SIAM J. COMPUT.
Vol. 12, No. 4, November 1983

0097-5397/83/1204-0004 $01.25/0

UNIFORM RANDOM GENERATION OF STRINGS IN A CONTEXT-FREE LANGUAGE*

TIMOTHY HICKEY† AND JACQUES COHEN†

Abstract. Let S be the set of all strings of length n generated by a given context-free grammar. A uniform random generator is one which produces strings from S with equal probability. In generating these strings, care must be taken in choosing the disjuncts that form the right-hand side of a grammar rule so that the produced string will have the specified length. Uniform random generators have applications in studying the complexity of parsers, in estimating the average efficiency of theorem provers for the propositional calculus, in establishing a measure of ambiguity of a grammar, etc. Two methods are presented for generating uniform random strings in an unambiguous context-free language. The first method will generate a random string of length n in linear time, but must use a precomputed table of size $O(n^{r+1})$, where r is the number of nonterminals in the grammar used to specify the language. The second method precomputes part of the table and calculates the other entries as they are called for. It requires only linear space, but uses $O(n^2(\log n)^2)$ time to generate each string. Both methods generate strings by leftmost derivations where the probability that a given production will be used depends on the history of the derivation. It is also shown that, in the special cases of finite-state or linear languages, the generation can be performed in linear time with constant space.

Key words. uniform random generation, balanced parenthesis strings, convolution, enumeration problems, leftmost derivations, generating functions, parsing

1. Introduction.

Given a context-free grammar it is straightforward to generate "random" terminal strings: starting with the main nonterminal the sentential forms are constructed by considering equiprobable each of the disjuncts forming the right-hand side of a rule. The resulting strings will be of varying lengths.

To generate strings of a fixed length, one may have to bypass the random choice of disjuncts and choose a particular disjunct which is likely to lead more directly to a terminal string. Consider the set of all strings of length n in the language. A uniform random generator is one which will produce strings from this set *with equal probability*. The generation of such strings is a subtle process because the probabilities with which disjuncts are used at each step in a derivation depend on the previous steps of the derivation.

A generator of uniform random strings has several applications in computer science:

(1) The complexity of parsers is usually expressed as a function of the length of the string being parsed; thus the generator is useful in estimating their average case efficiency. For example, Cohen and Roth [5] describe a method for determining the execution times of top-down and bottom-up parsers. The execution times are expressed in terms of the number of occurrences of each terminal in the given string. The analyses of these parsers could be either experimental (by generating and parsing a large number of strings) or analytic (by determining a formula which takes into account the above probabilities). The generation of uniform random strings is also useful in estimating the average speedup in parallel parsing [4].

(2) Formulas in the propositional calculus can be generated by context-free grammars. If the length of formulas is of interest, one can use a uniform random generator to estimate the number of tautologies of a given length, the percentage which use certain operators, etc. Uniform random generators are also useful in estimating the relative efficiencies of theorem provers from the propositional calculus.

* Received by the editors November 20, 1981, and in revised form September 27, 1982. This research was supported by the National Science Foundation under grant number MCS 79-05522.

† Computer Science Program, Brandeis University, Waltham, Massachusetts 02254.

(3) When the generating techniques described below are applied to ambiguous grammars, the probability that an ambiguous string will be generated is directly proportional to the number of parses of the string. By generating strings of length n one can estimate the degree of ambiguity in the language as a function of n.

2. Objectives. In a recent paper Arnold and Sleep [2] propose a method for generating uniform random strings of a predetermined length in a nested parentheses language. Their method involves computing the probabilities with which a left or right parentheses should be generated. Based on the number of unmatched left parentheses already generated (u), and the number of symbols remaining to be generated (s), a left parentheses is generated with probability:

$$p_L(u, s) = \frac{(s-u)(u+2)}{2s(u+1)}. \tag{1}$$

Using their method, every string of the chosen length has an equal probability of being generated.[1]

Our objective is to extend the Arnold and Sleep technique to unambiguous context-free languages. We generate tables of probabilities that are consulted when performing a leftmost derivation of a uniform random string. These tables specify the probability with which a production should be applied. A crucial point is that our probabilities depend on the current state of the derivation and not just the productions.

We describe and analyze two generation methods in the sequel. The first computes all of the probabilities that could possibly be needed to produce a string of the given length. It can then generate uniform random strings in linear time. The total time complexity is $O(n^{r+1}+nm)$ and the space needed is $O(n^{r+1})$ where n is the desired string length, m the number of random strings generated and r the number of nonterminals in the grammar. The second method computes the probabilities as they are needed. This reduces the needed space to $O(n)$ but requires more time to generate each string. The total time complexity is $O(n^2 \log n + mn^2(\log n)^2)$.

3. Notation and preliminary remarks. Let $G = (V, \Sigma, P, N_1)$ be an unambiguous cycle-free[2] context-free grammar containing no ε-productions where $V = N \cup \Sigma$, $N = \{N_1, \cdots, N_r\}$ is the nonterminal vocabulary, Σ is the set of terminals, N_1 is the start symbol, and $P = \{\pi_{ij} : N_i \to \alpha_{ij} | i = 1, \cdots, r, j = 1, \cdots, s_i\}$ is the set of productions (see [6] for detailed definitions).

We choose to generate strings by leftmost derivation, but any canonical method of derivation would be adequate. A leftmost derivation in G of a terminal string ω consists of a sequence $\beta_1, \beta_2, \cdots, \beta_t$ of sentential forms where $\beta_1 = N_1$, $\beta_t = \omega$, and β_{k+1} is obtained by rewriting the leftmost nonterminal in β_k using one of the productions in P. If N_i is the leftmost nonterminal in β_k, there are s_i different choices for β_{k+1} corresponding to the productions π_{ij}, $j = 1, \cdots, s_i$. The methods described in the next two sections generate random strings of length n by determining the probabilities $p_{ij}(\beta, n)$ with which production π_{ij} should be applied to β.

The probabilities are chosen so that every string of length n has an equal probability of being generated. Let $g_\beta(n)$ denote the number of terminal strings of length n that can be derived from a sentential form β and let γ be the result of

[1] The methods described by Wetherell [8] involve assigning a fixed probability to each production in the grammar. This scheme cannot, in general, be used to produce strings of a predetermined length because it negects the history of the derivation.

[2] G is cycle-free if no nonterminal can be derived from itself.

applying π_{ij} to β in a leftmost derivation. Since $g_\gamma(n)$ is the number of terminal strings of length n which can be derived from β by first applying π_{ij}, the probability that must be assigned to π_{ij} is:

$$p_{ij}(\beta, n) = \frac{g_\gamma(n)}{g_\beta(n)}. \tag{2}$$

In the remainder of this section generating functions are introduced and used to express $g_\beta(n)$ as a convolution (defined below) of the simpler functions $g_{N_i}(n)$. A terminal string of length n derived from the concatenation of two strings, β_1, β_2, is the concatenation of two terminal strings whose lengths sum to n. The number of such length n terminal strings can therefore be expressed as a convolution:

$$g_{\beta_1\beta_2}(n) = \sum_{n_1+n_2=n} g_{\beta_1}(n_1)g_{\beta_2}(n_2) = (g_{\beta_1} * g_{\beta_2})(n). \tag{3}$$

Note that if the grammar is ambiguous this convolution gives the number of parses of terminal strings of length n.

Several properties of the convolution operator that will be used in this paper are discussed in this paragraph. Let $g(n)$, $g_1(n)$ and $g_2(n)$ be functions defined on the integers which are zero for negative values of n. The convolution of g_1 and g_2 is defined by

$$(g_1 * g_2)(n) = \sum_{n_1+n_2=n} g_1(n_1)g_2(n_2) = \sum_{k=0}^{n} g_1(k)g_2(n-k). \tag{4a}$$

Although convolution can be studied directly, its properties are easier to understand in the context of generating functions (see [7, p. 86]). The generating function of g is the formal power series $Z[g](x) = \sum_{n=0}^{\infty} g(n)x^n$. A simple computation shows that the generating function of the convolution of two functions is the product of their respective generating functions

$$Z[g_1 * g_2] = Z[g_1]\, Z[g_2]. \tag{4b}$$

An immediate consequence of this interpretation is the associativity and commutativity of convolution:

$$g_1 * g_2 = g_2 * g_1, \tag{4c}$$

$$(g_1 * g_2) * g_3 = g_1 * (g_2 * g_3), \tag{4d}$$

which follow from the corresponding properties of power series multiplication. Let δ_k be the function whose generating function is x^k:

$$\delta_k(n) = \begin{cases} 1, & n = k, \\ 0, & n \neq k. \end{cases} \tag{4e}$$

It satisfies $Z[\delta_k * g] = Z[\delta_k] * Z[g] = x^k Z[g]$ by (4b), and this implies the next two properties:

$$(\delta_k * g)(n) = g(n-k), \qquad \delta_0 * g = g, \tag{4f}$$

$$(\delta_k * \delta_j) = \delta_{k+j}. \tag{4g}$$

The notation $g^{(k)}$ is used to denote the convolution of a function g with itself k times, with the convention that $g^{(0)} = \delta_0$. With this notation the relation $g^{(k)} * g^{(j)} = g^{(k+j)}$ holds for any nonnegative integers k and j.

These properties can now be used to show that the value of $g_\beta(n)$, for any string β, depends only on the number of terminal symbols that remain to be generated by the derivation process and on the number of occurrences of each nonterminal in the string. Let $T(\beta)$ be the number of terminals in β (so $n-T(\beta)$ terminals remain to be generated) and let $A(\beta)=(A_1(\beta),\cdots,A_r(\beta))$ be the vector whose ith component is the number of occurrences of N_i in β. Define the function $f(t, a)$ for $a=(a_1,\cdots,a_r)$ by

$$f(t,a)=(g_{N_1}^{(a_1)}*\cdots*g_{N_r}^{(a_r)})(t). \tag{5}$$

When the vector a is fixed the notation $f_a(t)$ will sometimes be used instead of $f(t, a)$. If a and a' are vectors and $a+a'$ is their componentwise sum, then the relation $f_{a+a'}=f_a*f_{a'}$ holds by virtue of the commutativity and associativity of convolution.

In this paragraph, the following fundamental relation will be proved by induction on the length of β

$$g_\beta=\delta_{T(\beta)}*f_{A(\beta)},\qquad g_\beta(n)=f(n-T(\beta),A(\beta)). \tag{6}$$

Notice that the second equation is equivalent to the first by the convolution property of the delta function (4f). If β is a single symbol, the relation certainly holds. Suppose β is the concatenation of two strings β_1, β_2, neither of which is the empty string, then $g_\beta=g_{\beta_1}*g_{\beta_2}$ by (3), and the inductive step follows easily from properties (4a)–(4g):

$$\begin{aligned}g_\beta&=g_{\beta_1}*g_{\beta_2}=\delta_{T(\beta_1)}*f_{A(\beta_1)}*\delta_{T(\beta_2)}*f_{A(\beta_2)}=\delta_{T(\beta_1)}*\delta_{T(\beta_2)}*f_{A(\beta_1)}*f_{A(\beta_2)}\\&=\delta_{T(\beta_1)+T(\beta_2)}*f_{A(\beta_1)+A(\beta_2)}=\delta_{T(\beta)}*f_{A(\beta)}.\end{aligned} \tag{7}$$

This implies that $g_\beta(n)$ depends only on $n-T(\beta)$ and $A(\beta)$, as we claimed.

In our new notation the formula (2) for the probability that production π_{ij} will be used becomes

$$p_{ij}(\beta,n)=\frac{f(n-T(\gamma),A(\gamma))}{f(n-T(\beta),A(\beta))}=\frac{f(n-T(\beta)-T(\alpha_{ij}),A(\beta)-A(N_i)+A(\alpha_{ij}))}{f(n-T(\beta),A(\beta))}. \tag{8}$$

Both methods use this formula to generate random strings. They differ in the way the values of f are calculated, and in the amount of precomputation performed.

4. Method 1.

4. Method 1. In the first method we compute $f(t, a)$ for every t and a that could possibly arise in a derivation of a string of length n and we store these values in an $r+1$ dimensional array, r being the number of nonterminals in the grammar. Since the grammar contains no ε-productions, the values of $f(t, a)$ that need to be considered are those with $0\leqq t\leqq n$ and $a_1+\cdots+a_r\leqq t$.

The entries in the array can be computed efficiently using the following recurrence

$$f(t,a)=\sum_{j=1}^{s_i}f(t-T(\alpha_{ij}),a+A(\alpha_{ij})-A(N_i)),\quad \textit{if } a_i\neq 0, \tag{9}$$

which we now explain. The initial conditions are that $f(t, a)=0$ for all a and all $t\leqq 0$, with the single exception $f(0,(0,\cdots,0))=1$. Let β be a nonterminal string with $A(\beta)=a$ whose leftmost symbol is N_i; this is possible if $a_i\neq 0$. The left-hand side of (9) is the number of terminal strings of length t that can be derived from β. The jth term on the right-hand side of (9) is the number of length t terminal strings that can be derived from β after the leftmost symbol is rewritten using production π_{ij}. Since one of the productions $\pi_{ij}(j=1,\cdots,s_i)$ must be used to rewrite N_i, (9) holds.

If each α_{ij} contains a terminal (as is the case with grammars in Greibach normal form) the recurrence above relates $f(t, a)$ to values of f with a smaller value of t. The array can then be filled for increasing values of t in a straightforward manner.

However, if some production does not contain any terminals, care must be taken to ensure that new values of f are expressed in terms of previously computed values. let $|a| = a_1 + a_2 + \cdots + a_r$ for a vector a. Observe that for all α_{ij} satisfying $T(\alpha_{ij}) = 0$ and for all a with $a_i > 0$ we have $|a| \leqq |a - A(N_i) + A(\alpha_{ij})|$ with equality holding if and only if production π_{ij} has the form $N_i \to N_k$ for some k. Since the grammar is not cyclic the nonterminals can be ordered in such a way that this occurs only when $i > k$. The following loop structure will now compute the values of f needed to derive any string of length n.

```
$f \leftarrow 0$; $f(0, (0, \cdots, 0)) \leftarrow 1$;
**for** $t \leftarrow 1$ **to** $n$ **do**
**for** $s \leftarrow t$ **downto** 0 **do**
{*$s$ represents the component summation* $|a| = a_1 + \cdots + a_r$}
  **for** $a_1 \leftarrow s$ **downto** 0 **do**
  **for** $a_2 \leftarrow s - a_1$ **downto** 0 **do**
  . . .
  **for** $a_{r-1} \leftarrow s - (a_1 + \cdots + a_{r-2})$ **downto** 0 **do**
    **begin**
      $a_r \leftarrow s - (a_1 + \cdots + a_{r-1})$;
      {*use* (9) *for any $i$ with $a_i \neq 0$ compute* $f(t, (a_1, \cdots, a_r))$}
    **end**
```

This method requires time and space $O(n^{r+1})$ to fill the table. Since the grammar contains no ε-productions and is not cyclic, the generation of a random terminal string of length n will involve $O(n)$ rewritings. Therefore, $O(n)$ probability computations which can each be formed in constant time are required for every terminal string generated. The total time complexity is therefore $O(n^{r+1} + mn)$, where m is the number of terminal strings generated, and the space complexity is $O(n^{r+1})$.

5. Method 2. In Method 2 the values of $f(t, a)$ needed in the probability formula (8) are computed while the string is being generated using the definition of $f(t, a)$ by convolution (5)

$$f(t, a) = (g_{N_1}^{(a_1)} * \cdots * g_{N_r}^{(a_r)})(t).$$

The functions $g_{N_i}(t)(1 \leqq t \leqq n)$ are precomputed in time $O(n^2 \log n)$ and space $O(n)$ using the technique described in the next paragraph. The computation of $f(t, a)$ is performed by first calculating the convolution powers, $g_{N_i}^{(a_i)}(t)(1 \leqq t \leqq n)$, using the precomputed functions $g_{N_i}(t)(1 \leqq t \leqq n)$, and then performing r convolutions. These powers can each be computed with at most $2 \cdot \log_2 n$ convolutions since a_i is no larger than n. The convolution of two functions defined for $1 \leqq t \leqq n$ can be performed in time $O(n \log n)$ using the fast Fourier transform (see [1, p. 263, Cor. 1]) or in time $O(n^2)$ directly. Since $O(n)$ probability computations must be performed for every terminal string generated, the total time complexity of method 2 is $O(n^2 \log n + mn^2(\log n)^2)$. The space requirement is $O(n)$ since the only precomputed values are $g_{N_i}(t)(1 \leqq t \leqq n, 1 \leqq i \leqq r)$,

The recurrence (9) for $f(t, a)$ and the convolution definition of f can be used to perform the precomputation of the functions g_{N_i} in time $O(n^2 \log n)$ and linear space.

Since $g_{N_i}(t)=f(t, A(N_i))$, (9) yields

$$g_{N_i}(t)=\sum_{j=1}^{s_i} f(t-T(\alpha_{ij}), A(\alpha_{ij})).$$

Let e_{kj} represent $a_k(\alpha_{ij})$. The convolution definition of f transforms this equation into one involving only the functions g_{N_j}:

$$(10)\qquad g_{N_i}(t)=\sum_{j=1}^{s_i} (g_{N_1}^{(e_{1j})} * \cdots * g_{N_r}^{(e_{rj})})(t-T(\alpha_{ij})).$$

Since $g_{N_i}(0)=0$ for all i and $T(\alpha_{ij})\geqq 0$ for all i and j, this equation expresses $g(t)$ in terms of the values of $g_{N_i}(t')(1\leqq t'<t, 1\leqq i\leqq r)$. Equation (10) involves a constant number of convolutions and must be applied nr times to calculate the requisite values of the g_{N_i}. The total precomputation performed in this way requires time $O(n^2 \log n)$ and space $O(n)$ as claimed.

6. Examples.

Consider the unambiguous context-free grammar $G_0=(\{N_1, N_2, (\,,\,)\}, \{(\,,\,)\}, P, N_1)$ with productions:

$$P: \quad \begin{aligned} &\pi_{11}: N_1\rightarrow (N_2\\ &\pi_{21}: N_2\rightarrow N_1N_2\\ &\pi_{22}: N_2\rightarrow), \end{aligned}$$

which generates strings of well-balanced parentheses enclosed in a pair of matching parentheses.

If the outermost pair of parentheses are stripped from the strings in the language $L(G_0)$ one obtains the language of Arnold and Sleep [2].

The recurrence relations used by method 1 to calculate the values of f are

$$f(t, (a_1, a_2))=\begin{cases} f(t-1, (a_1-1, a_2+1)), & \text{if } a_1\geqq 1,\\ f(t, (a_1+1, a_2))+f(t-1, (a_1, a_2-1)), & \text{if } a_2\geqq 1.\end{cases}$$

If both a_1 and a_2 are nonzero either formula may be used. If the leftmost nonterminal of a sentential form β is N_1 then production π_{11} must be used to rewrite β. If an N_2 is leftmost the probability formula (8) must be used to determine which of π_{21} and π_{22} should be used. Let $T(\beta)=b$ and $A(\beta)=(a_1, a_2)$, then the probability formulas are:

$$p_{21}(\beta, n)=\frac{f(n-b, (a_1+1, a_2))}{f(n-b, (a_1, a_2))}, \qquad \frac{p_{22}(\beta, n)=f(n-b-1, (a_1, a_2-1))}{f(n-b, (a_1, a_2))}.$$

Figure 1 shows the probabilities needed to generate any string of length 8. This figure is analogous to [2, Fig. 1]. Table 1 shows the values of $f(t, a)$ needed to compute those probabilities.

Method 2 computes the values of g_{N_1} and g_{N_2} for $1\leqq t\leqq n$ using the formulas

$$\begin{aligned} &g_{N_1}(t)=g_{N_2}(t-1),\\ &g_{N_2}(t)=(g_{N_1} * g_{N_2})(t)+(g_{N_1}^{(0)} * g_{N_2}^{(0)})(t-1)=(g_{N_2} * g_{N_2})(t-1)+\delta_1(t),\end{aligned}$$

and the initial conditions $g_{N_i}(t)=0$ for $t\leqq 0$.

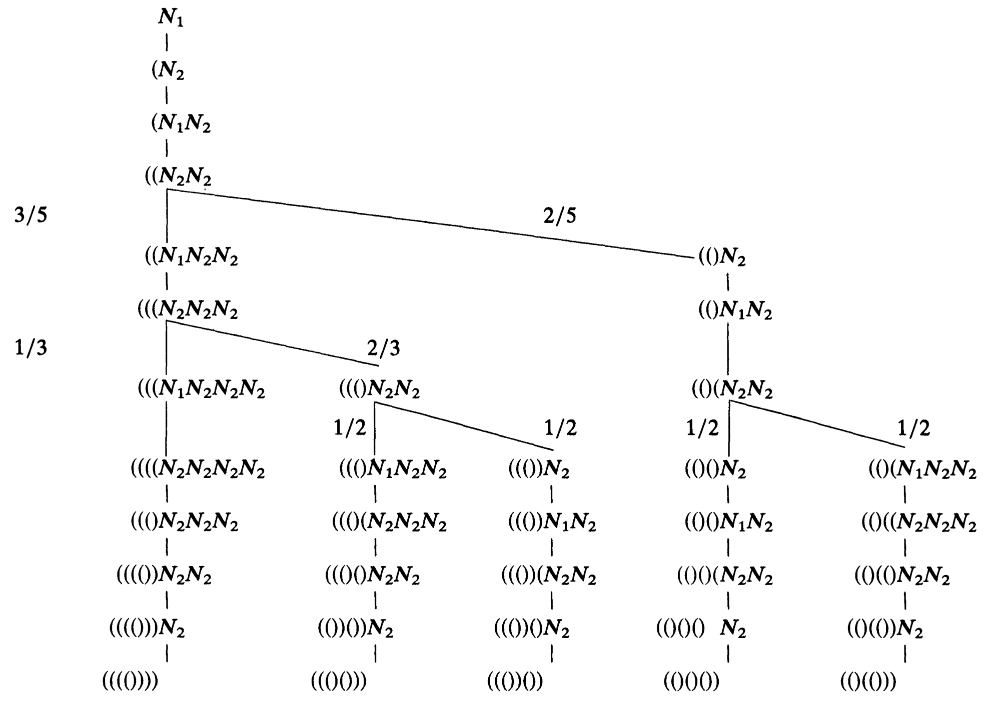

FIG. 1. *Generation of strings of length* 8. (*Note: The fractions indicate the computed probabilities.*)

TABLE 1
The function $f(t, (a_1, a_2)) = (g_{N_1}^{(a_1)} * g_{N_2}^{(a_1)})(t)$.

a_1	a_2	$f_{(a_1,a_2)}(t)$	t 0	1	2	3	4	5	6	7	8
0	0	$\delta_0(t)$	1	0	0	0	0	0	0	0	0
0	1	$g_{N_2}(t)$	0	1	0	1	0	2	0	5	0
0	2	$g_{N_2}^{(2)}(t)$	0	0	1	0	2	0	5	0	14
0	3	$g_{N_2}^{(3)}(t)$	0	0	0	1	0	3	0	9	0
0	4	$g_{N_2}^{(4)}(t)$	0	0	0	0	1	0	4	0	14
1	0	$g_{N_1}(t)$	0	0	1	0	1	0	2	0	5
1	1	$(g_{N_1} * g_{N_2})(t)$	0	0	0	1	0	2	0	5	0
1	2	$(g_{N_1} * g_{N_2}^{(2)})(t)$	0	0	0	0	1	0	3	0	9
1	3	$(g_{N_1} * g_{N_2}^{(3)})(t)$	0	0	0	0	0	1	0	4	0

7. Finite state grammars. In a leftmost derivation of a terminal string of length n using a regular grammar G, each sentential form β except for the last contains exactly one nonterminal. If β contains N_i then $g_\beta(n) = f(n - T(\beta), A(N_i)) = g_{N_i}(n - T(\beta))$ by the fundamental relation (6). Since the probabilities can be determined directly from the functions $g_{N_1}(t), \cdots, g_{N_r}(t)$, Methods 1 and 2 coincide. The recurrence relation (9) for G is a finite difference equation and so $g_{N_1}(t), \cdots, g_{N_r}(t)$, can be computed for $1 \leqq t \leqq n$ in linear time. Therefore the time complexity is $O(n + mn)$ and the space required is $O(n)$, but this is not always optimal! Cohen and Katcoff [3] describe and analyze an algorithm for finding closed form expressions for the functions $g_{N_1}, \cdots, g_{N_r}$ of a regular grammar G. The algorithm involves finding the (possibly complex) roots $Z_1, \cdots, Z_q$ of a polynomial with integer coefficients. The functions

g_{N_i} then have the form

$$g_{N_i}(t)=\sum_{j=1}^{g} U_{ij}Z_j^t+U_{i0}, \tag{11}$$

for some complex numbers U_{ij}. Let $R(n)$ be the time required to find the values of the Z_i and U_{ij} to an accuracy sufficient to determine $g_{N_i}(t)$ precisely for all $t \leqq n$. If the roots are integers then $R(n)$ is constant; otherwise it depends on the polynomial and the method used to find the roots.

The tth power of a complex number can be computed in time $O(\log n)$, if $t \leqq n$. This provides a method for generating random strings in time $O(R(n)+mn \log n)$ and constant space, where m is the number of terminal strings. This method will be superior to the previous one if complexity is measured by the product of the time-taken by the space-required.

Observe that the values of $g_{N_i}(t)$ can be computed from $Z_1^t, \cdots, Z_q^t$ in constant time and Z_i^t can be computed from Z_i^{t+1} and Z_i in constant time. This suggests that we precompute $Z_1, \cdots, Z_q$, and $Z_1^n, \cdots, Z_q^n$ in time $O(R(N)+\log n)$. The values of $g_{N_1}(t), \cdots, g_{N_r}(t)$ can then be computed from the stored values of $Z_1^t, \cdots, Z_2^t$ which are then replaced by $Z_1^{t-1}, \cdots, Z_q^{t-1}$ in constant time. This method will generate random terminal strings in the regular language in linear time using constant space. The total time requirement is $O(R(n)+\log n+mn)$. If $R(n)$ is constant this is the best possible performance for an infinite language.

Consider for example the regular grammar, G_1 given in [3] which generates all strings with an even numbers of 0's and 1's and any number of 2's and 3's

$$\begin{aligned}
&G_1=(\{N_1,N_2,0,1,2,3\},\{0,1,2,3\},P,N_1),\\
&P: N_1\to 0N_3|1N_2|2N_1|3N_1|2|3\\
&\quad N_2\to 0N_4|1N_1|2N_2|3N_2|1\\
&\quad N_3\to 0N_1|1N_4|2N_3|3N_3|0\\
&\quad N_4\to 0N_2|1N_3|2N_4|3N_4.
\end{aligned} \tag{12}$$

The formulas (13) for $g_{Ni}(1\leqq i\leqq 4)$ involve powers of the integer 2, and so $R(n)=0$.

$$\begin{aligned}
&g_{N_1}(t)=2^{2(t-1)}+2^{(t-1)}, &&g_{N_2}(t)=2^{2(t-1)},\\
&g_{N_3}(t)=2^{2(t-1)}, &&g_{N_4}(t)=2^{2(t-1)}-2^{(t-1)}.
\end{aligned} \tag{13}$$

The probabilities $p_{ij}(\beta, n)$ for this example are all very close to 0.25 when $n-T(\beta)$ is large. Let $\beta=\zeta N_i$ for some terminal string ζ of length $T(\beta)$ and let $t=n-T(\beta)$ be the number of terminal symbols still to be generated; then the probability with which production π_{ij} should be used is:

$$p_{ij}(\zeta N_i, n)=0.25+\frac{E(i,j)2^{t-3}}{g_{N_i}(t)} \quad \text{for } t>1,$$

where $E(i,j)$ is determined by Table 2.

When $t=1$, π_{15} and π_{16} have probability 0.5 while π_{25} and π_{35} have probability 1.0.

The results in this section extend naturally to linear grammars (where each production contains at most one nonterminal on its right-hand side).

TABLE 2

	$E(i,j)$	j = 1	2	3	4
i	1	−1	−1	1	1
	2	−2	2	0	0
	3	2	−2	0	0
	4	1	1	−1	−1

8. Final remarks. We have restricted our attention to unambiguous grammars. If our methods are applied to an ambiguous grammar, the probability that a given string of the predetermined length will be generated is proportional to the number of leftmost derivations of the string. In fact, our methods will generate uniform random derivations of terminal strings of length n. Since the strings with the most parses are most likely to be generated, these methods could be used empirically to search for ambiguity in a grammar.

Let A_n be the expected number of parses of a string of length n. If T_n is the number of strings of length n and D_n is the number of leftmost derivations of strings of length n, then $A_n = D_n/T_n$. The probability that the methods discussed in this paper will generate an ambiguous string of length n is the proportion of all derivations of length n strings that yield ambiguous strings. A lower bound for the proportion is $(D_n - T_n)/D_n = 1 - 1/A_n$, because at most T_n derivations can yield unambiguous strings. The probability that M samples will fail to yield an ambiguous string is less than $(1/A_n)^M$. This observation can be used to obtain an upper bound on the average ambiguity of strings of length n for a grammar with a given level of confidence. If $A_n \geqq c$, the probability that M samples by our method fail to yield an ambiguous string is at most $(1/c)^M$. Thus, the estimate $A_n < c$ can be made with a confidence level of $1 - 1/R$ provided at least $\log_c(R)$ samples are generated and reveal no ambiguity.

Arnold and Sleep's algorithm requires linear time and constant space even though the language is not regular. Their formula, (1), for the probabilities can be derived from the present work as follows. In a leftmost derivation using the grammar G_0 of § 6 the sentential forms β will have the form $\tau N_1 N_2^u$ or τN_2^u where τ is a terminal string of length $T(\beta)$. Rather than storing β we can output τ, store u, and store the Boolean value "N_1 *is in the stack*". In this way, the current state of the derivation can be stored using two locations.

Let the probabilities $p_{2j}(\tau N_2^u, n)$ for $j = 1, 2$ be denoted by the abbreviated notation $p_{2j}(u, s)$, where $s = n - T(\beta)$. Observe that u is the number of unmatched open parentheses in τ and s is the number of terminals that remain to be generated. Since the strings generated by Arnold and Sleep can be obtained from the strings in the language $L(G_0)$ by removing the outermost pair of parentheses, our probabilities are related to theirs by:

$$p_{21}(u, s) = p_L(u-1, s-1), \qquad p_{22}(u, s) = p_R(u-1, s-1).$$

In the Appendix an analytic formula for $f(t, a)$ is derived using generating functions and the Lagrange inversion formua [6]. When $a = (0, k)$ the formula specializes to

$$f(t, (0, k)) = \frac{k}{t} \frac{t!}{((t-k)/2)!\,((t+k)/2)!}.$$

By relation (9) $f(t,(1,k))=f(t-1,(0,k+1))$ and so by relation (8) we obtain a simple formula for the probability of generating a left parenthesis:

$$(14)\qquad p_{21}(u,s)=\frac{f(s-1,(0,u+1))}{f(s,(0,n))}=\frac{(u+1)(s-u)}{2u(s-1)}.$$

This agrees with Arnold and Sleep's formula (1).

An interesting open problem is to determine the class of languages for which random strings can be generated in linear time and space. We have shown that all regular languages are in this class. Moreover, any grammar for which the probability functions can be computed in constant time (as in (14)) will generate a language in this class.

Appendix. This appendix illustrates the use of the Lagrange inversion formula (see [6, p. 40]) to obtain analytic formulas for the function $f(t,a)$ in the case of the parentheses grammar, G_0, of § 6. The method we describe is applicable to any unambiguous grammar in Greibach normal form which has only one nonterminal symbol (i.e., generalized prefix polish notation).

Let $G_i(x)=\sum_{t=0}^{\infty} g_{N_i}(t)x^t$ be the generating functions of g_1 and g_2. In § 6 the following equations, (15), were needed to perform the precomputation of method 2

$$(15)\qquad g_{N_1}=\delta_1 * g_{N_2},\qquad g_{N_2}=\delta_1 * g_{N_2} * g_{N_2}+\delta_1.$$

These equations translate into the following set of power series equations for generating functions (see 4b):

$$G_1=xG_2,\qquad G_2=xG_2^2+x=x(G_2^2+1).$$

The Lagrange inversion formula states that if $H_1(w)$ and $H_2(w)$ are analytic functions of w (e.g., polynomial) and if $G(x)$ is a power series in x that satisfies $G(x)=xH_1(G(x))$, then $H_2(G(x))$ has the following form:

$$(18)\qquad H_2(G(x))=\sum_{m=0}^{\infty}\frac{x^m}{m!}\left(\frac{d^{m-1}}{dw^{m-1}}[H_2'(w)\cdot H_1(w)^m]\bigg|_{w=0}\right).$$

In our case $H_1(w)=w^2+1$.

To derive a general formula for $f(t,(a_1,a_2))$ we can use the recurrence equations derived in § 6 to simplify the problem. Recall that $f(t,(a_1,a_2))=f(t-1,(a_1-1,a_2+1))$ whenever $a_1>0$. Iteration of this relation shows that $f(t,(a_1,a_2))=f(t-a_1,(0,a_2+a_1))$. Thus it will suffice to find an analytic formula for $f(t,(0,k))=g_{N_2}^{(k)}(t)$. Since the power series of $g_{N_2}^{(k)}$ is G_2^k, (16) with $H_2(w)=w^k$ will give the desired formula.

The polynomial $H_2'H_1^m$ when expanded by the binomial theorem has the form

$$kw^{k-1}\sum_{j=0}^{m}\binom{m}{j}w^{2j}.$$

The $(m-1)$st derivative of this evaluated at zero and divided by $m!$ will yield the formula

$$g_{N_2}^{(k)}(m)=\frac{1}{m!}k(m-1)!\binom{m}{(m-k)/2}=\frac{k}{m}\frac{m!}{((m-k)/2)!((m+k)/2)!}.$$

From this we obtain the desired result:

$$f(t,(a_1,a_2))=\frac{(a_1+a_2)}{(t-a_1)}\frac{(t-a_1)!}{((t-2a_1-a_2)/2)!((t+a_2)/2)}.$$

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1978.

[2] D. B. ARNOLD AND M. R. SLEEP, *Uniform random generation of balanced parentheses strings*, ACM Trans. Programming Languages and Systems, 2 (1980), pp. 122–128.

[3] J. COHEN AND J. KATCOFF, *Automatic solution of a certain class of combinatorial problems*, Inform. Proc. Letters, 6 (1977), pp. 101–104.

[4] J. COHEN, T. HICKEY AND J. KATCOFF, *Upper bounds for speedup in parallel parsing*, J. Assoc. Comput. Mach., 29 (1982), pp. 408–428.

[5] J. COHEN AND M. ROTH, *Analyses of deterministic parsing algorithms*, Comm. ACM, 21 (1978), pp. 448–458.

[6] M. A. HARRISON, *Introduction to Formal Language Theory*, Addison-Wesley, Reading, MA, 1978.

[7] D. E. KNUTH, *The Art of Computer Programming, Vol.* 1: *Fundamental Algorithms*, 2nd ed., Addison-Wesley, Reading, MA, 1973.

[8] C. S. WETHERELL, *Probabilistic languages: A review and some open questions*, ACM Computing Surveys, 12 (1980), pp. 361–379.