15
Learning Context-Free Grammars

Too much faith should not be put in the powers of induction, even when aided by intelligent heuristics, to discover the right grammar. After all, stupid people learn to talk, but even the brightest apes do not.

Noam Chomsky, 1963

It seems a miracle that young children easily learn the language of any environment into which they were born. The generative approach to grammar, pioneered by Chomsky, argues that this is only explicable if certain deep, universal features of this competence are innate characteristics of the human brain. Biologically speaking, this hypothesis of an inheritable capability to learn any language means that it must somehow be encoded in the DNA of our chromosomes. Should this hypothesis one day be verified, then linguistics would become a branch of biology.

Niels Jerne, Nobel Lecture, 1984

Context-free languages correspond to the second 'easiest' level of the Chomsky hierarchy. They comprise the languages generated by context-free grammars (see Chapter 4).

All regular languages are context-free but the converse is not true. Between the languages that are context-free but not regular some 'typical' ones are:

- $\{\mathtt{a}^n\mathtt{b}^n : n \geq 0\}$. This is the classical text-book language used to show that automata cannot count in an unrestricted way.

- $\{w \in \{\mathtt{a},\mathtt{b}\}^* : |w|_{\mathtt{a}} = |w|_{\mathtt{b}}\}$. This language is a bit more complicated than the previous one. But the same argument applies: You cannot count the a's and the b's nor the difference between the number of occurrences of each letter.
- The language of *palindromes*: $\{w \in \{\mathtt{a},\mathtt{b}\}^* : |w| = n,\ \wedge\ \forall i \in [n]\ w(i) = w(n-i+1)\} = \{w \in \{\mathtt{a},\mathtt{b}\}^* : w = w^R\}$.
- *Dyck*, or the language of well formed brackets. The language of all bracketed strings or balanced parentheses is classical in formal language theory. If just working on one pair of brackets (denoted by a and b) it is defined by the rewriting system $\langle\{\mathtt{ab} \vdash \lambda\}, \lambda\rangle$, *i.e.* by those strings whose brackets all disappear by deleting iteratively every substring ab. The language is context-free and can be generated by the grammar $\langle\{\mathtt{a},\mathtt{b}\}, \{N_1\}, R, N_1\rangle$ with $R = \{N_1 \rightarrow \mathtt{a}N_1\mathtt{b}N_1; N_1 \rightarrow \lambda\}$. This known as $Dyck_1$, because it uses only one pair of brackets. And for each n, the language $Dyck_n$, over n pairs of brackets, is also context-free.
- The language generated by the grammar $\langle\{\mathtt{a},\mathtt{b}\}, \{N_1\}, R, N_1\rangle$ with $R = \{N_1 \rightarrow \mathtt{a}N_1N_1; N_1 \rightarrow \mathtt{b}\}$ is called the *Lukasiewicz* language.

It has been suggested by many authors that context-free grammars are a better model for natural language than regular grammars, even if it is also admitted that a certain number of constructs can only be found in context-sensitive languages.

There are other reasons for wanting to learn context-free grammars: These appear in computational linguistics or in the analysis of web documents where the tag languages need opening and closing tags. In bio-informatics, also, certain constructs of the secondary structure are context-free.

15.1 The difficulties

When moving up from the regular world to the context-free world we are faced with a whole set of new difficulties.

Do we learn context-free languages or context-free grammars? This is going to be the first (and possibly most crucial) question. When dealing with regular languages (and grammars) the issue was much less troublesome as the Myhill-Nerode theorem provides us with a nice one to one relationship between the languages and the automata. In the case of context-freeness, there are several reasons that make it difficult to consider grammars instead of languages:

- The first very serious issue is that equivalence of context-free grammars is undecidable. This is also the case for the subclass of the linear grammars. As an immediate consequence there will be the fact that canonical forms will be unavailable, at least in a constructible way. Moreover, the critical importance of the undecidability issue can be seen in the following problem:

Suppose class $\boldsymbol{\mathcal{L}}$ is learnable, and we are given two grammars G_1 and G_2 for languages in $\boldsymbol{\mathcal{L}}$. Then we could perhaps generate examples from $\mathbb{L}(G_1)$ and learn some grammar H_1 and do the same from $\mathbb{L}(G_2)$ obtaining H_2. Checking the syntactic equality between H_1 and H_2 corresponds in an intuitive way to solving the equivalence between G_1 and G_2. Moreover the fact that the algorithm constructs in some way a normal and canonical form, since it depends on the examples, is puzzling. The question we raise here is: 'Can we use a grammatical inference algorithm to solve the equivalence problem?'.

This is obviously not a tight argument. But if one requires 'learnable' to mean 'have characteristic samples' then the above reasoning at least proves that the so called characteristic samples have to be uncomputable.

- A second troubling issue is that of 'expansiveness': In certain cases the grammar can be exponentially smaller than any string in the language: Consider for instance grammar $G_n = \langle\{\mathtt{a}\}, \{N_k : k \in [n]\}, R_n, N_1\rangle$ with $R_n = \bigcup_{i<n}\{N_i \to N_{i+1}N_{i+1}\} \cup \{N_n \to \mathtt{a}\}$. Then the only string in the language $\mathbb{L}(G_n)$ is $\mathtt{a}^{2^{n-1}}$ which is of length 2^{n-1}. There is therefore an exponential relation between the size of the grammar and the length of even the shortest strings the grammar can produce. If we take a point of view where learning is seen as a compression question, then compressing into logarithmic size is surely not a problem and is most recommendable. But on the other hand if the question we ask is "what examples are needed to learn?", then we face a problem. In the terms we have been using so far, the characteristic samples would be exponential..
- When studying the learnability of regular languages, there was an important difference between learning deterministic representations and non-deterministic ones. In the case of context-freeness, things are even more complex as there are two different notions related to determinism.

The first possible notion corresponds to *ambiguity*: A grammar is ambiguous if it admits ambiguous strings, *i.e.* strings that have two different derivation trees associated. It is well known that there exist inherently ambiguous languages, *i.e.* languages for which all grammars have to be ambiguous. All reasonable questions relating to ambiguity are undecidable, so one cannot limit oneself to the class of the unambiguous languages, nor check the ambiguity of an individual string.

The second possible notion is used by *deterministic languages*. Here, determinism refers to the determinism of the pushdown automaton that recognises the language. There is a well represented subclass of the deterministic languages for which, furthermore, the equivalence problem is decidable. There have been no serious attempts to learn deterministic pushdown automata, so we will not enter this subject here.

- *Intelligibility* is another issue that becomes essential when dealing with context-free grammars. A context-free language can be generated by many very different grammars, some of which fit the structure of the language better than others. Take for example the grammar (based on the Lukasiewicz grammar) $\langle\{\mathtt{a},\mathtt{b}\},\{N_1,N_2\},R,N_1\rangle$ with $R=\{N_1\to\mathtt{a}N_2N_2;\ N_2\to\mathtt{b};\ N_1\to\mathtt{a}N_2; N_1\to\lambda\}$. Is this a better grammar to generate the single bracket language? An equivalent grammar would be the grammar in Chomsky (quadratic) normal form: $\langle\{\mathtt{a},\mathtt{b}\},\{N_1,N_2,N_3,A,B\},\ R,N_1\rangle$ with $R=\{N_1\to\lambda+N_2N_3;\ N_2\to AN_1;\ N_3\to BN_1;\ A\to\mathtt{a};\ B\to\mathtt{b}\}$.

  The question we are raising is that there is really a lot of semantics hidden in the structure defined by the grammar. This involves yet another reason for considering that the problem is about learning context-free grammars rather than context-free languages!

15.1.1 Dealing with linearity

As regular languages, linear languages and context-free languages all share the curse of not being learnable from positive examples, an alternative is to reduce the class of languages in order to obtain a family that would not be super-finite, but on the other hand that would be identifiable.

Definition 15.1.1 (Linear context-free grammars) *A context-free grammar $G=(\Sigma,V,R,\ N_1)$ is **linear** if$_{def}$ $R\subset V\times(\Sigma^*V\Sigma^*\cup\Sigma^*)$.*

Definition 15.1.2 (Even linear context-free grammars) *A context-free grammar $G=(\Sigma,\ V,R,N_1)$ is an **even linear** grammar if$_{def}$ $R\subset V\times(\Sigma V\Sigma\cup\Sigma\cup\{\lambda\})$.*

Thus languages like $\{a^nb^n : n\in\mathbb{N}\}$, or the set of all palindromes, are even linear without being regular. But using reduction techniques from Section 7.4, we find a clear relationship with the regular languages. Indeed the operation allowing to simulate an even linear grammar by a finite automaton is called a ***regular reduction***:

Definition 15.1.3 (Regular reduction) *Let $G = (\Sigma, V, R, N_1)$ be an even linear grammar. We say that the* NFA $\mathcal{A} = \langle \Sigma_R, Q, q_1, q_F, \emptyset, \delta_R \rangle$ *is the* ***regular reduction*** *of G if$_{def}$*

- $\Sigma_R = \{\langle ab \rangle : a, b \in \Sigma\} \cup \Sigma$;
- $Q = \{q_i : N_i \in V\} \cup \{q_F\}$;
- $\delta_R(q_i, \langle ab \rangle) = \{q_j : (N_i, aN_jb) \in R\}$;
- $\forall a \in \Sigma,\ \delta_R(q_i, a) = \{q_F : (N_i, a) \in R\}$;
- $\forall i$ *such that* $(N_i, \lambda) \in R$, $q_F \in \delta_R(q_i, \lambda)$.

Theorem 15.1.1 *Let G be an even linear grammar and let R be its regular reduction. Then $a_1 \cdots a_n \in \mathbb{L}(G)$ if and only if $\langle a_1 a_n \rangle \langle a_2 a_{n-1} \rangle \cdots \in \mathbb{L}(R)$.*

Proof This is clear by the construction of the regular reduction, but more detail can be found in the construction presented in Section 7.4.3 (page 184). □

The corollary of the above construction is that any technique based on learning the class of all regular languages or subclasses of regular languages can be transposed to subclasses of even linear languages. For instance, in the setting of learning from positive examples only, positive results concerning subclasses of even linear languages have been obtained.

Very simple grammars are a very restricted form of grammar that are not linear but are strongly deterministic. They constitute another class of context-free grammars for which positive learning results have been obtained. They are context-free grammars in a restricted Greibach normal form:

Definition 15.1.4 (Very simple grammars) *A context-free grammar $G = (\Sigma, V, R, N_1)$ is a* ***very simple grammar*** *if$_{def}$ $R \subset (V \times \Sigma V^*)$ and for any $a \in \Sigma$ $(A, a\alpha) \in R \wedge (B, a\beta) \in R \implies [A = B \wedge \alpha = \beta]$.*

Lemma 15.1.2 (Some properties of very simple grammars)
Let $G = (\Sigma, V, R, N_1)$ be a very simple grammar, let $\alpha, \beta \in V^+$ and let $x \in \Sigma^+, u, u_1, u_2 \in \Sigma^*$. Then:

- $N_1 \overset{*}{\Longrightarrow} x\alpha \wedge N_1 \overset{*}{\Longrightarrow} x\beta \Rightarrow \alpha = \beta$ *(forward determinism);*
- $\alpha \overset{*}{\Longrightarrow} x \wedge \beta \overset{*}{\Longrightarrow} x \Rightarrow \alpha = \beta$ *(backward determinism);*
- $N_1 \overset{*}{\Longrightarrow} u_1\alpha \overset{*}{\Longrightarrow} u_1 x \wedge N_1 \overset{*}{\Longrightarrow} u_2\beta \overset{*}{\Longrightarrow} u_2 x \Rightarrow u_1^{-1}L = u_2^{-1}L$.

Very simple grammars are therefore deterministic both for a top-down and a bottom-up parse. Moreover, a nice congruence can be extracted, which will prove to be the key to building a succesfull identification algorithm. One should point out that they are nevertheless quite limited: Each symbol in the final alphabet can only appear once in the entire grammar.

Example 15.1.1 *Grammar $G = (\Sigma, V, R, N_1)$ (with $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}, \texttt{f}\}$) is a very simple grammar:*

$$\begin{aligned} N_1 &\rightarrow \texttt{a}N_1N_2 + \texttt{f} \\ N_2 &\rightarrow \texttt{b}N_2 + \texttt{c} + \texttt{d}N_3N_3 \\ N_3 &\rightarrow \texttt{e} \end{aligned}$$

The language generated by G can be represented by the following extended regular expression: $\texttt{a}^n\texttt{f}\big(\texttt{b}^*(\texttt{c} + \texttt{dee})\big)^n$.

Theorem 15.1.3 *The class of very simple grammars can be identified in the limit from text by an algorithm that*

- *has polynomial update time,*
- *makes a polynomial number of implicit prediction errors and mind changes.*

Proof [sketch] Let us describe the algorithm. As in a very simple grammar for any $a \in \Sigma$ there is exactly one rule of shape $(N, a\alpha) \in R$, so the number of rules in the grammar is exactly $|\Sigma|$ and there are at most $|\Sigma|$ non-terminals. The algorithm goes through three steps:

Step 1 For each $a \in \Sigma$ and making use of equations in Lemma 15.1.2 determine the left part of the only rule in which a appears.

Step 2 As there is exactly one rule for each terminal symbol the rules applied in the parsing of any string are known. Then, we can construct an equation, for each training string, that relates the length of the string and the lengths of the right part of the rules used in the derivation of the string. We now solve the system of equations to determine the length of the right-hand side of each rule.

Step 3 Simulating the parse for each training string, we determine the order in which the rules are applied and the non-terminals that appear on the right-hand side of the rules.

□

We run the sketched algorithm on a simple example. Suppose the data consists of the strings $\{\texttt{afbc}, \texttt{f}, \texttt{afbbc}, \texttt{aec}, \texttt{afbdee}\}$. Step 1 will allow us to cluster the letters into 3 groups: $\{\texttt{b}, \texttt{c}, \texttt{d}\}$, $\{\texttt{a}, \texttt{f}\}$ and $\{\texttt{e}, \texttt{g}\}$. Indeed since we have $N_1 \overset{*}{\Longrightarrow} \texttt{afb}\alpha \overset{*}{\Longrightarrow} \texttt{afbc} \wedge N_1 \overset{*}{\Longrightarrow} \texttt{afb}\beta \overset{*}{\Longrightarrow} \texttt{afbbc}$ we deduce that $\alpha = \beta$ and that the left-hand side of the rules for `b` and `c` are identical. Now for step 2 and simplifying we can deduce that the rules corresponding to `c`, `e` and `f` are all of length 1 (so $N_e \leftarrow \texttt{e}$). It follows that the rules for letter `b` is of length 2 and those for `a` and `d` are of length 3. We can now materialise this by bracketing the strings in the learning sample:

$$\{(\texttt{af(bc)}), (\texttt{f}), (\texttt{af(b(bc))}), (\texttt{aec}), (\texttt{af(b(dee))})\}$$

And by reconstruction we get the fact that the rules are:

$$\begin{aligned} N_1 &\rightarrow \texttt{a}N_1N_2 + \texttt{f} \\ N_2 &\rightarrow \texttt{b}N_2 + \texttt{c} + \texttt{d}N_3N_3 \\ N_3 &\rightarrow \texttt{e}. \end{aligned}$$

One should note that the complexity will rise exponentially with the size of the alphabet.

15.1.2 Dealing with determinism

It might seem from the above that the key to success is to limit ourselves to linear grammars, but if we consider Definition 7.3.3 the results are negative:

Theorem 15.1.4 *For any alphabet Σ of size at least two, $\mathcal{LIN}(\Sigma)$ cannot be identified in the limit by polynomial characteristic samples from an informant.*

Proof Consider two linear languages, at least one string of their symmetric difference should appear in the characteristic sample in order to be able to distinguish them. But the length of the smallest string in the symmetric difference cannot be bounded by any polynomial in the size of the grammar since deciding if two linear grammars are equivalent is undecidable.

It should be noted that this result is independent of the sort of representation that is used. Further elements concerning this issue are discussed in Section 6.4. □

Corollary 15.1.5 *For any alphabet Σ of size at least two, $\mathcal{CFG}(\Sigma)$ cannot be identified in the limit by polynomial characteristic samples from an informant.*

We saw in Chapter 12, that DFA were identifiable in the limit by polynomial characteristic samples (POLY-CS polynomial time) from an informant.

So if we want to get positive results in this setting, we need to restrict further the class of linear grammars.

Deterministic linear grammars provide a non-trivial extension of the regular grammars:

Definition 15.1.5 (Deterministic linear grammars) *A **deterministic linear context-free grammar** $G = (\Sigma, V, R, N_1)$ is a (linear) grammar where $R \subset \ \times (\Sigma \ V\Sigma^* \cup \{\lambda\})$ and $(N, a\alpha), (N, a\beta) \in R \Rightarrow \alpha = \beta$.*

Definition 15.1.6 (Deterministic linear grammar normal form) *A deterministic linear grammar $G = (\Sigma, V, R, N_1)$ is in **normal form** if$_{def}$*

(i) *G has no useless non-terminals;*
(ii) *$\forall (N, aN'w) \in R, w = \mathrm{lcs}(a^{-1}L_G(N))$;*
(iii) *$\forall N, N' \in R,\ L_G(N) = L_G(N') \Rightarrow N = N'$.*

Remember that $\mathrm{lcs}(L)$ is the least common suffix of language L. Having a *nice* normal form allows us to claim:

Theorem 15.1.6 *The class of deterministic linear grammars can be identified in the limit in polynomial time and data from an informant.*

Proof [sketch] The algorithm works by an incremental (by levels) construction of the canonical grammar.

The algorithm maintains a queue of non-terminals to explore. At the beginning the start symbol is added to the grammar and to the exploration queue. At each step, a non-terminal (N) is extracted from the queue and a terminal symbol (a) is chosen in order to further parse the data. From these a new rule is proposed, based on the second condition of Definition 15.1.6 of the normal form for deterministic linear grammars: $N \rightarrow aN_? w$. Each time a new rule is proposed the only non-terminal that appears on its right-hand side ($N_?$) is checked for equivalence with a non-terminal in the grammar. We denote this non-terminal by $N_?$ in order to indicate that it is still to be named.

If a compatible non-terminal is found, the non-terminal in the rule is named after it. If no non-terminal is found, a new non-terminal is added to the grammar (corresponding to a promotion) and to the exploration list. In both cases the rule is added to the grammar.

By simulating the run of this algorithm over a particular grammar, a characteristic sample can be constructed. □

We run the sketched algorithm on an example. Let the learning sample consist of sets $S_+ = \{\texttt{abbabb}, \texttt{bba}, \texttt{babbaaa}, \texttt{aabbabbbb}, \texttt{baabbabbaa}\}$ and $S_- = \{\texttt{b}\}$.

The first non terminal is N_1 and the first terminal symbol we choose is $\texttt{a}$. Therefore a rule with profile $N_1 \rightarrow \texttt{a}N_?w$ is considered. First, string w is sought by computing lcs($\{\texttt{bbabb}, \texttt{abbabbbb}\}$)$=$ $\texttt{bb}$.

Therefore rule $N_1 \rightarrow \texttt{a}N_?\texttt{bb}$ is suggested as a starting point (N_1 being the axiom). Can non-terminal $N_?$ be merged with N_1? Since rule $N_1 \rightarrow \texttt{a}N_1\texttt{bb}$ does not create a conflict, the merge is accepted and the rule is kept. Thus the current set of rules is $N_1 \rightarrow \texttt{a}N_1\texttt{bb}; N_1 \overset{*}{\Longrightarrow} \texttt{bba}+\texttt{babbaaa}+\texttt{baabbabbaa}$.

Now terminal symbol $\texttt{b}$ is brought forward and the different elements of the corresponding rule $N_1 \rightarrow \texttt{b}N_?w$ have to be identified. We start with w which is lcs($\{\texttt{ba}, \texttt{abbaaa}, \texttt{aabbabbaa}\}$) so $\texttt{a}$. Again adding rule $N_1 \rightarrow \texttt{b}N_1\texttt{a}$ is considered but the resulting grammar (with rules $N_1 \rightarrow \texttt{a}N_1\texttt{bb}$; $N_1 \rightarrow \texttt{b}N_1\texttt{a}$; $N_1 \overset{*}{\Longrightarrow} \texttt{b} + \texttt{aabbabba}$) would generate string $\texttt{b}$ which is in S_-.

So the current grammar is $\{N_1 \rightarrow \texttt{a}N_1\texttt{bb};\ N_1 \rightarrow \texttt{b}N_2\texttt{a};\ N_2 \overset{*}{\Longrightarrow} \texttt{b} + \texttt{abbaaa} + \texttt{aabbabba}\}$. We compute lcs($\{\texttt{bbaaa}, \texttt{aabbabba}\}$) $=$ $\texttt{a}$. Therefore $N_2 \rightarrow \texttt{a}N_?\texttt{a}$ is accepted. We are left with $\texttt{b}$ (this leads to the rule $N_2 \rightarrow \texttt{b}$) and finally the grammar contains the following rules:

$$\begin{array}{lcl} N_1 & \rightarrow & \texttt{a}N_1\texttt{bb} + \texttt{b}N_2\texttt{a} \\ N_2 & \rightarrow & \texttt{a}N_1\texttt{a} + \texttt{b}. \end{array}$$

15.1.3 Dealing with sparseness

A string is the result of many rules that have all got to be learnt in 'one shot'. In a certain sense, there is an *all or nothing* issue here: Hill climbing seems to be impossible, and the number of examples needed to justify the set of all the rules can easily seem too large.

Moreover, from string to string (in the language) local modifications may not work. This can be measured in the following way: Given two strings u and v in L, the number of modifications one needs to make to string u in order to obtain string v is going to be such that one will not be able to use the couple (u, v) to learn an isolated rule which allows to get from u to v.

15.2 Learning reversible context-free grammars

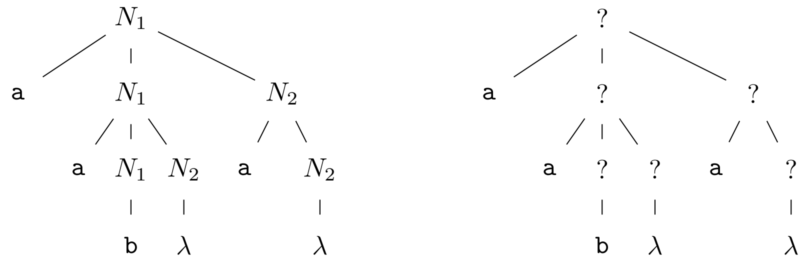
In Section 11.2 (page 262), we introduced a class of look-ahead languages. Learning could take place by eliminating the sources of ambiguity through merging. This was done in the context of the regular languages. We now show how this idea can also lead to an algorithm for context-free languages, even if we will need some extra information about the structure of the strings.

15.2.1 Unlabelled trees or skeletons

In practice the positive data from which we will usually be learning from cannot be trees, labelled at the internal nodes. It will either just consist in the strings themselves or, in a more helpful setting, in bracketed strings. As explained in Section 3.3.1 (page 60), these correspond to trees with unlabelled internal nodes.

Definition 15.2.1 *Let $G = \langle \Sigma, V, R, N_1 \rangle$ be a context-free grammar, a skeleton for string α (over $\Sigma \cup V$) is a derivation tree with frontier α and in which all internal nodes are labelled by a new symbol '?'.*

In Figure 15.1(a) we represent a parse tree for `aaba`, and in Figure 15.1(b) the corresponding skeleton.



(a) A parse tree. (b) The corresponding skeleton.

Fig. 15.1. A parse tree for `aaba` and the corresponding skeleton. Some of the grammar rules are $N_1 \rightarrow \mathtt{a}N_1N_2$, $N_1 \rightarrow \mathtt{b}$, $N_2 \rightarrow \mathtt{a}N_2$ and $N_2 \rightarrow \lambda$.

15.2.2 K-contexts

Definition 15.2.2 *Let $G =< \langle \Sigma, V, R, N_1 \rangle$ be a context-free grammar. A **k-deep** derivation in G is a derivation*

$$\begin{aligned} N_{i_0} &\Rightarrow \alpha_1 N_{i_1} \beta_1 \\ &\Rightarrow \alpha_1 \alpha_2 N_{i_2} \beta_2 \beta_1 \\ &\overset{k}{\Longrightarrow} \alpha_1 \alpha_2 \cdots \alpha_{k-1} \alpha_k N_{i_k} \beta_k \beta_{k-1} \cdots \beta_2 \beta_1 \end{aligned}$$

where $\forall l \leq k,\ \alpha_l, \beta_l \in (\Sigma \cup V)^*$.

Intuitively, this corresponds to a tree with just one long branch of length k.

Definition 15.2.3 *Let $G =< \langle \Sigma, V, R, N_1 \rangle$ be a context-free grammar and N_i, N_j be two non-terminal symbols in V. N_j is a **k-ancestor** of N_i if$_{def}$ there exists a k-deep derivation of N_j into $\alpha N_i \beta$, $(\alpha, \beta \in (\Sigma \cup V)^*)$.*

We define these as sets, *i.e.* k-ancestors(N) is the set of all k-ancestors of non-terminal N.

Example 15.2.1 *From Figure 15.1(a), we can compute:*

- *2-ancestors*$(N_1) = \{N_1\}$, *2-ancestors*$(N_2) = \{N_1,\, N_2\}$,
- *1-ancestors*$(N_1) = \{N_1\}$, *1-ancestors*$(N_1) = \{N_1,\, N_2\}$.

Now a k-context is defined as follows:

Definition 15.2.4 *Let $G =< \langle \Sigma, V, R, N_1 \rangle$ be a context-free grammar and a specific non-terminal N_i in V. The **k-contexts** of N_i are all trees built as follows:*

(i) *Let t be the derivation tree for a derivation $N_j \overset{k}{\Longrightarrow} \alpha N_i \beta \overset{*}{\Longrightarrow} u N_i v$ where derivation $N_j \overset{k}{\Longrightarrow} \alpha N_i \beta$ is a k-deep derivation and $\alpha \overset{*}{\Longrightarrow} u$ and $\beta \overset{*}{\Longrightarrow} v$, with $u, v \in \Sigma^\star$.*

(ii) *Let $z_\$$ be the address of N_i in t $(t(z_\$) = N_i,\, |z_\$| = k)$.*

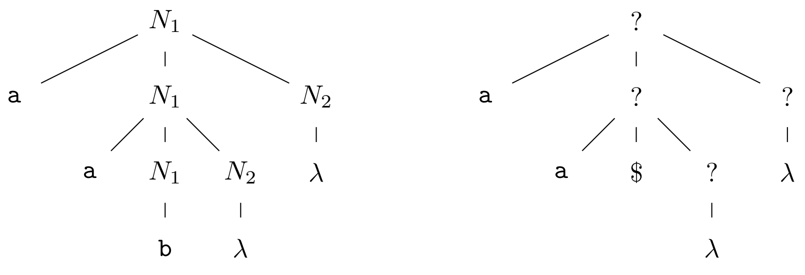
(iii) *We build the k-context $c[t, z_\$]$ as a tree of domain $\mathrm{Dom}(t) \setminus \{z_\$ a u : a \in \mathbb{N},\, u \in \mathbb{N}^*\}$ and such that $c[t, z_\$] : \mathrm{Dom}(t) \to \Sigma \cup\ V \cup \{\lambda, \$, ?\}$, with:*

- $c[t, z_\$] = \$$
- $c[t, u] = ?$ *if* $u1 \in \mathrm{Dom}(t)$ *(i.e. if u is an internal node of the tree)*
- $c[t, u] = t(u)$ *if not.*

Example 15.2.2 *Consider the grammar with rules $N_1 \to \mathtt{a} N_1 N_2$, $N_1 \to \mathtt{b}$, $N_2 \to \mathtt{a} N_2$ and $N_2 \to \lambda$. We show in Figure 15.2(a) a parse tree t, and the corresponding 2-context $c[t, 11]$ in Figure 15.2(b).*

Note that a non-terminal can have an infinity of k-contexts, but only one 0-context, which is always \$.

In practice we will not be given a grammar from which one would compute the k-contexts. Instead, a basic grammar is constructed from a learning sample. This will allow to be sure that the number of k-contexts remains finite. We thus denote by k-contexts(S_+, N_i) the set of all k-contexts of non-terminal N_i with respect to sample S_+.



(a) A parse tree t. (b) A 2-context.

Fig. 15.2. A parse tree t and the 2-context $c[t, 11]$.

15.2.3 K-reversible grammars

From the above we now define k-reversible grammars:

Definition 15.2.5 *A context-free grammar G is k-**reversible** if$_{def}$ the following two properties hold:*

(i) *if there exists two rules $N_l \rightarrow \alpha N_i \beta$ and $N_l \rightarrow \alpha N_j \beta$ then $N_i = N_j$ (invertibility condition);*
(ii) *if there exist two rules $N_i \rightarrow \alpha$ and $N_j \rightarrow \alpha$ and there is a k-context common to N_i and N_j, then $N_i = N_j$ (reset-free condition).*

To say that a language is k-reversible (for some k) is nevertheless not that informative; if being k reversible implies strong rules over the type of grammars, this is not true for the languages:

Theorem 15.2.1 *For any context-free language L, there exists a k-reversible grammar G such that $\mathbb{L}(G) = L$.*

Proof [sketch] The above result is already true even for fixed $k = 0$. One can transform any context-free grammar into a 0-reversible one, even if the transformation process can be costly. It should be noted that the above theorem says nothing about sizes. The corresponding grammar can in fact be of exponential size in the size of the original one. □

15.2.4 The algorithm

The first step consists in building from a sample of unlabelled trees an initial grammar. Basically it consists in converting the unlabelled trees into

Algorithm 15.1: INITIALISE-K-REV-CF.

```
Data: A positive sample of unlabelled strings $S_+$, $k \in \mathbb{N}$
Result: A grammar $G = (\Sigma, V, R, N_1)$ such that
        $\mathbb{L}(G) = \bigcup_{t \in S_+} \text{Frontier}(t)$
$V \leftarrow \emptyset$;
for $t \in S_+$ do
    $t(\lambda) \leftarrow N_1$;
    for $u \in \text{Dom}(t)$ do
        if $t(u) =$ '?' then $t(u) \leftarrow N_t^u$; $V \leftarrow V \cup \{N_t^u\}$
    end
    for $u \in \text{Dom}(t)$ do
        if $u1 \in \text{Dom}(t)$ then            /* $u$ is an internal node */
            $m \leftarrow \max\{i \in \mathbb{N} : ui \in \text{Dom}(t)\}$;
            $R \leftarrow R \cup \{t(u) \rightarrow t(u1) \dots t(um)\}$
        end
    end
end
return $G$
```

derivation trees where a unique non-terminal (the axiom) is used to label every root of the trees in sample S_+. All other internal nodes are labelled by non-terminals that are used exactly once. Algorithm K-REV-CF (15.2) first calls Algorithm INITIALISE-K-REV-CF (15.1) and then uses the labelled trees to merge the non-terminals until a k-reversible grammar is obtained.

Algorithm 15.2: K-REV-CF.

```
Data: A positive sample of unlabelled strings $S_+$, $k \in \mathbb{N}$
Result: A $k$ reversible grammar $G = (\Sigma, V, R, N_1)$
INITIALISE-K-REV-CF($S_+$, $k$);
while $G$ not reset-free and $G$ not invertible do
    if $\exists (N_l \rightarrow \alpha N_i \beta) \in R \wedge (N_l \rightarrow \alpha N_j \beta) \in R$
    then MERGE($N_i, N_j$);                      /* not reset-free */
    if $\exists i, j \in [|V|], \exists \alpha \in (\Sigma \cup V)^* : (N_i \rightarrow \alpha) \in R \wedge$
    $(N_j \rightarrow \alpha) \in R \wedge k\text{-context}(S_+, N_i) \cap k\text{-contexts}(S_+, N_j) \neq \emptyset$
    then MERGE($N_i, N_j$);                      /* not invertible */
end
return $G$
```

In Algorithm K-REV-CF (15.2), the MERGE function is very simple: It consists in taking two non-terminals and merging them into just one. All the occurrences of each non-terminal in all the rules of the grammar are then replaced by the new variable.

15.2.5 Running the algorithm

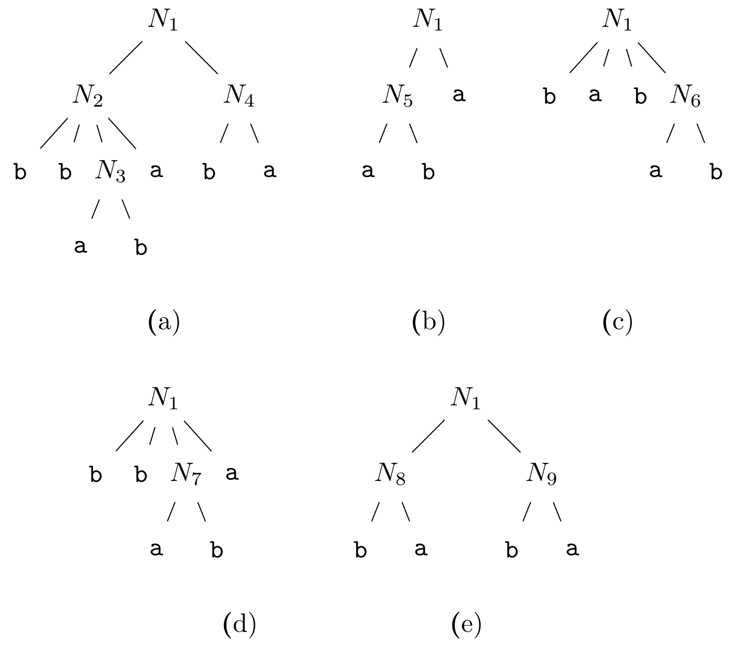


Fig. 15.3. After running Algorithm INITIALISE-K-REV-CF.

Consider the learning sample containing trees:

- ?(?(b b ?(a b)a)?(b a))
- ?(?(a b)a)
- ?(b a b ?(a b))
- ?(b b ?(a b)a)
- ?(?(b a)?(b a))

The first step (running Algorithm 15.1) leads to renaming the nodes labelled by '?':

- N_1(N_2(b b N_3(a b)a)N_4(b a))
- N_1(N_5(a b)a)
- N_1(b a b N_6(a b))

- N_1(b b N_7(a b)a)
- N_1(N_8(b a)N_9(b a))

The corresponding trees are represented in Figure 15.3

There are six 1-contexts, shown in Figure 15.4(a–f).

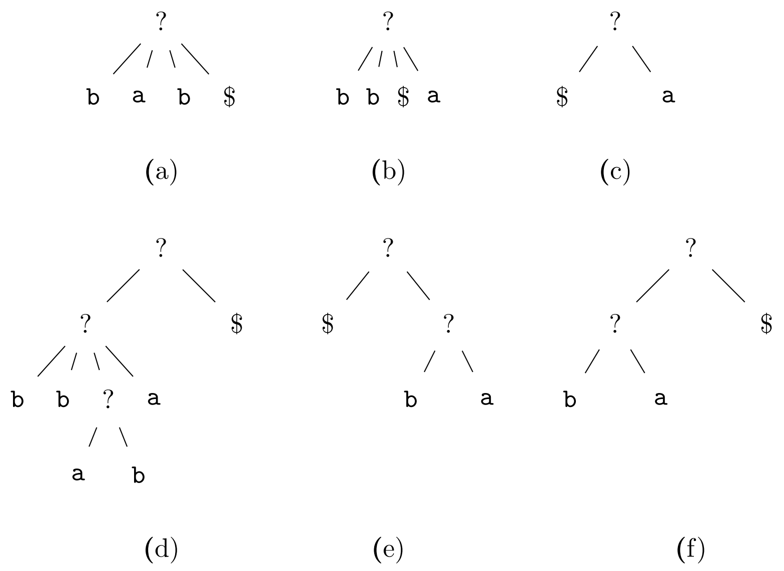


Fig. 15.4. 1-contexts.

With each non-terminal is associated its k-contexts. Here, and with k=1,

- 1-contexts(N_1)= $\emptyset$
- 1-contexts(N_2)=$\{$?($?(b a))$\}$ (1-context (e))
- 1-contexts(N_3)=$\{$?(b b $ a)$\}$(1-context (b))
- 1-contexts(N_4)=$\{$?(?(b b ?(a b)a)$)$\}$(1-context (d))
- 1-contexts(N_5)=$\{$?($ a)$\}$ (1-context (c))
- 1-contexts(N_6)=$\{$?(b a b $)$\}$ (1-context (a))
- 1-contexts(N_7)=$\{$?(b b $ a)$\}$ (1-context (b))
- 1-contexts(N_8)=$\{$?($?(b a))$\}$ (1-context (e))
- 1-contexts(N_9)=$\{$?(?(b a)$)$\}$ (1-context (f))

Now suppose we are running the Algorithm 15.2 with $k = 1$.

- The initial grammar is
  $N_1 \rightarrow N_2N_4 + N_5\texttt{a} + \texttt{bab}N_6 + \texttt{bb}N_7\texttt{a} + N_8N_9$; $N_2 \rightarrow \texttt{bb}N_3\texttt{a}$; $N_3 \rightarrow \texttt{ab}$; $N_4 \rightarrow \texttt{ba}$; $N_5 \rightarrow \texttt{ab}$; $N_6 \rightarrow \texttt{ab}$; $N_7 \rightarrow \texttt{ab}$; $N_8 \rightarrow \texttt{ba}$; $N_9 \rightarrow \texttt{ba}$.

- Since we have rules $N_3 \rightarrow \texttt{ab}$ and $N_7 \rightarrow \texttt{ab}$, and N_3 and N_7 share a common 1-context, the grammar is not reset-free, so N_3 and N_7 are merged (into N_3). At this point our running grammar is
  $N_1 \rightarrow N_2N_4 + N_5\texttt{a} + \texttt{bab}N_6 + \texttt{bb}N_3\texttt{a} + N_8N_9$; $N_2 \rightarrow \texttt{bb}N_3\texttt{a}$; $N_3 \rightarrow \texttt{ab}$; $N_4 \rightarrow \texttt{ba}$; $N_5 \rightarrow \texttt{ab}$; $N_6 \rightarrow \texttt{ab}$; $N_8 \rightarrow \texttt{ba}$; $N_9 \rightarrow \texttt{ba}$.
- The algorithm then discovers that for N_1 and N_2 the invertibility condition doesn't hold, so they are merged resulting in grammar
  $N_1 \rightarrow N_1N_4 + N_5\texttt{a} + \texttt{bab}N_6 + \texttt{bb}N_3\texttt{a} + N_8N_9$; $N_3 \rightarrow \texttt{ab}$; $N_4 \rightarrow \texttt{ba}$; $N_5 \rightarrow \texttt{ab}$; $N_6 \rightarrow \texttt{ab}$; $N_8 \rightarrow \texttt{ba}$; $N_9 \rightarrow \texttt{ba}$.
- At this point the algorithms halts. One can notice that the grammar can be simplified without modifying the language.

15.2.6 Why the algorithm works

The complexity of the algorithm is clearly polynomial in the size of the learning sample. Moreover,

Theorem 15.2.2 *Algorithm 15.1 identifies context-free grammars in the limit from structured text and admits polynomial characteristic samples.*

We do not give the proof here. As usual, the tree notations make things extremely cumbersome. But some of the key properties are as follow:

Properties 15.2.3

- *The order in which the merges are done doesn't matter. The resulting grammar is the same.*
- *Complexity is polynomial (for fixed k) with the size of the sample.*
- *The algorithm admits polynomial characteristic samples.*

15.3 Constructive rewriting systems

An altogether different way of generating a language is through rewriting systems. It is possible to define special systems by giving a base and a rewrite mechanism, such that $b \in L$, and if $w \in L$ then $R(w) \in L$.

These systems can be learnt from text or from an informant depending on the richness of the class of rewriting systems considered.

15.3.1 The general mechanism

Let Σ be an alphabet, B be a finite subset of $\Sigma^\star$ called the base, and R a set of rules: $(\Sigma^\star)^n \rightarrow \Sigma^\star$ which is some *constructive* function.

We expect that a certain number of properties hold. Informally,

- the smallest string(s) in L should be in B;
- from two strings u and v such that $R(u) = v$ one should be able to deduce something about R;
- if the absence in L of such or such string is needed to deduce rules from R then an informant will be needed.

Obviously other issues are raised here that we have seen in previous sections: when attempting to learn a rule, this rule should not be masked by a different rule that is somehow learnt before. The question behind this remark is the one of the existence of a normal form.

It should be noticed that this type of mechanisms avoids the difficult question of non-linearity. The difference of size between two positive examples (strings) u and v, such that v is obtained by applying R once to v is going to be small.

15.3.2 Pure grammars

A typical case of learning constructive rewriting systems concerns inferring *pure grammars* from text. Pure grammars are basically context-free grammars where there is just one alphabet: The non-terminal and terminal symbols are interchangeable.

Definition 15.3.1 (Pure grammars) *A* ***pure grammar*** $G = (\Sigma, R, u)$ *is a triple where* Σ *is an alphabet,* $R \subset \Sigma \times \Sigma^{\star}$ *is the set of rules and* u *is a string from* $\Sigma^{\star}$ *called the axiom.*

Derivation is expressed as in usual context-free grammars. The only difference is that the set of variables and the set of terminal symbols coincide.

Example 15.3.1 *Let* $G = (\Sigma, R, \mathtt{b})$ *with* $\Sigma = \{\mathtt{a}, \mathtt{b}\}$*,* $R = \{\mathtt{b} \rightarrow \mathtt{abb}\}$ *and the axiom is* `b`*. The smallest strings in* $\mathbb{L}(G)$ *are* `b`*,* `abb`*,* `aabbb`*,* `ababb`*. It is easy to see that* $\mathbb{L}(G)$ *is the Lukasiewicz language.*

The fact that there is only one alphabet means that one the long term, the different strings involved in a derivation will appear in the sample. This (avoiding the curse of expansiveness) allows learning pure grammars to become feasible, even from text. Some restrictions to the class of grammars have nevertheless to be added in order to obtain stronger results, like those involving polynomial bounds.

Definition 15.3.2
A pure grammar $G = (\Sigma, R, u)$ is ***monogenic*** *if*$_{def}$ $u \overset{*}{\Longrightarrow} w \rightarrow w'$ *means that there are unique strings v_1 and v_2 such that $w = v_1 x v_2$, $w' = v_1 y v_2$ and $(xy) \in R$.*
A pure grammar G is ***deterministic*** *if*$_{def}$ *for each symbol a in Σ there is at most one production with a on the left hand side.*
*A pure grammar G is k-**uniform** if*$_{def}$ *all rules (l, r) in R have $|r| = k$.*
A language is pure if there is a pure grammar that generates it. It is deterministic if there is a pure deterministic grammar that generates it. And it is k-uniform if there is a pure k-uniform grammar that generates it.
Let us denote, for an alphabet Σ, by $\mathcal{PURE}(\Sigma)$, $\mathcal{PURE} - \mathcal{DET}(\Sigma)$ and $\mathcal{PURE}$-k-$\mathcal{UNIFORM}(\Sigma)$ the classes respectively of pure, pure deterministic and k-uniform languages over Σ.

Example 15.3.2 *The Lukasiewicz language, if we consider the pure grammar with unique rule* $\mathtt{b} \rightarrow \mathtt{abb}$*, is clearly monogenic and deterministic. It also is 3-uniform, trivially.*

Theorem 15.3.1 *The class $\mathcal{PURE}(\Sigma)$ of all pure languages over alphabet Σ is not identifiable in the limit from text.*

Proof It is easy to notice that with any non-empty finite language L over Σ we can associate a pure grammar of the form $G = (\Sigma \cup \{\mathtt{a}\}, R, \mathtt{a})$, with as many rules $(\mathtt{a}, w)$ as there are strings w in L. Notice that symbol $\mathtt{a}$ does not belong to Σ. In this case we have $L = \mathbb{L}(G) \setminus \{\mathtt{a}\}$. And since one can also generate infinite languages the theorem follows easily using Gold's results (Theorem 7.2.3, page 173). □

Theorem 15.3.2 *The class $\mathcal{PURE}$-k-$\mathcal{UNIFORM}(\Sigma)$ of all k-uniform pure languages over alphabet Σ is identifiable in the limit from text.*

Proof We provide a non-constructive proof. Finding the axiom is easy (the smallest string) and so is finding the k (*i.e.* by looking at the differences between the lengths of the strings). Then since the number of possible rules is finite, the class has therefore finite thickness and is learnable from text. □

The above result does not give us directly an algorithm for learning (such an algorithm is to be built in Exercise 15.9. To give a flavour of the type of algorithmics involved in this setting, let us run an intuitive version of the intended algorithm. Given a learning sample S_+, we can build a pure grammar as follows:

Suppose the learning data is $\{$`a`, `bab`, `cac`, `bccaccb`, `cbcacbc`$\}$. The axiom is found immediately and is `a` since it is the shortest string. k is necessarily 3. Then `bab` is chosen and is obtained from the axiom by applying rule `a` $\rightarrow_R$ `bab`. The sample is simplified and is now $\{$`cac`, `bccaccb`, `cbcacbc`$\}$. Rule `a` $\rightarrow_R$ `cac` is invented to cope with `cac`. The set of rules is now capable of generating the entire sample.

15.4 Reducing rewriting systems

An alternative to using context-free grammars, which are naturally expanding (starting with the axiom) is to use reducing rewriting systems. The idea is that the rewriting system should eventually halt, so, in some sense, the left-hand side of the rules should be larger than the right-hand sides. With just the length, this is not too difficult, but the class of languages is then of little interest.

In order to study a class containing all the regular languages, but also some others, we introduce delimited string-rewriting systems (SRS). This class, since it contains all the regular languages, will require more than text to be learnable. We therefore study the learnability of this class from an informant.

The rules of delimited string-rewriting systems allow to replace substrings in strings. There are variants where variables are allowed, but these usually give rise to extremely powerful classes of languages, so for grammatical inference purposes we concentrate on simple variable free rewriting systems.

15.4.1 Strings, terms and rewriting systems

Let us introduce two new symbols \$ and £ that do not belong to the alphabet Σ and will respectively mark the beginning and the end of each string. The languages we are concerned with are thus subsets of $\$\Sigma^*£$. As for the rewrite rules, they will be made of pairs of ***terms*** *partially* marked; a term is a string over alphabet $\{\$, £\} \cup \Sigma$. Such strings have the restriction that the symbol \$ may only appear in first position whereas symbol £ may only appear in last position. Each term therefore belongs to $(\lambda + \$)\,\Sigma^\star(\lambda + £)$. We denote by $\mathbf{T}(\Sigma)$ this set.

Formally, $\mathbf{T}(\Sigma) = \$\,\Sigma^\star\,£ \cup \$\,\Sigma^\star \cup \Sigma^\star\,£ \cup \Sigma^\star = (\$ + \lambda)\,\Sigma^\star(£ + \lambda)$.

The forms of the terms will constrain their use either to the beginning, or to the end, or to the middle, or even to the string taken as a whole.

Terms in $\mathbf{T}(\Sigma)$ can be of one of the following types:

- (Type 1.) $w \in \Sigma^\star$ (used to denote substrings) or
- (Type 2.) $w \in \$\,\Sigma^\star$ (used to denote prefixes) or
- (Type 3.) $w \in \Sigma^\star\,£$ (used to denote suffixes) or
- (Type 4.) $w \in \$\,\Sigma^\star\,£$ (used to denote whole strings).

Given a string w in $\mathbf{T}(\Sigma)$, the ***root*** of w is the string $\$^{-1}w£^{-1}$, $\$^{-1}w$, $w£^{-1}$ and w, respectively.

We define a specific order relation over $\mathbf{T}(\Sigma)$:

$$u <_{\text{DSRS}} v \quad \textit{if}_{\textit{def}} \quad \text{root}(u) <_{\textit{lex-length}} \text{root}(v) \vee \big[\text{root}(u) = \text{root}(v) \wedge \text{type}(u) < \text{type}(v)\big]$$

Example 15.4.1 $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. *then* $\$\mathtt{a}£$,$\$\mathtt{aab}£$ *and* $\$£$ *are strings in* $\$\Sigma^*£$. aa, $\$\mathtt{b}\ £$ *and* $\$syb\mathtt{aa}£$ *are terms (elements of* $\boldsymbol{T}(\Sigma)$*), of respective types 1, 2 3 and 4. The root of both* $\$\mathtt{aab}£$ *and* $\$\mathtt{aab}£$ *is* aab*.*

Finally, we have $\mathtt{ab} <_{\text{DSRS}} \$\mathtt{ab} <_{\text{DSRS}} \mathtt{ab}£ <_{\text{DSRS}} \$\mathtt{ab}£ <_{\text{DSRS}} \mathtt{ba}$.

We can now define the rewriting systems we are considering:

Definition 15.4.1 (Delimited string-rewriting system)

- *A* rewrite rule ρ *is an ordered pair of terms* $\rho = (l, r)$*, generally written as* $\rho = l \vdash r$*.* l *is called the left-hand side of* R *and* r *its right-hand side.*
- *We say that* $\rho = l \vdash r$ *is a* delimited rewrite rule $\textit{if}_{\textit{def}}$ l *and* r *are of the same type.*
- *By a* delimited string-rewriting system *(*DSRS*), we mean any finite set* $\mathcal{R}$ *of delimited rewrite rules.*

The order $<_{\text{DSRS}}$ extends to rules: $(l_1, r_1) <_{\text{DSRS}} (l_2, r_2)$ $\textit{if}_{\textit{def}}$ $l_1 <_{\text{DSRS}} l_2 \vee \big[l_1 = l_2 \wedge r_1 <_{\textit{lex-length}} r_2\big]$.

A system is ***deterministic*** $\textit{if}_{\textit{def}}$ no two rules share a common left-hand side.

Given a system $\mathcal{R}$ and a string w, there may be several rules that seem to be applicable upon w. Nevertheless only one rule is eligible. This is the rule having smallest left-hand side, for the order $<_{\text{DSRS}}$. Formally, a rule $\rho = l \vdash r$ is eligible for string w if

$$\exists u, v \in \mathbf{T}(\Sigma) : \ w = ulv$$
$$\forall u'l'v' : \exists \rho' = l' \vdash r', l <_{\text{DSRS}} l'$$

One should note that a same rule might be eligible in different places. We systematically privilege the left-most position.

Example 15.4.2 *With system* $(\{\texttt{ab} \vdash \lambda;\ \texttt{ba} \vdash \lambda\}, \$\pounds)$*, if we consider string* $\$\texttt{ababbaba}\pounds$*, both rules* $\texttt{ab} \vdash \lambda$ *and* $\texttt{ab} \vdash \lambda$ *can be used, and each in various positions. The eligible rule is the first and it must be used in the leftmost position, therefore:*

$$\$\underline{\texttt{ab}}\texttt{abbaba}\pounds \vdash \$\texttt{abbaba}\pounds$$

Given a Dsrs $\mathcal{R}$ and two strings $w_1, w_2 \in \mathbf{T}(\Sigma)$, we say that w_1 ***rewrites in one step into*** w_2, written $w_1 \vdash_{\mathcal{R}} w_2$ or simply $w_1 \vdash w_2$, $\mathit{if}_{\mathit{def}}$ there exist an eligible rule $(l \vdash r) \in \mathcal{R}$ for w_1, and there are two strings $u, v \in (\lambda + \$)\,\Sigma^\star(\lambda + \pounds)$ such that $w_1 = ulv$ and $w_2 = urv$, and furthermore u is shortest for this rule.

A string w is ***reducible*** $\mathit{if}_{\mathit{def}}$ there exists w' such that $w \vdash w'$, and ***irreducible*** otherwise.

Example 15.4.3 *Again for system* $(\{\texttt{ab} \vdash \lambda;\ \texttt{ba} \vdash \lambda\}, \$\pounds)$*, string* $\$\texttt{ababa}\pounds$ *is reducible whereas* $\$\texttt{bbb}\pounds$ *is not.*

The constraints on \$ and £ are such that these symbols always remain in their typical positions during the reductions at the beginning and the end of the string.

Let $\vdash^*_{\mathcal{R}}$ (or simply $\vdash^*$) denote the reflexive and transitive closure of $\vdash_{\mathcal{R}}$. We say that w_1 ***reduces to*** w_2 or that w_2 ***is derivable from*** w_1 $\mathit{if}_{\mathit{def}}$ $w_1 \vdash^*_{\mathcal{R}} w_2$.

Definition 15.4.2 (Language induced by a Dsrs) *Given a* Dsrs $\mathcal{R}$ *and an irreducible string* $e \in \Sigma^\star$*, we define the language* $\mathbb{L}(\mathcal{R}, e)$ *as the set of strings that reduce to* e *using the rules of* $\mathcal{R}$*:*

$$\mathbb{L}(\mathcal{R}, e) = \{w \in \Sigma^\star : \$w\pounds \vdash^*_{\mathcal{R}} \$e\pounds\}.$$

Deciding whether a string w belongs to a language $\mathbb{L}(\mathcal{R}, e)$ or not consists in trying to obtain e from w by a rewriting derivation. We will denote by $\textsc{Apply}_{\mathcal{R}}(\mathcal{R}, w)$ the string obtained by applying the different rules in $\mathcal{R}$ until no more rule can be applied. We extend the notation to a set of strings:

$$\textsc{Apply}_{\mathcal{R}}(\mathcal{R}, S) = \{\mathcal{R}(w) :\ w \in S\}.$$

Example 15.4.4 *This time let us consider the Lukasiewicz language which* *can be represented by the system* $(\{\texttt{abb} \vdash \texttt{b}\}, \$\texttt{b}\pounds)$. *But there is an alternative system:* $(\{\$\texttt{ab} \vdash \$;\ \texttt{aab} \vdash \texttt{a}\}, \$\texttt{b}\pounds)$.

Let us check that for either system string aababbabb *can be obtained as an element of the language:*

$$\$\texttt{aab}\underline{\texttt{abb}}\texttt{abb}\pounds \vdash \$\texttt{a}\underline{\texttt{abb}}\texttt{abb}\pounds \vdash \$\texttt{ab}\underline{\texttt{abb}}\pounds \vdash \$\underline{\texttt{abb}}\pounds \vdash \$\texttt{b}\pounds$$

$$\$\underline{\texttt{aab}}\texttt{abbabb}\pounds \vdash \$\underline{\texttt{aab}}\texttt{babb}\pounds \vdash \underline{\$\texttt{ab}}\texttt{abb}\pounds \vdash \underline{\$\texttt{ab}}\texttt{b}\pounds \vdash \$\texttt{b}\pounds$$

Let $|\mathcal{R}|$ be the number of rules of $\mathcal{R}$ and $\|\mathcal{R}\|$ be the sum of the lengths of the strings $\mathcal{R}$ is involved in: $\|\mathcal{R}\| = \sum_{(l \vdash r) \in \mathcal{R}} |lr|$.

Here are some examples of DSRS and associated languages:

Example 15.4.5 *Let* $\Sigma = \{\texttt{a}, \texttt{b}\}$.

- $\mathbb{L}(\{\texttt{ab} \vdash \lambda\}, \lambda)$ *is the Dyck language. Indeed, since this single rule erases substring* ab, *we get the following example of a derivation:*

$$\$\texttt{aabb}\underline{\texttt{ab}}\pounds \vdash \$\texttt{a}\underline{\texttt{ab}}\texttt{b}\pounds \vdash \$\underline{\texttt{ab}}\pounds \vdash \$\pounds$$

- $\mathbb{L}(\{\texttt{ab} \vdash \lambda; \texttt{ba} \vdash \lambda\}, \lambda)$ *is the language* $\{w \in \Sigma^\star : |w|_\texttt{a} = |w|_\texttt{b}\}$, *because every rewriting step erases one* a *and one* b.
- $\mathbb{L}(\{\texttt{aabb} \vdash \texttt{ab}; \$\texttt{ab}\pounds \vdash \$\pounds\}, \lambda) = \{\texttt{a}^n\texttt{b}^n : n \geq 0\}$. *For instance,*

$$\$\texttt{aa}\underline{\texttt{aabb}}\texttt{bb}\pounds \vdash \$\texttt{a}\underline{\texttt{aabb}}\texttt{b}\pounds \vdash \$\underline{\texttt{aabb}}\pounds \vdash \$\underline{\texttt{ab}}\pounds \vdash \$\pounds$$

Notice that the rule $\$\texttt{ab}\pounds \vdash \$\pounds$ *is necessary for deriving* λ *(last derivation step).*
- $\mathcal{L}(\{\$\texttt{ab} \vdash \$\}, \lambda)$ *is the regular language* $(\texttt{ab})^*$. *Indeed,*

$$\underline{\$\texttt{ab}}\texttt{abab}\pounds \vdash \underline{\$\texttt{ab}}\texttt{ab}\pounds \vdash \underline{\$\texttt{ab}}\pounds \vdash \$\pounds$$

We claim that given any regular language L there is a system $\mathcal{R}$ such that $\mathbb{L}(\mathcal{R}) = L$. The fact that the rules can only be applied in a left first fashion is a crucial reason for this. One can associate with every state in the DFA rules rewriting to the shortest string that reaches the state for the lex-length order.

15.4.2 Algorithm LARS

The algorithm we present (Learning Algorithm for Rewriting Systems) generates the possible rules between those that can be applied over the positive data, tries using them and keeps them if they do not create inconsistency (using the negative sample for that). Algorithm LARS (15.4) calls function NEWRULE (15.3) which generates the next possible rule to be checked.

For this, one should choose *useful* rules, *i.e.* those that can be applied

```
Algorithm 15.3: NEWRULE.
  Input: $S_+$, rule $\rho$
  Output: a new rule $(l, r)$
  returns the first rule for $<_{\text{DSRS}}$ after $\rho$ such that $\Sigma^\star\, l\, \Sigma^\star \cap L \neq \emptyset$
```

on at least one string from S_+. One might also consider useful a rule that allows to diminish the size of the set S_+: a rule which, when added, has the property that two different strings rewrite into an identical string. The goal of usefulness is to avoid an exponential explosion in the number of rules to be checked. Function CONSISTENT (15.5) checks that by adding the new rule to the system, this, one does not rewrite a positive example and a negative example into a same string.

```
Algorithm 15.4: LARS.
  Input: $S_+, S_-$
  Output: $\mathcal{R}$
  $\mathcal{R} \leftarrow \emptyset$;
  $\rho \leftarrow$ NEWRULE$(S_+, (\lambda, \lambda))$;
  while $|S_+| > 1$ do
      if LARS-CONSISTENT$(S_+, S_-, \mathcal{R} \cup \{\rho\})$ then
          $\mathcal{R} \leftarrow \mathcal{R} \cup \{\rho\}$;
          $S_+ \leftarrow$ APPLY$_{\mathcal{R}}(\mathcal{R}, S_+)$;
          $S_- \leftarrow$ APPLY$_{\mathcal{R}}(\mathcal{R}, S_-)$
      end
      $\rho \leftarrow$ NEWRULE$(S_+, r)$
  end
  $w \leftarrow \min(S_+)$;
  return $\langle \mathcal{R}, w \rangle$
```

The goal is to be able to learn with LARS any DSRS. The simplified version proposed here can be used as basis for that, and does identify in the limit any DSRS. But, a formal study of the qualities of the algorithm (as far as mind changes, characteristic samples) is beyond the scope of this book.

15.4.3 Running LARS

We give an example run of algorithm LARS on the following sample: $S_+ = \{\texttt{abb}, \texttt{b}, \texttt{aabbb}, \texttt{abababb}\}$, and $S_- = \{\lambda, \texttt{a}, \texttt{ab}, \texttt{ba}, \texttt{bab}, \texttt{abbabb}\}$. LARS tries the following rules:

Algorithm 15.5: LARS-CONSISTENT.

```
Input: $S_+$,$S_-$,$\mathcal{R}$
Output: a boolean indicating if the current system is consistent with
        ($S_+$,$S_-$)
if $\exists x \in S_+, y \in S_- : \text{APPLY}_{\mathcal{R}}(\mathcal{R}, x) = \text{APPLY}_{\mathcal{R}}(\mathcal{R}, y)$ then
    return false
else
    return true
end
```

- the smallest rule for the order proposed is $\mathtt{a} \vdash \lambda$, which fails because `ab` and `ba` would both rewrite, using this rule, into `b`; But $\mathtt{ab} \in S_-$ and $\mathtt{b} \in S_+$;
- the next rule is $\$\mathtt{a} \vdash \$$, which fails because `ab` would again rewrite into `b`;
- no other rule based on the pair $(\mathtt{a}, \lambda)$ is tried, because the rule would apply to no string in S_+. So the next rule is $\mathtt{b} \vdash \lambda$, and fails because `b` rewrites into λ;
- again $\$\mathtt{b} \vdash \$$, $\mathtt{b}£ \vdash £$ and $\$\mathtt{b}£ \vdash \$£$ all fail because `b` would rewrite into λ;
- $\mathtt{b} \vdash \mathtt{a}$, fails because `b` rewrites into `a`;
- $\$\mathtt{b} \vdash \$\mathtt{a}$, fails because `b` rewrites into `a`;
- $\mathtt{ab} \vdash \lambda$ is the next rule to be generated; it is rejected because `bab` would rewrite into `b`;
- $\$\mathtt{ab} \vdash \$$ is considered next and is this time accepted. The samples are parsed using this rule and are updated to $S_+ = \{\mathtt{b}, \mathtt{aabbb}\}$ and $S_- = \{\lambda, \mathtt{a}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \mathtt{bab}\}$.
- Rules with left-hand side `ba`, `bb`, `aaa` and variants are not analysed, because they cannot apply to S_+.
- The next rule to be checked (and rejected) is $\mathtt{aab} \vdash \lambda$ but then `aabbb` would rewrite into `bb` which (now) belongs to S_-.
- Finally, $\mathtt{aab} \vdash \mathtt{a}$ is checked, causes no problem, used to parse the samples obtaining $S_+ = \{\mathtt{b}\}$ and $S_- = \{\lambda, \mathtt{a}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \mathtt{bab}\}$. The algorithm halts with system $(\{\$\mathtt{ab} \vdash \$; \mathtt{aab} \vdash \mathtt{a}\}, \$\mathtt{b}£)$.

15.5 Some heuristics

The theoretical results about context-free grammar learning are essentially negative and state that no efficient algorithm can be found that can learn in polynomial conditions the entire class of the context-free languages, whether we want to learn with queries, from characteristic samples or in the PAC setting.

This has motivated the introduction of many specific heuristic techniques. It would be difficult to show them all, but some ideas are also given in Chapter 14. In order to present different approaches we present here only two lines of work. The first corresponds to the use of minimum length encoding, the second to an algorithm that has not really been used in grammatical inference, but rather for compression tasks.

15.5.1 Minimum description length

We presented the minimum description length principle in a general way, but also for the particular problem of learning DFA, in Section 14.4. The principle basically states that the size of an encoding should be computed as the sum of the size of the model (here the grammar) and the size of the object (here the strings) when encoded by the grammar.

We present the ideas behind an algorithm called GRIDS whose goal is to learn a context-free grammar from text.

The starting point is a grammar that generates exactly the sample, with exactly one rule $N_1 \rightarrow w$ per string w in the sample.

Then iteratively, the idea is to try to better the score of the current grammar by trying one of the two following operations:

- A ***creating*** operation takes a substring of terminals and non-terminals, invents a new non-terminal that derives into the string, and replaces the string by the non-terminal in all the rules. This operator does not modify the generated language.
- A ***merging*** operation takes two non-terminals and merges them into one; This operator can generalise the generated language.

Example 15.5.1 *Suppose the current grammar contains:*
$N_1 \rightarrow \mathtt{a}N_2\mathtt{ba}$
$N_2 \rightarrow N_3\mathtt{ba}$

Then the merging operation could replace these rules by:
$N_1 \rightarrow \mathtt{a}N_2\mathtt{ba}$
$N_2 \rightarrow N_2\mathtt{ba}$

whereas the creating operation might replace the rules by:
$N_1 \rightarrow \mathtt{a}N_2N_4$

$N_2 \rightarrow N_3N_4$
$N_4 \rightarrow \texttt{ba}$

Iteratively each possible merge/creation is tested and a score is computed: The score takes into account the size of the obtained grammar and the size of the data set, when generated by the grammar. The best score that betters the current score indicates which operation, if any, determines the new grammar.

The way the score is computed is important: It has to count the number of bits needed to optimally encode the candidate grammar, and the number of bits needed to encode (also optimally) a derivation of the text for that grammar. This should therefore take into account that a given non-terminal can have various derivations or not. Some initial ideas towards using MDL can be found in Section 14.4, for the case of automata.

15.5.2 Grammar induction as compression

The idea of defining an operator over grammars that allows to transform one grammar into another has also been used by an algorithm that, although not of grammar induction (there is no generalisation) is close to the ideas presented here. Moreover, the structure found by this algorithm can be used as a first step towards inferring a context-free grammar. The algorithm is called SEQUITUR and is used to compress (without loss) a text by finding the repetitions and structure of this text. The idea is to find a grammar that exactly generates one string, *i.e.* the text that should be compressed. If the grammar encodes the text in less symbols than the text length, then a compression is performed.

Two conditions have to be followed by the resulting grammar:

- each rule (but for the 'initial' one) of the grammar has to be used at least twice;
- there can be no repeated substring of length more than one.

Example 15.5.2 *The following grammar is accepted. Notice that each non-terminal is used just once in a left-hand side of a rule, and that all non-terminals, with the exception of N_1 are used at least twice.*

- $N_1 \rightarrow N_2N_3N_4N_2$
- $N_2 \rightarrow \texttt{a}N_3\texttt{b}N_5$
- $N_3 \rightarrow N_5\texttt{a}N_4$
- $N_4 \rightarrow \texttt{ba}N_5$

- $N_5 \rightarrow \texttt{ab}$

The algorithm starts with just one rule, called the N_1 rule, which is $N_1 \rightarrow \lambda$. SEQUITUR works sequentially by reading one symbol of the text at the time, adds it to the N_1 rule and attempts to transform the running grammar by either:

- using an existing rule,
- creating a new rule,
- deleting some rule that is no longer needed.

Some of these actions may involve new actions taking place.

There is little point in giving the code of SEQUITUR because on one hand the algorithmic ideas are clear from the above explanation, but on the other hand, the implementation details are far beyond the scope of this section, but they are essential in order to keep the algorithm quasi-linear.

Let us instead run SEQUITUR on a small example.

Example 15.5.3 *Suppose the entry string is* `aaababababaab`.

- *The initial grammar is $N_1 \rightarrow \lambda$, corresponding to having read the empty prefix of the entry string.* SEQUITUR *reads the first letter of the input (*`a`*). Current grammar becomes therefore $N_1 \rightarrow \texttt{a}$. Nothing more happens (i.e. the grammar is accepted to far).*
- SEQUITUR *reads next symbol (*`a`*). Total input is now* `a`**`a`**. *The current grammar becomes $N_1 \rightarrow \texttt{aa}$.*
- *Next symbol is read (input is* `aa`**`a`***). The grammar is updated to $N_1 \rightarrow \texttt{aaa}$.*
- *Another symbol is read (input is* `aaa`**`b`***). The grammar is therefore updated to $N_1 \rightarrow \texttt{aaab}$. No transformation of the grammar is so far possible.*
- *Next symbol (*`a`*) is read for total input of* `aaab`**`a`**. *Current grammar becomes $N_1 \rightarrow \texttt{aaaba}$.*
- *The next symbol (*`b`*) is read. Input is now* `aaaba`**`b`**. *Current grammar becomes $N_1 \rightarrow \texttt{aaabab}$, but substring* `ab` *appears twice. So a new non-terminal is introduced and the grammar is $N_1 \rightarrow \texttt{aa}N_2N_2$, $N_2 \rightarrow \texttt{ab}$.*
- *Another symbol is read. Input is now* `aaabab`**`a`**. *Current grammar becomes $N_1 \rightarrow \texttt{aa}N_2N_2\texttt{a}$, $N_2 \rightarrow \texttt{ab}$.*
- *Another symbol is read. Input is now* `aaababa`**`b`**. *Current grammar becomes $N_1 \rightarrow \texttt{aa}N_2N_2\texttt{ab}$, $N_2 \rightarrow \texttt{ab}$, but rule $N_2 \rightarrow \texttt{ab}$ can be used, so we have $N_1 \rightarrow \texttt{aa}N_2N_2N_2$, $N_2 \rightarrow \texttt{ab}$.*

- *Another symbol is read. Input is now* aaababab**a**. *Current grammar becomes* $N_1 \rightarrow \mathtt{aa}N_2N_2N_2\mathtt{a},\ N_2 \rightarrow \mathtt{ab}$.
- *Another symbol is read. Input is now* aaabababa**a**. *Current grammar becomes* $N_1 \rightarrow \mathtt{aa}N_2N_2N_2\mathtt{aa},\ N_2 \rightarrow \mathtt{ab}$.
- *Another symbol is read. Input is now* aaabababaa**b**. *Current grammar becomes* $N_1 \rightarrow \mathtt{aa}N_2N_2N_2\mathtt{aab},\ N_2 \rightarrow \mathtt{ab}$. *We can now apply rule* $N_2 \rightarrow \mathtt{ab}$ *and obtain* $N_1 \rightarrow \mathtt{aa}N_2N_2N_2\mathtt{a}N_2,\ N_1 \rightarrow \mathtt{ab}$ *but now* $\mathtt{a}N_2$ *appears twice so we introduce* $N_3 \rightarrow \mathtt{a}N_2$ *and the grammar is* $N_1 \rightarrow \mathtt{a}N_3N_2N_2N_3,\ N_2 \rightarrow \mathtt{ab},\ N_3 \rightarrow \mathtt{a}N_2$.
- *As the string is entirely parsed, the algorithm halts.*

15.6 Exercises

15.1 Is the following grammar G very simple deterministic?
$G = \langle \Sigma, V, R, N_1\rangle$, $\Sigma = \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}$, $V = \{N_1, N_2, N_3, N_4\}$, $R = \{N_1 \rightarrow \mathtt{b}N_2 + \mathtt{a}N_2N_4;\ N_2 \rightarrow \mathtt{c}N_2N_2 + \mathtt{d};\ N_3 \rightarrow \mathtt{a}N_3\mathtt{b}; N_4 \rightarrow \mathtt{ab}\}$.

15.2 Complete Theorem 15.1.3: Does the proposed algorithm for very simple deterministic grammars have polynomial characteristic samples, does it make a polynomial number of mind changes?

15.3 Consider the learning sample containing trees:

- $N_1(N_2(\mathtt{b\,b}\,N_3(\mathtt{a\,b})\mathtt{a})N_3(\mathtt{b\,a}))$
- $N_1(N_3(\mathtt{a\,b})\mathtt{a})$
- $N_1(\mathtt{b\,a\,b}N_3(\mathtt{a\,b}))$
- $N_1(\mathtt{b\,b}\,N_2(\mathtt{a\,b})\mathtt{a})$
- $N_1(N_3(\mathtt{b\,a})N_2(\mathtt{b\,a}))$

Take k=1 and k=2. What are the k-ancestors of N_3 that we can deduce from the sample? Draw the corresponding k-contexts of N_3.

15.4 Why can we not deduce from Theorem 15.2.1 that context-free grammars are identifiable from text?

15.5 Is the following grammar $G = \langle \Sigma, V, R, N_1\rangle$ k-reversible? For what values of k? $\Sigma = \{\mathtt{a},\mathtt{b}\}$, $V = \{N_1, N_2, N_3, N_4\}$, $R = \{N_1 \rightarrow \mathtt{b}N_2 + \mathtt{a}N_4 + \mathtt{bab};\ N_2 \rightarrow \mathtt{a}N_2\mathtt{b}N_2 + \mathtt{a};\ N_3 \rightarrow \mathtt{a} + \mathtt{b};\ N_4 \rightarrow \mathtt{a}N_3\mathtt{b}N_2 + \mathtt{ab}\}$.

15.6 Find a context-free grammar for which the corresponding 0-reversible grammar is of exponential size.

15.7 Learn a pure grammar (see Section 15.4) from the following sample: $S_+ = \{\mathtt{aba}, \mathtt{a}^3\mathtt{ba}^3, \mathtt{a}^6\mathtt{a}^6\}$. Suppose the grammar is k-uniform.

15.8 Learn a pure grammar from sample $S_+ = \{\mathtt{aba}, \mathtt{a}^3\mathtt{ba}^3, \mathtt{a}^6\mathtt{a}^6\}$. Suppose the grammar is deterministic.

15.9 Write a learning algorithm corresponding to the proof of Theorem 15.3.2. What is its complexity? Prove that it admits polynomial characteristic samples.

15.10 Run algorithm SEQUITUR over the following input texts. What sort of grammatical constructs does SEQUITUR find. Which are the ones it cannot find?

$$\begin{aligned} w_1 &= \texttt{aaaabbbb} \\ w_2 &= \texttt{a}^{256}\texttt{b}^{256} \\ w_3 &= (\texttt{ab})^{256} \end{aligned}$$

15.7 Conclusions of the chapter and further reading

15.7.1 Bibliographical background

The discussion about if natural language is context-free has been going on since the class of context-free languages was invented [Cho57]. For a more grammatical inference flavour, see [BB06, RS07]. The discussion about the specific difficulties of learning context-free grammars relies mostly on work by Rémi Eyraud [Eyr06] and Colin de la Higuera [dlH06a]. There are few negative results corresponding to learning context-free grammars. Dana Angluin conjectured that these were not learnable by using an MAT and the proof intervened in [AK91]. Colin de la Higuera [dlH97] proved that polynomial characteristic sets were of unboundable size.

A first line of research has consisted in limiting the class of context-free grammars to be learnt: Even linear grammars [Tak88], deterministic linear grammars [dlHO02], or very simple grammars [Yok03] have been proved learnable is different settings.

Much work has been done on the problems relating to even linear grammars [Tak88, Tak94, SG94, Mäk96, KMT97, SN98]. Positive results concerning subclasses of even linear languages have been obtained when learning from text [KMT00]. Takashi Yokomori [Yok03] introduced the class of simple deterministic grammars and obtained the different results reported here.

The special class of deterministic linear languages was introduced by Colin de la Higuera and Jose Oncina [dlHO02]. The class was later adapted in order to take into account probabilities [dlHO03].

The algorithm for learning k-reversible grammars is initially due to Yasubumi Sakakibara [Sak92], based on Dana Angluin's algorithm for regular languages [Ang82]. Further work by Jérôme Besombes and Jean-Yves Marion [BM04a] and Tim Oates *et al.* [ODB02] is used in this presentation. The proof of Theorem 15.2.1 is constructive but beyond the scope of this book. It can be found in the above cited papers.

Pure grammars are learnt by Takeshi Koshiba *et al.* [KMT00] whereas the work we describe on the rewriting systems is by Rémi Eyraud *et al.* [EdlHJ06]. In both cases the original ideas and algorithms have been (over) simplified in this Chapter, and many alternative ideas and explanations can be found in the original papers.

Based on the MDL principle, Gerry Wolff [Wol78] introduced an algorithm called GRIDS whose ideas were further investigated by Pat Langley and Sean Stromsten [LS00], and then by George Petasis *et al.* [PPK+04]. Alternatively, the same operators can be used with a genetic algorithm [PPSH04], but the results are not better.

Algorithm SEQUITUR is due to Craig Neville-Manning and Ian Witten [NMW97a]. Let us note that (like in many grammar manipulation programs) it relies on hash tables and other programming devices to enable the algorithm to work in linear time. Experiments were made by SEQUITUR over a variety of sequential files, containing text or music: Compression rates are good, but more importantly, the structure of the text is discovered. If one runs SEQUITUR on special context-free grammars (such as Dyck languages) results are poor: SEQUITUR is good at finding repetitions of patterns, not necessarily at finding complex context-free structures. On the other hand, in our opinion, no algorithm today is good at this task.

The question of the relationship between learning (and specifically grammar learning) and compression can also be discussed. SEQUITUR performs a lossless compression: No generalisation, or loss, takes place. One can argue that other grammar induction techniques also perform a compression of the text (the learning data) into a grammar. But in that case it is expected that the resulting language is not equal to the initial text. Now the questions that arise from these remarks are: Is there a continuum between the SEQUITUR type-of lossless compression techniques and the GRIDS type-of compression with loss techniques? In other words could we tune SEQUITUR to obtain a generalisation of the input text or GRIDS to obtain a grammar equivalent to the initial one? How do we measure the loss due to the generalisation, in such a way as to incorporate it into the learning/compression algorithm?

15.7.2 Some alternative lines of research

If to the idea of simplifying the class of grammars we add that of using queries there are positive results concerning the class of simple deterministic languages. A language is simple deterministic when it can be recognised by a deterministic push-down automaton by empty store, that only uses one state. All languages in this class are thus necessarily deterministic, λ-free and prefix.

Hiroki Ishizaka [Ish95] learns these languages using 2-standard forms: his algorithm makes use of membership queries and extended equivalence queries.

There have been no positive known result relating any form of learning with usual queries and the entire class of context-free languages. It is shown that context-free grammars have *approximate fingerprints* and therefore are not learnable from equivalence queries alone [Ang90], but also that membership queries alone [AK91] are insufficient even if learning in a PAC setting (under typical cryptographic assumptions), and it is conjectured [Ang01] that an MAT is not able to cope with context-free languages.

If one reads proceedings of genetic algorithms or evolutionary computing conferences dating from the seventies or eighties, one will find a number of references concerning grammatical inference. Genetic algorithms require a linear encoding of the solutions (hence here of the grammars) and a careful definition of the genetic operators one wants to use, usually a crossing-over operator and a mutation operator. Is also required some (possibly numerical) measure of the quality of a solution.

The mutation operator takes a grammar and modifies a bit somewhere and returns a new grammar. The crossing-over operator would take two grammars, cut these into two halves and build two new grammars by mixing the halves [Wya94]. One curious direction [KB96] to deal with this issue is to admit that in that case, parts of the string will not be used to encode any more, and would correspond to what is known as *junk* DNA. Other ideas correspond to very specific encodings of the partitions of non-terminals and offer resistant operators [Dup94, SK99]. Yasubumi Sakakibara *et al.* [SK99, SM00] represented the grammars by parsing tables and attempted to learn by a genetic algorithm these tabular representations.

Between the several pragmatic approaches, let us mention the BOISDALE algorithm [SF04] which makes use of special forms of grammars, SYNAPSE [NM05] which is based on parsing, LARS [EdlHJ06] which learns rewriting systems and Alexander Clark's [Cla04] algorithm (which won the 2004 OMPHALOS competition [SCvZ04a]) concentrates on deterministic languages. An alternative is to learn a k-testable tree automaton, and estimate the probabilities of the rules [RJCRC02]. An earlier line of research is based on exhaustive search, either by use of a *version space* approach [VB87], or by using operators such as the *Reynolds cover* [Gio94].

There have also been some positive results concerning learning context-free languages from queries. Identification in the limit is of course trivially possible with the help of strong equivalence queries, through an enumeration process. A more interesting positive result concerns grammars in a special normal form, and when the queries will enable some knowledge about the structure of the grammar and not just the language. The algorithm is a natural extension of Dana Angluin's LSTAR algorithm [Ang87a] and was designed by Yasubumi Sakakibara [Sak90] who learns context-free grammars from structural queries (a query returns the context in which a substring is used).

In the field of computational linguistics, efforts have been made to learn context-free grammars from more informative data, such as trees [Cha96], following theoretical results by Yasubumi Sakakibara [Sak92]. Learning from structured data has been a line followed by many: learning tree automata [KS94, Fer02, HBJ02], or context-free grammars from bracketed data [Sak90] allows to obtain better results, either with queries [Sak92], regular distributions [Kre97, COCR01, RJCRC02], or negative information [GO93]. This has also led to different studies concerning the probability estimation of such grammars [LY90, CRC98].

A totally different direction of research has been followed by authors working with categorial grammars. These are as powerful as context-free grammars. They are favoured by computational linguists who have long been interested in working on grammatical models that do not necessarily fit into Chomsky's hierarchy. Furthermore, their objective is to find suitable models for syntax and semantics to be interlinked, and provide a logic based description language. Key ideas relating such models with the questions of language identification can be found in Makoto Kanazawa's book [Kan98], and discussion relating this to the way children learn language can be found in papers by a variety of authors, as for instance those by Isabelle Tellier [Tel98]. The situation is still unclear, as positive results can only be obtained for special classes of grammars [FN02], whereas, here again, the corresponding combinatorial problems (for instance that of finding the smallest consistent grammar) appear to be intractable [Flo02].

The situation concerning learnability of context-free grammars has evolved since 2003 with renewed interest caused by workshops [dlHAvZO03], and more importantly by the OMPHALOS context-free language learning competition [SCvZ04b], where state of the art techniques were unable to solve even the easiest tasks. The method [Cla07] that obtained best results used a variety of informations about the distributions of the symbols, substitution graphs and context. The approach is mainly empirical and does not provide a convergence theorem.

About mildly context-sensitive languages, we might mention Leo Becerra *et al.* [OABBA06]: These systems can learn languages that are not context-free. But one should be careful: Mildly context-sensitive languages usually do not contain all the context-free. A related approach is through kernels: Semi-linear sets [Abe95] and planar languages [CFWS06] are alternative ways of tackling the problem.

15.7.3 Open problems and possible new lines of research

Work in learning context-free grammars is related to several difficult aspects: language representation issues, decidability questions, tree automata... Yet the subject is obviously very much open with several teams working on the question. Between the most notable open questions these are some that are worth looking into:

(i) In the definition of identification in the limit from polynomial time and data, the size of a set of strings is taken as the number of bits needed to encode the set. If we take, as some authors propose [CGLN05], the size of the set as the number of strings in the sample, then it is not known if context-free grammars admit characteristic samples of polynomial size.

(ii) The question of learning probabilistic context-free grammars is widely open with very few recent results, but no claim that these may be learnable in some way or another. Yet these grammars are of clear importance in fields such as computational biology [JLP01] or linguistics [Cha96]. In a way, these grammars would bridge the gap between learning languages and grammars, as the very definition of a probabilistic context-free language requires a grammar.

(iii) Another question that has been left largely untouched is that of learning pushdown automata. Yet these provide us with mechanisms that would allow to add extra bias: The determinism of the automata, the number of turns, the number of symbols in the stack can be controlled.

(iv) As may be seen in the section about pure grammars, there are several open questions and problems to be solved in that context: What is identifiable and what is not? What about polynomial bounds? The reference paper where these problems are best described is by Takeshi Koshiba *et al.* [KMT00].

(v) A curious problem is that of learning a context-free language L which is the intersection between another context-free language L_C and a regular language L_R, when the context-free language is given [dlH06b].

(vi) Finally, an interesting line of research was proposed in [SM00], where the learning data was partially structured, *i.e.* some brackets were given. Although the purpose of the paper was to prove that the number of generations of the genetic learning algorithm was lower when more structure was given, the more general question of how much structure is needed to learn is of great interest.

(vii) Most importantly, there is a real need for good context-free grammar learning algorithms. There are three such algorithms used in computational linguistics [AvZ04] we have not described here, in part because they rely on too many and complex different algorithmic mechanisms:

- Algorithm Emile, by Pieter Adriaans and Marco Vervoort [AV02] relies on substitutability as chief element: can a string be substituted by another? Is a context equivalent to another? Graph clustering techniques are used to solve these questions.
- Algorithm Abl, by Menno van Zannen [vZ00] (Alignment based learning) relies on aligning the different sentences in order to then associate non-terminals with clusters of substrings.
- Algorithm Adios, by Zach Solan *et al.*[SRHE02] also represents the information by a graph, but aims to find best paths in the graphs.
- The key idea of substitutability (which is essential in the above works) is studied in more detail by Alexander Clark and Rémi Eyraud [CE07]. Two substrings are congruent if they can be replaced one by the other in any context. This allows, by discovering this congruence, to build a grammar. This powerful mechanism enables to assemble context-free grammars, and corresponds to one of the most promising lines of research in the field.