Context-free languages

Bruno Courcelle
Bordeaux I University, courcell@labri.fr

May 2007 (Revised June 2008)

1 Introduction

The *Theory of Formal Languages* deals with sets of sequences of letters. Such a sequence is called a *word* and a set of words is called a *language*. "Formal" means that there is no semantics attached to a word, contrary to the case of *Programming Languages, Logical Languages* and *Natural Languages*.

The following topics are considered in this theory :

1) Finite descriptions of languages : they can be generative (*grammars*, regular expressions), algorithmic (deterministic finite automata, *parsers*), declarative (logical formulas, informal mathematical descriptions).

2) Various types of descriptions can be compared. One wants algorithms transforming a description of some (formalized) type into an *equivalent one* of another type. Kleene's Theorem provides such an algorithm for transforming a generative description by a regular expression into an algorithmic one by a deterministic finite automaton.

3) One wants methods for *proving that a description is correct* with respect to a specification, either formalized or informal.

4) One wants methods (even better *algorithms*) for transforming descriptions: for instance for restricting a described set by a logically specified condition.

5) However, certain important problems have no algorithmic solutions. Others have algorithmic solutions, but with algorithms that take too much time to be really usable.

6) Transformations of words (decoding is a good example) can also be defined in declarative or algorithmic ways. This topic is also an important part of language theory.

Applications concern *description of programming languages* (correct syntax), *compilation*, *text processing and searching* (indexing, think of Google), *DNA reconstruction* from short subsequences, *codings* of texts, sounds and images.

More fundamentally, finite and infinite words can model program behaviours and algorithms. Many parts of *Fundamental Computer Science* are based on Formal Languages : *Complexity Theory and Semantics*.

Extensions of Formal Language Theory involve the study of sets of finite or countable terms, graphs or other combinatorial structures (like plane drawings of graphs).

This course is limited to an introduction to two main aspects : *Finite Automata and Context-Free grammars*.

Finite automata are presented in the document by P. Weil (LaBRI, Bordeaux University).

On line document :

J. Power, *Formal Languages and parsing*,

http://www.cs.nuim.ie/~jpower/Courses/parsing/new-main.html.

Book :

M. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley series in computer science, 1978 (First edition).

2 Context-Free Grammars

Context-free grammars, introduced by Chomsky (linguist) around 1950 have proved to be especially useful for describing Programming Languages.

They form a *generative description method*, by which every generated word has at least one *syntactic tree*, called below a *derivation tree*. These trees are useful to describe natural languages, and, in the case of implementations of programming languages, to translate the analyzed program into an executable program. *Ambiguity* means that a word has several syntactic trees, and more fundamentally, several *meanings*. It is unavoidable in the case of natural languages, where the meaning of a word or a sentence depends on the "outside world" and on a situation known by the speaker and by the listener (or by the writer and the reader). On the contrary, ambiguity is carefully avoided by the mathematician writing in natural language. Ambiguity is at the basis of the political as well of the humoristical discourse. The grammars describing programming languages must be *nonambiguous* for obvious reasons : one wishes that a program works in the same way on all machines.

2.1 Rewriting rules.

Let A be a finite alphabet, and R a finite set of pairs of words on A. This finite binary relation is extended in two steps into a binary relation on A^*.

First, into the binary relation defined by :

$u \longrightarrow_R v$ iff $u = u_1 x u_2$, $v = u_1 y u_2$ for some words u_1, u_2 and some pair (x, y) in R.

It is called the *one-step rewriting relation associated with R.*

Second, by taking a transitive closure :

$u \longrightarrow_R^+ v$ iff there exists a sequence of words $v_1, ..., v_k$, with $v = v_k$, such that $u \longrightarrow_R v_1 \longrightarrow_R ... \longrightarrow_R v_k$: the sequence $(u, v_1, ..., v_k)$ is called a *derivation sequence.*

2.2 Context-free grammars

A context-free grammar G is defined with two alphabets, the *terminal alphabet* A, and the *non-terminal one* N. A *production rule* is a pair (S, w) where $S \in N$, $w \in (A \cup N)^*$.

Let R be a finite set of rules of this type. Every nonterminal symbol S can be taken as the start symbol of a derivation sequence $S \longrightarrow_R v_1 \longrightarrow_R ... \longrightarrow_R v_k$ starting by S. This sequence is called *terminal* if v_k has no occurence of nonterminal symbols (so that there is no way to continue the rewritings). We say v_k that is *generated from* S by the grammar.

The language of all words in A^* generated from S is denoted by $L(G, S)$. The grammar itself is specified by the triple (A, N, R) or the 4-tuple (A, N, R, S) if we want to specify a unique start symbol S.

The term *context-free* means that the rewriting rules are applied independently of the *context* (the pair (u_1, u_2) in the definition of $\longrightarrow_R$). More complex grammars, called *context-sensitive* have applicability conditions depending on the context.

In grammars that define programming languages, one usually denote nonterminals by short names between brackets :

The `if-then-else` construction can be defined by the production rule like :

$$\langle Inst \rangle \longrightarrow \texttt{if}\langle Bool \rangle\texttt{then}\langle Inst \rangle\texttt{else}\langle Inst \rangle$$

where $\langle Inst \rangle$ generates instructions and $\langle Bool \rangle$ generates Boolean conditions.

In examples we write a production rule $S \longrightarrow m$ rather than (S, m).

2.3 A first example.

Let $A = \{x, y, z, +, *, (,)\}$, let $N = \{E\}$ and R be the set of rules :

$\{E \longrightarrow E + E, \quad E \longrightarrow E * E,$

$E \longrightarrow (E), \quad E \longrightarrow x, \quad E \longrightarrow y, \quad E \longrightarrow z\}$.

Then $L(G, E)$ contains the following words :

$$x, \quad x + y, \quad x + y * z, \quad (x + y) * z, \quad (((((((x)))))))),$$

and infinitely many others. These words are "arithmetic" expressions written with the three variables *x,y,z,* the two binary operations + and $*$, and the two parentheses.

2.4 Exercises : (1) Give derivation sequences of these words. Prove that $(x + y))$ has no derivation sequence.

(2) Prove by induction on the length of a derivation that every generated word has the same number of opening and closing parentheses.

2.5 Derivation trees : Example.

We use the above example 2.3. Let us give names $(+, *, par, var - x, var - y, var - z)$ to the production rules :

$+ : E \longrightarrow E + E,$
$* : E \longrightarrow E * E,$
$par : E \longrightarrow (E),$
$var - x : E \longrightarrow x,$
$var - y : E \longrightarrow y,$
$var - z : E \longrightarrow z.$

Let us consider the following derivation sequence :

$E \longrightarrow E + E \longrightarrow E + E * E \longrightarrow E + (E) * E$
$\longrightarrow E + (E + E) * E \longrightarrow E + (x + E) * E$
$\longrightarrow y + (x + E) * E \longrightarrow y + (x + E) * x \longrightarrow y + (x + z) * x.$

Note that by comparing two successive steps one can determine which rule is applied and at which position. (This not true for all grammars. Which conditions insure that this is true?)

Let us organize the derivation sequence into a sequence of trees, see Figures 1 and 2.

The final tree, called the *derivation tree of the sequence*, indicates the "structure" of the generated expression. It can be used to evaluate the expression, assuming that particular values are given to x, y, z. The integers 1,2 ,...,8 in Figure 2 indicate in which order the production rules labelling the nodes are chosen.

From any *topological order* of the derivation tree (that is, any linear of the set of nodes such that any node comes before its sons), one gets a derivation sequence having the same tree. Here are two important cases :

The topological order 1,6,2,3,4,5,8,7 gives the *leftmost derivation* : at each step, the leftmost nonterminal symbol is rewritten. That is :

$E \longrightarrow E + E \longrightarrow y + E \longrightarrow y + E * E$
$\longrightarrow y + (E) * E \longrightarrow E + (E + E) * E$
$\longrightarrow y + (x + E) * E \longrightarrow y + (x + z) * E \longrightarrow y + (x + z) * x.$

The topological order 1,2,7,3,4,8,5,6 gives the *rightmost derivation* : at each step, the rightmost nonterminal symbol is rewritten. That is :

$E \longrightarrow E + E \longrightarrow E + E * E \longrightarrow E + E * x$
$\longrightarrow E + (E) * x \longrightarrow E + (E + E) * x$
$\longrightarrow E + (E + z) * x \longrightarrow E + (x + z) * x \longrightarrow y + (x + z) * x.$

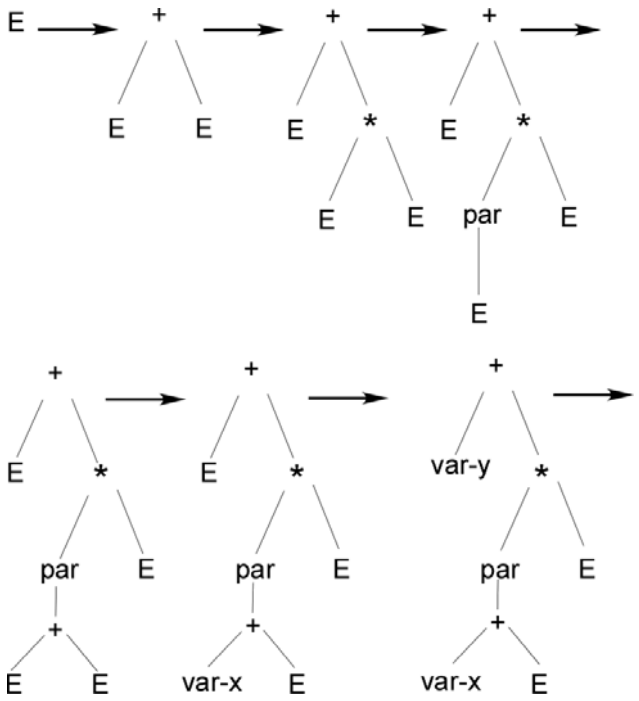

Figure 1: Derivation sequence of 2.5 ; first part.

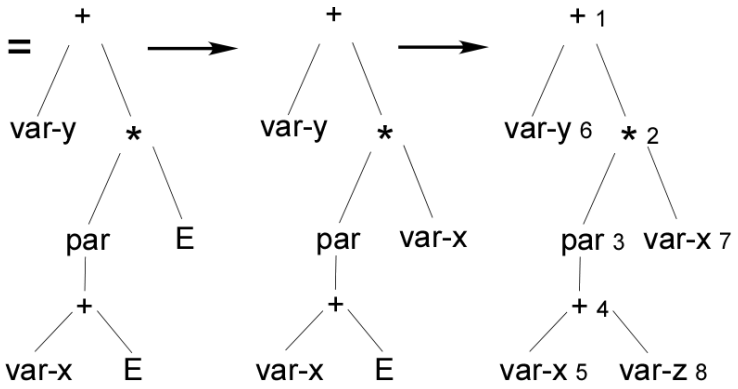

Figure 2: Derivation sequence of 2.5 ; second part.

(There are yet other topological orders like 1,6,2,3,7,4,5,8 yielding a derivation with no particular name.)

A grammar is *linear* if each righthand side of a production rule has at most one nonterminal. In this case, the derivation tree is a string. At each step of the derivation there is a unique nonterminal symbol to replace, unless the derivation is terminal. There is a single derivation sequence for each tree. However, such a grammar may be ambiguous (see 2.6). One may have several trees for a single word.

2.6 Ambiguity.

A grammar is *ambiguous* is some generated word has derivation sequences with different derivation trees. It is *nonambiguous* otherwise.

The grammar of Example 2.3 is ambiguous because the expression $x + y * z$ has two derivation trees, corresponding to the correct reading $x + (y * z)$ and to the incorrect one $(x+y)*z$. These trees give two different values if they are used to evaluate the expression. Hence this grammar G is correct in the weak sense that it only generates well-formed terms, but it is actually not because some derivation trees are incorrect with respect to the intended use of the grammar.

The expression $x + y + z$ has also two derivation trees corresponding to $x + (y + z)$ and to $(x + y) + z$. However this is harmless because addition is associative.

Here is another grammar H for our expressions. Check on examples that it is not ambiguous and gives "correct derivation trees". (A formal proof that the grammar is not ambiguous but equivalent to the first one is a bit complicated). To understand it, consider that E generates "expressions", T generates "terms", F generates "factors". An expression is a sum of terms, a term is a product of factors. Here are the rules of H.

$$\{E \longrightarrow E + T, \qquad E \longrightarrow T,$$
$$T \longrightarrow T * F, \qquad T \longrightarrow F,$$
$$F \longrightarrow (E), \qquad F \longrightarrow x, \qquad F \longrightarrow y, \qquad F \longrightarrow z\}.$$

We have $L(G, E) = L(H, E)$, H is nonambiguous and its derivation trees are correct for the evaluation of expressions, with the usual priorities for $+$ and $*$.

The grammar H remains correct for evaluation if we add the following rule for difference : $E \longrightarrow E - T$.

2.7 Exercise : For some examples of words in $L(G, E)$ compare their derivation trees relative to grammars G and H. (Of course, names must be chosen for the rules of H).

2.8 Derivation trees : General definitions.

In order to define derivation trees for a grammar (A, N, R), we identify production rules by names. If p names a rule (U, w) where :

$$w = w_1U_1w_2U_2w_3...w_iU_iw_{i+1},$$

$U_1, U_2, ..., U_i$ are nonterminal symbols and $w_1, w_2, w_3, ..., w_i$ are terminal words (i.e., $w_1, w_2, w_3, ..., w_i \in A^*$), then we say that p has *rank* i. A terminal rule has rank 0.

The derivation tree of a terminal derivation sequence will have its nodes labelled by names of production rules and a node labelled by p of rank i will have i sons, forming a sequence (the order of sons *is significant*). The derivation tree of a derivation sequence that is not terminal will have nodes labelled by names of production rules as above and some leaves labelled by nonterminal symbols.

More precisely, for a derivation sequence $S \longrightarrow_R v_1 \longrightarrow_R ... \longrightarrow_R v_k$ where $v_k = y_1X_1y_2X_2y_3...y_iX_iy_{i+1}$, $X_1, X_2, ..., X_i$ are nonterminal symbols and $y_1, y_2, ..., y_i \in A^*$, the associated derivation tree has leaves labelled by $X_1, X_2, ..., X_i$, in this order (if one traverses the set of leaves from left to right), called *nonterminal leaves* and other leaves labelled by terminal production rules, called terminal leaves. Figures 1 and 2 illustrate this. The terminal leaves are labelled by $var - x, var - y, var - z$.

Formal definition

The formal definition of the derivation tree $Der(d)$ associated with a derivation $d = S \longrightarrow_R v_1 \longrightarrow_R ... \longrightarrow_R v_k$ is defined as follows by induction on the length of the sequence :

1) If d has length 0, i.e. is reduced to S, the corresponding tree has a single node, which is both a leaf and the root, labelled by S.

2) If d has length 1, i.e. $d = S \longrightarrow_R v_1$, then $Der(d)$ has a root (replacing the previous S) labelled by p, the name of the production rule (S, v_1).

If this rule is terminal then the root has no son, the construction is finished.

Otherwise, we can write $v_1 = w_1U_1w_2U_2w_3...w_iU_iw_{i+1}$, with $U_1, U_2, ..., U_i$ nonterminal symbols and $w_1, w_2, w_3, ..., w_i \in A^*$. Then the root has i sons, labelled from left to right by $U_1, U_2, ..., U_i$.

3) We now give the general inductive step.

We let $d = S \longrightarrow_R v_1 \longrightarrow_R ... \longrightarrow_R v_{k-1} \longrightarrow_R v_k$. We assume that the derivation tree $Der(d')$ of $d' = S \longrightarrow_R v_1 \longrightarrow_R ... \longrightarrow_R v_{k-1}$ has been constructed, and that $v_{k-1} = w_1U_1w_2U_2w_3...w_iU_iw_{i+1}$, with $U_1, U_2, ..., w_1, ...$ as above. The nonterminal leaves are labelled by $U_1, U_2, ...$ in this order.

Let the last step of d, $v_{k-1} \longrightarrow_R v_k$ replace U_j by $y_1Z_1y_2Z_2y_3...y_mZ_my_{m+1}$, using rule $p = (U_j, y_1Z_1y_2Z_2y_3...y_mZ_my_{m+1})$. Hence :

$$v_k = w_1U_1w_2U_2w_3...w_jy_1Z_1y_2Z_2y_3...y_mZ_my_{m+1}w_{j+1}...w_iU_iw_{i+1}.$$

Then $Der(d)$ is obtained from $Der(d')$ by changing the label U_j of the j-th leaf into p, and attaching m sons to this node, labelled $Z_1, Z_2, ..., Z_m$ from left to right.

In this step we have $m = 0$ if p is a terminal production rule. The number of nonterminal leaves of $Der(d)$ is one less than the number of those of $Der(d')$.

(Verify that the trees built from grammar H in Exercise 2.7 are correct for this definition.)

Note that the order in which the nodes of $Der(d)$ get their "final" labels by names of production rules corresponds to the order in which these production rules are used in d. Conversely :

2.9 Proposition : For every topological order $\leq$ of the derivation tree $Der(d)$ of a derivation sequence, there is a derivation sequence of the same (terminal or nonterminal word) that creates the same derivation tree in the order defined by $\leq$. The prefix order gives the corresponding leftmost derivation sequence and the postfix order gives the rightmost one.

2.10 Concrete syntax

Derivation trees (as defined above) do not contain information about the terminal symbols of the generated words, because production rules are only given by their names and rank. Several concrete syntaxes can be associated with a same grammar.

For example, the `if-then-else` construction that can be defined by a production rule like :

$$\langle Inst \rangle \longrightarrow \texttt{if}\ \langle Bool \rangle\ \texttt{then}\ \langle Inst \rangle\ \texttt{else}\ \langle Inst \rangle$$

may have name `cond` (for *conditional instruction*) in several concrete implementations in various languages, giving in French :

$$\langle Inst \rangle \longrightarrow \texttt{si}\ \langle \texttt{Bool} \rangle\ \texttt{alors}\ \langle Inst \rangle\ \ \texttt{sinon}\ \langle Inst \rangle.$$

One can also enrich the derivation trees by terminal symbols, linked to the nodes of the corresponding production rules. This is best understood from Figure 3 than from a formal description. Such *concrete derivation trees* are useful in syntactic analysis. The derivation trees initially defined are better suited to semantics and translation.

3 Context-free languages

3.1 Context-free languages

A language L is *context-free* if it is generated by a context-free grammar. By generate, we mean *exactly* : $L = L(G, S)$ for some grammar. It is not sufficient to say "every word in L is generated by the grammar G".

We will denote by $CFL(A^*)$ the family of context-free languages over the alphabet A.

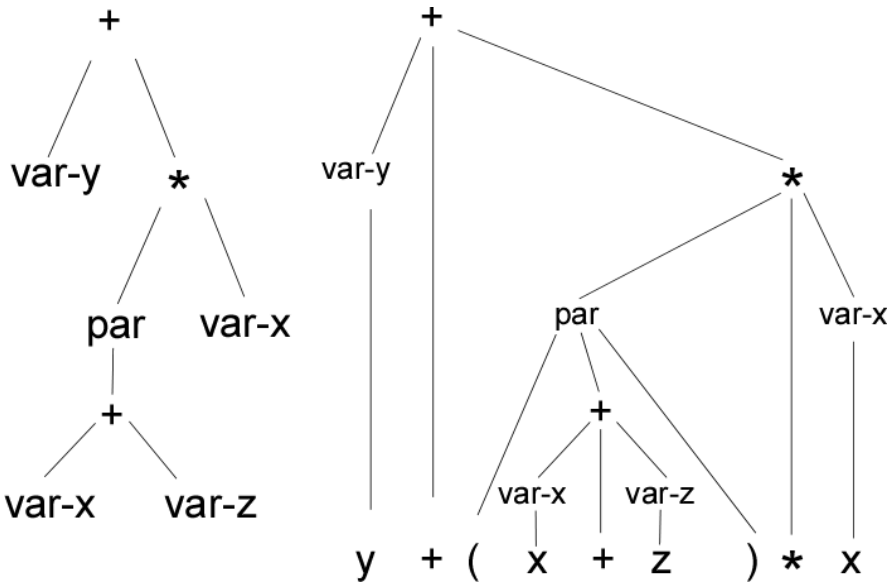

Figure 3: A derivation tree and its version showing the concrete syntax.

3.2 Exercises : (1) Construct context-free grammars that generate the following languages :

$L = A^*abA^*$,

$M = \{a^n b^n \mid n \geq 1\}$, and recall that this language is not recognizable,

$P = \{w\widetilde{w} \mid w \in A^*\}$, (recall that $\widetilde{w}$ is the mirror image of w),

any finite language.

(2) Do the same for these languages :

$M.M = \{a^n b^n a^m b^m \mid n, m \geq 1\}$,

$R = \{a^n b^n c^m \mid n, m \geq 1\}$,

$Q = \{a^n b^n, a^n b^{2n} \mid n \geq 1\}$.

(3) Try to do the same and observe that this is not possible for the languages:

$T = \{a^n b^n c^n \mid n \geq 1\}$,

$C = \{ww \mid w \in A^*\}$ (consider also the case where A has only one letter?).

(4) Which of the above grammars are linear? Construct examples of derivation trees.

(5) Modify the grammars for 0 instead of 1 in the above definitions.

Closure properties of the family $CFL(A^*)$

3.3 Theorem : (1) If L and K are context-free languages then the languages $L \cup K, L.K, L^*$ are context-free. There exist algorithms that construct grammars for these languages from grammars defining L and K.

(2) There exist context-free languages L and K such that the languages $L \cap K, L - K$ are not context-free.

Proof :

(1) Let $L = L(G, S), K = L(H, U)$ for grammars G, H and initial nonterminals S, U.

Step 1 : One changes names of nonterminals if necessary so that the two gramars have disjoint sets of nonterminals.

Step 2 for union : One takes the union of the sets of rules of G and H and one adds a new nonterminal X with rules $X \longrightarrow S, X \longrightarrow U$.

Step 2 for concatenation : One takes the union of the sets of rules of G and H and one adds a new nonterminal X with rule $X \longrightarrow SU$.

Unique step for star (iteration) : For $L(G, S)^*$ one adds a new nonterminal X and rules $X \longrightarrow \varepsilon, X \longrightarrow SX$.

(2) The language T is not context-free (the proof is technical; see books to find it) but T is the intersection of the two context-free languages $R = \{a^n b^n c^m \mid n, m \geq 1\}$ and $S = \{a^n b^m c^m \mid n, m \geq 1\}$.

If $CFL(A^*)$ would be closed under difference it would be closed under intersection since it is also closed under union. (Exercise : Complete the proof).□

3.4 Exercise: Give an example showing that omitting Step 1 may yield an incorrect construction.

Comparison with finite automata.

3.5 Proposition : For every alphabet A : $Rec(A^*) \subseteq CFL(A^*)$, the inclusion is strict if A has at least 2 letters.

Proof:

Fact : For every rational expression R one can construct (by an unique algorithm) a context-free grammar that generates $L(R)$, the language defined by R.

The proof is by induction on the structure of R using Theorem 3.3. (Work out some examples). Since every L in $Rec(A^*)$ can be (effectively) given by a rational expression R the result follows from the Fact.

Prove the strictness assertion with the above examples. □

3.6 Exercise : Prove this proposition by translating a finite automaton into a *right linear grammar*, i.e. a grammar with rules of the form $X \longrightarrow w$ or $X \longrightarrow wY$ with w terminal word and Y nonterminal symbol.

3.7 Exercise : Prove that the family $CFL(A^*)$ is closed under morphisms: $A^* \longrightarrow A^*$.

4 Some transformations of context-free grammars

4.1 Grammar cleaning

A nonterminal S of G is *productive* if $L(G, S)$ is not empty.

4.2 Proposition : (1) There exists an algorithm that determines the set of productive nonterminals.

(2) If $L(G, S)$ is not empty then it is equal to $L(G', S)$ where G' is obtained from G by deleting improductive nonterminals and the production rules where they occur.

Proof : Let $G = (A, N, R)$.

(1) One defines an increasing sequence of subsets of N as follows :

N_1 is the set of nonterminals S such that $(S, w) \in R$ for some w with no nonterminal symbol.

N_{i+1} is N_i augmented with the set of nonterminals S such that $(S, w) \in R$ for some $w \in (N_i \cup A)^*$.

Fact : For all i, N_i is the set of nonterminals for which some terminal derivation sequence has derivation tree of height at most i.

For this fact, two proofs must be done : one by induction on i that every S in N_i has a terminal derivation sequence with derivation tree of height at most i. The other one that if a terminal derivation sequence of length n has a derivation tree of height at most i, then its initial nonterminal is in N_i. One uses an induction on n.

Then, necessarily (why ?) $N_{p+1} = N_p$ for some p and then $N_p = N_q$ for all $q > p$. It follows that N_p is computable and is the set of productive nonterminals.

(2) Clearly, $L(G, S) \supseteq L(G', S)$. For the other direction, observe that all nonterminals occurring in a terminal derivation sequence are productive. □

4.3 Corollary : There is an algorithm that decides whether a context-free grammar generates a nonempty language.

A nonterminal U is *useful for* S if it is used in a terminal derivation sequence starting with S. In this derivation sequence, all nonterminals are productive.

4.3 Proposition : (1) There exists an algorithm that determines the set of nonterminals that are useful for a given nonterminal S.

(2) If $L(G, S)$ is not empty then it is equal to $L(G', S)$ where G' is obtained from G by deleting all nonterminals that are not useful for S, and all production rules where they occur.

Proof : Let $G = (A, N, R, S)$ be a grammar without improductive nonterminals.

(1) One defines an increasing sequence of subsets of N as follows :

$M_1 = \{S\}$, M_{i+1} is M_i augmented with the set of nonterminals W such that $(U, wWw') \in R$ for some $U \in M_i$.

By using an argument similar to that of Proposition 4.2, prove that the increasing sequence M_i is constant from a certain point, and that its value at this point is the set of nonterminals useful for S.

(2) Complete the proof. □

A simple grammar transformation

Let G be a grammar (A, N, R) let U be a nonterminal symbol which has no occurrence in the righthand sides of its rules. In other words, if (U, w) is a rule in R, then U has no occurrence in w. With this hypothesis :

4.4 Proposition : One can transform G into a grammar $G' = (A, N - \{U\}, R')$ such that $L(G', S) = L(G, S)$ for every S in $N - \{U\}$.

Proof : We will transform G into G' by iterating an elementary transformation step.

Elementary step :

Choose any rule (X, w) of the grammar G where $w = w_1 U w_2 ... w_p U w_{p+1}$, U has p occurrences in w. The words $w_1, w_2, ... w_p, w_{p+1}$ may contain occurrences of nonterminals other than U.

Let $y_1, ..., y_k$ be the righthand sides of the production rules with lefthand side U.

Replace the rule (X, w) by the set of rules (X, w') for all words $w' = w_1 z_1 w_2 ... w_p z_p w_{p+1}$ where $z_1, ..., z_p \in \{y_1, ..., y_k\}$.

One obtains thus k^p rules.

For an example, if the rules with lefthand side U are $U \longrightarrow aS, U \longrightarrow b$, and if $w = cSUdUTf$, then one obtains the 4 rules :

$$X \longrightarrow cSaSdaSTf, \qquad X \longrightarrow cSbdaSTf,$$
$$X \longrightarrow cSaSdbTf, \qquad X \longrightarrow cSbdbTf.$$

Claim 1 : The obtained grammar, call it H, verifies the two properties :

$L(H, S) = L(G, S)$ for every S (even if $S = U$),

U has no occurrence in the righthandsides of its rules in H.

Proof : For every word x, and every nonterminal S, x has a derivation sequence from S in G iff it has one from S in H. These proofs use inductions on the lengths of derivation sequences. Note that the derivation in H is shorter (or of equal length) than the corresponding one in G.□

Claim 2 : By finitely many applications of this elementary step to the given grammar G, one obtains a grammar M such that U has no occurrence in any righthand side.

Proof: This elementary step is applicable n times where n is the number of production rules with U in the righthand side.

Each application decreases this number by one.

At each step the obtained grammar G_i is equivalent to the previous one G_{i-1}. That is : $L(G_i, S) = L(G_{i-1}, S)$ for every S (even if $S = U$).

After n elementary steps, one obtains by these two claims a grammar M equivalent to G. But U is useless for generating words from S in $N - \{U\}$. By deleting it one obtains a grammar as desired.□

Here is an application.

4.5 Proposition : One can transform a grammar $G = (A, N, R)$ into one $H = (A, N', R')$ such that $N' \supseteq N$, the righthand sides of rules of R' have length at most 2 and $L(G, S) = L(H, S)$ for every S in N.

Proof : One repeats as many times as necessary the following step :

If G contains a rule (S, w) with w of length n at least 3, then one writes $w = w_1 w_2$ with w_1 and w_2 of strictly smaller lengths (say $\lceil n/2 \rceil$ and $n - \lceil n/2 \rceil$) one introduces two new nonterminals U and W and one replaces the rule (S, w) by the three rules (S, UW), (U, w_1) and (W, w_2). If w_2 has length 1, one does not introduce W and one uses instead the rule (S, Uw_2).

The new grammar G_1 verifies by the last proposition $L(G_1, S) = L(G, S)$ for every S in N.

After finitely many steps (give an upper bound) one obtains a grammar as desired.□

4.6 Example :

Let G be defined by rules :

$S \longrightarrow aSSSS, \qquad S \longrightarrow bcd.$

After one step one obtains

$S \longrightarrow UW, \quad U \longrightarrow aSS, \quad W \longrightarrow SS, \quad S \longrightarrow bcd,$

Then :

$S \longrightarrow UW, \quad U \longrightarrow TS, \quad T \longrightarrow aS, \quad W \longrightarrow SS, \quad S \longrightarrow bcd,$

and finally :

$S \longrightarrow UW, \quad U \longrightarrow TS, \quad T \longrightarrow aS, \quad W \longrightarrow SS,$

$S \longrightarrow Zd, \quad Z \longrightarrow bc.$

Another application is a kind of *factorization* : see Exercise 5.3.

5 LL(1) parsing

This section is a quick introduction to the notion of *parsing*, also called *syntactic analysis*. (*Lexical analysis* is done by finite automata.) We will present *top-down parsing*, that is, a construction of a derivation tree from its root, usually represented on top of the figure, while the input word is read from left to right. A leftmost derivation is constructed.

5.1 Example

Consider the grammar with rules :

$S \longrightarrow E$
$E \longrightarrow TE'$
$E' \longrightarrow +E \qquad E' \longrightarrow \varepsilon$
$T \longrightarrow FT'$
$T' \longrightarrow *T \qquad T' \longrightarrow \varepsilon$
$F \longrightarrow (E) \qquad F \longrightarrow \mathbf{id}$

Figure 4 shows the progressive top-down construction of a derivation tree, while the input word is $\mathbf{id} + \mathbf{id} * \mathbf{id}$ is read. The marker | shows to its left the prefix of the input word read at that point.

5.2 Simple deterministic grammars.

A grammar is *simple deterministic* if the rules are all of the form (S, aw) for some terminal symbol a, and for every S and a there is at most one rule of this form. A context-free language is *simple deterministic* if it is generated by at least one simple deterministic grammar (and necessarily others which are not simple deterministic (Exercise : why?)).

The class of *LL(1) grammars* contains simple deterministic grammars. It is defined in terms of an extension of the parsing algorithm given below. The main features of LL(1) grammars are shared by simple deterministic grammars.

5.3 Examples

(1) The grammar K with rules :

$+ : E \longrightarrow +EE,$
$* : E \longrightarrow *EE,$
$var - x : E \longrightarrow x,$
$var - y : E \longrightarrow y,$
$var - z : E \longrightarrow z$

is simple deterministic. It generates arithmetic expressions written in *Polish prefix notation*. This grammar is simple deterministic, hence nonambiguous (see below 5.8). Its derivation trees are correct for evaluation, however long expressions are rather unreadable.

The grammars of Examples 2.3 and 2.6 are not simple deterministic. We will prove below that every simple deterministic language L is *prefix-free*: this means that if u and v are two words such that u and uv belong to L, then v must be the empty word. It follows that the language $L(G, E)$ of Example 2.3 is not simple deterministic, because it contains the words x and $x + x$. The grammar G cannot be transformed into a simple deterministic one generating $L(G, E)$.

(2) Let us consider the grammar J with rules :

$I \longrightarrow$ `if` B `then` I `else` I
$I \longrightarrow$ `if` B `then` I
$I \longrightarrow$ `assignment`

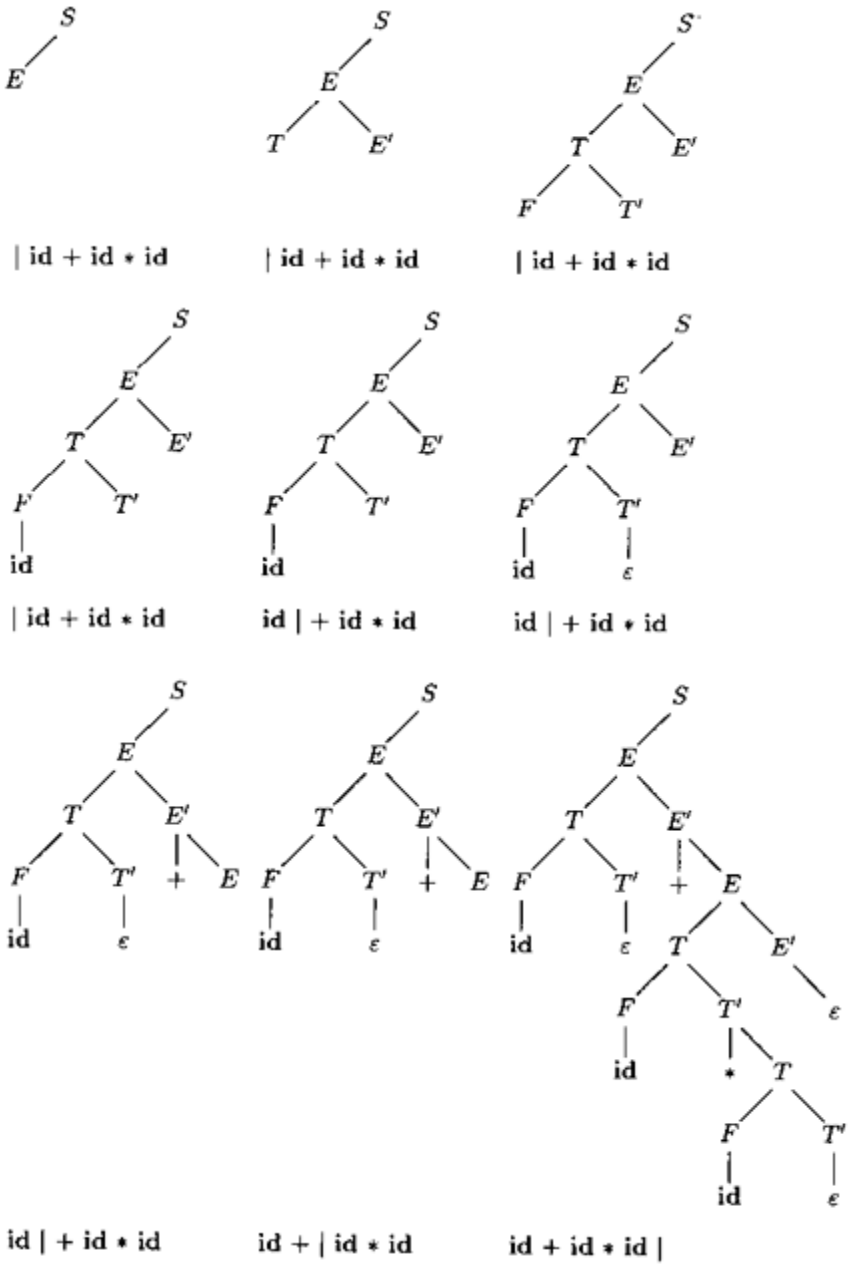

Figure 4: Top-down parsing

$B \longrightarrow$ `true`
$B \longrightarrow$ `false`

It is not simple deterministic and the language it generates is not prefix-free.

Let us modify the grammar J by adding terminal symbols playing the role of closing parentheses, obtaining J'

$I \longrightarrow$ `if` B `then` I `else` I `fi`
$I \longrightarrow$ `if` B `then` I `fi`
$I \longrightarrow$ `assignment`
$B \longrightarrow$ `true`
$B \longrightarrow$ `false`

The generated language is prefix-free, but the grammar is not yet simple deterministic. However, it can be transformed into the following simple deterministic equivalent one J":

$I \longrightarrow$ `if` B `then` IZ
$Z \longrightarrow$ `else` I `fi`
$Z \longrightarrow$ `fi`
$I \longrightarrow$ `assignment`
$B \longrightarrow$ `true`
$B \longrightarrow$ `false`

5.4 Exercise : Prove that $L(J', I) = L(J", I)$. One can use Proposition 4.4.

For a fixed grammar $G = (A, N, R)$ we denote by $w \Longrightarrow^* w'$ for $w, w' \in (A \cup N)^*$ a leftmost derivation; the star in $\Longrightarrow^*$ means that we allow the particular case of $w' = w$, i.e, of a derivation of length 0.

The good properties of simple deterministic grammars regarding parsing are in the following theorem.

5.5 Theorem : Let $G = (A, N, R)$ be a simple deterministic grammar. Let w in $(A \cup N)^*$ and let x be a terminal word (in A^*).

Then $w \Longrightarrow^* x$ iff

(1) either $w = \varepsilon$ and $x = \varepsilon$, (ε denotes the empty word),

(2) or $w = aw'$ for some a in A and then, $x = ax'$ and $w' \Longrightarrow^* x'$,

(3) or $w = Sw'$ for some S in N and then, $x = ax'$ for some a in A and $mw' \Longrightarrow^* x'$, where $(S, am) \in R$.

In Case (2) if x does not start with a, this is a syntax error.

In Case (3) if x does not start with a, or if no rule of the form (S, am) is found, this is a syntax error.

In Case (3) there is at most one rule (S, am) to continue the analysis.

Proof : Let us assume $w \Longrightarrow^* x$ then :

If $w = \varepsilon$ the only possibility (for any grammar) is $x = w = \varepsilon$, because no rewriting from w is possible.

If $w = aw'$ for some a in A then, $x = ax'$ and $w' \Longrightarrow^* x'$. This is actually true for any grammar (nonterminal symbols remain in derivation steps).

If $w = Sw'$ for some S in N we cannot have $x = \varepsilon$ because there is no rule in a simple deterministic grammar with empty right handside. So no rewriting can erase nonterminals. One can only replace them by nonempty words.

Hence we must have $x = ax'$ for some a in A. The leftmost derivation $w \Longrightarrow^* x$ must rewrite S first, and by a rule (S, bm). Hence $w \Longrightarrow^* x$ is of the form $w = Sw' \Longrightarrow bmw' \Longrightarrow^* x = ax'$. Hence we must have $b = a$. If there is in R no rule of the form (S, am) there is an error, the hypothesis $w \Longrightarrow^* x$ is wrong.

Then by (2) we have $mw' \Longrightarrow^* x'$.

The proofs in the other direction are simple verifications.□

5.6 Example :

Consider the grammar following variant K' of K of 5.3.

$+ : E \longrightarrow +EF,$
$* : E \longrightarrow *EF,$
$var - x : E \longrightarrow x,$
$var - y : E \longrightarrow y,$
$var - z : E \longrightarrow z,$
$par : F \longrightarrow (E),$
$\% : F \longrightarrow \%E\$F.$

Consider the word $++x(y)\%z\$(z)$. By applying Theorem 5.5 we can see that:

$E \Longrightarrow^* ++x(y)\%z\$(z)$	iff	(by Case (3) for rule +)
$EF \Longrightarrow^* +x(y)\%z\$(z)$	iff	(by Case (3) for rule +)
$EFF \Longrightarrow^* x(y)\%z\$(z)$	iff	(by Case (3) for rule $var - x$)
$FF \Longrightarrow^* (y)\%z\$(z)$	iff	(by Case (3) for rule par)
$E)F \Longrightarrow^* y)\%z\$(z)$	iff	(by Case (3) for rule $var - y$)
$)F \Longrightarrow^*)\%z\$(z)$	iff	(by Case (2))
$F \Longrightarrow^* \%z\$(z)$	iff	(by Case (3) for rule %)
$E\$F \Longrightarrow^* z\$(z)$	iff	(by Case (3) for rule $var - z$)
$\$F \Longrightarrow^* \$(z)$	iff	(by Case (2))
$F \Longrightarrow^* (z)$	iff	(by Case (3) for rule par)
$E) \Longrightarrow^* z)$	iff	(by Case (3) for rule $var - z$)
$) \Longrightarrow^*)$	iff	(by Case (2))
$\varepsilon \Longrightarrow^* \varepsilon$		which is valid by Case (1).

The list of rules applied is that of the leftmost terminal derivation that we have recognized. This computation can be organized into the following table (see 5.7).

Input read	Input not read	Content of pushdown
ε	$++x(y)\%z\$(z)$	E
$+$	$+x(y)\%z\$(z)$	EF
$++$	$x(y)\%z\$(z)$	EFF
$++x$	$(y)\%z\$(z)$	FF
$++x($	$y)\%z\$(z)$	$E)F$
$++x(y$	$)\%z\$(z)$	$)F$
$++x(y)$	$\%z\$(z)$	F
$++x(y)\%$	$z\$(z)$	$E\$F$
$++x(y)\%z$	$\$(z)$	$\$F$
$++x(y)\%z\$$	(z)	F
$++x(y)\%z\$($	$z)$	$E)$
$++x(y)\%z\$(z$	$)$	$)$
$++x(y)\%z\$(z)$	ε	ε

Here is another example with the word $++x(y\%)\%z\$(z)$.

$E \Longrightarrow^* ++x(y\%)\%z\$(z)$ iff (by Case (3) for rule $+$)

$EF \Longrightarrow^* +x(y\%)\%z\$(z)$ iff (by Case (3) for rule $+$)

$EFF \Longrightarrow^* x(y\%)\%z\$(z)$ iff (by Case (3) for rule $var - x$)

$FF \Longrightarrow^* (y\%)\%z\$(z)$ iff (by Case (3) for rule par)

$E)F \Longrightarrow^* y\%)\%z\$(z)$ iff (by Case (3) for rule $var - y$)

$)F \Longrightarrow^* \%)\%z\$(z)$.

Here we find a mismatch between the two terminal symbols) and %. The given word is not derivable from E.

5.7 Pushdown automata

Observe that in the first (successful) example, the input word is processed letter by letter from left to right. This processing is similar to what is done by a deterministic automaton recognizing a rational language. Here the successive words $E, EF, EFF, FF, E)F,)F$ etc ... play the role of states. We have actually an automaton *with an infinite set of states*, which is a subset of the language $\{E, F, +, *, \%, (,), x, y, z\}^*$. The transitions from a state to another one are defined in particular way : they can only erase the leftmost symbol of the word representing the state or replace it by a word w coming from a production rule of the form (S, aw). We call this a *pushdown automaton* because of the particular way transitions use and modify the words representing the states. It can be implemented with a (*pushdown*) *stack,* a data structure which manipulates lists of objects (letters or pointers) by modifying them only by deleting or adding items at one end. The automaton is deterministic because each transition is determined in a unique way by the next letter. The automaton may reach an error state, as we have seen in the example of the word $++x(y\%)\%z\$(z)$.

Its table is shown below : m stands for any word in $\{E, F, +, *, \%, (,), x, y, z\}^*$. Hence this table represents infinitely many transitions, but described in a finitary way. Any case not covered in the table yields a transition to an error state

(or in a compiler, calls an error recovery procedure). Three examples are given at the bottom of the table.

Next input symbol	Pushdown content	Transition to	Recognized rule
End of input	ε	SUCCESS	
$+$	Em	EFm	$+$
$*$	Em	EFm	$*$
x	Em	m	$var - x$
y	Em	m	$var - y$
z	Em	m	$var - z$
(	Fm	$E)m$	par
)	$)m$	m	
%	Fm	$E\%Fm$	%
\$	$\$m$	m	
End of input	not ε	ERROR	
Any symbol	ε	ERROR	
x	Fm	ERROR	

5.8 Corollary : A simple deterministic language is prefix-free. A simple deterministic grammar is nonambiguous.

Proof : It follows from Theorem 5.5 that if G is simple deterministic, if $S \Longrightarrow^* uUm$, and $S \Longrightarrow^* uTm'$ for u a terminal word and U, T nonterminals, then $Um = Tm'$.

Hence if $L(G, S)$ contains u and uv, we have a single leftmost derivation $S \Longrightarrow^* uUm$, and we should have $Um \Longrightarrow^* v$ and $Um \Longrightarrow^* \varepsilon$. This last fact is not possible if v is not the empty word..

Every generated word is generated by a single leftmost derivation, also by Theorem 5.5, hence G is nonambiguous. □

6 Inductive properties and intersection with recognizable languages.

The corresponding notions will be presented by representative exercises.

6.1 Proofs by induction :

All words u in $L(G, E)$ for the grammar G of Example 2.3 have odd length, written as property $P(u)$. They also have as many opening as closing parentheses, written as property $Q(u)$.

This fact can be proved by induction on the length of derivation sequences in two different ways.

First method : Induction the length of terminal leftmost derivation sequences.

More precisely, one proves, by induction on n, that :

For every positive integer n, if $E \Longrightarrow^+ u$ is a terminal leftmost derivation sequence of length n, written $E \Longrightarrow^n u$, then $P(u)$ holds.

One proves also by induction on n that:

For every positive integer n, if $E \Longrightarrow^n u$ then $Q(u)$ holds.

Second method :

Properties P and Q are meaningful also for words w in $(\{E\} \cup A)^*$ where A is the alphabet.

One can prove by induction on n, that :

For every integer $n \geq 0$, if $E \longrightarrow^* w$ is a derivation sequence of length n, then $P(w)$ holds.

and that :

For every integer $n \geq 0$, if $E \longrightarrow^* w$ is a derivation sequence of length n, then $Q(w)$ holds.

This gives the results for the words in $L(G, E)$. (Work out the proof).

6.2 Proposition : For every context-free grammar G and nonterminal S of G, for every finite automaton B, one can construct a context-free grammar H and a nonterminal U such that $L(H, U) = L(G, S) \cap L(B)$. Hence, the intersection of a context-free and a rational language is a context-free language.

Compare with Theorem 3.3 and Kleene's Theorem for rational languages. We do not give a formal proof but we present a representative example:

6.3 Exercise :

Let G be the grammar with rules

$E \longrightarrow aEF \qquad E \longrightarrow bEc \qquad E \longrightarrow c \qquad E \longrightarrow da$

$F \longrightarrow aFF \qquad F \longrightarrow dF \qquad F \longrightarrow cE.$

Let K be the rational language consisting of all *even words* in $\{a, b, c, d\}^*$, i.e., those of even length.We define nonterminals E_0 and F_0 intended to produce the even words generated by E and F, and similarily, E_1 and F_1 intended to produce the *odd words* generated by E and F.

By some elementary observations like that a word avw is even if v is even and w is odd, we deduce the following rules of the new grammar H

$E_0 \longrightarrow aE_0F_1 \qquad E_0 \longrightarrow aE_1F_0$

$E_0 \longrightarrow bE_0c \qquad E_0 \longrightarrow da$

$E_1 \longrightarrow aE_0F_0 \qquad E_1 \longrightarrow aE_1F_1$

$E_1 \longrightarrow bE_1c \qquad E_1 \longrightarrow c.$

$F_0 \longrightarrow aF_0F_1 \qquad F_0 \longrightarrow aF_1F_0$

$F_0 \longrightarrow dF_1 \qquad F_0 \longrightarrow cE_1$

$F_1 \longrightarrow aF_0F_0 \qquad F_1 \longrightarrow aF_1F_1$

$F_1 \longrightarrow dF_0 \qquad F_1 \longrightarrow cE_0.$

By using the first method of 6.1, prove that :

Every word u of $L(G,E)$ (resp. of $L(G,F)$) is generated from E_0, resp. from F_0 if it is even and from E_1, resp. from F_1 if it is odd.

The four assertions are proved in a unique induction as follows. Let us write $Even(u)$ ($Odd(u)$) the property to be even (resp.odd). The proof is, by induction on n, that :

For every positive integer n :

for every word u in $\{a,b,c,d\}^*$:

{ if $E \Longrightarrow^n u$ and $Even(u)$ then $E_0 \Longrightarrow^n u$,
and if $F \Longrightarrow^n u$ and $Even(u)$ then $F_0 \Longrightarrow^n u$,
and if $E \Longrightarrow^n u$ and $Odd(u)$ then $E_1 \Longrightarrow^n u$,
and if $F \Longrightarrow^n u$ and $Odd(u)$ then $F_1 \Longrightarrow^n u$ }.

Remark : The grammar G is simple deterministic, but H is not.

6.4 Corollary : There exists an algorithm that decides, for every given context-free grammar G and nonterminal S, and every finite automaton D whether $L(G,S) \subseteq L(D)$.

Proof : Clearly, $L(G,S) \subseteq L(D)$ iff $L(G,S) \cap (A^* - L(D)) = \varnothing$. One builds a finite automaton B such that $L(B) = A^* - L(D)$, one constructs H and U such that :

$$L(H,U) = L(G,S) \cap L(B) = L(G,S) \cap (A^* - L(D))$$

One tests whether $L(H,U) = L(G,S) \cap L(B)$ is empty using 4.3 this gives the desired answer.□

The first assertion of 6.1 can be established also by applying 6.4, because property P defines a rational language. But this is not the same for property Q of 6.1.

7 Exercises (a first list)

7.1

Construct context-free grammars generating : $L = \{a^n b^{p+2} \mid n \neq p+2\}$, $M = \{a^n b^p \mid n \leq p \leq 2n\}$ and $N = \{a^n b^m \mid m \geq 3n+1, n \geq 0\}$.

7.2

For each k, define a context-free grammar generating the language L_k consisting of a single word a^{2^k}. Construct one of minimal size. (The *size of a grammar* is the sum of lengths of its production rules).

7.3

1) Construct a context-free grammar that generates the mirror image of the language $L(K', E)$ of 5.6. Prove the correctness of the construction.

2) Give a construction for an arbitrary context-free grammar.

7.4

We say that u is a *prefix* of v if $v = uw$ for some w, a *suffix* of v if $v = wu$ for some w, a *factor* of v if $v = wuw'$ for some w, w'.

Let G be the grammar with rules

$E \longrightarrow aEFF, \qquad E \longrightarrow bEc, \qquad E \longrightarrow cd,$

$F \longrightarrow aFF, \qquad F \longrightarrow dF, \qquad F \longrightarrow cE.$

Let M be the set of prefixes of the words of $L(G, E)$, N be the set of suffixes of the words of $L(G, E)$, P the set of factors of the words of $L(G, E)$. Construct grammars with as few rules as possible, that generate M, N, P.

7.5

Show that for some morphisms h to be determined we have

$h(L) = \{b^{3n}a^{2n} \mid n \geq 1\}$,

$h'(L) = \{d^n \mid n \geq 1\}$,

where $L = \{a^n b^n c^n \mid n \geq 1\}$.

Hence the image of a noncontext-free language under a homomorphism may be context-free or even rational. Compare with Exercise 3.7.

7.6 (More difficult)

1) Give a simple deterministic grammar for the language $\{a^n b^n \mid n \geq 1\} \cup \{c^n d^n \mid n \geq 1\}$.

2) Prove that $\{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\}$ is not simple deterministic. Compare with 3.7 and exercise 7.5.

3) Prove that $\{a^n cb^n \mid n \geq 1\} \cup \{a^n db^{2n} \mid n \geq 1\}$ is not simple deterministic.

For a simple deterministic grammar G, let $P(G, S)$ be the set of words m such that $S \Longrightarrow^* uUm$ for some terminal word u and some nonterminal symbol U.

4) Prove that for the grammar H with rules :

$$S \longrightarrow bTUT, \qquad T \longrightarrow aU, \qquad T \longrightarrow b, \qquad U \longrightarrow aU, \qquad U \longrightarrow bT,$$

the set $P(H, S)$ is finite. Determine this set. Deduce from it that the language $L(H, S)$ is rational, and construct an automaton for it.

5) For a general simple deterministic grammar G, prove that if $P(G, S)$ is finite, then $L(G, S)$ is rational.

6) For a nonempty language L, let $\| L \|$ denote the length of a shortest word of L.

For two nonempty languages L and K, compare $\| L \|$, $\| K \|$, $\| L.K \|$ and $\| L \cup K \|$.

7) Prove that if G has only productive nonterminals and $P(G, S)$ is infinite, then :

For every n, there is a terminal word u such that $\| L(G, S)/u \| > n$. Conclude that $L(G, S)$ is not rational. We recall that L/u denotes $\{w \in A^* \mid uw \in L\}$.

Is the grammar of 6.3 of this form ?