\Sigma, Q, q_0, F, \delta)$ with the following definitions is an incomplete DFA:*

- $\Sigma = \{a, b\}$,
- $Q = \{q_0, q_1\}$,
- $F = \{q_1\}$,
- $\delta = (q, x) \mapsto \begin{cases} q_1 \text{ if } q = q_0 \text{ and } x = b \\ q_0 \text{ if } q = q_1 \text{ and } x = a \end{cases}$.

This incomplete DFA is graphically represented in figure 2.4.

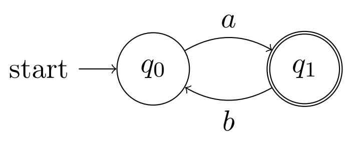


Figure 2.4: Graphical representation of the incomplete DFA from example 2.20.

Remark 2.12 *All definitions given above related to transition, computation, acceptance and recognition, are naturally extended to incomplete DFAs. The main thing to keep in mind is that for an incomplete DFA of transition function δ, there is no transition of source state q and label a such that δ is undefined on the pair (q, a); in terms of machine, the machine stops with a failure when given a in state q in that case.*

Theorem 2.8 (Equivalence of DFAs and incomplete DFAs) *Let L be a language on some alphabet Σ; there is a DFA that recognises L iff there is an incomplete DFA that recognises L.*

Proof 2.8 (Equivalence of DFAs and incomplete DFAs) *Let L be a language on some alphabet Σ.*

- *($\Rightarrow$) Assume that A is a DFA that recognises L.*
  A, being a DFA, is also an incomplete DFA.
  So, there is an incomplete DFA that recognises L.

- *($\Leftarrow$) Assume that $A = (\Sigma, Q, q_0, F, \delta)$ is an incomplete DFA that recognises L.*
  Let $q' \notin Q$, define $Q' = Q \cup \{q'\}$ and then the function δ' from $Q' \times \Sigma$ to Q' with $\delta' = (q, x) \mapsto \begin{cases} \delta((q, x)) \text{ if } \delta \text{ is defined on } (q, x) \\ q' \text{ otherwise} \end{cases}$.
  Consider the DFA $A' = (\Sigma, Q', q_0, F, \delta')$ and note that q' is a dead state of A'.
  Let us prove that $\mathcal{L}(A) = \mathcal{L}(A')$.

  - *($\subseteq$) It is clear that any (successful) computation of A is a (successful) computation of A'.*
    So, $\mathcal{L}(A) \subseteq \mathcal{L}(A')$.
  - *($\supseteq$) Consider any $w \in \mathcal{L}(A')$.*
    By definition, w is the input of a successful computation $(t_i)_{i \in [\![1,|w|]\!]}$ in A'.
    Because q' is a dead state of A', one knows from Theorem 2.5 that $(t_i)_{i \in [\![1,|w|]\!]}$ does not go through q'.
    So, $(t_i)_{i \in [\![1,|w|]\!]}$ is a computation of A, and even more, a successful computation of A.
    So, $w \in \mathcal{L}(A)$.
    So, $\mathcal{L}(A') \subseteq \mathcal{L}(A)$.

  So, there is a DFA that recognises L.

Intuition 2.2 (Non-deterministic finite-state automaton; NFA) *The transition function of a(n incomplete) DFA is deterministic in the sense that the image of a pair of a state and a letter, (if defined) is a state; in other words, there is only one (or zero) transition for a given source state and a given label.* Non-deterministic finite-state automata *differ from (incomplete) DFAs in that their transition functions associate sets of states, rather than states, to pairs of a state and a letter. As a consequence, there might be two or more transitions for a given source state and a given label.*

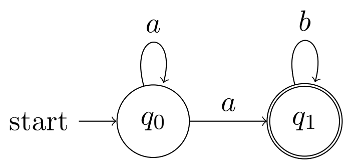


Figure 2.5: Graphical representation of the NFA from example 2.21.

Definition 2.17 (Non-deterministic finite-state automaton; NFA) *A* non-deterministic finite-state automaton *(NFA) A is a quintuple $(\Sigma, Q, q_0, F, \delta)$ where:*

- *Σ is a finite set the elements of which are called 'symbols';*
- *Q is a finite set the elements of which are called 'states';*
- *q_0 is an element of Q called 'the initial state' (or 'start state');*
- *F is a subset of Q the elements of which are called 'final states' (or 'accepting states');*
- *δ is a function from $Q \times \Sigma$ to $\mathcal{P}(Q)$ called 'the transition function'.*

Example(s) 2.21 *$A = (\Sigma, Q, q_0, F, \delta)$ with the following definitions is an NFA:*

- *$\Sigma = \{a, b\}$,*
- *$Q = \{q_0, q_1\}$,*
- *$F = \{q_1\}$,*
- $\delta = (q, x) \mapsto \begin{cases} \{q_0, q_1\} \text{ if } q = q_0 \text{ and } x = a \\ \{q_1\} \text{ if } q = q_1 \text{ and } x = b \\ \emptyset \text{ otherwise} \end{cases}$.

This NFA is graphically represented in figure 2.5. Its transition function δ can be represented by the following matrix:

	a	b
q_0	$\{q_0, q_1\}$	$\emptyset$
q_1	$\emptyset$	$\{q_1\}$

Definition 2.18 (Transition in an NFA) *Given an NFA $A = (\Sigma, Q, q_0, F, \delta)$, a* transition *in A is a triple $t = (q, a, q') \in Q \times \Sigma \times Q$ such that $q' \in \delta((q, a))$. q, a and q' are respectively the* source state*, the* label*, and the* target state*, of the transition.*

Remark 2.13 *All definitions given above related to computation, acceptance and recognition, are naturally extended to NFAs using the notion of transition given by Definition 2.18 rather than given by Definition 2.6. The main things to keep in mind is that for an NFA, there might be zero, one or more computations with the same input, and that a word is accepted if it is the input of at least one successful computation.*

Example(s) 2.22 *Let A be the NFA from example 2.21.*

- *There are exactly two computations of input $a\,a$ in A:*
  - $((q_0, a, q_0), (q_0, a, q_0))$;
  - $((q_0, a, q_0), (q_0, a, q_1))$.

There is exactly one computation of input $a\,a\,b$ in A:

– $((q_0, a, q_0), (q_0, a, q_1), (q_1, b, q_1))$.

- (q_0, a, q_1), *but not* (q_0, a, q_0), *is a successful (non-trivial) computation in* A. $((q_0, a, q_0), (q_0, a, q_1), (q_1, b, q_1))$ *is also a successful (non-trivial) computation in* A.
- a *and* $a\,a\,b$ *are two words accepted by* A.
- $\mathcal{L}(A) = \{a^n\, b^m \mid n, m \in \mathbb{N},\, n \geq 1\}$.

Intuition 2.3 (NFA with ϵ-moves) *In all automata seen so far, all transitions are labelled with a letter. In contrast, in an* NFA with ϵ-moves*, the transitions can also be labelled with* ϵ*. Such transitions, called 'ϵ-moves' or 'ϵ-transitions', can be taken without 'consuming' any letter of the input word.*

Definition 2.19 (NFA with ϵ-moves) *An* NFA with ϵ-moves A *is a quintuple* $(\Sigma, Q, q_0, F, \delta)$ *where:*

- Σ *is a finite set the elements of which are called 'symbols';*
- Q *is a finite set the elements of which are called 'states';*
- q_0 *is an element of* Q *called 'the initial state' (or 'start state');*
- F *is a subset of* Q *the elements of which are called 'final states' (or 'accepting states');*
- δ *is a function from* $Q \times (\Sigma \cup \{\epsilon\})$ *to* $\mathcal{P}(Q)$ *called 'the transition function'.*

Example(s) 2.23 $A = (\Sigma, Q, q_0, F, \delta)$ *with the following definitions is an NFA:*

- $\Sigma = \{a, b, c\}$,
- $Q = \{q_0, q_1, q_2\}$,
- $F = \{q_1, q_2\}$,
- $\delta = (q, x) \mapsto \begin{cases} \{q_0, q_1\} \text{ if } q = q_0 \text{ and } x = a \\ \{q_1\} \text{ if } q = q_1 \text{ and } x = b \\ \{q_2\} \text{ if } q = q_0 \text{ and } x = \epsilon \\ \{q_2\} \text{ if } q = q_2 \text{ and } x = c \\ \emptyset \text{ otherwise} \end{cases}$.

This NFA with ϵ-moves is graphically represented in figure 2.6. Its transition function δ can be represented by the following matrix:

	a	b	c	ϵ
q_0	$\{q_0, q_1\}$	$\emptyset$	$\emptyset$	$\{q_2\}$
q_1	$\emptyset$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_2	$\emptyset$	$\emptyset$	$\{q_2\}$	$\emptyset$

Remark 2.14 *All notions related to transition, computation, acceptance and recognition, for NFAs directly apply to NFAs with ϵ-moves. The only thing to keep in mind is that ϵ is the neutral element of the concatenation operation (see Remark 1.18).*

Example(s) 2.24 *Let A be the NFA with ϵ-moves from example 2.23.*

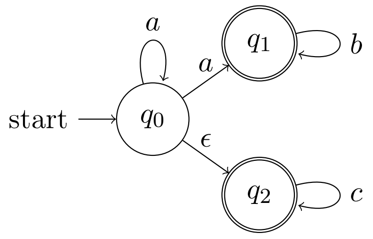


Figure 2.6: Graphical representation of the NFA with ϵ-moves from example 2.23.

- *There are exactly three computations of input $a\,a$ in A:*
  - $((q_0, a, q_0), (q_0, a, q_0))$;
  - $((q_0, a, q_0), (q_0, a, q_1))$;
  - $((q_0, a, q_0), (q_0, a, q_0), (q_0, \epsilon, q_2))$.

  There is exactly one computation of input $c\,c\,c$ in A:
  - $((q_0, \epsilon, q_2), (q_2, c, q_2), (q_2, c, q_2), (q_2, c, q_2))$.
- $\mathcal{L}(A) = \{a^n\, b^m \mid n, m \in \mathbb{N},\, n \geq 1\} \cup \{a^n\, c^m \mid n, m \in \mathbb{N}\}$.

Definition 2.20 (Determinised version of an NFA with ϵ-moves) *Let $A = (\Sigma, Q, q_0, F, \delta)$ be an NFA with ϵ-moves. Consider the DFA $A' = (\Sigma, Q', S_0, F', \delta')$ where*

- $Q' = \mathcal{P}(Q)$ *(the states of the DFA A' are exactly the sets of states of A),*
- $S_0 = \{q' \in Q \mid q_0 \xrightarrow{\epsilon}_A q'\}$ *(the initial state of the DFA A' is the set that contains exactly the initial state q_0 of A and all states of A that are the target of a non-empty computation of source q_0 and input ϵ in A),*
- $F' = \{S \subseteq Q \mid (S \cap F) \neq \emptyset\}$ *(the final states of the DFA A' are exactly the sets of states of A that include one of its final state),*
- δ' *is the function from $Q' \times \Sigma$ to Q' defined as $(S, x) \mapsto \{q' \in Q \mid \exists q \in S,\ q \xrightarrow{x}_A q'\}$.*

Let Q" be the set of states of A' that are accessible from S_0, i.e. S_0 and any state that is the target of some computation of source S_0 in A'. Q" is the smallest subset of Q' that contains S_0 and such that for any $S \in Q$" and any $x \in \Sigma$, $\delta'((S, x)) \in Q$". Let δ" be the restriction of δ' to Q", i.e. the function from Q" $\times\, \Sigma$ to Q" defined as $(S, x) \mapsto \delta'((S, x))$, and F" be the restriction of F' to Q", i.e. $F' \cap Q$". The determinised version *of A is the DFA A" $= (\Sigma, Q$", S_0, F", δ").*

Example(s) 2.25 *The determinised version of the NFA with ϵ-moves from example 2.23 is graphically represented in figure 2.7.*

- Given an NFA with ϵ-moves A, algorithm 1 computes the determinised version of A. This algorithm does not reflect the structure of Definition 2.20 as it directly builds the determinised version of A (A" in Definition 2.20). This operation is done by incrementally building the set of states Q" and the transition function δ" starting from the initial states S_0.

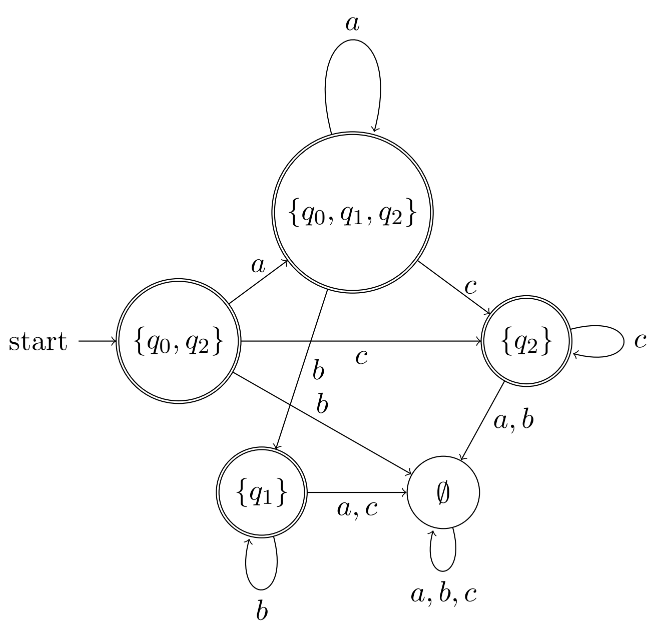


Figure 2.7: Graphical representation of the determinised version of the NFA with ϵ-moves from example 2.23.

```
// Input:  an NFA with ϵ-moves $A = (\Sigma, Q, q_0, F, \delta)$
// Output:  the determinised version of $A$
$S_0 := \{q' \in Q \mid q_0 \xrightarrow{\epsilon}_A q'\}$;
$Q" :=$ ordered_set($\{S_0\}$);
$\delta" :=$ dictionary();
$k := 0$;
while $k < len(Q")$ do
    $S := Q"[k]$;
    for $a \in \Sigma$ do
        $S' := \{q' \in Q \mid \exists q \in S,\ q \xrightarrow{a}_A q'\}$;
        $Q"$.add($S'$);
        $\delta"[(S, a)] = S'$;
    $k$ += 1;
$F" := \{S \in Q" \mid (S \cap F) \neq \emptyset\}$;
return $(\Sigma, Q", S_0, F", \delta")$;
```

Algorithm 1: Determinisation of an NFA with ϵ-moves.

Remark 2.15 *There is a convenient way to execute algorithm 1 with pen and paper by progressively writing down the transition matrix of the DFA under construction.*

For example, applying the algorithm to the NFA with ϵ-moves from example 2.23 can be seen as building the following transition matrix from the first line (corresponding to the initial state $S_0 = \{q' \in Q \mid q_0 \overset{\epsilon}{\to}_A q'\} = \{q_0, q_2\}$) to the last line and, for each line, from the first column to the last column (if Σ is always iterated over as the sequence (a, b, c)).

	a	b	c
$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_1\}$	$\{q_2\}$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{q_2\}$	$\emptyset$	$\emptyset$	$\{q_2\}$
$\{q_1\}$	$\emptyset$	$\{q_1\}$	$\emptyset$

Theorem 2.9 *Given an NFA with ϵ-moves A and its determinised version A', $\mathcal{L}(A) = \mathcal{L}(A')$.*

Proof 2.9 *TODO*
The idea is, given an NFA with ϵ-moves $A = (\Sigma, Q, q_0, F, \delta)$ and its determinised version $A' = (\Sigma, Q', S_0, F', \delta')$, to first prove by induction on $n \in \mathbb{N}$ that for any $w \in \Sigma^n$, $\{q \in Q \mid q_0 \overset{w}{\to}_A q\}$ is exactly the (unique) state $S \in Q'$ such that $S_0 \overset{w}{\to}_{A'} S$.
Then, proceed by double inclusion using the definition of the final states of A'.

Theorem 2.10 (Equivalence of DFAs and NFAs with ϵ-moves) *Let L be a language on some alphabet Σ; there is a DFA that recognises L iff there is an NFA with ϵ-moves that recognises L.*

Proof 2.10 (Equivalence of DFAs and NFAs with ϵ-moves) *Let L be a language on some alphabet Σ.*

- *($\Rightarrow$) Assume that $A = (\Sigma, Q, q_0, F, \delta)$ is a DFA that recognises L.*
  Define the function δ' from $Q \times \Sigma$ to $\mathcal{P}(Q)$ with $\delta' = (q, x) \mapsto \{\delta(q, x)\}$.
  Consider the NFA with ϵ-moves $A' = (\Sigma, Q, q_0, F, \delta')$.
  It is clear that the (successful) transitions of A' are exactly the (successful) transitions of A.
  So, $\mathcal{L}(A') = \mathcal{L}(A)$.
  So, there is an NFA with ϵ-moves that recognises L.

- *($\Leftarrow$) Assume that $A = (\Sigma, Q, q_0, F, \delta)$ is an NFA with ϵ-moves that recognises L.*
  Let A' be the determinised version of A.
  According to Theorem 2.9, $\mathcal{L}(A) = \mathcal{L}(A')$.
  So, there is a DFA that recognises L.

Remark 2.16 *The determinised version of a NFA of states Q can have as many as $|\mathcal{P}(Q)|$ (i.e. $2^{|Q|}$) states. NFAs can represent exactly the same languages as DFAs can, but often in an extremely more compact way. This compacity, however, comes at the cost of determinism. Or, conversely, the determinism of DFAs comes at the cost of space.*

2.3 Regular grammars

Definition 2.21 (Grammar) *A* regular grammar *G is a quadruple (N, Σ, P, S) where:*

- *N is a finite set the elements of which are called 'non-terminal symbols';*

- *Σ is a finite set the elements of which are called 'terminal symbols';*
- *P is a subset of $N \times (\{\epsilon\} \cup \Sigma \cup (\Sigma\, N))$ and the elements of which are called 'production rules';*
- *S is an element of N called 'the axiom';*

and such that $N \cap \Sigma = \emptyset$ (they are disjoint sets).

Notation 2.3 *A production rule (of a regular grammar) is a pair (X, γ) where $X \in N$ and $\gamma \in (\{\epsilon\} \cup \Sigma \cup (\Sigma\, N))$. Such a rule is usually written '$X \to \gamma$'.*

For example, $(S, a\, S)$ is usually written '$S \to a\, S$'.

Example(s) 2.26 *$(\{A, B\}, \{a, b\}, \{A \to \epsilon, A \to a\, B, B \to b\, A\}, A)$ is a regular grammar consisting of:*

- *two non-terminal symbols: A and B;*
- *two terminal symbols: a and b;*
- *three production rules: $A \to \epsilon$, $A \to a\, B$ and $B \to b\, A$.*

Its axiom is A.

Notation 2.4 (Production rules sharing the same left-hand side) *Given a non-terminal symbol X, a collection of production rules $X \to \gamma_1$, $X \to \gamma_2$, $\cdots$, $X \to \gamma_n$ can be written '$X \to \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_n$'.*

Definition 2.22 (Derivation step) *Given a regular grammar $G = (N, \Sigma, P, S)$, a* derivation step *of G is a quadruple $(\alpha, i, X \to \gamma, \beta) \in (N \cup \Sigma)^\star \times \mathbb{N} \times P \times (N \cup \Sigma)^\star$ such that:*

- *$i \in [\![1, |\alpha|]\!]$ and $\alpha_i = X$;*
- *$\beta = \alpha_{1:i-1}\, \gamma\, \alpha_{i+1:|\alpha|}$.*

α, i, $X \to \gamma$ and β are respectively the source*, the* rewriting position*, the* rule *and the* target *of the derivation step.*

Notation 2.5 *Given a regular grammar $G = (N, \Sigma, P, S)$ and $\alpha, \beta \in (N \cup \Sigma)^\star$, the fact that there exists a derivation step of G of source α and target β is written '$\alpha \underset{G}{\Rightarrow} \beta$'.*

Example(s) 2.27 *Let G be the grammar from example 2.26.*

1. *$b\, a\, A\, b \underset{G}{\Rightarrow} b\, a\, b$ (with a derivation step of rewriting position 3 and rule $A \to \epsilon$).*
2. *$b\, a\, A\, b \underset{G}{\Rightarrow} b\, a\, a\, B\, b$ (with a derivation step of rewriting position 3 and rule $A \to a\, B$).*
3. *$B\, B \underset{G}{\Rightarrow} b\, A\, B$ (with a derivation step of rewriting position 1 and rule $B \to b\, A$).*
4. *$B\, A \underset{G}{\Rightarrow} B\, a\, B$ (with a derivation step of rewriting position 2 and rule $A \to a\, B$).*
5. *$A \underset{G}{\Rightarrow} a\, B$ (with a derivation step of rewriting position 1 and rule $A \to a\, B$).*

Definition 2.23 (Derivation) *Given a regular grammar $G = (N, \Sigma, P, S)$, a* derivation *of G is a finite sequence of derivation steps of G such that for any two subsequent derivation steps, the target of the first is the source of the second.*

When the derivation is non-empty (i.e. of length $n \geq 1$), the source (resp. target) of the first (resp. last) derivation step is the source *(resp.* target*) of the derivation.*

Notation 2.6 *Given a regular grammar $G = (N, \Sigma, P, S)$ and $\alpha, \beta \in (N \cup \Sigma)^\star$, the fact that there exists a non-empty derivation of G of source α and target β is written '$\alpha \underset{G}{\overset{+}{\Rightarrow}} \beta$'.*

Notation 2.7 *Given a regular grammar $G = (N, \Sigma, P, S)$ and $\alpha, \beta \in (N \cup \Sigma)^\star$, the notation '$\alpha \underset{G}{\overset{\star}{\Rightarrow}} \beta$' indicates that:*

- *either $\alpha = \beta$,*
- *or $\alpha \underset{G}{\overset{+}{\Rightarrow}} \beta$.*

Example(s) 2.28 *Let G be the grammar from example 2.26.*

1. $A\,B \underset{G}{\overset{+}{\Rightarrow}} a\,B\,b\,A$ *(with a derivation of length 2), and so also* $A\,B \underset{G}{\overset{\star}{\Rightarrow}} a\,B\,b\,A$.
2. $A \underset{G}{\overset{+}{\Rightarrow}} a\,b\,A$ *(with a derivation of length 2), and so also* $A \underset{G}{\overset{\star}{\Rightarrow}} a\,b\,A$.
3. $A \underset{G}{\overset{+}{\Rightarrow}} a\,b$ *(with a derivation of length 3), and so also* $A \underset{G}{\overset{\star}{\Rightarrow}} a\,b$.
4. $b\,a\,A\,b \underset{G}{\overset{\star}{\Rightarrow}} b\,a\,A\,b$ *(with a derivation of length 0).*

Definition 2.24 (Generation by a regular grammar) *Given a regular grammar $G = (N, \Sigma, P, S)$, a word $w \in \Sigma^\star$ is* generated *by G if it is the target of a non-empty derivation of G of source S.*

Example(s) 2.29 *ϵ, $a\,b$ and $a\,b\,a\,b$ are three words generated by G, the regular grammar from example 2.26. a and $b\,b$ are two words not generated by G.*

Remark 2.17 *Given a regular grammar $G = (N, \Sigma, P, S)$, a word $w \in \Sigma^\star$ is generated by G iff $S \underset{G}{\overset{\star}{\Rightarrow}} w$.*

Definition 2.25 (Language generated by a regular grammar) *Given a regular grammar $G = (N, \Sigma, P, S)$, the* language generated by G*, written '$\mathcal{L}(G)$', is the set of all words generated by G: $\mathcal{L}(G) = \{w \in \Sigma^\star \mid w \text{ is generated by } G\}$.*

Example(s) 2.30 *The language generated by the regular grammar from example 2.26 is $\{(a\,b)^n \mid n \in \mathbb{N}\}$.*

This (fairly intuitive) results can be proven by proving, by induction on $k \in \mathbb{N}^*$, (i) that there are exactly two derivations of G of source A and length $2k - 1$, and that their target is respectively $(a\,b)^{k-1}$ and $(a\,b)^{k-1}\,a\,B$, and (ii) that there is exactly one derivation of G of source A and length $2k$, and that its target is $(a\,b)^k\,A$.

Remark 2.18 (Right and left linear grammars) *The grammars defined in definition 3.14 are also called 'right linear grammars', because in a rule of the form $A \to a\,B$, the non-terminal in the right-hand side (B) is on the right. It is possible to define a class of grammars called 'left linear grammars' by allowing rules of the form $A \to B\,a$ instead of rules of the form $A \to a\,B$. It can be shown that the languages recognised by left linear grammars are exaclty the languages recognised by right linear grammars; these two classes of grammars are equivalent.*

Allowing both rules of the form $A \to a\,B$ and rules of the form $A \to B\,a$, however, leads to a strictly more powerful formalism. $\{a^n\,b^n \mid n \in \mathbb{N}\}$, for instance, is not recognised by any regular grammar but is recognised by $(\{S, X\}, \{a, b\}, \{S \to \epsilon \mid a\,b \mid a\,X, X \to S\,b\}, S)$.

2.4 Equivalences between formalisms

Definition 2.26 (Regular grammar equivalent of a DFA) *Let $A = (\Sigma, Q, q_0, F, \delta)$ be a DFA. The* regular grammar equivalent of A *is the regular grammar $G = (N, \Sigma, P, S)$ where*

- *$N = Q$ (the non-terminal symbols of the regular grammar G are the states of A),*
- *$P = \{q \to \epsilon \mid q \in F\} \cup \{q \to x\,\delta((q, x)) \mid (q, x) \in Q \times \Sigma\}$,*
- *$S = q_0$ (the axiom of the regular grammar G is the initial state of A).*

Example(s) 2.31 *The regular grammar equivalent of the DFA from example 2.7 (graphically represented in figure 2.1) is $G = (\Sigma, \{q_0, q_1, q_2\}, P, q_0)$ where $P = \left\{ \begin{array}{l} q_0 \to \epsilon \mid a\,q_0 \mid b\,q_1, \\ q_1 \to a\,q_1 \mid b\,q_2, \\ q_2 \to a\,q_2 \mid b\,q_0 \end{array} \right\}$.*

Theorem 2.11 *Given a DFA A and its regular grammar equivalent G, $\mathcal{L}(A) = \mathcal{L}(G)$.*

Proof 2.11 *TODO*

Definition 2.27 (NFA equivalent of a regular grammar) *Let $G = (N, \Sigma, P, S)$ be a regular grammar. The* NFA equivalent of G *is the NFA $A = (\Sigma, Q, q_0, F, \delta)$ where*

- *$Q = N \cup \{Z\}$, where Z is anything* not *in N (the states of the NFA A are the non-terminal symbols of G plus an additional one),*
- *$q_0 = S$ (the initial state of the NFA A is the axiom of G),*
- *$F = \{X \mid X \to \epsilon \in P\} \cup \{Z\}$,*
- *δ is the function from $Q \times \Sigma$ to Q defined as $(X, x) \mapsto \{Y \mid X \to x\,Y \in P\} \cup \{Z \mid X \to x \in P\}$.*

Example(s) 2.32 *The NFA equivalent of the regular grammar from example 2.26 is $A = (\{a, b\}, \{A, B, Z\}, A, \{A, Z\}, \delta)$ where $\delta = (X, x) \mapsto \left\{ \begin{array}{l} \{B\} \text{ if } X = A \text{ and } x = a \\ \{A\} \text{ if } X = B \text{ and } x = b \\ \emptyset \text{ otherwise} \end{array} \right.$. This NFA is represented in figure 2.8. (The state Z is 'useless' in this case; this is due to the fact that there is no production rule of the form $X \to x$ in P.)*

Theorem 2.12 *Given a regular grammar G and its NFA equivalent A, $\mathcal{L}(G) = \mathcal{L}(A)$.*

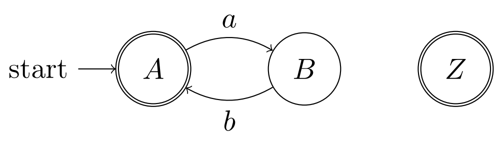


Figure 2.8: Graphical representation of NFA equivalent of the regular grammar from example 2.26.

Proof 2.12 *TODO*

Theorem 2.13 (Equivalence of regular grammars and DFAs) *Let L be a language on some alphabet Σ; there is a regular grammar that generates L iff there is a DFA that recognises L.*

Proof 2.13 (Equivalence of regular grammars and DFAs) *Let L be a language on some alphabet Σ.*

- *($\Rightarrow$) Assume that G is a regular grammar that generates L.*
  Consider A the NFA equivalent of G.
  A is an NFA that recognises L.
  Consider A' the determinised version of A.
  A' is a DFA that recognises L.
  So, there is a DFA that recognises L.
- *($\Leftarrow$) Assume that A is a DFA that recognises L.*
  The regular grammar equivalent of A is a regular grammar that generates L.
  So, there is a regular grammar that generates L.

Definition 2.28 (Regular expression equivalent of a DFA) *TODO (state elimination method)*

Example(s) 2.33 *TODO*

Theorem 2.14 *Given a DFA A and its regular expression equivalent ϕ, $\mathcal{L}(A) = \mathcal{L}(\phi)$.*

Proof 2.14 *TODO*

Definition 2.29 (NFA equivalent of a regular expression) *Given some alphabet Σ, the* NFA equivalents *of the regular expressions of Σ are NFAs without any incoming transition to their initial state and with a single final state without any outgoing transition, defined inductively by:*

- *the NFA equivalent of $\varnothing$, illustrated in figure 2.9, is $(\Sigma, \{q_{i,\varnothing}, q_{f,\varnothing}\}, q_{i,\varnothing}, \{q_{f,\varnothing}\}, \delta_\varnothing)$ where $\delta_\varnothing = (q, x) \mapsto \emptyset$, and the NFA equivalent of ε, illustrated in figure 2.10, is $(\Sigma, \{q_{i,\varepsilon}, q_{f,\varepsilon}\}, q_{i,\varepsilon}, \{q_{f,\varepsilon}\}, \delta_\varepsilon)$ where $\delta_\varepsilon = (q, x) \mapsto \begin{cases} \{q_{f,\varepsilon}\} \text{ if } q = q_{i,\varepsilon} \text{ and } x = \epsilon \\ \emptyset \text{ otherwise} \end{cases}$;*
- *for any $a \in \Sigma$, the NFA equivalent of a, illustrated in figure 2.11, is $(\Sigma, \{q_{i,a}, q_{f,a}\}\, q_{i,a}, \{q_{f,a}\}, \delta_a)$ where $\delta_a = (q, x) \mapsto \begin{cases} \{q_{f,a}\} \text{ if } q = q_{i,a} \text{ and } x = a \\ \emptyset \text{ otherwise} \end{cases}$;*
- *for any regular expression ϕ of NFA equivalent $(\Sigma, Q, q_i, \{q_f\}, \delta)$, the NFA equivalent of $(\phi)^\star$, illustrated in figure 2.12, is $(\Sigma, (Q \cup \{q_{i,s}, q_{f,s}\}), q_{i,s}, \{q_{f,s}\}, \delta_s)$ where $q_{i,s}, q_{f,s} \notin Q$ and $\delta_s = (q, x) \mapsto \begin{cases} \{q_i, q_{f,s}\} \text{ if } q = q_{i,u} \text{ and } x = \epsilon \\ \{q_i, q_{f,s}\} \text{ if } q = q_f \text{ and } x = \epsilon \\ \delta((q, x)) \text{ otherwise and if } q \in Q \\ \emptyset \text{ otherwise} \end{cases}$;*

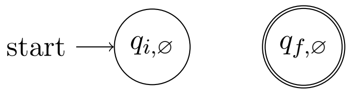


Figure 2.9: Graphical representation of the NFA equivalent of the regular expression $\varnothing$.

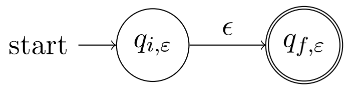


Figure 2.10: Graphical representation of the NFA equivalent of the regular expression ε.

- *for any two regular expressions ϕ_1 and ϕ_2 of NFA equivalent $(\Sigma, Q_1, q_{i,1}, \{q_{f,1}\}, \delta_1)$ and $(\Sigma, Q_2, q_{i,2}, \{q_{f,2}\}, \delta_2)$ respectively (after having renamed the states in Q_2 so that $(Q_1 \cap Q_2) = \emptyset$),*
  - *the NFA equivalent of $(\phi_1 \,|\, \phi_2)$, illustrated in figure 2.13, is $(\Sigma, (Q_1 \cup Q_2 \cup \{q_{i,u}, q_{f,u}\}), q_{i,u}, \{q_{f,u}\}, \delta_u)$ where $q_{i,u}, q_{f,u} \notin (Q_1 \cup Q_2)$ and $\delta_u = (q, x) \mapsto \begin{cases} \{q_{i,1}, q_{i,2}\} \text{ if } q = q_{i,u} \text{ and } x = \epsilon \\ \{q_{f,u}\} \text{ if } q = q_{f,1} \text{ and } x = \epsilon \\ \{q_{f,u}\} \text{ if } q = q_{f,2} \text{ and } x = \epsilon \\ \delta_1((q,x)) \text{ otherwise and if } q \in Q_1 \\ \delta_2((q,x)) \text{ otherwise and if } q \in Q_2 \\ \emptyset \text{ otherwise} \end{cases}$,*
  - *the NFA equivalent of $(\phi_1\, \phi_2)$, illustrated in figure 2.14, is $(\Sigma, (Q_1 \cup Q_2), q_{i,1}, \{q_{f,2}\}, \delta_c)$ where $\delta_c = (q, x) \mapsto \begin{cases} \{q_{i,2}\} \text{ if } q = q_{f,1} \text{ and } x = \epsilon \\ \delta_1((q,x)) \text{ otherwise and if } q \in Q_1 \\ \delta_2((q,x)) \text{ if } q \in Q_2 \end{cases}$.*

This definition is called 'Thompson's construction'.

Example(s) 2.34 *The NFA equivalent of the regular expression $(a \mid b)^\star\, b$ is illustrated in figure 2.15.*

Remark 2.19 *Thomspon's construction (i.e. Definition 2.29) shows, among other things:*

- *how to build a non-deterministic finite-state automaton with ϵ-moves that recognises the Kleene closure of the language recognised by a given non-deterministic finite-state automaton with ϵ-moves,*
- *how to build a non-deterministic finite-state automaton with ϵ-moves that recognises the union of the languages recognised by two given non-deterministic finite-state automata with ϵ-moves.*
- *how to build a non-deterministic finite-state automaton with ϵ-moves that recognises the concatenation of the languages recognised by two given non-deterministic finite-state automata with ϵ-moves,*

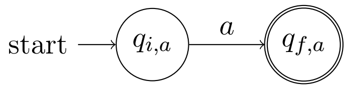


Figure 2.11: Graphical representation of the NFA equivalent of the regular expression a for $a \in \Sigma$.

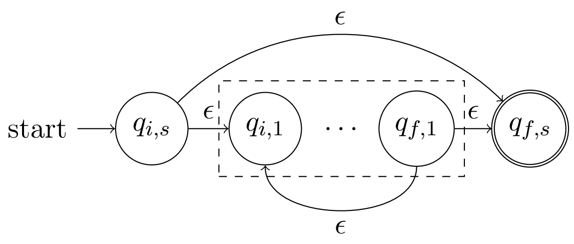


Figure 2.12: Schematic representation of the NFA equivalent of the regular expression $(\phi)^\star$ for a regular expression ϕ.

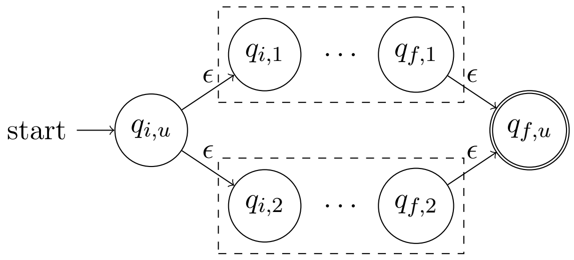


Figure 2.13: Schematic representation of the NFA equivalent of the regular expression $(\phi_1 \mid \phi_2)$ for two regular expressions ϕ_1 and ϕ_2.

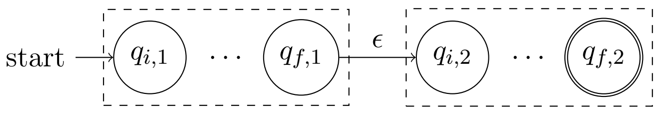


Figure 2.14: Schematic representation of the NFA equivalent of the regular expression $(\phi_1\,\phi_2)$ for two regular expressions ϕ_1 and ϕ_2.

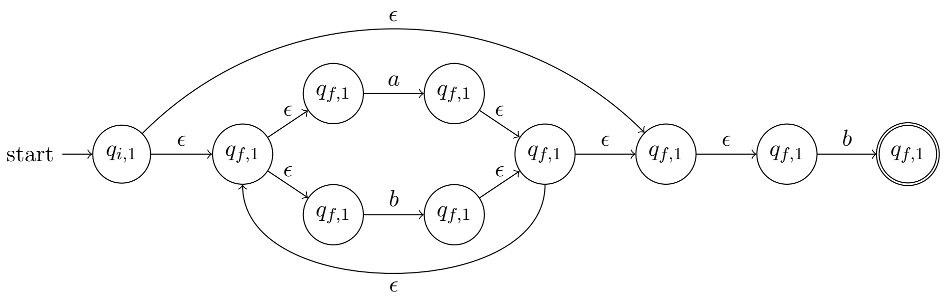


Figure 2.15: Graphical representation of the NFA equivalent of the regular expression $(a \mid b)^\star\, b$.

Theorem 2.15 *Given a regular expression ϕ and its NFA equivalent A, $\mathcal{L}(\phi) = \mathcal{L}(A)$.*

Proof 2.15 *TODO*

Theorem 2.16 (Equivalence of regular expressions and DFAs) *Let L be a language on some alphabet Σ; there is a regular expression that generates L iff there is a DFA that recognises L.*

Proof 2.16 (Equivalence of regular expressions and DFAs) *Let L be a language on some alphabet Σ.*

- *($\Rightarrow$) Assume that ϕ is a regular expression that generates L.*
  Consider A the NFA equivalent of ϕ.
  A is an NFA that recognises L.
  Consider A' the determinised version of A.
  A' is a DFA that recognises L.
  So, there is a DFA that recognises L.

- *($\Leftarrow$) Assume that A is a DFA that recognises L.*
  The regular expression equivalent of A is a regular expression that generates L.
  So, there is a regular expression that generates L.

Remark 2.20 *Theorems 2.13 and 2.16 together state that regular expressions, DFAs and regular grammars have the same expressive power, in the sense that these three types of object can encode exactly the same formal languages. Together with theorem 2.4, they state that these formal languages are the regular languages.*

2.5 Closure properties and pumping lemma

Theorem 2.17 (Closure properties) *The set of regular languages on an alphabet Σ is closed under union, concatenation, Kleene closure, intersection and complementation. In other words, if L_1 and L_2 are two regular languages, then $L_1 \cup L_2$, $L_1 L_2$, $L_1{}^\star$, $L_1 \cap L_2$ and $\overline{L_1}$ (i.e. $\Sigma^\star \setminus L_1$) are regular languages.*

Proof 2.17 (Closure properties) *That the set of regular languages on an alphabet Σ is closed under union, concatenation and Kleene closure is trivial from the definition of the regular languages on Σ (Definition 2.1).*

That the set of regular languages on an alphabet Σ is closed under intersection is given by the construction of product automata (see Definition 2.14 and Theorem 2.6) and the fact that the languages recognised by the DFAs on Σ are the regular languages on Σ.

That the set of regular languages on an alphabet Σ is closed under complement is given by the construction of complement automata (see Definition 2.15 and Theorem 2.7) and the fact that the languages recognised by the DFAs on Σ are the regular languages on Σ.

Theorem 2.18 (Pumping lemma) *If L is a regular language on alphabet Σ, then there exists a $k \in \mathbb{N}$ such that:*

$$\forall w \in L \text{ s.t. } |w| \geq k,\ \exists x, y, z \in \Sigma^\star,\ w = x\,y\,z,\ |x\,y| \leq k,\ |y| > 0 \text{ and } (x\,y^\star\,z) \subseteq L$$

Remark 2.21 (The pumping property) *A language L on an alphabet Σ is said to have the pumping property iff there exists a $k \in \mathbb{N}$ such that:*

$$\forall w \in L \text{ s.t. } |w| \geq k,\ \exists x, y, z \in \Sigma^\star,\ w = x\,y\,z,\ |x\,y| \leq k,\ |y| > 0 \text{ and } (x\,y^\star\,z) \subseteq L$$

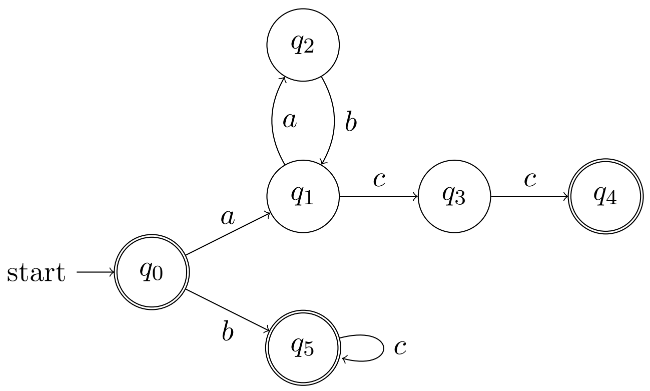


Figure 2.16: Graphical representation of a DFA that recognises $\{a\,(a\,b)^n\,c\,c \mid n \in \mathbb{N}\} \cup \{b\,c^n \mid n \in \mathbb{N}\}$.

The pumping lemma states that all regular languages have the pumping property.

Intuition 2.4 (Pumping lemma; in terms of regular grammar) *Consider an infinite regular language L and a regular grammar $G = (N, \Sigma, P, S)$ such that $L = \mathcal{L}(G)$. For any word $w \in \mathcal{L}(G)$ above a certain length k, w must have been derived via a loop on some non-terminal A:*

- $S \overset{\star}{\underset{G}{\Rightarrow}} x\,A$;
- $A \overset{\star}{\underset{G}{\Rightarrow}} y\,A$ *with* $|y| > 0$;
- $A \overset{\star}{\underset{G}{\Rightarrow}} z$;

with $w = x\,y\,z$; the additional property $|x\,y| \leq k$ corresponds to the fact that the second occurrence of A must have been generated within the first k steps.

Intuition 2.5 (Pumping lemma; in terms of DFAs) *Consider $L = \{a\,(a\,b)^n\,c\,c \mid n \in \mathbb{N}\} \cup \{b\,c^n \mid n \in \mathbb{N}\}$. L is recognised by the DFA A represented in figure 2.16. Consider any word $w \in L$ such that the recognition of w by A involves going at least twice through a same state. For instance, $w = a\,a\,b\,c\,c$, the recognition of which involves going twice through q_1 (once after reading the first a and once after reading the b). Let $x = a$, the prefix of w read before reaching q_1 for the first time, $z = c\,c$, the suffix of w read after leaving q_1 for the second time, and $y = a\,b$, the subword of w read in between. $w = x\,y\,z$ and for any $n \in \mathbb{N}$, $x\,y^n\,z = a\,(a\,b)^n\,c\,c$ is also recognised by A.*

Now, is there necessarily a threshold k such that the recognition of any $w \in L$ of length $|w| \geq k$ involves going twice through a same state? Yes. For any DFA A, the number of states of A is a suitable threshold k; the recognition of any $w \in L$ of length $|w| \geq k$ involves going twice through a same state.

Note that a DFA does not necessarily contain a loop. But even for such a DFA A is the number of states of A a suitable threshold k. There will simply be no $w \in L$ of length $|w| \geq k$, making the quantification in the pumping lemma trivially true. What the pumping lemma states is that any regular language is either finite (no loop) or is infinite due to the repetition of one or more patterns (loops).

Proof 2.18 (Pumping lemma) *Consider L, a regular language on alphabet Σ.*
Consider A, a deterministic finite-state automaton such that $L = \mathcal{L}(A)$.

Let k be the size of A, i.e. the number of its states.
Consider any $w \in L$ such that $|w| \geq k$ and let $(s_i)_{0 \leq i \leq |w|}$ be the sequence of states visited during the acceptance of w by A (s_0 is the start state of A and $s_{|w|}$ is a final state of A).
Because there are only k states in A, there must be a state s appearing (at least) twice in the first $k+1$ states of $(s_i)_i$: $\exists i_1, i_2 \in [\![0, k]\!],\ i_1 < i_2,\ s_{i_1} = s_{i_2} = s$.
Let $x = w_1\, w_2 \cdots w_{i_1}$, $y = w_{i_1+1}\, w_{i_1+2} \cdots w_{i_2}$ and $z = w_{i_2+1}\, w_{i_2+2} \cdots w_{|w|}$.
It is the case that $w = x\, y\, z$.
It is the case that $|x\, y| = i_2 \leq k$.
It is the case that $|y| = i_2 - i_1 > 0$.
Lastly, let us prove that $(x\, y^\star\, z) \subseteq L$.
Consider any $n \in \mathbb{N}$.
During the reading of $x\, y^n\, z$, the sequence of visited states is:

$$s_0 \xrightarrow{x_1} s_1 \xrightarrow{x_2} \cdots \xrightarrow{x_{i_1}} s_{i_1} \xrightarrow{y_1} (s_{i_1+1} \xrightarrow{y_2} s_{i_1+2} \xrightarrow{y_2} \cdots \xrightarrow{y_{i_2-i_1}} s_{i_2})^n \xrightarrow{z_1} s_{i_2+1} \xrightarrow{z_2} s_{i_2+2} \xrightarrow{z_3} \cdots \xrightarrow{z_{|w|-i_2}} s_{|w|}$$

(The transition from s_{i_2} (which is also s_{i_1}) to s_{i_1+1}, not explicit here, is labelled y_1 (i.e. w_{i_1+1}.)
This sequence ends on a final state ($s_{|w|}$ is a final state because $w \in \mathcal{L}(A)$), which means that $x\, y^n\, z \in \mathcal{L}(A)$, i.e. $x\, y^n\, z \in L$.

Remark 2.22 *The fundamental idea behind the pumping lemma is that because finite-state automata have a finite number of states, any long computation by one of them must involve a loop plus a limited number of other operations before and after the loop.*

Remark 2.23 (Usage of the pumping lemma) *The pumping lemma can sometimes be used to show that a given language L is not regular. To do so, one has to show that for any $k \in \mathbb{N}$, there is a $w \in L$ of length $|w| \geq k$ that does not have the properties mentioned in the lemma.*

The pumping lemma, however, cannot be used to show that a given language is regular. This is because the pumping property does not characterise regular languages, in the sense that some languages that are not regular still have this property. (See Exercise 2.2 for an example.)

Remark 2.24 (Using the pumping lemma in practice) *Imagine that one is considering some language L on alphabet Σ that they believe is not regular. If they are lucky, L is indeed not regular and this fact can be proved using the pumping lemma.*

If L were regular, then, according to the pumping lemma, there would be some $k \in \mathbb{N}$ such that $\forall w \in L$ s.t. $|w| \geq k$, $\exists x, y, z \in \Sigma^\star$, $w = x\, y\, z$, $|x\, y| \leq k$, $|y| > 0$ and $(x\, y^\star\, z) \subseteq L$. To build a proof by contradiction, one can assume that L is regular and thus that there is such a k. To obtain a contradiction, one just needs to exhibit some $w \in L$ s.t. $|w| \geq k$ that cannot be decomposed as $w = x\, y\, z$ for any $x, y, z \in \Sigma^\star$ s.t. $|x\, y| \leq k$, $|y| > 0$ and $(x\, y^\star\, z) \subseteq L$. If such a w is found, then the contradiction proves that L is not regular.

Exercise 2.2 *Let $\Sigma = \{a, b\}$.*

1. *Use the pumping lemma to prove that $L = \{a\, b^n\, a^n \mid n \in \mathbb{N}\}$ is not regular.*

2. *Prove that $L' = L \cup \{a^n\, w' \mid n \in \mathbb{N} \setminus \{1\},\ w' \in \Sigma^\star\}$ has the pumping property.*

3. *Prove that L' is not regular.*

2.6 Advanced: Computability

Notation 2.8 ($\mathbb{B}$) *The set* $\{0,1\}$ *can be written '*$\mathbb{B}$*'. This notation refers to the name of George Boole.*[1]

Definition 2.30 (Binary function) *Given some set* S*, a* binary function on S *is any function from* S *to* $\mathbb{B}$*.*

Example(s) 2.35

1. *The following function is a binary function on* $\mathbb{N}$*:* $n \in \mathbb{N} \mapsto \begin{cases} 1 \text{ if } \exists k \in \mathbb{N},\ n = 3k \\ 0 \text{ otherwise} \end{cases}$.
2. *The following function is a binary function on* $\mathbb{N}$*:* $n \in \mathbb{N} \mapsto \begin{cases} 1 \text{ if } n \text{ is prime} \\ 0 \text{ otherwise} \end{cases}$.
3. *For a set* Σ *of English words and punctuation marks, the following function is a binary function on* $\Sigma^\star$*:* $s \in \Sigma^\star \mapsto \begin{cases} 1 \text{ if } s \text{ is a grammatical English sentence} \\ 0 \text{ otherwise} \end{cases}$.
4. *For* P *any set of planets in the universe, the following function is a binary function on* P*:* $p \in P \mapsto \begin{cases} 1 \text{ if there is currently life on } p \\ 0 \text{ otherwise} \end{cases}$.
5. *The following function is a binary function on* $\mathbb{B}^\star$*:*
   $w \in \mathbb{B}^\star \mapsto \begin{cases} 1 \text{ if } \exists u, v \in \mathbb{B}^\star,\ w = u\,v \text{ and } v \text{ is the Zip compression of } u \\ 0 \text{ otherwise} \end{cases}$.
6. *For* S *some set of elements encoding meteorological data,* l *some location and* t *some point in time, the following function is a binary function on* S*:*
   $s \in S \mapsto \begin{cases} 1 \text{ if, according to the data in } s, \text{ the chance of rain at } l \text{ and } t \text{ is above } 50\% \\ 0 \text{ otherwise} \end{cases}$.

Remark 2.25 *As already illustrated by the previous examples, many problems can been reduced to the computation of some binary function on some set* S*. Often, one can furthermore assume that* $S = \Sigma^\star$ *for some alphabet* Σ*; this assumption corresponds to the possibility of encoding the input of the problem as a word on* Σ*. In practice, modern computers represent all sorts of data as sequences of* 0*·s and* 1*·s, i.e. as words on* $\mathbb{B}$*.*

Note, however, that the fact that some function is mathematically definable does not mean that is it actually computable.

Definition 2.31 (Characteristic function) *Given two sets* $S' \subseteq S$*, the* characterisitic function of S' in S *is the binary function* f *on* S *that sends any member of* S *to* 1 *iff it is also a member of* S' *(and to* 0 *otherwise):* $f = x \in S \mapsto \begin{cases} 1 \text{ if } x \in S' \\ 0 \text{ otherwise} \end{cases}$.

Remark 2.26 (Binary functions as characteristic functions) *Given some set* S*, any binary function* f *on* S *is the characteristic function of some set in* S*, namely, the set* S' *that contains exactly the elements of* S *sent to* 1 *by the function* f*:* $S' = \{x \in S \mid f(x) = 1\}$*.*

Remark 2.27 (DFA and computation) *Each DFA* A *implicitely encodes some binary function on* $\Sigma^\star$*, namely, the function* f_A *that sends any word on* Σ *to* 1 *iff this word is accepted*

[1] See `https://en.wikipedia.org/wiki/George_Boole`.

by A (and to 0 otherwise): $f_A = w \in \Sigma^\star \mapsto \begin{cases} 1 \text{ if } A \text{ accepts } w \\ 0 \text{ otherwise} \end{cases}$. Thus, f_A is the characteristic function of $\mathcal{L}(A)$, the language recognised by A, and $\mathcal{L}(A) = \{w \in \Sigma^\star \mid f_A(w) = 1\}$.

When analysing a word $w \in \Sigma^\star$, the machine executing A (see Remark 2.10) effectively computes $f_A(w)$.

Remark 2.28 (Computation and the emergence of complexity) *When running a DFA A on a word w, each of the computation steps are, in a sense, trivial: they each consist in performing a predefined basic action based only on the current state of A and the letter currently under the tape head. Still, as we have seen, global computation performed by such a machine is not always trivial; in particular, DFAs can differentiates dates and times from other sequences of letters, and perform certain arithmetic operations such as computing remainders of Euclidean divisions (for any arbitrary but fixed divisor). In other words, the computation performed when running a DFA can be* locally *trivial but* globally *non-trivial.*

True, DFAs are quite limited in terms of computation — they can only encode the characteristic functions of regular languages; they cannot, for instance determine whether an integer, written as a sequence of digits, is a prime number or not, or even whether a word contains as many occurrences of a as occurrences of b —, but much more powerful kinds of machines also rely on locally trivial computational steps. The basic operations performed by any modern computer are all trivial and yet, these machines can run any definable algorithm. Similarly, the functioning of an organic brain (human or otherwise) also arguably relies on trivial basic operations.

Vocabulary

- a regular language: *un langage régulier*
- a regular expression: *une expression régulière*
- a finite-state automaton: *un automate fini*
- a computation: *un calcul*
- a successful computation: *un calcul réussi*
- a regular grammar: *une grammaire régulière*

Chapter 3

Phrase structure grammars

3.1 Grammars

Definition 3.1 (Grammar) *A* phrase structure grammar *G is a quadruple (N, Σ, P, S) where:*

- *N is a finite set the elements of which are called 'non-terminal symbols';*
- *Σ is a finite set the elements of which are called 'terminal symbols';*
- *P is a finite subset of $(N \cup \Sigma)^+ \times (N \cup \Sigma)^\star$ and the elements of which are called 'production rules';*
- *S is an element of N called 'the axiom';*

and such that $N \cap \Sigma = \emptyset$ (they are disjoint sets).

Remark 3.1 *An alternative definition of phrase structure grammars requires each production rule to be an element of $((N \cup \Sigma)^\star N (N \cup \Sigma)^\star) \times (N \cup \Sigma)^\star$ — that is to say, to contain at least one non-terminal symbol in its left-hand side (see Remark 3.2). This apparently more constrained definition does not in fact impact the expressive power of the formalism: Any language generated by a phrase structure grammar according to the definition 3.1 (see Definition 3.5) is also generated by a grammar the production rules of which each satisfies this additional constraint.*

Notation 3.1 *A production rule is a pair (α, β) where $\alpha \in (N \cup \Sigma)^+$ and $\beta \in (N \cup \Sigma)^\star$. This means that there is some $m \geq 1$ such that $\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_m) \in (N \cup \Sigma)^m$, and similarly for β with some $n \geq 0$. Such a rule is usually written '$\alpha_1\, \alpha_2\, \cdots\, \alpha_m \rightarrow \beta_1\, \beta_2\, \cdots\, \beta_n$'.*

For example, $((S), (a, S, b))$ is usually written '$S \rightarrow a\, S\, b$'.

Example(s) 3.1 *$(\{S\}, \{a, b\}, \{S \rightarrow a\, S\, b, S \rightarrow a\, b\}, S)$ is a grammar consisting of:*

- *one non-terminal symbol: S;*
- *two terminal symbols: a and b;*
- *two production rules: $S \rightarrow a\, S\, b$ and $S \rightarrow a\, b$.*

Its axiom is S.

Remark 3.2 (Sides of a production rule) *Given a production rule $r = \alpha \rightarrow \beta$,*

- *α is called the 'left-hand side' of r;*

- *β is called the 'right-hand side' of r.*

Remark 3.3 (Alphabet) *The set of terminal symbols of a grammar $G = (N, \Sigma, P, S)$ (i.e. Σ) is also called its 'alphabet'.*

Remark 3.4 (Vocabulary) *The* vocabulary *of a grammar (N, Σ, P, S) is the set $V = N \cup \Sigma$.*

Notation 3.2 (Production rules sharing the same left-hand side) *A collection of production rules $\alpha \to \beta_1$, $\alpha \to \beta_2$, $\cdots$, $\alpha \to \beta_n$, sharing the same left-hand side (here α), can be written '$\alpha \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$'.*

Definition 3.2 (One-step derivation) *Given a grammar $G = (N, \Sigma, P, S)$ and two words on the vocabulary of the grammar $\alpha, \beta \in (N \cup \Sigma)^\star$, β is the result of a* one-step derivation *from α iff:*

- *there exist some γ_1, γ_2, α', β' each in $(N \cup \Sigma)^\star$ such that:*
  - *α can be decomposed as $\alpha = \gamma_1\, \alpha'\, \gamma_2$;*
  - *β can be decomposed as $\beta = \gamma_1\, \beta'\, \gamma_2$;*
- *$\alpha' \to \beta' \in P$ (it is a production rule of G).*

If this is the case, one can write '$\alpha \underset{G}{\Rightarrow} \beta$'.

Example(s) 3.2 *With G the grammar from example 3.1, $a\,S\,b \underset{G}{\Rightarrow} a\,a\,b\,b$, $a\,S\,b \underset{G}{\Rightarrow} a\,a\,S\,b\,b$, $S\,S \underset{G}{\Rightarrow} S\,a\,b$, $S \underset{G}{\Rightarrow} a\,S\,b$.*

Definition 3.3 (Derivation) *Given a grammar $G = (N, \Sigma, P, S)$, two words on the vocabulary of the grammar $\alpha, \beta \in (N \cup \Sigma)^\star$ and $n \in \mathbb{N}$, β is the result of a derivation from α of length n iff there are $\gamma_0, \gamma_1, \cdots, \gamma_n \in (N \cup \Sigma)^\star$ such that $\alpha = \gamma_0 \underset{G}{\Rightarrow} \gamma_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} \gamma_n = \beta$. If this is the case, one can write '$\alpha \overset{\star}{\underset{G}{\Rightarrow}} \beta$'.*

Example(s) 3.3 *With G the grammar from example 3.1, $S \overset{\star}{\underset{G}{\Rightarrow}} a\,a\,b\,b$ (with a derivation of length 2), $S\,S \overset{\star}{\underset{G}{\Rightarrow}} a\,b\,a\,b$ (with derivation of length 2), $S \overset{\star}{\underset{G}{\Rightarrow}} a\,a\,a\,b\,b\,b$ (with a derivation of length 3), $S\,a \overset{\star}{\underset{G}{\Rightarrow}} S\,a$ (with a derivation of length 0).*

Notation 3.3 *Given a grammar $G = (N, \Sigma, P, S)$ and two words on the vocabulary of the grammar $\alpha, \beta \in (N \cup \Sigma)^\star$, the fact that β is the result of a derivation from α of length at least 1 is written '$\alpha \overset{+}{\underset{G}{\Rightarrow}} \beta$'.*

Definition 3.4 (Words and sentential forms) *Given a grammar $G = (N, \Sigma, P, S)$,*

- *a* sentential form *(on (N, Σ)) is any $w \in (N \cup \Sigma)^\star$;*
- *a* sentential form *of G is a $w \in (N \cup \Sigma)^\star$ such that $S \overset{\star}{\underset{G}{\Rightarrow}} w$, i.e. any sequence of terminal or non-terminal symbols that can be derived from the axiom;*
- *a* word *(on Σ) is any $w \in \Sigma^\star$;*

- *a* word *of G is a $w \in \Sigma^\star$ such that $S \overset{\star}{\underset{G}{\Rightarrow}} w$, i.e. any sequence of terminal symbols (only) that can be derived from the axiom.*

Example(s) 3.4 *With G the grammar from example 3.1, $a\,b$ and $a\,a\,b\,b$ are two words of G, and S and $a\,S\,b$ are two sentential forms of G.*

Remark 3.5 *Given a grammar G, any word of G is also a sentential form of G but the converse is not true.*

Definition 3.5 (Language generated by a grammar) *Given a grammar $G = (N, \Sigma, P, S)$, the* language generated by G*, written '$\mathcal{L}(G)$', is the set of all words of G: $\mathcal{L}(G) = \{w \in \Sigma^\star \mid S \overset{\star}{\underset{G}{\Rightarrow}} w\}$.*

Example(s) 3.5 *With G the grammar from example 3.1, $\mathcal{L}(G) = \{a^n\,b^n \mid n \in \mathbb{N}^*\}$.*

Definition 3.6 (Minimal derivation) *A derivation $\gamma_0 \underset{G}{\Rightarrow} \gamma_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} \gamma_n$ is* minimal *iff $\forall i, j \in [\![0, n]\!]$, if $i \neq j$ then $\gamma_i \neq \gamma_j$.*

Example(s) 3.6 *Let $G = (\{S, A, B, C\}, \{a, b, c\}, P, S)$ where* $P = \left\{ \begin{array}{l} S \to A \mid B\,C, \\ A \to B \mid a, \\ B \to A \mid \epsilon, \\ C \to B\,C \mid c \end{array} \right\}$.

- $S \underset{G}{\Rightarrow} A \underset{G}{\Rightarrow} B \underset{G}{\Rightarrow} A \underset{G}{\Rightarrow} B \underset{G}{\Rightarrow} A \underset{G}{\Rightarrow} a$ *is not minimal, and* $S \underset{G}{\Rightarrow} A \underset{G}{\Rightarrow} a$ *is.*
- $S \underset{G}{\Rightarrow} B\,C \underset{G}{\Rightarrow} C \underset{G}{\Rightarrow} B\,C \underset{G}{\Rightarrow} C \underset{G}{\Rightarrow} c$ *is not minimal, and* $S \underset{G}{\Rightarrow} B\,C \underset{G}{\Rightarrow} C \underset{G}{\Rightarrow} c$ *is.*

Theorem 3.1 (Sufficiency of minimal derivations) *Given a grammar $G = (N, \Sigma, P, S)$ and $\alpha, \beta \in (N \cup \Sigma)^\star$ such that $\alpha \overset{\star}{\underset{G}{\Rightarrow}} \beta$, then there is a minimal derivation $\alpha = \gamma_0 \underset{G}{\Rightarrow} \gamma_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} \gamma_n = \beta$.*

In other words, if a sequence is derivable from another, it can be derived with a minimal derivation.

Proof 3.1 (Sufficiency of minimal derivations) *Given a grammar $G = (N, \Sigma, P, S)$ and $\alpha, \beta \in (N \cup \Sigma)^\star$ such that $\alpha \overset{\star}{\underset{G}{\Rightarrow}} \beta$, let us prove that β can be derived from α with a minimal derivation.*

Consider $Q = \{n \in \mathbb{N} \mid \exists$ a derivation of length n from α to β in $G\}$. By hypothesis, there is at least one derivation of β from α in G, and so $Q \neq \emptyset$. Because Q is a non-empty set of natural numbers, it has a smallest element $n = \min Q$. So, there is a derivation of length n from α to β in G: There are $\gamma_0, \gamma_1, \cdots, \gamma_n \in (N \cup \Sigma)^\star$ such that $\alpha = \gamma_0 \underset{G}{\Rightarrow} \gamma_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} \gamma_n = \beta$. Let us prove by contradiction that this derivation is a minimal derivation.

Let us assume that this derivation is not a minimal one. Then, there exist $i, j \in [\![0, n]\!]$ with $i < j$ such that $\gamma_i = \gamma_j$. Then, $\alpha = \gamma_0 \underset{G}{\Rightarrow} \gamma_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} \gamma_i \underset{G}{\Rightarrow} \gamma_{j+1} \underset{S}{\Rightarrow} \gamma_{j+2} \underset{S}{\Rightarrow} \cdots \underset{S}{\Rightarrow} \gamma_n = \beta$ is a derivation of length $n - (j - i)$. So, $n - (j - i) \in Q$ and $n - (j - i) < n$ which contradict the fact that $n = \min Q$: We have a contradiction. By contradiction, this proves that the derivation of length n considered is a minimal derivation.

Thus, there is at least one minimal derivation of β from α in G.

Remark 3.6 *Theorem 3.1 entails that when trying to determine the language generated by a grammar (see Definition 3.5), one can consider minimal derivations only.*

Remark 3.7

- *Phrase structure grammars are systems that generate words through derivations.*
- *As will be formalised later, a derivation of a word corresponds to a (syntactic) structure of this word.*
- *A derivation or, equivalently, a syntactic structure, can be seen as a proof that the corresponding sequence is a word of the grammar under consideration.*

3.2 Chomsky(-Schützenberger) hierarchy

- Noam Chomsky[1] and Marcel-Paul Schützenberger[2] have contributed to the definition of three types of phrase structure grammars in addition to the general one introduced above.
- These four types of phrase structure grammars are called 'type n grammars' with $n \in \{0, 1, 2, 3\}$.
- These four types of phrase structure grammars are organised in a hierarchy of increasing complexity in the sense that all type 3 grammars are also type 2 grammars, all type 2 grammars are also type 1 grammars and all type 1 grammars are also type 0 grammars. This structure is illustrated in figure 3.1.
- The type 0 is the most general type of grammars; any phrase structure grammar is a type 0 grammar.
- The three additional types of phrase structure grammars are defined through restrictions of the kind of production rules allowed in the grammar.
- To each type of grammars naturally corresponds the sets of all languages that can be generated by a grammar of this type. So, the hierarchy is also a hierarchy of sets (or 'classes') of formal languages.
- The more constrained the production rules are, the less *expressive* the grammars can be and the less *complex* the languages can be.
- Informally, a simple language exhibits a lot of regularity, is easy to describe and can be recognised easily (in the sense that the task of determining whether a sequence of symbols is a word of this language — i.e. is grammatically correct — is a simple task).
- Other classes of formal languages, not included in the original hierarchy, can be easily defined. For example, the class of all finite languages.
- The type 0 class is the largest class of the hierarchy, but not all formal languages belong to this class. Languages that are not type 0 cannot be described by a phrase-structure grammar, nor by any other computational formalism.
- One of the most interesting questions in the field for linguists is: What class characterises best natural languages?

[1] `https://en.wikipedia.org/wiki/Noam_Chomsky`

[2] `https://en.wikipedia.org/wiki/Marcel-Paul_Sch%C3%BCtzenberger`

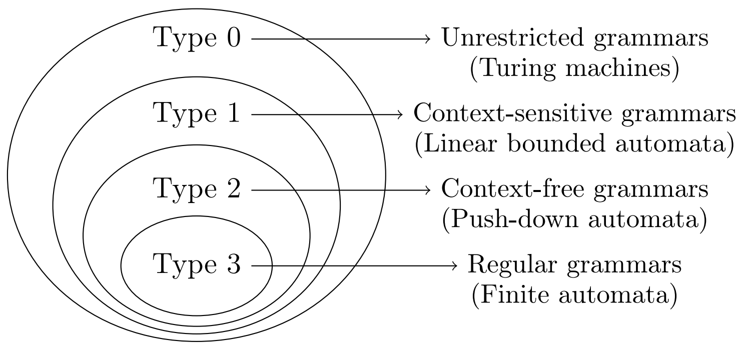

Type 0
Unrestricted grammars (Turing machines)
Type 1
Context-sensitive grammars (Linear bounded automata)
Type 2
Context-free grammars (Push-down automata)
Type 3
Regular grammars (Finite automata)

Figure 3.1: Illustration of the Chomsky hierarchy. (Type 1 and type 2 are the classes of languages recognised by, respectively, non-deterministic linear bounded automata and non-deterministic push-down automata. The distinction between deterministic and non-deterministic machines, however, is not relevant for type 0 and type 3.)

3.2.1 Type 0 (unrestricted) grammars

Definition 3.7 (Type 0 grammar) *A* type 0 grammar *is a phrase structure grammar.*

Example(s) 3.7 $G = (\{S, A, B\}, \{a, b\}, P, S)$ *where* $P = \left\{ \begin{array}{l} S \to A\,B, \\ a\,A \to b\,B, \\ A \to \epsilon \mid a\,A, \\ B\,b \to A\,a, \\ B \to \epsilon \mid b\,B, \\ a\,b \to b\,a \end{array} \right\}$ *is a type 0 grammar.*

Remark 3.8 *Type 0 grammars are also called 'unrestricted grammars'.*

Definition 3.8 (Recursively enumerable set) *A set of words on some alphabet Σ is* recursively enumerable *iff there is an algorithm (e.g. a Turing machine, a program in Python) the output of which is a list of all words of the set. (In the case of an infinite set, the output is infinite and the algorithm never halts.)*

Remark 3.9 (Alternative formulation) *The notion of a recursively enumerable set is equivalent to the notion of a* semidecidable *set. A set of words on some alphabet Σ is semidecidable iff there is an algorithm such that:*

- *for any word of the set given as input, the algorithm halts and outputs 1;*
- *for any other word given as input, the algorithm does not halt or halts and outputs 0;*

Theorem 3.2 *A set is recursively enumerable iff it is the language generated by some type 0 grammar.*

Remark 3.10

- *This means that for any set L, if there exist an algorithm able to list exactly the elements of S, then there exists a phrase structure grammar G such that $L = \mathcal{L}(G)$.*
- *In other words, any sets of words that can be characterised computationally can be characterised by a phrase structure grammar. Type 0 grammar have thus the maximal expressive power one can theoretically expect from a (computational) grammar formalism.*

- *There are formal languages that are not recursively enumerable*[3]*, but they can be quite hard to think about. They will not be of any interest to us here. (In fact, many languages that are recursively enumerable are already quite weird.)*

Example(s) 3.8 *Let P be the set of all algorithms that expect a sequence of bits as input. Let f be an injection from $(P \times \mathbb{B}^\star)$ to $\mathbb{N}$. Such a function* encodes *each pair composed of an algorithm from P and a finite sequence of bits into a natural number, which can be interpreted as an identifier for the pair. Then, $\{a^{f(p,s)} \mid p \in P,\ s \in \mathbb{B}^\star,\ p$ halts on input s (instead of looping indefinitely)$\}$ is a recursively enumerable (or alternatively, semidecidable) set.*[4]

3.2.2 Type 1 (context-sensitive) grammars

Definition 3.9 (Type 1 grammar) *A* type 1 grammar *is a phrase structure grammar $G = (N, \Sigma, P, S)$ such that each rule of P is of one of the following forms:*

- *$S \to \epsilon$, provided that S does not appear in the right-hand side of any rule of P;*
- *$\alpha_1\, A\, \alpha_2 \to \alpha_1\, \gamma\, \alpha_2$ where, $A \in N$, $\alpha_1, \alpha_2 \in (N \cup \Sigma)^\star$ and $\gamma \in (N \cup \Sigma)^+$.*

Remark 3.11 *A type 1 grammar is then such that any production rule only rewrites one non-terminal symbol (A). The other symbols mentioned in the left-hand side of the rule (those constituting α_1 and α_2) form the* context *of the rule. With a type 1 grammar, a one-step derivation does not touch the context of the rule that is applied.*

Example(s) 3.9 *$G = (\{S, S', B, C, H\}, \{a, b, c\}, P, S)$ where $P =$*

$$\left\{\begin{array}{l} S \to \epsilon \mid S', \\ S' \to a\, S'\, B\, C \mid a\, B\, C, \\ C\, B \to H\, B, \\ H\, B \to H\, C, \\ H\, C \to B\, C, \\ a\, B \to a\, b, \\ b\, B \to b\, b, \\ b\, C \to b\, c, \\ c\, C \to c\, c \end{array}\right\} \text{ is a type 1 grammar.}$$

Remark 3.12 *Type 1 grammars are also called 'context-sensitive grammars'.*

Definition 3.10 (Context-sensitive language) *A* context-sensitive language *is any set L such that there exists a context-sensitive grammar G such that $L = \mathcal{L}(G)$.*

Example(s) 3.10 *It can be shown that $L = \{a^n\, b^n\, c^n \mid n \in \mathbb{N}\}$ is the language generated by the grammar G from example 3.9. As G is a context-sensitive grammar, L is a context-sensitive language.*

Definition 3.11 (Decidable set) *A set of words on some alphabet Σ is* decidable *iff there is an algorithm such that:*

- *for any word of the set given as input, the algorithm halts and outputs 1;*

[3] In fact, most sets of words on any (non-empty) alphabet Σ are not recursively enumerable. If you build a language L by considering in turn all (infinitely many) words on Σ and by randomly choosing for each of them whether to include it in L, the language L you build is almost surely (that is, with probability 1) not recursively enumerable. This results come from the fact that true randomness is not computable.

[4] See the *Halting problem* (`https://en.wikipedia.org/wiki/Halting_problem`).

- *for any other word given as input, the algorithm halts and outputs 0;*

Remark 3.13 (Decidability vs semidecidability)

- *With a decidable set of words on an alphabet Σ, there exists an algorithm such that, for any word on Σ, the algorithm can tell in a finite amount of time whether or not this word is a member of the set.*
- *With a strictly semidecidable set of words on an alphabet Σ, there only exist algorithms such that, for any word on Σ that happens to be a member of the set, the algorithms can tell in a finite amount of time that this word is a member of the set; for an infinite number of words not in the set, the algorithms are not able to tell in a finite amount of time that these words are not members of the set.*

Example(s) 3.11 $\{a^n \mid n \in \mathbb{N} \text{ is a prime number}\}$ *is a decidable set.*

Theorem 3.3 *All context-sensitive languages are decidable.*

Theorem 3.4 *Not all decidable sets are context-sensitive. In other words, there are decidable sets such that no context-sensitive grammar can generate exactly these sets.*

3.2.3 Type 2 (context-free) grammars

Definition 3.12 (Type 2 grammar) *A* type 2 grammar *is a phrase structure grammar $G = (N, \Sigma, P, S)$ such that each rule of P is of the following form:*

- $A \to \beta$, *where* $A \in N$ *and* $\beta \in (N \cup \Sigma)^\star$.

Remark 3.14 *A type 2 grammar is then such that any production rule only rewrites one non-terminal symbol (A) without this rewriting being restricted by any context.*

Example(s) 3.12 $G = (\{S\}, \{a, b\}, \{S \to \epsilon \mid a\,S\,b\}, S)$ *is a type 2 grammar.*

Remark 3.15 *Type 2 grammars are also called 'context-free grammars' (CFG).*

Definition 3.13 (Context-free language) *A* context-free language *is any set L such that there exists a context-free grammar G such that $L = \mathcal{L}(G)$.*

Example(s) 3.13 *It can be shown that $L = \{a^n\,b^n \mid n \in \mathbb{N}\}$ is the language generated by the grammar G from example 3.12. As G is a context-free grammar, L is a context-free language.*

Remark 3.16 *According to definition 3.12, context-free grammars are not strictly speaking a kind of context-sensitive grammars. Indeed, according to this definition, context-free grammars may include rules of the form $A \to \epsilon$ for any $A \in N$ while such rules are only allowed in a context-sensitive grammar when A is the axiom of the grammar (and provided that a certain additional condition is met). There is, however, an alternative definition of context-free grammars according to which context-free grammars form a subset of context-sensitive grammars and such that the set of languages generated (the set of context-free languages) is still exactly the same set of languages. This shows, by the way, that the set of context-free languages is a subset of the set of context-sensitive languages.*

Some of the proofs and algorithms studied in this course could be simplified with such an alternative definition constraining the occurrences ϵ-productions in CFG. We still stick to definition 3.12, however, because we will often use CFG to model the syntax of natural language and that, in this context, ϵ-productions are very practical. Indeed, syntactic theories *of natural language often postulate the existence of* empty categories*: syntactic constituents that are neither written nor pronounced.*[5] *For example, it is often assumed that languages, such as Italian or Japanese, which allow (non-infinitive) verbs to be realised without any apparent subject, in fact realise them with an empty pronoun. Figure 3.2 illustrates this with* Ho fame*, the Italian for* I am hungry*, which does not contain any apparent subject. An empty category X can be modelled in a CFG with an ϵ-production $X \to \epsilon$.*

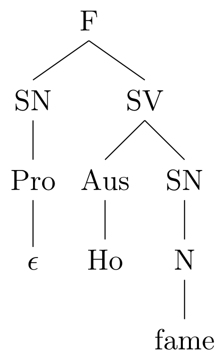


Figure 3.2: A derivation tree (for the Italian sentence *Ho fame*) involving an ϵ-production.

Remark 3.17 *More will be said about context-free grammar in Chapter 4.*

3.2.4 Type 3 (regular) grammars

Definition 3.14 (Type 3 grammar) *A* type 3 grammar *is a phrase structure grammar $G = (N, \Sigma, P, S)$ such that each rule of P is of one of the following forms:*

- $A \to \epsilon$*, where* $A \in N$*;*
- $A \to a$*, where* $A \in N$ *and* $a \in \Sigma$*;*
- $A \to a\,B$*, where* $A \in N$*,* $a \in \Sigma$ *and* $B \in N$*.*

Example(s) 3.14 $G = (\{A, B\}, \{a, b\}, P, A)$ *where* $P = \left\{ \begin{array}{l} A \to \epsilon \mid a\,B, \\ B \to b\,A \end{array} \right\}$ *is a type 3 grammar.*

Remark 3.18 *Type 3 grammars are also called 'regular grammars'.*

Definition 3.15 (Regular language) *A* regular language *is any set L such that there exists a regular grammar G such that $L = \mathcal{L}(G)$.*

Example(s) 3.15 *It can be shown that $L = \{(a\,b)^n \mid n \in \mathbb{N}\}$ is the language generated by the grammar G from example 3.14. As G is a regular grammar, L is a regular language.*

3.3 Computability and complexity

- The formalism of phrase structure grammars can be seen as a programming language, and phrase structure grammars themselves can be seen as programs in this language. This programming language is not very human-friendly, but it is maximally expressive:

[5] https://en.wikipedia.org/wiki/Empty_category

any computable function f can be encoded as a phrase-structure grammar such that for any input w, if $f(w)$ is defined, then w has a unique derivation from which $f(w)$ can be read off. There is in fact a procedure that can convert a Turing machine into a grammar that encodes the same algorithm.

- An interesting feature of this programming language (the language of phrase structure grammars) — a feature that is reflected in the hierarchy — is that some restrictions on the kind of rules allowed lead to the definition of sublanguages that are less expressive but can be executed on more simple machines (roughly, machines with more simple memory mechanisms).
- TODO continue

Relevant reading

- 'Formal languages and syntactic complexity' (slides) by Bernard and Amsili (2023a).

Vocabulary

- a phrase structure grammar: *une grammaire syntagmatique*
- a one-step derivation: *une dérivation immédiate*
- a word: *un mot*
- a sentential form: *un proto-mot*
- an unrestricted grammar: *une grammaire générale*
- a context-sensitive grammar: *une grammaire contextuelle/une grammaire algébrique*
- a context-free grammar: *une grammaire hors-contexte*
- a regular grammar: *une grammaire régulière*
- the pumping lemma: *le lemme de pompage/le lemme de l'étoile*

Chapter 4

Context-free grammars

4.1 Derivations and derivation trees

Reminder 4.1 (Context-free grammar (CFG)) *A* type 2 grammar *is a phrase structure grammar $G = (N, \Sigma, P, S)$ such that each rule of P is of the following form:*

- *$A \to \beta$, where $A \in N$ and $\beta \in (N \cup \Sigma)^\star$.*

Example(s) 4.1 *Consider $G = (N, \Sigma, P, S)$ where:*

- *$N = \{Int\}$;*
- *$\Sigma = \{0, 1, \cdots, 9, +, -\}$;*
- *$P = \{Int \to Int + Int \mid Int - Int \mid 0 \mid 1 \mid \cdots \mid 9\}$;*
- *$S = Int$.*

G is a CFG.

Definition 4.1 (Recursive rule) *In a CFG $G = (N, \Sigma, P, S)$, a production rule $A \to \alpha \in P$ is* recursive *iff $\exists i \in [\![1, |\alpha|]\!]$ such that $\alpha_i = A$ (in other words, the symbol in the left-hand side of the rule also appears in the right-hand side of the rule).*

Example(s) 4.2 *The rule $Int \to Int - Int$ from example 4.1 is recursive.*

Definition 4.2 (Left derivation) *A* left derivation *is a derivation in which each derivation step rewrites the leftmost non-terminal of the sentential form it applies to.*

If β can be derived from α by a left derivation of G, one can write '$\alpha \underset{G}{\overset{\star L}{\Rightarrow}} \beta$'.

Example(s) 4.3 *Using G from example 4.1, the following is the only left derivation of $2 - 4$:*

$$Int \underset{G}{\Rightarrow} Int - Int \underset{G}{\Rightarrow} 2 - Int \underset{G}{\Rightarrow} 2 - 4$$

The following derivation of $2 - 4$, is not a left derivation:

$$Int \underset{G}{\Rightarrow} Int - Int \underset{G}{\Rightarrow} Int - 4 \underset{G}{\Rightarrow} 2 - 4$$

Theorem 4.1 (Sufficiency of left derivations) *Given a CFG $G = (N, \Sigma, P, S)$ and $u, v \in (N \cup \Sigma)^\star$, if $u \underset{G}{\overset{\star}{\Rightarrow}} v$ then $u \underset{G}{\overset{\star L}{\Rightarrow}} v$.*

In other words, if a sequence is derivable from another by a CFG, it can be derived with a left derivation.

Definition 4.3 (Right derivation) *A* right derivation *is a derivation in which each derivation step rewrites the rightmost non-terminal of the sentential form it applies to.*

If β can be derived from α by a right derivation of G, one can write '$\alpha \overset{\star R}{\underset{G}{\Rightarrow}} \beta$'.

Example(s) 4.4 *Using G from example 4.1, the following is the only right derivation of $2-4$:*

$$Int \underset{G}{\Rightarrow} Int - Int \underset{G}{\Rightarrow} Int - 4 \underset{G}{\Rightarrow} 2 - 4$$

The following derivation of $2-4$, is not a right derivation:

$$Int \underset{G}{\Rightarrow} Int - Int \underset{G}{\Rightarrow} 2 - Int \underset{G}{\Rightarrow} 2 - 4$$

Theorem 4.2 (Sufficiency of right derivations) *Given a CFG $G = (N, \Sigma, P, S)$ and $u, v \in (N \cup \Sigma)^\star$, if $u \overset{\star}{\underset{G}{\Rightarrow}} v$ then $u \overset{\star R}{\underset{G}{\Rightarrow}} v$.*

In other words, if a sequence is derivable from another by a CFG, it can be derived with a right derivation.

Definition 4.4 (Derivation tree) *Given a CFG $G = (N, \Sigma, P, S)$, a* derivation tree *of G is an ordered tree*[1] *such that:*

- *the root of the tree is labelled with S;*
- *each other interval node is labelled with a symbol from N;*
- *each other node (i.e. a leaf) is labelled with a symbol from $N \cup \Sigma$ (i.e. the vocabulary);*
- *each internal node labelled A with exactly n (ordered) children labelled $\alpha_1\,\alpha_2 \cdots \alpha_n$ is such that $A \to \alpha_1\,\alpha_2 \cdots \alpha_n \in P$.*

The set of all derivation trees of G is written '$\mathcal{T}(G)$'.

Example(s) 4.5 *Figure 4.1 shows a derivation of the grammar from example 4.1.*

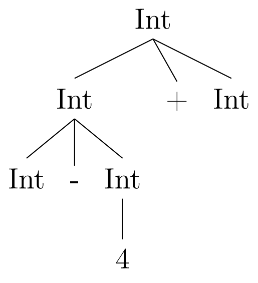


Figure 4.1: A derivation tree.

Definition 4.5 (Yield of a derivation tree) *The* yield *of a derivation tree T is the sequence of symbol consisting of the labels of the (ordered) leaves of T.*

Theorem 4.3 *The yield of a derivation tree of a CFG G is a sentential form of G.*

Definition 4.6 (Tree of a derivation) *Given a CFG $G = (N, \Sigma, P, S)$, the* trees *of all derivations from S by G are derivation trees of G defined inductively on the length n of the derivations:*

[1]See https://en.wikipedia.org/wiki/Tree_(graph_theory).

- *for $n = 0$, the only derivation from S of length n is the empty derivation from S to S and its tree, $T(S)$, is defined as the tree that consists of a single node labelled S;*
- *for $n \geq 1$, the tree of a derivation of length n $S = \gamma_0 \underset{G}{\Rightarrow} \gamma_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} \gamma_n$ such that γ_n is the result of rewriting $\gamma_{n-1,i}$ by $\gamma_{n,i}\,\gamma_{n,i+1}\,\cdots\,\gamma_{n,i+k-1}$ in γ_{n-1}, $T(\gamma_0 \underset{G}{\Rightarrow} \gamma_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} \gamma_n)$, is defined as the tree obtained from $T(\gamma_0 \underset{G}{\Rightarrow} \gamma_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} \gamma_{n-1})$ by adding k (ordered) children labelled $\gamma_{n,i}\,\gamma_{n,i+1}\,\cdots\,\gamma_{n,i+k-1}$ to its i^{th} leaf.*

Example(s) 4.6 *The trees of derivations*

- *Int (i.e. the empty derivation from Int),*
- $Int \underset{G}{\Rightarrow} Int + Int$,
- $Int \underset{G}{\Rightarrow} Int + Int \underset{G}{\Rightarrow} Int + 7$,
- $Int \underset{G}{\Rightarrow} Int + Int \underset{G}{\Rightarrow} Int + 7 \underset{G}{\Rightarrow} Int - Int + 7$,
- *and* $Int \underset{G}{\Rightarrow} Int + Int \underset{G}{\Rightarrow} Int + 7 \underset{G}{\Rightarrow} Int - Int + 7 \underset{G}{\Rightarrow} Int - 4 + Int$

are shown in this order in figure 4.2.

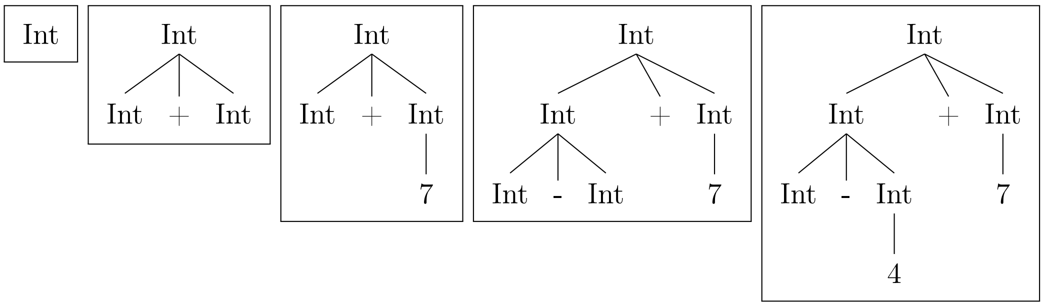


Figure 4.2: The sequence of trees of the derivations from example 4.6.

Property 4.1 *The yield of the tree of a derivation $S \overset{\star}{\Rightarrow} \alpha$ is α.*

Remark 4.1 *Multiple derivations can have the same tree.*

Example(s) 4.7 *Both derivations*

- $Int \underset{G}{\Rightarrow} Int + Int \underset{G}{\Rightarrow} Int - Int + Int \underset{G}{\Rightarrow} Int - Int + 4$
- *and* $Int \underset{G}{\Rightarrow} Int + Int \underset{G}{\Rightarrow} Int + 4 \underset{G}{\Rightarrow} Int - Int + 4$

have the same tree, shown in figure 4.3.

Remark 4.2 *For a given derivation tree T, there is a unique left (resp. right) derivation that is minimal and has T for tree.*

Example(s) 4.8 *TODO*

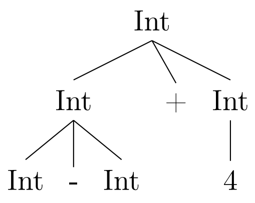


Figure 4.3: The tree of both derivations from example 4.7.

Definition 4.7 (Syntactic structure) *Given a CFG G, the* syntactic structures *of a word $w \in \mathcal{L}(G)$ are the trees of all possible derivations of w by G.*

Definition 4.8 (Ambiguous grammar) *A CFG G is* ambiguous *iff there is a word $w \in \mathcal{L}(G)$ that has at least two distinct syntactic structures.*

Example(s) 4.9 *G, the grammar from example 4.1, is ambiguous: The word $2-4+9$ admits two distinct syntactic structures, shown in figure 4.4. The tree on the left and the tree on the right are respectively the trees of the following left derivations:*

- $Int \underset{G}{\Rightarrow} Int + Int \underset{G}{\Rightarrow} Int - Int + Int \underset{G}{\Rightarrow} 2 - Int + Int \underset{G}{\Rightarrow} 2 - 4 + Int \underset{G}{\Rightarrow} 2 - 4 + 9$;
- $Int \underset{G}{\Rightarrow} Int - Int \underset{G}{\Rightarrow} 2 - Int \underset{G}{\Rightarrow} 2 - Int + Int \underset{G}{\Rightarrow} 2 - 4 + Int \underset{G}{\Rightarrow} 2 - 4 + 9$.

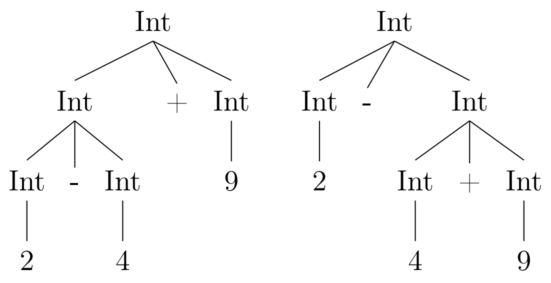


Figure 4.4: Two syntactic structures for $2 - 4 + 9$.

Remark 4.3 *A CFG G is ambiguous iff there is a word $w \in \mathcal{L}(G)$ that can be derived by at least two distinct minimal left (resp. right) derivations.*

Remark 4.4 (Syntactic structure and meaning) *For languages, formal or natural, that are interpreted (i.e. languages that come with a notion of meaning), it is often assumed that meaning is the result of a function of syntactic structures (the* interpretation function*). Thus, a sentence that admits two syntactic structures is associated with two meanings.*

Remark 4.5 (Spurious ambiguity) *A situation in which a word admits two distinct syntactic structures both associated with the same meaning is a case of* spurious ambiguity*. Spurious ambiguities are usually seen as unwanted redundancies. Linguists, in particular, favour grammars that do not lead to any spurious ambiguity, except if well justified.*

Remark 4.6 *To avoid ambiguities in formal languages it is common to either use parentheses or, equivalently, to define conventions about associativity and priority. The effect of such conventions is to rule out some derivations a posteriori.*

For example, without any priority convention, the arithmetics expression $3 \times 5 - 2$ could be either evaluated to 13 or to 9. This ambiguity is usually avoided either by adding parentheses (e.g. $(3 \times 5) - 2$ and $3 \times (5 - 2)$) and/or by specifying a priority ordering of the operators (e.g. specifying that $\times$ has priority over $-$, leading to the value 13).

Definition 4.9 (Ambiguous language) *A context-free language L is* ambiguous *iff all CFG G such that $L = \mathcal{L}(G)$ are ambiguous.*

Example(s) 4.10 *It can be proved that $L = \{a^n\, b^n\, c^m \mid n, m \in \mathbb{N}\} \cup \{a^n\, b^m\, c^m \mid n, m \in \mathbb{N}\}$ is an ambiguous context-free language. The idea is that for any CFG G such that $L = \mathcal{L}(G)$, for any $n \in \mathbb{N}^*$, $a^n\, b^n\, c^n$ admits at least two syntactic structures.*

Remark 4.7 (Ambiguity of natural language) *Natural language is massively ambiguous. One of the main sources of this ambiguity is the attachment of prepositional phrases. For instance, the first sentence of (1) — famously uttered by a character played by Groucho Marx in Victor Heerman's film* Animal Crackers *(1930) — is ambiguous because the prepositional phrase* in my pajamas *may be attached to the noun phrase* an elephant *(see figure 4.5) or to the verb phrase* shot an elephant *(see figure 4.6). If this sentence is considered in isolation, the most probable option is the attachment of the prepositional phrase to the verb phrase. In the context of the following sentence, however, the most probable option is the attachment of the prepositional phrase to the noun phrase.*

(1) One morning, I shot an elephant in my pajamas. How he got in my pajamas, I don't know.

The two different (semantic) interpretations of the first sentence of (1) form a strong argument for the existence of the (syntactic) ambiguity discussed here.

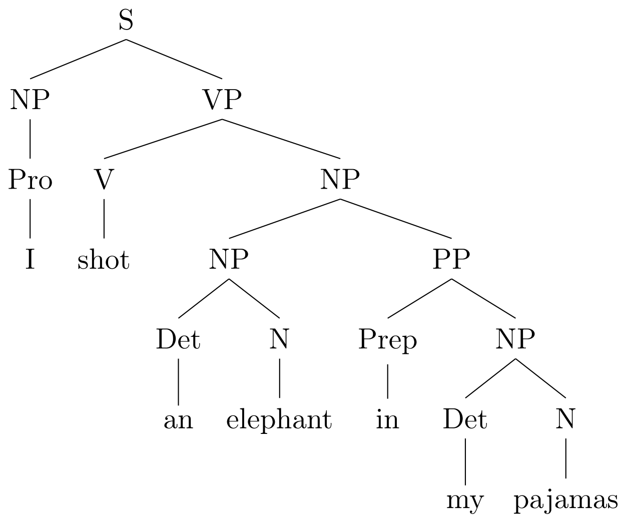


Figure 4.5: Syntactic structure of *I shot an elephant in my pajamas* with NP attachment of the prepositional phrase.

Remark 4.8 *The same language may be generated by both an ambiguous grammar and a non-ambiguous one. Such a language is then not ambiguous (because it is generated by at least one non-ambiguous grammar).*

Example(s) 4.11 *Let* $G_1 = (\{Int, Dig\}, [\![0, 9]\!], P_1, Int)$ *where* $P_1 = \left\{ \begin{array}{l} Int \to Dig \mid Int\, Int, \\ Dig \to 0 \mid 1 \mid \cdots \mid 9 \end{array} \right\}$ *and* $G_2 = (\{Int, Dig\}, [\![0, 9]\!], P_2, Int)$ *where* $P_2 = \left\{ \begin{array}{l} Int \to Dig \mid Dig\, Int, \\ Dig \to 0 \mid 1 \mid \cdots \mid 9 \end{array} \right\}$. *Both grammars generate the same language* $L =$

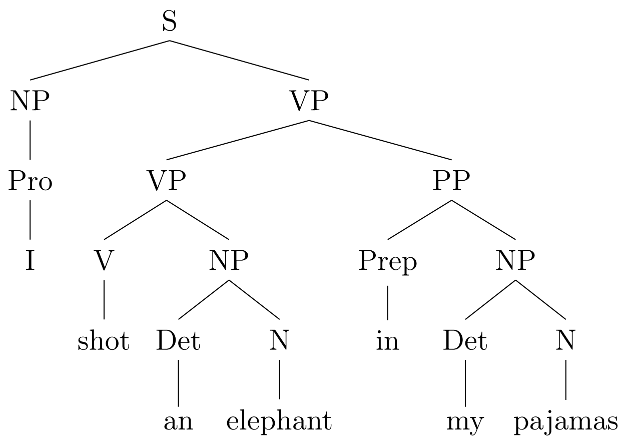


Figure 4.6: Syntactic structure of *I shot an elephant in my pajamas* with VP attachment of the prepositional phrase.

$\mathcal{L}((0 \mid 1 \mid \cdots \mid 9)^\star)$, *but* G_1 *is ambiguous — any three-digit long word in* L, *for instance, admits two distinct syntactic structure in* G_1 *— while* G_2 *is not.* L *is not an ambiguous language.*

Definition 4.10 (Weak equivalence of grammars) *Two CFG* G_1 *and* G_2 *are* weakly equivalent *iff* $\mathcal{L}(G_1) = \mathcal{L}(G_2)$.

Definition 4.11 (Strong equivalence of grammars) *Two CFG* G_1 *and* G_2 *are* strongly equivalent *iff* $\mathcal{T}(G_1) = \mathcal{T}(G_2)$.

Remark 4.9 *If two CFG* G_1 *and* G_2 *are strongly equivalent, they are also weakly equivalent.*

4.2 Properties of CFG

Theorem 4.4 (Pumping lemma – context-free version) *If* L *is a context-free language on alphabet* Σ, *then there exists* $k \in \mathbb{N}$ *such that:*

$$\forall s \in L \text{ s.t. } |s| > k,\ \exists u, v, w, x, y \in \Sigma^\star,\ s = u\,v\,w\,x\,y,\ |v\,w\,x| \leq k,\ |v| + |x| > 0 \text{ and}$$
$$\{u\,v^n\,w\,x^n\,y \mid n \in \mathbb{N}\} \subseteq L$$

Intuition 4.1 (Pumping lemma) *Consider an infinite context-free language* L *and a CFG* $G = (N, \Sigma, P, S)$ *such that* $L = \mathcal{L}(G)$. *For any word* $s \in \mathcal{L}(G)$ *above a certain length* k, s *must have been derived via a generative loop on some non-terminal* A:

- $S \overset{\star}{\underset{G}{\Rightarrow}} u\,A\,y$ *(A is accessible from the axiom);*
- $A \overset{\star}{\underset{G}{\Rightarrow}} v\,A\,x$ *with* $|v| + |x| > 0$ *(the loop on A is generative);*
- $A \overset{\star}{\underset{G}{\Rightarrow}} w$ *(the loop can end);*

with $s = u\,v\,w\,x\,y$.

Proof 4.1 *TODO (let* q *be the maximum of the length of the right-hand side of the production rules in the grammar; a syntactic structure of a word of length* $> k = q^{|N|+1}$ *has a height* $> |N| + 1$ *which means that its longest branch must contain (at least) two occurrences of the same non-terminal, at depth* $i, j \in [\![0, |N|]\!]$ *with* $i \neq j$; *the portion of the tree delimited by these two nodes can be 'repeated' arbitrarily many times)*

Theorem 4.5 (Closure properties) *The set of context-free languages on an alphabet Σ is closed under union, concatenation and Kleene closure. In other words, if L_1 and L_2 are two context-free languages, then $L_1 \cup L_2$, $L_1 L_2$ and $L_1{}^\star$ are context-free languages.*

Proof 4.2 *Let L_1 and L_2 be two context-free languages on an alphabet Σ. There exist two CFG $G_1 = (N_1, \Sigma, P_1, S_1)$ and $G_2 = (N_2, \Sigma, P_2, S_2)$ that generate L_1 and L_2 respectively. Without loss of generality, we can assume that $N_1 \cap N_2 = \emptyset$. Consider $S \notin N_1 \cup N_2 \cup \Sigma$.*

- *Consider $G = (N_1 \cup N_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \to S_1 \mid S_2\}, S)$. G is a context-free grammar and $\mathcal{L}(G) = L_1 \cup L_2$.*
- *Consider $G = (N_1 \cup N_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S)$. G is a context-free grammar and $\mathcal{L}(G) = L_1 L_2$.*
- *Consider $G = (N_1 \cup \{S\}, \Sigma, P_1 \cup \{S \to S S_1 \mid \epsilon\}, S)$. G is a context-free grammar and $\mathcal{L}(G) = L_1{}^\star$.*

Theorem 4.6 (Complement and intersection) *Unlike the set of regular languages on an alphabet Σ, the set of context-free languages on Σ is* neither *closed under complementation nor intersection.*

Proof 4.3 *Consider $L_1 = \{a^n\, b^n\, c^m \mid n, m \in \mathbb{N}\}$ and $L_2 = \{a^n\, b^m\, c^m \mid n, m \in \mathbb{N}\}$. L_1 and L_2 are both context-free languages.*

- *It can be proved using the pumping lemma that $L_1 \cap L_2 = \{a^n\, b^n\, c^n \mid n \in \mathbb{N}\}$ is not context-free. This shows that there are two context-free languages the intersection of which is not context-free.*
- *It also happens that $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. This means, among other things, that the complement of $\overline{L_1} \cup \overline{L_2}$ is not context-free (see the previous point). Now, either $\overline{L_1} \cup \overline{L_2}$ itself is context-free or it is not.*
  - *If $\overline{L_1} \cup \overline{L_2}$ is context-free, then its complement is not.*
  - *Otherwise, because the union of two context-free languages is context-free, either $\overline{L_1}$ or $\overline{L_2}$ is not context-free.*

  In all cases, there is a context-free language the complement of which is not context-free.

Theorem 4.7 (Intersection with a regular language) *If L_1 is a context-free language on an alphabet Σ and L_2 is a regular language on Σ, then $L_1 \cap L_2$ is a context-free language on Σ.*

Remark 4.10 *The most common proof of Theorem 4.7 involves a notion of the product automaton of a push-down automaton and a DFA. It is possible, however, to prove Theorem 4.7 without involving push-down automata.*

Let L_1 be a context-free language on Σ generated by a CFG (N, Σ, P, S) and L_2 be a regular language on Σ recognised by a DFA $(\Sigma, Q, q_i, F, \delta)$. One can prove that $L_1 \cap L_2$ is a context-free language by defining a CFG and proving that this CFG generates $L_1 \cap L_2$. Let $S' \notin N$ and $N' = (Q \times N \times Q) \cup \{S'\}$. Consider the CFG $G' = (N', \Sigma, P', S')$ where $P' = P'_1 \cup P'_2 \cup P'_3$ with

- *$P'_1 = \{S' \to (q_i, S, q_f) \mid q_f \in F\}$,*
- *$P'_2 = \{(q, A, q) \to \epsilon \mid q \in Q$ and $A \to \epsilon \in P\}$,*

- *with g the function from $Q \times (N \cup \Sigma) \times Q$ to $N' \cup \Sigma$ such that* $g((q, x, q')) = \begin{cases} (q, x, q') \text{ if } x \in N \\ x \text{ if } x \in \Sigma \end{cases}$,

$$P'_3 = \left\{ \begin{array}{c} (q_1, A, q_{n+1}) \to g((q_1, \alpha_1, q_2))\, g((q_2, \alpha_2, q_3)) \cdots g((q_n, \alpha_n, q_{n+1})) \\ \mid n > 0,\, q_1, q_2, \cdots, q_{n+1} \in Q,\ A \to \alpha_1\alpha_2 \cdots \alpha_n \in P \\ \text{and } \forall i \in [\![1, n]\!] \text{ s.t. } \alpha_i \in \Sigma,\, \delta((q_i, \alpha_i)) = q_{i+1} \end{array} \right\}.$$

All there is left to prove Theorem 4.7 is to prove that $\mathcal{L}(G') = L_1 \cap L_2$.

4.3 Grammar simplification

In this section, we study algorithms that aim at *simplifying* CFGs. These algorithms do not affect the language generated by the grammars they simplify (i.e. they output a weakly equivalent grammar of their input grammar) but produce grammars that are more practical both to use for syntactic parsing and to derive interesting theoretical results.

4.3.1 Useless items

Definition 4.12 (Language generated by a non-terminal) *Let $G = (N, \Sigma, P, S)$ be a CFG and $A \in N$. The* language generated by A in G, *written '$\mathcal{L}_A(G)$' is the set of all words on Σ that can be derived from A: $\mathcal{L}_A(G) = \{w \in \Sigma^\star \mid A \overset{\star}{\underset{G}{\Rightarrow}} w\}$.*

Remark 4.11 *Let $G = (N, \Sigma, P, S)$ be a CFG and $A \in N$. Then, $\mathcal{L}_A(G) = \mathcal{L}((N, \Sigma, P, A))$.*

Remark 4.12 *Let $G = (N, \Sigma, P, S)$ be a CFG. Then, $\mathcal{L}_S(G) = \mathcal{L}(G)$.*

Definition 4.13 (Productivity of a non-terminal) *Let $G = (N, \Sigma, P, S)$ be a CFG and $A \in N$. A is* productive *iff $\mathcal{L}_A(G) \neq \emptyset$. Otherwise, A is* unproductive.

Remark 4.13 *A non-terminal A is unproductive, for instance, if there is no rule that rewrites it. More generally A is unproductive if all sequences derivable from it always contain a non-terminal.*

Definition 4.14 (Accessibility of a non-terminal) *Let $G = (N, \Sigma, P, S)$ be a CFG and $A \in N$. A is* accessible *iff A appears in a sequence derivable from S: for some $\alpha, \beta \in (N \cup \Sigma)^\star$, $S \overset{\star}{\underset{G}{\Rightarrow}} \alpha\, A\, \beta$. Otherwise, A is* inaccessible.

Definition 4.15 (Usefulness of a production rule) *Let $G = (N, \Sigma, P, S)$ be a CFG and a production rule $A \to \alpha \in P$. $A \to \alpha$ is* useful *iff there is a word $w \in \mathcal{L}(G)$ that admits a derivation involving $A \to \alpha$: $S \overset{\star}{\underset{G}{\Rightarrow}} x\, A\, y \underset{G}{\Rightarrow} x\, \alpha\, y \overset{\star}{\underset{G}{\Rightarrow}} w$. Otherwise, $A \to \alpha$ is* useless.

Definition 4.16 (Usefulness of a non-terminal) *Let $G = (N, \Sigma, P, S)$ be a CFG and $A \in N$. A is* useful *iff there is a useful production rule $A \to \alpha \in P$. Otherwise, A is* useless.

Remark 4.14 *If a non-terminal is useful, then it is both productive and accessible. The converse, however, is not true; a terminal may be both productive and accessible without being useful.*

Consider, for instance, A in $G = (\{S, A\}, \{a\}, \{S \to S\, A, A \to a\}, S)$; A is productive ($A \overset{\star}{\underset{G}{\Rightarrow}} a$) and accessible ($S \overset{\star}{\underset{G}{\Rightarrow}} S\, A$), and yet useless (remark that $\mathcal{L}(G) = \emptyset$).

What is true, however, is that in a grammar without any unproductive non-terminals, a non-terminal is useful iff it is accessible.

- There is an algorithm for removing from a CFG all useless production rules and non-terminals without impacting the language generated. This algorithm is structured in two steps:

  1. detection and deletion of all unproductive non-terminals and of all production rules in which they appear;
  2. detection and deletion of all inaccessible non-terminals and of all production rules in which they appear.

- Concerning the first step, in practice, for this course, you are simply asked to enumerate the productive non-terminals and to give for each of them an example of a sequence of letters that it generates. You do not need to provide any formal justification about non-terminals that are unproductive.

- The second step is performed by algorithm 2.

Example(s) 4.12 (Removing unproductive non-terminals) *Let* $G = (\{S, A, B, C\}, \{a\,b\,c\}, P, S)$ *where* $P = \left\{\begin{array}{l} S \to \epsilon \mid A\,B \mid A\,A, \\ A \to a \mid A\,B, \\ B \to b\,B, \\ C \to c\,A \end{array}\right\}$. *S, A and C are productive. For instance,* $S \overset{\star}{\Rightarrow} \epsilon$, $A \overset{\star}{\Rightarrow} a$ *and* $C \overset{\star}{\Rightarrow} c\,a$. *B, however, is unproductive.*

The result of the deletion of all unproductive non-terminals and of all production rules in which they appear is the grammar $G' = (\{S, A, C\}, \{a\,b\,c\}, P', S)$ *where* $P' = \left\{\begin{array}{l} S \to \epsilon \mid A\,A, \\ A \to a, \\ C \to c\,A \end{array}\right\}$.

```
// Input:  a CFG $G = (N, \Sigma, P, S)$
// Output:  the list all inaccessible non-terminals of $G$
$N_{\text{acc}} := [S]$;      // List (under construction) of accessible non-terminals.
$i := 0$;
while $i < len(N_{acc})$ do
    $A := N_{\text{acc}}[i]$;
    foreach $(A \to \alpha) \in P$ do
        for $j := 1$ to $|\alpha|$ do
            if $\alpha_i \in N$ and $\alpha_i$ not in $N_{acc}$ then
                $N_{\text{acc}}.\text{append}(\alpha_i)$;
    $i$ += 1;
return [$A$ for $A$ in $N$ if $A$ not in $N_{acc}$];
```
Algorithm 2: Detection of inaccessible non-terminals.

Example(s) 4.13 (Removing inaccessible non-terminals) *Let* $G' = (\{S, A, C\}, \{a\,b\,c\}, P', S)$ *where* $P' = \left\{\begin{array}{l} S \to \epsilon \mid A\,A, \\ A \to a, \\ C \to c\,A \end{array}\right\}$.

When executing algorithm 2, N_{acc} *is initialised as* $[S]$.

- *During the first iteration of the outer loop ($i = 0$), the rules of right-hand side S are considered: $S \to \epsilon$ does not affect N_{acc} but $S \to A\,A$ leads to appending A to it.*
- *During the second iteration ($i = 1$), the rules of right-hand side A are considered: $A \to a$ does not affect N_{acc}.*

The outer loop then stops and $[C]$ is returned: C is indeed the only inaccessible non-terminal of G'.

The result of the deletion of all inaccessible non-terminals and of all production rules in which they appear is the grammar $G" = (\{S, A\}, \{a\,b\,c\}, P", S)$ where $P' = \left\{ \begin{array}{l} S \to \epsilon \mid A\,A, \\ A \to a \end{array} \right\}$.

Remark 4.15 *Because the deletion of all unproductive non-terminals and of all production rules in which they appear (i.e. the first step) may introduce inaccessible non-terminals, the two steps must be performed in the specified order.*

Example(s) 4.14 *Let $G = (\{S, A, B\}, \{a, b\}, \{S \to a \mid A\,B, A \to b\}, S)$.*

- *By first applying the first step, B and all rules involving it are removed because B is unproductive in G. Then, by applying the second step, A and all rules involving it are removed because A is inaccessible in the grammar produced by the first step. The result is $(\{S\}, \{a, b\}, \{S \to a\}, S)$.*
- *By first applying the second step, nothing is removed, as all non-terminals are accessible in G. By then applying the first step, B and all rules involving it are removed because B is unproductive in G. The result is $(\{S, A\}, \{a, b\}, \{S \to a, A \to b\}, S)$, which contains an inaccessible non-terminal (A).*

4.3.2 Non-generative production rules

Definition 4.17 (Non-generative production rule) *Let $G = (N, \Sigma, P, S)$ be a CFG. $A \to \alpha \in P$ is* non-generative *iff its right-hand side is a single non-terminal: $\alpha = B \in N$.*

Definition 4.18 (Singular accessibility) *Let $G = (N, \Sigma, P, S)$ be a CFG and $A, B \in N$. B is* singularly accessible *from A iff there is a sequence of $n \geq 1$ non-generative production rules $(X_i \to X_{i+1})_{1 \leq i \leq n}$ of G such that $X_1 = A$, and $X_{n+1} = B$:*

$$A = X_1 \underset{G}{\Rightarrow} X_2 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} X_{n+1} = B$$

Remark 4.16 *Given the above definitions, B being singularly accessible from A is stronger than simply $A \overset{\star}{\Rightarrow} B$. However, in a grammar with no other ϵ-production than, possibly, the one from the axiom if the axiom is not found in the right-hand side of any rule, then the two notions are equivalent.*

Definition 4.19 (Cycle of non-generative production rules) *Let G be a grammar. A* cycle of non-generative production rules *is a sequence of $n \geq 1$ non-generative production rules $(X_i \to X_{i+1})_{1 \leq i \leq n}$ of G such that $X_{n+1} = X_1$:*

$$X_1 \underset{G}{\Rightarrow} X_2 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} X_{n+1} = X_1$$

Remark 4.17 *A grammar admits a cycle of non-generative production rules iff it contains a non-terminal symbol that is singularly accessible from itself.*

Remark 4.18 *The possibility of cycles of non-generative production rules in a grammar is problematic for some generation and parsing algorithms as it may make them loop infinitely.*

Theorem 4.8 (Removal of non-generative production rules) *Let $G = (N, \Sigma, P, S)$ be a CFG. There is a CFG G' such that $\mathcal{L}(G') = \mathcal{L}(G)$ and G' does not contain any non-generative production rule.*

- Theorem 4.8 can be proved by providing an algorithm that builds G'. There is indeed such an algorithm, organised in two steps:
  1. computation of $(C_X)_{X \in N}$ where $C_X = \{Y \in N \mid Y = X$ or Y is singularly accessible from $X\}$;
  2. construction of a new set of production rules in which non-generative ones are 'by-passed'.
- The first step is performed by algorithm 3 or, alternatively, by algorithm 4. Algorithm 3 might be simpler to understand but is less efficient than algorithm 4.
- The second step is performed by algorithm 5.

```
// Input:  a CFG $G = (N, \Sigma, P, S)$
// Output:  the dictionary $C$ of all $C_X$ for $X \in N$
// Initialisation
$C := \{X : \{X\}$ for $X$ in $N\}$;
foreach $(X \to Y) \in P$ where $X, Y \in N$ do
    $C[X]$.add($Y$);
// Main loop
while True do  // can be exited with a 'return' or a 'break' instruction
    change := False;
    foreach $X \in N$ do
        tmp := set();
        foreach $Y \in C[X]$ do
            tmp.update($C[Y] \setminus C[X]$);
        if $len(tmp) > 0$ then
            change := True;
            $C[X]$.update(tmp);
    if change := False then return $C$;
```

Algorithm 3: Computation of $(C_X)_{X \in N}$, where $C_X = \{Y \in N \mid Y = X$ or Y is singularly accessible from $X\}$.

Remark 4.19 *There is a convenient way to execute algorithms 3 and 4 with pen and paper using a boolean matrix with rows and columns indexed by non-terminals: a value True for coefficient (X, Y) means that $X \overset{\star}{\Rightarrow} Y$, and as a consequence each line of the matrix represents a C_X. Instead of writing True/False in the matrix, it is easier to write nothing for False·s and some arbitrary mark, e.g. '$\times$', for True·s.*

For example, the following matrix (obtained by running any of the two algorithms for $P = \{S \to A \mid \epsilon, A \to a\,A \mid B \mid C, B \to b\,B \mid A, C \to c\}$) encodes $C_S = \{S, A, B, C\}$,

```
// Input:  a CFG $G = (N, \Sigma, P, S)$
// Output:  the dictionary $C$ of all $C_X$ for $X \in N$
$C := \{X : \{X\}$ for $X$ in $N\}$;
foreach $(X \to Y) \in P$ where $X, Y \in N$ do
    foreach $Z \in N$ s.t. ($X$ in $C[Z]$) do
        $C[Z]$.update($C[Y]$);
return $C$;
```

Algorithm 4: Computation of $(C_X)_{X \in N}$, where $C_X = \{Y \in N \mid Y = X$ or Y is singularly accessible from $X\}$. This algorithm is mainly based on the fact that, for three non-terminals X, Y and Z, if there is a rule $X \to Y$ and if $X \in C_Z$, then $C_Y \subseteq C_Z$ (i.e. any non-terminal W that is either equal to Y or singularly accessible from it is also either equal to Z or singularly accessible from it).

$C_A = C_B = \{A, B, C\}$ *and* $C_C = \{C\}$*:*

	S	A	B	C
S	$\times$	$\times$	$\times$	$\times$
A		$\times$	$\times$	$\times$
B		$\times$	$\times$	$\times$
C				$\times$

- *In algorithm 3, one iteration of the main loop consists, roughly, in iterating over the current* $\times$*·s of the matrix and, for a* $\times$ *that corresponds to* $X \overset{\star}{\Rightarrow} Y$*, copying all* $\times$*·s from the line of* Y *to the line of* X*. If no new* $\times$ *is added during an iteration, then the algorithm stops at the end of this iteration.*

- *Algorithm 4 consists in looping once over all non-generative production rules and, for a non-generative rule* $X \to Y$*, copying all* $\times$*·s from the line of* Y *to all lines that contain* X*.*

Example(s) 4.15 (Computing the $(C_X)_{X \in N}$; algorithm 3) *Let* $G = (\{S, A, B, C, D\}, \{a\,b\,c, d\}, P, S)$ *where* $P = \left\{\begin{array}{l} S \to A \mid \epsilon, \\ A \to a\,A \mid B \mid C, \\ B \to b\,A, \\ C \to A \mid D, \\ D \to d \end{array}\right\}$*. At the end of the initialisation of algorithm 3,* $C[S] = \{S, A\}$*,* $C[A] = \{A, B, C\}$*,* $C[B] = \{B\}$*,* $C[C] = \{A, C, D\}$ *and* $C[D] = \{D\}$*:*

	S	A	B	C	D
S	$\times$	$\times$			
A		$\times$	$\times$	$\times$	
B			$\times$		
C		$\times$		$\times$	$\times$
D					$\times$

During the first iteration of the main loop:

- *When considering* S*, the set computed for tmp is* $\{B, C\}$ *and* $C[S]$ *is updated to*

$\{S, A, B, C\}$:

	S	A	B	C	D
S	×	×	×	×	
A		×	×	×	
B			×		
C		×		×	×
D					×

In addition, continue is set to True.

- *When considering A, the set computed for tmp is $\{D\}$ and $C[A]$ is updated to $\{A, B, C, D\}$:*

	S	A	B	C	D
S	×	×	×	×	
A		×	×	×	×
B			×		
C		×		×	×
D					×

- *When considering B, the set computed for tmp is empty and nothing is updated.*
- *When considering C, the set computed for tmp is $\{B\}$ and $C[C]$ is updated to $\{A, B, C, D\}$:*

	S	A	B	C	D
S	×	×	×	×	
A		×	×	×	×
B			×		
C		×	×	×	×
D					×

- *When considering D, the set computed for tmp is empty and nothing is updated.*

During the second iteration:

- *When considering S, the set computed for tmp is $\{D\}$ and $C[S]$ is updated to $\{S, A, B, C, D\}$:*

	S	A	B	C	D
S	×	×	×	×	×
A		×	×	×	×
B			×		
C		×	×	×	×
D					×

In addition, continue is set to True.

- *When considering A, the set computed for tmp is empty and nothing is updated. Idem for B, C and D.*

During the third iteration:

- *When considering S, the set computed for tmp is empty and nothing is updated. Idem for A, B, C and D.*

The variable continue has not been set to True during this third iteration; the main loop thus stops and the $(C_X)_{X \in N}$ are returned: $C_S = \{S, A, B, C, D\}$, $C_A = \{A, B, C, D\}$, $C_B = \{B\}$, $C_C = \{A, B, C, D\}$ and $C_D = \{D\}$.

Example(s) 4.16 (Computing the $(C_X)_{X\in N}$; algorithm 4) *Let* $G = (\{S, A, B, C, D\}, \{a\,b\,c, d\}, P, S)$ *where* $P = \left\{\begin{array}{l} S \to A \mid \epsilon, \\ A \to a\,A \mid B \mid C, \\ B \to b\,A, \\ C \to A \mid D, \\ D \to d \end{array}\right\}$.

When executing algorithm 3, $C[S]$ is initialised as $\{S\}$, $C[A]$ as $\{A\}$, $C[B]$ as $\{B\}$, $C[C]$ as $\{C\}$ and $C[D]$ as $\{D\}$:

	S	A	B	C	D
S	$\times$				
A		$\times$			
B			$\times$		
C				$\times$	
D					$\times$

- *During the first iteration of the outer loop ($X \to Y = S \to A$), S is the only $Z \in N$ such that $S \in C[Z]$, and so the line of S is updated with the content of the line of A:*

	S	A	B	C	D
S	$\times$	$\times$			
A		$\times$			
B			$\times$		
C				$\times$	
D					$\times$

- *During the first iteration of the outer loop ($X \to Y = A \to B$), both S and A are $Z \in N$ such that $A \in C[Z]$, and so the lines of S and A are updated with the content of the line of B:*

	S	A	B	C	D
S	$\times$	$\times$	$\times$		
A		$\times$	$\times$		
B			$\times$		
C				$\times$	
D					$\times$

- *During the second iteration of the outer loop ($X \to Y = A \to C$), both S and A are $Z \in N$ such that $A \in C[Z]$, and so the lines of S and A are updated with the content of the line of C:*

	S	A	B	C	D
S	$\times$	$\times$	$\times$	$\times$	
A		$\times$	$\times$	$\times$	
B			$\times$		
C				$\times$	
D					$\times$

- *During the third iteration of the outer loop ($X \to Y = C \to A$), S, A and B are $Z \in N$ such that $C \in C[Z]$, and so the lines of S, A and C are updated with the content of the* *line of A:*

	S	A	B	C	D
S	$\times$	$\times$	$\times$	$\times$	
A		$\times$	$\times$	$\times$	
B			$\times$		
C		$\times$	$\times$	$\times$	
D					$\times$

- *During the fourth iteration of the outer loop ($X \to Y = C \to D$), S, A and B are $Z \in N$ such that $C \in C[Z]$, and so the lines of S, A and C are updated with the content of the line of D:*

	S	A	B	C	D
S	$\times$	$\times$	$\times$	$\times$	$\times$
A		$\times$	$\times$	$\times$	$\times$
B			$\times$		
C		$\times$	$\times$	$\times$	$\times$
D					$\times$

The outer loop then stops — all singular rules in P having been considered — and the $(C_X)_{X \in N}$ are returned: $C_S = \{S, A, B, C, D\}$, $C_A = \{A, B, C, D\}$, $C_B = \{B\}$, $C_C = \{A, B, C, D\}$ and $C_D = \{D\}$.

```
// Input:  a CFG $G = (N, \Sigma, P, S)$ and $(C_X)_{X \in N}$
// Output:  a new set of production rules
$P' := \emptyset$;
foreach $X \to \alpha \in P$ s.t. $\alpha \notin N$ do
    foreach $Z \in N$ s.t. $(X$ in $C[Z])$ do
        $P'$.add($Z \to \alpha$);
return $P'$;
```

Algorithm 5: Construction of a new set of production rules by 'by-passing' non-generative ones.

Example(s) 4.17 (By-passing non-generative rules; algorithm 5) *Let $G = (\{S, A, B, C, D\}, \{a\, b\, c, d\}, P, S)$ where*

$$P = \left\{ \begin{array}{l} S \to A \mid \epsilon, \\ A \to a\, A \mid B \mid C, \\ B \to b\, A, \\ C \to A \mid D, \\ D \to d \end{array} \right\}.$$

Let $C[S] = \{S, A, B, C, D\}$, $C[A] = \{A, B, C, D\}$, $C[B] = \{B\}$, $C[C] = \{A, B, C, D\}$ and $C[D] = \{D\}$.

When executing algorithm 5, P' is initialised as $\emptyset$.

- *During the first iteration of the outer loop ($X \to \alpha = S \to \epsilon$), S is the only non-terminal Z such that $S \in C[Z]$, and so $S \to \epsilon$ is added to P'.*

- *During the second iteration of the outer loop ($X \to \alpha = A \to a\, A$), S, A and C are the non-terminals Z such that $A \in C[Z]$, and so $S \to a\, A$, $A \to a\, A$ and $C \to a\, A$ are added to P'.*

- *During the third iteration of the outer loop ($X \to \alpha = B \to b\, A$), S, A, B and C are the non-terminals Z such that $B \in C[Z]$, and so $S \to b\, A$, $A \to b\, A$, $B \to b\, A$ and $C \to b\, A$ are added to P'.*

- *During the fourth iteration of the outer loop ($X \to \alpha = D \to d$), S, A, C and D are the non-terminals Z such that $D \in C[Z]$, and so $S \to d$, $A \to d$, $C \to d$ and $D \to d$ are added to P'.*

The outer loop then stops — all non-singular rules in P having been considered — and P' returned: $P' = \left\{ \begin{array}{l} S \to \epsilon \mid a\,A \mid b\,A \mid d, \\ A \to a\,A \mid b\,A \mid d, \\ B \to b\,A, \\ C \to a\,A \mid b\,A \mid d, \\ D \to d \end{array} \right\}$.

4.3.3 ϵ-productions

Definition 4.20 (ϵ-production) *An* ϵ-production *is a rule the right-hand side of which is ϵ. In other words, it is a rule of the form $A \to \epsilon$, where A is a non-terminal.*

Remark 4.20 *ϵ-productions can be problematic for some generation and parsing algorithms, either because these algorithms are simply not designed to handle ϵ-productions or because the cycles that ϵ-productions can introduce in derivations make them loop infinitely.*

Example(s) 4.18 *Let $G = (\{S, A, B\}, \{a\,b\}, P, S)$ where* $P = \left\{ \begin{array}{l} S \to A\,B, \\ A \to \epsilon \mid a, \\ B \to b \mid A\,B \end{array} \right\}$.

Here is a cyclic derivation of G that involves an ϵ-production: $A\,B \underset{G}{\Rightarrow} B \underset{G}{\Rightarrow} A\,B$.

Theorem 4.9 (Removal of ϵ-productions) *Let $G = (N, \Sigma, P, S)$ be a CFG. There is a CFG G' such that $\mathcal{L}(G') = \mathcal{L}(G)$ and G' does not contain any ϵ-production except possibly for a rule $S' \to \epsilon$ where S' is the axiom of G' and S' does not appear in the right-hand side of any rule of G'.*

- Theorem 4.9 can be proved by providing an algorithm that builds G'. There is indeed such an algorithm, organised in three steps:
  1. computation of $N_\epsilon = \{X \in N \mid X \overset{\star}{\underset{G}{\Rightarrow}} \epsilon\}$, the set of non-terminals that generate ϵ;
  2. construction of a new set of production rules in which ϵ-productions are 'by-passed';
  3. in case $S \in N_\epsilon$, addition of a new non-terminal S', set as the axiom of G' instead of S, and of two rules $S' \to \epsilon \mid S$. In that case, doing so allows ϵ to be generated (as in the original grammar) while ensuring that the (new) axiom does not appear in the right-hand side of any rule.
- The first step is performed by algorithm 6.
- The second step is performed by algorithm 7.

Example(s) 4.19 (Computation of N_ϵ; algorithm 6) *Let $G = (\{S, A, B\}, \{a, b, c\}, P, S)$ where* $P = \left\{ \begin{array}{l} S \to A\,S\,B \mid c, \\ A \to a\,A \mid B, \\ B \to b \mid \epsilon \end{array} \right\}$.

When executing algorithm 6, N_ϵ is initialised as $\emptyset$.
During the first iteration of the main loop:

```
// Input: a CFG $G = (N, \Sigma, P, S)$
// Output: the set $N_\epsilon$
$N_\epsilon := \emptyset$;
$continue := True$;
while $continue$ do
    $change := False$;
    foreach $(X \to \alpha) \in P$ s.t. $(X$ not in $N_\epsilon)$ do
        if $\forall i \in [\![1, len(\alpha)]\!],\ \alpha_i \in N_\epsilon$ then // in particular if $\alpha = \epsilon$
            $N_\epsilon$.add$(X)$;
            $change := True$;
    $continue := change$;
return $N_\epsilon$;
```

Algorithm 6: Computation of $N_\epsilon = \{X \in N \mid X \overset{*}{\underset{G}{\Rightarrow}} \epsilon\}$. This algorithm is essentially built around two nested loops. The outer loop simply controls when to stop running the inner one. The inner loop scans the grammar once in order to discover new non-terminals that belong to N_ϵ. If this does not happen, then it can be proved that all members of N_ϵ have been discovered.

- *When considering $S \to A\,S\,B$, nothing happens. Idem for $S \to c$, $A \to a\,A$, $A \to B$ and $B \to b$.*

- *When considering $B \to \epsilon$, N_ϵ is updated to $\{B\}$. In addition, continue is set to True.*

During the second iteration of the main loop:

- *When considering $S \to A\,S\,B$, nothing happens. Idem for $S \to c$ and $A \to a\,A$.*

- *When considering $A \to B$, N_ϵ is updated to $\{A, B\}$. In addition, continue is set to True.*

During the third iteration of the main loop:

- *When considering $S \to A\,S\,B$, nothing happens. Idem for $S \to c$.*

The variable continue has not been set to True during this third iteration; the main loop thus stops and N_ϵ is returned: $N_\epsilon = \{A, B\}$.

```
// Input: a CFG $G = (N, \Sigma, P, S)$ and $N_\epsilon$
// Output: a new set of production rules
$P' := \emptyset$;
foreach $(X \to \alpha) \in P$ do
    foreach $\beta$ obtained from $\alpha$ by deleting zero or more $\alpha_i$ s.t. $\alpha_i \in N_\epsilon$ do
        if $\beta \neq \epsilon$ then
            $P'$.add$(X \to \beta)$;
return $P'$;
```

Algorithm 7: Construction of a new set of production rules by 'by-passing' ϵ-productions.

Example(s) 4.20 (By-passing ϵ-productions; algorithm 7) *Let* $G = (\{S, A, B\}, \{a, b, c\}, P, S)$ *where* $P = \left\{\begin{array}{l} S \rightarrow A\,S\,B \mid c, \\ A \rightarrow a\,A \mid B, \\ B \rightarrow b \mid \epsilon \end{array}\right\}$. *Let* $N_\epsilon = \{A, B\}$.

When executing algorithm 7, P' *is initialised as* $\emptyset$.

- *During the first iteration of the outer loop (*$X \rightarrow \alpha = S \rightarrow A\,S\,B$*),* $S \rightarrow A\,S\,B \mid S\,B \mid A\,S \mid S$ *are added to* P'.
- *During the second iteration of the outer loop (*$X \rightarrow \alpha = S \rightarrow c$*),* $S \rightarrow c$ *is added to* P'.
- *During the third iteration of the outer loop (*$X \rightarrow \alpha = A \rightarrow a\,A$*),* $A \rightarrow a\,A \mid a$ *are added to* P'.
- *During the fourth iteration of the outer loop (*$X \rightarrow \alpha = A \rightarrow B$*),* $A \rightarrow B$ *is added to* P'.
- *During the fifth iteration of the outer loop (*$X \rightarrow \alpha = B \rightarrow b$*),* $B \rightarrow b$ *is added to* P'.
- *During the sixth iteration of the outer loop (*$X \rightarrow \alpha = B \rightarrow \epsilon$*), nothing happens.*

The outer loop then stops — all rules in P *having been considered — and* P' *is returned:*
$P' = \left\{\begin{array}{l} S \rightarrow A\,S\,B \mid S\,B \mid A\,S \mid S \mid c, \\ A \rightarrow a\,A \mid a \mid B, \\ B \rightarrow b \end{array}\right\}$.

4.3.4 Cleaning grammars

Definition 4.21 (Clean grammar) *A CFG* $G = (N, \Sigma, P, S)$ *is* clean *iff*

- *it does not contain any* ϵ*-production except possibly for the rule* $S \rightarrow \epsilon$ *provided that* S *does not appear in the right-hand side of any rule of* G*,*
- *it does not contain any non-generative production rule*
- *and it does not contain any useless item.*

Definition 4.22 (Cycle) *A* cycle *is a derivation of length at least* 1 *that derives a sequence from itself (‘*$\alpha \overset{+}{\Rightarrow} \alpha$*’).*

Property 4.2 *A clean grammar admits no cycle; all of its derivations are minimal (see definition 3.6).*

Remark 4.21 *If* G *is a clean grammar, for all* α *and* β *such that* $\alpha \underset{G}{\overset{\star}{\Rightarrow}} \beta$*, there exist only a finite number of derivations of* β *from* α*. As a consequence, for all* $w \in \mathcal{L}(G)$*,* w *has only a finite number of syntactic structures.*

Theorem 4.10 *It is possible to turn a CFG into a weakly equivalent one that is clean using the three following steps:*

1. *removal of* ϵ*-productions using the above algorithm;*
2. *removal of non-generative production rules using the above algorithm;*
3. *removal of useless items using the above algorithm.*

The order of these three steps is important. This specific order works because the algorithm used for removing non-generative production rules does not introduce any ϵ*-production and the algorithm used for removing useless items does not introduce any* ϵ*-production nor any non-generative production rule.*

4.3.5 Chomsky normal form (CNF)

Definition 4.23 (Chomsky normal form) *A CFG $G = (N, \Sigma, P, S)$ is in* Chomsky normal form *iff each rule of P is of one of the following forms:*

- *$A \to B\,C$, where $A, B, C \in N$;*
- *$A \to a$, where $A \in N$ and $a \in \Sigma$;*
- *$S \to \epsilon$, provided that S does not appear in the right-hand side of any rule of P.*

Remark 4.22

1. *Rules of the form $A \to B\,C$ with $A, B, C \in N$ are called 'binary rules'.*
2. *Rules of the forme $A \to a$ with $A \in N$ and $a \in \Sigma$ are called 'lexical rules'.*

Theorem 4.11 (Conversion to CNF) *Let $G = (N, \Sigma, P, S)$ be a CFG. There is a CFG G' such that $\mathcal{L}(G') = \mathcal{L}(G)$ and G' is in Chomsky normal form.*

- Theorem 4.11 can be proved by providing an algorithm that builds G'. There is indeed such an algorithm, organised in three steps:
  1. if G is not clean, cleaning of G;
  2. lexicalisation of the rules;
  3. binarisation of non-lexical rules.
- The second step is performed by algorithm 8.
- The third step is performed by algorithm 9.

```
// Input:  a clean CFG $G = (N, \Sigma, P, S)$
// Output:  a new grammar
$N' :=$ copy($N$);
$P' := \emptyset$;
foreach $a \in \Sigma$ do
    $X_a :=$ a new symbol not in $N' \cup \Sigma$;
    $N'$.add($X_a$);
    $P'$.add($X_a \to a$);
foreach $(A \to \alpha) \in P$ do
    if $\alpha \in \Sigma$ then // I.e.  if the rule is a lexical one.
        $P'$.add($A \to \alpha$);        // Pre-existing lexical rules are added as is.
    else
        $\beta :=$ the result of substituting in $\alpha$ any $\alpha_i \in \Sigma$ with $X_{\alpha_i}$;
        $P'$.add($A \to \beta$);
return $(N', \Sigma, P', S)$;
```

Algorithm 8: Lexicalisation of a grammar.

Example(s) 4.21 (Lexicalisation and binarisation of a CF rule) *Consider a CFG $G = (N, \Sigma, P, S)$ containing a rule $A \to a\,B\,C\,d\,E$ (with $A, B, C, E \in N$ and $a, d \in \Sigma$). Algorithm 8 replaces this rule with the following ones: $A \to X_a\,B\,C\,X_d\,E$, $X_a \to a$, $X_d \to d$, where X_a and X_d are two new fresh non-terminal symbols. Algorithm 9 then replaces $A \to X_a\,B\,C\,X_d\,E$ with the following ones (all binary): $A \to X_a\,C_1, C_1 \to B\,C_2, C_2 \to C\,C_3, C_3 \to X_d\,E$.*

```
// Input:  a CFG $G = (N, \Sigma, P, S)$
// Output:  a new grammar
$N' := \text{copy}(N)$;
$P' := \emptyset$;
foreach $(A \to \alpha) \in P$ do
    if $len(\alpha) \leq 2$ then
        $P'.\text{add}(A \to \alpha)$;
    else
        $m := \text{len}(\alpha)$;
        $C_1, C_2, \cdots, C_{m-2}$ := new symbols not in $N' \cup \Sigma$;
        $N'.\text{update}(\{C_1, C_2, \cdots, C_{m-2}\})$;
        $P'.\text{add}(A \to \alpha_1\, C_1)$;
        $P'.\text{update}(\{C_1 \to \alpha_2\, C_2, C_2 \to \alpha_3\, C_3, \cdots, C_{m-3} \to \alpha_{m-2}\, C_{m-2}\})$;
        $P'.\text{add}(C_{m-2} \to \alpha_{m-1}\, \alpha_m)$;
return $(N', \Sigma, P', S)$;
```

Algorithm 9: Binarisation of a grammar.

Example(s) 4.22 (Conversion to a CNF) *Consider $G = (N, \Sigma, P, S)$ where:*

- $N = \{S, NP, VP, PN, PAV, VI\}$;
- $\Sigma = \{Simone, Serge, that, believes, sleeps\}$;
- $P = \left\{\begin{array}{l} S \to NP\ VP, \\ NP \to PN, \\ PN \to Simone \mid Serge, \\ VP \to PAV\, that\, S \mid VI, \\ PAV \to believes, \\ VI \to sleeps \end{array}\right\}.$

The clean version of G is $G' = (N', \Sigma, P', S)$ where:

- $N' = \{S, NP, VP, PAV\}$;
- $P' = \left\{\begin{array}{l} S \to NP\ VP, \\ NP \to Simone \mid Serge, \\ VP \to PAV\, that\, S \mid sleeps, \\ PAV \to believes \end{array}\right\}.$

The CNF version of G is $G" = (N", \Sigma, P", S)$ where:

- $N" = \{S, NP, VP, PAV, X_{Simone}, X_{Serge}, X_{that}, X_{sleeps}, X_{believes}, C_1\}$;
- $P" = \left\{\begin{array}{l} X_{Simone} \to Simone, \\ X_{Serge} \to Serge, \\ X_{that} \to that, \\ X_{sleeps} \to sleeps, \\ X_{believes} \to believes, \\ S \to NP\ VP, \\ NP \to Simone \mid Serge, \\ VP \to PAV\, C_1 \mid sleeps, \\ C_1 \to X_{that}\, S, \\ PAV \to believes \end{array}\right\}.$

Remark 4.23 *The output of algorithm 9 often contains useless items. One might want to remove them to get a clean CFG.*

Relevant reading

- 'Three models for the description of language' by Chomsky (1956).

Vocabulary

- the yield of a tree: *la frontière d'un arbre*
- a clean grammar: *une grammaire propre*
- Chomsky normal form: *forme normale de Chomsky*
- a push-down automata: *un automate à pile*

Chapter 5

The complexity of natural language

TODO: see slides

Relevant reading

- ‘Evidence against the context-freeness of natural language’ by Shieber (1985).
- ‘Computational Complexity of Natural Languages: A Reasoned Overview’ by Branco (2018).

Chapter 6

Introduction to the analysis of context-free grammars

6.1 The search graph

- In this chapter, we study two related tasks: *recognition* and *parsing* for CFGs.
- Given a CFG $G = (N, \Sigma, P, S)$ and a string $u \in \Sigma^\star$, the recognition task consists in determining whether $u \in \mathcal{L}(G)$ and the parsing task consists, if indeed $u \in \mathcal{L}(G)$, in constructing at least one syntactic structure of u.
- We study these tasks from an algorithmic point of view: we are interested in algorithms that can solve them automatically.
- The algorithms given in this chapter are all recognition algorithm but they can easily be extended to also perform the parsing task (i.e. to also output a syntactic structure when asked to recognise a word of the grammar).

Definition 6.1 (Search graph) *Given a grammar $G = (N, \Sigma, P, S)$, the* search graph *of G is the directed graph (V, E) where:*

- *$V = (N \cup \Sigma)^\star$;*
- *$E = \{(\alpha, \beta) \in V^2 \mid \alpha \underset{G}{\Rightarrow} \beta\}$.*

Property 6.1 *The search graph (V, E) of a grammar is infinite, in the sense that it contains an infinite number of nodes, but it is also* locally finite, *in the sense that each node has a finite number of neighbours.*

Example(s) 6.1 *Consider $G = (\{S\}, \{a, +\}, \{S \to S + S \mid a\}, S)$. The search graph of G is partially shown in figure 6.1*

Remark 6.1 *Let $G = (N, \Sigma, P, S)$ be a CFG and (V, E) its search graph, the recognition task is equivalent to determining, for any given $w \in (N \cup \Sigma)^\star$, whether there is a path in (V, E) from the node S to the node w. The recognition algorithms studied in this chapter (and in Chapter 7) work by exploring the search graph.*

Remark 6.2 *Given a CFG $G = (N, \Sigma, P, S)$ and $w \in (N \cup \Sigma)^\star$, a directed path from S to w in the search graph directly corresponds to a derivation in G and can thus be converted to a derivation tree.*

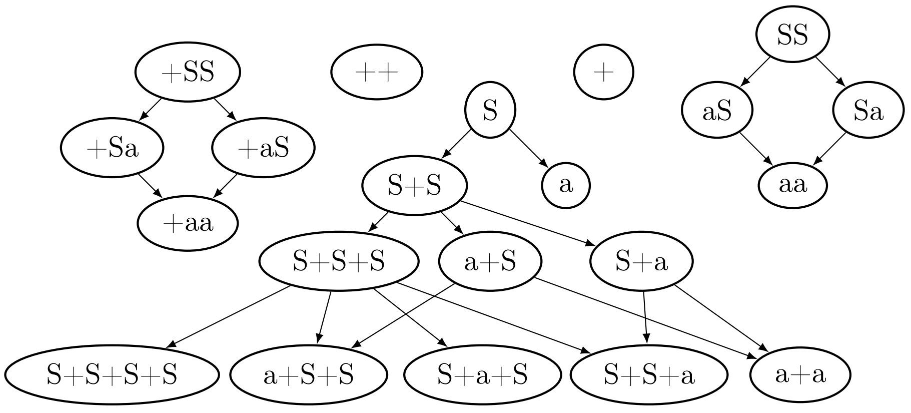


Figure 6.1: Part of the search graph of the grammar given in example 6.1.

Remark 6.3 *There are two main approaches to recognition/parsing: the* bottom-up *approach and the* top-down *approach. These two approaches differ in how they explore the search graph.*

Intuition 6.1

- *Bottom-up algorithms perform the recognition task on w starting the exploration from the w node and 'going up' the search graph from children nodes to parent ones until they find the axiom node or determine that it cannot be reached. They go from a node to one of its parents by 'reversing' a production rule. This corresponds to building (tentative) syntactic trees from the leaves (the terminal symbols in the input) to the root (the axiom).*

- *Top-down algorithms perform the recognition task on w starting the exploration from the axiom node and 'going down' the search graph from parent nodes to children ones until they find the w node or determine that it cannot be reached. They go from a node to one of its children by applying a production rule. This corresponds to building (tentative) syntactic trees from the root (the axiom) to the leaves (the terminal symbols in the input).*

6.2 Bottom-up recognition

- As said above, a bottom-up algorithm starts from w and then explores the search space from children nodes to parent ones.

- Let us consider a node $\alpha = \alpha_1\,\alpha_2\,\cdots\,\alpha_n \in (N \cup \Sigma)^\star$. The parents of α in the search graph are all nodes $\beta \in (N \cup \Sigma)^\star$ such that there exist $i \in [\![1, n+1]\!]$, $j \in [\![i-1, n]\!]$, and $A \in N$ such that $A \to \alpha_i\,\alpha_{i+1}\,\cdots\,\alpha_j \in P$ and $\beta = \alpha_1\,\alpha_2\,\cdots\,\alpha_{i-1}\,A\,\alpha_{j+1}\,\alpha_{j+2}\,\cdots\,\alpha_n$. (Cases where $j = i-1$ correspond to ϵ-productions between α_j and α_i.)

- As a node might have multiple parents, a bottom-up algorithm needs to specify how to explore all the corresponding paths.

- When facing a choice, a *depth-first* algorithm selects one option, records all alternatives, fully explores the selected option and only explores the alternatives if necessary and one by one. Fully exploring an option means exploring it until the S node is reached or it can be proved that the S node cannot be reached this way.

- This idea is implemented in algorithm 10, a *recursive* algorithm.[1] Because treating ϵ-productions as any other rule would lead to an infinite loop, this algorithm requires that if the grammar contains an ϵ-production, it is $S \to \epsilon$ and that in that case S does not appear in the right-hand side of any rule. This possible rule is handled separately from the others, in a terminal case.

```
// Input:  $\alpha \in (N \cup \Sigma)^\star$
// Output:  True if $S \overset{\star}{\underset{G}{\Rightarrow}} \alpha$, False otherwise
Function budfr($\alpha$): bool
    if $\alpha = S$ then return True;
    if $\alpha = \epsilon$ and $S \to \epsilon \in P$ then return True;
    // Beginning of the window.
    for $i := 1$ to $len(\alpha)$ do
        // End of the window.
        for $j := i$ to $len(\alpha)$ do
            foreach $(A \to \alpha_i \ldots \alpha_j) \in P$ do
                if budfr($\alpha_1 \ldots \alpha_{i-1}\, A\, \alpha_{j+1} \ldots \alpha_{len(\alpha)}$) $=$ True then
                    return True;
    return False;
```

Algorithm 10: Naive bottom-up depth-first recognition. This algorithm works with CFGs that contain no other ϵ-production than, possibly, the one from the axiom if the axiom is not found in the right-hand side of any rule, and no cycle of non-generative production rules.

- When called on $\alpha = \alpha_1\,\alpha_2\,\cdots\,\alpha_n \in (N \cup \Sigma)^\star$, if neither $\alpha \neq S$ nor $\alpha = \epsilon$ and $S \to \epsilon \in P$, this algorithm first checks whether there is any rule the right-hand side of which is α_1. For all such rule $A \to \alpha_1$, the algorithm calls itself on $\beta = A\,\alpha_2\,\cdots\,\alpha_n$, the result of reversing $A \to \alpha_1$. If the call returns `True`, which means that $S \overset{\star}{\Rightarrow} \beta$, then one knows that $S \overset{\star}{\Rightarrow} \alpha$ (because $\beta \Rightarrow \alpha$) and the algorithm returns `True`, otherwise, it proceeds with the next rule that reverses α_1. If the recognition was not succesful on reversing α_1,
  - the algorithm tries to reverse $\alpha_1\,\alpha_2$ using the same process,
  - then $\alpha_1\,\alpha_2\,\alpha_3$,
  - ...,
  - $\alpha_1\,\alpha_2\,\cdots\,\alpha_n$,
  - α_2,
  - $\alpha_2\,\alpha_3$,
  - ...,
  - $\alpha_2\,\alpha_3\,\cdots\,\alpha_n$,
  - α_3,

[1]A recursive algorithm is one that works by checking whether its input is a particular case for which the output is known ('terminal case') and, if not the case, by calling itself on a modified input, somewhat closer to a terminal case, and from which the output can be determined.

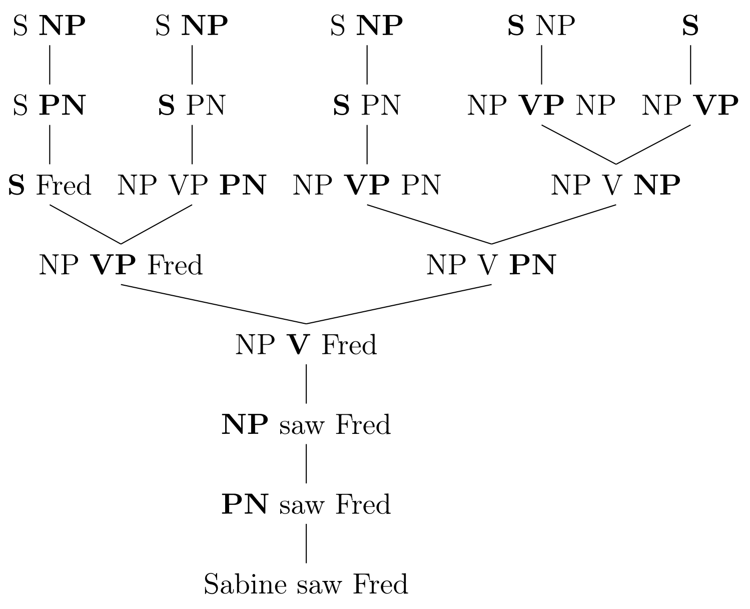


Figure 6.2: The nodes explored during a bottom-up depth-first recognition of *Sabine saw Fred*. This exploration tree is built from bottom to top and from left to right. The non-terminal in the left-hand side of the last rule reversed is indicated in bold.

- ...,
- ...,
- $\alpha_{n-1}\,\alpha_n$.
- α_n.

If nothing was successful, then one can deduce that α cannot be derived from S and the algorithm returns `False`.

Example(s) 6.2 *Consider $G = (N, \Sigma, P, S)$ where:*

- $N = \{S, NP, VP, PN, DET, N\}$*;*
- $\Sigma = \{saw, prepared, the, a(n), truck, experiment, Sabine, Fred, Jamy\}$*;*
- $P = \left\{ \begin{array}{l} S \rightarrow NP\ VP, \\ NP \rightarrow DET\ N \mid PN, \\ VP \rightarrow V \mid V\ NP, \\ DET \rightarrow the \mid a(n), \\ N \rightarrow truck \mid experiment, \\ PN \rightarrow Sabine \mid Fred \mid Jamy, \\ V \rightarrow saw \mid prepared \end{array} \right\}.$

Figure 6.2 illustrates as a tree the exploration of algorithm 10 on Sabine saw Fred.

Remark 6.4 *As in shown in the previous example, this algorithm can perform a lot of redundant computation.*

Exercise 6.1 *Modify algorithm 10 with an additional (mutable) argument so as to not visit the same branch twice.*

Remark 6.5 *Two paths in the search graph from S to $w \in \Sigma^\star$ may correspond to two distinct syntactic structures, even if they start or end identically. This is something to keep in mind when one is interested in all syntactic structures of a word.*

Remark 6.6 (The problem with ϵ-productions) *Because in algorithm 10, only windows of 1 or more symbols are considered, no ϵ-production is ever considered. One could make, in the inner loop, the range of j start from $i-1$ instead of i, thus adding windows of size 0 to the consideration. This move, however, would open the door to infinite loops. Indeed, if the grammar used does contain a rule $X \to \epsilon$, then this variant of algorithm 10 would be forever stuck with $i = 1$, $j = 0$ and $(A \to \alpha_i \cdots \alpha_j) = (X \to \epsilon)$: any call $\texttt{budfr}(\alpha)$ would generate a call $\texttt{budfr}(X\,\alpha)$, and this call would itself generate a call $\texttt{budfr}(X\,X\,\alpha)$, and so on and so forth. This problem is related to the fact that given a sequence α, there might be any arbitrary number of empty categories at any position before, within, or after α. This is why algorithm 10 must be run on a grammar devoid of ϵ-production, such as the ones obtained using the procedure described in Chapter 4.*

Remark 6.7 *When used on a grammar that contains a cycle of non-generative production rules, algorithm 10 might not terminate. This might happen whether or not the input is a sentential form of the grammar. To solve this problem, either:*

- *use a grammar that does not allow cycles (e.g. a clean one);*
- *implement the improvement suggested in exercise 6.1.*

TODO: Illustrate the problem with cycles (for example with a rule NP → NP).

Remark 6.8 *Even on a clean grammar, the complexity of algorithm 10 is terrible (i.e. exponential in the length of the input word). To see this, consider for example the grammar $(\{S, A\}, \{a\}, \{S \to A\,S \mid a, A \to a\}, S)$ and the words a^n (for $n \in \mathbb{N}$) if reversing the rule $S \to a$ is always tried before reversing $A \to a$: $\forall n \in \mathbb{N}$, parsing a^{n+1} takes more that twice as much time as parsing a^n.*

- Alternatively, a *breadth-first* algorithm explores all options 'at the same time', partly exploring each of them in turn.
- This idea is implemented in algorithm 11, a non-recursive algorithm. Because treating ϵ-productions as any other rule would still lead to an infinite loop in case the input is not generated by the grammar, this algorithm too requires that if the grammar contains an ϵ-production, it is $S \to \epsilon$ and that in that case S does not appear in the right-hand side of any rule. This possible rule is handled separately from the others.

Example(s) 6.3 *Consider the same grammar G as in example 6.2:*

- $N = \{S, NP, VP, PN, DET, N\}$;
- $\Sigma = \{saw, prepared, the, a(n), truck, experiment, Sabine, Fred, Jamy\}$;
- $P = \left\{ \begin{array}{l} S \to NP\ VP, \\ NP \to DET\ N \mid PN, \\ VP \to V \mid V\ NP, \\ DET \to the \mid a(n), \\ N \to truck \mid experiment, \\ PN \to Sabine \mid Fred \mid Jamy, \\ V \to saw \mid prepared \end{array} \right\}.$

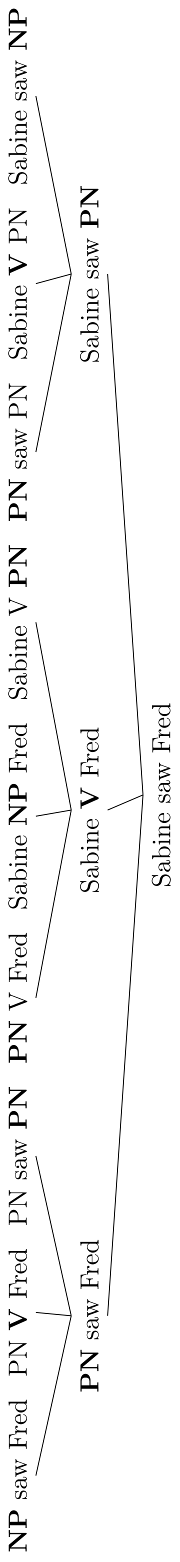


Figure 6.3: The first nodes explored during a bottom-up breadth-first recognition of *Sabine saw Fred*. This exploration tree is built from left to right and from bottom to top (when looked with the root at the bottom). The non-terminal in the left-hand side of the last rule reversed is indicated in bold.

```
// Input:  $\alpha \in (N \cup \Sigma)^\star$
// Output:  True if $S \overset{\star}{\underset{G}{\Rightarrow}} \alpha$, False otherwise
Function bubfr($\alpha$): bool
    $l := [\alpha]$;
    $k := 0$;
    while $k < len(l)$ do
        $u := l[k]$;
        if $u = S$ then return True;
        if $u = \epsilon$ and $S \to \epsilon \in P$ then return True;
        // Beginning of the window.
        for $i := 1$ to $len(u)$ do
            // End of the window.
            for $j := i$ to $len(u)$ do
                foreach $(A \to u_i \dots u_j) \in P$ do
                    $l$.append($u_1 \dots u_{i-1}\, A\, u_{j+1} \dots u_n$);
        $k$ += 1;
    return False;
```

Algorithm 11: Naive bottom-up breadth-first recognition. This algorithm works with CFGs that contain no other ϵ-production than, possibly, the one from the axiom if the axiom is not found in the right-hand side of any rule, and no cycle of non-generative production rules.

Figure 6.3 illustrates as a tree the exploration of algorithm 11 on Sabine saw Fred.

Remark 6.9 *Similarly to algorithm 10, algorithm 11 can perform a lot of redundant computation.*

Exercise 6.2 *Modify algorithm 11 with an additional argument so as to not visit the same branch twice.*

Remark 6.10 *When run on a sentential form of the grammar, algorithm 11 always terminates, even if the grammar contains cycles of non-generative production rules, because the correct path will be found eventually. When run on other inputs, however, cycles can lead the algorithm to not terminate. To solve this problem, either:*

- *use a grammar that does not allow cycles (e.g. a clean one);*
- *implement the improvement suggested in exercise 6.2.*

Exercise 6.3 *Taking inspiration from algorithm 11 and using a list as a stack instead of as a queue (i.e. using the* pop *method instead of iterating over its elements/their positions in a linear fashion), rewrite algorithm 10 in a non-recursive way.*

Remark 6.11 *Two distinct paths in the search graph from S to $w \in \Sigma^\star$ might encode the same syntactic structure, and in this case there is no point in exploring them both — even if one is interested in the full parsing task rather than simply in the recognition task. However, a syntactic structure always corresponds to a unique left (resp. right) derivation, so one might safely restrict themselves to left (resp. right) derivations.*

The shift-reduce *approach introduced below relies on this idea to reduce useless computation, by considering only right derivations.*

Definition 6.2 (Shift-reduce state, stack and buffer) *A shift-reduce* state *is a pair* $(\beta, v) \in (N \cup \Sigma)^\star \times \Sigma^\star$*. The first element of the pair* (β) *is the* stack *and the second* (v) *is the* buffer*.*

Definition 6.3 (Shift-reduce action) *There are two types of shift-reduce* actions*, which are partial functions from states to states:*

- *the shift action is defined on all states* (β, v) *such that* $v \neq \epsilon$ *and is defined by* $\mathit{shift}((\beta_1\,\beta_2\,\cdots\beta_n, v_1\,v_2\,\cdots\,v_m)) = (\beta_1\,\beta_2\,\cdots\beta_n\,v_1, v_2\,\cdots\,v_m)$*,*
- *for each rule* $A \to \alpha$ *the* $\mathit{reduce}[A \to \alpha]$ *action is defined on all states* (β, v) *such that* β *ends with* α *and is defined by* $\mathit{reduce}[A \to \alpha]((\beta_1\,\beta_2\,\cdots\beta_n, v_1\,v_2\,\cdots\,v_m)) = (\beta_1\,\beta_2\,\cdots\beta_{n-|\alpha|}\,A, v_1\,v_2\,\cdots\,v_m)$*,*

Remark 6.12

- *The shift action is described as dequeuing the first letter of the buffer and pushing it onto the stack. This action is only possible if the buffer is not empty.*
- *A* $\mathit{reduce}[A \to \alpha]$ *action is described as popping* α *from the stack and pushing* A *instead. This action is only possible if* α *is present on top of the stack.*

Definition 6.4 (The shift-reduce paradigm) *Given a CFG* (N, Σ, P, S) *and* $w \in \Sigma^\star$*, the shift-reduce paradigm consists in analysing* w *by exploring the state spaces from* (ϵ, w) *in search of* (S, ϵ)*, going from one state to another by appling an action.*

Remark 6.13

- *While a shift-reduce analysis is defined as an exploration of the state space, it also corresponds to an exploration of the search graph: Applying the shift action is not seen as a move in the search graph, but applying a reduce action from state* (β, v) *to state* (β', v') *is seen as a move from node* $\beta\,v$ *to node* $\beta'\,v'$ *(which is always a parent node).*
- *A shift-reduce analysis follows the bottom-up approach.*
- *Because two or more different actions might be defined on the same state, a shift-reduce algorithm needs to specify how to explore all the corresponding paths. This can both be done in a depth-first and in a breadth-first fashion.*

Example(s) 6.4 *Consider the grammar from example 6.2. Figure 6.4 illustrates as a tree the depth-first shift-reduce recognition of* Sabine saw Fred*.*

Exercise 6.4 *Taking inspiration from algorithm 10, write an algorithm that takes as input a shift-reduce state* (β, v) *and, following the shift-reduce paradigm with the depth-first approach, outputs a boolean indicating whether* $S \underset{G}{\overset{\star}{\Rightarrow}} \beta\,v$*.*

Exercise 6.5 *Taking inspiration from algorithm 11, write an algorithm that takes as input a shift-reduce state* (β, v) *and, following the shift-reduce paradigm with the breadth-first approach, outputs a boolean indicating whether* $S \underset{G}{\overset{\star}{\Rightarrow}} \beta\,v$*.*

Remark 6.14 *Shift-reduce analysis works by 'undoing' right derivations. Such an analysis might visit the same nodes twice, but not via paths that are equivalent in terms of syntactic structures. This can be illustrated with an ambiguous word* w *of the grammar; all paths found from* (ϵ, w) *to* (S, ϵ) *correspond to different syntactic structures.*

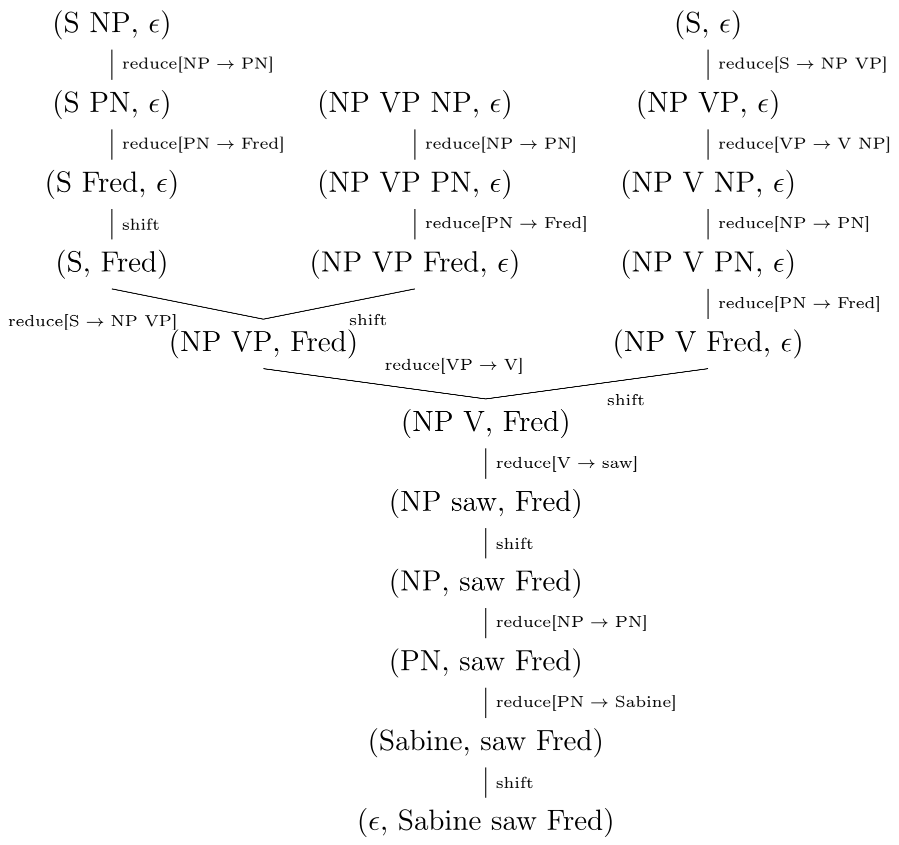


Figure 6.4: The states explored during a depth-first shift-reduce recognition of *Sabine saw Fred*, assuming that reduce actions are always tried before the shift action. This exploration tree is built from bottom to top and from left to right.

6.3 Top-down recognition

- As said above, a top-down algorithm starts from S and then explores the search space from parent nodes to children ones.
- The bottom-up algorithms studied above always terminate when the grammar is clean. This is because any node in the search graph of a clean grammar has only a finite number of ancestors. One would like a similar guarantee for the top-down approach.
- There is no way to make sure that the S node (or any other node) has only a finite number of descendants in the search space. However, it is possible to get a satisfying guarantee using the following property.

Property 6.2 *Let $\alpha \in (N \cup \Sigma)^\star$ and $u \in \Sigma^\star$ such that $\alpha \neq u$. Let $\gamma \in \Sigma^\star$ be their longest common prefix, and $\alpha' \in (N \cup \Sigma)^\star$ and $u' \in \Sigma^\star$ be their remaining suffixes (i.e. $\alpha = \gamma\, \alpha'$ and $u = \gamma\, u'$). If $\alpha' = \epsilon$ or if $\alpha'_1 \in \Sigma$, then u cannot be found among the descendants of α in the search graph.*

Example(s) 6.5 *Let G be once again the grammar from example 6.2.*

1. *Consider $\alpha =$ Sabine saw and $u =$ Sabine saw Fred. Their longest common prefix is $\gamma =$ Sabine saw, and $\alpha' = \epsilon$ and $u' =$ Fred are their remaining suffixes. The fact that $\alpha' = \epsilon$ proves that u is not a descendant of α in the search graph.*
2. *Consider $\alpha =$ Sabine saw a(n) N and $u =$ Sabine saw Fred. Their longest common prefix is $\gamma =$ Sabine saw, and $\alpha' =$ a(n) N and $u' =$ Fred are their remaining suffixes. The fact that $\alpha'_1 \in \Sigma$ proves that u is not a descendant of α in the search graph.*

Definition 6.5 (Possible left-ancestor) *Let $G = (N, \Sigma, P, S)$ be a grammar and $u \in \Sigma^\star$. Any $\alpha \in (N \cup \Sigma)^\star$ with $\alpha \neq u$ is* a possible left-ancestor *of u iff:*

- $S \overset{\star L}{\Rightarrow} \alpha$;
- *with $\alpha' \in (N \cup \Sigma)^\star$ and $u' \in \Sigma^\star$ the remaining suffixes of α and u once their longest common prefix has been removed, $|\alpha'| \geq 1$ and $\alpha'_1 \in N$.*

Property 6.3 *Let $\alpha \in (N \cup \Sigma)^\star$ and $u \in \Sigma^\star$ such that $\alpha \neq u$.*

- *If α is not a possible left-ancestor of u, then u cannot be derived from α.*
- *If α is a possible left-ancestor of u, then u might or might not be derived from α.*

Definition 6.6 (Left-recursive derivation) *Let G be a grammar. A* left-recursive derivation *is a derivation of length $n \geq 1$ from a non-terminal A to a sequence of symbols $A\, \beta$ where $\beta \in (N \cup \Sigma)^\star$:*

$$A \underset{G}{\overset{+}{\Rightarrow}} A\, \beta$$

Property 6.4 *Let $G = (N, \Sigma, P, S)$ be a grammar that admits no left-recursive derivation. For any $u \in \Sigma^\star$, there is always only a finite number of nodes in the search graph that are possible left-ancestors of u.*

Remark 6.15 *There is an algorithm that can turn any CFG into a weakly equivalent CFG that admits no left-recursive derivation.*

- A simple top-down approach for the recognition task on $u \in \Sigma^\star$ then consists in exploring the search graph starting from S and rewriting the leftmost non-terminal of the possible left-ancestors of u encountered until u is found or there is no more possible left-ancestor of u to rewrite.

- For purely practical reasons, in the algorithms below, instead of directly working with a sentential form and the word to recognise, their longest common prefix will be removed; we will work on their remaining suffixes. Given $\alpha \in (N \cup \Sigma)^\star$ and $u \in \Sigma^\star$, let $\gamma \in \Sigma^\star$ be their longest common prefix, and $\alpha' \in (N \cup \Sigma)^\star$ and $u' \in \Sigma^\star$ be their remaining suffixes; the pair (α', u') is a *parsing state*.

- Let us consider $\alpha' \in (N \cup \Sigma)^\star$ such that $|\alpha'| \geq 1$ and $\alpha'_1 \in N$. As there might be multiple rules with α'_1 as left-hand side, our top-down algorithms also need to specify how to explore all the corresponding branches.

- Algorithm 12, a recursive algorithm, implements the top-down approach in a depth-first fashion.

```
// Input:  $\alpha \in (N \cup \Sigma)^\star$ and $u \in \Sigma^\star$
// Output:  True if $\alpha \overset{\star}{\Rightarrow} u$, False otherwise
Function tddfr($\alpha$, $u$): bool
    if $\alpha = u$ then return True;
    // Remaining suffixes; $\alpha = \gamma\,\alpha'$ and $u = \gamma\,u'$
    $\alpha', u' :=$ remaining_suffixes($\alpha, u$);
    if $len(\alpha') = 0$ or $\alpha'_1 \in \Sigma$ then return False;
    // Can $\alpha'$ be rewritten as $u'$?
    foreach $(\alpha'_1 \to \beta) \in P$ do
        if tddfr($\beta\,\alpha'_2\,\alpha'_3 \ldots \alpha'_{len(\alpha')}$, $u'$) = True then
            return True;
    return False;
```

Algorithm 12: Naive top-down depth-first recognition. This algorithm works with CFGs that admit no left-recursive derivation.

Example(s) 6.6 *Consider $G = (N, \Sigma, P, S)$ where:*

- $N = \{S, NP, VP, PN, DET, N\}$;
- $\Sigma = \{saw, prepared, the, a(n), truck, experiment, Sabine, Fred, Jamy\}$;
- $$P = \left\{ \begin{array}{l} S \to NP\ VP, \\ NP \to DET\,N \mid PN, \\ VP \to V \mid V\,NP, \\ DET \to the \mid a(n), \\ N \to truck \mid experiment, \\ PN \to Sabine \mid Fred \mid Jamy, \\ V \to saw \mid prepared \end{array} \right\}.$$

Figure 6.5 illustrates as a tree the exploration of algorithm 12 on Sabine saw Fred.

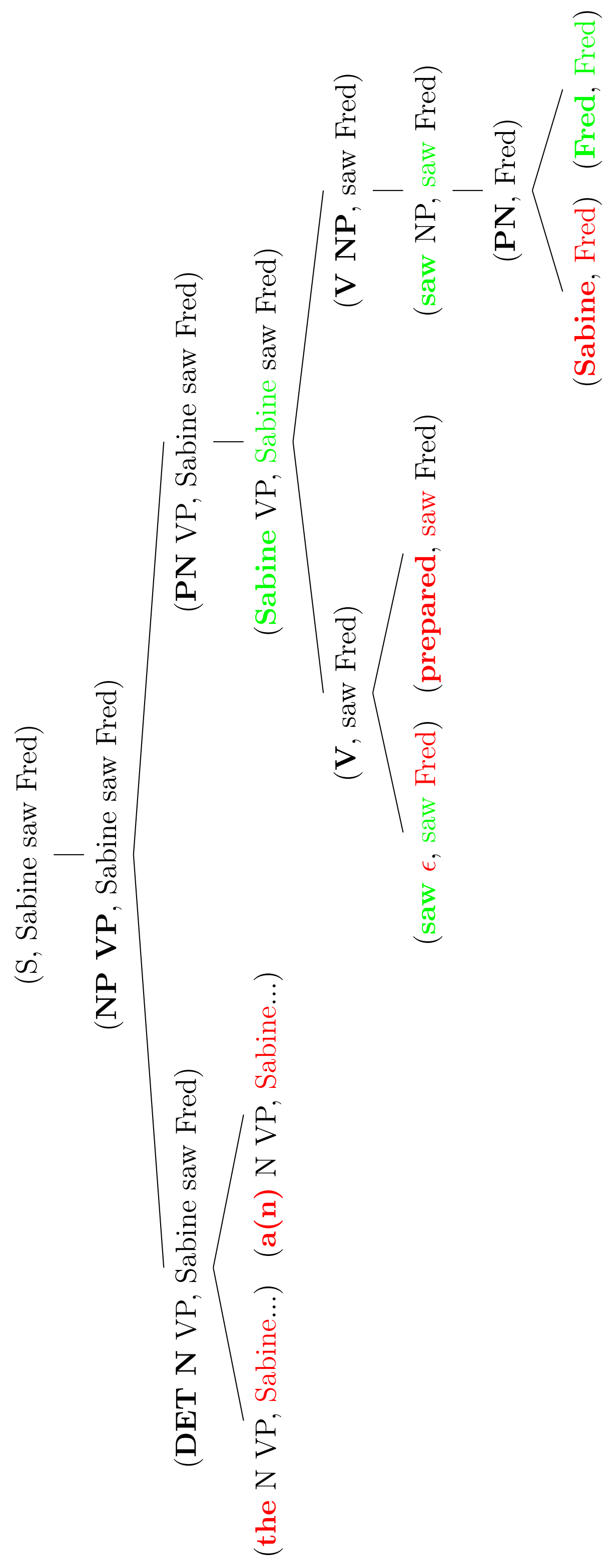


Figure 6.5: The parsing states seen during a top-down depth-first recognition of *Sabine saw Fred*. This exploration tree is built from top to bottom and from left to right (when looked with the root at the top). The symbols generated by the last rule applied are indicated in bold. Longest common prefixes are indicated in green.

Remark 6.16 *When used on a grammar that admits left-recursive derivations, algorithm 12 might not terminate. This might happen whether or not the input is a pair* (α, u) *such that* $\alpha \overset{\star}{\Rightarrow} u$.

Remark 6.17 *Even on a grammar that admits no left-recursive derivation, the complexity of algorithm 12 is terrible (i.e. exponential in the length of the input word). To see this, consider for example the grammar* $(\{S, A, B\}, \{a, b\}, \{S \to A\,a \mid B\,b, A \to a\,A \mid a\,B \mid a, B \to a\,A \mid a\,B \mid a\}, S)$ *and the words* $a^n\,b$ *(for* $n \in \mathbb{N}$*) if applying the rule* $S \to A\,a$ *is tried before applying* $S \to B\,b$*:* $\forall n \in \mathbb{N}$*, parsing* $a^{n+1}\,b$ *takes roughly (that is to say, ignoring a small constant additive factor) twice as much time as parsing* $a^n\,b$*.*

- Algorithm 13, a non-recursive algorithm, implements the top-down approach in a breadth-first fashion.

```
// Input:  $u \in \Sigma^\star$
// Output:  True if $S \overset{\star}{\underset{G}{\Rightarrow}} u$, False otherwise
Function tdbfr($u$): bool
    $l := [(S, u)]$;
    $i := 0$;
    while $i < len(l)$ do
        $(\alpha, v) := l[i]$;
        if $\alpha = v$ then return True;
        // Remaining suffixes; $\alpha = \gamma\,\alpha'$ and $v = \gamma\,v'$
        $\alpha', v' :=$ remaining_suffixes$(\alpha, v)$;
        if $len(\alpha') \geq 1$ and $\alpha'_1 \in N$ then
            // Can $\alpha'$ be rewritten as $v'$?
            foreach $(\alpha'_1 \to \beta) \in P$ do
                $l$.append$((\beta\,\alpha'_2\,\alpha'_3\,\ldots\alpha'_{\text{len}(\alpha')}, v'))$
        $i$ += 1;
    return False;
```

Algorithm 13: Naive top-down breadth-first recognition. This algorithm works with CFGs that admit no left-recursive derivation.

Remark 6.18 *When run on a pair* (α, u) *such that* $\alpha \overset{\star}{\Rightarrow} u$*, algorithm 13 always terminates, even if the grammar admits left-recursive derivation, because the correct path will be found eventually. When run on other inputs, however, left-recursive derivations can lead the algorithm to not terminate.*

Exercise 6.6 *Taking inspiration from algorithm 13 and using a list as a stack instead of as a queue (i.e. using the* `pop` *method instead of iterating over its elements/their positions in a linear fashion), rewrite algorithm 12 in a non-recursive way.*

Vocabulary

- the search graph of a grammar: *le graphe de recherche d'une grammaire*
- bottom-up analysis: *analyse ascendante*

- top-down analysis: *analyse descendante*
- depth-first analysis: *analyse en profondeur (d'abord)*
- breadth-first analysis: *analyse en largeur (d'abord)*

Chapter 7

Deterministic analysis of context-free grammars

7.1 Introduction

Remark 7.1 *All of the algorithms of the previous chapter have an exponential worst-case complexity. One of the reasons for this is that they identify only very late that a given direction in the search graph cannot be successful.*

Example(s) 7.1 *Consider* $G = (N, \Sigma, P, S)$ *where:*

- $N = \{S, A, B\}$*;*
- $\Sigma = \{a, b, c\}$*;*
- $P = \left\{ \begin{array}{l} S \to C\,A \mid C\,B, \\ A \to a, \\ B \to b, \\ C \to c\,C \mid c \end{array} \right\}.$

Figure 7.1 illustrates with the exploration tree of the top-down depth-first algorithm 12 on $c\,c\,b$ *how a naive algorithm can spend most of its time exploring a branch that in principle could have been identified as useless from the start.*

Definition 7.1 (Deterministic analysis) *An analysis is* deterministic *iff only a single path of the search graph is explored (i.e. iff the exploration tree is a path), and the analysis requires only constant-time work (in terms of the length of the input) to go from one node to another.*

Intuition 7.1 *At each step of a deterministic analysis, one knows whether to stop or not, and if not, exactly which parent/child to visit.*

Property 7.1 *If a recognition or parsing algorithm only performs deterministic analyses, the size of its exploration tree is linear (in the length of the input sequence of symbols).*

Remark 7.2 *There is no deterministic recogniser or parser for all CFGs. There are, however, known deterministic recognisers and parsers for some specific classes of CFGs. They usually require some preprocessing of the grammar.*

- There are two main families of deterministic parsers: LL parsers and LR ones.

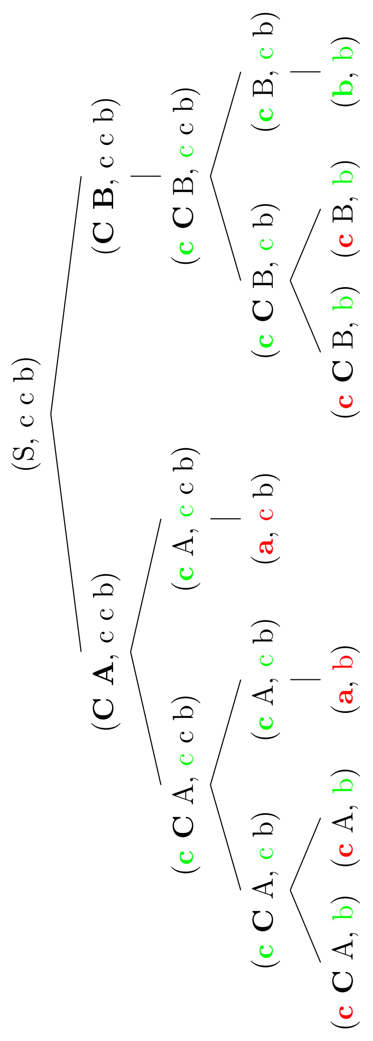


Figure 7.1: The parsing states seen during a top-down depth-first recognition of $c\,c\,b$. Knowing that the only letter that A can generate is a, it is obvious that the left-branch (rooted on the sentential form $C\,A$) cannot lead to a recognition of the word, which ends in b. A naive algorithm does no see this and fully explores this useless branch.

- 'LL' stands for 'Left-to-right Leftmost'. They follow a top-down approach rewriting the non-terminals from left to right, which thus produces a left-derivation.
- 'LR' stands for 'Left-to-right Rightmost'. They follow a bottom-up approach reversing the spans from left to right, which thus produces a right-derivation.

Remark 7.3 *Remember that a derivation being a left or right one has nothing to do with the shape of the derivation tree, but is about the order of the rule applications in the derivation from the axiom.*

Consider the simple grammar $G = (\{S, A, B\}, \{a, b\}, \{S \to A\,B, A \to a, B \to b\}, S)$ *and the word* $a\,b$.

- *A left-to-right top-down analysis is:* $S \Rightarrow A\,B \Rightarrow a\,B \Rightarrow a\,b$ *(a left-derivation).*
- *A left-to-right bottom-up analysis is:* $a\,b \Leftarrow A\,b \Leftarrow A\,B \Leftarrow S$ *(a right-derivation).*

Both derivations have the same derivation tree.

Remark 7.4 *The LL and LR algorithms studied below are designed to find the only correct derivation of their input, which means that they cannot work with ambiguous grammars.*

7.2 LL analysis

Definition 7.2 (SLL(1) grammar) *A CFG* $G = (N, \Sigma, P, S)$ *is SLL(1) iff:*

- $\forall A \to \alpha \in P$, $|\alpha| \geq 1$ *and* $\alpha_1 \in \Sigma$;
- $\forall A \to \alpha, A \to \beta \in P$, $(\alpha \neq \beta) \Rightarrow (\alpha_1 \neq \beta_1)$.

Property 7.2 *SLL(1) grammars are super easy to parse with.*

Let $G = (N, \Sigma, P, S)$ *be an SLL(1) grammar. Let us assume that one is trying to analyse* $u \in \Sigma^\star$ *in a top-down fashion. At node* $\alpha \in (N \cup \Sigma)^\star$,

- *if* $\alpha = u$, *then* u *has been recognised;*
- *otherwise, let* α' *and* u' *be the remaining suffixes of* α *and* u *after their longest common prefix has been removed (note that* α' *and* u' *cannot both be empty because* $\alpha \neq u$*); if* $|\alpha'| = 0$, *then* $u' \neq \emptyset$ *cannot be derived from* $\alpha' = \emptyset$, *and if* $|u'| = 0$, *then* $u' = \emptyset$ *cannot be derived from* $\alpha' \neq \emptyset$ *(as there is no* ϵ*-production in an SLL(1) grammar); in both cases,* u *cannot be derived from* α;
- *otherwise, the only way that* u *could be derived from* α *is if there is a rule* $\alpha'_1 \to \beta \in P$ *such that* $\beta_1 = u'_1$*; if there is none, then* u *cannot be derived from* α;
- *otherwise, because* G *is SLL(1), there is only one* $\alpha'_1 \to \beta \in P$ *such that* $\beta_1 = u'_1$ *and the only way to proceed consists in going to the node obtained by rewriting* α'_1 *with this rule.*

Example(s) 7.2 *Consider* $G = (\{S, A, B\}, \{a, b, c\}, P, S)$ *where* $P = \left\{ \begin{array}{l} S \to a\,A\,B, \\ A \to a\,A \mid b\,B \mid c, \\ B \to a\,B \mid b\,A \mid c \end{array} \right\}$.

G *is an SLL(1) grammar. Figure 7.2 illustrates the deterministic top-down analysis of* $a\,a\,c\,a\,b\,c$.

$$\begin{tikzcd}
(S,\, a\,a\,c\,a\,b\,c) \arrow[d, dash] \\
(\boldsymbol{a\,A\,B},\, a\,a\,c\,a\,b\,c) \arrow[d, dash] \\
(\boldsymbol{a\,A}\,B,\, a\,c\,a\,b\,c) \arrow[d, dash] \\
(\boldsymbol{c}\,B,\, c\,a\,b\,c) \arrow[d, dash] \\
(\boldsymbol{a\,B},\, a\,b\,c) \arrow[d, dash] \\
(\boldsymbol{b\,A},\, b\,c) \arrow[d, dash] \\
(\boldsymbol{c},\, c)
\end{tikzcd}$$

Figure 7.2: The parsing states seen during a deterministic top-down recognition of $a\,a\,c\,a\,b\,c$. The symbols generated by the last rule applied are indicated in bold. Longest common prefixes are indicated in green.

Remark 7.5 *'SLL(1)' stands for 'Simple LL with 1 lookahead'.*

Exercise 7.1 *Taking inspiration from the top-down algorithms of the previous chapter, use property 7.2 to write a deterministic recognition algorithm for SLL(1) grammars.*

Remark 7.6 *Unfortunately, very few languages are generated by an SLL(1) grammar. One can, however, generalise a little bit the idea behind SLL(1) parsing.*

Example(s) 7.3 *Consider $G = (N, \Sigma, P, S)$ where:*

- $N = \{S, A, B\}$;
- $\Sigma = \{a, b, c\}$;
- $P = \left\{ \begin{array}{l} S \to A\,B\,S \mid c, \\ A \to a, \\ B \to A\,c \mid b \end{array} \right\}.$

This grammar is not SLL(1). Yet, it is easy to see that if α' is a suffix of a sentential form of G, if α' is of the form

- *$S\cdots$, it can only generates sequences of letters starting with a, if the initial S is rewritten with $S \to A\,B\,S$, or starting with c, if the initial S is rewritten with $S \to c$,*
- *$A\cdots$, it can only generates sequences of letters starting with a, if the initial A is rewritten with $A \to a$,*
- *$B\cdots$, it can only generates sequences of letters starting with a, if the initial B is rewritten with $B \to A\,c$, or starting with b, if the initial B is rewritten with $B \to b$,*

So, during a top-down analysis, if the state is (α', u') once the longest common prefix has been removed, one just needs to look at the first symbol of α' and the first symbol of u' (if they exist) to determine whether the recognition is a failure, or which is the unique rule that needs to be considered next:

- *if $\alpha' = \epsilon$ or $\alpha'_1 \in \Sigma$, no rule needs to be considered as u' cannot be generated from α' (failure);*
- *if $(\alpha', u') = (S \cdots, a \cdots)$, only $S \to A\,B\,S$ needs to be considered — we will write this property as '$r_{S,a} = S \to A\,B\,S$' —;*
- *if $(\alpha', u') = (S \cdots, b \cdots)$, no rule needs to be considered as u' cannot be generated from α' (failure) — we will write this property as '$r_{S,b} = 0$' —;*
- *if $(\alpha', u') = (S \cdots, c \cdots)$, only $S \to c$ needs to be considered — we will write this property as '$r_{S,c} = S \to c$' —;*
- *if $(\alpha', u') = (S \cdots, \epsilon)$, no rule needs to be considered as u' cannot be generated from α' (failure) — we will write this property as '$r_{S,\#} = 0$' —;*
- *if $(\alpha', u') = (A \cdots, a \cdots)$, only $A \to a$ needs to be considered — we will write this property as '$r_{A,a} = A \to a$' —;*
- *if $(\alpha', u') = (A \cdots, b \cdots)$, no rule needs to be considered (failure) — we will write this property as '$r_{A,b} = 0$' —;*
- *if $(\alpha', u') = (A \cdots, c \cdots)$, no rule needs to be considered (failure) — we will write this property as '$r_{A,c} = 0$' —;*
- *if $(\alpha', u') = (A \cdots, \epsilon)$, no rule needs to be considered (failure) — we will write this property as '$r_{A,\#} = 0$' —;*
- *if $(\alpha', u') = (B \cdots, a \cdots)$, only $B \to A\,c$ needs to be considered — we will write this property as '$r_{B,a} = B \to A\,c$';*
- *if $(\alpha', u') = (B \cdots, b \cdots)$, only $B \to b$ needs to be considered — we will write this property as '$r_{B,b} = B \to b$';*
- *if $(\alpha', u') = (B \cdots, c \cdots)$, no rule needs to be considered (failure) — we will write this property as '$r_{B,c} = 0$';*
- *if $(\alpha', u') = (B \cdots, \epsilon)$, no rule needs to be considered (failure) — we will write this property as '$r_{B,\#} = 0$'.*

If, for example, the rule $S \to a$ were also present in the grammar, then it would not always be possible to select a single rule to apply next based only on the first symbol of α' and u'. In particular, if $(\alpha', u') = (S \cdots, a \cdots)$, both $S \to A\,B\,S \mid a$ would have to be considered.

Intuition 7.2 (LL(1) grammar) *An LL(1) grammar is a grammar, like the one from example 7.3, for which it is possible to perform a deterministic top-down analysis such that in a state (α, u),*

- *acceptation is triggered by the condition $\alpha = u$*
- *and the other actions (i.e. rejection, application of a given rule) can be taken based only on the first symbol of the remaining suffixes (after suppression of the longest common prefix) α' and u' of α and u if they exist.*

For such a grammar, in a parsing state (α, u) such that $\alpha \neq u$, $\alpha' \neq \epsilon$ and $\alpha'_1 \in N$, with $X = \alpha'_1$ and $x = (v\,\#)_1$ (i.e. $\#$ if $v = \epsilon$ and v_1 otherwise),

- *if one rule needs to be considered, this rule is named '$r_{X,x}$',*

- *otherwise, we write '$r_{X,x} = 0$'.*

The '1' in 'LL(1)' refers to the fact that, in the case $\alpha \neq u$, the decision is taken looking at at most one symbol of u'.

Remark 7.7 *The grammar in example 7.3 does not contain any ϵ-production, which has the two following consequences:*

1. $\forall X \in N$, $r_{X,\#} = 0$.
2. $\forall X \in N$, $\forall x \in \Sigma$, *as X cannot be reduced to ϵ, in order to determine $r_{X,x}$, it is not necessary to look at the symbols that may appear on the right of X in derivations from S.*

In general, these two properties are not satisfied by grammars that contain ϵ-productions.

Example(s) 7.4 *Consider $G = (N, \Sigma, P, S)$ where:*

- $N = \{S, A, B\}$;
- $\Sigma = \{a, b, c\}$;
- $P = \left\{ \begin{array}{l} S \to A\,B\,S \mid c \mid \epsilon, \\ A \to a, \\ B \to b\,B \mid \epsilon \end{array} \right\}$.

Let α' be a suffix of a sentential form of G. If α' is of the form

- $S \cdots$, *it can only generates*
  - *sequences of letters starting with a, if the initial S is rewritten with $S \to A\,B\,S$,*
  - *or starting with c, if the initial S is rewritten with $S \to c$,*
  - *or the empty sequence ϵ, if the initial S is rewritten with $S \to \epsilon$ (notice here that given G, S can only appear as the rightmost symbol in a sentential form of G so if α' is of the form $S \cdots$, in fact $\alpha' = S$),*
- $A \cdots$, *it can only generates sequences of letters starting with a, if the initial A is rewritten with $A \to a$.*
- $B \cdots$, *it can only generates*
  - *sequences of letters starting with b, if the initial B is rewritten with $B \to b\,B$,*
  - *or starting with a, if the initial B is rewritten with $B \to \epsilon$,*
  - *or starting with c, if the initial B is rewritten with $B \to \epsilon$,*
  - *or the empty sequence ϵ, if the initial B is rewritten with $B \to \epsilon$.*

So, let (α', u') be a parsing state once the longest common prefix has been removed,

- *if $(\alpha', u') = (S \cdots, a \cdots)$, only $S \to A\,B\,S$ needs to be considered: $r_{S,a} = S \to A\,B\,S$;*
- *if $(\alpha', u') = (S \cdots, b \cdots)$, no rule needs to be considered: $r_{S,b} = 0$;*
- *if $(\alpha', u') = (S \cdots, c \cdots)$, only $S \to c$ needs to be considered: $r_{S,c} = S \to c$;*
- *if $(\alpha', u') = (S \cdots, \epsilon)$, only $S \to \epsilon$ needs to be considered: $r_{S,\#} = 0$;*

- *if* $(\alpha', u') = (A \cdots, a \cdots)$, *only* $A \to a$ *needs to be considered:* $r_{A,a} = A \to a$;
- *if* $(\alpha', u') = (A \cdots, b \cdots)$, *no rule needs to be considered:* $r_{A,b} = 0$;
- *if* $(\alpha', u') = (A \cdots, c \cdots)$, *no rule needs to be considered:* $r_{A,c} = 0$;
- *if* $(\alpha', u') = (A \cdots, \epsilon)$, *no rule needs to be considered:* $r_{A,\#} = 0$;
- *if* $(\alpha', u') = (B \cdots, a \cdots)$, *only* $B \to \epsilon$ *needs to be considered:* $r_{B,a} = B \to \epsilon$;
- *if* $(\alpha', u') = (B \cdots, b \cdots)$, *only* $B \to b\,B$ *needs to be considered:* $r_{B,b} = B \to b\,B$;
- *if* $(\alpha', u') = (B \cdots, c \cdots)$, *only* $B \to \epsilon$ *needs to be considered:* $r_{B,c} = B \to \epsilon$;
- *if* $(\alpha', u') = (B \cdots, \epsilon)$, *only* $B \to \epsilon$ *needs to be considered:* $r_{B,\#} = B \to \epsilon$.

Definition 7.3 (LL(1) grammar) *A CFG* $G = (N, \Sigma, P, S)$ *is an* LL(1) *grammar iff for all pairs* $(A, a) \in N \times \Sigma$,

- *either there is no* $\gamma \in \Sigma^\star$, $\alpha \in (N \cup \Sigma)^\star$ *and* $v \in \Sigma^\star$ *such that* $S \overset{\star}{\Rightarrow} \gamma\,A\,\alpha \overset{\star}{\Rightarrow} \gamma\,a\,v$ *— in this case, we write '*$r_{A,a} = 0$*' —,*
- *or there is a rule* $A \to \beta \in P$ *such that for all* $\gamma \in \Sigma^\star$, $\alpha \in (N \cup \Sigma)^\star$ *and* $v \in \Sigma^\star$ *such that* $S \overset{\star}{\Rightarrow} \gamma\,A\,\alpha \overset{\star}{\Rightarrow} \gamma\,a\,v$, *then* $A \to \beta$ *is the unique rule that can be used to rewrite* A *in order to generate* $\gamma\,a\,v$ *from* $\gamma\,A\,\alpha$ *— in this case, we define* $r_{A,a} = A \to \beta$ *—*

and for all $A \in N$,

- *either there is no* $\gamma \in \Sigma^\star$ *and* $\alpha \in (N \cup \Sigma)^\star$ *such that* $S \overset{\star}{\Rightarrow} \gamma\,A\,\alpha \overset{\star}{\Rightarrow} \gamma$ *— in this case, we write '*$r_{A,\#} = 0$*' —,*
- *or there is a rule* $A \to \beta \in P$ *such that for all* $\gamma \in \Sigma^\star$ *and* $\alpha \in (N \cup \Sigma)^\star$ *such that* $S \overset{\star}{\Rightarrow} \gamma\,A\,\alpha \overset{\star}{\Rightarrow} \gamma$, *then* $A \to \beta$ *is the unique rule that can be used to rewrite* A *in order to generate* γ *from* $\gamma\,A\,\alpha$ *— in this case, we define* $r_{A,\#} = A \to \beta$.

Remark 7.8 (Alternative formulation) *Let* $G = (N, \Sigma, P, S)$ *be a CFG and* $\# \notin (N \cup \Sigma)$, *G is an LL(1) grammar iff all pairs* $(A, a) \in N \times (\Sigma \cup \{\#\})$,

- *either there is no* $\gamma \in \Sigma^\star$, $\alpha \in (N \cup \Sigma)^\star$ *and* $v \in \Sigma^\star$ *with* $a = (v\,\#)_1$ *(i.e.* $\#$ *if* $v = \epsilon$ *and* v_1 *otherwise) such that* $S \overset{\star}{\Rightarrow} \gamma\,A\,\alpha \overset{\star}{\Rightarrow} \gamma\,v$ *— in this case, we write '*$r_{A,a} = 0$*' —,*
- *or there is a rule* $A \to \beta \in P$ *such that for all* $\gamma \in \Sigma^\star$, $\alpha \in (N \cup \Sigma)^\star$ *and* $v \in \Sigma^\star$ *such that* $S \overset{\star}{\Rightarrow} \gamma\,A\,\alpha \overset{\star}{\Rightarrow} \gamma\,v$ *with* $a = (v\,\#)_1$, *then* $A \to \beta$ *is the unique rule that can be used to rewrite* A *in order to generate* $\gamma\,v$ *from* $\gamma\,A\,\alpha$ *— in this case, we define* $r_{A,a} = A \to \beta$.

The use of a fresh 'end of word' letter # here somewhat simplifies the definition of LL(1) grammars.

Property 7.3 *LL(1) grammars are super easy to parse with.*

Let $G = (N, \Sigma, P, S)$ *be an LL(1) grammar. Let us assume that one is trying to analyse* $u \in \Sigma^\star$ *in a top-down fashion. At node* $\alpha \in (N \cup \Sigma)^\star$,

- *if* $\alpha = u$, *then* u *has been recognised;*
- *otherwise, let* α' *and* u' *be the remaining suffixes of* α *and* u *after their longest common prefix has been removed; if* $|\alpha'| = 0$, *or* $\alpha'_1 \in \Sigma$, *then* u *cannot be derived from* α;

- *otherwise, let $A = \alpha'_1$ and $a = (u'\,\#)_1$ (i.e. $\#$ if $u' = \epsilon$ and u'_1 otherwise), if $r_{A,a} = 0$, then u cannot be derived from α;*
- *otherwise, the only way to proceed consists in going to the node obtained by applying $r_{A,a}$.*

Definition 7.4 *Let $G = (N, \Sigma, P, S)$ be a CFG. For $(A, a) \in N \times \Sigma$,*

- $R_{A,a} = \{A \to \alpha \in P \mid \exists \gamma \in \Sigma^\star,\, \beta \in (N \cup \Sigma)^\star,\, v \in \Sigma^\star,\ S \overset{\star}{\Rightarrow} \gamma\, A\, \beta \Rightarrow \gamma\, \alpha\, \beta \overset{\star}{\Rightarrow} \gamma\, a\, v\}$*;*
- $R_{A,\#} = \{A \to \alpha \in P \mid \exists \gamma \in \Sigma^\star,\, \beta \in (N \cup \Sigma)^\star,\ S \overset{\star}{\Rightarrow} \gamma\, A\, \beta \Rightarrow \gamma\, \alpha\, \beta \overset{\star}{\Rightarrow} \gamma\}$*.*

Remark 7.9 *Let $G = (N, \Sigma, P, S)$ be a CFG. G is an LL(1) grammar iff for all pairs $(A, a) \in N \times (\Sigma \cup \{\#\})$, $|R_{A,a}| \leq 1$.*

Remark 7.10 *There exists an algorithm that, given as input a CFG $G = (N, \Sigma, P, S)$, computes $(R_{A,a})_{(A,a) \in N \times (\Sigma \cup \{\#\})}$.*

Intuition 7.3 (LL(1) parsing table) *Let $G = (N, \Sigma, P, S)$ be a CFG. The* LL(1) parsing table *of G is the 2D table with lines labelled in N and columns labelled in $\Sigma \cup \{\#\}$ such that the cell (A, a) contains the set of rules that would need to be considered, when perfoming a top-down analysis, if the only things that are known about the parsing state (α, u) are that $\alpha \neq u$ and that the remaining suffixes α' and u' are such that α' starts with A and $a = (u'\,\#)_1$.*

In case G is LL(1), a cell (A, a) is empty if $r_{A,a} = 0$ and contains only $r_{A,a}$ otherwise. In case G is not LL(1), however, at least one cell contains two rules or more.

In practice, in a LL(1) parsing table, only the right-hand sides of the rules are written. This notation is unambiguous as the corresponding left-hand side is the non-terminal that labels the corresponding line.

Definition 7.5 (LL(1) parsing table) *Let $G = (N, \Sigma, P, S)$ be a CFG. The* LL(1) parsing table *of G is the 2D table with lines labelled in N and columns labelled in $\Sigma \cup \{\#\}$ such that the cell (A, a) contains $R_{A,a}$, or more concisely the list of the right-hand sides of the rules of $R_{A,a}$.*

Example(s) 7.5 *Consider the grammar G from example 7.3. Then:*

- $R_{S,a} = \{S \to A\, B\, S\}$*;*
- $R_{S,c} = \{S \to c\}$*;*
- $R_{A,a} = \{A \to a\}$*;*
- $R_{B,a} = \{B \to A\, c\}$*;*
- $R_{B,b} = \{B \to b\}$*;*
- *for all other $(X, x) \in N \times (\Sigma \cup \{\#\})$, $R_{X,x} = \emptyset$.*

The LL(1) parsing table of G is shown in table 7.1. G is LL(1), which is reflected by the fact that each cell of this table is either empty or contains (the right-hand side of) only one rule.

Figure 7.3 illustrates the LL(1) analysis of $a\,a\,c\,c$; for each parsing state (α, u), the next rule to apply is the one corresponding in the LL(1) parsing table to the first symbol of α and u following their longest common prefix.

$$\begin{tikzcd}
(S,\ a\,a\,c\,c) \arrow[d, dash] \\
(\boldsymbol{A\,B\,S},\ a\,a\,c\,c) \arrow[d, dash] \\
(\boldsymbol{a}\,B\,S,\ a\,a\,c\,c) \arrow[d, dash] \\
(\boldsymbol{A\,c}\,S,\ a\,c\,c) \arrow[d, dash] \\
(\boldsymbol{a}\,c\,S,\ a\,c\,c) \arrow[d, dash] \\
(\boldsymbol{c},\ c)
\end{tikzcd}$$

Figure 7.3: The parsing states seen during an LL(1) recognition of $a\,a\,c\,c$. The symbols generated by the last rule applied are indicated in bold. Longest common prefixes are indicated in green.

	a	b	c	$\#$
S	$A\,B\,S$		c	
A	a			
B	$A\,c$	b		

Table 7.1: The LL(1) parsing table of the grammar of examples 7.3 and 7.5.

Example(s) 7.6 *Consider $G = (N, \Sigma, P, S)$ where:*

- $N = \{S, A, B\}$;
- $\Sigma = \{a, b, c\}$;
- $P = \left\{ \begin{array}{l} S \to A\,B\,S \mid c, \\ A \to a \mid B, \\ B \to b \mid \epsilon \end{array} \right\}$.

Then:

- $R_{S,a} = \{S \to A\,B\,S\}$;
- $R_{S,b} = \{S \to A\,B\,S\}$;
- $R_{S,c} = \{S \to A\,B\,S, S \to c\}$;
- $R_{A,a} = \{A \to a, A \to B\}$;
- $R_{A,b} = \{A \to B\}$;
- $R_{A,c} = \{A \to B\}$;
- $R_{B,a} = \{B \to \epsilon\}$;
- $R_{B,b} = \{B \to b, B \to \epsilon\}$;
- $R_{B,c} = \{B \to \epsilon\}$;
- *for all other* $(X, x) \in N \times (\Sigma \cup \{\#\})$, $R_{X,x} = \emptyset$.

	a	b	c	$\#$
S	$A\,B\,S$	$A\,B\,S$	$A\,B\,S$ c	
A	a B	B	B	
B	ϵ	ϵ b	ϵ	

Table 7.2: The LL(1) parsing table of the grammar of example 7.6.

The LL(1) parsing table of G is shown in table 7.2. G is not LL(1), which is reflected by the fact that some cell of this table contains (the right-hand side of) strictly more than one rule.

Exercise 7.2 *Use property 7.3 to write a deterministic recognition algorithm for LL(1) grammars that takes the LL(1) parsing table of the grammar considered as an argument.*

Remark 7.11 (Programming languages) *Programming languages are often designed to have an LL(1) grammar, or 'almost LL(1)'.*

Remark 7.12 (LL(k) grammar) *Very few CF languages have an LL(1) grammar. It is, however, possible to further extend the idea that by looking at the beginning of the remaining suffix during a top-down analysis, one can determine exactly which production rule to use next. For $k \in \mathbb{N}$, a grammar is LL(k) if its LL(k) parsing table contains at most one rule per cell, where the LL(k) parsing table is similar to the LL(1) parsing table but where the columns correspond to all possible sequences in $(\Sigma \cup \{\#\})^k$.*

Property 7.4 *Ambiguous CF languages are never LL(k) for any k, but there are even non-ambiguous grammars that are not LL(k) for any k.*

Example(s) 7.7 *Let $G = (\{S\}, \{0,1\}, \{S \to 0\,S\,0 \mid 1\,S\,1 \mid \epsilon\}, S)$. G generates the language of all palindromes on the alphabet $\{0,1\}$. G is unambiguous (any word in $\mathcal{L}(G)$ has exactly one derivation in G) but is non-deterministic, which means in particular that it is not LL(k) for any k.*

7.3 LR analysis

Remark 7.13 *For practical reasons, in this section we assume that the set of shift-reduce actions is extended with two new types of actions (which are not partial functions from states to states):*

- *the accept action indicates the success of the analysis and only possible if the stack only contains the axiom and the buffer is empty,*
- *the fail action indicates the failure of the analysis.*

Remark 7.14 *A deterministic shift-reduce parser can determine at each step, based on the content of the stack and the content of the buffer, which action to perform. Because such a parser always follows a single path in the state space, it can keep track of the current state with mutable structures (a mutable stack and a mutable buffer).*

Remark 7.15 (Python implementation)

Stack	Buffer	Action
ϵ	$a\,a\,b\,b\,a$	shift
a	$a\,b\,b\,a$	shift
$a\,a$	$b\,b\,a$	shift
$a\,a\,b$	$b\,a$	reduce($A \to b$)
$a\,a\,A$	$b\,a$	reduce($A \to a\,A$)
$a\,A$	$b\,a$	reduce($A \to a\,A$)
A	$b\,a$	shift
$A\,b$	a	shift
$A\,b\,a$	ϵ	reduce($B \to a$)
$A\,b\,B$	ϵ	reduce($B \to b\,B$)
$A\,B$	ϵ	reduce($S \to A\,B$)
S	ϵ	accept

Figure 7.4: A deterministic shift-reduce analysis of $a\,a\,b\,b\,a$.

- *A stack can be implement as a Python list. If* `stack` *is a Python list that represents a stack, the top of the stack can be accessed with* `stack[-1]`*, popping the stack can be done with* `stack.pop` *and pushing something on top of the stack can be done with* `stack.append`*.*
- *The kind of buffer needed for shift-reduce parsing (i.e. a buffer to which no element is ever added after its initialisation) can be implemented in different ways.*
  - *As a Python list, initialised 'backward' (i.e. with the last letter of the word at the first position of the list and the first letter of the word at the last position of the list). If* `buffer` *is such a Python list that represents a buffer, the next element of the buffer can be accessed with* `buffer[-1]` *and dequeueing the buffer can be done with* `buffer.pop`*.*
  - *As a string paired with an integer that encodes the position in the string that corresponds to the beginning of the buffer (this integer is initialised as 0). If* `(buffer_s, buffer_i)` *is such a pair that represents a buffer, the next element of the buffer can be accessed with* `buffer_s[buffer_i]` *and dequeueing the buffer can be done by incrementing* `buffer_i` *and then returning* `buffer_s[buffer_i - 1]`*.*

Example(s) 7.8 *Consider $G = (N, \Sigma, P, S)$ where:*

- $N = \{S, A, B\}$*;*
- $\Sigma = \{a, b\}$*;*
- $P = \left\{ \begin{array}{l} S \to A\,B, \\ A \to a\,A \mid b, \\ B \to b\,B \mid a \end{array} \right\}.$

A deterministic shift-reduce analysis of $a\,a\,b\,b\,a$ is given in figure 7.4.

Property 7.5 *Let $G = (N, \Sigma, P, S)$ be a CFG and $u \in \Sigma^\star$. At any step of a shift-reduce analysis of u, if the value of the stack is $\beta \in (N \cup \Sigma)^\star$ and the value of the buffer is $v \in \Sigma^\star$,* *then $\beta\, v \overset{\star}{\Rightarrow} u$. If fact, the value of the buffer is always a suffix of the input, so $u = u'\, v$ with some $u' \in (N \cup \Sigma)^\star$ s.t. $\beta \overset{\star}{\Rightarrow} u'$.*

Definition 7.6 (LR(0) context) *Let $G = (N, \Sigma, P, S)$ be a CFG and $A \to \alpha \in P$. The* LR(0) context *of $A \to \alpha$, written '$L_{A\to\alpha}$', is the set of stack values for which there exist a value of the buffer for which reducing $A \to \alpha$ is a correct action in a successful shift-reduce analysis:*

$$L_{A\to\alpha} = \{\beta \in (N \cup \Sigma)^\star \mid \exists \gamma \in (N \cup \Sigma)^\star,\ \beta = \gamma\, \alpha \text{ and } \exists v \in \Sigma^\star,\ S \overset{\star}{\Rightarrow} \gamma\, A\, v \Rightarrow \underbrace{\gamma\, \alpha}_{\beta}\, v\}$$

In this formula, $(\overbrace{\gamma\, \alpha}^{\beta}, v)$ can be interpreted as a parsing state reached right before performing reduce($A \to \alpha$) ($(\gamma\, A, v)$ is then the resulting parsing state).

Remark 7.16 *In definition 7.6, (β, v) represents a parsing state.*

Property 7.6 *If, during a shift-reduce analysis of u, the current value of the stack is not in the LR(0) context of some production rule $A \to \alpha$, then reducing the stack with $A \to \alpha$ cannot lead to a successful recognition.*

Remark 7.17 *If, during a shift-reduce analysis of u, the current state is (β, v) with $\beta = \gamma\, \alpha \in L_{A\to\alpha}$, then:*

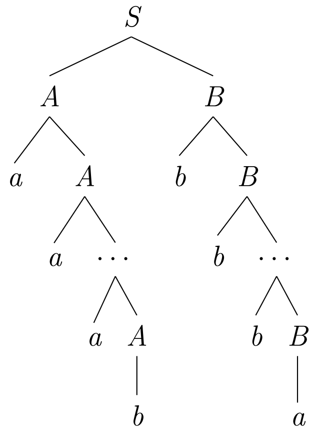
- *$\beta\, v \overset{\star}{\Rightarrow} u$, i.e. $\gamma\, \alpha\, v \overset{\star}{\Rightarrow} u$;*
- *$\exists v' \in \Sigma^\star,\ S \overset{\star}{\Rightarrow} \gamma\, A\, v' \Rightarrow \gamma\, \alpha\, v'$.*

It is not sure that reducing the stack with $A \to \alpha$ would lead to a recognition of u because it is unknown whether the current value of the buffer, v, is one of the v'·s that satisfy the existential quantifier in the second property.

Example(s) 7.9 *Consider $G = (N, \Sigma, P, S)$ where:*

- *$N = \{S, A, B\}$;*
- *$\Sigma = \{a, b\}$;*
- *$P = \left\{ \begin{array}{l} S \to A\, B, \\ A \to a\, A \mid b, \\ B \to b\, B \mid a \end{array} \right\}$.*

$\mathcal{L}(G) = \{a^n\, b\, b^m\, a \mid n, m \in \mathbb{N}\}$ and for all $n, m \in \mathbb{N}$, $a^n\, b\, b^m\, a$ has a single syntactic structure, of following form:



This helps us see that, for any $n, m \in \mathbb{N}$, the only possible shift-reduce analysis of $a^n\, b\, b^m\, a$ is the one in figure 7.5. This shows that:

- $L_{A \to b} = \bigcup_{n,m \in \mathbb{N}} \{a^n\, b\} = \mathcal{L}(a^*\, b)$
- $L_{A \to a\,A} = \bigcup_{n,m \in \mathbb{N}} \{a^n\, A, a^{n-1}\, A, \ldots, a\, A\} = \mathcal{L}(a^*\, a\, A)$
- $L_{B \to a} = \bigcup_{n,m \in \mathbb{N}} \{A\, b^m\, a\} = \mathcal{L}(A\, b^*\, a)$
- $L_{B \to b\,B} = \bigcup_{n,m \in \mathbb{N}} \{A\, b^m\, B, A\, b^{m-1}\, B, \ldots, A\, b\, B\} = \mathcal{L}(A\, b^*\, b\, B)$
- $L_{S \to A\,B} = \bigcup_{n,m \in \mathbb{N}} \{A\, B\} = \mathcal{L}(A\, B)$

In this case, the LR(0) contexts have two interesting properties:

- *they are all disjoint from each other,*
- *and no element of an LR(0) context is a proper prefix of an element of the same or another LR(0) context.*

A consequence of this is that a deterministic analysis is possible. At each step, if the current state is not $([S], \epsilon)$,

- *if the current value of the stack is in the LR(0) context of a rule $X \to \alpha$, then the next action should be reduce$(X \to \alpha)$,*
- *otherwise, the next action should be either shift or fail depending on whether the buffer is empty or not.*

If two different LR(0) contexts had an element β in common, then β would be a stack value for which there would be a choice between two different reduce·s. And if an element β of an LR(0) context were a proper prefix of another element of an LR(0) context, then β would be a stack value for which there would be a choice between a reduce and a shift.

Theorem 7.1 *Let $G = (N, \Sigma, P, S)$ be a CFG and $A \to \alpha \in P$. $L_{A \to \alpha}$ is a regular language on $(N \cup \Sigma)$.*

Property 7.7 *Let $G = (N, \Sigma, P, S)$ be a CFG. Because for all $A \to \alpha \in P$, $L_{A \to \alpha}$ is regular, $\bigcup_{A \to \alpha \in P} L_{A \to \alpha}$ is regular too and can be recognised by a deterministic finite-state automaton (DFA) on $(N \cup \Sigma)$. (Here and below, 'DFA' is used for complete and incomplete DFAs alike.)*

For some CFGs, this language can be recognised not only by a DFA, but by a DFA $(Q, (N \cup \Sigma), \delta, q_0, F)$ such that:

1. *each $A \to \alpha \in P$ is associated with a distinct final state $q_{A \to \alpha} \in F$ in such a way that the DFA $(Q, (N \cup \Sigma), \delta, q_0, \{q_A \to \alpha\})$ recognises $L_{A \to \alpha}$.*
2. *there is no transition going out of any final state.*

When this is the case, the DFA can be used for deterministic shift-reduce parsing.

Remark 7.18

Stack	Buffer	Action
ϵ	$a^n\, b\, b^m\, a$	shift
a	$a^{n-1}\, b\, b^m\, a$	shift
a^2	$a^{n-2}\, b\, b^m\, a$	shift
$\dots$	$\dots$	$\dots$
a^n	$b\, b^m\, a$	shift
$a^n\, b$	$b^m\, a$	reduce($A \to b$)
$a^n\, A$	$b^m\, a$	reduce($A \to a\, A$)
$a^{n-1}\, A$	$b^m\, a$	reduce($A \to a\, A$)
$\dots$	$\dots$	$\dots$
$a\, A$	$b^m\, a$	reduce($A \to a\, A$)
A	$b^m\, a$	shift
$A\, b$	$b^{m-1}\, a$	shift
$A\, b^2$	$b^{m-2}\, a$	shift
$\dots$	$\dots$	$\dots$
$A\, b^m$	a	shift
$A\, b^m\, a$	ϵ	reduce($B \to a$)
$A\, b^m\, B$	ϵ	reduce($B \to b\, B$)
$A\, b^{m-1}\, B$	ϵ	reduce($B \to b\, B$)
$\dots$	$\dots$	$\dots$
$A\, b\, B$	ϵ	reduce($B \to b\, B$)
$A\, B$	ϵ	reduce($S \to A\, B$)
S	ϵ	accept

Figure 7.5: A deterministic shift-reduce analysis of $a^n\, b\, b^m\, a$.

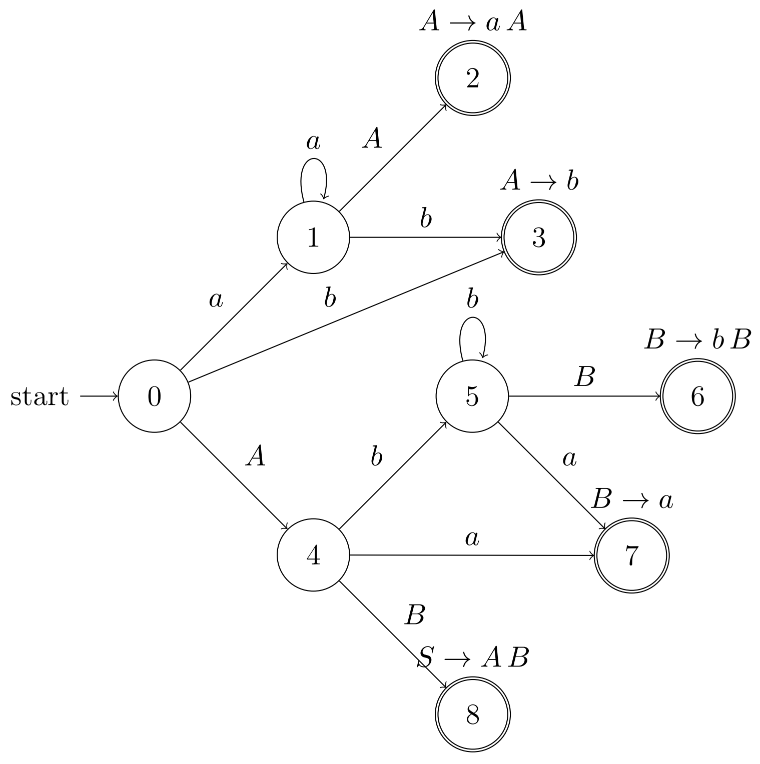


Figure 7.6: A DFA that recognises the LR(0) contexts of some CFG. The composition of the $F_{A\to\alpha}$ sets is indicated using annotations above the final states; $F_{A\to a\,A} = \{2\}$, $F_{A\to b} = \{3\}$, $F_{B\to b\,B} = \{6\}$, $F_{B\to a} = \{7\}$, $F_{S\to A\,B} = \{8\}$. One can remark that in this DFA, there is no transition going out of any final state.

- *The first of the two properties mentioned in property 7.7 entails that the LR(0) contexts are non-overlapping.*

- *The second property entails that no LR(0) context contains a sequence that is a proper prefix of a sequence from the same or another LR(0) context.*

Together, they entail that for any two distinct rules $A \to \alpha$ and $B \to \beta$, there is no $\gamma \in L_{A\to\alpha}$ and $\gamma' \in (N \cup \Sigma)^\star$ such that $\gamma\,\gamma' \in L_{B\to\beta}$. This means that if during a shift-reduce analysis the stack value is $\gamma \in L_{A\to\alpha}$, the only viable option is to reduce the stack with $A \to \alpha$; in particular, no amount of shifting could lead the stack to be in an LR(0) context again.

Example(s) 7.10 *The LR(0) contexts of the grammar from example 7.9 can be recognised by the DFA in figure 7.6, which satisfies the properties mentioned in property 7.7.*

Remark 7.19 *In practice, one does not first determine the LR(0) contexts and then try to build a DFA with the properties mentioned above. We will study below an algorithm that can be used to build such a DFA when possible. But first, let's see how to use such a DFA.*

Example(s) 7.11 (Parsing with a DFA) *Consider $G = (N, \Sigma, P, S)$ where:*

- *$N = \{S, A, B\}$;*

- *$\Sigma = \{a, b\}$;*

- $P = \left\{ \begin{array}{l} S \to A\,B, \\ A \to a\,A \mid b, \\ B \to b\,B \mid a \end{array} \right\}.$

Let us analyse $a\,a\,b\,b\,a$ using the DFA in figure 7.6.

1. *The initial parsing state is $(\epsilon, a\,a\,b\,b\,a)$.* ***We will ensure that at all time, the current state of the DFA is the one we get to by reading the content of the stack.*** *In particular the current stack is empty and so we set $q = 0$, the initial state of the DFA. Each time we shift a terminal symbol x from the buffer, we will feed it to the DFA (in addition to pushing it onto the stack).*
2. *$q = 0$ is not a final state, which means that the empty string is not in the LR(0) contexts of any rule, which means that there is no reduce to consider now (as the current stack is indeed empty). However, from this state it is possible to reach a final state following transitions labelled with terminal symbols, which means that by shifting it may be possible to reach a final state, at which point reducing will be an option.*
3. *Let us shift. The parsing state is now $(a, a\,b\,b\,a)$. Let $q = \delta(q, a) = \delta(0, a) = 1$. $q = 1$ is not a final state (reducing is not an option) but it is still possible to reach a final state following transitions labelled with terminal symbols (shifting is an option).*
4. *Let us shift. The parsing state is now $(a\,a, b\,b\,a)$. Let $q = \delta(q, a) = \delta(1, a) = 1$. $q = 1$ is not a final state (reducing is not an option) but it is still possible to reach a final state following transitions labelled with terminal symbols (shifting is an option).*
5. *Let us shift. The parsing state is now $(a\,a\,b, b\,a)$. Let $q = \delta(q, b) = \delta(1, b) = 3$. $q = 3$ is a final state associated with $A \to b$, which means that the sequence of symbols we have fed the DFA so far, i.e. the content of the stack, is in the LR(0) context of this rule. Reducing the stack with $A \to b$ is the only viable option; in particular, there is no transition going out of $q = 3$, which indicates that shifting is not an option.*
6. *Let us reduce with $A \to b$. The parsing state is now $(a\,a\,A, b\,a)$. We need to feed the content of the stack to the DFA starting back from the initial state: $q = \delta^*(0, a\,a\,A) = 2$. $q = 2$ is a final state associated with $A \to a\,A$ and there is no transition going out of $q = 2$ (reducing the stack with $A \to a\,A$ is the only viable option).*
7. *Let us reduce with $A \to a\,A$. The parsing state is now $(a\,A, b\,a)$. Let $q = \delta^*(0, a\,A) = 2$. $q = 2$ is a final state associated with $A \to a\,A$ and there is no transition going out of $q = 2$ (reducing the stack with $A \to a\,A$ is the only viable option).*
8. *Let us reduce with $A \to a\,A$. The parsing state is now $(A, b\,a)$. Let $q = \delta^*(0, A) = 4$. $q = 4$ is not a final state (reducing is not an option) but it is still possible to reach a final state following transitions labelled with terminal symbols (shifting is an option).*
9. *Let us shift. The parsing state is now $(A\,b, a)$. Let $q = \delta(q, b) = \delta(4, b) = 5$. $q = 5$ is not a final state (reducing is not an option) but it is still possible to reach a final state following transitions labelled with terminal symbols (shifting is an option).*
10. *Let us shift. The parsing state is now $(A\,b\,a, \epsilon)$. Let $q = \delta(q, a) = \delta(5, a) = 7$. $q = 7$ is a final state associated with $B \to a$ and there is no transition going out of $q = 7$ (reducing the stack with $B \to a$ is the only viable option).*
11. *Let us reduce with $B \to b$. The parsing state is now $(A\,b\,B, \epsilon)$. Let $q = \delta^*(0, A\,b\,B) = 6$. $q = 6$ is a final state associated with $B \to b\,B$ and there is no transition going out of $q = 6$ (reducing the stack with $B \to b\,B$ is the only viable option).*

12. *Let us reduce with $B \to b\,B$. The parsing state is now $(A\,B, \epsilon)$. Let $q = \delta^*(0, A\,B) = 8$.* *$q = 8$ is a final state associated with $S \to A\,B$ and there is no transition going out of $q = 8$ (reducing the stack with $S \to A\,B$ is the only viable option).*

13. *Let us reduce with $S \to A\,B$. The parsing state is now (S, ϵ). This configuration indicates a successful recognition.*

This analysis corresponds to figure 7.4 above.

7.3.1 Construction of the LR(0) automaton

Definition 7.7 (Dotted rule) *Let $G = (N, \Sigma, P, S)$ be a CFG. A* dotted rule *of G is a triple $(A, \alpha, \alpha') \in N \times (N \cup \Sigma)^\star \times (N \cup \Sigma)^\star$ such that $A \to \alpha\,\alpha' \in P$.*

Notation 7.1 *A dotted rule (A, α, α') is usually written '$A \to \alpha \bullet \alpha'$'.*

Example(s) 7.12 *If a grammar contains the rule $A \to a\,B\,c$, its dotted rules include $A \to \bullet a\,B\,c$, $A \to a \bullet B\,c$, $A \to a\,B \bullet c$ and $A \to a\,B\,c\bullet$.*

Definition 7.8 (Initial rule) *A dotted rule $A \to \alpha \bullet \alpha'$ is* initial *iff $\alpha = \epsilon$.*

Definition 7.9 (Completed rule) *A dotted rule $A \to \alpha \bullet \alpha'$ is* completed *iff $\alpha' = \epsilon$.*

Definition 7.10 (LR(0) NFA and LR(0) automaton) *Let $G = (N, \Sigma, P, S)$ be a CFG. Let S' and $\#$ be two new fresh symbols (i.e. not in $N \cup \Sigma$). Let $G' = (N', \Sigma', P', S') = ((N \cup \{S'\}), (\Sigma \cup \{\#\}), (P \cup \{S' \to S\,\#\}), S')$, the grammar obtained by adding the production rule $S' \to S\,\#$ to G. Let $\mathcal{A}_N = (Q, (N' \cup \Sigma'), \delta, q_0, F)$ the NFA with ϵ-production with:*

- *$Q = \{A \to \alpha \bullet \alpha' \mid A \to \alpha\,\alpha' \in P'\}$, the set of all dotted rules of G';*
- *the transition function δ is defined by (in what follows, $x \in N' \cup \Sigma'$, $X \in N'$, $A \in N'$, $\alpha, \beta \in (N' \cup \Sigma')^\star$):*
  - *$\delta(A \to \alpha \bullet x\,\beta, x) = \{A \to \alpha\,x \bullet \beta\}$;*
  - *$\delta(A \to \alpha \bullet X\,\beta, \epsilon) = \{X \to \bullet\gamma \mid \forall (X \to \gamma) \in P'\}$;*
  - *$\delta(A \to \alpha\bullet, x) = \delta(A \to \alpha\bullet, \epsilon) = \emptyset$;*
- *$q_0 = S' \to \bullet S\,\#$;*
- *$F = \{A \to \alpha \bullet \ \mid A \to \alpha \in P'\}$.*

$\mathcal{A}_N$ is the LR(0) NFA *of G. The determinised version of $\mathcal{A}_N$, $\mathcal{A}$, is the* LR(0) automaton *of G.*

Example(s) 7.13 *Consider $G = (N, \Sigma, P, S)$ where:*

- *$N = \{S, A, B\}$;*
- *$\Sigma = \{a, b\}$;*
- $P = \left\{ \begin{array}{l} S \to A\,B, \\ A \to a\,A \mid b, \\ B \to b\,B \mid a \end{array} \right\}.$

Let $G' = ((N \cup \{S'\}), (\Sigma \cup \{\#\}), (P \cup \{S' \to S\,\#\}), S')$. The LR(0) NFA of G is shown in figure 7.7. This NFA is determinised according to table 7.3 and the resulting LR(0) automaton is shown in figure 7.8.

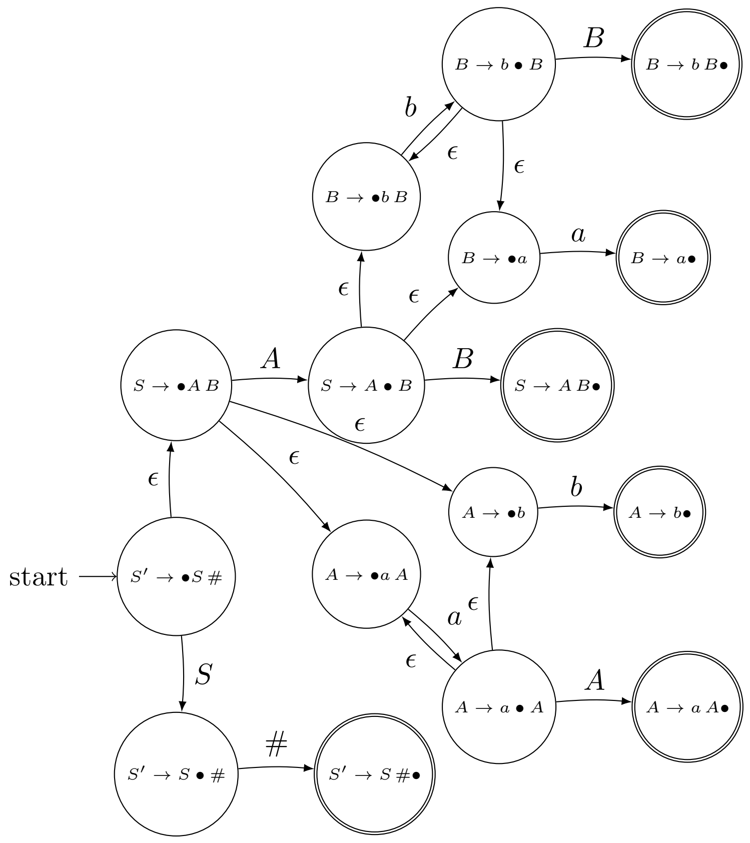


Figure 7.7: The LR(0) NFA of the grammar of example 7.13.

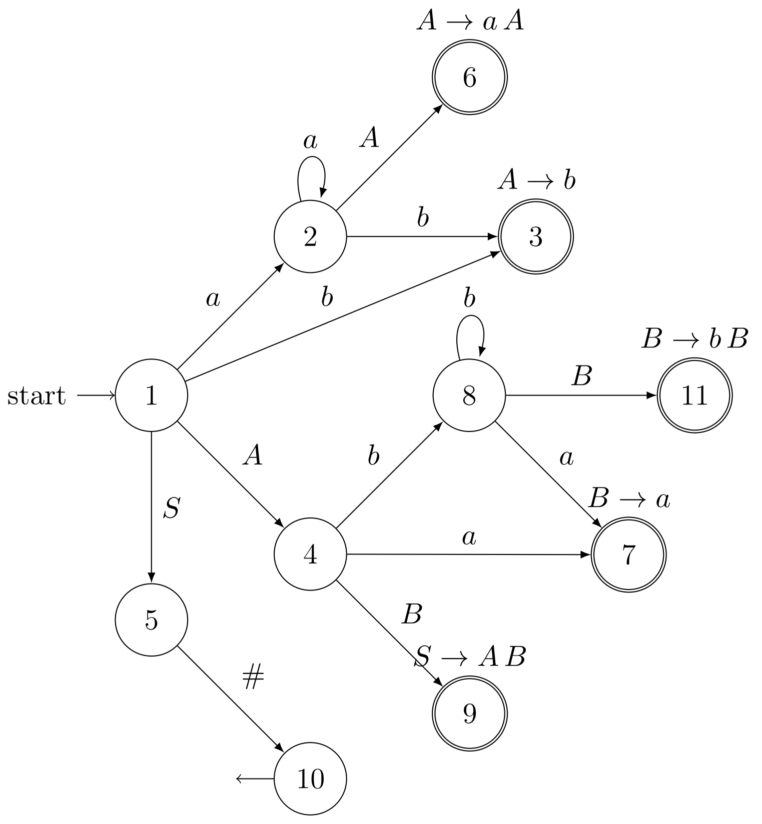


Figure 7.8: The LR(0) automaton of the grammar of example 7.13.

	a	b	A	B	S	$\#$
(1) $= \{S' \to \bullet S\#,$ $S \to \bullet AB,$ $A \to \bullet aA,$ $A \to \bullet b\}$	$(2) = \{A \to a \bullet A,$ $A \to \bullet aA,$ $A \to \bullet b\}$	$(3) = \{A \to b\bullet\}$	$(4) = \{S \to A \bullet B,$ $B \to \bullet bB,$ $B \to \bullet a\}$	$\emptyset$	$(5) = \{S' \to S \bullet \#\}$	$\emptyset$
(2)	(2)	(3)	$(6) = \{A \to aA\bullet\}$	$\emptyset$	$\emptyset$	$\emptyset$
(3)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
(4)	$(7) = \{B \to a\bullet\}$	$(8) = \{B \to b \bullet B,$ $B \to \bullet bB,$ $B \to \bullet a\}$	$\emptyset$	$(9) = \{S \to AB\bullet\}$	$\emptyset$	$\emptyset$
(5)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$(10) = \{S' \to S\#\bullet\}$
(6)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
(7)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
(8)	(7)	(8)	$\emptyset$	$(11) = \{B \to bB\bullet\}$	$\emptyset$	$\emptyset$
(9)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
(10)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
(11)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 7.3: Determinisation table of the LR(0) NFA of the grammar of example 7.13. The initial state is indicated in italics and the terminal states are indicated in bold.

Theorem 7.2 *Let $G = (N, \Sigma, P, S)$ be a CFG, $\mathcal{A}_N$ its LR(0) NFA and $\mathcal{A}$ its LR(0) automaton. Remember that:*

- *a state of $\mathcal{A}$ corresponds to a set of states of $\mathcal{A}_N$, i.e. a set of dotted rule of G';*
- *a final state of $\mathcal{A}$ is a set that contains at least one final state of $\mathcal{A}_N$, i.e. at least one completed rule.*

For all $A \to \alpha \in P$, the language recognised through the states of $\mathcal{A}$ that contain $A \to \alpha\bullet$ is $L_{A\to\alpha}$.

Remark 7.20 *Theorem 7.2 proves theorem 7.1. (Note that we have not proved theorem 7.2, however.)*

Definition 7.11 (LR(0) grammar) *Let $G = (N, \Sigma, P, S)$ be a CFG and $\mathcal{A}$ its LR(0) automaton. If each final state of $\mathcal{A}$ contains exactly one completed rule and if there is no transition going out of any of them, then $\mathcal{A}$ satisfies the properties mentioned in property 7.7 and can be used for deterministic shift-reduce parsing. In this case, G is an* LR(0) grammar*.*

7.3.2 Parsing with the LR(0) automaton

- Given an LR(0) grammar $G = (N, \Sigma, P, S)$, its LR(0) automaton $\mathcal{A} = (Q, (N' \cup \Sigma'), \delta, q_0, F)$ and rule, the function that sends any final state of $\mathcal{A}$ to the unique completed rule it contains, algorithm 14 performs deterministic shift-reduce recognition.

Remark 7.21 *Algorithm 14 only goes through (at most) a linear number of parsing states, but its complexity is not linear. Indeed, after each reduce, the full content of the stack is fed to the DFA, and the stack can be as long as the sequence to recognise itself. It can be shown that the complexity of this algorithm is quadratic ($O(n^2)$, where n is the length of the input sequence).*

Remark 7.22 *To ensure that at all time, the current state of the DFA is the one we get to by reading the content of the stack, there is an alternative to reinitialising the DFA and feeding it the full content of the stack. Indeed, if the content of the stack is $\gamma\,\alpha \in L_{A\to\alpha}$, the target state after reducing it with $A \to \alpha$ (i.e. $\delta^*(q_0, \gamma\, A)$) is the one obtained by feeding A to the DFA from the state q that was reached when the content of the stack was γ (i.e. $q = \delta^*(q_0, \gamma)$). The alternative solution is then to use an additional stack to store DFA states. Doing so ensures that the complexity of each reduce operation is bound by a constant, which leads to an overall linear complexity. This technique is implemented in algorithm 15.*

Example(s) 7.14 *Consider $G = (N, \Sigma, P, S)$ where:*

- *$N = \{S, A, B\}$;*
- *$\Sigma = \{a, b\}$;*
- $P = \left\{ \begin{array}{l} S \to A\,B, \\ A \to a\,A \mid b, \\ B \to b\,B \mid a \end{array} \right\}$.

A successful LR(0) analysis of $a\,a\,b\,b\,a$ is given in figure 7.9.

```
// Input:  $u \in \Sigma^\star$
// Output:  True if $S \overset{\star}{\underset{G}{\Rightarrow}} u$, False otherwise
// This algorithm presupposes knowledge of the LR(0) automaton (through
   $q_0$, $\delta$, $F$ and rule).
Function LR0-shift-reduce($u$): bool
    stack := init_stack([]);
    buffer := init_buffer([$u_1, u_2, \cdots, u_{|u|}, \#$]);
    $q := q_0$;                                   // Initialisation of the DFA.
    while True do
        // Success?
        if len(stack) $\neq 0$ and stack.last() $= \#$ then return True;
        // Reduce?
        if $q \in F$ then
            $(A \to \alpha)$ := rule($q$);        // $A \to \alpha$ is the rule associated with $q$.
            foreach $a$ in $\alpha$ do stack.pop();
            stack.push($A$);
            $q := q_0$;                           // Reinitialisation of the DFA.
            foreach $x$ in stack do
                if $\delta(q, x)$ is not defined then return False;
                $q := \delta(q, x)$;
        // Shift?
        else if (len(buffer) $\neq 0$) then
            $a$ := buffer.dequeue();
            stack.push($a$);
            if $\delta(q, a)$ is not defined then return False;
            $q := \delta(q, a)$;
        // Failure
        else
            return False;
```

Algorithm 14: LR(0) recognition in quadratic time. rule is the function from F to P such that rule($q_{A\to\alpha}$) $= A \to \alpha$. stack.last returns the element on top of the stack; stack.pop removes the element on top of the stack and returns it; stack.push adds an element on top of the stack; buffer.next returns the next element in the buffer; buffer.dequeue removes the next element in the buffer and returns it.

```
// Input:  $u \in \Sigma^\star$
// Output:  True if $S \overset{\star}{\underset{G}{\Rightarrow}} u$, False otherwise
// This algorithm presupposes knowledge of the LR(0) automaton (through
   $q_0$, $\delta$, $F$ and rule).
Function LR0-shift-reduce-linear($u$): bool
    stack := init_stack([]);
    state_stack := init_stack([$q_0$]);
    buffer := init_buffer([$u_1, u_2, \cdots, u_{|u|}$, #]);
    $q := q_0$;                    // q will always be equal to state_stack.last().
    while True do
        // Success?
        if len(stack) $\neq 0$ and stack.last() = # then return True;
        // Reduce?
        if $q \in F$ then
            $(A \to \alpha)$ := rule($q$);          // $A \to \alpha$ is the rule associated with $q$.
            foreach $a$ in $\alpha$ do
                stack.pop();
                state_stack.pop();
            $q$ = state_stack.last();                               // Top of the stack
            if $\delta(q, A)$ is not defined then return False;
            stack.push($A$);
            $q := \delta(q, A)$;
            state_stack.push($q$);
        // Shift?
        else if (len(buffer) $\neq 0$) then
            $a$ := buffer.dequeue();
            stack.push($a$);
            if $\delta(q, a)$ is not defined then return False;
            $q := \delta(q, a)$;
            state_stack.push($q$);
        // Failure
        else
            return False;
```

Algorithm 15: LR(0) recognition in linear time. rule is the function from F to P such that rule($q_{A\to\alpha}$) $= A \to \alpha$. stack.last returns the element on top of the stack; stack.pop removes the element on top of the stack and returns it; stack.push adds an element on top of the stack; buffer.next returns the next element in the buffer; buffer.dequeue removes the next element in the buffer and returns it.

Stack	Buffer	Action
$[(@,1)]$	$a\,a\,b\,b\,a\,\#$	shift
$[(@,1),(a,2)]$	$a\,b\,b\,a\,\#$	shift
$[(@,1),(a,2),(a,2)]$	$b\,b\,a\,\#$	shift
$[(@,1),(a,2),(a,2),(b,3)]$	$b\,a\,\#$	reduce($A \to b$)
$[(@,1),(a,2),(a,2),(A,6)]$	$b\,a\,\#$	reduce($A \to a\,A$)
$[(@,1),(a,2),(A,6)]$	$b\,a\,\#$	reduce($A \to a\,A$)
$[(@,1),(A,4)]$	$b\,a\,\#$	shift
$[(@,1),(A,4),(b,8)]$	$a\,\#$	shift
$[(@,1),(A,4),(b,8),(a,7)]$	$\#$	reduce($B \to a$)
$[(@,1),(A,4),(b,8),(B,11)]$	$\#$	reduce($B \to b\,B$)
$[(@,1),(A,4),(B,8)]$	$\#$	reduce($S \to A\,B$)
$[(@,1),(S,5)]$	$\#$	shift
$[(@,1),(S,5),(\#,10)]$	ϵ	accept

Figure 7.9: Deterministic shift-reduce analysis of $a\,a\,b\,b\,a$ with the LR(0) automaton of the grammar of example 7.14.

Exercise 7.3 *Assume two types/classes to represent trees:*

- `Leaf`*: given a symbol* `a`*,* `Leaf(a)` *refers to a tree composed of a single node labelled* `a`*;*
- `InternalNode`*: given a symbol* `a` *and an arbitrary number of trees* `t1`*,* `t2`*, ...,* `tn` *(*`Leaf`*·s or* `InternalNode`*·s),* `InternalNode(a, t1, t2, ..., tn)` *refers to a tree composed of a node labelled* `a` *with* `t1`*,* `t2`*, ...,* `tn` *as children (in this order);*

Both of these classes have a property `label` *(equals to the symbol* `a` *used for the definition of the tree).*

Modify algorithm 14 or algorithm 15 so that the algorithm outputs the derivation tree of the input word in case of a successful analysis, and `None` *otherwise.*

Definition 7.12 (LR(0) parsing table) *In practice, an LR(0) automaton $(Q,(N' \cup \Sigma'),\delta,q_0,F)$ is often represented as an* LR(0) parsing table*. This table indicates not only the transition function but also, for each state, which is the only action that can lead a successful analysis. The transitions are sometimes grouped according to the kind of symbol that labels them: a* shift *is labelled by a letter (from Σ') while a* goto *is labelled by a non-terminal (from N').*

Example(s) 7.15 *The LR(0) parsing table that represents the automaton of figure 7.8 is shown in figure 7.10*

Remark 7.23 (LR(k) grammar) *Very few CF languages have an LR(0) grammar. It is, however, possible to extend the idea that by looking at the stack one can exclude some actions by also considering the beginning of the buffer. For $k \in \mathbb{N}$, an LR(k) automaton is built from an NFA the states of which are not simply dotted rules but pairs of a dotted rule and a k-length string that intuitively represents the k next symbols in the buffer. When an LR(k) automaton satisfies some specific properties, it is then possible to use it to perform deterministic shift-reduce analysis with an algorithm very similar to algorithm 14.*

State	Action	Shift			Goto		
		a	b	#	A	B	S
1	shift	2	3		4		5
2	shift	2	3		6		
3	reduce($A \to b$)						
4	shift	7	8			9	
5	shift			10			
6	reduce($A \to a\,A$)						
7	reduce($B \to a$)						
8	shift	7	8			11	
9	reduce($S \to A\,B$)						
10	accept						
11	reduce($B \to b\,B$)						

Figure 7.10: The parsing table that represents the LR(0) automaton of figure 7.8.

7.4 Deterministic CFG

- Deterministic CFG are the grammars derived from deterministic push-down automata.
- Deterministic CFG can be parsed deterministically.
- Ambiguous CFG are never deterministic.
- Some non-ambiguous CFG are not deterministic.

  Ex: $G = (\{S\}, \{0, 1\}, \{S \to 0\,S\,0 \mid 1\,S\,1 \mid \epsilon\}, S)$ is non-ambiguous and not deterministic. $\mathcal{L}(G)$ is the set of all palindromes of even length (on $\{0, 1\}$).

Vocabulary

a parsing table:	*une table d'analyse*
a stack:	*une pile*
a buffer:	*un tampon*
a dotted rule:	*une règle pointée*

Chapter 8

Chart parsing

Remark 8.1 (Changing paradigm) *The algorithms that we have studied so far work by exploring the search graph, trying to connect the axiom and the input word. If there are nodes in the graph from which multiple directions must be explored in order to ensure the correctness of the analysis, one faces a problem of* combinatorial explosion. *Indeed, consider one of these nodes: multiple directions must be explored, and each of these directions might lead to another node at which multiple directions must be explored, and so on. It is easy to update most algorithms in such a way that the same node is never visited more than once, but the root of the problem is that the number of nodes that have to be visited can grow exponentially with the length of the input word. This is why the algorithms studied so far have an exponential worst-case complexity (when the set of CFGs considered is not restricted to specific kinds of deterministic grammars).*

Natural language is intrinsically ambiguous and as such cannot be analysed deterministically. In this situation, one should change paradigm and stop trying to explicitly explore the search graph node after node.

Remark 8.2 *An ambiguous context-free language L is, by definition, a context-free language such that for any CFG G, if $\mathcal{L}(G) = L$, then G is ambiguous. In the strict meaning of the term, parsing consists in determining all of the syntactic structures of the input for the grammar considered. Parsing with an ambiguous grammar cannot then be deterministic, as for any ambiguous input, at least one choice between two valid options arises. Still, parsing in the weaker sense of determining one of the syntactic structures of the input (if at least one exists) can sometimes be done deterministically. (Parsing in this sense is very similar to recognising.)*

Consider for example $G = (\{Int\}, \Sigma, P, Int)$ where:

- $\Sigma = \{0, 1, \cdots, 9, +, -\}$;
- $P = \{Int \rightarrow Int + Int \mid Int - Int \mid 0 \mid 1 \mid \cdots \mid 9\}$.

This grammar is ambiguous (see chapter 4) but deterministic shift-reduce parsing (in the weaker sense) is possible by always reducing when possible (i.e. even when a shift would still lead to a successful analysis).

In this chapter (as in the previous ones), however, we focus on parsing in the strict sense of the term.

Definition 8.1 (Structure of a set of leaves in a tree) *Given a labelled tree $T = (V, E, f)$ (f is the labelling function that sends each node to its label) and L a set of leaves of T, the* structure *of L in T is the labelled forest $F_L = (V_L, E_L, f_L)$ where*

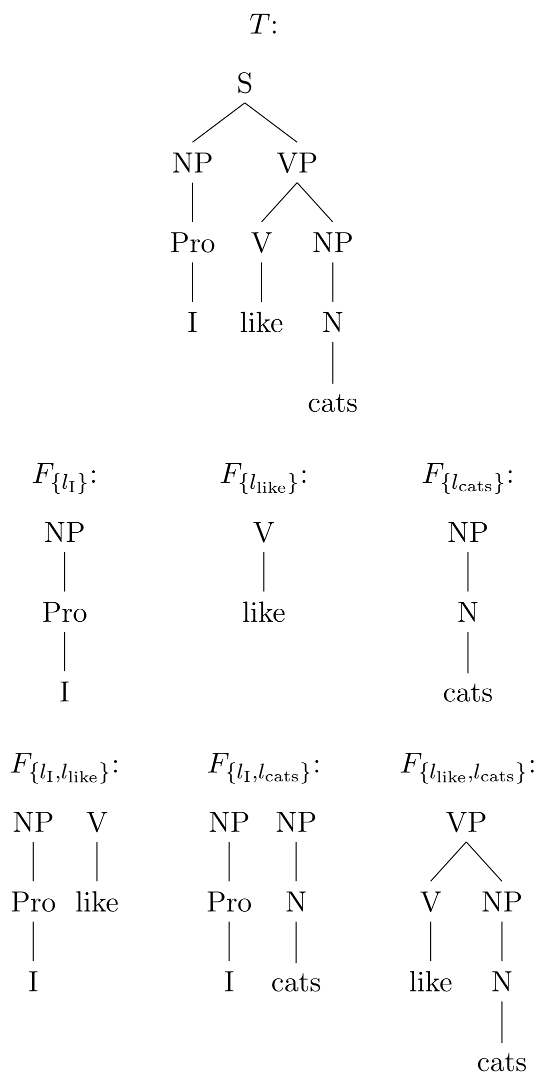

T:
S
NP
VP
Pro
V
NP
I
like
N
cats
$F_{\{l_{\mathrm{I}}\}}$:
$F_{\{l_{\mathrm{like}}\}}$:
$F_{\{l_{\mathrm{cats}}\}}$:
$F_{\{l_{\mathrm{I}},l_{\mathrm{like}}\}}$:
$F_{\{l_{\mathrm{I}},l_{\mathrm{cats}}\}}$:
$F_{\{l_{\mathrm{like}},l_{\mathrm{cats}}\}}$:

Figure 8.1: A syntactic tree T and the structure of its six non-empty proper subsets of leaves; four of them are trees and the other two are proper forests. l_{I}, l_{like}, and l_{cats} are the leaves of T labelled *I*, *like* and *cats* respectively.

- $V_L = \{v \in V \mid leaves(v) \subseteq L\}$ *is the set of nodes of* V *the leaves of which are all in* L*;*
- $E_L = \{(v_1, v_2) \in E \mid (v_1, v_2) \in {V_L}^2\}$ *is the restriction of* E *to* V_L*;*
- f_L *is the restriction of* f *to* V_L*.*

Example(s) 8.1 *Let* T *be as depicted in figure 8.1, and* l_I*,* l_{like}*, and* l_{cats} *be the leaves of* T *labelled* I, like *and* cats *respectively.*

$F_{\{l_I,l_{like},l_{cats}\}}$*, the structure of the set of all three leaves in* T *is* T *itself.* $F_\emptyset$*, the structure of the empty set of leaves in* T *is the empty graph. The structure of each of the six other subsets of leaves in* T *are shown in figure 8.1.*

Definition 8.2 (Local ambiguity) *Given a sequence* $u \in \Sigma^\star$*,* u *is* locally ambiguous *iff there exist* $w, w' \in \Sigma^\star$ *such that* u *is a subword of both* w *and* w'*, and there exist a derivation tree* T *of* w *and a derivation tree* T' *of* w' *such that the structures of* u *in* T *and* T' *are distinct.*

Example(s) 8.2

1. *In English,* Sabine saw a cat *is locally ambiguous. For example, this span forms an S constituent in (1a) (i.e. its structure in the corresponding tree is a tree rooted in a S node) and does not form a constituent in (1b) (i.e. its structure in the corresponding tree is not a single tree but a proper forest).*

   (1) a. *(Sabine saw a cat)$_S$.*
   b. *(A friend of Sabine)$_{NP}$ saw a cat.*

2. *In English,* saw a kid *is locally ambiguous. For example, this span forms a VP constituent in (2a) and does not form a constituent in (2b).*

   (2) a. *I (saw a kid)$_{VP}$ with my glasses.*
   b. *I saw (a kid with my glasses)$_{NP}$.*

3. *In English,* her duck *is locally ambiguous. For example, this span forms an NIC (for 'naked infinitive clause') constituent in (3a) and forms a VP constituent in (3b).*

   (3) a. *I saw (her duck$_V$)$_{NIC}$.*
   b. *I saw (her duck$_N$)$_{VP}$.*

Remark 8.3 *For a locally ambiguous u, even though u is associated with more than one structure, it may still appear in sentences that are not ambiguous. These sentences are not ambiguous because the ambiguity in u is resolved thanks to information outside of u, hence the name 'local ambiguity'.*

Definition 8.3 (Global ambiguity) *Given a word $w \in \Sigma^\star$ and a subword u of w, u is* globally ambiguous *in w iff there exist two derivation trees T_1 and T_2 of w such that the structures of u in T_1 and T_2 are distinct.*

Example(s) 8.3

1. *In English,* saw a kid *is globally ambiguous in* I saw a kid with my glasses. *For example, this span can either form a VP constituent or not form a constituent.*

2. *In English,* her duck *is globally ambiguous in* I saw her duck. *For example, this span can either form an NIC constituent or a VP constituent.*

Remark 8.4 *Global ambiguity entails local ambiguity in the sense that if u is globally ambiguous in w, then u is locally ambiguous. However, local ambiguity does not entail global ambiguity; for instance,* Sabine saw a cat *is locally ambiguous, but is not globally ambiguous in all sentences in which it appears, e.g.* A friend of Sabine saw a cat.

Definition 8.4 (Independence of ambiguity) *Given a word $w \in \Sigma^\star$ and two subwords u and u' of w both globally ambiguous in w, u and u' are* independently ambiguous *if for each possible structure F of u in w and each possible structure F' of u' in w, there exist a derivation tree of w such that the structures of u and u' in this tree are F and F' respectively.*

Example(s) 8.4

1. the man with a kid *and* a hat with a telescope *are independently ambiguous in* I saw the man with a kid wearing a hat with a telescope*; the* man with a kid *might or might not be a noun phrase depending on who is wearing the hat, and* a hat with a telescope *might or might not be a noun phrase depending on who/what is with the telescope, and all four combinations are possible.*

2. her duck *and* her fly *are not independently ambiguous in* I saw her duck and her fly *(provided that one does not accept heterogeneous coordinations); they are either both noun phrases or both naked infinitive clauses.*

3. *More trivially,* saw a kid *and* a kid with my glasses *are not independently ambiguous in* I saw a kid with my glasses*; either* saw a kid *is a verb phrase and then* a kid with my glasses *does not form a constituent, or* a kid with my glasses *in a noun phrase and then* saw a kid *does not form a constituent.*

Remark 8.5 *The exact number of nodes that must be explored for the analysis of a given word w depends on the particular algorithm used. However, this number is necessarily at least as large as the number of syntactic structures of w, which can be exponential in the length of w. Intuitively, the impact of non-global ambiguities on the number of nodes to consider can be very different from one algorithm to another (and from one ambiguous span to another) but naive algorithms can spend most of their time inferring the different structures of ambiguous spans.*

Remark 8.6 (Factoring the computation) *Imagine that one is trying to perform a top-down analysis of some word $w = w_1\, w_2 \cdots w_n$ and that one knows that $A\,B$ is a sentential form of the grammar: $S \overset{\star}{\Rightarrow} A\,B$, and the question is now whether $A\,B \overset{\star}{\Rightarrow} w$.*

If $A\,B \overset{\star}{\Rightarrow} w$, then there must be an $i \in [\![0, n]\!]$ such that $A \overset{\star}{\Rightarrow} w_{1:i}$ ($i = 0$ corresponds to $A \overset{\star}{\Rightarrow} \epsilon$) and $B \overset{\star}{\Rightarrow} w_{(i+1):n}$ ($i = n$ corresponds to $B \overset{\star}{\Rightarrow} \epsilon$). For any given i, because the grammar is context-free, trying to rewrite A as $w_{1:i}$ and trying to rewrite B as $w_{(i+1):n}$ are two subtasks that can be performed independently *from each other.*

Doing so is computationally appealing because, intuitively, if in the search graph there are c_1 nodes of interest in the search for $w_{1:i}$ from A and c_2 nodes of interest in the search for $w_{(i+1):n}$ from B, then there are $c_1 \times c_2$ nodes of interest in the search for w from $A\,B$ (while respecting the split at i), which is usually much bigger than $c_1 + c_2$, the number of nodes considered if the two subtasks are performed independently. Furthermore, the same factorisation process that reduces multiplications to sums can be recursively applied to each of the two subtasks in a way that, in the end, can turn an exponential number of nodes into a polynomial number of items. (This reasoning can be extended to $A_1\, A_2 \cdots A_k$ for any $k \geq 2$ and not just $A\,B$.)

At the beginning of this remark, we assumed a top-down analysis, but this aspect was not crucial; the same line of reasoning also applies to bottom-up approaches, for instance.

Example(s) 8.5 *Sentence (4), which describes a robot chasing a mouse, contains a syntactic ambiguity within its main noun phrase and another one within its main verb phrase. These two binary ambiguities lead to a four-way syntactic ambiguity overall. Similarly in (5), which describes a robot giving a mouse to a kid, three binary ambiguities lead to an eight-way syntactic ambiguity overall.*

(4) *(The robot wearing the hat that I made)$_{NP}$ (was chasing a mouse with a fork)$_{VP}$.*

a. *(The robot (wearing the hat that I made)$_{VP}$)$_{NP}$ (was chasing (a mouse with a fork)$_{NP}$)$_{VP}$.*

b. *(The robot (wearing the hat that I made)$_{VP}$)$_{NP}$ (was chasing (a mouse)$_{NP}$ with a fork)$_{VP}$.*

c. *(The robot (wearing the hat)$_{VP}$ that I made)$_{NP}$ (was chasing (a mouse with a fork)$_{NP}$)$_{VP}$.*

d. *(The robot (wearing the hat)$_{VP}$ that I made)$_{NP}$ (was chasing (a mouse)$_{NP}$ with a fork)$_{VP}$.*

(5) *(The robot wearing the hat that I made)$_{NP}$ (gave a mouse with a smile)$_{VP}$ (to the kid eating the cake that I had seen earlier)$_{PP}$.*

These binary ambiguities are independent of each other. As a consequence, a decision for one of them does not constrain the other ones. It would make sense to find a way to factor these ambiguities during parsing, for example by analysing non-overlapping spans independently of each other.

Intuition 8.1 (Chart parsing) *The idea behind chart parsing is to decompose the analysis of a word w into the independent analyses of spans of w. The result of these subanalyses are then combined to provide analyses of the whole w.*

To avoid redundant computation (i.e. analysing multiple times the same span), information about the analysis of each span $w_{i:j}$ (at least, the set of all non-terminals A such that $A \overset{\star}{\Rightarrow} w_{i:j}$) is stored in a data structure called a 'chart'. A chart is generally a bidimensional table each cell of which corresponds to a span. As we will see, the chart is not just used to avoid redundant computation, it is also a memory-efficient way to represent a possibly very large set of derivation trees in a compact way.

Remark 8.7 *The decomposition of a problem into overlapping subproblems with intermediary results stored in a data structure so as to prevent redundant computation is a characteristic feature of* dynamic programming.[1]

- We are about to study two famous algorithms: the CYK algorithm and the Earley parser.
- The CYK algorithm is a bottom-up algorithm that works only for grammars in Chomsky normal form (CNF).
- The Earley parser is a top-down algorithm that works for any CFG.

8.1 The CYK algorithm

Remark 8.8

*The CYK algorithm was named after some of its discoverers: John **C**ocke, Daniel **Y**ounger and **K**asami Tadao.*

Definition 8.5 (Constituent chart) *Let $G = (N, \Sigma, P, S)$ a CFG. The* constituent chart *of $w \in \Sigma^\star$ is the (partial) bidimensional table T with columns and rows indexed with $[\![1, |w|+1]\!]$, such that for $i \leq j$, the cell (i, j) contains $T[i, j] = \{A \in N \mid A \underset{G}{\overset{\star}{\Rightarrow}} w_{i:j-1}\}$ and that for $i > j$, the cell (i, j) is undefined.*

Example(s) 8.6 *Consider $G = (N, \Sigma, P, S)$ where:*

- *$N = \{S, NP, VP, PN, DET, N\}$;*

[1] See https://en.wikipedia.org/wiki/Dynamic_programming.

5	{S}	{VP}	{NP}	{N}	$\emptyset$
4	$\emptyset$	$\emptyset$	{DET}	$\emptyset$	
3	{S}	{V, VP}	$\emptyset$		
2	{PN, NP}	$\emptyset$			
1	$\emptyset$				
j/i	1	2	3	4	5
	Sabine	likes	this	truck	

Figure 8.2: Constituent chart of example 8.6.

3	{SV, F}	{N}	{Pro, SN}
2	{Aus}	{Pro, SN}	
1	{Pro, SN}		
j/i	1	2	3
	ho	fame	

Figure 8.3: Constituent chart of example 8.7.

- $\Sigma = \{saw, prepared, this, a(n), truck, experiment, Sabine, Fred, Jamy\}$;

- $$P = \left\{ \begin{array}{l} S \to NP\ VP, \\ NP \to DET\ N \mid PN, \\ VP \to V \mid V\ NP, \\ DET \to this \mid a(n), \\ N \to truck \mid experiment, \\ PN \to Sabine \mid Fred \mid Jamy, \\ V \to likes \mid prepared \end{array} \right\}.$$

Figure 8.2 shows the constituent chart of Sabine likes this truck.

Example(s) 8.7 *Consider $G = (N, \Sigma, P, F)$ where:*

- $N = \{F, SN, SV, Pro, Aus, N\}$;

- $\Sigma = \{ho, fame\}$;

- $$P = \left\{ \begin{array}{l} F \to SN\ SV, \\ SN \to Pro, \\ SV \to Aux\ SN, \\ Pro \to \epsilon, \\ Aus \to ho, \\ N \to fame \end{array} \right\}.$$

Figure 8.3 shows the constituent chart of the Italian sentence ho fame.

Property 8.1 *By definition of the constituent chart, $w \in \Sigma^\star$ is a word generated by a CFG $G = (N, \Sigma, P, S)$ iff its constituent chart T is such that $S \in T[1, |w| + 1]$.*

Remark 8.9 *As we will see below, the CYK algorithm works only with grammars in Chomsky normal form (CNF; see §4.3.5). Such a grammar may contain an ϵ-production $S \to \epsilon$ where S is the axiom, if S does not appear in the right-hand side of any rule.*

In what follows, we ignore this possibility. Indeed, this rule is only used to generate the empty word, and while it would be relatively easily to update everything written below to take *such an ϵ-production into account, the result might feel sometimes a little bit cumbersome. Instead, we assume that the empty word is handled as a trivial special case, and only study the analysis of all other words, which can ignore the possible $S \to \epsilon$ rule.*

Remark 8.10 *Consider $G = (N, \Sigma, P, S)$ and a sequence $w \in \Sigma^\star$. If G is in CNF with no ϵ-production, then the constituent chart is fairly easy to compute:*

- *(ϵ case; $j - i = 0$) $\forall i \in [\![1, |w| + 1]\!]$,*

  $$T[i, i] = \emptyset$$

  Indeed, there is no ϵ-production in G.

- *(lexical case; $j - i = 1$) $\forall i \in [\![1, |w|]\!]$,*

  $$T[i, i + 1] = \{A \in N \mid A \to w_i \in P\}$$

- *(binary case; $j - i \geq 2$) $\forall i \in [\![1, |w| - 1]\!], j \in [\![i + 2, |w| + 1]\!]$,*

  $$T[i, j] = \{A \in N \mid \exists k \in [\![i + 1, j - 1]\!],\ \exists B \in T[i, k], C \in T[k, j],\ A \to B\,C \in P\}$$

  Indeed, by definition of the constituent chart, $A \in T[i, j]$ iff $A \overset{\star}{\Rightarrow} w_{i:j-1}$. Suppose that $i + 2 \leq j$, i.e. that $|w_{i:j-1}| \geq 2$. $A \overset{\star}{\Rightarrow} w_{i:j-1}$ iff $\exists A \to B\,C \in P$, $\exists k \in [\![i + 1, j - 1]\!]$, $B \overset{\star}{\Rightarrow} w_{i:k-1}$ and $C \overset{\star}{\Rightarrow} w_{k:j-1}$ — i.e. iff $B \in T[i, k]$ and $C \in T[k, j]$.

Remark 8.11 *A consequence of remark 8.10 is that for a CFG in CNF G,*

- *$\forall i \in [\![1, |w| + 1]\!]$, the cell $T[i, i]$ can be ignored;*
- *$\forall i \in [\![1, |w|]\!]$, the cell $T[i, i + 1]$ can be computed at any time;*
- *$\forall i \in [\![1, |w| - 1]\!], j \in [\![i + 2, |w| + 1]\!]$, to compute the cell $T[i, j]$, it is necessary and sufficient to know the content of $T[i, i + 1]$, $T[i, i + 2]$, $\cdots$, $T[i, j - 1]$ and $T[i + 1, j]$, $T[i + 2, j]$, $\cdots$, $T[j - 1, j]$. This case is illustrated in figure 8.4.*

This strongly constrains the order with which one can fill the constituent chart, but still leaves room for different strategies. Here are two of the most common strategies.

- Diagonal filling *(figure 8.5): the chart is filled starting from the diagonal $i = j + 1$ (in any order), followed by the diagonal $i = j + 2$, etc., until the top-left corner which is the diagonal $i = j + n$. This strategy corresponds to first analysing independently all spans of length 1 (i.e. single words), then all spans of length 2, etc., and finally the unique spans of length n.*

- Line filling *(figure 8.6): the chart is filled starting from the cell $(1, 2)$ which is the 1st line, followed by the second line from right to left, then the third line from right to left, etc., until the nth line from right to left. This strategy corresponds to first analysing the only non-empty spans ending at w_1, then all non-empty spans ending at w_2, etc., and finally all non-empty spans ending at w_n.*

- Given a grammar in CFG $G = (N, \Sigma, P, S)$, algorithm 16 builds the constituent table of any word using the diagonal strategy.

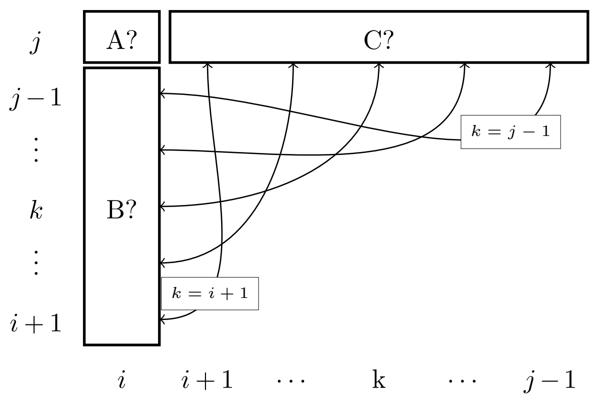


Figure 8.4: How to fill a cell $T[i, j]$ $(i + 2 \leq j)$ using a binary rule $A \to B\,C$. Knowledge of the 'B?' and 'C?' portions is necessary and sufficient to determine the 'A?' cell.

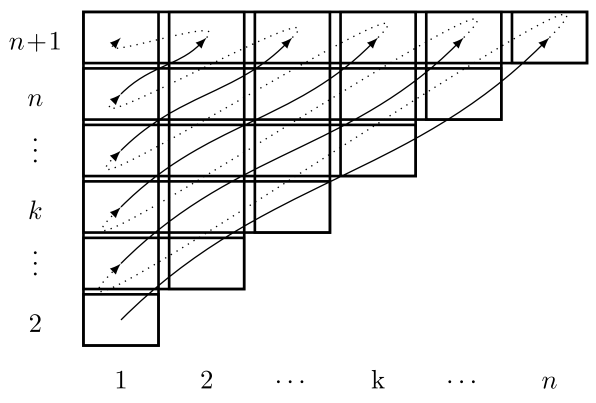


Figure 8.5: Diagonal filling of the constituent chart with a CFG in CNF, filling each diagonal from bottom-left to top-right.

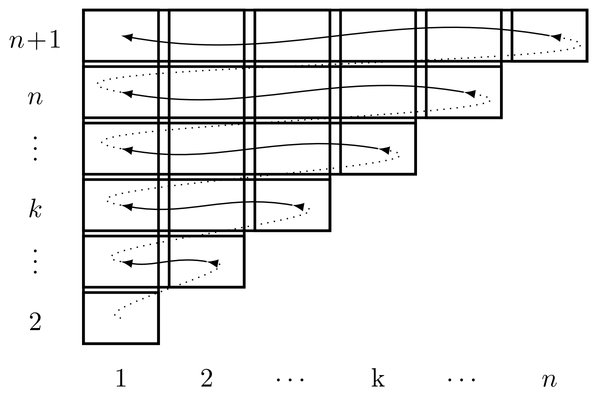


Figure 8.6: Line filling of the constituent chart with a CFG in CNF.

```
// Input:  $u \in \Sigma^\star$
// Output:  the constituent chart of $u$
Function CYK-diagonal($u$)
    $T :=$ empty chart($u$);
    // First diagonal (unary cases)
    for $i := 1$ to $|u|$ do
        foreach $(A \to u_i) \in P$ do
            $T[i, i+1]$.add($A$);
    // Other diagonals (binary cases)
    for $l := 2$ to $|u|$ do                          // loop on the length of the span
        for $i := 1$ to $|u| + 1 - l$ do          // loop on the beginning of the span
            $j := i + l$;                                    // end of the span
            for $k := i + 1$ to $j - 1$ do            // loop on the splitting point
                foreach $(A \to B\,C) \in P$ do
                    if $B \in T[i, k]$ and $C \in T[k, j]$ then
                        $T[i, j]$.add($A$);
    return $T$;
```

Algorithm 16: CYK analyser using a diagonal strategy for filling the constituent chart.

Property 8.2 (Complexity of CYK) *Algorithm 16 is divided in two parts.*

The unary cases part is build around a double loop. The outer loop performs n iterations (where n is the length of the word to analyse). The inner loop performs $|P|$ iterations. The cost of each inner iteration is constant. This first part thus has a complexity in $O(n \times |P|)$.

The binary cases part is build around a quadruple loop. The three outer loops perform collectively $O(n^3)$ iterations: intuitively, they contribute a $O(n)$ factor each, but more formally, they loop once over each possible triple of values in $[\![1, n]\!]$, and there are $\binom{n}{3}$ of them — which is $O(n^3)$. The inner loop performs $|P|$ iterations. The cost of each inner iteration is constant. This second part thus has a complexity in $O(n^3 \times |P|)$.

The worst-time complexity of this algorithm is thus in $O(\mathbf{n^3} \times |P|)$.

Exercise 8.1 *Write the pseudo-code of a CYK analyser that uses the line strategy for filling the constituent chart.*

Derivation trees

Remark 8.12 (Decoding the constituent chart) *Algorithm 17 implements a basic decoding of the constituent chart: this algorithm generates a list of all possible derivation trees of a word from its constituent chart.*

Remark 8.13 *Clearly, the complexity of algorithm 17 is at least the number of resulting trees (the number of operations performed by any algorithm is at least equal to the number of objects it creates), but one can also see that this algorithm does a lot of computation that could be avoided:*

1. *very often, a recursive call to the algorithm is redundant with a previous call,*
2. *which values for k and, for each of them, which rules $A \to B\,C$ should be considered has in fact already been computed when filling the constituent chart.*

```
// Input:  $T$, $A \in N$ and $1 \le i < j \le |u| + 1$
// Output:  a list of all syntactic trees of root $A$ for span $u_{i:j-1}$
Function CYK-trees($T$, $A$, $i$, $j$)
    if $A \notin T[i, j]$ then return [];
    // Unary case
    if $i = j - 1$ then
        return [$UnaryTree(A, Leaf(u_i))$];
    // Binary case
    trees := [];
    for $k := i + 1$ to $j - 1$ do                // loop on the splitting point
        foreach $(A \to B\,C) \in P$ do
            left_trees := CYK-trees($T$, $B$, $i$, $k$);
            right_trees := CYK-trees($T$, $C$, $k$, $j$);
            foreach $l \in$ left_trees, $r \in$ right_trees do
                trees.append(BinaryTree($A, l, r$));
    return trees;
```

Algorithm 17: Enumeration of all derivation trees from the constituent chart T. The initial value of A must be S, these of i and j must be 1 and $|u| + 1$ respectively. (This algorithm is quite inefficient.)

Of these two sources of inefficiency, only the first has an impact on the worst-case *time complexity.*

Remark 8.14 (Efficient decoding) *Efficient decoding can be achieved using dynamic programming, which amounts here to using another structure T' to store intermediate results. The intermediate results are all trees anchored in a given span: in algorithm 18, which implement this idea, $T'[i, j, A]$ (when defined) stores all trees of root labelled A that span $w_{i:j-1}$.*

8.2 Earley algorithm(s)

TODO: see slides; algorithm 19 and algorithm 20

Vocabulary

- chart parsing: *analyse syntaxique tabulaire*
- the correctness of an analysis: *la correction d'une analyse*
- combinatorial explosion: *explosion combinatoire*
- worst-time complexity: *complexité dans le pire des cas*
- constituent chart: *table des constituents*

```
// Input:  $T, T'$,  $A \in N$ and $1 \leq i < j \leq |u| + 1$
// Output:  a list of all syntactic trees of root labelled $A$ that span
    $u_{i:j-1}$
Function CYK-trees($T$, $T'$, $A$, $i$, $j$)
    if $T'[i, j, A]$ is defined then return $T'[i, j, A]$;
    if $A \notin T[i, j]$ then return [];
    // Unary case
    if $i = j - 1$ then
        return [$UnaryTree(A, Leaf(u_i))$];
    // Binary case
    trees := [];
    for $k := i + 1$ to $j - 1$ do                    // loop on the splitting point
        foreach $(A \to B\,C) \in P$ do
            left_trees := CYK-trees($T$, $T'$, $B$, $i$, $k$);
            right_trees := CYK-trees($T$, $T'$, $C$, $k$, $j$);
            foreach $l \in$ left_trees, $r \in$ right_trees do
                trees.append(BinaryTree($A, l, r$));
    $T'[i, j, A]$ := trees;
    return trees;
```

Algorithm 18: Enumeration of all derivation trees from the T chart. The initial value of A must be S, these of i and j must be 1 and $|u| + 1$ respectively, that of T' must an empty chart (all cells are undefined).

```
// Input:  $u \in \Sigma^\star$
// Output:  the Earley chart for $u$
Function earley-simple($u$)
    // Initialisation
    $T :=$ empty chart$(u)$;
    for $j := 1$ to $|u| + 1$ do
        $T[j] :=$ ordered_set();
        foreach $(A \to \alpha) \in P$ do $T[j]$.add$((A \to \bullet\, \alpha, j))$;
    // Main part
    for $j := 1$ to $|u| + 1$ do                                // loop on the cells
        $k := 0$;                                              // index in the cell T[j]
        while $k < len(T[j])$ do                    // loop on the items of the cell T[j]
            $(A \to \alpha \bullet \beta, i) := T[j][k]$;
            if $\beta = \epsilon$ then
                // comp?
                $k' := 0$;                                     // index in the cell T[i]
                while $k' < len(T[i])$ do           // loop on the items of the cell T[i]
                    $(A' \to \alpha' \bullet \beta', i') := T[i][k']$;
                    if $\beta'_1 = A$ then
                        $T[j]$.add$((A' \to \alpha' \beta'_1 \bullet \beta'_{2:|\beta'|}, i'))$;       // output of comp
                    $k' \mathrel{+}= 1$;
            else if $j < |u| + 1$ then
                // scan?
                if $\beta_1 = u_j$ then
                    $T[j+1]$.add$((A \to \alpha \beta_1 \bullet \beta_{2:|\beta|}, i))$;          // output of scan
            $k \mathrel{+}= 1$;
    return $T$;
```

Algorithm 19: Simple Earley analysis.

```
// Input: $u \in \Sigma^\star$
// Output: the Earley chart for $u$
Function earley($u$)
    // Initialisation
    $T :=$ empty chart($u$);
    for $j := 1$ to $|u| + 1$ do $T[j] :=$ ordered_set();
    foreach $(S \to \alpha) \in P$ do $T[1]$.add($(S \to \bullet\,\alpha, 1)$);
    // Main part
    for $j := 1$ to $|u| + 1$ do                                   // loop on the cells
        $k := 0$;                                                  // index in the cell T[j]
        while $k < len(T[j])$ do                    // loop on the items of the cell T[j]
            $(A \to \alpha \bullet \beta, i) := T[j][k]$;
            if $\beta = \epsilon$ then
                // comp?
                $k' := 0$;                                         // index in the cell T[i]
                while $k' < len(T[i])$ do           // loop on the items of the cell T[i]
                    $(A' \to \alpha' \bullet \beta', i') := T[i][k']$;
                    if $\beta'_1 = A$ then
                        $T[j]$.add($(A' \to \alpha'\,\beta'_1 \bullet \beta'_{2:|\beta'|}, i')$);     // output of comp
                    $k' \mathrel{+}= 1$;
            else if $\beta_1 \in N$ then
                // pred?
                foreach $(\beta_1 \to \gamma) \in P$ do $T[j]$.add($(\beta_1 \to \bullet\,\gamma, j)$); // output of pred
            else if $j < |u| + 1$ then
                // scan?
                if $\beta_1 = u_j$ then $T[j+1]$.add($(A \to \alpha\,\beta_1 \bullet \beta_{2:|\beta|}, i)$) ;   // output of
                 scan
            $k \mathrel{+}= 1$;
    return $T$;
```

Algorithm 20: Earley analysis

Chapter 9

Grammar-less parsing

TODO: see slides

Relevant reading

- 'A Classifier-Based Parser with Linear Run-Time Complexity' by Sagae and Lavie (2005).
- 'Dependency parsing' by Nivre (2010).
- 'Modern approaches to parsing' (slides) by Bernard and Amsili (2023b).

Bibliography

Bernard, Timothée and Pascal Amsili (2023a). *Formal languages and syntactic complexity.* Natural language syntax: parsing and complexity (ESSLLI course). University of Ljubljana, Ljubljana, Slovenia. URL: https://lattice.cnrs.fr/amsili/EnsPerm/esslli23_files/NLSPC23_day1_v1.pdf.

Bernard, Timothée and Pascal Amsili (2023b). *Modern approaches to parsing.* Natural language syntax: parsing and complexity (ESSLLI course). University of Ljubljana, Ljubljana, Slovenia. URL: https://lattice.cnrs.fr/amsili/EnsPerm/esslli23_files/NLSPC23_day4_v0.pdf.

Blix, Hagen and Adina Williams (2022). *Phases and Phrases: Some thoughts on Weak Equivalence, Strong Equivalence, and Empirical Coverage.* URL: https://wp.nyu.edu/morphlab/2022/03/09/phases-and-phrases-some-thoughts-on-weak-equivalence-strong-equivalence-and-empirical-coverage/.

Branco, António (2018). 'Computational Complexity of Natural Languages: A Reasoned Overview'. In: *Proceedings of the Workshop on Linguistic Complexity and Natural Language Processing.* Santa Fe, New-Mexico: Association for Computational Linguistics, pp. 10–19. URL: https://aclanthology.org/W18-4602.

Chomsky, N. (1956). 'Three models for the description of language'. In: *IRE Transactions on Information Theory* 2.3, pp. 113–124. DOI: 10.1109/TIT.1956.1056813.

Nivre, Joakim (2010). 'Dependency Parsing'. In: *Language and Linguistics Compass* 4.3, pp. 138–152. DOI: 10.1111/j.1749-818X.2010.00187.x.

Sagae, Kenji and Alon Lavie (2005). 'A Classifier-Based Parser with Linear Run-Time Complexity'. In: *Proceedings of the Ninth International Workshop on Parsing Technology.* Vancouver, British Columbia: Association for Computational Linguistics, pp. 125–132. URL: https://www.aclweb.org/anthology/W05-1513.

Savitch, W. J. et al. (1987). 'Introduction'. In: *The Formal Complexity of Natural Language.* Ed. by W. J. Savitch et al. Studies in Linguistics and Philosophy. Springer Netherlands, pp. vii–xv.

Shieber, Stuart M. (1985). 'Evidence against the context-freeness of natural language'. In: *Linguistics and Philosophy* 8.3, pp. 333–343. DOI: 10.1007/BF00630917.

Yvon, François and Akim Demaille (2016). 'Théorie des langages'. Lecture notes. URL: https://www.lrde.epita.fr/~akim/thl/lecture-notes/theorie-des-langages-2.pdf.