
Review article

Context-free parallel grammars and their applications to generating context-sensitive languages

Context-free
parallel
grammars

131

Edward T. Lee

L and P Culture and Technology, Inc., San Diego, California, USA

Introduction

Since its publication, almost four decades ago, the pioneering work of Chomsky[1] has had a profound influence on the development of modelling natural languages, in addition to stimulating much of the early works in stochastic languages[2] and fuzzy languages[3], as well as in syntactic[2], structural[4], and linguistic[5] pattern recognition. Sequential productions and sequential grammars were introduced by Chomsky and further investigation results may be found in[6,7]. In this paper, parallel productions and parallel grammars are presented and applied to generate context-sensitive languages[6,7]. Properties of parallel grammars are investigated. The advantages of using parallel grammars to describe computer programming languages[8-10] are also stated. We begin by briefly recapitulating the definitions of the four types of sequential grammars. We use the notations and terminologies as used in[6,7].

Four types of grammars

Informally, a grammar may be viewed as a set of rules for generating the elements of a set. More concretely, a grammar is a quadruple $G = (V_N, V_T, P, S)$ in which V_T is a set of terminals, V_N is a set of non-terminals. The intersection of V_T and V_N is the empty set. P is a set of sequential productions, and S is a member in V_N . Essentially, the elements of V_N are labels for certain subsets of V_T called *syntactic categories*. Informally, V_T^* represents the set of all possible finite strings composed of elements of V_T . More specifically, V_T^* denotes the Kleene closure of V_T and is defined as

$$V_T^* = \varepsilon + V_T + V_T V_T + V_T V_T V_T + V_T V_T V_T V_T + \dots \quad (1)$$

where ε is the null string[6,7], and $+$ denotes the union operation. Note that the meaning of the multiple concatenations $V_T V_T V_T, V_T V_T V_T V_T, \dots$ is unambiguous because of the associativity of concatenation. S is the label for the syntactic category "sentence". The sequential productions in P define conditioned sets in $(V_T \cup V_N)^*$ where $(V_T \cup V_N)$ denotes the union of V_T and V_N .

More concretely, the sequential productions in P are expressions of the form

$$\alpha \rightarrow \beta \quad (2)$$

where α and β are strings in $(V_T \cup V_N)^*$.

The expression $\alpha \rightarrow \beta$ represents a rewriting rule. Thus, if $\alpha \rightarrow \beta$ and γ and δ are arbitrary strings in $(V_T \cup V_N)^*$, then

$$\gamma \alpha \delta \rightarrow \gamma \beta \delta \quad (3)$$

and $\gamma \beta \delta$ is said to be *directly derivable* from $\gamma \alpha \delta$.

If $\alpha_1, \alpha_2, \dots, \alpha_{m-1}, \alpha_m$ are strings in $(V_T \cup V_N)^*$ and

$$\alpha_1 \rightarrow \alpha_2 \dots \rightarrow \alpha_{m-1} \rightarrow \alpha_m \quad (4)$$

then α_1 is said to derive α_m in grammar G , or, equivalently, α_m is *derivable* from α_1 in grammar G . This is expressed by $\alpha_1 \xrightarrow{G} \alpha_m$ or simply $\alpha_1 \Rightarrow \alpha_m$. The expression (4) will be referred to as a *derivation chain* from α_1 to α_m .

A grammar G generates a language $L(G)$ in the following manner. A string of terminals x is said to be in $L(G)$ if and only if x is derivable from S . This means that

$$\{x \text{ is a string in } L(G)\} \text{ if and only if } \{S \Rightarrow x\}. \quad (5)$$

Sequential grammars can be classified into four types.

Type 0 grammar

In this case, sequential productions are of the general form

$$\alpha \rightarrow \beta \quad (6)$$

where α and β are strings in $(V_T \cup V_N)^*$.

Type 1 grammar (context-sensitive)

Here the sequential productions are of the form

$$\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2 \quad (7)$$

with α_1, α_2 , and β in $(V_T \cup V_N)^*$, A in V_N , and $\beta \neq \epsilon$. In addition, the production $S \rightarrow \epsilon$ is allowed.

Type 2 grammar (context-free)

The allowable productions are of the form

$$A \rightarrow \beta \quad (8)$$

where A is an element in V_N , β is a string in $(V_T \cup V_N)^*$, and β is not equal to the null string ϵ . Furthermore, $S \rightarrow \epsilon$ is allowed.

Type 3 grammar (regular)

In this case the allowable productions are of the form

$$A \rightarrow a B \quad (9)$$

or

$$A \rightarrow a \quad (10)$$

where a is an element in V_T , A and B belong to V_N . In addition, $S \rightarrow \epsilon$ is allowed.

It should be clear that every regular grammar is context free; every context-free grammar is context sensitive; every context-sensitive grammar is type 0. This nested relationship of these four types of grammars is illustrated in Figure 1. The results of the investigation of the these four types of grammars forms the kernel of the theory of formal languages[6,7].

Four types of formal languages and four types of automata

The study of formal languages constitutes an important sub-area of theoretical computer science. We shall call a language that can be generated by a type 0 grammar a type 0 language. A type 0 language is also called a recursively enumerable set. A language generated by a context-sensitive, context-free or regular grammar is called a context-sensitive, context-free, or regular language respectively. This relationship is also shown in Figure 1.

In the last section, we introduced a generating scheme – the grammar. A grammar is one way of finite specification of a language. Note that a language may have infinite members. In this section another method of finitely specifying infinite-languages is introduced through the use of the recognizer. As shown in Figure 1, the recognizers for type 0, context-sensitive, context-free, and regular languages are called Turing machines, linear bounded automata, pushdown automata, and finite automata respectively. The definitions of these four types of automata and the proofs of these four correspondences may be found in[6]. The equivalences of these four types of automata with respect to their

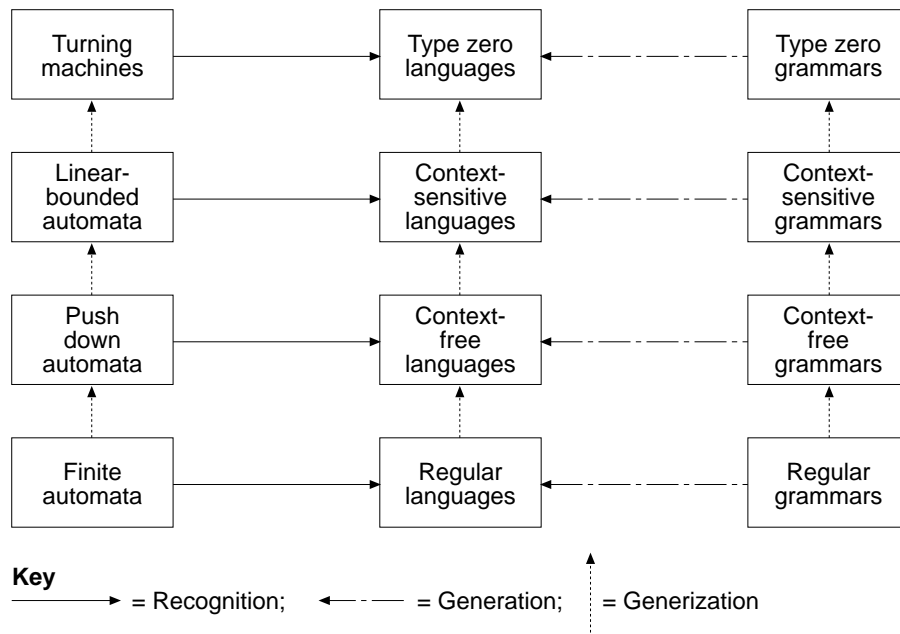


Figure 1.
Overview of formal language theory

corresponding four types of grammars are the main results in the theory of formal languages[6,7]. The results are shown concisely in Figure 1.

The relationships between computer programming languages and context-sensitive languages

In[1], Chomsky gave a mathematical model of a grammar in connection with his study of natural languages. As we know, natural languages are intrinsically context sensitive. For example, “a tall building” in downtown New York City has a different meaning compared with “a tall building” in Bradford (UK). Another good example to illustrate the context sensitive property of hand-written character recognition is shown in Figure 2. The middle characters in Figure 2(a) and 2(b) are identical, being neither character A nor character H. However, if it is preceded by character T and followed by character E, then we interpret it as an H as shown in Figure 2(a). On the other hand, if it is preceded by character C and followed by character T, then we interpret it as an A as shown in Figure 2(b). Additional Chomsky’s work in this area may be found in[11-15]. The concept of a grammar was found to be of great importance to computer programmers when the syntax of programming languages such as ALGOL may be defined by a context-free grammar. This development led naturally to syntax-directed compiling and the concept and the development of compilers[16].

A grammar where no account of context is taken when making a substitution is a context-free grammar. Context-sensitive grammars are more complex than context-free grammars because we must have provisions to express the fact that a substitution of one part of the derivation tree[6] may influence the structure of another part. Almost all languages, both natural and artificial, are context sensitive. Programming languages generally have some context sensitive rules. For example, “an identifier may not be declared more than once in a program” is a context sensitive rule.

In order to avoid the added complexity of context-sensitive languages, the syntax of programming languages is commonly defined in a context-free form with additional, usually informal, rules in order to describe the context-sensitive properties. Therefore computer programming languages may be

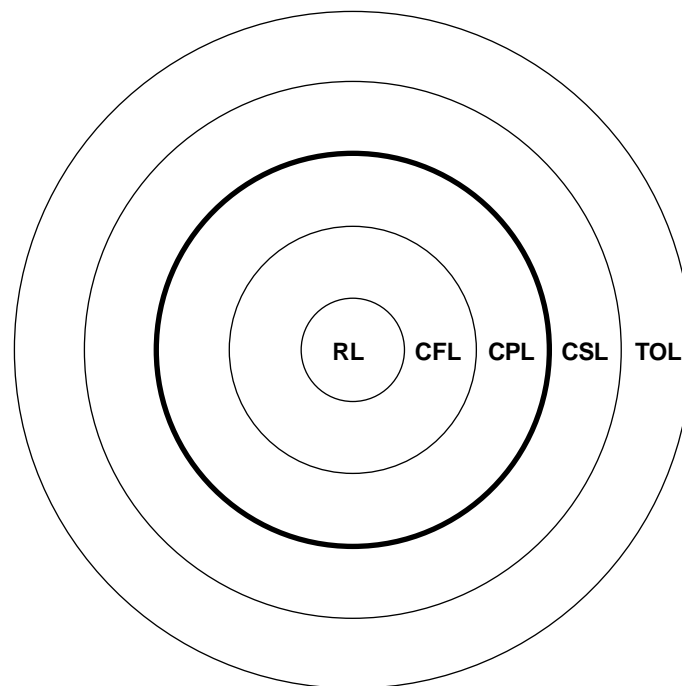
(a) T H E

(b) C A T

Figure 2.
Context-sensitive
property of recognizing
hand-written characters

represented as a dark circle between context-free languages and context-sensitive languages as shown in Figure 3. In other words, computer programming languages are usually more general than context-free languages. However, context-sensitive languages are more general than computer programming languages.

Context-free
parallel
grammars



Note: CPL = Computer programming languages

Figure 3.
The relationship
between computer
programming
languages and formal
languages

Parallel productions and parallel grammars

In what follows, parallel productions and parallel grammars are introduced in order to generate some simple context-sensitive languages.

We first briefly state the motivation, the concept, the notation, the implementation and the applications of a parallel production.

Intuitively, a parallel production models the fact that a group of productions are always applied in parallel (at the same time). In other words, these productions can always be dealt with as a single group. Thus, conceptually, productions can be treated as a set of groups. Each group is called a parallel production.

Formally, a parallel production PP_M consists of a set of productions denoted as

$$PP_m = \{P_{m.1}, P_{m.2}, \dots, P_{m.r}\} \quad (11)$$

where m represents the group number and $P_{m,1}, P_{m,2}, \dots, P_{m,n}$ are productions in group m .

When a parallel production is applied, this means that all the productions within this group are carried out in parallel. Furthermore, note that the generation process will not continue until the completion of the execution of all the productions in this group. This precedence condition is referred to as the precedence relationship of executing a parallel production.

As for the implementation, we assume that we have a parallel computer[17] which consists of at least n processors. Therefore, n processors can be assigned to work on the execution of the n productions in parallel. This means that one processor works on the execution of one production. We wish to emphasize the assumption that the generation process will not continue until the execution of all the productions in a group are completed.

The concept of a parallel production has many desirable features, especially in the picture generation areas. For example, assume that we wish to generate a picture which is symmetrical with respect to the diagonal axis. In addition, the origin is used as the starting point and the generation process is symmetrical with respect to the diagonal axis.

We further assume that P_1 is a production for the generation of this picture along the horizontal axis, and P_2 is the corresponding production along the vertical axis. Clearly, we can combine productions P_1 and P_2 in order to form a parallel production $\{P_1, P_2\}$.

In addition, the concept of a parallel production also has useful applications in two-dimensional[18,19] as well as multi-dimensional[2] grammars. Two-dimensional programmed grammars[18] are the results of applying the concepts of programmed grammars[20,21] on two-dimensional grammars. Parallel productions can also be used as an effective tool in modelling parallel processing as well as parallel distributed processing[22,23].

A parallel grammar is simply a grammar consisting of parallel productions.

Comparison between grammars and parallel grammars in terms of generating context-sensitive languages

We know that

$$L(G) = \{a^n b^n c^n \text{ where } n \geq 1\} \quad (12)$$

is a context-sensitive language.

As shown in[6], the following sequential grammar G_S can generate $\{a^n b^n c^n\}$, where

$$G_S = (V_{SN}, V_{ST}, P_S, S),$$

$$V_{SN} = \{S, B, C\},$$

$$V_{ST} = \{a, b, c\}, \text{ and}$$

P_S consists of the following seven productions:

$$S \xrightarrow{1} a S B C$$

$$\begin{aligned}
 S &\xrightarrow{2} a B C \\
 CB &\xrightarrow{3} B C \\
 a B &\xrightarrow{4} a b \\
 b B &\xrightarrow{5} b b \\
 b C &\xrightarrow{6} b c \\
 c C &\xrightarrow{7} c c.
 \end{aligned}$$

For example, $a^2 b^2 c^2$ can be generated by G_S as follows:

$$\begin{aligned}
 S &\xrightarrow{1} a \underline{S} B C \xrightarrow{2} a^2 B \underline{C} B C \xrightarrow{3} a a \underline{B} B C C \xrightarrow{4} \\
 &a^2 b \underline{B} C C \xrightarrow{5} a^2 b b \underline{C} C \xrightarrow{6} a^2 b^2 \underline{c} C \xrightarrow{7} a^2 b^2 c^2
 \end{aligned} \tag{13}$$

where the symbols that are to be replaced are underlined and the number above the arrows indicates the production that is applied.

A parallel grammar G_P which also can generate $\{a^n b^n c^n\}$ is constructed.

$$\begin{aligned}
 G_P &= (V_{PN}, V_{PT}, P_P, S) \\
 V_{PN} &= \{S, A, B, C\} \\
 V_{PT} &= \{a, b, c\},
 \end{aligned}$$

and P_P consists of one sequential production P_1 and two parallel productions PP_2 and PP_3 where each parallel production composed of three sequential productions.

$$\begin{aligned}
 P_1 &= \{S \xrightarrow{1} A B C\} \\
 PP_2 &= \{A \xrightarrow{2.1} a A, B \xrightarrow{2.2} b B, C \xrightarrow{2.3} c C\} \\
 PP_3 &= \{A \xrightarrow{3.1} a, B \xrightarrow{3.2} b, C \xrightarrow{3.3} c\}
 \end{aligned}$$

G_P can generate $a^2 b^2 c^2$ in the following manner.

$$S \xrightarrow{P_1} A B C \xRightarrow{PP_2} a A b B c C \xRightarrow{PP_3} a^2 b^2 c^2. \tag{14}$$

There are two main advantages of using G_P over G_S .

First of all, G_P is easy to understand and easy to remember. On the other hand, for a person who is not familiar with this field, it is difficult to prove that G_S generates precisely $\{a^n b^n c^n\}$ and nothing else. As a matter of fact, Hopcroft and Ullman[6, p. 12] devoted a whole page to proving that

$$L(G_S) = \{a^n b^n c^n\}. \tag{15}$$

Second, for the same word, the length of the derivation chain in G_P is considerably shorter than the length of the derivation chain in G_S . It is true because the length of the derivation chain for converting

$$(BC)^n \xrightarrow{3} B^n C^n \tag{16}$$

in general, is a large number in G_S . However, this portion of derivation chain is eliminated in G_P .

In what follows, we shall define a class of parallel grammars called context-free parallel grammars.

Definition 1

If all the sequential productions in a parallel grammar are context-free, then this parallel grammar is called a *context-free parallel grammar*.

Theorem 1

There exists context-sensitive languages which can be generated by context-free parallel grammars.

Proof

We prove this theorem by constructing an example.

We know that $\{a^n b^n c^n\}$ is a context-sensitive language and G_p is a context-free parallel grammar.

Furthermore,

$$L(G_p) = \{a^n b^n c^n\} \quad (17)$$

Therefore, this Theorem is true.

The languages generated by context-free parallel grammars are called context-free parallel languages (CFPL).

Lemma 1

Context-free languages (CFL) are a proper subset of context-free parallel languages (CFPL).

$$CFL \subset CFPL \quad (18)$$

Lemma 2

CFPL is a more effective tool to model computer programming languages compared with CFL, especially for parallel computer programming languages, for example, the ADA Programming Language[24].

Conclusion

The main results of the theory of formal languages are reviewed and presented in an intuitive manner. More precisely, the fact that the four types of formal languages can be generated by four types of grammars or can be recognized by four types of automata is summarized in Figure 1. The relationships between computer programming languages and context-sensitive languages are also presented. Parallel productions, parallel grammars, and context-free parallel grammars are defined and investigated. It is shown that there exist context-sensitive languages which can be generated by context-free parallel grammars. The generation of context-sensitive languages using context-free parallel grammars appears to be a fertile field for further study. Much further work remains to be done. The results obtained in this paper may have useful applications in artificial intelligence[25], modelling parallel computers, programming languages, software engineering[26,27], expert systems[28,29], picture descriptions[30,31], picture generation and robotics[25,32].

References

1. Chomsky, N., "Three models for the description of language", *PGIT*, Vol. 2, 1956, pp. 113-24.
2. Fu, K.S., *Syntactic Pattern Recognition and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1982.
3. Lee, E.T. and Zabeth, L.A., "Note on fuzzy languages", *Information Sciences*, Vol. 1, 1969, pp. 421-34.
4. Pavlidis, T., *Structural Pattern Recognition*, Springer-Verlag, New York, Berlin, 1977.
5. Miller, W.F. and Shaw, A.C., "Linguistic methods in picture processing: a survey", *Proceedings Fall Joint Computer Conference*, 1968, pp. 279-90.
6. Hopcroft, J.E. and Ullman, J.D., *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.
7. Hopcroft, J.E. and Ullman, J.D., *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, MA, 1979.
8. Peyton-Jones, S.L., *Implementation of Functional Programming Languages*, Prentice-Hall, Englewood Cliffs, NJ, 1987.
9. Ledgard, H. and Marcotty, M., *The Programming Language Landscape*, Science Research Associates, Chicago, IL, 1981.
10. Pratt, T.W., *Programming Languages: Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1975.
11. Chomsky, N., "On certain formal properties of grammars", *Information and Control*, Vol. 2, 1959, pp. 137-67.
12. Chomsky, N., "Context-free grammars and pushdown storage", *Quart. Prog.*, Dept. No. 65, MIT Research Laboratory in Electronics, 1962, pp. 187-94.
13. Chomsky, N., "Formal properties of grammars", *Handbook of Mathematical Psychology*, Vol. 1, Wiley, New York, 1963, pp. 323-418.
14. Chomsky, N. and Miller, G.A., "Finite state languages", *Information and Control*, Vol. 1, 1958, pp. 91-112.
15. Chomsky, N. and Schutzenberger, M.P., "The algebraic theory of context-free languages", *Computer Programming and Formal Systems*, North-Holland, Amsterdam, 1963, pp. 118-61.
16. Gries, D., *Compiler Construction for Digital Computers*, John Wiley & Sons, Chichester, 1971.
17. Hwang, K. and Briggs, F.A., *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, NY, 1984.
18. Lee, E.T., "On two-dimensional programmed grammars", *Robotics*, Vol. 3, 1987, pp. 427-31.
19. Lee, E.T., Pan, Y.J. and Chu, P., "An algorithm for region filling using two-dimensional grammars", *International Journal of Intelligence Systems*, Vol. 2 No. 3, 1987, pp. 255-63.
20. Rosenkrantz, D.J., "Programmed grammar – a new device for generating formal languages", *8th IEEE Annual Symposium on Switching Automata Theory*, Austin, TX, 1967.
21. Rosenkrantz, D.J., "Programmed grammars – a new device for generating formal languages", doctoral dissertation, Columbia University, 1967.
22. Rumelhart, D.E. (Ed.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations*, MIT Press, Cambridge, MA, 1986.
23. McClelland, J.L. (Ed.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 2: Psychological and Biological Models*, MIT Press, Cambridge, MA, 1986.
24. Pyle, I.C., *The ADA Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

25. Staugaard, A.C., *Robotics and AI – An introduction to Applied Machine Intelligence*, Prentice-Hall, Englewood Cliffs, NJ, 1987.
26. Bell, D., Morrey, I. and Pugh, J., *Software Engineering: A Programming Approach*, Prentice-Hall International (UK) Ltd, Hemel Hempstead, 1987.
27. Sommerville, I., *Software Engineering*, Addison-Wesley, London, 1982.
28. Waterman, D.A., *A Guide to Expert Systems*, Addison-Wesley, Reading, MA, 1986.
29. Keller, R., *Expert System Technology – Development and Application*, Prentice-Hall, Englewood Cliffs, NJ, 1987.
30. Lee, E.T., Chu, P. and Peng Wu, C., "Applications of entity-relationship model to picture description", *Robotica*, Vol. 5, 1987, pp. 231-4.
31. Lee, E.T., "Relationship hierarchy for picture representation using entity-relationship approach", *Kybernetes*, Vol. 17 No. 3, 1988, pp. 45-51.
32. Fu, K.S., *Robotics: Control, Sensing, Vision, and Intelligence*, McGraw-Hill, New York, NY, 1987.

Further reading

- Fagin, R., Halpern, J.Y., Moses, Y. and Vardi, M.Y., *Reasoning about Knowledge*, MIT Press, Cambridge, MA., 1995.
- Finkel, R.A., *Advanced Programming Language Design*, Addison-Wesley, Menlo Park, CA, 1996.
- Kaplan, R.M., *Constructing Languages Processors for Little Languages*, John Wiley & Sons, New York, NY, 1994.
- Sebesta, R.W., *Concepts of Programming Languages*, 3rd ed., Addison-Wesley, Menlo Park, CA, 1996.
- Winskel, G., *The Formal Semantics of Programming Languages: An Introduction*, MIT Press, Cambridge, MA, 1993.
- Winston, P.H., *Artificial Intelligence*, 3rd ed., Addison-Wesley, Reading, MA, 1992.