MECHANICAL GENERATION OF CONTEXT-FREE LANGUAGES

SUMMARY

A simple algorithm for generating terminal strings from a context-free phrase structure grammar is described, incorporating provision for the weighting of various possible constructions. A description of a program using this algorithm is given, together with some suggestions for extending its usefulness.

INTRODUCTION

Much work has been done in recent years on the syntactic analysis of terminal strings in context-free, phrase structure languages. A resumé of this work may be found in the recent papers of Brasseur and Cohen (Brasseur and Cohen, 1965, 1 and 2), to which the reader is referred for a fuller bibliography. In this paper we shall consider the converse (and much easier) problem of generating terminal strings from a given grammar. A program for doing this has a number of possible applications; the generation of random, syntactically correct programs can be used for checking out sections of a compiler; the program can be used for evaluating a grammar of a natural language and even for producing plausible anagrams.

Loeckx and Wodon, in their report (Loeckx and Wodon, 1965), describe a sophisticated program, based on the algorithm given in Chomsky and Schützenberger, 1963, for generating all the terminal strings of a context-free language, together with their ambiguities (i.e. the number of distinct structural descriptions assigned by the grammar, to each string). Their program used list-processing techniques and the authors state that, although the program is a useful tool for investigating the properties of formal languages, it was written as an exercise in the techniques they employ and so was in no sense optimized in core space used or computing time taken. The process which we shall describe is much less sophisticated but is easily programmed and seems to be efficient in the use of core space and computing time. Loeckx and Wodon's program will only deal with grammars with just a single non-terminal symbol; the one described here is not subject to this restriction. It does not, however, necessarily generate all the terminal strings, but only a random selection of them, and it does not assign any ambiguity to them. It is, therefore, perhaps, less useful for the investigation of formal languages, but more useful for the other applications referred to above.

THE BASIC ALGORITHM

The basic algorithm which we have used for generating terminal strings is the simple recursive procedure implied by the form of a phrase structure grammar. The procedure takes as parameter a string Q of mixed terminal and non-terminal symbols, and is first entered with parameter S, the defined initial symbol. The procedure examines the left-most symbol, A, of Q;

if A is terminal then it is output, deleted from Q to produce a new string Q', and the procedure re-entered with parameter Q'. If A is non-terminal then it is replaced in the string Q by the right-hand side of any production having left-hand side A, and the procedure is re-entered with this new string as parameter; the procedure terminates when the value of the parameter on entry is the empty string. We may call this process 'left-to-right' generation; it is equally possible to generate from right-to-left, but in this case the output symbols must be placed on a stack and output from there when the procedure has terminated.

It is clear that the algorithm as described above amounts to no more than simulating the action of the push-down store automation defined by the grammar. However, at one critical point, the algorithm is imprecise; how do we choose which production to apply at any given opportunity ?

WEIGHTING

The production is chosen using a pseudo-random number generator, which produces pseudo-random numbers with a rectangular distribution over the range [0,1]. The simplest way of doing this is to choose each of n alternatives with probability 1/n, i.e. choose the rth alternative if the output, x, of the random number generator lies in the range $(r-1)/n \leqslant x < r/n$.

For many purposes, however, it is convenient to be able to weight constructions; this is done in the following way. The syntax (which is generally presented in Backus Normal Form) consists of rules either of the form

$$\langle A \rangle ::= C_1 \mid C_2 \mid \dots \mid C_n,$$

where A is a non-terminal symbol and the C_i are strings, or of the form

$$\langle A \rangle ::= \lambda_1, C_1, \mid \lambda_2, C_2 \mid \ldots \mid \lambda_n, C_n$$

where λ_i are integers. In the former case, all constructions are taken with equal probability; in the latter case the ith construction is taken with probability

$$\lambda_i / \sum_{j=1}^{n} \lambda_j$$

The utility of this device can be seen from the following example. Suppose we have the construction

⟨expression⟩ ::= ⟨identifier⟩|⟨identifier⟩⟨operator⟩ ⟨expression⟩;

if the two constructions are taken with equal probability then half the expressions produced will consist of a single identifier. If, on the other hand, the constructions are weighted in the following manner

⟨expression⟩ ::= 1,⟨identifier⟩|5,⟨identifier⟩ ⟨operator⟩ ⟨expression⟩

then a more realistic output will be obtained in which only one sixth of the expressions are single identifiers.

THE PROGRAM

The program based on this algorithm was written, in machine code, for the Titan computer in the University Mathematical Laboratory, Cambridge; the Titan is a fast computer with 128K of 48-bit core, with half-words addressable by hardware. Normal input is on 7-track paper tape produced by Flexowriters.

The program falls into two main parts and a small output routine. The first part of the program reads the syntax and stores it in a suitable form. The syntax is presented in Backus Normal Form, together with a number of extra conventions :

(1) All productions having the same left-hand side must be amalgamated; if this is not done, only the last one is used.
(2) Weighting is recognized as described above.
(3) If a non-terminal symbol has occurred in the right-hand sides of the productions, but has not been a left-hand side by the time the whole system has been read, it is assumed to stand for itself e.g. if the following production has occurred

    ⟨comparison operator⟩ ::= ⟨⟨ ⟩ | ⟨ ⟩⟩

    then the productions

    ⟨⟨ ⟩ ::= ⟨ and ⟨ ⟩⟩ ::= ⟩

    are added to the syntax if no other productions with these left-hand sides has been found. This provides a means of using the metalinguistic symbols as elements of the alphabet.

The whole syntax is terminated by a solidus which is followed by the initial symbol from which terminal strings are to be produced together with an integer specifying how many terminal strings are required. Thus the complete input to the program for a simple syntax might appear as follows :

```
        <var>   ::=  A | B | C | D
```

```
⟨add op⟩  ::= + | -
⟨term ⟩ ::= |, ⟨ var⟩ | 2, ⟨ var⟩  ⟨add op⟩  ⟨term⟩
⟨exp ⟩ ::=3, (⟨term⟩) (⟨term⟩) | 1, (⟨term⟩) / (⟨var⟩) | 1,⟨term⟩
⟨statement ⟩ ::= ⟨ var⟩  ::=⟨exp⟩
 /
⟨statement⟩
 5
```

REFERENCES

1. M. Brasseur and J. Cohen : 'Algorithmes d'analyse syntaxique pour langages 'context-free', (Parts I and II), Chiffres, 1965, Nos. 2 and 3.

2. J. Loeckx and P.-L. Wodon : 'Mechanical generation of a context-free language by list processing techniques'. Report R27 of the Research Laboratory of the Manufacture Belgique de Lampes et de Matériel Electronique, Brussells, 1965.

M. F. BOTT
University of Cambridge