CONTEXT-SENSITIVE LANGUAGES

The grammar of a natural language consists of rules for building sentences where some linguistic terms are used as intermediate steps. For instance, the most general linguistic concept ⟨*sentence*⟩ can be presented as

⟨*noun-phrase*⟩⟨*verb-phrase*⟩

or

⟨*noun-phrase*⟩⟨*verb*⟩⟨*direct-object-phrase*⟩

If we continue with the construction of a sentence, we have to choose some ⟨*noun-phrase*⟩ and some ⟨*verb-phrase*⟩ in the former case (which we consider by reasons of simplicity). A ⟨*noun-phrase*⟩ can be a ⟨*proper-noun*⟩ or a construct ⟨*determiner*⟩⟨*common-noun*⟩, and a ⟨*verb-phrase*⟩ can be a ⟨*verb*⟩ or a construct ⟨*verb*⟩⟨*adverb*⟩. If we follow the second possibility in both cases, we obtain the structure

⟨*determiner*⟩⟨*common-noun*⟩⟨*verb*⟩⟨*adverb*⟩

for the sentence. Now we can replace any of these terms by a corresponding word, for example, ⟨*determiner*⟩ by *the,* ⟨*common-noun*⟩ by *person,* ⟨*verb*⟩ by *goes* and ⟨*adverb*⟩ by *slowly,* and we get the sentence

the person goes slowly

However, we can also choose *a, book, writes, frequently,* respectively, yielding the sentence *a book writes frequently,* which is syntactically correct but semantically nonsense. Hence by such rules we can cover only the syntax of a language. We see that the basic idea in the construction of a sentence is the substitution of some linguistic construct by one or more refined constructs or (finally) by words.

The same idea can be found in the theory of programming languages. For example, in a manual for PASCAL, one can find the well-known ⟨*if statement*⟩

if ⟨*expression*⟩ **then** ⟨*statement*⟩

as a ⟨*conditional statement*⟩. Now one has to replace ⟨*expression*⟩ and ⟨*statement*⟩ in a sequence of steps to get a PASCAL program. For example, in some steps we can substitute ⟨*expression*⟩ and ⟨*statement*⟩ by $x + 4 \leq y - 3$ and $x := y * 3$, respectively, which gives the program part

$$\textbf{if } x + 4 \leq y - 3 \textbf{ then } x := y * 3$$

To realize automatic translations of natural languages into each other or automatic compilation of a high-level programming language into a machine language, it is necessary to develop formal concepts, called formal grammars and languages, and methods for such substitution processes describing features of grammars for natural languages and manuals for programming languages.

On one hand, the rules of the model cannot be too general because we have to be able to solve some problems within the model. For example, there has to be an algorithm which checks whether or not a given sentence is syntactically correct within the model. If we do not restrict the form of the rules (type-zero grammars), then one can show that such an algorithm does not exist. On the other hand, the rules cannot be too simple. For instance, in the previous rules for the English language, we cannot choose the words for the ⟨*determiner*⟩ and the ⟨*common-noun*⟩ independently of each other. If we choose *person* for the ⟨*common-noun*⟩, then we can take *the* or *a* for the ⟨*determiner*⟩ but not *an.* We have to take into consideration some context conditions.

The same holds for programming languages. For example, variables used at the end of the program must be already declared in the program heading. Note that in the case of English the context mentioned is local whereas it is global in PASCAL. In this article we consider context-sensitive grammars as an approach satisfying these requirements. Such grammars use local contexts. However, they can simulate global contexts by local contexts.

Now we give a further motivation for the study of context-sensitive languages. To ensure efficiency of computations, one is interested in computations that use only bounded resources. Special attention is given to computations that are limited in time and/or space (e.g., storage). If one considers computations by Turing machines (which is the most general model of computations), then context-sensitive languages form the class of problems solvable with the restriction that the space of the computation is bounded by a linear function in the size of the input. Therefore context-sensitive languages form a very natural class of languages in the framework of complexity theory.

This article is organized as follows. In the first section we give the formal definition of general phrase structure grammars, specialize it to that of context-sensitive grammars, and illustrate the concepts by some examples. In the second section we present another type of grammar called length-increasing that also characterize exactly the family of context-sensitive languages. Moreover, we present a normal form stating that any context-sensitive language can be generated by a context-sensitive grammar where the rules are of very restricted form. In the third section we introduce Turing machines and linear-bounded automata and languages accepted by these devices. We show that any context-sensitive language can be accepted by a linear-bounded automaton. Moreover, these automata accept only context-sensitive languages.

The fourth section contains a discussion of the question whether or not the application of some operations to context-sensitive languages yield context-sensitive languages again. The answer is positive with respect to union, intersection, complement, product, Kleene closure, and nonerasing morphisms whereas it is negative for erasing morphisms. In the fifth section we study decidability problems. We give an algorithm which decides whether a given word is in the language generated by a given context-sensitive grammar. Furthermore, we present three fundamental problems which cannot be solved algorithmically.

In the last section we summarize some results on context-free and regular languages that form the most important subclasses of context-sensitive languages. Thus we present only the most interesting and important results on context-sensitive languages, and mostly, we give only the basic ideas of the proofs. For more detailed information, we refer to (1) [especially to (2)], (3,4,5).

DEFINITIONS AND EXAMPLES

The aim of this section is to present the definition of context-sensitive grammars and languages and to illustrate these concepts by examples. To define a grammar, first we need two sets. The elements of one set correspond to lingistic constructs or constructs of a programming language, such as ⟨*expression*⟩, ⟨*statement*⟩. The elements of the second set represent the symbols occuring in the language or the program as digits, characters, or special words (e.g., **if, goto,** etc.). Further we need some rules that describe the possible substitutions for transforming the constructs into programs or syntactically correct sentences. Further we need some element where the transformation process starts.

We begin with some basic notions on alphabets (which describe the sets mentioned), words, and languages. An *alphabet* is a finite, nonempty set. The elements of an alphabet are called *letters* or *symbols.* A finite sequence of letters of an alphabet V is a *word* over V. Words are represented by simply writing one letter after another. The *length* of a word w denoted by $|w|$ is defined as the number of occurrences of letters in the word (each letter is counted as often as it occurs in the word). By λ we denote the empty word which corresponds to the empty sequence and contains no letter. Obviously, $|\lambda| = 0$. By V^* we designate the set of all words over V (including λ), and we set $V^+ = V^* \backslash \{\lambda\}$. Any subset L of V^* is called a *language* over the alphabet V.

We define the product w_1w_2 of two words w_1 and w_2 by simply writing w_2 after w_1. The word v is called a subword of $w \in V^*$ if $w = u_1vu_2$ holds for some $u_1, u_2 \in V^*$. As an example we consider the alphabet V consisting of the symbols a, b, c, and d, that is, $V = \{a, b, c, d\}$. Then $w = abba$, $v = acdc$, and $u = bb = b^2$ are words over V. They have lengths 4, 4, and 2, respectively. u is a subword of w. Furthermore, $uv = bbacdc$, $vu = acdcbb$ (note that $uv \neq vu$) and $w^2 = abbaabba = ab^2a^2b^2a$.

Now we give the formal definition of a general grammar as a language generating device. Later we shall give a specialization to context-sensitive grammars and languages. A (*type-zero or phrase structure*) *grammar* is a quadruple $G = (N, T, P, S)$ where N and T are disjoint alphabets, P is a finite subset of $(V^* \backslash T^*) \times V^*$, where $V = N \cup T$, and S is an element of N.

The elements of N and T are called *nonterminals* and *terminals,* respectively. The elements of P are called rules. For a pair (α, β) in P, we shall write $\alpha \to \beta$ in what follows because this expresses the intuition that a step of a derivation is a substitution. S is the axiom from which the derivation process starts. Given a grammar G as above and two words w and v over V, we say that w *directly derives* v, written as $w \Rightarrow v$, if there are a rule $\alpha \to \beta$ in P and a decomposition of $w = w_1\alpha w_2$ such that $v = w_1\beta w_2$. Intuitively, a derivation step $w \Rightarrow v$, according to a rule $\alpha \to \beta$, is the substitution of an occurrence of α in w by β.

The *language* $L(G)$ *generated by* G is defined as the set of all words $z \in T^*$ such that $S \Rightarrow z$ or there are an integer $n \geq 1$ and words $w_1, w_2, \ldots, w_n$ over V such that

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n \Rightarrow z$$

Thus the language generated consists of all words z over the terminal alphabet that can be obtained by a sequence of direct derivation steps from the axiom.

A language $L \subseteq T^*$ is called a *type-zero language* if there is a type-zero grammar $G = (N, T, P, S)$ such that $L = L(G)$. As a first example we consider the grammar $G_1 = (N_1, T_1, P_1, S_1)$ with

$$N_1 = \{S_1, S_1'\}, \quad T_1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

(T_1 is the set of digits) and

$$P_1 = \{S_1 \to xS_1' : x \in T_1 \backslash \{0\}\} \cup \{S_1' \to xS_1' : x \in T_1\} \cup \{S_1' \to \lambda\}$$

Then any derivation has the form

$$\begin{aligned} S_1 &\Rightarrow x_1S_1' \Rightarrow x_1x_2S_1' \Rightarrow x_1x_2x_3S_1' \Rightarrow \cdots \\ &\Rightarrow x_1x_2x_3\ldots x_nS_1' \Rightarrow x_1x_2x_3\ldots x_n \end{aligned}$$

with $x_1 \in \{1, 2, \ldots, 9\}$ and $x_i \in \{0, 1, 2, \ldots, 9\}$ for $2 \leq i \leq n$, that is, the generated word is a sequence of digits where the first digit is different from 0. (By the rules for S_1, we exclude zero and leading zeros.) Thus the generated language $L(G_1)$ is the set of all positive integers in decimal representation. The grammar $G_2 = (N_2, T_2, P_2, S_2)$ with

$$\begin{aligned} N_2 &= \{S_2, A, A', B, C, C', D, D'\} \\ T_2 &= \{a, b, c\} \end{aligned}$$

and

$$\begin{aligned} P_2 = \{&S_2 \to abc, S_2 \to aaABBc, Ac \to A'cc, Ac \to cc, aA' \to aaAB, \\ &B \to b, AB \to CB, CB \to CD, CD \to BD, BD \to BA \\ &BA' \to C'A', C'A' \to C'D', C'D' \to A'D', A'D' \to A'B\} \end{aligned}$$

generates the language

$$L(G_2) = \{a^nb^nc^n : n \geq 1\}$$

This can be seen as follows. If we use the first rule, we obtain abc. Let us assume that $a^nAB^nc^{n-1}$, $n \geq 2$, is already generated (by the second rule we get such a word with $n = 2$). Besides $B \to b$, we can apply only the four rules of the second line of P_2 in succession, which yields an exchange of B and A, because we perform the derivation

$$\begin{aligned} a^nABB^{n-1}c^{n-1} &\Rightarrow a^nCBB^{n-1}c^{n-1} \Rightarrow a^nCDB^{n-1}c^{n-1} \\ &\Rightarrow a^nBDB^{n-1}c^{n-1} \Rightarrow a^nBAB^{n-1}c^{n-1} \end{aligned}$$

We apply these four rules again and again, thus moving A to the right until $a^nB^nAc^{n-1}$ is obtained.

Now we can apply $Ac \to cc$ or $Ac \to A'cc$. In the former case we obtain $a^nB^nc^n$. In the latter case we derive $a^nB^nA'c^n$, and move A' to the left by iterated application of the four rules of the third line of P_2 until $a^nA'B^nc^n$ is obtained from which we generate $a^{n+1}AB^{n+1}c^n$ by applying $aA' \to aaAB$, that is, we have increased the exponents by one and can iterate the derivation.

To terminate a derivation, we apply the rule $B \to b$ to any occurrence of B and derive $a^nb^nc^n$. If we apply this rule at an earlier step, then the shifting of A or A' is blocked, and we cannot terminate the derivation.

A further example is given by the grammar $G_3 = (N_3, T_3, P_3, S_3)$ with

$$\begin{aligned} N_3 &= \{S_3, S_3', X, Y, Z\}, \quad T_3 = \{a, b, c\} \\ P_3 &= \{S_3 \to abc, S_3 \to S_3', S_3' \to aS_3'XY, S_3' \to aZX \\ &\quad YX \to XY, ZX \to bZ, ZY \to cZ', Z'Y \to cZ', Z'Y \to cc\} \end{aligned}$$

Using the first rule, we generate abc. If we apply the second rule, then the third rule n times, $n \geq 1$, and then the fourth rule, we obtain $a^{n+1}ZX(XY)^n$. By the exchange rule $YX \to XY$, we order the letters and obtain $a^{n+1}ZX^{n+1}Y^n$. Using the rule $ZX \to bZ\,(n + 1)$ times, the rule $ZY \to cZ'$ once, the rule $Z'Y \to cZ'$ $(n - 2)$ times, and finally $Z'Y \to cc$, we move the letters Z and Z', respectively, to the right, replace any X by b, any Y by c, and finally $Z'Y$ by cc, which yields $a^{n+1}b^{n+1}c^{n+1}$. Besides the order in which the rules are used, it is easy to see that this is the only way to generate a terminal word. Hence $L(G_3) = \{a^nb^nc^n : n \geq 1\}$, too.

We mention that the family of type-zero languages is the most general and universal family in the following sense. Any family of languages generated by some algorithmic device (as grammars, automata, domains of some computable functions, etc.) is contained in or equal to the family of type-zero languages.

Now we define context-sensitive grammars. A grammar $G = (N, T, P, S)$ is called *context-sensitive* or *type-one* if all rules of P are of the form $uAv \to uwv$ where $u, v \in V^*$, $w \in V^+$, and $A \in N$. By a rule $uAv \to uwv$ of a context-sensitive grammar, only the nonterminal A is substituted by a nonempty word w. This substitution, however, is allowed only if the words u and v occur in the word before and after A, respectively. The words u and v are the (left and right) contexts of A. Note that the contexts can be an empty word. Thus $uA \to uw$, $Av \to wv$, and $A \to w$ are context-sensitive rules where one context or both contexts are empty.

A language $L \subseteq T^*$ is called *context-sensitive* if there is a context-sensitive grammar $G = (N, T, P, S)$ such that $L = L(G)$. G_1 is not context-sensitive because its set P_1 of rules contains the erasing rule $S' \to \lambda$, where the right-hand side is not in V^+. We note, however, that the language $L(G_1)$ of all positive integers in decimal representation is a context-sensitive language because the grammar $G'_1 = (N_1, T_1, P'_1, S_1)$ with

$$P = \{S_1 \to x : x \in T_1 \backslash \{0\}\} \cup \{S_1 \to xS'_1 : x \in T_1 \backslash \{0\}\}$$
$$\cup \{S'_1 \to xS'_1 : x \in T_1\} \cup \{S'_1 \to x : x \in T_1\}$$

is a context-sensitive grammar (all contexts are empty) and generates all decimal representations of positive integers. G_2 is context-sensitive. G_3 is not a context-sensitive grammar. For example, $YX \to XY$ does not have the form required for context-sensitive grammars.

By definition, context-sensitive grammars cannot generate an empty word. If one is interested in the generation of λ, then one can use the following modification of the definition. We allow the exception $S \to \lambda$ for the axiom S and require that S does not occur in the right-hand side of a rule. Hence the exception rule can be used only in the first step of a derivation, that is it can be used only to add an empty word to the language.

GRAMMATICAL CHARACTERIZATIONS

In this section we present another type of grammar that also generates context-sensitive languages and give some normal forms for context-sensitive grammars.

A phrase-structure grammar $G = (N, T, P, S)$ is called *length-increasing* if $|\alpha| \leq |\beta|$ holds for any rule $\alpha \to \beta \in P$. For the grammars G_1, G_2, and G_3 of the preceding section, we find that G_1 is not length-increasing and that G_2 and G_3 are length-increasing grammars. By $w \in V^+$, $|uAv| \leq |uwv|$ holds for any rule $uAv \to uwv$ of a context-sensitive grammar. Thus any context-sensitive grammar is also length-increasing. Hence any context-sensitive language is also a length-increasing language. Now we show by a simulation technique that the converse statement is also true.

Let L be a length-increasing language. Then $L = L(G)$ holds for some length-increasing grammar $G = (N, T, P, S)$. First we construct another length-increasing grammar $G' = (N', T, P', S)$ in the following way. With any terminal $a \in T$, we associate a new nonterminal X_a and set

$$N' = N \cup \{X_a : a \in T\}$$

Further, for a rule p of P, we define p' as the rule obtained from p by replacing any occurrence of a terminal a in p by X_a and set

$$P' = \{p' : p \in P\} \cup \{X_a \to a\}$$

Then there is a derivation

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n \in T^+$$

in G if and only if there is a derivation

$$S \Rightarrow w'_1 \Rightarrow w'_2 \Rightarrow \cdots \Rightarrow w'_n$$

in G' where, for $1 \leq i \leq n$, the intermediate word w'_i is obtained from w_i by replacing any occurrence of a terminal a in w_i by X_a. Finally, we replace any occurrence of X_a in w_n by a according to the rules $X_a \to a$ which yields w_n. Therefore any word of $L(G)$ also belongs to $L(G')$, that is $L(G) \subseteq L(G')$. Moreover, up to the order of the application of the rules, only such derivations are possible in G', which proves the converse inclusion $L(G') \subseteq L(G)$. Thus $L(G') = L(G)$.

The second step of the construction is the definition of a context-sensitive grammar $G'' = (N'', T, P'', S)$ such that $L(G'') = L(G')$ and therefore $L(G'') = L(G)$ hold. The rules $X_a \to a$ of P' already have the desired context-sensitive form and are taken to P''. Therefore let $q = A_1A_2 \ldots A_r \to B_1B_2 \ldots B_s$ be a length-increasing rule of P'. Obviously, $1 \leq r \leq s$. If $r = s = 1$, q has a context-sensitive form and is taken to P''. Otherwise, we add the following rules associated with q to P'':

$$A_1 \to Y_{q,1} \text{ if } r = 1 \quad \text{and} \quad A_1 \to X_{q,1} \text{ if } r \geq 2$$
$$X_{q,1}A_2 \to X_{q,1}X_{q,2},\ X_{q,2}A_3 \to X_{q,2}X_{q,3}, \ldots,$$
$$X_{q,r-2}A_{r-1} \to X_{q,r-2}X_{q,r-1},\ X_{q,r-1}A_r \to X_{q,r-1}X_{q,r},$$
$$X_{q,1}X_{q,2} \to B_1X_{q,2},\ B_1X_{q,2},\ X_{q,2}X_{q,3} \to B_2X_{q,3}, \ldots,$$
$$X_{q,r-2}X_{q,r-1} \to B_{r-2}X_{q,r-1},\ X_{q,r-1}X_{q,r} \to B_{r-1}Y_{q,r}$$
$$Y_{q,r} \to B_rY_{q,r+1},\ Y_{q,r+1} \to B_{r+1}Y_{q,r+2}, \ldots,$$
$$Y_{q,s-2} \to B_{s-2}Y_{q,s-1},\ Y_{q,s-1} \to B_{s-1}B_s$$

where the letters $X_{q,t}$ and $Y_{q,k}$ are not in N, are pairwise different, and are added to the set of nonterminals. Obviously, all of these rules are context-sensitive. If we apply all of these rules in succession to the word $A_1A_2 \ldots A_r$, we derive $B_1B_2 \ldots B_s$. Moreover, if we start with the first rule of such a group, then we have to apply all rules and thus to simulate the application of q. Therefore we obtain $L(G'') = L(G') = L(G) = L$. This shows that L can be generated by a context-sensitive grammar, that is that L is a context-sensitive language. In summary, the following statement has been proved: A language is generated by a length-increasing grammar if and only if it is context-sensitive.

But the construction of the context-sensitive grammar previously given leads to a grammar with rules of a very special type known as the Kuroda normal form. *For any context-sensitive language L over T, there is a context-sensitive grammar $G = (N, T, P, S)$ such that any rule of P has one of the following forms:*

$$A \to a,\ A \to BC,\ AB \to AC\ AB \to CB \text{ with } A, B, C \in N,\ a \in T$$

In the case of length-increasing grammars, we can combine the last two types of rules into a rule of type $AB \to CD$ (which is not a context-sensitive rule). In the normal form presented, there are rules with left context and rules with right context. This can be improved to the following normal form which uses no rules with (nonempty) left contexts (in our formulation, an analogous statement without right contexts is also valid). For a proof we refer to (6). *For any context-sensitive language L over T, there is a context-sensitive grammar $G = (N, T, P, S)$ such that any rule of P has one of the following forms:*

$$A \to a,\ A \to BC,\ AB \to AC \text{ with } A, B, C \in N,\ a \in T$$

CHARACTERIZATION BY AUTOMATA

Whereas in the preceding section we discussed different types of grammars generating context-sensitive languages, in this section we define a special type of automata which accepts the context-sensitive languages exactly. We start with an informal definition of a more general type of automata introduced in a slightly different form by Alan Turing in (7). For a completely formal definition of the automata we refer to (4) and (5).

A *Turing machine* consists of

- an infinite input tape divided into cells that can store symbols from the input alphabet X and the blank symbol $*$ (representing an empty cell);
- a head that can read a symbol in a cell of the input tape and can move to the neighboring cells or stay in its position;
- an infinite work tape divided into cells that can store symbols from the work alphabet Y and the blank symbol $*$;
- a head that can read a symbol in a cell of the work tape, can write a symbol into a cell of the work tape, and can move to the neighboring cells or stay in its position;
- a register storing a state of a finite set Z of states that contain a special initial state z_0 and a special subset F of final states; and
- a control unit that realizes the following instruction mapping:

$$\delta : (Z \backslash F) \times (X \cup \{*\}) \times (Y \cup \{*\}) \to \mathscr{P}[Z \times (Y \cup \{*\}) \times \{R, L, N\} \times \{R, L, N\}]$$

$(z', y', m_1, m_2) \in \delta(z, x, y)$ has the following meaning: if the current state of the register is z, the head reads x in the cell c of the input tape and the other head reads y in the cell c' of the work tape, then the machine changes the contents of the register to the state z', writes y' into the cell c' of the work tape, moves the head of the input tape from cell c to its right neighbor if $m_1 = R$, to its left neighbor if $m_1 = L$, and performs no move if $m_1 = N$, and moves the head of the work tape from the cell c' according to $m_2 \in \{R, L, N\}$. A Turing machine is illustrated by Fig. 1.

A computation of Turing machine $\mathscr{M}$ (given by the above components) on a nonempty word w over X is done as follows:

- initially the input tape contains a nonempty word w over X in some cells in succession and the remaining cells are filled with the blank symbol; the head of the input tape enters the cell that contains the first letter of w; the work tape is completely filled with blank symbols; and the register contains the state z_0;
- changes of the work tape, of the head positions, and of the register are done according to the instruction mapping,
- the machine stops its computation if a final state $z \in F$ is obtained.

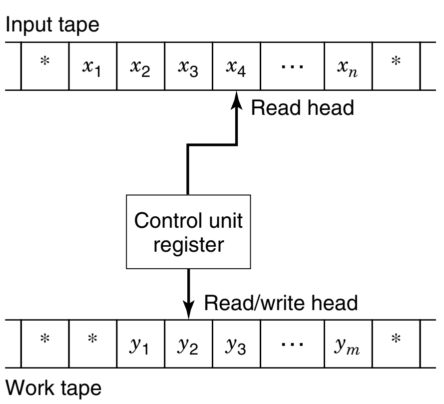

Figure 1. Scheme of a Turing machine.

Note that $\mathscr{M}$ can perform some computations on a word because $\delta(z, x, y)$ is a finite set, and hence some reactions to a given state and given symbols read at the input and work tape are possible. Hence this machine works nondeterministically.

The *language $T(\mathscr{M})$ accepted by a Turing machine $\mathscr{M}$* is defined as the set of words for which there is a computation of the Turing machine $\mathscr{M}$ on w that stops after a finite number of steps. One can show that a language is accepted by a Turing machine if and only if it can be generated by a type-zero grammar.

A Turing machine $\mathscr{M}$ is called a *linear-bounded automaton* if there is a constant c such that, for any word w of length n and any computation of $\mathscr{M}$ on w, the head of the work tape enters at most $c \cdot n$ different cells.

For any context-sensitive language L, there is a linear-bounded automaton which accepts L, and conversely, any language accepted by a linear-bounded automaton is context-sensitive. We prove only the first part of this statement. Let L be an arbitrary context-sensitive language. Let G be a length-increasing grammar $G = (N, T, P, S)$ in Kuroda normal form with $L(G) = L$. Then we construct the Turing machine M with the input alphabet T, the work alphabet $N \cup T$, and states and instructions such that the following steps can be carried out:

1. $\mathscr{M}$ copies the contents w of the input tape to the work tape.
2. $\mathscr{M}$ checks whether or not S is the content of the work tape. If the answer is affermative, $\mathscr{M}$ enters a final state, that $\mathscr{M}$ accepts the input word w.
3. $\mathscr{M}$ nondeterministically chooses a rule $A \to a$ or $AB \to CD$ or $A \to CD$ of P (this can be done using states) and searches for a in some cell or CD in some neighboring cells, respectively. If it does not find a or CD, respectively, then $\mathscr{M}$ enters a special state that preserves the situation. Otherwise, $\mathscr{M}$ substitutes a by A or CD by AB or $A*$, respectively, and in the latter case $\mathscr{M}$ shifts the subword following the introduced $*$ one cell to the left.

Steps 2 and 3 are performed alternately as long as no final state is entered.

By this construction, step 3 is the simulation of a derivation step in G. If $v' \Rightarrow v$ holds in G, then $\mathscr{M}$ transforms v on the work tape into v'. Thus we have a derivation

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_{n-1} \Rightarrow w_n = w$$

in G if and only if the input word w on the work tape is transformed by step 3 of $\mathscr{M}$ in succession into the words w_{n-1}, w_{n-2}, . . ., w_2, w_1, S. Hence $w \in L(G)$ if and only if w is accepted by $\mathcal{M}$. Therefore $L = L(G) = T(M)$. Moreover, $\mathcal{M}$ enters at most $(n + 2)$ cells of the work tape. n cells are needed for the copy of the input word. To recognize the beginning and ending of the word, one has to enter the cells before and after the word. Step 2 does not change the length of the word on the work tape, and step 3 does not increase its length. Thus $\mathcal{M}$ is a linear-bounded automaton.

By definition Turing machines and linear-bounded automata are nondeterministic because $\delta(z, x, y)$ is a finite set. We obtain deterministic versions if we require that, for any $z \in Z \setminus F$, $x \in X$ and $y \in Y$, $\delta(z, x, y)$ contains exactly one element.

In the case of Turing machines we can show that the restriction to deterministic machines does not decrease the power. A language can be accepted by a (nondeterministic) Turing machine if and only if it can be accepted by a deterministic Turing machine. Such a relationship is not known for linear-bounded automata so far. Because the deterministic linear-bounded automata are special (nondeterministic) linear-bounded automata, the deterministic linear-bounded automata accept context-sensitive languages. It is an open problem whether or not deterministic linear-bounded automata can accept all context-sensitive languages.

OPERATIONS ON CONTEXT-SENSITIVE LANGUAGES

In this section we consider again the question whether the application of an operation to context-sensitive languages yields a context-sensitive language. We consider this problem for the set-theoretic operations as union, intersection, complement and algebraic operations as product, Kleene closure and homomorphisms.

The first statement shows that the family of context-sensitive languages has positive properties with respect to the set-theoretic operations previously mentioned. *Let L_1 and L_2 be two arbitrary context-sensitive languages over an alphabet T. Then $L_1 \cup L_2$, $L_1 \cap L_2$ and $T^+ \setminus L_1$ are also context-sensitive languages.*

To prove the statement for the union, we consider context-sensitive grammars $G_1 = (N_1, T, P_1, S_1)$ and $G_2 = (N_2, T, P_2, S_2)$ with $L(G_1) = L_1$ and $L(G_2) = L_2$ and assume (without loss of generality) that N_1 and N_2 are disjoint sets (if necessary, we rename the nonterminals). Then we construct the context-sensitive grammar

$$G = (N_1 \cup N_2 \cup \{S\}, T, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}, S)$$

Let $S \Rightarrow S_1 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n = w \in T^*$ be a derivation in G. By construction, besides the first step we can apply only rules from P_1, that is, $S_1 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n = w$ is a derivation in G_1. Hence $w \in L(G_1)$. Analogously, if we start the derivation by applying $S \to S_2$, then we generate a word $v \in L(G_2)$. Therefore $L(G) = L(G_1) \cup L(G_2) = L_1 \cup L_2$.

With respect to intersection we start with two linear-bounded automata $\mathcal{M}_1$ and $\mathcal{M}_2$ with $T(\mathcal{M}_1) = L_1$ and $T(\mathcal{M}_2) = L_2$ and construct the linear-bounded automaton $\mathcal{M}$ that works as follows (we give only an informal description for reasons of space):

- First $\mathcal{M}$ works as $\mathcal{M}_1$ on the input w of length n using at most $c \cdot n$ cells of the work tape for some constant c;
- If w is not accepted by $\mathcal{M}_1$, $\mathcal{M}$ enters a nonfinal state that cannot be changed by $\mathcal{M}$, that is, $\mathcal{M}$ does not stop the work (w is not in L_1 and hence not in the intersection); if w is accepted by $\mathcal{M}_1$, $\mathcal{M}$ deletes all symbols at the work tape;
- Finally, $\mathcal{M}$ works as $\mathcal{M}_2$ on w using at most $d \cdot n$ cells for some constant d and accepts if and only if $\mathcal{M}_2$ accepts.

Obviously, $\mathcal{M}$ accepts w if and only if both $\mathcal{M}_1$ and $\mathcal{M}_2$ accept w, that is if and only if w is contained in L_1 and also in L_2. Moreover, the computation uses at most $\max\{c, d\} \cdot n$ cells of the work tape.

The problem of whether $T^+ \setminus L$ is context-sensitive for a context-sensitive language was posed in the sixties and solved independently by N. Immerman (8) and R. Szelepcsenyi (9) in 1988. We omit the technically complicated proof for this statement (for reasons of space) and refer to (2,8,9).

Now we define algebraic operations which are often used in the theory of formal languages. The product $L_1 \cdot L_2$ of two languages is defined as

$$L_1 \cdot L_2 = \{w_1 w_2 : w_1 \in L_1, w_2 \in L_2\}$$

For a language L and an integer $n \geq 1$, we define L^n inductively by

$$L^1 = L$$

and

$$L^{i+1} = L^i \cdot L \text{ for } i \geq 1$$

and the Kleene closure L^+ by

$$L^+ = \bigcup_{i \geq 1} L^i = \{v_1 v_2 \ldots v_i : i \geq 1, v_j \in L \text{ for } 1 \leq j \leq i\}$$

With respect to these two operations we have the following result. *For any two context-sensitive languages L_1 and L_2, their product $L_1 \cdot L_2$ and the Kleene closure L_1^+ are also context-sensitive languages.* If $G_1 = (N_1, T, P_1, S_1)$ and $G_2 = (N_2, T, P_2, S_2)$ are two context-sensitive grammars in Kuroda normal form with disjoint alphabets of nonterminals generating L_1 and L_2, respectively, then

$$G = (N_1 \cup N_2 \cup \{S\}, T, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S)$$

with $S \notin N_1 \cup N_2$ generates $L_1 \cdot L_2$.

Furthermore, the grammar $G' = (N_1 \cup \{S, S'\}, T, P_1', S)$ with

$$P'_1 = P_1 \cup \{S \to S_1, S \to S_1 S'\} \cup \bigcup_{a \in T} \{aS' \to aS_1 S', aS' \to aS_1\}$$

generates L_1^+ because a typical derivation in G' is given by

$$\begin{aligned} S &\Rightarrow S_1 S' \Rightarrow \cdots \Rightarrow v_1 a_1 S' \Rightarrow v_1 a_1 S_1 S' \Rightarrow \cdots \\ &\Rightarrow v_1 a_1 v_2 a_2 S' \Rightarrow v_1 a_1 v_2 a_2 S_1 S' \Rightarrow \cdots \\ &\Rightarrow v_1 a_1 \ldots v_{i-1} a_{i-1} S' \Rightarrow v_1 a_1 \ldots v_{i-1} a_{i-1} S_1 \Rightarrow \cdots \\ &\Rightarrow v_1 a_1 \ldots v_{i-1} a_{i-1} v_1 a_i \end{aligned}$$

where, for $1 \leq j \leq i$, the derivations $S_1 \Rightarrow \cdots \Rightarrow v_j a_j$ with $v_j \in T^*$ and $a_j \in T$ also hold in G_1. Such a derivation generates $v_1a_1v_2a_2 \ldots v_ia_i \in L_1^+$, and up to the order of the applications of rules we have only such derivations. Let X and Y be two alphabets. A mapping h from X^* to Y^* is called a *morphism* if the following conditions are satisfied:

- $h(\lambda) = \lambda$;
- For any $x \in X$, $h(x)$ is a word over Y;
- For any two words w and v over X, $h(wv) = h(w)h(v)$.

By the third condition it is sufficient to give the image of any letter $x \in X$ under h and to extend this by $h(x_1x_2 \ldots x_n) = h(x_1)h(x_2) \ldots h(x_n)$ to words. Moreover, we extend a morphism $h: X^* \to Y^*$ to a language L over X by

$$h(L) = \{h(w) : w \in L\}$$

We call a morphism *nonerasing* if, for any nonempty word w over X, $h(w) \neq \lambda$ holds. h is nonerasing iff $h(x) \neq \lambda$ holds for any letter $x \in X$. We call a *morphism* a *weak coding* if, for any letter x of X, $h(x) \in Y$ or $h(x) = \lambda$ holds.

For a context-sensitive grammar $G = (N, T, P, S)$ in Kuroda normal form, such that $L(G) = L$, and for a nonerasing morphism h, let $G' = (N, T, P', S)$ be the grammar where P' is obtained from P by substituting any rule $A \to a$ by $A \to h(a)$. Then G' generates $h[L(G)]$. Thus the following statement is valid. *If L is a context-sensitive language and h is a nonerasing morphism, then $h(L)$ is also a context-sensitive language.*

If h is an erasing morphism, then the grammar G' constructed previously is not context-sensitive. Moreover, the statement is no longer true for erasing morphisms. This follows from the fact that there are type-zero languages that are not context-sensitive and from the following consideration.

Let $H = (N, T, P, S)$ be an arbitrary type-zero grammar. We construct the length-increasing grammar $H' = (N \cup \{\$\}, T \cup \{\S\}, P', S)$, where \$ is an additional nonterminal and § is an additional terminal, as follows: Let $p = \alpha \to \beta$ be a rule of P. If $|\alpha| \leq |\beta|$, then we incorporate p in P', and if $|\alpha| > |\beta|$, then we add $\alpha \to \beta\$^{|\alpha|-|\beta|}$ to P. Moreover, we add to P' the rule $\$ \to \S$ and all rules $X\$ \to \$X$ and $\$X \to X\$$ with $X \in N \cup T$. Note that all rules of P' are length-increasing. Obviously, H and H' generate the same words up to occurrences of §. Hence we obtain $h[L(H')] = L(H)$ for the weak coding h where $h(a) = a$ for $a \in T$ and $h(\S) = \lambda$.

Formulated in terms of languages instead of grammars, we obtain the following statement. *For any type-zero language L, there are a context-sensitive language L' and a weak coding h such that $L = h(L')$.*

DECISION PROBLEMS

One of the most important questions about a given program is whether or not the program is syntactically correct. Formally this means whether or not a word (program) w belongs to a language L (a set of syntactically correct programs). Therefore the previous question can be formulated as follows:

- *Membership Problem.* Given a grammar G and a word w over the terminal alphabet of G, decide whether or not $w \in L(G)$ holds.

Besides this central problem we shall also discuss the following problems:

- *Emptiness Problem.* Given a grammar G, decide whether or not $L(G)$ is empty [i.e., $L(G)$ contains no word].
- *Finiteness Problem.* Given a grammar G, decide whether or not $L(G)$ is a finite set.
- *Equivalence Problem.* Given two grammars G_1 and G_2, decide whether or not $L(G_1) = L(G_2)$ holds (i.e., whether both grammars generate the same language).

We discuss only the existence of algorithms which solve the problems given previously for context-sensitive grammars. [By an algorithm we mean a sequence of commands such that any command can be carried out without intelligence, any command has a uniquely determined successor command, there is a uniquely determined first command, and the algorithm stops with a special command. For a more formal definition of an algorithm, we refer to (11) and (12). The most general formalization can be given by means of Turing machines with an additional output tape; such machines induce functions which map the word on the input tape to the word on the output tape; a function is called algorithmically computable if it can be induced by a Turing machine; functions not defined on words can be handled by codings.]

First, we note that, for any of the problems mentioned, there is no algorithm that solves the problem for type-zero grammars. With respect to context-sensitive grammars, the situation is slightly better. *For context-sensitive grammars, there is an algorithm that solves the membership problem, but there are no algorithms that solve the emptiness, finiteness, and equivalence problems.*

We present an algorithm for the membership problem. Let the context-sensitive (length-increasing) grammar $G = (N, T, P, S)$ and $w \in T^+$ of length n be given, and let c and d be the cardinalities of the sets $V = N \cup T$ and P, respectively. Let us assume that there is a derivation

$$S = w_0 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_r = w$$

in G. If $w_i = w_j$ holds for some integers i, j with $0 \leq i < j \leq n$, then there is also a derivation

$$S = w_0 \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_i \Rightarrow w_{j+1} \\ \Rightarrow w_{j+2} \Rightarrow \cdots \Rightarrow w_r = w$$

in G. Therefore we can assume that there is a derivation for w in G that contains any word at most once. Let $z_0 \Rightarrow z_1 \Rightarrow z_2 \Rightarrow \cdots \Rightarrow z_m$ be a derivation in G containing no word twice and starting with z_0 of length s. Because of the length increase of the rules of G, $|z_i| \leq |z_{i+1}|$ for $i \geq 0$. However, the equality of the lengths can hold at most c^s steps because there only exist c^s different words of length s over V.

Hence there is a derivation of w with at most

$$\sum_{s=1}^{n} c^s = \frac{c^{n+1} - c}{c - 1} \leq c^{n+1}$$

steps. Since we can apply one of the d rules in any step, there are at most $d^{c^{n+1}}$ different derivations with c^{n+1} steps. We perform all these (finitely many) derivations. If w is generated

Table 1. Closure Properties with Respect to Operations[a]

	Union	Intersection	Complement	Product	Kleene Closure	Morphism
Type-zero	+	+	−	+	+	+
Context-sensitive	+	+	+	+	+	−
Context-free	+	−	−	+	+	+
Regular	+	+	+	+	+	+

[a] + means that the application of the operation to context-free (regular, etc.) languages yields a context-free (regular, etc.) language, again, whereas − means that there are context-free (regular, etc.) languages such that the application of the operation yields a noncontext-free (nonregular, etc.) language.

by one of these derivations, then $w \in L(G)$ holds. Otherwise, $w \notin L(G)$. Obviously, the algorithm presented requires an (super)exponential number of steps in the worst cases. We note that no algorithm is known so far where the number of steps is a polynomial in the length of word.

The proofs for the nonexistence of algorithms for the other three problems are given by reduction, that is we show that the existence of an algorithm for one of these problems implies the existence of an algorithm for another problem for which there is no algorithmic solution.

First, let us assume that there is an algorithm for the emptiness problem for context-sensitive (length-increasing) grammars. Then we consider an arbitrary type-zero grammar G and the context-sensitive grammar G' such that $L(G) = h(L(G')$ for some homomorphism h (see the preceding section). Obviously, $L(\mathrm{G})$ is empty if and only if $L(G')$ is empty because, by assumption, there is an algorithm that decides the emptiness of $L(G')$ and thus the emptiness of $L(G)$. We have already mentioned, however, that there is no algorithm for the emptiness problem for type-zero grammars. Therefore our assumption has to be false.

Now let us assume that there is an algorithm for the finiteness problem for context-sensitive grammars. Then we consider an arbitrary context-sensitive grammar G and construct a context-sensitive grammar G' such that $L(G') = L(G) \cdot T^+$ (T^+ is generated by the grammar

$$G'' = (\{S\}, T, \{S \to aS : a \in T\} \cup \{S \to a : a \in T\}, S)$$

and for the product see the preceding section). If $L(G)$ is not empty, then $L(G')$ is infinite, and if $L(G)$ is empty, then $L(G')$ is empty (and finite), too. Therefore $L(G')$ is finite if and only if $L(G)$ is empty. Hence the existence of an algorithm deciding the finiteness of $L(G')$ implies the existence of an algorithm deciding the emptiness of $L(G)$ which does not exist.

Now let us assume that there is algorithm for the equivalence problem for context-sensitive grammars. Because the equality $T^+ = T^+ \backslash L(G)$ holds if and only if $L(G)$ is empty, we can reduce the equivalence problem to the emptiness problem which cannot be solved algorithmically.

TWO SPECIAL CASES

As we have seen in the preceding section, the family of context-sensitive languages has some negative features with respect to the existence of algorithms for the most important problems. Thus it is of interest to consider special cases with better properties and with sufficient power for some applications. We discuss here only context-free and regular languages. Moreover, we just present some definitions and results and omit justifications for which we refer to (3,4,5,12–14).

A grammar $G = (N, T, P, S)$ is called *context-free* if any rule of P has the form

$$A \to w \quad \text{with} \quad A \in N \text{ and } w \in (N \cup T)^*$$

A grammar $G = (N, T, P, S)$ is called *regular* if any rule of P has the form

$$A \to w \text{ or } A \to wB \quad \text{with} \quad A, B \in N \text{ and } w \in T^*$$

A language L is called context-free (or regular) if there is a context-free (or regular) grammar G such that $L(G) = L$. The grammar G_1 for the decimal presentation of positive integers presented in the first section is regular. Obviously, any regular grammar is also context-free. Hence, any regular languages is also context-free. The converse relationship is not true. $\{a^n b^n : n \geq 1\}$ is a context-free language which is not regular.

Context-free and regular grammars allow erasing rules $A \to \lambda$ that are not context-sensitive (and not length-increasing). However, for any context-free grammar G, there is a context-free grammar G' such that $L(G') = L(G) \backslash \{\lambda\}$ (i.e., besides the empty word, G' and G generate the same terminal words) and G' has no erasing rules. Hence this grammar G' is context-sensitive, and therefore up to the empty word any con-

Table 2. Decidability Properties[a]

	Membership Problem	Emptyness Problem	Finiteness Problem	Equivalence Problem
Type-zero	−	−	−	−
Context-sensitive	+	−	−	−
Context-free	+	+	+	−
Regular	+	+	+	+

[a] + means the existence of an algorithm to solve the problem, and − means that there is no such algorithm.

text-free (and regular) language is context-sensitive. (We can also use the modification of the definition of a context-sensitive grammar with the exception erasing rule, as mentioned in the first section, which says directly that any context-free or regular language is context-sensitive.) The language $\{a^nb^nc^n : n \geq 1\}$ generated by the context-sensitive grammar G_2 in the first section is not a context-free language.

Tables 1 and 2 summarize the properties of regular and context-free languages with respect to the operations discussed above and their decidability properties. For the sake of completeness we add the results for context-sensitive and type-zero languages in both tables.

As one can see from Table 2, context-free and regular languages have much better properties than context-sensitive languages. On the other hand, there are a lot of grammatical structures and constructs in programming languages that cannot be covered by context-free grammars. Therefore there are a large number of grammar types that are more powerful than context-free grammars and behave better than context-sensitive grammars. We refer to (15,16).

BIBLIOGRAPHY

1. G. Rozenberg and A. Salomaa, *Handbook of Formal Languages,* Berlin: Springer-Verlag, 1997, vol. 1–3.
2. A. Mateescu and A. Salomaa, Aspects of classical formal language theory, in (1), vol. 1, pp. 175–251.
3. A. Salomaa, *Formal Languages,* New York: Academic Press, 1973.
4. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation,* Reading, MA: Addison-Wesley, 1979.
5. T. A. Sudkamp, *Languages and Machines,* Reading, MA: Addison-Wesley, 1988.
6. M. Penttonen, One-sided and two-sided context in formal grammars, *Inf. Control,* **25**: 371–392, 1974.
7. A. Turing, On computable numbers with an application to the Entscheidungsproblem. *Proc. London Math Soc.,* **42**: 230–265, 1936. A correction, ibid., **43**: 544–546, 1936.
8. N. Immerman, Nondeterministic space is closed under complementation, *SIAM J. Comput.,* **17**: 935–938, 1988.
9. R. Szelepcsenyi, The method of forced enumeration for nondeterministic automata, *Acta Informatica,* **26**: 279–284, 1988.
10. M. Davis, *Computability and Unsolvability,* New York: Dover, 1958 and 1982.
11. N. J. Cutland, *Computability,* Cambridge: Cambridge University Press, 1980.
12. J.-M. Autebert, J. Berstel, and L. Boasson, Context-free languages and push-down automata, in (1), vol. 1, pp. 111–174.
13. Sh. Yu, Regular languages, in (1), vol. 1, pp. 41–110.
14. M. Harrison, *Introduction to Formal Language Theory,* Reading, MA: Addison-Wesley, 1978.
15. J. Dassow and G. Paun, *Regulated Rewriting in Formal Language Theory,* EATCS Monographs in Theoretical Computer Science, Berlin: Springer-Verlag, 1989, Vol. 18.
16. J. Dassow, G. Paun, and A. Salomaa, Grammars with controlled derivations, in (1), vol. 2, pp. 101–154.

JÜRGEN DASSOW
Otto-von-Guericke-Universität
Magdeburg

CONTENT-BASED RETRIEVAL. See MULTIMEDIA INFORMATION SYSTEMS.

CONTINUATION METHODS. See HIGH DEFINITION TELEVISION; HOMOTOPY METHODS FOR COMPUTING DC OPERATING POINTS.