A Note on Generating Well-formed Parenthesis Strings Lexicographically

M. C. Er
Department of Computing Science, The University of Wollongong, P.O. Box 1144, Wollongong, NSW 2500, Australia

An efficient recursive algorithm for generating well-formed parenthesis strings lexicographically is shown. This algorithm can be easily adapted to generate stack-sortable permutations without changing the main control structures of the algorithm. The connection between well-formed parenthesis strings and ordered trees is also illustrated.

INTRODUCTION

The combinatorial problem of generating all well-formed parenthesis strings arises in many different contexts.[1–3] This paper presents an efficient recursive algorithm for generating all well-formed parenthesis strings in lexicographic order. The connections between well-formed parenthesis strings and both stack-sortable permutations and ordered trees are shown.

DEFINITION

Let $P = p_1p_2 \dots p_{2n}$ be a string of parentheses such that p_i is either a '(' or a ')'. We say that P is *feasible* if, when scanning the string from p_1 to p_{2n}, the number of right parentheses does not exceed the number of left parentheses at any state. A string of parentheses P is said to be *well-formed* when P is feasible and the numbers of left and right parentheses are equal. Clearly, if P is well-formed, then p_1 = '(' and P_{2n} = ')'.

Let $P(n)$ be a set of well-formed parenthesis strings of length $2n$. To generate $P(n)$ lexicographically, first of all, the lexicographic order ought to be defined. Here, we assume that '(' > ')'.

Definition 1: Lexicographic order

Let $P = p_1p_2 \dots p_{2n}$ and $P' = p'_1p'_2 \dots p'_{2n}$, where $P, P' \in P(n)$. We say that $P < P'$ if $p_i = p'_i$ for $i = 1, 2, \dots j-1$ and $p_j < p'_j$, where $1 \leq j \leq 2n$.

GENERATING ALGORITHM

All well-formed parenthesis strings can be described by the context-free grammar: $S \rightarrow (S)S \mid \epsilon$. However this production rule does not lend itself to an implementation of a generating algorithm. This is not surprising, because all well-formed parenthesis strings except the last one generated by the first non-terminal symbol S have disappeared when the second non-terminal symbol S starts producing well-formed parenthesis strings. In consequence, a generating algorithm based on the above context-free grammar cannot be implemented using the conventional stack-based recursion. A more sophisticated control structure is needed.

Here, we present a generating algorithm for the problem of enumerating all well-formed parenthesis strings. This algorithm could be implemented using conventional recursion without violating strict stack discipline. The following crucial observations lead to the construction of an efficient recursive generating algorithm: (1) the number of right parentheses should not exceed the number of left parentheses when generating a parenthesis string from left to right; (2) the total numbers of left and right parentheses are equal in each well-formed parenthesis string. These observations of course follow the definition of well-formed parenthesis string very closely. The generating algorithm is now given.

```
procedure  GenParentheses (L, R: integer);
  { Generate L left parentheses and L + R right
  parentheses lexicographically. To generate P(n), the
  set of well-formed parenthesis strings, lexicographi-
  cally, activate this procedure as GenParentheses (n, 0).
  The global variable n indicates the number of pairs of
  parentheses. }
  var i: integer;
  begin
    if L = 0 and R = 0 then PrintParentheses ()
      else begin
        i := 2 * (n − L) − R + 1;
        if R ≠ 0 then
          begin p[i] := ')';
                GenParentheses (L, R − 1)
          end;
        if L ≠ 0 then
          begin p[i] := '(';
                GenParentheses (L − 1, R + 1)
          end
        end
  end { GenParentheses };
```

The numbers of left and right parentheses are controlled by two parameters L and R. When a ')' is generated, R is decremented by 1. Whereas when a '(' is generated, L is decremented by 1 and R is incremented by 1. Therefore, when the procedure is activated as *GenParentheses* $(n, 0)$, $P(N)$ will be generated. Owing to the sequential nature of the algorithm, the order in which parentheses are generated is crucial. As an attempt is made to generate ')' before '(', the $P(n)$ so generated is always in lexicographic order. Note that the procedure call *PrintParentheses* () simply prints a string of well-formed parentheses stored in the global array p.

CCC 0010–4620/83/0026–0205$01.50

Let $C(n, 0)$ be the number of procedure calls to *GenParentheses* for generating $P(n)$. It is given by the following recurrence relation:

$$C(x, y) = C(x, y - 1) + C(x - 1, y + 1) + 1$$

with boundary condition

$$C(x, y) = 0 \text{ for } y < 0 \text{ or } x < 0$$

STACK-SORTABLE PERMUTATIONS

A set of well-formed parentheses has a connection with a set of stack-sortable permutations defined in Ref. 4. Let 1, 2, . . ., n be a stream of consecutive input numbers arranged in ascending order. Whenever a '(' is generated, the next input number is pushed onto a stack. Conversely, whenever a ')' is generated, the top element is popped off from the stack and sent to the output stream. These push and pop operations will generate a set of stack-sortable permutations in lexicographic order. This way of generating stack-sortable permutations is of course better than the ballot sequence approach taken by Rotem and Varol[4] as no inversion needs to be constructed. Furthermore, the resulting stack-sortable permutations so generated are in lexicographic order, whereas Rotem and Varol's are not. Consequently, the recursive algorithm for generating balanced parenthesis strings can be easily adapted to generate stack-sortable permutations. The detailed algorithm is given below.

```
procedure StackPermut (L, R, e: integer);
  { Generate a set of stack-sortable permutations of n
  numbers (1, 2, . . . , n) in lexicographic order. Activate
  this procedure initially as StackPermut (n, 0, 1). }
  var i: integer;
  begin
    if L = 0 and R = 0 then newline ()
      else begin
      if R ≠ 0 then
        begin i := pop ();
              print (n − L − R + 1, i);
              StackPermut (L, R − 1, e);
              push (i)
        end;
      if L ≠ 0 then
        begin push (e);
              StackPermut (L − 1, R + 1, e + 1);
              pop ()
        end
      end
  end { StackPermut };
```

The procedure *print* (c, i), once activated, prints the integer i in column c. As *StackPermut* preserves exactly the control structures of *GenParentheses*, therefore the number of procedure calls to *StackPermut* for generating the set of stack-sortable permutations of n numbers is given by $C(n, 0)$.

ORDERED TREES

The connection between the ordered trees and the ballot sequences is observed by Rotem and Varol,[4] whereas the relation between the ordered forests and well-formed parenthesis strings is noted by Even.[2] Here we show the connection between the ordered trees and the well-formed parenthesis strings.

For a binary tree of n nodes, there are $(n + 1)$ null leaves. There seems no obvious connection between a binary tree of n nodes and a well-formed parenthesis string of n pairs. However, if a binary tree is travelled in preorder, the last node visited must be a null leaf irrespective of the shape of the tree. Consequently, if this null leaf is removed from consideration, there are n nodes and n null leaves left. Suppose we interpret a well-formed parenthesis string when scanning from left to right as follows: '(' and ')' indicate visiting a node and a null leaf, respectively, of a binary tree in preorder. Then the following theorem is immediate because one cannot visit a null leaf without visiting its parent node first.

Theorem 1

Let $T(n)$ and $P(n)$ be the set of ordered trees of n nodes and the set of well-formed parenthesis strings of n pairs respectively. The mapping between $T(n)$ and $P(n)$ is one–one.

As a consequence of Theorem 1, a lexicographic ordering is imposed on $T(n)$ because of the lexicographic ordering of $P(n)$. Not surprisingly, this imposed lexicographic ordering on $T(n)$ is exactly the *local ordering* on $T(n)$ as defined below.

Definition 2: Local ordering

Let T and T' be two binary trees of n nodes. Then $T < T'$ if

(i) T is a null leaf and T' is not; or
(ii) both T and T' are not null leaves, and $\text{left}(T) < \text{left}(T')$; or
(iii) both T and T' are not null leaves, and $\text{left}(T) = \text{left}(T')$, and $\text{right}(T) < \text{right}(T')$.

We summarize the above discussion in the following theorem. It is not hard to prove.

Theorem 2

The lexicographic ordering of $P(n)$ is preserved in the local ordering of $T(n)$. Namely, the mapping between $P(n)$ and $T(n)$ is isotone.

We have shown above the theoretical connection between the ordered trees and the well-formed parenthesis strings. Now we show that this theoretical connection can also be implemented in an algorithm. The details are presented in the following coding.

```
function BuildTree (var i: integer): ↑tree;
  { Construct a binary tree of n nodes from the
  corresponding string of well-formed parentheses of n
  pairs stored in the array p[1 . . 2 * n]. Activate this
  procedure initially as BuildTree (x), where x is
  initialized to zero. }

  var node: ↑tree;
  begin i := i + 1;
    if i > 2 * n or p[i] = ')' then return (NIL);
    New (node);
    node↑ · left := BuildTree (i);
    node↑ · right := BuildTree (i);
    return (node)
  end {BuildTree};
```

Note that *tree* is of the following type:

```
type tree = record
        left: ↑tree;
        right: ↑tree
    end
```

As *BuildTree* constructs a binary tree of n nodes from the given well-formed parenthesis string of n pairs, the time complexity of this algorithm is thus $O(n)$.

CONCLUSION

We have shown that the lexicographic orderings of the well-formed parenthesis strings, of the stack-sortable permutations and of the ordered trees are the same class of combinatorial problem. Indeed, the efficient recursive algorithm for generating lexicographically a set of well-formed parenthesis strings can be easily adapted to generate a set of stack-sortable permutations without changing the structures of the algorithm at all. Furthermore, the well-formed parenthesis strings can be used for guiding the construction of ordered trees. The isotone property guarantees that if a set of well-formed parenthesis strings is enumerated lexicographically, the corresponding set of ordered trees is also generated in lexicographic order.

Acknowledgement

This research was supported by RGC under grant 05–143–105.

REFERENCES

1. D. B. Arnold and M. R. Sleep, Uniform random generation of balanced parenthesis strings. *ACM Trans. Programming Language and Systems* **2**(1), 122–128 (1980).
2. S. Even, *Graph Algorithms*, Computer Science Press (1979).
3. D. Tamari, The algebra of bracketings and their enumeration. *Nieuw Archief voor wiskunde* **3**, 131–146 (1962).
4. D. Rotem and Y. L. Varol, Generation of binary trees from ballot sequences. *Journal of the ACM* **25**(3), 396–404 (1978).

Received August 1982