Psychological Review
1982, Vol. 89, No. 6, 693-707

0033-295X/82/8906-0693$00.75

Theory of Serial Pattern Production: Tree Traversals

René Collard and Dirk-Jan Povel
University of Nijmegen, The Netherlands

Structural information about a serial pattern may be encoded in the form of a hierarchical memory code. Conversely, when the pattern is to be produced, the memory code should be decoded. In this article a theory on serial pattern production is proposed that builds on Restle's theory of structural trees and, especially, on the process models for sequence production developed by Greeno and Simon. According to the tree traversal interpreter, presented in this article, a structural tree corresponds to the interpretive process that operates on a hierarchical memory code. A comparative analysis of computational properties and psychological relevance is substantiated by empirical test and extension to recent work on the internal representation of music.

Serial pattern production refers to the process by which the elements of a sequence are produced from a hierarchical memory representation in an orderly serial fashion. The idea that a hierarchical code may be used to represent a serial pattern has been proposed by Simon and Sumner (1968) and Leeuwenberg (1969), among others. The concept of structural trees was introduced by Restle (1970) to depict the inner structure of serial patterns as represented by hierarchical codes. Essentially, Restle showed that high-level transitions in a structural tree cause more difficulty in anticipating serial patterns than do low-level transitions. He foresaw that an integration of memory code and structural tree could provide the basis of a theory of how serial patterns might actually be produced. Greeno and Simon (1974) showed that, in fact, different production models are conceivable. In the present article a model is presented that reflects the exact relation between memory code and structural tree. According to this model, called the tree traversal interpreter, a structural tree corresponds not so much to a hierarchical memory code as to the interpretive process that operates on that code.

This research was supported in part by the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Requests for reprints should be sent to René Collard, Department of Experimental Psychology, University of Nijmegen, Montessorilaan 3, 6500 HE Nijmegen, The Netherlands.

The psychological issue under consideration may be best introduced by means of an example. Consider a person who wants to whistle a tune learned long ago. Assume that the tune is stored in memory not as a long series of notes but in a concise hierarchical code reflecting the melodic structure, for example, of the form proposed by Deutsch and Feroe (1981). For the whistler to actually produce the tune, the memory presentation must be decoded in order to generate the proper sequence of notes. Theoretically, the decoding can be done in several different ways, as shown by Greeno and Simon (1974), who proposed the following models: the doubling, the recompute, and the push-down interpreters. These three interpreters and the tree traversal interpreter may be characterized as follows: First, the whistler might be able to decode in advance the memory representation to a complete series of notes that is temporarily stored in a short-term memory (the doubling interpreter). Second, it may be that the whistler recomputes each consecutive note to be produced by applying the memory code to the very first note of the tune (the recompute interpreter). Third, the whistler may use a push-down stack to store and recover notes preceding the last one in order to generate the next note (the push-down interpreter). Finally, the tree traversal interpreter assumes that the whistler uses the hierarchical memory code in an on-line fashion to compute each note from the immediately preceding one. It seems reasonable to

expect that the amount of processing required at each location in the sequence relates to performance, be it latencies or errors made. In that case the different production models lead to distinguishably different predictions, which may be tested empirically.

This article is mainly concerned with the mathematical description of the tree traversal interpreter and the comparative analysis of computational properties and psychological relevance of processes for sequence production. To that end we will first recapitulate the basic framework underlying serial pattern research (Simon, 1972). For the general relation between such research and theories of perception, cognitive organization, and language that share the idea of hierarchical representation, the reader is referred to Greeno and Simon (1974), Restle (1979), and Deutsch and Feroe (1981).

Serial Pattern Research

Serial pattern research has mainly focused on the development and application of coding models for the internal representation of patterned sequences, such as letter and number series (Geissler, Klix, & Scheidereiter, 1978; Jones & Zamostny, 1975; Leeuwenberg, 1969; Simon & Kotovsky, 1963; Vitz & Todd, 1969), patterns of lights (Restle 1970, 1976; Restle & Brown, 1970), temporal patterns (Povel, 1981), and musical patterns (Collard, Vos, & Leeuwenberg, 1981; Deutsch, 1980; Deutsch & Feroe, 1981; Leeuwenberg, 1971; Restle, 1970; Simon & Sumner, 1968). In each application the coding model provides a precise notation to describe the internal representation of serial patterns, usually in the form of a hierarchical code. As such, a code is a static structural description, independent of the processes that may be involved.

The underlying processes of encoding and decoding have been studied less extensively. Encoding corresponds to the process of the acquisition of pattern representations. So far, computer programs that simulate this process have been confined to strictly periodic patterns, for example, from the well-known Thurstone Letter Series Completion Test (Klahr & Wallace, 1970; Kotovsky & Simon, 1973; Simon & Kotovsky, 1963). A major theoretical problem in encoding is the inherent structural ambiguity of serial patterns: Generally, a pattern may be represented in so many different ways that it is neither practical nor feasible to generate all possible representations. But even if that could be done, the problem remains that only the most economical codes should be generated because only these appear to be perceptually relevant (Leeuwenberg, 1971; Simon, 1972). Recently, a solution to this thorny problem has been sought in a set-theoretical order of codes that is derived from the structural information of pattern codes, defined as the set of patterns having the same structural relations (Collard & Buffart, in press). The main subject of this article will be the opposite process, that of decoding, which corresponds to recall or production from memory.

In principle, each of the three stages—encoding, memory representation, and decoding—can be described in a separate model, with the constraint that the encoding model generates codes that belong to the coding model and the decoding model operates on codes from the coding model. Therefore, the coding model plays a central role. As Simon (1972) showed, all coding models proposed so far bear upon the same elementary transformations within a given alphabet, and the central core of the different notations can most conveniently be captured in what is usually called the TRM-model after the following basic operations: T for transpose, R for repeat, and M for mirror (Restle, 1970, 1976; Simon, 1972, 1978).

Notational Preliminaries

Following Greeno and Simon (1974), we will first restrict the analysis to strictly hierarchical codes with operations T_k of the form $T_k(x) = (x\ t_k(x))$ where t_k is a transposition on an ordered set of integers, defined by $t_k(x) = x + k$. Thus, $t_0(x) = x$, $t_1(x) = x + 1$, and so on. Within the TRM-model, R is a shorthand for T_0. So, sequence (2 2) is represented by $R(2)$ because $R(2) = (2\ t_0(2)) = (2\ 2)$. Similarly, (1 2) is represented by $T_1(1)$, because $T_1(1) = (1\ t_1(1)) = (1\ 2)$. The transpositions act on sequences of numbers in the same way, for example, $t_1(1\ 2\ 2\ 3) = (t_1(1)\ t_1(2)\ t_1(2)\ t_1(3)) = (2\ 3\ 3\ 4)$. There-

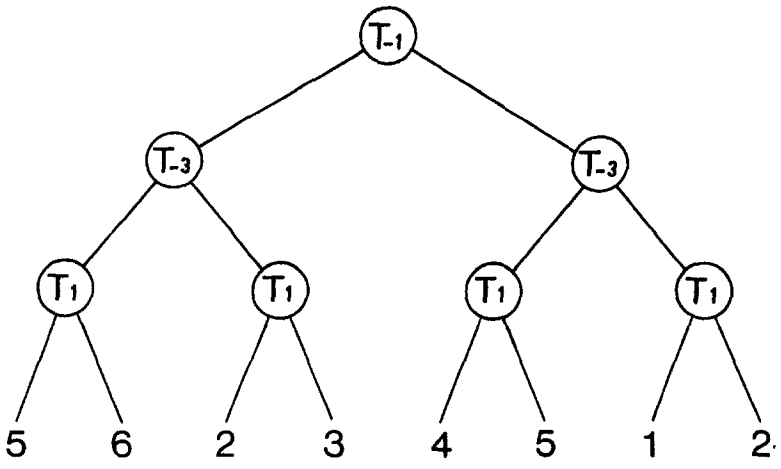

Figure 1. Structural tree for code $T_{-1}(T_{-3}(T_1(5)))$ of sequence (5 6 2 3 4 5 1 2).

fore, the operations may also be used hierarchically so that (2 2 3 3) can be represented by $T_1(R(2))$, because $T_1(R(2)) = T_1(2\ t_0(2)) = T_1(2\ 2) = (2\ 2\ t_1(2\ 2)) = (2\ 2\ t_1(2)\ t_1(2)) = (2\ 2\ 3\ 3\ )$, and, similarly, (3 2 3 2) by $R(T_{-1}(3))$. The mirror operation, M, and further generalizations will be defined later in this article. For the moment we will focus on some important group-theoretical properties of the transpositions, or transformations as they are generally called.

The above transpositions constitute a well-known transformation group: the addition modulo K. The theory, however, applies to any set of operations derived from commutative transformation groups that act on sequences. In this context, a transformation on a set X is a one-to-one mapping u: $X \rightarrow X$ or, in other words, a permutation of X. The transformations act on sequences of elements of X as if applied to each element individually, that is, $u(x_1\ x_2\ \ldots\ x_n) = (u(x_1)\ u(x_2)\ \ldots\ u(_n))$. To each transformation u, an operation U corresponds, defined by $U(x) = (x\ u(x))$. In this way the transformations are denoted by lowercase letters, whereas the uppercase letters denote the corresponding operations. Let "o" denote composition of transformations; then we may summarize the basic properties of a commutative transformation group G as follows: for transformations t, u, and v from G,

Associativity: $(t \circ u) \circ v = t \circ (u \circ v)$;

Commutativity: $t \circ u = u \circ t$;

Inverse: For each t there is a $\hat{t}$ such that $t \circ \hat{t} = e$, where e is the identity transformation. Also recall that inverse distributes over composition: If $t = u \circ v$, then $\hat{t} = \hat{u} \circ \hat{v}$.

It is easy to see that these properties apply to the transpositions within the TRM-model. For instance, the transpositions commute because $t_k \circ t_l = t_{k+l} = t_{l+k} = t_l \circ t_k$.

Using the commutative property, we may depict a hierarchical code as a so-called structural tree (Restle, 1970). In this way each strictly hierarchical code corresponds to a strictly nested binary tree (Greeno & Simon, 1974). The special property of such a tree is that the operations within each level are identical (see Figure 1). This equivalence, however, is a static mathematical fact that does not show how codes relate effectively to structural trees. If one assumes that the internal representation of serial patterns is in the form of an abstract hierarchical formula but at the same time claims that the actual behavior in serial pattern processing is based on the corresponding structural tree (Restle, 1970, p. 487), one must conclude that memory code and structural tree should be related by means of appropriate interpretive processing of the code. Without such processing, a code remains just a formula and will not account for characteristics due to its structural tree. Simply writing out the formulas,

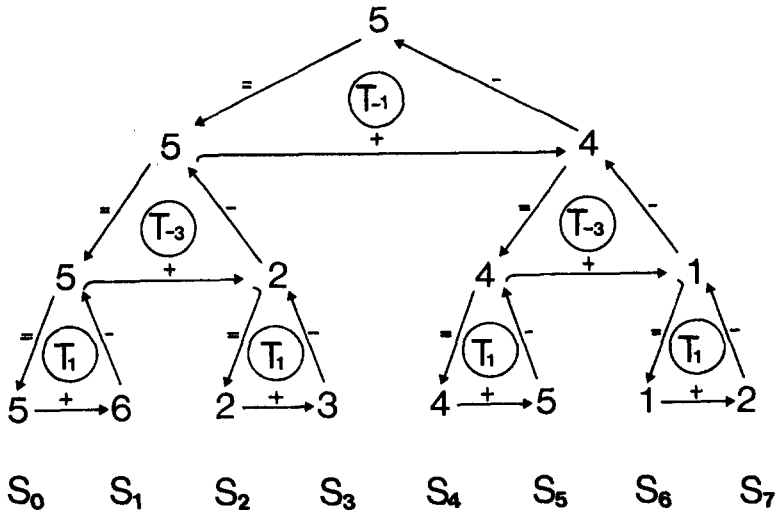

Figure 2. Computational diagram of the tree traversal interpreter, converting code $T_{-1}(T_{-3}(T_1(5)))$ to sequence (5 6 2 3 4 5 1 2). (Each element, S_i, is computed from its immediate predecessor, S_{i-1}, by traversing the tree.)

for example, $T_{-1}(T_{-3}(T_1(5))) \Rightarrow T_{-1}(T_{-3} \times (5\ 6)) \Rightarrow T_{-1}(5\ 6\ 2\ 3) \Rightarrow (5\ 6\ 2\ 3\ 4\ 5\ 1\ 2)$, will not do either, because then the generation of the second half of a sequence would be quite different from that of the first half, and therefore incompatible with the apparent symmetry of the structural tree.

The Tree Traversal Interpreter

We shall first explain the operation of the tree traversal interpreter informally. The basic idea is illustrated in Figure 2 for sequence $S = (5\ 6\ 2\ 3\ 4\ 5\ 1\ 2)$ from code $T_{-1}(T_{-3}(T_1(5)))$. The computational diagram in Figure 2 is obtained from the structural tree in Figure 1 by surrounding each node with a triangle of transformations, according to $T_k(x) = (x\ t_k(x))$. Within each triangle, the left-hand arrow denotes the identity, the bottom arrow corresponds to t_k, and the right-hand arrow corresponds to the inverse, $\hat{t}_k$.

The ith element of the sequence, S_i, can be computed from its immediate predecessor S_{i-1} by traversing the tree and applying the operations that are encountered in the following way: straightforward when moving to the right, inversely when moving upward. Thus, $T_{-1}(T_{-3}(T_1(5)))$ is converted subsequently to S as follows: 5, $t_1(5) = 6$, $t_{-3}(\hat{t}_1(6)) = 2$, $t_1(2) = 3$, $t_{-1}(\hat{t}_{-3}(\hat{t}_1(3))) = 4$, $t_1(4) = 5$, $t_{-3}(\hat{t}_1(5)) = 1$, $t_1(1) = 2$. Because composition is associative, we might as well compose the transformations first: 5, $t_1(5) = 6$, $(t_{-3} \circ \hat{t}_1)(6) = t_{-4}(6) = 2$, and so on. In the formal analysis below, we will adopt the intermediate notation $t_k \circ t_l(S_i)$ that leaves either possibility open. Technically speaking, the path through the tree should be called a preorder traversal (see Aho & Ullman, 1974, for instance). Because the leaves (sequence elements) must be produced from left to right anyway, we may call it simply a tree traversal in the present context.

The Tree Traversal Theorem

In order to specify the interpreter by a general equation $S_i = r_i(S_{i-1})$, we need some notation regarding position of leaves in binary trees. Let m be the depth of the tree; then the binary representation of leaf $i < 2^m$ will be denoted by $b_i = b_{i,m}b_{i,m-1}, \ldots, b_{i,1}$, where $b_{i,j}$ is either 0 or 1. In terms of binary trees, this means that the left branches are labeled with a 0 and the right branches with a 1. The binary representation of leaf i can be seen as the path from the top of the tree to leaf i. For instance, in a binary tree of depth 3, the leaves are denoted from left to right as follows: S_0 by 000, S_1 by 001, S_2 by 010, and so on. Now the tree traversal equation reads:

THEOREM. *For operations* U_m, U_{m-1}, ..., U_1, *based on a commutative transformation group* U, *let* $S = (S_0, S_1, \ldots, S_{2^m-1})$ *be the sequence coded by* $U_m(U_{m-1} \times (\cdots(U_1(S_0))\cdots))$; then

$$S_i = r_i(S_{i-1}), \quad \textit{where}$$

$$r_i = r_{i,m} \circ r_{i,m-1} \circ \cdots r_{i,1};$$

$$\begin{aligned} r_{i,j} &= u_j \quad \textit{if} \quad b_{i-1,j} = 0 \quad \textit{and} \quad b_{i,j} = 1, \\ &= \hat{u}_j \quad \textit{if} \quad b_{i-1,j} = 1 \quad \textit{and} \quad b_{i,j} = 0, \\ &= e \quad \textit{if} \quad b_{i-1,j} = b_{i,j}. \end{aligned}$$

PROOF. From Greeno and Simon's (1974) Equation 19, we have

$$S_i = p_i(S_0), \quad \text{where}$$

$$p_i = p_{i,m} \circ p_{i,m-1} \circ \cdots p_{i,1};$$

$$\begin{aligned} p_{i,j} &= u_j \quad \text{if} \quad b_{i,j} = 1, \\ &= e \quad \text{if} \quad b_{i,j} = 0. \end{aligned}$$

Replacing S_0 by $\hat{p}_{i-1}(S_{i-1})$, we can rewrite this equation by associativity and commutativity of composition in U and by distributivity of inverse:

$$S_i = (\hat{p}_{i-1,m} \circ p_{i,m}) \circ (\hat{p}_{i-1,m-1} \circ p_{i,m-1}) \cdots \circ (\hat{p}_{i-1,1} \circ p_{i,1})(S_{i-1}).$$

Denoting $(\hat{p}_{i-1,j} \circ p_{i,j})$ by $r_{i,j}$, we have

$$\text{if } b_{i-1,j} = 0 \text{ and } u_{i,j} = 1, \quad \text{then}$$

$$r_{i,j} = p_{i,j} = u_j;$$

$$\text{if } b_{i-1,j} = 1 \text{ and } u_{i,j} = 0, \quad \text{then}$$

$$r_{i,j} = \hat{p}_{i-1,j} = \hat{u}_j;$$

$$\text{if } b_{i-1,j} = b_{i,j} \quad \text{then either} \quad r_{i,j} = e \circ e = e$$

$$\text{or} \quad r_{i,j} = \hat{p}_{i-1,j} \circ p_{i,j} = \hat{u}_j \circ u_j = e.$$

The Interpreter

The tree traversal theorem may be implemented in several different ways, depending on the machinery available. In essence, an m-bit binary counter has to be mapped on the code: adding one to the binary representation of $i-1$, b_{i-1} corresponds to traversing the tree from position $i-1$ to i, whereas a bit-change $0 \Rightarrow 1$ of the jth bit dictates the application of the transformation at the jth level, u_j, and a change $1 \Rightarrow 0$ dictates the application of the inverse, $\hat{u}_j$. For the example of Figure 2, $T_{-1}(T_{-3}(T_1(5)))$, we have $U_3 = T_{-1}$, $U_2 = T_{-3}$, and $U_1 = T_1$. So $u_3 = t_{-1}$, $u_2 = t_{-3}$, and $u_1 = t_1$. Table 1 shows once more the derivation of sequence (5 6 2 3 4 5 1 2), using a binary counter.

Table 1
Code $T_{-1}(T_{-3}(T_1(5)))$ *Converted to Sequence (5 6 2 3 4 5 1 2)*

Binary counter			Transformation to be applied			Current element
T_{-1}	T_{-3}	T_1				
0	0	0				$S_0 = 5$
		↓			t_1	
0	0	1				$S_1 = 6$
	↓	↓		t_{-3}	$\hat{t}_1$	
0	1	0				$S_2 = 2$
		↓			t_1	
0	1	1				$S_3 = 3$
↓	↓	↓	t_{-1}	$\hat{t}_{-3}$	$\hat{t}_1$	
1	0	0				$S_4 = 4$
		↓			t_1	
1	0	1				$S_5 = 5$
	↓	↓		t_{-3}	$\hat{t}_1$	
1	1	0				$S_6 = 1$
		↓			t_1	
1	1	1				$S_7 = 2$
↓	↓	↓	$\hat{t}_{-1}$	$\hat{t}_{-3}$	$\hat{t}_1$	
0	0	0				

Note. A bit change $0 \Rightarrow 1$ of the jth bit dictates the jth transformation to be applied, a bit change $1 \Rightarrow 0$, the inverse.

Properties

The interpreter employs both commutativity and inverse of the transformations. Therefore, the transformations must constitute a commutative group. Using an argument similar to that of Greeno and Simon (1974, p. 195), we may prove that the tree traversal interpreter is the most efficient algorithm to compute S_i from S_{i-1}. (Leaving out a u_j or a $\hat{u}_j$ in the tree traversal theorem would change the value of some S_i. Therefore, no shortcut is possible in computing S_i from S_{i-1}.)

The number of transformations to be applied equals the number of bit changes in the successive increments of b_i. Let s denote the length of the sequence, that is, $s = 2^m$; then the total number of bit changes equals $2(s-1)$. Figure 3 shows the computational costs per element.

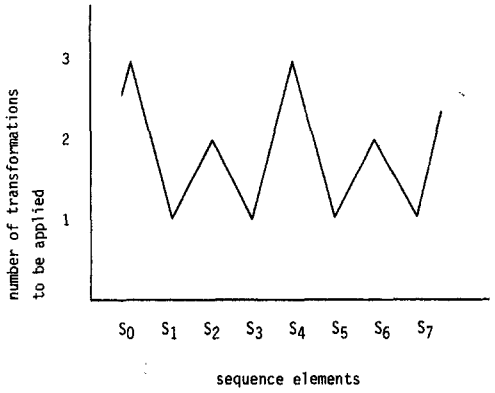

Figure 3. Performance profile of the tree traversal interpreter for code $T_{-1}(T_{-3}(T_1(5)))$.

From Table 1 it is easily seen that the "housekeeping operations" on the binary counter yield the same profile as the transformations do. So no matter what assumptions are made about the costs of elementary processing steps, a jag-shaped performance profile results that corresponds to tree traversing. There are no special memory requirements for storing sequence elements, except possibly one location for the current element.

On-Line Production

The tree traversal interpreter can also be seen as emitting a series of transformations rather than sequence elements. For the example $T_{-1}(T_{-3}(T_1(\ )))$, we obtain by tree traversing t_1, $t_{-3} \circ t_1$, t_1, $t_{-1} \circ t_{-3} \circ t_1$, t_1, $t_{-3} \circ t_1$, t_1. Thus, a hierarchical code that applies to the elements of the sequence may be decoded to a series of possibly composite transformations that correspond to the differences between subsequent elements. Such a way of on-line operating may prove valuable if the absolute value of sequence elements is hard to represent internally. On-line production cannot be performed by the interpreters reviewed below.

Comparison

Greeno and Simon (1974) proposed three different models for sequence production: the push-down, the recompute, and the doubling interpreters. The first one needs a push-down stack to store elements in the sequence prior to the last one produced. It uses the binary counter (see Table 1) to recover elements stored in the push-down stack in a

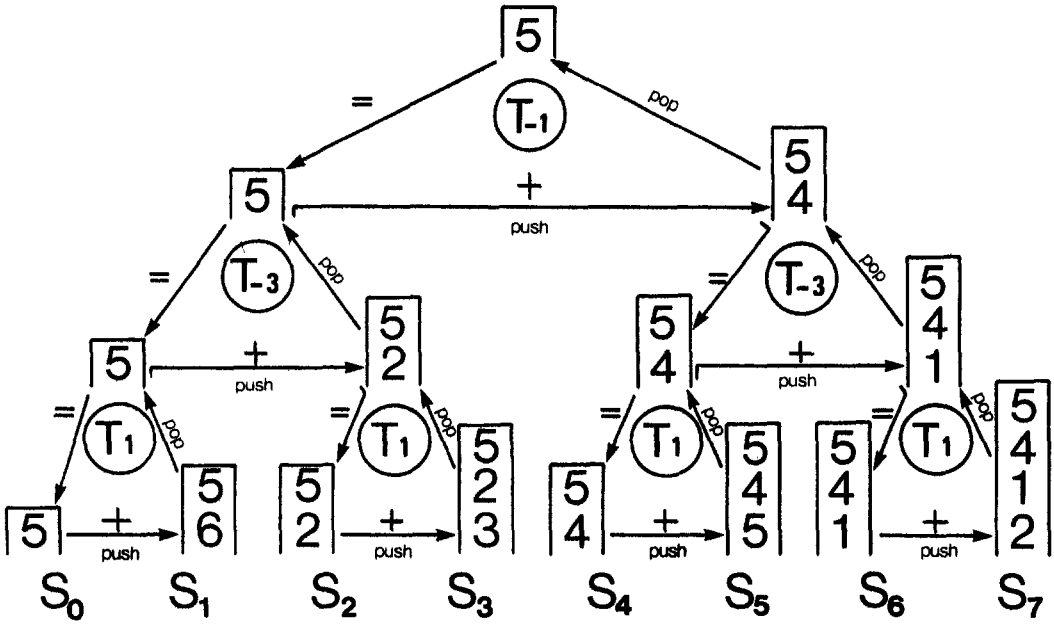

Figure 4. Computational diagram of the push-down interpreter, converting code $T_{-1}(T_{-3}(T_1(5)))$ to sequence (5 6 2 3 4 5 1 2). (Each element, S_i, is computed from some previously computed element, stored in the push-down stack.)

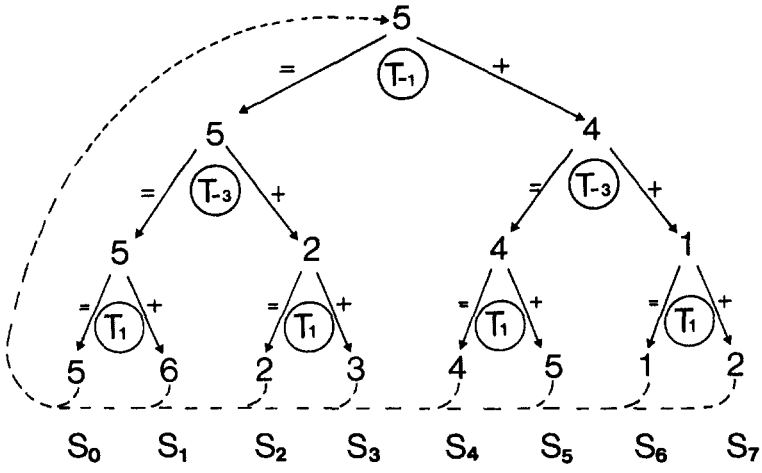

Figure 5. Computational diagram of the recompute interpreter, converting code $T_{-1}(T_{-3}(T_1(5)))$ to sequence (5 6 2 3 4 5 1 2). (Each element, S_i, is computed from the first element, S_0, by descending the tree.)

similar way as the tree traversal interpreter employs the binary counter to apply the inverse transformations. Thus, it can be seen as a semi-tree-traversal because upward movements in the tree are performed by popping the push-down stack. See Figure 4.

The recompute interpreter computes each element, S_i, from the first element, S_0. Instead of bit changes (see Table 1), the absolute values of the bits are used to apply the appropriate transformations. Therefore, it can be seen as a tree descender. See Figure 5.

The doubling interpreter evaluates nested formulas in the usual way, that is, inside to outside and therefore does not incorporate a structural tree, as has been shown already in the introduction. Moreover, doubling requires a short-term memory that must be capable of storing half the sequence. Hence, we will examine tree interpreters only.

Comparison of Tree Interpreters

In Table 2 some properties of the three tree interpreters are summarized. The tree traversal interpreter fits the special group-theoretical properties underlying the TRM-model exactly, whereas the push-down interpreter also applies to systems in which the transformations do not have inverses. The recompute interpreter computes the elements in a top-down fashion, or vice versa, and therefore, the transformations do not

Table 2
Comparison of Three Tree Interpreters for Strictly Nested Binary Trees

	Transformation			Housekeeping operation	
Interpreter	Must form	Total number to be applied	Memory requirements	Pushes and pops	Binary counter
Tree traversal	Commutative group	$2(s-1)$	1	—	$2(s-1)$
Push down	Commutative semigroup	$s-1$	$\frac{1}{2}\log s$	$2(s-1)$	$2(s-1)$
Recompute	Any	$\frac{1}{2}s\log s$	1	—	$\frac{1}{2}s\log s$

Note. s = sequence length.

Table 3
Comparison of Tree Interpreters for Repeating Sequence (2 3 2 3 5 6 5 6) From Code T_3 ($R(T_1(2))$)

Interpreter	Sequence
Tree traversal	2 - 3 - - 2 - 3 - - - 5 - 6 - - 5 - 6 - - - 2 - 3 - - 2 . . .
Push down	2 - 3 - 2 - 3 - 5 - 6 - 5 - 6 2 - 3 - 2. . .
Recompute	2 - 3 - 2 - - 3 - 5 - - 6 - - 5 - - - 6 2 - 3 - 2 . . .

Note. Each short line denotes a transformation to be applied.

need to commute, a point that we will return to later.

The product of computational and memory costs of the tree traversal interpreter is linearly proportional to sequence length s, denoted $O(s)$, whereas the other two tree interpreters require $O(s \log s)$. Unlike the recompute interpreter, the push-down interpreter may produce a similar performance profile for the popping of the push-down stack or for the housekeeping operations on the binary counter (Simon, 1978) as does the present tree traversal interpreter. The tree traversal profile for transformations, however, (see Table 3) does not apply to the push-down interpreter because it computes each element of the sequence, except S_0, by applying just one transformation to some previously stored element. In effect, the tree traversal interpreter is the one that comes closest to the basic idea underlying structural trees, namely, that the difficulty of any location in the sequence equals the level of transition immediately preceding it (Restle, 1970).

Generalizations

The main generalizations are from binary to n-ary trees and from strictly nested to not strictly nested trees. Unfortunately, Greeno and Simon (1974) did not provide a complete analysis for these generalizations. In fact, such an analysis should involve automata-theoretical issues that are beyond the scope of this paper too. Without elaborating all details, it is not difficult, however, to predict the behavior of the different interpreters for generalizations. The picture that emerges is that the recompute interpreter becomes less efficient when operating on n-ary trees, whereas the push-down interpreter demands a somewhat more sophisticated scheme for manipulating the push-down store when applied to not strictly hierarchical codes. The one that remains stable under both generalizations is the tree traversal interpreter. As indicated above, we will not elaborate at this time on all automata-theoretical details, but merely indicate the most characteristic differences.

Following Restle (1970), codes representable by n-ary structural trees are obtained by generalizing operations T_k: $T_k(x) = x\, t_k(x)$ to

$$T_k^n(x) = xT_k^{n-1}(t_k(x)), \quad \text{with} \quad T_k^0 = e.$$

In this way sequence (2 3 4 5) may be represented by $T_1^3(2)$. In generating the sequence from the code, we need an n-ary counter instead of a binary counter. The details are straightforward (see Greeno & Simon, 1974). What changes in Table 2 with respect to efficiency is the factor $\log s$. For the push-down interpreter, the maximum size of the push-down stack remains proportional to the depth of the tree, whereas for the recompute interpreter the number of housekeeping operations and transformations to be applied may increase to $(s - 1)(s - 2)/2$, that is, $O(s^2)$. "Recomputing" $T_1^n(1)$, for instance, yields 1, $t_1(1)$, $t_1(t_1(1))$, $t_1(t_1(t_1(1)))$, and so on. All other properties in Table 2 remain unchanged.

Codes that are not strictly hierarchical can be represented by concatenation. For instance, (1 1 2 3 2 2 3 4) may be represented by $T_1(R(1), T_1(2))$. Obviously, the number of memory locations needed for interpretive processing of not strictly hierarchical codes equals, at least, the number of independent initial elements. Again, the details of the counter are not very much more complicated (see Greeno & Simon, 1974). The push-down interpreter, however, needs a somewhat different scheme for manipulating the push-down store. The problem is that a push-down

store is a "first in, last out" device. So, if a sequence is made up hierarchically of two different patterns and some elements of the first pattern are kept in the push-down stack and the appropriate elements of the second pattern are pushed on top of these, then the first-pattern elements cannot be released without losing the elements of the second pattern. It turns out that one needs either a rather complex scheme for transmitting information from the push-down stack to the memory locations for initial elements, and vice versa, or several different push-down stacks.

Mirroring

We have postponed a precise account of mirroring, denoted by m, because its interpretation is somewhat more complicated than that of transposition. An example may help to clarify the problem of interpreting the mirror operation. Suppose we have an ordered set of six elements, say (1 2 3 4 5 6), then $m(1) = 6$, $m(2) = 5$, $m(3) = 4$, and so on, and the sequence (2 3 5 4) can be represented by $M(T_1(2))$, because $M(T_1(2)) = M(2\ t_1(2)) = M(2\ 3) = (2\ 3\ m(2\ 3)) = (2\ 3\ m(2)\ m(3)) = (2\ 3\ 5\ 4)$. If this code is depicted in a tree (see Figure 6), a problem arises because the right-hand T_1 applies inversely.

The tree of $T_1(M(2))$, however, fits correctly (2 5 3 4), but straightforward evaluation of $T_1(M(2))$ via $T_1(2\ 5)$ yields (2 5 3 6). Greeno and Simon (1974) concluded that M should be viewed as an operation that affects the way the other operations work: Transposition has to be applied in a negative direction if and only if it is preceded by an odd number of reflections, depending on the type of interpretive process. This idea, however, does not seem easy to implement. (Also, it does not apply to the doubling interpreter.) Therefore, we propose another way to handle reflections that does not demand any special considerations for interpretive processing.

The problem is that mirroring, as used above, does not commute with transposition, that is, $m \circ t_k \neq t_k \circ m$. In order to let the transformations commute, a double representation for sequence elements is needed. Suppose we have an ordered set E of six elements, (a b c d e f), then E will be represented by counting from the left, $E_L = (1_L\ 2_L\ 3_L\ 4_L\ 5_L\ 6_L)$, and by $E_R = (6_R\ 5_R\ 4_R\ 3_R\ 2_R\ 1_R)$ counting from the right. Thus Element b, for example, is both the second element from the left and the fifth element from the right. Within each representation of E, transpositions apply as usual, whereas mirroring has the function of changing from representation. For instance, $t_1(2_R) = 3_R$ and $t_1(2_L) = 3_L$, whereas $m(2_R) = 2_L$ and $m(3_L) = 3_R$. If E is stored in memory as a list with two-way access, or two lists with one-way access, the elements of the double representation can be viewed as retrieval operations. Thus, for the examples of Figure 6, no arithmetic complement needs to be taken. See Figure 7.

In this way codes, trees, and interpretive processing fit perfectly again. Note that double representation is more powerful: For example, sequence (2 1 6 5 3 2 5 4) may be represented hierarchically by $T_1(T_{-1}(2_L), T_1(1_R))$, a code that could not be expressed in the earlier formalism.

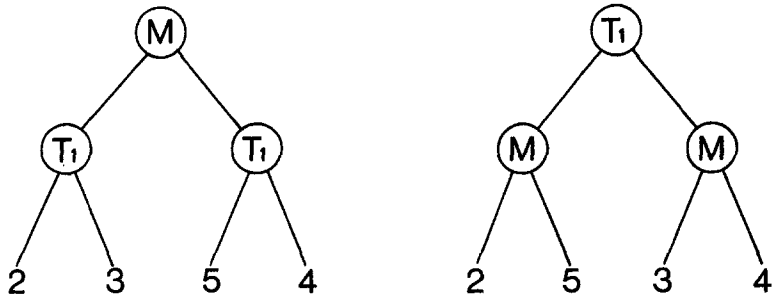

Figure 6. Two examples of mirroring: For code $M(T_1(2))$ the right-hand T_1 in the structural tree must apply inversely, whereas straightforward evaluation of code $T_1(M(2))$ yields (2 5 3 6).

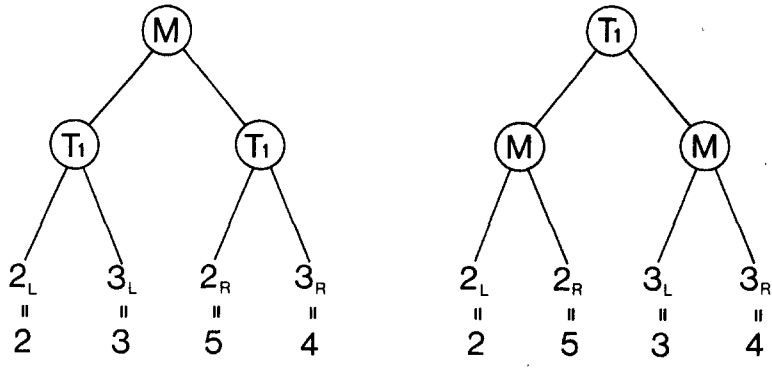

Figure 7. A double representation of elements solves the mirroring problem.

The above construction applies to any commutative transformation group T. Let V_h denote the set of subscripted elements v_h, h being L or R. Moreover let $\hat{h}$ denote L if h = R, and conversely. Then transformations t_k and transformations with reflection m_k are defined by $t_k(v_h) = (t_k(v))_h$ and $m_k(v_h) = (t_k(v))_{\hat{h}}$, respectively. Particularly, if the t_k's are transpositions modulo K, we have $t_k(v_h) = (v + k)_h$ and $m_k(v_h) = (v + k)_{\hat{h}}$, where + is addition modulo K. In fact, Greeno and Simon (1974) provided a similar group-theoretical definition. Basically, they employed an inverted copy, $\hat{E}$, of an ordered set of elements E in such a way that mirroring both changes from copy and takes inverse, whereas transposition operates straightforwardly in E, and inversely in $\hat{E}$. Some typical applications of mirroring will be given below.

Application

The main area of experimental research where the present theory may prove valuable is serial pattern production. In experiments, however, one usually has to account for all the processes involved. Kotovsky and Simon (1973) analyzed errors in the Thurstone Letter Series Completion Test and concluded that some errors are due to failures in the acquisition of pattern concepts whereas others are caused by faulty extrapolation of correct pattern descriptions. Also, in Restle's (1970) experiments on serial pattern learning, where college students tried to anticipate patterns of lights correctly, the jag-shaped error profile may not have been due to learning only. Because of this potential difficulty in assigning errors, latency might be a variable that lends itself better to compare different production models. Simon (1978) describes an experiment in which subjects were trained to apply each of the three interpretive models of Greeno and Simon (1974). For instance, a subject was asked to simulate the doubling interpreter and to decode M(R(T(1))) yielding the number sequence (1 2 1 2 6 5 6 5). Thereafter, the subject was asked to simulate the recompute interpreter and then the push-down interpreter. The latencies for performing the calculations as observed in the experiment qualitatively fit the theoretical performance curves very well, though considerable differences were found between subjects. Thus, the experiment shows that subjects can learn to mimic these particular methods to produce sequences. A somewhat different question is what method do subjects actually use in production tasks while whistling a tune, imitating a rhythm, or tapping a "finger pattern."

Tapping the Memory Code

Povel and Collard (in press) studied subjects who tapped finger patterns—sequences of taps made with different fingers. A pattern to be tapped was presented as a sequence of numbers, for example, (1 2 3 2 3 4), where the numbers 1 through 4 correspond with the index, middle, ring, and little fingers, respectively. After some test trials, subjects tapped the pattern as fast as possible in a repeated fashion. Apart from the possible effect of the memory code and the interpretive processing, the motor demands to produce the ac-

tual finger transitions might, of course, also determine the latencies. In order to disentangle these potential factors, sets of stimuli rather than single stimuli were used. Each set consisted of a number of cyclic permutations of one base sequence, for example, (3 2 1 2 3 4), (2 1 2 3 4 3), and (1 2 3 4 3 2). Such a set has two typical properties: First, when the sequences are tapped repeatedly, the motor demands will be equal for all. Second, if one sequence within the set can be coded hierarchically, so can the other sequences. In the above example the last three elements persist to mirror the first three elements. The sequences can be coded by $M(T_{-1}{}^2(3))$, $M(2\ 1\ 2)$, and $M(T_1{}^2(1))$, respectively. Therefore, the timing profile as predicted from the tree traversal interpreter for all three patterns has the form—*S*1–*S*2–*S*3—*S*4–*S*5–*S*6, where "—" indicates a long interval, and "–" a short interval in a way similar to that in Table 3.

Such a set of cyclic permutations provides an interesting opportunity to study the production process: If the latencies were determined by the motor demands to produce the finger transitions, one would expect to find identical profiles when the observed timing profiles are displayed in a diagram such that the finger transitions are aligned. Conversely, if the latencies were mainly determined by the interpretive processing of the memory code, we would expect to find identical profiles when they are displayed with the codes aligned. For each of the 20 subjects in the experiment, the mean latencies between the elements were calculated for five repetitions. Next, this profile was transformed into ranks numbered 1 through 6. The profiles, shown in Figure 8, present means ranks obtained by averaging over subjects.

Figure 8 clearly shows that in the finger-transition-aligned display the profiles are quite dissimilar, whereas in the codes-aligned display the profiles are almost identical; moreover, the general form follows the specified prediction from the tree traversal interpreter.

It should be noted here that in case a pattern is ambiguous, further analysis is appropriate (Collard & Leeuwenberg, 1981; Restle, 1976, 1979). Here, the last pattern of the above example, (1 2 3 4 3 2), may alternatively be coded by $T_1{}^3(1)$, $T_{-1}(3)$, that is, as a run of four elements followed by a group of two elements. Indeed, inspection of the individual profiles revealed that the timing profiles of some subjects correspond to this latter code rather than to the former code. Such an analysis (Povel & Collard, in press),

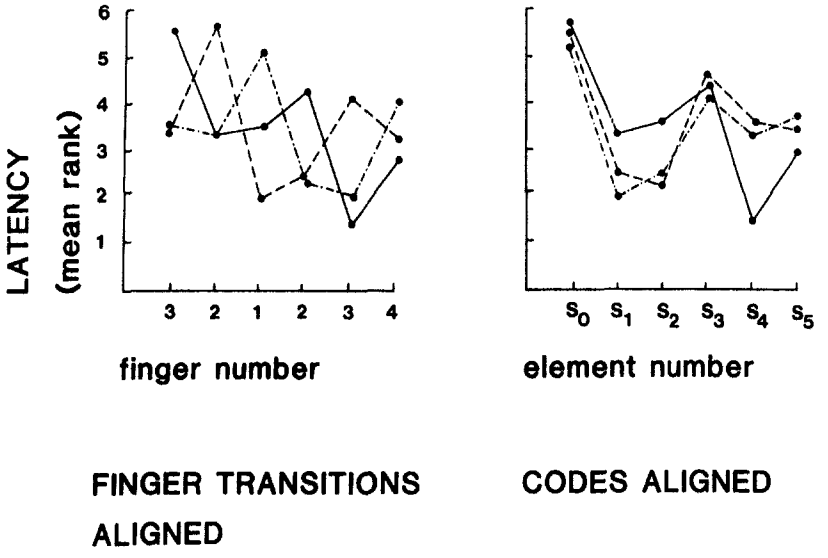

Figure 8. Latency profiles for the three patterns—(3 2 1 2 3 4), (2 1 2 3 4 3), and (1 2 3 4 3 2)—indicated with a straight line, a dashed line, and a dotted–dashed line, respectively.

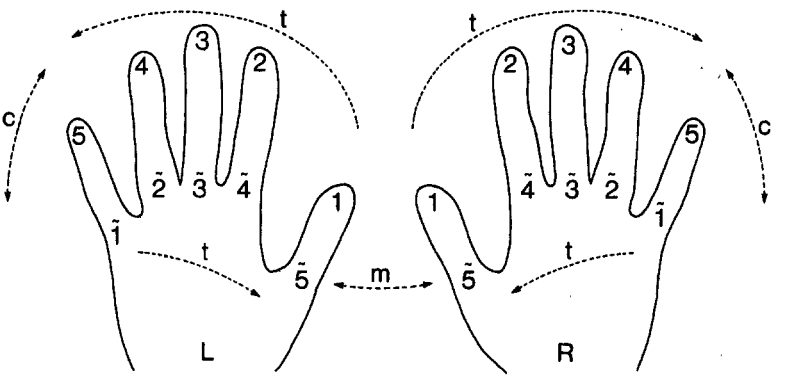

Figure 9. Two different applications of mirroring. Each finger has a complementary finger in the same hand: For instance, $c(1_R) = \tilde{1}_R = 5_R$. In addition, each finger of the right hand can be mirrored to the left hand, and vice versa: For instance, $m(1_R) = 1_L$.

which takes structural ambiguity into account, renders more pronounced profiles than those given in Figure 8.

A Handy Model

The relation between finger patterns and number sequences depends on the fact that adjacent fingers are denoted by consecutive numbers. If the numbers are assigned to the fingers out of order, the serial instructions appear almost impossible to follow. Similarly, the transformations used in the codes can be conceived as descriptions of finger transitions. For example, t_1 may mean "tap with the next finger." This conception is compatible with the subjective experience found when repeatedly producing a finger pattern: Once the movement has started, the actual tapping seems to be done in relational terms, and one no longer realizes what fingers are actually involved. This may correspond to the property of the tree traversal interpreter that we have described as on-line production.

The model for the internal representation of the finger patterns used in the experiment is part of the model given in Figure 9. With the help of the transformations, given in Figure 9, patterns tapped with both hands can also be adequately represented. For an experiment involving both hands, see Rosenbaum, Kenny, and Derr (in press). The model of Figure 9 provides a good example to illustrate the group-theoretical definition of the mirror operation given above. Within each hand there is a transformation of taking complement, denoted by *c*. Consequently, each finger is represented twice. For instance, the right-hand little finger is represented both as the first finger counting from the right and as the fifth finger counting from the left. In addition, there is a mapping from hand to hand that accounts for mirroring hands, denoted by *m*. Here, the double representation of fingers refers, one to one, to the fingers of different hands. Thus, the right-hand little finger, for instance, can be mirrored to the left-hand little finger and vice versa. The basic point of this illustration is that the nature of the mirroring transformation depends on the type of patterns involved. Essentially, mirroring has the function of changing from copy, whereas the copies may or may not coincide.

Extension

From a mathematical point of view, one might ask the question: How is the theory of serial pattern production affected if the transformations used in the codes do not commute? The answer to that question leads to the recent work of Deutsch and Feroe (1981). In their article on the structural aspects of music perception, Deutsch and Feroe propose a coding model that, as they state, "differs from earlier ones in its basic architecture" (p. 504). First, we will see that in case the transformations commute, the frame-

work underlying the classical TRM-model can be converted to their coding model. Second, this conversion is no longer possible if the transformations do not commute: Their approach is the one that relates consistently to structural trees. Third, the tree traversal interpreter as well as the other interpreters apply equally well in this extended framework.

Converting the Memory Code

We will consider only those aspects of the Deutsch and Feroe (1981) model that are essential for analyzing the basic architecture. Essentially, they converted the framework underlying the TRM-model while retaining the same elementary transformations. The conversion will be seen more easily if we rewrite the symbols T_k to the corresponding string of transformations, $[e\ t_k]$ (Greeno & Simon, 1974). Recall that e denotes the identity transformation. So, instead of $T_1(2)$ we will write $[e\ t_1](2)$, because $[e\ t_1](2) = (e(2)\ t_1(2)) = (2\ 3)$. In hierarchical codes, each transformation in turn is applied to the whole string of sequence elements: Thus, $T_3(T_1(1))$ can be rewritten as $[e\ t_3]([e\ t_1](1)) = [e\ t_3](1\ 2) = (e(1\ 2)\ t_3(1\ 2)) = (1\ 2\ 4\ 5)$. We will further refer to this type of application as the classical way.

It is also possible to apply the transformations to sequence elements the other way around; that is, the string of transformations is first applied to the first sequence element, then to the second element, and so on. Let us denote this type of application by writing the sequence elements at the left side of the formulas, for example, $(1\ 4)T_1 = (1\ 4)[e\ t_1] = ((1)[e\ t_1]\ (4)[e\ t_1]) = (1\ 2\ 4\ 5)$. This type of application is called the top-down way because the higher level elements are generated first, which in turn serve to generate the lower level elements (Deutsch & Feroe, 1981). For instance, $((1)T_3)T_1$ can be rewritten as $((1)[e\ t_3])\ [e\ t_1] = (1\ 4)[e\ t_1] = ((1)[e\ t_1]\ (4)[e\ t_1]) = (1\ 2\ 4\ 5)$. Note that in the top-down application the first and third elements are generated before the second and fourth.

The relation between classical codes and top-down codes is fairly straightforward: The codes have only to be converted such that the highest level becomes the lowest level, the second highest the second lowest, and so on. For the above example, $T_3(T_1(1))$ can be converted to $((1)T_3)T_1$ and vice versa, while both represent the sequence (1 2 4 5). Using the commutative property of the transformations, the corresponding theorem for converting hierarchical codes can easily be proved. Why use a converted notation if the old one is equally good? The reason is that the model of Deutsch and Feroe (1981) has a feature that cannot be adequately captured in the classical framework.

Where the Routes Diverge

In their model, Deutsch and Feroe allowed transformations to operate in different alphabets occurring at different levels of a hierarchy. This feature of representation nicely takes into account that in music perception, different alphabets, familiar through prior learning, may be simultaneously involved. The important thing here is that if different alphabets are allowed in one and the same representation, the coding model is, in practice, noncommutative and cannot therefore be incorporated in the classical framework underlying the TRM-model. A typical, though somewhat artificial, example may help to clarify the two possible routes. Suppose k is a transformation that squares its argument, for example, $k(2) = 4$, $k(3) = 9$, and so on. Thus, $K(2) = (2\ 4)$ and $K(3) = (3\ 9)$. Note that k does not commute with transpositions, for example, $t_1(k(2)) = 5$ whereas $k(t_1(2)) = 9$. Now, the two ways of defining hierarchical codes appear to be no longer compatible: $T_1(K(2)) = (2\ 4\ 3\ 5)$ whereas, conversely, $((2)T_1)K = (2\ 4\ 3\ 9)$. So, if the transformations fail to commute, the conversion between classical codes and top-down codes is no longer possible.

The left-hand tree of Figure 6 already provided an example showing that structural trees do not apply to classical codes if the transformations do not commute; however, as will be clear now, they do apply precisely to codes defined in the top-down fashion. So, we conclude that the top-down way of defining hierarchical codes fits exactly the concept of a structural tree, whereas the classical way is appropriate only if the transformations commute. As the basic idea proposed in this

article is that a structural tree actually corresponds to the interpretive process that operates on a hierarchical code, we will examine more closely the way in which Deutsch and Feroe (1981) handled the production of pitch sequences from memory.

Tree Traversals Again

In their article, Deutsch and Feroe (1981) employed the doubling interpreter to write out the codes in a top-down fashion. Furthermore, they assumed that intermediate results of the decoding remain stored in a fully redundant tree, similar to the one given in Figure 5. The main advantages of these assumptions are that errors made at a high level will be reflected at lower levels because the elements are generated in a top-down fashion and that the higher level elements will be recalled better because they are more often represented. This is in accordance with musical intuitions and assumptions generally made by music theorists, as Deutsch and Feroe (1981) state. In fact, the same two advantages apply to the tree traversal interpreter, as can be seen in Figure 2: Errors made while traversing the higher levels will be carried over to the lower levels, and the higher the element in the tree, the more often the element will be encountered.

A final point to be considered is that Deutsch and Feroe's assumptions imply that a sequence of pitches is represented internally, during production, in a completely elaborated network of notes as if the sequence were a visual display, where all information is given in parallel. The tree traversal interpreter, however, assumes that the pitches are produced in an on-line fashion from the knowledge of a concise memory code. Even if the hierarchical structure of a serial pattern is indeed represented internally, during production, as a redundant network of elements, then the tree traversal interpreter may be seen as assembling such a network, not in a parallel top-down fashion like the doubling interpreter but on-line in a serial fashion. These extensions to other sorts of coding models need further elaboration and empirical testing. A point that certainly deserves more attention is the different possible implementations of the production models along with refined assumptions about human processing of structural information.

Discussion

In this article a production model has been presented that fits exactly the relation between hierarchical memory codes and structural trees. According to the tree traversal interpreter proposed, a sequence that is represented internally by a hierarchical code is produced by traversing the corresponding structural tree. The comparative study of different possible production models can be detached from the way codes are defined (Greeno & Simon, 1974). For instance, in case the transformations commute, hierarchical codes may also be defined in terms of the coding model of Deutsch and Feroe (1981), which appears to be appropriate for extensions as well. Generally, the importance of commutative systems, like the classical TRM-model, lies in the fact that codes are invariant under transformations, that is, $t(C(x)) = C(t(x))$. Thus, when a pattern is transformed, the structure remains invariant while the parameters vary, a point also emphasized by Restle (1979) for motion patterns and by Collard and Buffart (in press) for the encoding of serial patterns.

The experiments discussed in this article refer to particular applications in serial pattern research. A production model that has proven to be valid in one application is not necessarily equally valid in another. Therefore, empirical testing is required for each specific application. The main difficulty in testing models for serial pattern production is to effectively disentangle all the processes involved. In the simple experiment on finger tapping referred to earlier, a relatively complicated design was required to separate experimentally the process of decoding from the process of execution. In principle, both processes may affect performance measures such as latencies and errors. When more complex tasks like typing, speaking, and piano playing are studied (Shaffer, 1982; Sternberg, Monsell, Knoll, & Wright, 1978), the problem becomes even more intricate. For, in such tasks, the interaction between the structural characteristics of the memory

representation and the physical motor demands of the execution process are likely to be much more complicated.

Another instance of tangled processes can be observed when someone listens to a piece of music: On the one hand, internal representations are built up or activated; on the other hand, inferences are drawn from those representations. With respect to the inferences, it may be that the listener makes his or her expectations in an on-line fashion on the basis of the preceding tones and the structural information available.

References

Aho, A. V., & Ullman, J. D. *The design and analysis of computer algorithms.* Reading, Mass.: Addison-Wesley, 1974.

Collard, R. F. A., & Buffart, H. F. Minimization of structural information: A set-theoretical approach. *Pattern Recognition*, in press.

Collard, R. F. A., & Leeuwenberg, E. L. J. Temporal order and spatial context. *Canadian Journal of Psychology*, 1981, *35*, 323–329.

Collard, R. F. A., Vos, P. G., & Leeuwenberg, E. L. J. What melody tells about metre in music. *Zeitschrift fur Psychology*, 1981, *189*, 25–33.

Deutsch, D. The processing of structured and unstructured tonal sequences. *Perception & Psycophysics*, 1980, *28*, 381–389.

Deutsch, D., & Feroe, J. The internal representation of pitch sequences in tonal music. *Psychological Review*, 1981, *88*, 503–522.

Geissler, H., Klix, F., & Scheidereiter, U. Visual recognition of serial structure. In E. Leeuwenberg & H. Buffart (Eds.), *Formal theories of visual perception.* New York: Wiley, 1978.

Greeno, J. G., & Simon, H. A. Processes for sequence production. *Psychological Review*, 1974, *81*, 187–197.

Jones, M. R., & Zamostny, K. P. Memory and rule structure in the prediction of serial patterns. *Journal of Experimental Psychology: Human Learning and Memory*, 1975, *104*, 295–306.

Klahr, D., & Wallace, J. G. The development of serial completion strategies: An information processing analysis. *British Journal of Psychology*, 1970, *61*, 243–357.

Kotovsky, K., & Simon, H. A. Empirical tests of a theory of human acquisition of concepts for sequential patterns. *Cognitive Psychology*, 1973, *4*, 399–424.

Leeuwenberg, E. L. J. Quantitative specification of information in sequential patterns. *Psychological Review*, 1969, *76*, 216–220.

Leeuwenberg, E. L. J. A perceptual coding language for visual and auditory patterns. *American Journal of Psychology*, 1971, *84*, 307–349.

Povel, D. J. Internal representation of simple temporal patterns. *Journal of Experimental Psychology: Human Perception and Performance*, 1981, *7*, 3–18.

Povel, D. J., & Collard, R. F. A. Structural factors in patterned finger tapping. *Acta Psychologica*, in press.

Restle, F. Theory of serial pattern learning: Structural trees. *Psychological Review*, 1970, *77*, 481–495.

Restle, F. Structural ambiguity in serial pattern learning. *Cognitive Psychology*, 1976, *8*, 357–381.

Restle, F. Coding theory of the perception of motion configurations. *Psychological Review*, 1979, *86*, 1–24.

Restle, F., & Brown, E. R. Serial pattern learning. *Journal of Experimental Psychology*, 1970, *83*, 120–125.

Rosenbaum, D. A., Kenny, S. B., & Derr, M. A. Hierarchical control of rapid movement sequences. *Journal of Experimental Psychology: Human Perception and Performance*, in press.

Shaffer, L. H. Rhythm and timing in skill. *Psychological Review*, 1982, *89*, 109–122.

Simon, H. A. Complexity and the representation of patterned sequences of symbols. *Psychological Review*, 1972, *79*, 369–382.

Simon, H. A. Induction and representation of sequential patterns. In E. L. J. Leeuwenberg & H. F. J. M. Buffart (Eds.), *Formal theories of visual perception.* New York: Wiley, 1978.

Simon, H. A., & Kotovsky, K. Human acquisition of concepts for sequential patterns. *Psychological Review*, 1963, *70*, 534–546.

Simon, H. A., & Sumner, R. K. Pattern in music. In B. Kleinmutz (Ed.), *Formal representation of human judgment.* New York: Wiley, 1968.

Sternberg, S., Monsell, S., Knoll, R. L., & Wright, C. E. The latency and duration of rapid movement sequences: Comparison of speech and writing. In G. Stelmach (Ed.), *Information processing in motor control and learning.* New York: Academic Press, 1978.

Vitz, P., & Todd, R. A coded element model of the perceptual processing of sequential stimuli. *Psychological Review*, 1969, *76*, 433–449.

Received December 23, 1981
Revision received May 25, 1982 ■