Syntax-Directed Transduction

P. M. LEWIS II AND R. E. STEARNS

General Electric Research and Development Center, Schenectady, New York

ABSTRACT. A transduction is a mapping from one set of sequences to another. A syntax-directed transduction is a particular type of transduction which is defined on the grammar of a context-free language and which is meant to be a model of part of the translation process used in many compilers. The transduction is considered from an automata theory viewpoint as specifying the input-output relation of a machine. Special consideration is given to machines called translators which both transduce and recognize. In particular, some special conditions are investigated under which syntax-directed translations can be performed on (deterministic) pushdown machines. In addition, some time bounds for translations on Turing machines are derived.

KEY WORDS AND PHRASES: automata, Turing machines, context-free languages, transduction, compilers, recognizers, syntax-directed compilers, translation, context-free grammar, deterministic machines

CR CATEGORIES: 5.22, 5.23, 5.24

1. *Introduction*

A transduction is any mapping from one set of sequences to another. The object of this paper is to study, from an automata theory viewpoint, a special class of transductions that models a part of the translation process used in many compilers.

Our starting point is like that of Irons [1] in that we define the translation/transduction by means of the syntax of a context-free grammar. We look upon this definition, however, as specifying only the input-output relation of the transducer and not as a part of the translation algorithm. Thus, the transducer is not required to grammatically analyze the input string. Examples can be given (Section 4) where transduction is computationally more difficult and others where it is computationally simpler than grammatical analysis.

Let L be a context-free language defined over terminal alphabet T and generated by context-free grammar G using nonterminal alphabet N, starting symbol S, and productions of the form $A \rightarrow w$ where A is in N and w in $(N \cup T)^*$. Let "$\Rightarrow$" represent the transitive completion of "$\rightarrow$", i.e., $w_1 \Rightarrow w_2$ if and only if w_2 can be generated from w_1 using productions in grammar G.

A *transduction grammar* G_t based on the context-free grammar G is a triple (G, T', g), where T' is a set of output terminals and g is a mapping defined on G which associates a word w' in $(T' \cup N)^*$ for each production $A \rightarrow w$ in G and which specifies a one-to-one correspondence that pairs each instance of a nonterminal in w with an instance of the same nonterminal in w'. We refer to word w' as the *transduction element* for production $A \rightarrow w$.

For example, production $A \rightarrow aBC$ might have transduction element $xCyB$,

Presented at the IEEE Seventh Annual Symposium on Switching and Automata Theory, October, 1966

Journal of the Association for Computing Machinery, Vol. 15, No. 3, July 1968, pp. 465–488.

which we indicate by writing $A \rightarrow aBC\{xCyB\}$. For production $A \rightarrow aBcDD$, we might write $A \rightarrow aBcD^{(1)}D^{(2)}\{xD^{(2)}zBD^{(1)}r\}$, where the superscripts are added to indicate the pairing of the D's.

If for each production $A \rightarrow w$ in transduction grammar G_t, the nonterminals of w occur in the same order as the corresponding nonterminals of the transduction element, then we call G_t a *simple* transduction grammar. If the productions have the additional property that the nonterminals of w' occur to the left of the symbols from T', then G_t is called a *simple Polish* transduction grammar. In these cases, the correspondence between nonterminals of w and w' can be determined directly from w and w' and no additional specification is required.

As an example of these concepts, the following is a simple Polish transduction grammar corresponding to the arithmetic portion of an ALGOL-like grammar.

$S \rightarrow E$	$\{E\}$	$T \rightarrow T*P$	$\{TP*\}$
$E \rightarrow E + T$	$\{ET+\}$	$T \rightarrow T/P$	$\{TP/\}$
$E \rightarrow E - T$	$\{ET-\}$	$T \rightarrow P$	$\{P\}$
$E \rightarrow T$	$\{T\}$	$P \rightarrow (E)$	$\{E\}$
$E \rightarrow -T$	$\{T \ominus\}$	$P \rightarrow i$	$\{i\}$

The transduction elements may be thought of as defining a second grammar G', where production $A \rightarrow w'$ is in G' if and only if w' is the transduction element associated with a production $A \rightarrow w$ in G. Each derivation from S using G has a corresponding derivation using G' which is obtained by substituting corresponding productions for corresponding nonterminals. Thus for each derivation of word w in L there is a corresponding derivation leading to a word w' in $(T')^*$. The word w' is called a *translation* of w *induced by* G_t.

Any mapping of L into $(T')^*$ which maps each w in L into a translation of w induced by G_t is called a *syntax-directed* (SD) *translation* of G induced by G_t. If G is unambiguous, there is only one induced translation. Any mapping of T^* into $(T')^*$ whose restriction to L is an SD translation induced by G_t is called a *syntax-directed transduction* of G induced by G_t.

Let M be a deterministic machine which

(1) reads a one-way input tape with input alphabet T and with a special marker used at the end of the tape;

(2) stops for all input tapes;

(3) outputs a sequence from alphabet T'.

Machine M is called a *transducer* for the transduction grammar G_t if and only if the mapping of T^* into $(T')^*$ defined by the machine is an SD transduction induced by G_t. If the stopping states of M are divided into accepting and rejecting states, then M is called a *translator* for G_t if it is both a transducer for G_t and a recognizer for L.

Informally, an SD translation can be thought of as a grammatical rearrangement followed by a dictionary look-up, as for example occurs in the translation of infix to Polish notation. More generally, a translation for w may be obtained in the following way:

1. Parse w into a tree.
2. Delete the terminals in T.
3. For each node in the tree, reorder the branches descending from that node

to correspond with the order in the transduction element associated with the production denoted by that node.

4. At each node add terminals from T' as indicated by the associated transduction element.
5. The terminal string in $(T')^*$ implied by the tree is the translation.

This latter description conforms more to the intuitive notion that the translation is somehow accomplished by parsing the given string. We do not care, however, whether or not the machine actually follows these steps. By focusing attention on the entire job of translation rather than on intermediate steps such as parsing, we are able to side-step any such issues as formats of parsing trees or what shall be considered a parse. This is in accord with our basic philosophy that implementing a translation should be viewed as an automata theory problem of machine capability and efficiency rather than as a problem of grammatical analysis. Most of our general results to date, however, are based on using well-known recognition or parsing schemes, and for this reason, the results are mainly about translations rather than transductions.

Although our primary interest is in deterministic machines that stop, we wish to define nondeterministic translation for purposes of characterization. (See Theorems 2 and 3.) Therefore, we say a nondeterministic machine M is a nondeterministic translator for transduction grammar G_t if and only if

(1) any sequence of machine operations leading to an accepting state produces an output which is a translation of the input sequence;

(2) for each w in the language L and each translation w' of w induced by G_t, there is a sequence of machine operations which has input sequence w and output sequence w', and which leads to an accepting state.

The definition of SD translation can be considered as a special case of a definition in Petrone [2]. A similar definition is made in Culik [3]; a translation program for it appears in Kopriva [4]. More general translations are considered in Younger [5].

2. *Relationship Between Simple SD Translation and Pushdown Machines*

Although SD transductions and translations are defined in order to study translators and compilers, they have a certain independent interest in automata theory because of the intimate relationship between simple SD translations and pushdown machines. This relationship can be best understood in terms of the further concepts of LR(k) grammars (Knuth [6]) and LL(k) grammars.

A. Properties of LR(k) and LL(k) Grammars. We first define LR(k) and LL(k) grammars. For a given word w and nonnegative integer k, we define w/k to be w if the length of w is less than or equal to k, and we define w/k to be the string consisting of the first k symbols of w if w has more than k symbols. If A is a nonterminal, w is a word in $(N \cup T)^*$, and p is the name of a production in G, we write $A \Rightarrow w\ (p)$ if and only if w can be derived from A after first applying production p.

A grammar is called LR(k) if it is unambiguous and for all w_1, w_2, w_3, w_3' in T^* and A in N, $S \Rightarrow w_1Aw_3$, $A \Rightarrow w_2$, $S \Rightarrow w_1w_2w_3'$, and $w_3/k = w_3'/k$ imply that $S \Rightarrow w_1Aw_3'$.

A grammar is called LL(k) if, for all w_1, w_2, w_2', w_3, w_3' in T^*, A in N, and p, p' in G, $S \Rightarrow w_1Aw_3$, $S \Rightarrow w_1Aw_3'$, $A \Rightarrow w_2\ (p)$, $A \Rightarrow w_2'(p')$, and $(w_2w_3)/k = (w_2'w_3')/k$ imply $p = p'$.

Stated informally in terms of parsing, an LR(k) grammar is a context-free grammar such that for any word in its language each production in its derivation can be identified and its descendants determined with certainty by inspecting the word from its beginning (left) to the kth symbol beyond the rightmost descendant. An LL(k) grammar is similar except that a production and its leftmost descendant can be identified from the symbols to the left of this leftmost descendant and the k symbols which follow (counting the leftmost descendant terminal as the first symbol). By reversing the roles of left and right one can obtain RL(k) and RR(k) grammars. In earlier versions of this work, LL(k) was called TD(k).

A number of properties of LR(k) grammars are given by Knuth [6]; similar properties can be derived for LL(k) grammars (Appendix I). Here some of these properties that relate to pushdown machines are informally discussed. Briefly, LL(k) and LR(k) grammars are those that can be recognized top-down and bottom-up respectively by (online, deterministic) pushdown machines, scanning from left to right.

More precisely, for LR(k) grammars: (1) Any context-free language that has an LR(k) grammar can be recognized (bottom-up) by a (deterministic) pushdown machine. Moreover, in a manner that can be precisely defined, the machine "uses" the grammar in the recognition process and can be said to recognize each production (Knuth [6]), and

(2) conversely, given any pushdown machine there is an effective procedure for finding an LR(1) grammar for the language recognized by the machine (Knuth [6]).

For LL(k) grammars: Any context-free language that has an LL(k) grammar can be recognized (top-down) by a (deterministic) pushdown machine. The machine uses a predictive recognition scheme (Oettinger [7]) and, in a manner that can be precisely defined, "uses" the grammar in the recognition process and can be said to recognize each production. This machine is derived in Appendix I.

The concepts of top-down and bottom-up recognition are clarified in the examples of the next two sections and in Appendix I.

B. Simple SD Translation and Predictive Recognition Schemes on Pushdown Machines. Since LL(k) and LR(k) grammars are "used" by the machine in the recognition process, translations defined on these grammars can often be performed by the same recognizing machine with only small modifications. Consider the LL(1) grammar and simple SD translation given by

$$S \rightarrow 1S2S \qquad \{xSySz\}$$
$$S \rightarrow 0 \qquad \{w\}$$

This simple SD translation is of a general form since all other simple SD translations can be obtained by substituting words for x, y, z, and w.

The language generated by this grammar can be recognized by a predictive scheme on an online pushdown machine. Initially there is an S written on the pushdown stack. In scanning the sequence from left to right:

(a) When the machine sees a 1, it has recognized, in a predictive manner, the first production. It verifies that there is an S on top of the stack and then pops the S off of the stack and pushes $S2S$ down on the stack.

(b) When the machine sees a 2, it verifies that there is a 2 on top of its stack, and then pops the 2 off of the stack.
(c) When the machine sees a 0, it has recognized the second production; it verifies that there is an S on top of the stack and pops the S off its stack.
(d) If the stack is empty at the end of the sequence, the machine recognizes the sequence.

The general simple SD translation can be performed by the same machine with only trivial modifications.

(a) When the machine sees a 1, it has recognized the first production. It therefore produces an output x, as required by the transduction element, verifies that there is an S on top of the stack, pops it, and pushes $Sy2Sz$ down on the stack (the top symbol being the leftmost S). The extra symbols, y and z, are ignored by the translation portion of the machine, but are produced as output when they pop up later in the process.
(b) When the machine sees a 0, it has recognized the second production. It therefore produces w as output, verifies that there is an S on top of the stack, pops the S off its stack, and then outputs whichever of the symbols y or z is below the S on the stack and pops off this symbol.
(c) When the machine sees a 2 it verifies that there is a 2 on top of the stack and then pops the 2 off of the stack.

This procedure is strongly dependent on the SD translation being simple. Since the nonterminals are not permuted in the transduction elements, all of the required information can be stored in the same stack that does the recognition. This example can be generalized in a rather straightforward manner as follows:

THEOREM 1. *Any simple SD translation of an LL(k) grammar can be performed on a (deterministic) pushdown machine.*

PROOF. All of the simple transduction elements have the form $w_0A_1w_1 \cdots A_nw_n$ for some $n \geq 0$ where A_1 to A_n are the nonterminals in the corresponding production p and w_0 to w_n are words over the transduction alphabet T'. The recognizer of Appendix I is converted to the desired transducer as follows: When the production p is identified, the word w_0 is immediately produced as output. For each i, the pushdown machine places w_i on its pushdown tape immediately below the corresponding entry (A_i, R). When the w_i entry is later encountered, it is to be produced as output and popped off. Because this encounter takes place immediately after all the descendants of A_i have been processed, this output assumes its proper place in the output sequence and the translation is produced as specified. |

A key factor in the above argument is that the recognition scheme is predictive; that is, after seeing only the first k symbols of the final result of a production, the recognizer can identify the entire production and thus knows what outputs to produce. For non-LL(k) grammars such a predictive scheme is not possible on a deterministic pushdown machine. However, a nondeterministic pushdown machine can recognize any context-free grammar in a predictive fashion (Chomsky [8]). Applying the reasoning of Theorem 1 to this result, we have:

THEOREM 2. *Any simple SD translation of an arbitrary context-free grammar can be performed on a nondeterministic pushdown machine.* |

This theorem has a converse which is the transduction analogy to the result that

the only sets recognized by nondeterministic pushdown machines are context-free languages.

THEOREM 3. *Any translation performed by a nondeterministic pushdown machine can be effectively described as a simple SD translation.*

PROOF. Given a pushdown machine, a grammar can be obtained whose nonterminals reflect the machine operation in a very natural way and transduction elements can be assigned so that the proper outputs are associated with these operations. A canonical way of doing this is described in Appendix II. It uses the same principles as a construction of Knuth [6]. |

C. SIMPLE POLISH SD TRANSLATION AND BOTTOM-UP RECOGNITION SCHEMES ON PUSHDOWN MACHINES. The case of LR(k) grammars is now considered. Even though such a grammar can always be used by a pushdown machine for bottom-up recognition, it is not always possible to perform a given simple SD translation for that grammar on a pushdown machine. This is because the transduction element may call for an output long before the occurrence of the production can be verified. However, in the case of a simple Polish translation, the outputs can be given within k inputs after their place in the transduction element (i.e., the end) is encountered, and so pushdown translation is possible.

As a simple example consider the LR(0) (but not LL(k)) grammar and simple Polish SD translation given by

$$S \rightarrow 1SS0 \qquad \{SSx\}$$

$$S \rightarrow 1SS1 \qquad \{SSy\}$$

$$S \rightarrow 2 \qquad \{z\}$$

This language can be recognized bottom-up on a pushdown machine. Initially the pushdown tape is blank. In scanning the input sequence from left to right:

(a) If the input symbol is a 2, the machine has recognized the third production and pushes an S down on the pushdown stack.

(b) If the input symbol is a 1, the machine checks whether or not the top three symbols on the stack are $1SS$ (the top symbol being the rightmost S).[1]

  (i) If the top symbols are $1SS$, the machine has recognized the second production. It therefore pops the symbols $1SS$ off the stack, replacing them with an S.

  (ii) If not, the machine pushes the 1 down on the stack.

(c) If the input symbol is a 0, the machine verifies that the top three symbols on the stack are $1SS$. If not, it rejects the sequence. If the top symbols are $1SS$, it has recognized the first production and pops the symbols $1SS$ off the stack, replacing them with an S.

(d) If the end of the input string is reached and the stack is reduced to a single S, the machine accepts the sequence.[2]

[1] Technically, this is not a legal pushdown operation, and must be implemented by storing the top two symbols in the finite-state control instead of on the tape or by keeping extra information on the tape as in the construction of Knuth [6].

[2] Note that in bottom-up recognition each production is recognized after all of its descendants and before all of its ancestors, while in the top-down recognition described earlier each production is recognized after all of its ancestors and before all of its descendants. Because of this, the

The given simple Polish translation of this grammar can be performed by the same machine with only simple modifications:

(a) Whenever the machine sees a 2 and has therefore recognized the third production, it outputs a z.

(b) Whenever the machine sees a 1 which it recognizes as the last symbol of the second production, it produces y as output.

(c) Similarly, when the machine recognizes the first production it outputs x.

This procedure is strongly dependent on the transduction being simple Polish. The machine cannot identify any production until it has seen the entire production, but since the transduction is simple Polish it need not produce any output until the production is completed. Since the standard construction of Knuth [6] always identifies a given production before the productions above it and before the productions to its right, this procedure of outputting the appropriate words as each production is identified will work for any simple Polish SD translation on an LR(k) grammar, and so we have:

THEOREM 4. *Any simple Polish SD translation of an LR(k) grammar can be performed on a (deterministic) pushdown machine.*

(A special case of this theorem, namely the translation of infix to Polish notation by a pushdown machine, was demonstrated by Oettinger [7].)

Theorem 4 has a converse which is the transduction analogy to the result of Knuth that only languages with an LR(k) grammar can be recognized on a pushdown machine.

THEOREM 5. *Any translation performed by a (deterministic) pushdown machine can be effectively described as a simple Polish SD translation on an LR(k) grammar.*

PROOF. The LR(k) grammar and simple Polish transduction elements are obtained as described in Appendix II. |

D. A TRANSLATION SCHEMA FOR LR(k) GRAMMARS. In this section, the translation problem is considered from the point of view of a person who has a grammar and wishes to specify transduction elements so that words from his language translate into appropriate object code. Presumably there are many ways this assignment can be made, and the difficulty of translation may depend heavily on what transduction elements are assigned. The object of the section is to show that, when the given grammar is LR(k), a rather general set of sufficient conditions can be obtained such that any assignment satisfying these conditions has a pushdown translator. These sufficient conditions are generally expressed by a set of parameterized transduction elements which we refer to as a "translation schema."

To see how such a translation schema could be derived in a simple case, consider the LR(0) grammar

$$S \rightarrow 1S0$$

$$S \rightarrow 1S1$$

$$S \rightarrow 2$$

The most general form of simple transduction elements that can be attached to this

nonterminals written on the tape are, for bottom-up recognition, ones that have already been recognized; for top-down recognition, the nonterminals written are those that have been predicted but have not yet been recognized.

grammar is as follows:

$$\begin{aligned} S &\to 1S0 && \{w_1Sw_2\} \\ S &\to 1S1 && \{w_3Sw_4\} \\ S &\to 2 && \{w_5\} \end{aligned}$$

Observe that any simple translation can be obtained by assigning proper values to the parameters w_i.

The only general way we have of obtaining a pushdown translation is through Theorem 4, so we look for a simple Polish description of this translation. This is done by an easy trick of introducing extra nonterminals W_i, and it results in the following description:

$$\begin{aligned} S &\to W_1SW_2 && \{W_1SW_2\} & \qquad W_1 &\to 1 && \{w_1\} \\ S &\to W_3SW_4 && \{W_3SW_4\} & \qquad W_2 &\to 0 && \{w_2\} \\ S &\to W_5 && \{W_5\} & \qquad W_3 &\to 1 && \{w_3\} \\ & && & \qquad W_4 &\to 1 && \{w_4\} \\ & && & \qquad W_5 &\to 2 && \{w_5\} \end{aligned}$$

Unfortunately this altered grammar is no longer LR(k) since an initial input 1 may represent either an application of rule $W_1 \to 1$ or an application of rule $W_3 \to 1$, and one cannot bound the number of additional input symbols that might be required before the identity of this production is determined. Thus Theorem 4 still cannot be applied. To circumvent this difficulty, we equate W_1 and W_3 to obtain

$$\begin{aligned} S &\to W_1SW_2 && \{W_1SW_2\} & \qquad W_1 &\to 1 && \{w_1\} \\ S &\to W_1SW_4 && \{W_1SW_4\} & \qquad W_2 &\to 0 && \{w_2\} \\ S &\to W_5 && \{W_5\} & \qquad W_4 &\to 1 && \{w_4\} \\ & && & \qquad W_5 &\to 2 && \{w_5\} \end{aligned}$$

This grammar is LR(0) and so Theorem 4 applies. Eliminating the W_i, we are assured that the schema

$$\begin{aligned} S &\to 1S0 && \{w_1Sw_2\} \\ S &\to 1S1 && \{w_1Sw_4\} \\ S &\to 2 && \{w_5\} \end{aligned}$$

has a pushdown translation for all values of w_1, w_2, w_4, and w_5. This concludes the example.

Any grammar can be converted into a "general form" Polish by changing a production such as

$$A \to 0AB10$$

into

$$\begin{aligned} A &\to W_1AW_2BW_3 \\ W_1 &\to 0 \end{aligned}$$

$$W_2 \rightarrow \epsilon$$

$$W_3 \rightarrow 10$$

Unfortunately, this form does not lend itself to the second step of making the result LR(k) again. However, there is a natural alternative to general form Polish which is ideally suited for the second step and which yields a schema with considerable freedom. We call this alternative *symbol Polish*.

A production is transformed into symbol Polish by replacing each instance of a symbol from $N \cup T$ on the right-hand side by a symbol $W_j{}^A$, where A is the symbol being replaced and j is some index to indicate which instance of the symbol A in the grammar is being replaced. The symbol $W_j{}^A$ is treated as a new nonterminal and a production of the form $W_j{}^A \rightarrow A$ is added to the grammar. Each of these new productions is assigned transduction element $Aw_j{}^A$ if A is a nonterminal or transduction element $w_j{}^A$ if A is a terminal. The transformed productions are assigned simple transduction elements without any terminals. This derived grammar is called the *derived symbol Polish* grammar. A pair of new nonterminals are called *compatible* if they have the same superscript.

For example, production

$$A \rightarrow BaAa$$

might be transformed into

$$\begin{aligned} A &\rightarrow W_1{}^B W_1{}^a W_1{}^A W_2{}^a && \{W_1{}^B W_1{}^a W_1{}^A W_2{}^a\} \\ W_1{}^B &\rightarrow B && \{Bw_1{}^B\} \\ W_1{}^A &\rightarrow A && \{Aw_1{}^A\} \\ W_1{}^a &\rightarrow a && \{w_1{}^a\} \\ W_2{}^a &\rightarrow a && \{w_2{}^a\} \end{aligned}$$

which would then be part of the derived Polish grammar. New nonterminals $W_1{}^a$ and $W_2{}^a$ would be a compatible pair. If need be, these compatibles and their respective parameters $w_1{}^a$ and $w_2{}^a$ could be equated or identified to obtain alternate productions

$$\begin{aligned} A &\rightarrow W_1{}^B W_1{}^a W_1{}^A W_1{}^a && \{W_1{}^B W_1{}^a W_1{}^A W_1{}^a\} \\ W_1{}^B &\rightarrow B && \{Bw_1{}^B\} \\ W_1{}^A &\rightarrow A && \{Aw_1{}^A\} \\ W_1{}^a &\rightarrow a && \{w_1{}^a\} \end{aligned}$$

A derived Polish grammar obviously defines the same language as the original grammar. In translating a word from this language, the recognition of a symbol A signals the occurrence of a $W_j{}^A \rightarrow A$ for some j. Thus, if all the compatible nonterminals and corresponding productions are equated, then an LR(k) grammar results. We show next how a minimal set of such identifications can be chosen so that the result is LR(k). To do this, we introduce some further notation.

A *sentential form* of a language L generated by grammar G is any word over $N \cup T$ that can be derived from S by that grammar. Given the derived symbol

Polish grammar G' for G and a pair of distinct compatible nonterminals W_i^A and W_j^A for some A in $N \cup T$, we define the *distinction index* for this pair to be the smallest integer greater than the lengths of the w in T^* such that, for some w_1, w_2, and w_3 in T^*, $w_1W_i^Aww_2$ and $w_1W_j^Aww_3$ are sentential forms of L derived from G'. The index is ∞ if these lengths are unbounded and 0 if no such w exists. For LR(k) grammars, these distinction indexes are computable and determine which compatibles should be identified.

THEOREM 6. *Given an LR(k) grammar, one can effectively compute the distinction index for each pair of compatible nonterminals in the corresponding derived symbol Polish grammar.*

PROOF. We assume that there are no productions in the grammar that cannot be used in generating sentences in the language. Such productions can always be identified and discarded from the grammar. It is also convenient to assume that an infinite number of strings can be generated from each nonterminal A. Any grammar can be modified to have this property by replacing each instance of an offending nonterminal by the various elements of T^* that can be derived from it; such a modification leaves the grammar LR(k), since the parsing process can imagine that these replacements have not really been made.

We make extensive use of the following property of LR(k) grammars: There is at most one correct way to parse the string preceding some given substring of k symbols, and this way is independent of what follows the substring. Expressed symbolically,

$$S \Rightarrow \bar{w}_1w_2w_3\,, \quad \bar{w}_1 \Rightarrow w_1\,, \quad S \Rightarrow w_1w_2\bar{w}_3\,, \quad w_2 \text{ in } T^*,$$

and

$$l(w_2) \geq k \text{ implies } S \Rightarrow \bar{w}_1w_2\bar{w}_3\,.$$

We first define some sets used in the proof. For each W_i^A, define

$$L(A, i) = \{w_1W^Aw_2 \mid S \Rightarrow w_1W_i^Aw_2w_3 \text{ for some } w_1, w_2, w_3 \text{ in } (N \cup T)^*\}.$$

For each pair W_i^A and W_j^A, define

$$D(A, i, j) = \{w_2 \mid w_1W^Aw_2 \text{ is in } L(A, i) \cap L(A, j) \text{ for some } w_1\}.$$

The distinction index for pair W_i^A, W_j^A is the smallest integer greater than the length of the longest word in $T^* \cap D(A, i, j)$ if this set is finite, or zero if this set is empty, or ∞ otherwise. Our method consists of a two-part test on this set.

Part I. If there is some wB in $D(A, i, j)$ for some nonterminal B, then the index must be infinite since an infinite number of words can be derived from B. To test for this condition, we need only consider those $w_1W^Aw_2B$ in $L(A, i)$ such that

$$(1) \quad \Downarrow S \Rightarrow \bar{w}_1W_i^A\bar{w}_2Bw_3\,, \quad \bar{w}_1 \Rightarrow w_1\,, \text{ and } \bar{w}_2 \Rightarrow w_2$$

for $\bar{w}_1$ and $\bar{w}_2$ in $(N \cup T)^*$ implies $\bar{w}_1 = w_1$ and $\bar{w}_2 = w_2$. In other words, we need only consider those w_1 which are as high as possible on a derivation tree, and we need not consider any w_1' which can be derived from a w_1. For if $w_1W^Aw_2B$ is also in $L(A, j)$, then $S \Rightarrow w_1W_j^Aw_2B\bar{w}_3$ for some $\bar{w}_3$; and since a word of at least k

symbols can be derived from B, the LR(k) property assures us that

$$S \Rightarrow w_1 W_j{}^A \bar{w}_2 B \bar{w}_3$$

and in fact that $S \Rightarrow \bar{w}_1 W_j{}^A \bar{w}_2 B \bar{w}_3$. This proves that $\bar{w}_1 W^A \bar{w}_2 B$ is also in $L(A, j)$ and $\bar{w}_2 B$ must also be $D(A, i, j)$; so the omission of $w_2 B$ does not affect the existence of such a form in $D(A, i, j)$.

Second, we need only consider those $w_1 W^A w_2 B$ in $L(A, i)$ such that

(2) $S \Rightarrow w_1 W_i{}^A w_2 B w_3$

for some w_3 and the B is a descendant of the production in which the $W_i{}^A$ is generated. In other words, we need not consider the situation where all of the descendants of the production that generates $W_i{}^A$ are to the left of B. For if all of the descendants of that production are to the left of B, the LR(k) property implies that $S \Rightarrow w_1 W_j{}^A w_2 B \bar{w}_3$ is possible for all $\bar{w}_3$ and hence $w_1 W^A w_2$ is not in $L(A, j)$.

It is easily seen that the sentential forms in $L(A, i)$ satisfying conditions (1) and (2) above may be derived from the grammar obtained as follows: For each B in N, introduce two new nonterminals $\mathfrak{B}$ and $\mathfrak{B}'$, represented by script letters. For each production

$$B \rightarrow B_1 \cdots B_n$$

in G, construct new production as follows for $1 \leq r \leq n$: If B_r is in N, construct the production

$$\mathfrak{B} \rightarrow B_1 \cdots B_{r-1} \mathfrak{B}_r .$$

If B_r is the instance of A to be replaced by $W_i{}^A$, B_s is in N for some $S > r$, and B_l is in T for $r < l < s$, construct

$$\mathfrak{B} \rightarrow B_1 \cdots B_{r-1} W^A B_{r+1} \cdots B_{s-1} \mathfrak{B}_s' .$$

If B_r is in N and B_l is in T for all $l < r$, construct

$$\mathfrak{B}' \rightarrow B_1 \cdots B_{r-1} \mathfrak{B}_r' .$$

Finally, for each $\mathfrak{B}'$, construct

$$\mathfrak{B}' \rightarrow B.$$

The starting symbol is $\mathfrak{S}$.

Considering the script symbols as the nonterminals and the symbols in $N \cup T$ as the terminals, this grammar is regular; hence the set of forms it generates is regular. By constructing the regular sets for $W_i{}^A$ and $W_j{}^A$ and testing their intersection for nonemptiness or emptiness, we can determine whether or not $D(A, i, j)$ contains a form wB. If it does, the index is infinite. If not, continue to Part II.

Part II. We now wish to construct $D(A, i, j) \cap T^*$ which, as is to be seen, is regular whenever the issue is not settled by the test of Part I. Having constructed the set, it is then a routine matter to compute the index.

To simplify the discussion, assume for the moment that symbol A is a nonterminal.

First of all, we need consider only those $w_1 W_i{}^A w_2$ in $L(A, i)$ such that

(1) w_2 is in T^* and, for all $\bar{w}_1$ and w_3, $S \Rightarrow \bar{w}_1 W^A w_2 w_3$ and $\bar{w}_1 \Rightarrow w_1$ implies $\bar{w}_1 = w_1$.

(In other words, we need consider only those w_2 that are all terminal symbols and those w_1 that are as high as possible on a derivation tree.) To see this, observe that $S \Rightarrow w_1 W_j{}^A w_2 \bar{w}_3$ for some $\bar{w}_3$ implies $S \Rightarrow \bar{w}_1 W_j{}^A w_2 w_3$ by the LR(k) property, since identical strings of length k or more can be generated from $W_i{}^A$ and $W_j{}^A$ (namely, those strings which can be generated from A). Also note that if w_2 has a length greater than $k-1$, the LR(k) property may be applied even when A is a terminal.

Our second observation is that we need not consider those $w_1 W^A w_2$ such that

$$(2) \quad S \Rightarrow w_1 W_i{}^A w_2' B w_2'' w_3\,, \quad w_2'' \text{ in } T^*, \quad l(w_2'') \geq k, \text{ and } w_2' B w_2'' \Rightarrow w_2\,.$$

This is because if w_2 is in $D(A, i, j)$, then so is $w_2'B$ by the LR(k) property, and this possibility has already been ruled out in Part I.

Finally, we may also rule out those $w_1 W^A w_2$ such that

$$(3) \quad S \Rightarrow w_1'' A' w_2'' w_3\,, \quad A' \rightarrow w_1' W_i{}^A w_2'\,, \quad w_2'' \text{ in } T^*, \quad l(w_2'') \geq k,$$
$$w_1 = w_1'' w_1', \text{ and } w_2' w_2'' \Rightarrow w_2\,.$$

This is because $S \Rightarrow w_1'' w_1' W_j{}^A w_2' w_2'' w_3$ would imply $A' \Rightarrow w_1' W_j{}^A w_2'$ by the definition of LR(k), and the grammar would then be ambiguous.

The sentential forms in $L(A, i)$ satisfying condition (1) and not conditions (2) or (3) may be derived from the grammar obtained as follows. For each B in N and w in T^* such that $l(w) \leq k$, introduce two new nonterminals $\mathfrak{B}(w)$ and $\mathfrak{B}'(w)$. For each production

$$B \rightarrow B_1 \cdots B_n$$

in G and each w in T^* such that $l(w) \leq k$, construct new productions as follows for $1 \leq r \leq n$: If B_r is in N, construct the production

$$\mathfrak{B}(w) \rightarrow B_1 \cdots B_{r-1} \mathfrak{B}_r(\bar{w})$$

for all $\bar{w}$ in $(L_{B_{r+1}} \cdots L_{B_n} w)/k$ (see the notation in the second paragraph of Appendix I). If B_r is the instance of A to be replaced by $W_i{}^A$ and B_s is in N for some $s > r$, and if B_l is in T for $r < l < s$, construct

$$\mathfrak{B}(w) \rightarrow B_1 \cdots B_{r-1} W^A B_{r+1} \cdots B_{s-1} \mathfrak{B}_s'(\bar{w})$$

for all $\bar{w}$ in $(L_{B_{s+1}} \cdots L_{B_n} w)/k$. If B_r is the instance of A and B_l is in T for all $l > r$, construct

$$\mathfrak{B}(w) \rightarrow B_1 \cdots B_{r-1} W^A \bar{w}$$

for each prefix $\bar{w}$ of $B_{r+1} \cdots B_n w$.

If B_r is the first nonterminal in the right-hand side of the production, construct

$$\mathfrak{B}'(w) \rightarrow B_1 \cdots B_{r-1} \mathfrak{B}_r'(\bar{w})$$

for all $\bar{w}$ in $(L_{B_{r+1}} \cdots L_{B_n} w)/k$ and

$$\mathfrak{B}'(w) \rightarrow B_1 \cdots B_l$$

for all $l < r$. If all the B_r are in T, construct

$$\mathfrak{B}'(w) \rightarrow \bar{w}$$

for each prefix $\bar{w}$ of $B_1 \cdots B_n w$. Take $\mathfrak{S}(\epsilon)$ to be the starting symbol.

Observe that this is a regular grammar. By constructing the intersection of the regular sets for W_i^A and W_j^A and removing the prefixes of the form wW^A, one obtains a regular expression for $D(A, i, j) \cap T^*$ from which to calculate the index.

If A is a terminal symbol, the regular set may lack certain words of length less than k. In particular, if the set is empty, the index will be some integer less than k and some further testing is required to determine the value. We omit proof that this testing can be done.

If one wishes to compute the index for some A which has only a finite number of derived words in T^*, this can be done by treating one instance of A as a terminal symbol. |

THEOREM 7. *If for some LR(k) grammar G, we compute the derived symbol Polish grammar and equate all the compatible pairs of nonterminals with distinction index greater than some $\bar{k} \geq k$, then the resulting grammar is LR($\bar{k}$) and Theorem 4 applies.*

PROOF. The old productions and the incompatible productions can all be distinguished from each other with look-ahead k, as previously observed. By definition of the distinction index, the unequated compatible parts can be distinguished in $\bar{k}$. The transduction description remains Polish under the identification of compatibles and so Theorem 4 applies. |

The pushdown translator of Theorem 4 for the LR($\bar{k}$) grammar may be thought of as distinguishing the maximum amount of information looking $\bar{k}$ ahead and outputting accordingly. Obviously, when distinctions of index $\bar{k} > k$ are being made, the translator is more complex than the usual LR(k) recognition machine.

Having constructed the derived grammar and identified the required compatibles, the W_i^A can be substituted out of the description and a rather general schema obtained for the original grammar. This schema can be enhanced by assigning arbitrary transduction elements to the ϵ-productions (normally their transduction elements would be ϵ). No additional generality is obtained by adding to the end of other productions as the last w_i^A can have an arbitrary transduction element independent of all others.

3. *Time Bounds for the Translation of LR(k) Grammars*

Polish translations of LR(k) grammars have already been considered. Because these can be done on pushdown machines, the number of required machine operations is proportional to the length n of the input sequence. In the terminology of Hartmanis and Stearns [9] we would say that the translation can be performed within time $T(n) = n$. In this section, we derive time bounds for more general translations of LR(k) grammars using multitape Turing machines. It is shown that the simple LR(k) translations can be done in time $T(n) = n$ and general LR(k) translations can be done in time $T(n) = n(\log n)^2$. For an arbitrary context-free grammar with arbitrary transduction elements, it is known as a special case of Younger [5] that a translation induced by these transduction elements can be obtained in time $T(n) = n^3$.

A. SIMPLE SD TRANSLATION OF LR(k) GRAMMARS. We now show how a simple SD translation of an LR(k) grammar can be done in time $T(n) = n$ on an (online) machine with three pushdown tapes. This method is more complex than that used for Polish translations in that it has three phases, each of which requires a number

of operations proportional to the input length. First the procedure is described and then an example is given.

Phase 1. During the first phase, the machine receives the input and uses one tape (tape 1) to perform the standard bottom-up recognition. Whenever a production is identified in the recognition process, the name of that production is placed on top of the second pushdown tape (tape 2).

Because these names are stored in the order in which they are recognized, all of the descendant productions of a given production are listed immediately below the name of the given production, and the right-left order of its immediate descendants is preserved in the top-down order. Thus, upon completion of phase 1, the top symbol on tape 2 is the name of the topmost production and the second symbol is the name of the production applied to the rightmost nonterminal of the top production. All the descendants of this rightmost nonterminal are listed before the name of the production applied to the second rightmost nonterminal is given. (Note that phase 1 is nothing but a general Polish translation, where the output symbol for the rth production is P_r.)

Phase 2. During the second phase, we generate a right-left translation and store it on the first pushdown tape (tape 1). First we remove the top symbol of tape 2 which is the name of the topmost production and push its transduction element onto tape 3 with its rightmost symbol on top. On subsequent operations, we check the top element of tape 3. If it is a terminal, we pop it off and push it on top of tape 1. If it is a nonterminal, we discard it, remove the next production from tape 2, and place the transduction element for that production on top of tape 3 (right symbol up). When tape 3 is empty, the second phase is completed. Tape 1 now contains the translation, left symbol up.

Phase 3. For phase 3, we simply pop up tape 1 and output the symbols. The translation is complete.

As an example, suppose an LR(k) translation has the following as part of its description:

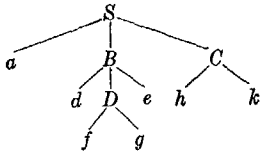
$$P_1: \quad S \rightarrow aBC \qquad \{xByCz\}$$

$$P_2: \quad C \rightarrow hk \qquad \{w\}$$

$$P_3: \quad B \rightarrow dDe \qquad \{Du\}$$

$$P_4: \quad D \rightarrow fg \qquad \{v\}$$

Suppose further that we are translating word *adfgehk* which has derivation



After input h is received during phase 1, the tapes would be as follows (top tape symbols to the right):

1. $a\ B\ h$

2. $P_4\ P_3$
3.

(The description of tape 1 is simplified in that extra symbols are generally required for the recognition.) At the end of phase 1, the tapes would look like:

1.
2. $P_4\ P_3\ P_2\ P_1$
3.

Intermediate in phase 2, we would have

1. $z\ w$
2. $P_4\ P_3$
3. $x\ B\ y$

The end of phase 2 would be

1. $z\ w\ y\ u\ v\ x$
2.
3.

and then the translation $x\ v\ u\ y\ w\ z$ would be read off.

We have proved:

THEOREM 8. *Any simple SD translation of an LR(k) grammar can be done in time $T(n) = n$ using a Turing machine with three pushdown tapes.*

B. GENERAL SD TRANSLATION OF LR(k) GRAMMARS. In this subsection, the methods of the previous subsection are extended to obtain a general translation method for LR(k) grammars that works in time $T(n) = n(\log n)^2$. The previous method breaks down after phase 1 because the natural right-left order of the various descendant productions is no longer suitable for translation. This difficulty can be circumvented by using a procedure which first parses the input string, then reorders the resulting list of productions, and finally performs the translation. This is in contrast with the previous procedure, in which the sequence of operations is parse-translate-reverse.

THEOREM 9. *Any SD translation of an LR(k) grammar can be performed in time $T(n) = n(\log n)^2$ on a Turing machine with three pushdown tapes.*

PROOF. We describe a procedure[3] whereby the list of productions generated by phase 1 can be rearranged into the order required by the transduction elements. The general idea is first to associate with each production in the list a number denoting its correct position in the rearranged list, and then to use a standard sorting algorithm to perform the rearrangement. The procedure is described and then an example is given.

Step A. The first phase is similar to that described in Subsection 3A. Tape 1 is used to perform the standard bottom-up recognition and the names of all the productions used in the derivation are placed on tape 2 in the order in which they are identified. However, in addition to saving the name of the production on tape 2, we also save d numbers with the name where d is the degree of the production (i.e. d is the number of nonterminals on its right-hand side). These numbers are stored on tape 2 immediately below the production name and represent the number of nonterminal nodes in the d subtrees descendant from that production. To accom-

[3] This simplified version of the algorithm was suggested by D. E. Knuth.

plish this, we use tape 3 as a scratch tape to store a list of numbers. These numbers are maintained in a one-to-one correspondence with the nonterminals stored on tape 1 as determined by their order, and each number records the number of nonterminal nodes descendant from its corresponding nonterminal. Thus when a production is identified using tape 1, the d numbers that are supposed to go with it on tape 2 are just the top d numbers on tape 3. After placing these numbers and the production name on top of tape 2, the recognition causes the top d nonterminals of tape 1 to be replaced by a single nonterminal, and so we replace the top d symbols of tape 3 by their sum plus one. Since this sum is the number of nonterminals descendant from the new nonterminal on top of tape 1, the correspondence and interpretation are preserved. Since the lists are initially in correspondence (both are null) the process operates correctly by induction. (Actually, these numbers could be stored on tape 1, each below its corresponding nonterminal, without interfering with the recognition process.)

Step B. The next objective is to place the list of production names back onto tape 1 and to include with each production name, a *sort key* which specifies the position of that production in the rearranged production list corresponding to the left-right top-down order specified by the transduction elements. In this rearranged list, the descendants of each production appear immediately to the right of that production name and the production names that apply to the d nonterminals on the right-hand side of the given production appear in the same order as the corresponding nonterminals are ordered in the transduction element.

We say that a sort key x is assigned to a given instance of a nonterminal A if x is the sort key for the production applied to A. If the production applied to A appears on the top of tape 2, then it is possible to compute the sort keys for the nonterminals on the right side of the production from x and the d numbers which follow the production name. For suppose that the production and its transduction element are

$$A \rightarrow A_1A_2 \cdots A_j \cdots A_k \cdots A_l \cdots A_d\{A_kA_jA_l \cdots A_s\},$$

and suppose that $r_1 \cdots r_d$ are the d numbers below the production name associated respectively with $A_1 \cdots A_d$. From the left-right order specified by the transduction element, the production A_k is to be the first in the rearranged list. Therefore, A_k is assigned the sort key $x + 1$. Then, on the rearranged list will come all of the descendants of A_k followed by the production applied to A_j and then all of its descendants, etc. But it is known from step A that A_k has exactly r_k descendants. Therefore, we can compute that the sort key for A_j should be $r_k + x + 1$. Similarly, the sort key for A_l is $r_k + r_j + x + 1$; the sort keys for the remaining nonterminals on the right-hand side are similarly obtained.

Step B can now be described. Tape 1 is initialized by making it blank (i.e. erasing the S left over from step A) and tape 3 is initialized with a 0 on it (replacing the number left over from phase A). After each step, the top number, x, on tape 3 is the sort key for the top production on tape 2. In the next step this production name and sort key are written on tape 1 and the d numbers $r_1 \cdots r_d$ which follow the production name are used with x as described in the previous paragraph to compute the sort keys for the right-hand nonterminals of the production. These sort keys then replace x on tape 3, the sort key for A_d being on top and the rest below it in order.

The production name and d numbers are then discarded from tape 2. This process is repeated until tape 2 is exhausted. The sort keys are matched with the proper production names since the motions of tape 3 are imitating the motions of a tape used for right-left top-down recognition of the production list.

Step C. At this point any standard sorting algorithm, for which the appropriate time bound can be proved, is used to sort the string on tape 1 using the sort key computed in phase B. For this proof we use the inverted radix algorithm, which is accomplished by sorting on the successive binary digits of the sort keys starting with the least significant digit. Assume that the binary digits of the sort key have been put on tape 1 with the least significant digit on top (if not, reverse the sequence by transferring it digit by digit from tape 1 to tape 2; then renumber the tapes to follow the remaining discussion).

(i) If the top (least significant) digit of the sort key of the top production on tape 1 is a 1, erase the 1 and put the production name and the remaining digits of the sort key on tape 2.

(ii) If the top digit is a 0 (or a blank), erase the digit and put the production name and the remaining digits of the sort key on tape 3.

(iii) Repeat for each production on tape 1.

(iv) Transfer the string on tape 2 one symbol at a time to tape 1.

(v) Transfer the string on tape 3 one symbol at a time to tape 1 (on top of the string transferred in step (iv)).

(vi) If there are no nonblank digits in any of the sort keys in the string on tape 1, that string is the desired sorted string. Otherwise repeat the procedure starting at step (i) on the new string appearing on tape 1.

Step D. During this phase, we generate the translation. First we remove the top symbol of tape 1 which is the name of the topmost production and push its transduction element onto tape 3 with its leftmost symbol on top. On subsequent operations, we check the top element of tape 3. If it is a terminal, we pop it off and produce it as output. If it is a nonterminal, we discard it, remove the next production from tape 1, and place the transduction element for that production on top of tape 3 (left symbol up). When tape 3 is empty, the translation is completed.

To verify the time bound for the procedure, first observe that the number of productions in the derivation of the given word is no greater than a constant multiple of n. (This follows since it can be shown that for unambiguous grammars, the length of derivation from a nonterminal symbol to an empty string is bounded.) Hence the maximum number required for a sort key is a constant times n, and thus only $\log n$ (for some base logarithm) binary digits per name are required to assign sort keys. Therefore, the maximum length of the tape processed at any time during the procedure is bounded by a constant times $n \log n$. We now count the number of operations required by each of steps A through D.

All of the steps except C are real-time operations that can be performed in time proportional to the length of the input string. Thus, for these steps the time bound is $T(n) = n \log n$. For step C, the string is processed $\log n$ times (once for each binary digit in the sort key), and each time the number of operations is proportional to the length $n \log n$. Thus, the time bound for step C and hence for the entire procedure is $T(n) = n (\log n)^2$.

As an example consider the following LR(0) grammar and nonsimple transduction elements:

$$P_1: \quad S \rightarrow aAB \qquad \{rBxAz\}$$

$$P_2: \quad A \rightarrow bc \qquad \{y\}$$

$$P_3: \quad B \rightarrow CD \qquad \{sCuDw\}$$

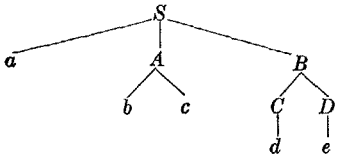
$$P_4: \quad C \rightarrow d \qquad \{t\}$$

$$P_5: \quad D \rightarrow e \qquad \{v\}$$

Assume the input string is

$$a\ b\ c\ d\ e$$

with derivation tree



After each phase of the procedure the tapes would be as follows (top tape symbols to the right): At the end of step A:

1 S
2 P_2 ; P_4 ; P_5 ; 1, 1, P_3 ; 11, 1, P_1
3 100

At the end of step B:

1 P_1 , 0; P_3 , 1; P_5 , 11; P_4 ; 10; P_2 , 100
2
3

At the end of step C:

1 P_2 , P_5 , P_4 , P_3 , P_1
2
3

Then the final output is produced in step D: $r\ s\ t\ u\ v\ w\ x\ y\ z$.

4. *Some Examples*

In this section an example of an SD transduction is given that is computationally easier, and another is given that is computationally more difficult than recognition and/or parsing.

A. Transduction Simpler Than Translation. Consider the following SD transduction defined on any arbitrary context-free grammar: For each production, the transduction element is just the right side of that production reversed. It follows that this transduction has the property of reversing all sequences. Thus for any sequence as input, the output of the transducer is just the reverse of that sequence.

The transduction can always be performed by a pushdown machine. In fact, the

same pushdown machine can be used for all context-free languages. As the sequence comes in it is stored on the pushdown stack, and after it is completed the now reversed sequence is popped off the stack and generated as output. This is a proper transduction; the machine is certainly not recognizing the language. Indeed the transduction can be performed on languages that cannot be recognized by pushdown machines.

Moreover the transduction is not in general simple nor simple Polish. The question may arise, therefore, as to how to apply Theorem 5, which states that any transduction performed by a pushdown machine can be represented as a simple Polish SD translation on an LR(k) grammar. Since the transducer does not reject any sequences, the language actually being *translated* by the machine is the finite-state language consisting of all sequences in the input vocabulary of the machine. Suppose this vocabulary is {0, 1}; then an LR(k) grammar for this language together with an appropriate simple Polish translation is given by

$$S \to 0S \qquad \{S0\}$$
$$S \to 1S \qquad \{S1\}$$
$$S \to \epsilon \qquad \{\epsilon\}$$

B. Transduction Harder Than Translation and Parsing. Consider the LR(0) grammar for a portion of arithmetic together with the simple prefix Polish transduction elements:

$$S \to (S + S) \qquad \{+S\ S\}$$
$$S \to (S * S) \qquad \{*S\ S\}$$
$$S \to 1 \qquad \{1\}$$
$$S \to 2 \qquad \{2\}$$

The transduction output is a simple prefix Polish representation of the input arithmetic expression.

Although this language can be recognized and parsed according to the given grammar by a pushdown machine, the given transduction cannot be performed by a pushdown machine. To verify this, consider the sequences of the form $((a_1 + \cdots + a_n) * 2)$ for a_i in {1, 2}. The transduction is supposed to begin with a *, but this cannot be determined until after all the a_i have been read in. After the *, the pushdown machine must put out a sequence which includes the symbols a_i in their given order. But these symbols cannot be reproduced in that order, as the pushdown machine could then be modified to recognize the subset of $\{1, 2, m\}^*$ given by $\{wmw \mid w \text{ in } \{1, 2\}^*\}$, contrary to the well-known fact that this set is not context-free.

APPENDIX I. Recognition of LL(k) Languages

The purpose of this appendix is to describe in detail how a deterministic pushdown machine can be constructed to recognize the language generated by a given LL(k) grammar. Other properties of LL(k) grammars are mentioned at the end.

Let L be the language generated from LL(k) grammar G using nonterminals N

and terminals T. Define L_A as follows: For A in N, let L_A be the set of words in T^* generated by G using starting symbol A; for A in T, let $L_A = \{A\}$. If R is a subset of T^*, let $R/k = \{w/k \mid w \text{ in } R\}$.

In order to construct the pushdown machine, we need to establish the following special property of LL(k) grammars.

LEMMA. *If G is LL(k), then for all A in N, w in T^*/k, and $R \subseteq T^*/k$ satisfying $R \subseteq R(w_1) = \{w_3/k \mid S \Rightarrow w_1Aw_3\}$ for some w_1 in T^*, there exists at most one production p such that $A \Rightarrow w_2$ (p) and $(w_2w_3)/k = w$ for some w_2 and w_3 in T^* such that w_3 is in R.*

PROOF. Suppose that $A \rightarrow w_2$ (p), $A \rightarrow \bar{w}_2$ $(\bar{p})$, and $(w_2w_3)/k = (\bar{w}_2\bar{w}_3)/k = w$ for some w_2, $\bar{w}_2$, w_3, $\bar{w}_3$, p, and $\bar{p}$ such that w_3/k and $\bar{w}_3/k$ are in R where $R \subseteq R(w_1)$ for some w_1. It follows that there must be w_3' and $\bar{w}_3'$ in T^* such that $S \Rightarrow w_1Aw_3'$, $S \Rightarrow w_1A\bar{w}_3'$, $w_3'/k = w_3/k$, and $\bar{w}_3'/k = \bar{w}_3/k$. The last two relations imply that $(w_2w_3')/k = (\bar{w}_2\bar{w}_3')/k = w$, and the fact that $p = \bar{p}$ follows from the definition of LL(k). Thus there is at most one such p.

The importance of this lemma can be stated informally as follows: The definition of LL(k) grammars specifies that for any sequence in the language, a production can be correctly identified from the sequence w_1 of symbols to the left of its first descendant and the k symbols which follow. The lemma states that the production can also be identified using only the k symbols which follow and the set $R(w_1)$ where $R(w_1)$ is the set of all k symbol sequences which can follow the rightmost descendant of that production. In other words, we can bound the amount of information that must be remembered about the initial sequence.

DESCRIPTION OF THE PUSHDOWN MACHINE. The finite-state control has enough memory to store an input string of length k or less and to perform such obvious tasks as reading in the first k inputs. The tape symbols are ordered pairs (A, R), where A is an element of $N \cup T$ and R is a subset of T^*/k. We design the machine so that if some $r + k$ inputs have been read in, the input string stored in the finite-state control is w, and the top tape symbol is (A, R), then the following are true.

(1) The word w stored in the finite-state control is the string consisting of the $(r + 1)$-th input through the $(r + k)$-th input. If the input word only has $r + k'$ symbols for $k' < k$, then w is the last k' symbols of the input word. In this latter case, it is convenient to say that $k - k'$ blank inputs have been read in after the completion of the input word as indicated by the special end-of-tape marker. These implicit blanks play the same role as the " $\dashv$" of Knuth [6].

(2) The symbol A represents the fact that the descendants of an A follow the rth input symbol. If A is a terminal symbol, this means that the $(r + 1)$-th input symbol must be an A. The symbol pair (A, R) or its replacement is to be popped up as soon as all the descendants of A have been identified.

(3) The set R represents R'/k, where R' is the set of all acceptable input sequences that could follow the descendants of the A. Thus, if (A_1, R_1) is the tape symbol below (A, R), then $R = (L_{A_1} \cdot R_1)/k$; and $R = \{\epsilon\}$ if (A, R) is the bottom tape symbol. The machine begins with the single symbol $(S, \{\epsilon\})$ on its pushdown tape.

We now describe the machine operations. Initially, the machine reads the first k inputs and stores them as the word w in the finite control. The pushdown tape is initialized with the symbol $(S, \{\epsilon\})$. This initialized configuration satisfies 1, 2, and

3 above and we take it as self-evident that the operations described below preserve these properties. After $r + k$ inputs have been read, the operations are as follows:

Case 1. If the top tape symbol is (A, R) and A is a nonterminal, then $R = \{w_3/k \mid S \Rightarrow w_1Aw_3\}$, where w_1 consists of the first r inputs. Therefore, by the lemma, there is at most one production p that could be applied to A in order to be consistent with w and R. Three subcases follow:

Case 1a. If there is no such p, the machine rejects the sequence.

Case 1b. If p is the production $A \rightarrow \epsilon$, then the top tape symbol is popped off.

Case 1c. If p has the form $A \rightarrow A_1 \cdots A_n$ for A_i in $N \cup T$, then the top symbol (A, R) is replaced by the sequence of symbols $(A_1, R_1) \cdots (A_n, R_n)$, where $R_n = R$ and $R_{i-1} = (L_{A_i} \cdot R_i)/k$ for $1 < i \leq n$.

Case 2. If the top tape symbol is (A, R) and A is a terminal, there are two subcases:

Case 2a. If $w = Aw'$ for some w', then the top tape symbol is popped off, the next input x is read, and word $w'x$ replaces w in the finite control.

Case 2b. If w does not begin with A, then the sequence is rejected.

Case 3. If there are no tape symbols on the pushdown tape, then if $w = \epsilon$, the sequence is accepted and otherwise it is rejected.

Although considerable information is encoded in the tape symbols (A, R), this is somewhat less information than is required for general LR(k) recognition. Furthermore, even this R information is needed only when the machine must choose among words in L_A which are shorter than k. To verify this, assume that (A, R) is the top tape symbol (i.e. the machine is looking for a production descendant from A) and that control word w is in L_A (i.e. that the next k symbols after the start of A are all descendants of A). Then it follows from the definition of context-free grammars that the past can give no information as to which A production was used, and hence the decision is independent of R. This situation always occurs for LL(1) grammars if there are no ϵ productions.

As an example where R information is necessary, consider the following grammar:

$$S \rightarrow 1A1B$$

$$S \rightarrow 0A0B$$

$$A \rightarrow 0$$

$$A \rightarrow 01$$

$$B \rightarrow 0B$$

$$B \rightarrow 0$$

This grammar is LL(3), but after $1 + 3$ inputs have been read and $w = 010$, one cannot determine which production to apply to A without consulting the corresponding R which will contain either $\{10\}$ or $\{00\}$, depending on which production was applied to S.

As with LR(k) grammars, there is no algorithm for determining whether or not there exists a k such that a given grammar is LL(k). Given a k, however, one can determine if the given grammar is LL(k) for that k. Proofs of these statements are omitted here.

APPENDIX II. Canonical Grammar and Simple Polish

Given a pushdown translator with input set T, endmarker $\#$, state set $\{s, t, \cdots\}$, tape symbol set $\{A, B, \cdots\}$, and output set $\{w, \cdots\}$, a simple Polish description can be obtained as follows:

Terminals. The terminal set is just the input set T.

Nonterminals. For states s and t, tape symbol A, and a in $T \cup \{\epsilon, \#\}$, we define nonterminals A_s, $A_{s,t}$, $\varphi_{s,A,a}$. Symbol A_s may be interpreted to mean that the machine is in state s with top symbol A and A does not get popped up. Symbol $A_{s,t}$ means that the machine is in state s, the top tape symbol is A, and the machine enters state t when A is eventually popped up. Symbol $\varphi_{s,A,a}$ means that the machine is in state s, the top tape symbol is A, and "input" a is received. The case $a = \epsilon$ represents the case of the machine operating without reading a new input and $a = \#$ represents the case of the machine reading the endmarker. Since this endmarker is not considered part of the input alphabet, it must be treated separately.

Productions. Suppose that when in state s with top tape symbol A, the input a produces output w.

(1) If the machine pushes a B onto the stack and enters state t, then for all states x and y, we write the productions

$$\begin{aligned}
A_s &\to \varphi_{s,A,a}B_{t,x}A_x && \{\varphi_{s,A,a}B_{t,x}A_x\},\\
A_s &\to \varphi_{s,A,a}B_t && \{\varphi_{s,A,a}B_t\},\\
A_{s,y} &\to \varphi_{s,A,a}B_{t,x}A_{x,y} && \{\varphi_{s,A,a}B_{t,x}A_{x,y}\},\\
\varphi_{s,A,a} &\to a && \{w\} \quad \text{if} \quad a \neq \#,\\
\varphi_{s,A,\#} &\to \epsilon && \{w\} \quad \text{if} \quad a = \#.
\end{aligned}$$

(2) If the machine changes to state t, then for all states x, we write the productions

$$\begin{aligned}
A_s &\to \varphi_{s,A,a}A_t && \{\varphi_{s,A,a}A_t\},\\
A_{s,x} &\to \varphi_{s,A,a}A_{t,x} && \{\varphi_{s,A,a}A_{t,x}\},\\
\varphi_{s,A,a} &\to a && \{w\} \quad \text{if} \quad a \neq \#,\\
\varphi_{s,A,\#} &\to \epsilon && \{w\} \quad \text{if} \quad a = \#.
\end{aligned}$$

(3) If the machine changes from state s to state t and pops the symbol A, then we write the production

$$\begin{aligned}
A_{s,t} &\to a && \{w\} \quad \text{if} \quad a \neq \#,\\
A_{s,t} &\to \epsilon && \{w\} \quad \text{if} \quad a = \#.
\end{aligned}$$

(4) If state s with tape symbol A is an accepting state for the machine, we write the production

$$A_s \to \epsilon \qquad \{\epsilon\}.$$

Starting Symbol. The starting symbol is Z_s, where Z is the bottom tape symbol and s is the starting state.

It can be assumed without loss of generality that the pushdown machine must

scan all the input symbols before stopping in an accepting state. A little reflection now shows that the above grammar is indeed a grammar for the language recognized by the pushdown machine. Given a sequence of machine operations leading to an accepting state, this history of operations defines a derivation. For example, if a symbol is pushed down, the history reveals whether this symbol was popped up and if so, which state the machine entered. The interpretation of the nonterminals then reveals which production under (1) above should be used. The other cases are even easier. Conversely, given a derivation in this grammar, the symbols describe a series of machine operations that recognize the derived word. When the machine is deterministic, each word in the language has a unique derivation, namely the one corresponding to the unique series of machine operations for that input word.

In the case of *recognition* with pushdown machines, it is known that the presence or absence of endmarkers does not affect the class of languages that can be recognized. Recognition without endmarkers is defined in terms of machines that enter accepting states if and only if the inputs read up to a given point are in the language. However, in the case of *transduction* or *translation*, endmarkers do add power even for finite-state machines as illustrated by

$$S \rightarrow \epsilon \{0\}$$

$$S \rightarrow 1 \{1\}$$

which can only be performed using an endmarker.

We now sketch a proof that, in the case of a deterministic machine, this grammar is LR(1). We do this by showing how each type of production can be identified. If, in processing a word from this language, all the descendants of an $A_{s,t}$ have been seen, this is detected by the fact that the machine was in state s when A was put on the tape and then state t was entered when A was finally popped off. If A was not popped off, then the rightmost descendant of an $A_{s,t}$ has not been seen. Nonterminal $\varphi_{s,A,a}$ can be identified by a machine operation which does not pop the pushdown tape. Nonterminals A_s only appear on the rightmost branch of a derivation, and this can be verified by looking one ahead to see if the input word has ended. This same proof is essentially given in Knuth [6] with more detail.

The transduction elements are in simple Polish form and obviously give the transduction defined by the machine, since the outputs are attached directly to the (possibly null) inputs that produce them.

ACKNOWLEDGMENT. The authors are grateful to D. E. Knuth for his detailed criticisms, which include a new proof of Theorem 9 and numerous suggestions for improving the clarity of the presentation.

REFERENCES

1. IRONS, E. T. A syntax directed compiler for ALGOL 60. *Comm. ACM 4*, 1 (Jan. 1961), 51–55.
2. PETRONE, LUIGI. Syntactic mappings of context-free languages. Proc. IFIP Congress, 1965, Vol. 2, pp. 590–591.
3. CULIK, K. Well-translatable grammars and ALGOL-like languages. In Steel, T. B., Jr. (Ed.), Proc. IFIP Working Conference on Formal Language Description Languages, North-Holland, Amsterdam, 1966, pp. 76–85.
4. KOPRIVA, JIRI. Realization of generalized good translation algorithm on computer. *Kybernetika Cislo 5*, Rocnik 3 (1967), 419–429.

5. YOUNGER, D. H. Recognition and parsing of context-free languages in time n^3. *Inform. Contr. 10*, 2 (Feb. 1967), 189–208.
6. KNUTH, D. E. On the translation of languages from left to right. *Inform. Contr. 8*, 6 (Dec. 1965), 607–639.
7. OETTINGER, A. Automatic syntactic analysis and the pushdown store. *In* Jacobson, R. (Ed.), Structure of Language and Its Mathematical Concepts, Proc. 12th Symposium in Appl. Math., Amer. Math. Soc., Providence, R. I., 1961, pp. 104–129.
8. CHOMSKY, N. Context-free grammars and pushdown storage. RLE Quart. Progr. Rep. No. 65, Research Lab. of Electronics, M.I.T., Cambridge, Mass., March 1962.
9. HARTMANIS, J., AND STEARNS, R. E. On the computational complexity of algorithms. *Trans. Amer. Math. Soc. 117*, 5 (May 1965), 285–306.

RECEIVED SEPTEMBER, 1966; REVISED APRIL, 1967