Stack Machines and Classes of Nonnested Macro Languages

JOOST ENGELFRIET

Twente University of Technology, Enschede, The Netherlands

ERIK MEINECHE SCHMIDT

Cornell University, Ithaca, New York

AND

JAN VAN LEEUWEN

The Pennsylvania State University, University Park, Pennsylvania

ABSTRACT. A new class of generalized one-way stack automata, called s-pd machines, is investigated The machines are obtained by augmenting a stack automaton with a pushdown store, whose bottom is attached to the top of the stack and whose top follows the movements of the stack-pointer into the stack. Motivations for the model include a possible protocol for macro expansion with intermittent parameter evaluation. The languages recognized by these machines are characterized by a natural class of grammars, viz, the class of OI macro grammars with set-parameters and nonnested function calls (the "extended basic" or EB macro grammars) If the stack is required to be nonerasing or checking, then a useful machine characterization for the ETOL languages is obtained, together with the known characterization of this family by means of extended "linear" basic or ELB macro grammars. It follows that the nonerasing one-way stack languages are (strictly) included in ETOL. It is proved that the family of unrestricted one-way stack languages and ETOL are incomparable, as are the general OI macro languages and the yields of ranges of topdown tree transducers. It follows that ETOL is strictly included in the family of EB macro languages (which, in turn, is strictly included in the family of indexed languages) Certain deterministic restrictions of s-pd machines lead to machine models for families of nonextended macro languages, viz, for Fischer's original linear basic macro (i e, EDTOL) and basic macro languages.

KEY WORDS AND PHRASES. macro expansion, macro grammars, stack automata, ETOL languages, language classification, machine characterization

CR CATEGORIES: 5.22, 5.23

1. *Introduction*

The theoretical models of stack automata have a traditional motivation from the compilation of higher level programming languages [18, 19] and the implementation of recursive procedures with parameters. A (one-way) stack automaton is best described as a pushdown automaton which may enter its store in a read-only mode. Writing and erasing can still occur at the top of the stack only. Stack languages have been the subject of various studies in the past [18, 19, 21], and much is known about their complexity (see, e.g., [26]) and possible syntactic characterization [9, 24].

In 1968 Aho [2] proposed a generalized stack automaton (the "nested stack" automaton)

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission

Authors' present addresses: J Engelfriet, Department of Applied Mathematics, Twente University of Technology, P O Box 217, 7500 AE Enschede, The Netherlands, E Meineche Schmidt, Department of Computer Science, Aarhus University, Ny Munkegade, DK-8000 Aarhus C, Denmark, J. van Leeuwen, Department of Computer Science, University of Utrecht, Budapestlaan 6, P O Box 80 012, 3508 TA Utrecht, the Netherlands

© 1980 ACM 0004-5411/80/0100-0096 $00.75

which permits the creation of embedded ("nested") stacks within an old stack, with the convention that an embedded stack must be destroyed if the machine is to raise the stack-pointer back up to the containing stack. The model was designed specifically for implementing the grammatical mechanism of indexed languages (Aho [1]) and is rather complex. About the same time Fischer [16] presented a detailed study of a language generating mechanism inspired by the use of recursive macros in assembly language programming. In this context a macro is viewed as a string generating function with symbolic parameters, with a macro body consisting of several possible strings containing new, perhaps nested, macro calls. A macro expansion (or derivation) is obtained by replacing all pertinent macro calls by a string from their corresponding macro body (after suitably passing the actual parameters) until no further macro calls are generated.

The prime motivation for introducing macro grammars (as well as indexed grammars) was their power to describe context-dependent features in the syntax of various programming languages. We shall only consider the OI ("outside-in") macro grammars, which always evaluate outermost calls first as in the call-by-name parameter passing mechanism. The relation between OI and IO ("inside-out") macro grammars was carefully analyzed in [13, 16]. Fischer [16] originally proved that the family of OI macro languages coincides with the family of indexed languages, thus providing a further practical motivation for the latter family. Fischer [16, Sec. 7] observed as a corollary that OI macro grammars which permit nested macro calls must generate a strictly larger class of languages than the macro grammars which do not permit such calls (as there is no difference between OI and IO for the latter). It is one of the main objectives of this paper to investigate precisely the generative capacity of macro grammars without nested calls (the "basic" macro grammars) and to present a class of simple stack machines which naturally corresponds to the class of languages generated. The results suggest an interesting protocol for the expansion of macros with intermittent parameter evaluations, and a natural machine model is obtained for symbolically evaluating nondeterministic recursive program schemes with parameters but no nesting of recursion within the parameters.

The study of basic and linear basic macro grammars was initiated in [16]. Downey [4] observed that such grammars tie in appropriately with the study of parallel rewriting systems (as in [25]) once we permit macros to have set-parameters. A set-parameter can be manipulated like any other symbolic parameter, but in addition one may use the constant $\emptyset$ (which denotes the empty set) and the operation of union (denoted by +) in the parameter positions of further macro calls in a defining body. In such "extended" basic macro grammars we assume that all symbolic parameters are actually set-parameters. Note that a set-parameter always denotes some finite set in a particular derivation. It was proved in [4] that the family of linear basic macro languages (LB) coincides with EDT0L and that the family of extended linear basic macro languages (ELB) corresponds exactly to the family ET0L. (For EDT0L and ET0L, see [25].) In this paper we consider the general family of extended basic macro languages (EB) and establish the relation of this family to several classes of stack languages. A practical motivation to study EB is that the weak nesting capacity in EB grammars (due to the presence of +) seems to be sufficient to describe some context-dependent features of the syntax of programming languages only described by general OI macro grammars until now.

The machine approach in studying EB macro grammars was inspired by a machine characterization of ET0L, i.e., the ELB macro languages, presented in [37] using cs-pd machines. The original model of the cs-pd machine as given in [37] (see Figure 1(a)) had a checking stack (left) and a pushdown store (right), with the top of the pushdown store forced to follow the same moves as the stack-pointer. (For the notion of a checking stack, see [21].) The machine emerged from a theoretical model for studying Dijkstra's DO-construct [5, 6] as a single control structure in programming, and was used to obtain a uniform characterization of certain hierarchies of complexity classes. It can be shown (by simulating a Post machine [28]) that a similar machine model with a nonerasing stack

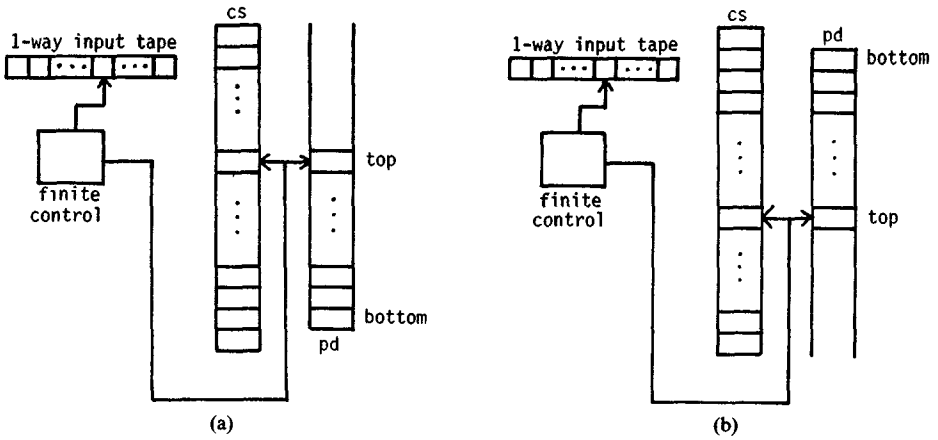


FIG. 1 The (a) old and (b) new versions of the cs-pd machine model

instead of a checking stack can accept all recursively enumerable languages. Hence there seems to be no way of meaningfully generalizing the model in this direction. If we turn the pushdown store "upside down" (Figure 1(b)) and attach its bottom to the top of the checking stack, then we obtain a natural, equivalent model which can be generalized. With this alternative description it has also become easier to see how the machine is a (very) restricted version of the nested stack automaton, allowing "nested stacks" of size 1 only.

A main objective of this paper will be to study the s-pd machine model. The machine works as the cs-pd model except that it now uses a general, unrestricted stack rather than just a checking stack. Note that in an s-pd machine the position of the bottom of the pushdown store changes dynamically with the movements of the top of the stack. We require as before that the pushdown-pointer move in parallel with the stack-pointer whenever the machine enters stack-reading mode (with the pushdown growing "downward"). We shall prove that the s-pd machines accept precisely the EB languages, generated by EB macro grammars.

Using these machines, Filè and van Leeuwen (see [15]) have recently obtained elegant characterizations of ETOL and EB by classes of indexed grammars. It turns out that EB is precisely the family of languages generated by "restricted indexed grammars" [1, 2], which were originally unidentified in terms of macro grammars. From the machine characterization it follows also that the nonerasing one-way stack languages are included in ETOL [37] and that the general one-way stack languages (S) are in EB (where obviously ETOL $\subseteq$ EB). We prove that ETOL and S are incomparable families by exhibiting a specific language which is in the latter family but not in the former. The result shows at the same time that ETOL is strictly included in EB, a substantial refinement of an earlier result of Ehrenfeucht, Rozenberg, and Skyum [8] asserting that ETOL is strictly included in the family of indexed languages. In other words, the known result that ETOL is strictly included in the family of OI macro languages is strengthened here to strict inclusion in the family of nonnested OI macro languages with set-parameters. (See [12] for indications that EB is a rather narrow strict subfamily of the OI macro languages.) The relationships are summarized in Figure 2.

If we divide the allowable operations on a stack into top operations (push, pop) and interior operations (movedown, moveup), then we can observe the following from Figure 2. The incomparability of ETOL with S shows that it is apparently impossible to separate top operations from interior operations by making two separate tracks in the stack, one of which is used as a pushdown store and one as a read-only tape. This holds even when pushing is allowed on the second track (i.e., NES-PD = CS-PD). The incomparability of CF with NES shows that the top operations are incomparable with the interior operations

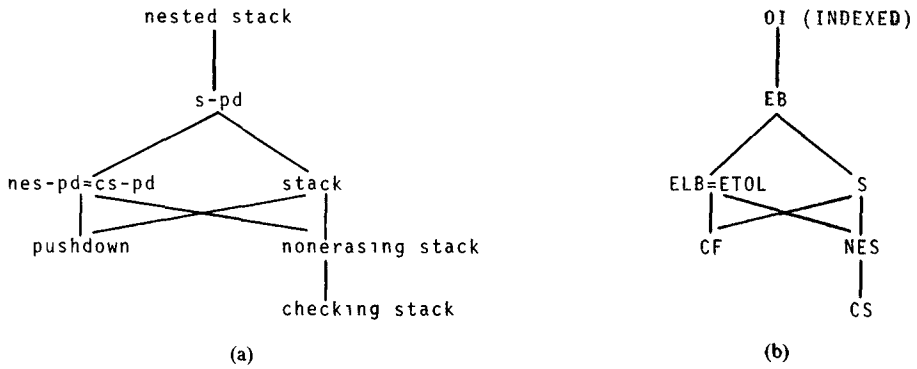


FIG 2 The relation between machine models and language families (a) storage structure, (b) corresponding language families (where possible by grammar name) Solid lines indicate proper inclusion

(even when "push" is added to the latter). The results together illustrate the power of the pop operation in stack machines.

The remaining part of this paper consists of Sections 2 to 5 and a conclusion. Section 2 contains the necessary definitions and some preliminary results. In Section 3 we exhibit a particular stack language that is not an ETOL language (or even a tree transformation language). Section 4 contains a proof of the equality EB = S-PD (where S-PD denotes the family of languages accepted by s-pd machines) and several related characterizations. In Section 5 we put certain deterministic restrictions on the s-pd machine and obtain generating machine models for the basic and linear basic macro languages. In Section 5 we also give a complete inclusion diagram relating the many families of languages discussed in the paper.

2. *Preliminaries*

The terminology and notations used in this paper largely follow standard texts in automata theory and formal language theory [26, 35]. The reader is assumed to have some acquaintance with AFL theory [17, 20], stack automata (see, e g., [26]), and the theory of parallel rewriting [25], and we do not redefine the usual concepts from these areas here.

We denote the empty word by λ and the length of a word w by $|w|$.

An *OI macro grammar* G consists of an alphabet N of nonterminals (macro names, each with a specified number of arguments or rank), a set $\{x_1, x_2, \ldots\}$ of formal parameter names (variables), an alphabet Σ (terminal symbols), an initial nonterminal S (initial macro, of rank 0), and a finite set P containing the production rules (i.e., macro definitions). A formal definition is given in [13, 16]. Each macro definition is a rule of the form $F(x_1, \ldots, x_n) \rightarrow \Theta$, where F is a macro name of some rank n in N and Θ is a well-formed term composed of variables in $\{x_1, \ldots, x_n\}$, terminals, and macro names by substitution and concatenation. Formally, a term is either (i) an atomic term, i.e., an element of $\{x_1, x_2, \ldots\} \cup \Sigma \cup \{\lambda\}$, (ii) of the form $H(t_1, \ldots, t_m)$ where H is a macro name of some rank m and $t_1, \ldots, t_m$ are terms, or (iii) of the form $t_1 t_2$ where t_1 and t_2 are terms. Macros are expanded with outermost calls first, in the usual OI manner. The collection of all words over Σ generated by G is called an *OI macro language.*

In *basic macro grammars* no term Θ in a macro definition is allowed to have macro calls within the parameters of other macro calls, i.e , they are "nonnesting"; in *linear basic macro grammars* each term Θ can only have at most one macro call. The classes of OI, basic, and linear basic macro languages are denoted by OI, B, and LB, respectively. An *extended macro grammar* [4] permits the use of $\varnothing$ (denoting the empty set) and finite unions (denoted by +) in the macro definitions. Parameters become set-parameters, which can denote arbitrary finite sets of string-values (as opposed to singletons) during derivations.

Formally, an extended macro grammar is obtained by modifying the definition of terms as follows: (i') each element of $\{x_1, x_2, \ldots\} \cup \Sigma \cup \{\lambda, \varnothing\}$ is an atomic term, and (iii') if t_1 and t_2 are terms, then so are $(t_1 + t_2)$ and $t_1 t_2$. Instead of formalizing a separate notion of derivation for these extended grammars, we view them as ordinary OI macro grammars in which + is taken as a macro of rank 2 (usually written infix) with rules $+(x, y) \to x$ and $+(x, y) \to y$, and $\varnothing$ as a macro of rank 0 without rules.

Although the extension by set-parameters does not increase the generating power of general OI macro grammars, it does increase the power of nonnesting macro grammars. We define the *extended basic* (*EB*) and the *extended linear basic* (*ELB*) macro grammars to be those extended macro grammars which are basic and linear basic, respectively, when viewed as ordinary macro grammars with "terminals" + and $\varnothing$.

Example 2.1. In the description of the syntax of a programming language the finite sets in the arguments of the nonterminals can be used by an EB grammar to store the declared identifiers of a same type. This can be seen from the following ELB grammar for the language $\{u_1 \# u_2 \# \cdots \# u_n \# u \mid n \geq 1 \text{ and } u \in \{u_1, \ldots, u_n\}\}$:

$$\begin{aligned} &S \to F(\varnothing), \\ &F(x) \to G(x, \lambda), \\ &F(x) \to x, \\ &G(x, y) \to aG(x, ya) \qquad \text{for all} \quad a \in \Sigma, \\ &G(x, y) \to \#F(x + y). \end{aligned}$$

A derivation of the string $ab\#a\#ab$ is

$$\begin{aligned} S &\Rightarrow F(\varnothing) \Rightarrow G(\varnothing, \lambda) \Rightarrow aG(\varnothing, a) \Rightarrow abG(\varnothing, ab) \\ &\Rightarrow ab\#F(\varnothing + ab) \Rightarrow ab\#G(\varnothing + ab, \lambda) \\ &\Rightarrow ab\#aG(\varnothing + ab, a) \Rightarrow ab\#a\#F((\varnothing + ab) + a) \\ &\Rightarrow ab\#a\#((\varnothing + ab) + a) \Rightarrow ab\#a\#(\varnothing + ab) \\ &\Rightarrow ab\#a\#ab. \end{aligned}$$

□

A further extension of macro grammars to *regular extended basic* (*REB*) and *regular extended linear basic* (*RELB*) macro grammars is obtained if we permit arbitrary regular expressions over $\{x_1, \ldots, x_n\} \cup \Sigma$ (involving +, ·, and *) to occur in macro definitions. Formally, t^* is now allowed as a term in EB and ELB macro grammars also (provided t is), and the definition of a derivation is modified again by viewing * as a nonterminal A of rank 1 with rules $A(x) \to xA(x)$ and $A(x) \to \lambda$. This further extension does not increase the generative capacity of extended macro grammars, but there will be technical advantages in using it in later proofs. The following lemma shows that one can always eliminate the * and reformulate REB and RELB macro grammars as ordinary extended macro grammars, without introducing any nesting in the parameters

LEMMA 2.2. (*i*) *REB* = *EB*. (*ii*) *RELB* = *ELB*.

PROOF. We shall only prove (ii), as (i) follows in a similar manner Consider a macro definition from an arbitrary RELB grammar

$$F(x_1, \ldots, x_n) \to \Theta_1 G(E_1, \ldots, E_m)\Theta_2,$$

where Θ_1, Θ_2, $E_1, \ldots, E_m$ are regular expressions over $\{x_1, \ldots, x_n\} \cup \Sigma$. We may assume without loss of generality that $\Theta_1 = \Theta_2 = \lambda$ (compare the construction in the beginning of the proof of Lemma 4.1). We shall replace the macro definition by an equivalent set of ELB macro definitions, which can generate any finite approximation to $E_1, \ldots, E_m$ in the respective parameter positions. Since in the derivation of each string in the language only a finite number of strings from the set-arguments of the macros are really used, the resulting grammar generates the same language. (This can be derived formally from the fixed point characterization of the grammars and the continuity of the operation of language substitution [13].)

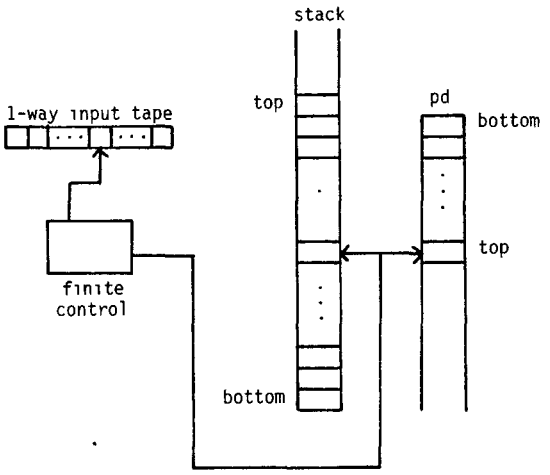


FIG 3 The s-pd machine model

Let $M_i = \langle Q_i, \{x_1, \ldots, x_n\} \cup \Sigma, q_0, \delta_i, F_i \rangle$ be a deterministic finite automaton defining E_i, for $1 \leq i \leq m$. Replace the former RELB macro definition of F by

$$F(x_1, \ldots, x_n) \to G_1^{q_0}(\lambda, x_1, \ldots, x_n, \underbrace{\varnothing, \ldots, \varnothing}_{m \text{ copies}})$$

and define new ELB macros $G_i^p(y, x_1, \ldots, x_n, y_1, \ldots, y_m)$, $p \in Q_i$ and $1 \leq i \leq m$, as given below. The macros G_i^p will be called after completion of approximations $y_1, \ldots, y_{i-1}$ for $E_1, \ldots, E_{i-1}$, with y_i a current approximation (a finite subset) to E_i and y a partially completed new member of E_i which will eventually be added to y_i.

$$G_i^p(y, x_1, \ldots, x_n, y_1, \ldots, y_m) \to \begin{cases} G_i^q(ya, x_1, \ldots, x_n, y_1, \ldots, y_m) \\ \quad \text{for all } a \in \Sigma, \quad \text{with } \delta_i(p, a) = q, \\ G_i^q(yx_k, x_1, \ldots, x_n, y_1, \ldots, y_m) \\ \quad \text{for all } x_k, \quad \text{with } \delta_i(p, x_k) = q, \\ G_i^{q_0}(\lambda, x_1, \ldots, x_n, y_1, \ldots, y_i + y, \ldots, y_m) \\ \quad \text{if } p \in F_i, \\ G_{i+1}^{q_0}(\lambda, x_1, \ldots, x_n, y_1, \ldots, y_i + y, \ldots, y_m) \\ \quad \text{if } p \in F_i \quad \text{and} \quad i < m, \\ G(y_1, \ldots, y_{m-1}, y_m + y) \\ \quad \text{if } i = m \quad \text{and} \quad p \in F_m. \end{cases}$$

□

We now turn to machines. The words "machine" and "automaton" will be used synonymously. The model of a cs-pd machine was motivated and defined in [37]. In our present treatment we assume that the pushdown store is actually initiated at the top of the checking stack rather than at its bottom (as in [37]), but the constraint that the stack-pointers move up and down simultaneously (in parallel) remains in effect.

The model of an *s-pd machine* (see Figure 3) is obtained from an ordinary (one-way) stack automaton by adding a pushdown store again, with its bottom rooted at the top of the stack and its top-pointer following the movements of the stack-pointer whenever the latter makes a read-excursion into the stack. As the pushdown becomes "active" only when a read-excursion begins, we assume that its first (bottom) square actually begins one square below the stack top (see Figure 3). The forced coupling of pointers implies that the machine can change the contents of the stack only when the pushdown is empty. Conversely, when the pushdown is active the machine can only read its stack and not alter it

The basic model of the s-pd machine is nondeterministic, with acceptance by final state

(as usual). It always begins with empty stack and empty pushdown. The stack-pointer will always be located at the highest nonempty stack-square (rather than at the first empty square on the stack), unless it moves "inside" the stack on a read-excursion. A similar convention is made for the pushdown-pointer. Particular s-pd machines will be specified by writing (nondeterministic) programs in a symbolic language of instructions, tests, and standard identifiers which can be implemented in a straightforward manner (see Section 4). The definition of s-pd machines as an x-tuple would not add to the understanding of the model and is left as an exercise to the reader.

By restricting the stack to be nonerasing or checking, respectively, we obtain the *nes-pd* and *cs-pd machine* models. A program for an nes-pd machine is not allowed to use pop-instructions for the stack. A program for a cs-pd machine must execute some number of push-instructions for the stack first (filling it up as a checking stack), but after completing this phase it is not allowed to use any push- or pop-instructions at the top of the stack ever again for the remainder of the computation (i.e., the stack-contents are fixed for "checking"). The one-way stack, one-way nonerasing stack, and one-way checking stack automata [18, 19, 21] are readily obtained by dropping the pushdown facility from the extended machines. Unless stated otherwise, we assume from now on that all machines considered are one-way. We use capital letters to denote the class of languages accepted by machines whose "type" is written in equivalent small letters. In particular, S denotes the family of ordinary (one-way) stack languages.

Since the cs machine is less powerful than the nes machine [21], it is remarkable that cs-pd machines are just as powerful as nes-pd machines (cf. also [37, Th. 2.4]).

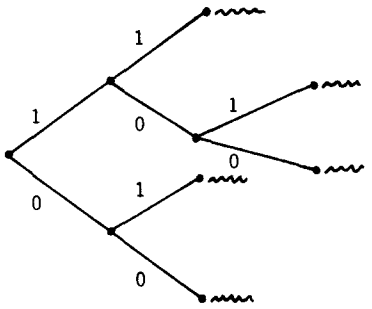
THEOREM 2.3. *NES-PD = CS-PD.*

PROOF. Obviously CS-PD $\subseteq$ NES-PD. To prove the converse, we show a direct simulation of an nes-pd machine M on a cs-pd machine M_1. Recall that an nes-pd machine M alternates pushing on its stack with read-excursions, with the stack readily incrementing. The simulating machine M_1 nondeterministically fills its checking stack to some height and tries to interpret it as the ultimate contents of the nonerasing stack in an accepting computation of M on the input, as it proceeds. After filling its checking stack, M_1 returns to the bottom of the stack to begin a simulation of M while filling up the pushdown with dummy $\not{c}$'s. Simulating M, M_1 now verifies that its stack contains the symbols M would have written (meanwhile popping $\not{c}$'s off the pushdown) until M wants to stop pushing. If M is about to enter its stack for a read-excursion, M_1 marks the current pushdown top with a \$ and uses the part of the pushdown from the \$-marked square on downward to simulate M's instantaneous pushdown store. M_1 "knows" when M returns to the "current" top of its nonerasing stack, because the simulation will simultaneously return to the \$-marked square on the pushdown store. If M continues pushing, then M_1 removes the marker and verifies the next symbols on its stack. This simulation of pushing and read-excursions repeats until M stops. □

Finally we give a brief description of ET0L grammars (see [25, 33] for more details). An *ET0L grammar* [33] is a structure $G = \langle V, \Sigma, \{\tau_1, \ldots, \tau_n\}, S\rangle$ with V an alphabet, Σ a set of terminal symbols ($\Sigma \subseteq V$), S a start symbol ($S \in V$), and $\tau_1, \ldots, \tau_n$ finite substitutions over V. The language generated by G is defined to be $L(G) = \{\tau_1, \ldots, \tau_n\}^*(S) \cap \Sigma^*$. Any such language is called an *ET0L language*. If the $\tau_1, \ldots, \tau_n$ are homomorphisms, then the resulting language is called an *EDT0L language*. The relevance of ET0L and EDT0L languages for this paper follows from the equalities ET0L = CS-PD [37], ET0L = ELB [4], and EDT0L = LB. An alternative proof of the first two equalities is included in Section 4.

3. *ET0L and One-Way Stack Automata*

It follows immediately from Theorem 2.3 and the equality ET0L = CS-PD [37] that the nonerasing stack languages are included in ET0L [37]. (Strict inclusion follows because



(a)

⟨00,01,100,101,11⟩

(b)

FIG 4 (a) The infinite binary tree, (b) a cut

$\{a^n b^{n^2} c^n \mid n \geq 1\}$ is in ETOL but not in S [30].) In other words, each nonerasing stack language can be defined by an ELB macro grammar. We prove in this section that this is not true for all one-way stack languages (cf. [12]). In Section 4 it will be shown that each stack language can be defined by an EB macro grammar.

We need some additional preliminaries. Let L be a language over the alphabet Σ. We say that L has "property P_3" [14] if and only if for all $x, u, y, v, z \in \Sigma^*$: $xuyvz$, $xuyuz$, $xvyuz$, and $xvyvz \in L$ implies $u = v$. Property P_3 states that there can be no two different, nonoverlapping substrings of a string in L which may replace one another without leaving L. If L were defined by a "nondeterministic" grammar, then having property P_3 intuitively means that there can be no two occurrences of the same "nondeterministic" nonterminal in a sentential form. For a definition of topdown tree transducers we refer to [10, 32]. Let yD_1 denote the family of tree transformation languages (i.e., the yields of images of recognizable tree languages under topdown tree transducers) and let $ydetD_1$ denote the subfamily of deterministic tree transformation languages. It was shown in [11] that ETOL $\subseteq yD_1$ and EDTOL $\subseteq ydetD_1$ The following result was obtained in [14, 36].

LEMMA 3.1. *(i) If $L \in$ ETOL has property P_3, then $L \in$ EDTOL. (ii) If $L \in yD_1$ has property P_3, then $L \in ydetD_1$.*

We now present a specific language L_0, which can be recognized by a one-way stack automaton but which is not in ETOL (indeed not even in yD_1). L_0 will be the language of all possible (properly coded) cuts of the infinite binary tree (Figure 4(a)). A *cut* is a tuple of lexicographically ordered paths such that the nodes which are endpoints of these paths form a cross section of the tree. A cut is also known as a complete binary code. Formally, a cut is a finite nonempty tuple of words over $\{0, 1\}$ defined recursively as follows:

(i) $\langle\lambda\rangle$ is a cut;
(ii) if $\langle v_1, \ldots, v_k\rangle$ and $\langle w_1, \ldots, w_n\rangle$ are cuts, then so is $\langle 0v_1, \ldots, 0v_k, 1w_1, \ldots, 1w_n\rangle$.

The strings w_i in a cut $\langle w_1, \ldots, w_n\rangle$ are called nodes An example of a cut for Figure 4(a) is given in Figure 4(b). The following properties of cuts are well known and easy to prove.

C_1. All nodes in a cut are different.
C_2. If $\langle w_1, \ldots, w_n\rangle$ is a cut, then $\sum_{i=1}^{n} 2^{-|w_i|} = 1$
C_3 For given integers $k_1, \ldots, k_n$ there is at most one cut $\langle w_1, \ldots, w_n\rangle$ such that $|w_i| = k_i$ for $1 \leq i \leq n$.

Definition 3.2. Let a and b be symbols different from 0 and 1

$$L_0 = \{aw_1 0 b w_1 1 a w_2 0 b w_2 1 \cdots a w_n 0 b w_n 1 \mid \langle w_1, \ldots, w_n\rangle \text{ is a cut}\}.$$

Note that for a string $s = aw_1 0 b w_1 1 \cdots a w_n 0 b w_n 1 \in L_0$ the tuple $\langle w_1 0, w_1 1, \ldots, w_n 0, w_n 1\rangle$ is a cut also. This cut will be called the "cut corresponding to s," whereas $\langle w_1, \ldots, w_n\rangle$ will

be called the "cut underlying s." One can define L_0 by the following basic macro grammar:

$$S \to F(\lambda),$$
$$F(x) \to F(x0)F(x1),$$
$$F(x) \to ax0bx1.$$

Thus $L_0 \in$ EB. L_0 can be recognized on a one-way stack automaton, as shown in the following lemma.

LEMMA 3.3. $L_0 \in S$.

PROOF. We shall program a one-way stack automaton which generates a cut in its stack. The stack automaton starts off with an arbitrary number of 0's in its stack, thus guessing the first node of the cut. The stack will contain the path description of next nodes of the cut in subsequent stages. To read in its stack, the machine will use a simple subroutine VERIFY, which will be called when the stack-pointer is at the top and which "thunks" the pointer to the bottom square to test in subsequent moves that the stack (from bottom to top) matches a next portion of the input. After a successful match the stack-pointer is back at the top and the input-head is pointing to the beginning of the (alleged) next node of the cut. The process repeats until the cut is verified or a mismatch occurs (in which case the machine rejects). Let "read(x)" be the symbolic instruction for reading an input symbol and moving the input-head one square to the right, provided x was the symbol read, and rejecting otherwise.

The subroutine PUSHZEROS (nondeterministically) pushes zero or more 0's on top of the stack when it is called. It is given by

```
begin
    repeat (push(0) or exit) until false
end PUSHZEROS;
```

The text for VERIFY is

```
begin
    while not bottom stack do move stack-pointer down od;
    while not top stack
        do read(stacksymbol), move stack-pointer up od;
    read(stacksymbol)
end VERIFY,
```

For the further programming of the machine, observe that two nodes w_1 and w_2 are consecutive nodes in a cut if and only if there is an "ancestor" w such that $w_1 \in w01^*$ and $w_2 \in w10^*$. Hence, starting with an empty stack, the machine can essentially follow a preorder traversal strategy on the code tree to visit the consecutive nodes of a cut. The subroutine PUSHZEROS is used each time to guess how "deep" the traversal strategy must descend. The complete program can be described as follows:

```
begin
        PUSHZEROS;
  loop: push(0); read(a), VERIFY,
        pop, push(1); read(b); VERIFY,
        while stacksymbol = 1 and not stack empty do pop od,
        if stack empty then halt
            else pop, push(1), PUSHZEROS fi,
        goto loop
end
```

It is left to the reader to prove this program correct. □

In the next lemma it is shown that L_0 is not a tree transformation language (and hence not in ETOL).

LEMMA 3.4. $L_0 \notin yD_1$.

PROOF. We first prove that L_0 has property P_3. By Lemma 3.1(ii) it then suffices to prove that $L_0 \notin \text{ydetD}_1$, which we will do using a pumping lemma for ydetD_1 due to Perrault [31].

To prove that L_0 has property P_3, assume that the strings $s_1 = xuyvz$, $s_2 = xuyuz$, $s_3 = xvyuz$, and $s_4 = xvyvz$ are all in L_0. We have to argue that $u = v$. Note first that u (or v) cannot contain two occurrences of symbols a or b, because otherwise the cut corresponding to s_2 (or s_4 respectively) would not satisfy C_1 above. Hence there remain three cases: (1) $u, v \in \{0, 1\}^*$, (2) $u, v \in \{0, 1\}^* a\{0, 1\}^*$, and (3) $u, v \in \{0, 1\}^* b\{0, 1\}^*$. Mixed cases cannot occur since in both s_1 and s_2 symbols a and b have to alternate. In case (1) it follows from C_2, applied to the cuts corresponding to s_1 and s_2, that $|u| = |v|$ and then from C_3 that $u = v$. In case (2) equality of u and v follows easily from the fact that the nodes surrounding any b in s_1 and s_2 are of the form $w0$ and $w1$ respectively, so that a change around any a would influence at most one of these. In case (3) application of C_2 and C_3 to the cuts underlying s_1 and s_2 yields $u = v$ (similar to case (1)). This proves that L_0 has property P_3.

We now show that $L_0 \notin \text{ydetD}_1$. In [31] an intercalation lemma for tree transducer languages is proved that, in a straightforward way, gives rise to the following intercalation lemma for ydetD_1: For each $L \in \text{ydetD}_1$ there is an integer p such that every u in L longer than p can be written as $u = u_1 u_2 \cdots u_k$ with (a) $|u_i| \leq p$ for $1 \leq i \leq k$, and (b) for each N there are strings $v_1, \ldots, v_k$ such that $|v_1 \cdots v_k| > N$, $v_1 \cdots v_k \in L$, and, for $1 \leq i \leq k$, $\text{alph}(v_i) = \text{alph}(u_i)$ (where $\text{alph}(s)$ denotes the set of symbols occurring in any string s). Assume that $L_0 \in \text{ydetD}_1$. By the intercalation lemma cited every long string in L_0 has small substrings which can be pumped while staying on the same alph, without leaving L_0. Consider a string $u = aw_1 0 b w_1 1 \cdots a w_n 0 b w_n 1$ in L_0 with $|w_i| \geq p$ for $1 \leq i \leq n$ and let $u = u_1 \cdots u_k$ be as in the intercalation lemma. No u_i contains both symbols a and b because of the length restriction. If we pump any u_i and v_i (as above), the number of a's and b's cannot change (because of the alternation of the a's and b's) and so pumping of u to v does not change the a's and b's. This gives a contradiction because there can be no arbitrarily long cuts with the same number of nodes (i.e., $2n$). □

We can immediately conclude the following result:

THEOREM 3.5. *ETOL and S are incomparable, as are yD_1 and OI.*

PROOF. Straightforward from Lemmas 3.3 and 3.4, using also that $\text{yD}_1\text{–OI} \neq \emptyset$ (see [14]). □

The last part of this result settles an open problem in [14], where only the existence of a language in $\text{yD}_1\text{–OI}$ was shown.

4. *Extended Basic Macro Grammars and Stack Machines*

In this section we obtain a machine characterization of the family EB of languages definable by extended basic (or EB) macro grammars. We shall prove that the family EB coincides with the family of languages accepted by s-pd machines. The result immediately shows that $S \subseteq EB$ and demonstrates what power is cut off from arbitrary OI macro grammars by the constraint of not allowing nested calls. (Note that the s-pd machine is much more restrictive than the general nested stack automaton.) At the same time we obtain interesting, alternative proofs for the known results that ELB = ETOL [4] and ETOL = CS-PD [37]. The proofs will make use of the new machine models.

In order to describe particular s-pd machines we shall use a symbolic programming language with the following primitives:

Instructions

read(a): if the current input symbol is a, then move the input-pointer one square to the right, else reject the input string.

push(γ): push the symbol γ on top of the stack.
pop: pop the top symbol off the stack.

Both push(γ) and pop can be executed only when the stack-pointer is at the top of the stack, and they keep it at the top (an empty stack is assumed to have "the stack-pointer at its top").

movedown(γ): move the stack-pointer one square down and simultaneously push the symbol γ on top of the pushdown.
moveup: move the stack-pointer one square up and simultaneously pop the top symbol off the pushdown.

Both movedown(γ) and moveup keep the stack-pointer at the same level as the top of the pushdown.

Tests

bottom stack: true iff the stack-pointer is at the bottom square of the stack.
top stack: true iff the stack-pointer is at the top square of the stack.
stack empty: true iff the stack is empty.
pd empty: true iff the pushdown is empty.

Note that top stack and pd empty are equivalent tests.

Identifiers

stacksymbol: denotes the square the stack-pointer points at, and its contents.
pdsymbol: denotes the top square of the pushdown, and its contents.

LEMMA 4.1. *ELB $\subseteq$ CS-PD and EB $\subseteq$ S-PD.*

PROOF. In order to provide some intuition as to why EB languages can be recognized on s-pd machines, we first show how ELB languages are recognized on cs-pd machines. Consider an arbitrary ELB grammar G. We may assume that all rules in G are of the form $F(x_1, \ldots, x_n) \to F'(\theta_1, \ldots, \theta_m)$ or $F(x_1, \ldots, x_n) \to \theta$, where $\theta, \theta_1, \ldots, \theta_m$ are macro-free terms and θ does not contain + or $\emptyset$. Otherwise (cf. [4]), replace rules $F(\cdots) \to \psi_1 F'(\cdots)\psi_2$ and $F(\cdots) \to \psi$ by $F(x, \ldots, y) \to F'(x\psi_1, \ldots, \psi_2 y)$ and $F(x, \ldots, y) \to Z(x\psi y)$ respectively, where x and y are new arguments in which the left and right context of the nonterminal are stored and Z is a new nonterminal with rule $Z(x) \to x$. We shall first write a recursive program to recognize G's language using only a checking stack for storage, and then argue how the recursion can be implemented using the extra pushdown facility of the cs-pd machine.

The cs machine applies the rules of G to generate a complete symbolic expansion of the initial macro F_0 on its stack, without substituting actual for formal parameters quite yet (see Figure 5): $F_0 \to F_1(\theta_1, \ldots, \theta_{k_1})$, $F_1(x_1, \ldots, x_{k_1}) \to F_2(\cdots), \ldots, F_i(x_1, \ldots, x_{k_i}) \to F_{i+1}(\cdots), \ldots, F_n(x_1, \ldots, x_{k_n}) \to \theta$. Thus, the checking stack symbols code right-hand sides of rules and F_0. After completing an expansion, the machine moves down one square and calls the recursive procedure EVAL to "evaluate" the symbolic term θ, i.e., to determine actual values for the constituent parameters in it by retracing back to the start of the macro expansion. The overall (nondeterministic) program can be described as follows:

```
begin
    push(F0),
    while stacksymbol ≠ F(···) do
    push(any righthand side of a rule for F) od,
    θ := stacksymbol, movedown,
    EVAL(θ)
end
```

The procedure EVAL has one argument, which always is a string of terminal symbols and formal parameters x_j. It determines (and reads) a possible value of x_j at the current

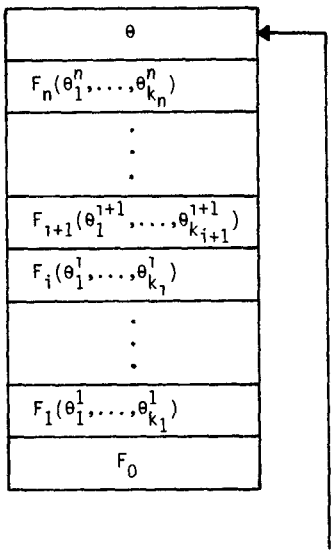


FIG 5 The symbolic expansion of the initial macro

level in the stack, which is the current level of macro expansion. We program EVAL as a recursive routine as follows. Later we will show that EVAL can be implemented on a cs-pd machine. By head(θ) and tail(θ) we denote the first symbol of θ and the string obtained from θ by erasing head(θ), respectively. The ith terminal symbol is denoted as a_i.

```
procedure EVAL(θ),
    if θ ≠ λ then
        case head(θ) of
            a_i read(a_i);
            x_j: begin ψ := any element of the (finite) set of strings of terminal symbols and formal parameters
                            denoted by the jth argument of stacksymbol,
                    movedown; EVAL(ψ); moveup
                end
        esac;
        EVAL(tail(θ))
    fi;
```

Note that "a_i: *read*(a_i)" abbreviates "a_1: *read*(a_1); ...; a_k: *read*(a_k)" where $\Sigma = \{a_1, \ldots, a_k\}$. Similarly for the x_j-clause. In the program it is understood that the machine rejects if there is no value for ψ, i.e., if the set denoted by the jth argument of stacksymbol is empty.

It is not hard to see that the procedure works correctly and verifies that the expansion generated in the stack represents a derivation of the input. Since the program runs with an ordinary checking stack as storage, we only have to argue that the recursion can be implemented using a pushdown store which moves in parallel with the stack. It should be clear that this can be done by storing the current argument of EVAL in the pushdown square at the current level of the checking stack. Note that in no call to EVAL is its argument longer than the right-hand side of a rule. Hence the pushdown symbols can just be codes for these arguments. This leads to the following iterative program for the recognition of G's language on a cs-pd machine:

```
begin
    push(F_0),
    while stacksymbol = F(···) do
    push(any right-hand side of a rule for F) od,
    movedown(stacksymbol),
    EVAL
end,
```

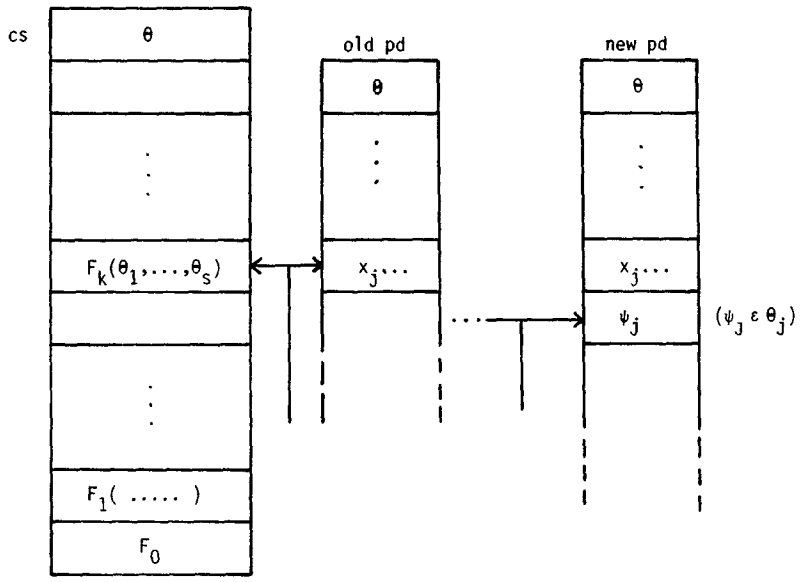


FIG 6 A move of the cs-pd machine

where EVAL now denotes the following routine text (with no parameter, because we keep an explicit pushdown to store the argument this time):

```
begin
    loop. if pdsymbol = λ
        then moveup;
            if pd empty then halt
                    else pdsymbol = tail(pdsymbol) fi
        else case head(pdsymbol) of
            a_i begin read(a_i), pdsymbol := tail(pdsymbol) end,
            x_j begin ψ = any element of the (finite) set of strings of terminal symbols and formal parameters
                        denoted by the jth argument of stacksymbol,
                    movedown(ψ)
                end
            esac fi,
        goto loop
end EVAL
```

A typical change of the cs-pd store effected by the "$\psi := \ldots$; *movedown*(ψ)" statements is indicated in Figure 6. Note that the arguments of F_1 can only consist of terminal symbols, and the pushdown store will not grow beyond the bottom of the checking stack.

As an example we consider the grammar with rules

$$
\begin{aligned}
&S \to F(\lambda, \varnothing),\\
&F(z, x) \to G(z, x, \lambda),\\
&F(z, x) \to zx,\\
&G(z, x, y) \to G(za, x, ya) \qquad \text{for all} \quad a \in \Sigma,\\
&G(z, x, y) \to F(z\#, x + y),
\end{aligned}
$$

which generates the language of Example 2.1. Snapshots from the recognition of the string $a\#b\#a$ on a cs-pd machine following the given algorithm are shown in Figure 7.

We continue our proof of Lemma 4.1 and show now that EB $\subseteq$ S-PD. Consider an arbitrary EB macro grammar. We shall program an s-pd machine which parses the grammar in a direct manner. On its stack the machine will symbolically expand macro calls in leftmost order as if the grammar were context-free. Each time a leftmost part

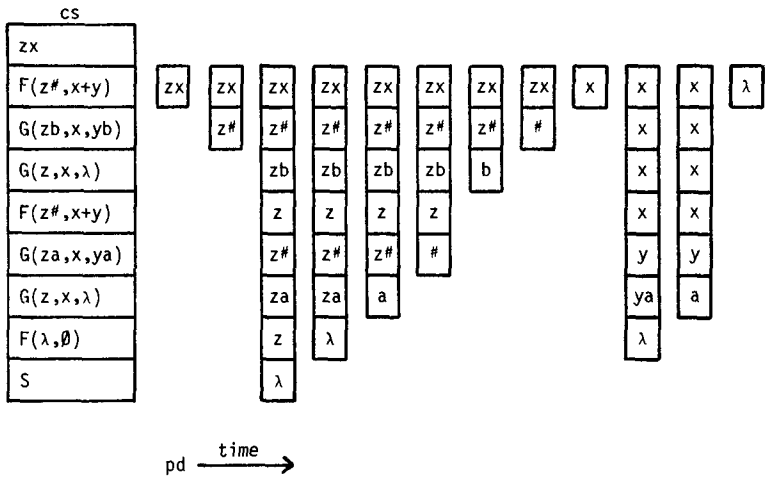


FIG 7 Recognition of $a\#b\#a$ on a cs-pd machine

resulting from the expansion does not contain further macro calls, the machine will evaluate the formal parameters of this part as in the ELB case! Thus the stack symbols of the machine will be codes for right-hand sides of rules and their suffixes. Assume, as we may, that the symbol + occurs only in the arguments of macro calls. Thus, each right-hand side θ of a rule is of the form $\theta = \theta_1\theta_2 \cdots \theta_k$ such that, for $1 \leq i \leq k$, either θ_i is terminal, or $\theta_i = x_j$ for some formal parameter x_j, or θ_i is a macro call $F(\psi_1, \ldots, \psi_n)$. We shall denote θ_1 by head(θ) and $\theta_2 \cdots \theta_n$ by tail(θ). The program for the s-pd machine is as follows:

```
begin push(initial nonterminal),
cycle  if stacksymbol = λ
       then pop, if stack empty then halt
                                 else stacksymbol = tail(stacksymbol) fi
       else case head(stacksymbol) of
            F(···)  push(any righthand side of a rule for F),
                a_i  begin read(a_i), stacksymbol = tail(stacksymbol) end,
                x_j  begin movedown(x_j),
                           EVAL,
                           stacksymbol = tail(stacksymbol)
                     end
            esac fi,
       goto cycle
end.
```

where EVAL is the same routine as in the ELB case, except for the assignment to ψ which should now read as follows:

ψ = *any element of the (finite) set of strings of terminal symbols and formal parameters denoted by the jth argument of head(stacksymbol).*

Thus the new parameter value for EVAL is always picked from the leftmost macro call in the current stack square. Note that all stack symbols that are not at the top of the stack start with a macro call. It is an easy exercise for the reader to verify that the program above indeed recognizes the given EB language on an s-pd machine. □

For proving the converse of Lemma 4.1 we use the following well-known fact from the theory of AFA and AFL (see [17, Chap. 5] for details): Each family of languages defined by a class of "well-behaving" one-way nondeterministic acceptors of a same "type" is a full principal AFL. Thus, to prove that S-PD $\subseteq$ EB we only have to show that EB is a full

AFL containing the full AFL generator of S-PD. (For the unexplained terminology from AFL theory, see [17].)

LEMMA 4.2. *EB is a full AFL.*

PROOF. It is straightforward to prove closure of EB under union, concatenation, and Kleene star (cf. [16]). To prove that EB is a full AFL it now suffices to show closure under regular substitution and under intersection with regular sets.

Closure under regular substitution is easy: just replace in a given EB grammar each terminal symbol a by a regular expression for the regular language to which it is mapped. This gives an REB grammar for the substitution-image, which can be transformed into an ordinary EB grammar by Lemma 2.2(i).

Closure under intersection with regular sets can be shown as follows (cf. [4, 16]). Let an EB grammar G be given with terminal alphabet Σ and rules of the form $F(x_1, \ldots, x_n) \to G_1(\cdots)G_2(\cdots)\cdots G_k(\cdots)$ or $F(x_1, \ldots, x_n) \to \theta$, where θ does not contain nonterminals. Let a regular language R be given as $R = h^{-1}(E)$ where h is a homomorphism mapping Σ^* into a finite monoid H and $E \subseteq H$. An EB grammar G' for $L(G) \cap R$ is constructed as follows. For each x_i and each $f \in H$, introduce a new formal parameter $\langle x_i, f\rangle$ which will serve to store all strings w, originally stored in x_i, such that $h(w) = f$. We extend h to these symbols by defining $h(\langle x_i, f\rangle) = f$. The nonterminals of G' are of the form $[F, f]$ where F is a nonterminal of G of rank n and $f \in H$; $[F, f]$ has all formal parameters $\langle x_i, g\rangle$ with $1 \le i \le n$ and $g \in H$, in some order. The rule $F(x_1, \ldots, x_n) \to G_1(\cdots)\cdots G_k(\cdots)$ is changed into all rules of the form

$$[F, f](\cdots) \to [G_1, f_1](\cdots)[G_2, f_2](\cdots)\cdots[G_k, f_k](\cdots),$$

such that $f_1 f_2 \cdots f_k = f$ and the arguments of $[G_m, f_m]$, $1 \le m \le k$, are computed as follows. Let S_i be the set denoted by the ith argument of G_m in the right-hand side of the original rule; let ϕ be the finite substitution with $\phi(x_t) = \{\langle x_t, f\rangle \mid f \in H\}$ for $1 \le t \le n$, and the identity otherwise; then the argument of $[G_m, f_m]$ on the position of $\langle x_i, g\rangle$ is any term denoting the finite set $\{w \in \phi(S_i) \mid h(w) = g\}$. Each rule $F(x_1, \ldots, x_n) \to \theta$ is replaced by all rules of the form $[F, f](\cdots) \to \theta'$, where θ' is any term denoting the set $\{w \in \phi(S) \mid h(w) = f\}$, S being the set denoted by θ. Finally a new initial nonterminal S_0 is introduced with all rules $S_0 \to [S, f]$, where S is the initial nonterminal of G and $f \in E$. The formal proof of this construction is a standard exercise. □

LEMMA 4.3. *S-PD ⊆ EB and CS-PD ⊆ ET0L.*

PROOF. What makes an s-pd machine extend a finite automaton is the way its stack-instructions can be nested and intertwined. If we could find a language L which codes each possible sequencing of stack-instructions, then only AFL operations would be needed to insert the input symbols at the proper places and to make a "selection-pass" to extract those sequences which are consistent with the finite state behavior of a particular machine, i.e., L would be a full AFL generator of S-PD. By a standard encoding we may assume that the stack and pushdown alphabets of the s-pd machine are $\{0, 1\}$. In order to obtain a manageable L we reformulate the s-pd machine to have the following basic instructions (aside from the read instruction) for $a, \gamma \in \{0, 1\}$:

a: push the symbol a on top of the stack.
a^{E}: pop the symbol a off the stack.
a^{D}_{γ}: movedown from the stacksymbol a and push γ on top of the pushdown.
a^{U}_{γ}: moveup to the stacksymbol a and pop γ off the pushdown.

It should be obvious that the new instructions can be simulated by the old ones and vice versa.

Checking the complicated definitions in [17, Secs. 5.2, 5.3] shows that s-pd machines form a reduced, finitely encoded AFA satisfying [17, Th. 5.3.2]. (The laborious task of verifying it is left to the reader.) Hence S-PD is a full principal AFL, with a generator L

obtained by taking all permissible sequences of basic instructions which lead from empty stack to empty stack. L can be defined by an REB grammar with the following six rules:

$$\begin{aligned}
&1, 2.\ S \to aF(\lambda)a^{E}S, \qquad a \in \{0, 1\},\\
&3.\ S \to \lambda,\\
&4, 5.\ F(x) \to xaF((a_0^{D}xa_0^{U} + a_1^{D}xa_1^{U})^{*})a^{E}F(x), \qquad a \in \{0, 1\},\\
&6.\ F(x) \to x.
\end{aligned}$$

The set-parameter x stands for the set of all sequences of a_γ^{D} and a_γ^{U} instructions which can be executed on a certain stack s_x, starting and ending at the top of s_x. In rules 1 and 2 this stack is set to the one containing only the symbol a, whereas in rules 4 and 5 the symbol a is pushed on this stack for the first F in the right-hand side, and it stays the same for the second F. $F(x)$ generates the set of all instruction sequences that can be executed starting and ending at the top of stack s_x without changing its contents in the intermediate steps. By formalizing these statements one can easily prove the grammar correct By Lemma 2.2(i) one may convert the REB grammar into an equivalent EB grammar, and it follows that $L \in$ EB. As L is a full AFL generator of S-PD and EB is a full AFL, we conclude that S-PD $\subseteq$ EB.

The proof of CS-PD $\subseteq$ ET0L is very similar. Without making the definition of the cs-pd machine as AFA precise, it should be clear to the reader that a full AFL generator of CS-PD consists of the language of all instruction sequences $w \in \{a_0^{D}, a_0^{U}, a_1^{D}, a_1^{U}\}^{*}$ such that, on some stack s, w can be executed starting and ending at the top of s. This language is generated by the following RELB grammar:

$$\begin{aligned}
&1.\ S \to F(\lambda),\\
&2, 3.\ F(x) \to F((a_0^{D}xa_0^{U} + a_1^{D}xa_1^{U})^{*}), \qquad a \in \{0, 1\},\\
&4.\ F(x) \to x.
\end{aligned}$$

In the original proof of CS-PD $\subseteq$ ET0L given in [37], the essential idea was to construct an ET0L grammar with initial symbol x and regular (rather than finite) substitutions f_a and g such that: $f_a(x) = (a_0^{D}xa_0^{U} + a_1^{D}xa_1^{U})^{*}$, $g(x) = \lambda$ and the identity otherwise, which clearly generates this language also. Since an ET0L grammar with regular substitutions can be transformed into one with finite substitutions only (compare e.g. [3]), it follows again that the language is in ET0L. Since ET0L is a full AFL [33], we can conclude that CS-PD $\subseteq$ ET0L. □

Combining Lemmas 4.1 and 4.3, we obtain the main result of this section.

THEOREM 4.4. *EB = S-PD.*

We also obtain the following (known) characterizations of ELB (cf. [4, 37]).

THEOREM 4.5. *ELB = ET0L = CS-PD = NES-PD.*

PROOF. CS-PD = NES-PD was shown in Theorem 2.3. ELB $\subseteq$ CS-PD $\subseteq$ ET0L was shown in Lemmas 4.1 and 4.3. The proof of ET0L $\subseteq$ ELB is straightforward (cf. [4]). The ELB grammar for simulating an ET0L grammar has, in addition to its initial macro S, only one macro F. For each finite substitution τ of the ET0L grammar a rule $F(x_1, \ldots, x_n) \to F(\tau(x_1), \ldots, \tau(x_n))$ is included, where $x_1, \ldots, x_n$ are renamings of the symbols $a_1, \ldots, a_n$ of the ET0L grammar ($\tau(x_i)$ denotes the renaming of $\tau(a_i)$). The final rule is $F(x_1, \ldots, x_n) \to x_1$ (if a_1 is the initial symbol of the ET0L grammar) and the initial rule is $S \to F(\delta_1, \ldots, \delta_n)$ where $\delta_i = a_i$ if a_i is terminal and $\delta_i = \varnothing$ otherwise. During a derivation the actual parameters of F denote the set of all terminal strings derivable from the individual symbols $a_1, \ldots, a_n$ for a particular (arbitrary) sequence of substitution applications. Whenever macro expansion stops (with the final rule) we produce a set of terminal strings derivable from a_1. All words of the language can be obtained in this way. □

As the one-way stack automaton and the cs-pd machine both are degenerate versions of

the s-pd machine, and since we have shown in Section 3 that S and ETOL are incomparable, we get the following proper inclusions.

COROLLARY 4.6. *(i)* $S \subsetneq EB$. *(ii)* $ETOL \subsetneq EB$.

Corollary 4.6(i) shows that each stack language can be defined by an EB macro grammar, but not vice versa. It is open whether or not there is a natural restriction on EB grammars which characterizes S. Corollary 4.6(ii) seems to be the strongest ramification presently known of the hard result that ETOL $\subsetneq$ INDEXED [8].

The characterization of EB by s-pd machines gives us a handle on the study of various subfamilies of EB like ELB by simply varying restrictions on s-pd machines. It is interesting to see how such restrictions are directly reflected in the generator for S-PD, thus providing us with generators for the subfamilies.

Recall that the parameter for S-PD was defined by the grammar

$$\begin{aligned}
&S \to aF(\lambda)a^{E}S, \qquad a \in \{0, 1\},\\
&S \to \lambda,\\
&F(x) \to xaF((a_0^{D}xa_0^{U} + a_1^{D}xa_1^{U})^{*})a^{E}F(x), \qquad a \in \{0, 1\},\\
&F(x) \to x, \qquad a \in \{0, 1\}.
\end{aligned}$$

A generator for the family S is obtained by dropping the pushdown facility, i.e., by changing $(a_0^{D}xa_0^{U} + a_1^{D}xa_1^{U})^{*}$ into $(a^{D}xa^{U})^{*}$. A generator for NES-PD is obtained by dropping the $a^{E}F(x)$ part (and the $a^{E}S$). Thus the following RELB grammar defines a generator for ETOL (= NES-PD):

$$\begin{aligned}
&S \to aF(\lambda), \qquad a \in \{0, 1\}\\
&S \to \lambda,\\
&F(x) \to xaF((a_0^{D}xa_0^{U} + a_1^{D}xa_1^{U})^{*}), \qquad a \in \{0, 1\},\\
&F(x) \to x.
\end{aligned}$$

When we drop the pushdown facility from this grammar (as we did for the family S), we get a generator for NES, and dropping the "work in the stack, before you push more"-term xa produces a generator for CS:

$$\begin{aligned}
&S \to F(\lambda),\\
&F(x) \to F((a^{D}xa^{U})^{*}), \qquad a \in \{0, 1\},\\
&F(x) \to x.
\end{aligned}$$

Digressing on EB (and remembering that OI = INDEXED [1, 16]), we may ask how the restriction of nonnested nonterminals in EB macro grammars perhaps corresponds to an equally natural restriction on indexed grammars. Filè and van Leeuwen [15] have recently obtained characterizations of both EB and ELB by means of simple classes of indexed grammars. It turns out that the indexed grammars corresponding to EB are the "restricted indexed grammars" (RIG's; see Aho [1, p. 670]), although in [15] a more attractive equivalent form is given. A RIG is like an ordinary indexed grammar, except that nonterminals produced by flag-consuming productions cannot themselves ever introduce new flags. (Such nonterminals were called "intermediates" in [1].) It is remarkable that the class of restricted indexed grammars appears to be "natural" after all, from the point of view of macro grammars. One can show (cf. [15]) the following result.

THEOREM 4.7. *The family of languages generated by Aho's restricted indexed grammars is precisely equal to EB.*

Note that Aho's result that S $\subseteq$ RIG [2] is now an immediate consequence of our machine characterization of EB. Corollary 4.6 confirms the conjecture in [2] that S is strictly included in RIG (= EB).

5. *Deterministic Restrictions*

In this section we show that there is a natural deterministic restriction on the stack-handling capability of s-pd machines which yields a machine characterization of the

"original" (i.e., nonextended) nonnesting or basic macro grammars. We continue the work of Fischer [16] to give further useful characterizations of the family of basic macro languages (B). The same restriction for cs-pd (equivalently, nes-pd) machines gives a machine model for the family of linear basic macro (or EDT0L) languages (LB). At the end of this section we position all families of this paper in a diagram and argue the correctness of the incomparabilities and proper inclusions shown.

In order to explain the particular deterministic restriction for s-pd machines it is convenient to view s-pd machines as generating machines (i.e., generators or machines with output) rather than as accepting machines (i.e., acceptors or machines with input), by simply changing the instruction read(a) into write(a). It should be clear that this makes no difference with respect to the power of general (nondeterministic) s-pd machines but it obviously changes their mode of operation. We say that an s-pd machine is *stack-deterministic (as a generating machine)* if and only if it is deterministic in stack-reading mode, i.e., if it acts completely deterministically when it moves up and down the stack using the pushdown facility or when it is on top of the stack and must choose between staying at the top or moving down into the stack. Thus, a stack-deterministic machine can be nondeterministic only in stack-writing mode. From the acceptor point of view it means that we put restrictions on the program of the machine such that during inspection of the stack (with the added pushdown facility) at most one possible piece of input can be recognized. The reader is urged to ponder this concept of stack determinism before continuing.

We shall abbreviate "stack-deterministic s-pd" by ds-pd. The same restriction can be put on restricted versions of the s-pd generating machine (with a similar notation). In particular a dcs-pd generating machine first builds a checking stack nondeterministically, then generates output while checking its stack deterministically.

We now show that stack determinism provides a characterization of the basic and linear basic macro languages. Intuitively, deterministic handling of the stack corresponds to "deterministic arguments" (i.e., arguments not involving + and $\varnothing$) in the macro bodies of the grammar.

LEMMA 5.1. *$B \subseteq$ DS-PD and $LB \subseteq$ DCS-PD.*

PROOF. In the proof of Lemma 4.1 the procedure EVAL describes the stack inspection of the s-pd and cs-pd machines. It should be clear that for basic grammars EVAL can be made deterministic by changing the assignment "$\psi :=$ any element..." into "$\psi :=$ *the* element...." By changing "read(a_i)" into "write(a_i)" throughout the program also, a stack-deterministic program for the s-pd or cs-pd generating machine is obtained which generates the language of the given basic grammar. □

LEMMA 5.2. *DS-PD $\subseteq B$ and DNES-PD $\subseteq LB$.*

PROOF. Unfortunately no simple proof analogous to the nondeterministic case (Lemma 4.3) is known. The reason is that neither B nor LB is an AFL [16], which means that the technique of finding a general AFL generator no longer works. We shall prove the inclusions stated by providing a direct construction to obtain the basic or linear basic macro grammar for the language of each given ds-pd or dnes-pd generating machine, respectively.

Let a ds-pd generating machine M be given in the usual way by a set of states Q, a state transition function, output alphabet Σ, pushdown alphabet Γ, etc. We assume that M accepts by final state and empty stack. Let \$ be a new symbol, used to indicate an empty pushdown. For each $q \in Q$ and $\gamma \in \Gamma \cup \{\$\}$, introduce a formal parameter $x(q, \gamma)$ and denote the sequence of these parameters (in some order) by $\vec{x}$. The nonterminals of the basic macro grammar to be constructed are of the form $[A; p, q; f]$ with arguments $\vec{x}$, where A is an element of the stack alphabet, p and q are in Q, and f is a partial function from $Q \times (\Gamma \cup \{\$\})$ into Q.

The idea behind the construction of the basic macro grammar for M's language is as follows. Suppose we have a stack with an occurrence of the symbol A. Let s be the portion of the stack below it. Assume that the generating machine, when started in state r and reading A and with γ in the "opposite" pushdown square, works on s (and A) for a while and then returns to the occurrence of A without being able to immediately move back into s. Let $w(r, \gamma) \in \Sigma^*$ be the output generated during this computation, and let $f(r, \gamma)$ be the state in which the machine returns to A for the last time. (Note that $w(r, \gamma)$ and $f(r, \gamma)$ are unique, due to the restriction of stack determinism.) We want $[A; p,q; f](\vec{w})$, with $\vec{w}$ denoting the sequence of all $w(r, \gamma)$'s, to generate $v \in \Sigma^*$ if and only if M can produce output v in the computation resulting when M is started in state p with A on top of the stack (and empty pushdown at that time), continued until M pops this A (for the first time) and enters state q.

This idea can be implemented with the following rules.

(1) (Compare rule 6 in the EB grammar of Lemma 4.3.) $[A; p,q; f](\vec{x}) \to x(p, \$)w$ if M, in state $f(p, \$)$ with A on top of the stack (and empty pushdown), pops A and goes into state q producing output w (according to the state transition function).

(2) (Compare rules 4 and 5 in the EB grammar of Lemma 4.3.) $[A; p,q; f](\vec{x}) \to x(p, \$)w[B; p_1,p_2; g](\vec{u})[A; p_2,q; f](\vec{x})$ (any p_2) if (a) and (b) hold:

(a) M, in state $f(p, \$)$ with A at the top of the stack (and empty pushdown), pushes B and goes into state p_1 producing output w.

(b) $u(r, \gamma)$ and $g(r, \gamma)$ are obtained by writing the (symbolic) output of M, starting in state r at the top symbol B of the stack with γ in the opposite pushdown square (empty if $\gamma = \$$).

For instance, if M moves down (to A) in state r_1 pushing $\gamma_1 \in \Gamma$ on the pushdown and producing output $v_1 \in \Sigma^*$, then we write $u(r, \gamma) = v_1x(r_1, \gamma_1)\cdots$ and continue in state $f(r_1, \gamma_1)$. If M now moves up (to B) and down again into state r_2 pushing γ_2 on the pushdown and producing output v_2, then we write $u(r, \gamma) = v_1x(r_1, \gamma_1)v_2x(r_2, \gamma_2)\cdots$ and continue in state $f(r_2, \gamma_2)$. If M finally moves up to B in state r_n, producing output v_n, and it cannot move down again, then we set $u(r, \gamma) = v_1x(r_1, \gamma_1)v_2x(r_2, \gamma_2)\cdots v_n$ and $g(r, \gamma) = r_n$. Note that each $x(r_i, \gamma_i)$ occurs at most once in $u(r, \gamma)$, since otherwise M would loop. In case "something goes wrong," $g(r, \gamma)$ is left undefined and $u(r, \gamma)$ is defined arbitrarily. It is left to the reader to construct the initial rules, to fill in the tedious details, and to prove the correctness of the construction formally. It shows that DS-PD $\subseteq$ B.

A dnes-pd generating machine M can be viewed as a ds-pd generating machine in which all pops happen at the end of the computation (without output and, say, in a certain final state q_e). This gives a linear basic grammar with the following rules:

(1) $[A; p; f](\vec{x}) \to x(p, \$)$ if M, in state $f(p, \$)$ with A at the top of the stack, goes into state q_e.

(2) $[A; p; f](\vec{x}) \to x(p, \$)w[B; p_1; g](\vec{u})$ if (a) and (b) hold as before.

This shows that DNES-PD $\subseteq$ LB. □

Lemmas 5.1 and 5.2 together give a machine characterization of the nonnesting macro grammars.

THEOREM 5.3. *(i) B = DS-PD. (ii) EDT0L = LB = DCS-PD = DNES-PD.*

The families of languages discussed in this paper are put together in Figure 8. The dimensions in the diagram (without yD_1 and OI) can be interpreted as follows. To the right, first add "push," then add "pop"; downward, add "D"; from the reader away, add "pd."

We note that the family DCS is known from the literature. Since a checking stack (or cs-pd) machine, considered as a generating machine, can be viewed as a transducer (with the checking stack contents as input), DCS is recognized to be the image of the regular

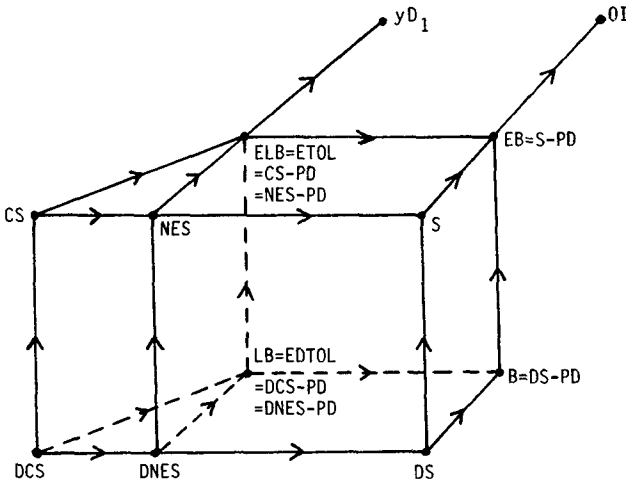


FIG 8 The relation between various machine and language classes Some lines are dotted to improve perspective All inclusions shown are proper

languages under two-way deterministic finite state transducers [27]. Note that a stack-deterministic cs-pd generating machine is the same as a deterministic cs-pd transducer. DCS is also equal to the class of languages accepted by "finite visit" cs-pd machines (cf. [22]) and equal to the class of languages generated by ETOL grammars of finite index (cf. [34]). We also note that all families on the upper level of Figure 8 and DCS are full AFL's. The other four families are closed under $\cup$, $\cdot$, $*$, and deterministic gsm mappings (obvious for the machines), but not under h^{-1}.

The correctness of the inclusions and incomparabilities shown in Figure 8 follows from the existence of languages in the following classes:

(1) CS−B: The language $\{w \in \{a, b\}^* |$ the number of a's in w is not prime$\}$ is in CS [21], but not in B (not even in the class of IO macro languages); the latter follows by observing that the proof in [16, Sec. 3.4] showing the existence of a language in OI-IO proves, in fact, that if $L \subseteq a^*$ and $h^{-1}(L) \in$ IO (where $h(a) = a$ and $h(b) = \lambda$), then L is regular.

(2) LB−S: $\{a^n b^{n^2} c^n | n \geq 1\}$; see [30].

(3) DNES−CS: $\{a^{n^2} | n \geq 1\}$; see [21]. It is easy to see that this language can be generated by a dnes generating machine which after having produced a^{k^2} as output, has a^k in its stack.

(4) DS−yD$_1$: The language L_0 of Section 3 is in DS as can easily be seen after changing "read" into "write" in the program in Lemma 3.3.

(5) OI−EB: See [12].

(6) yD$_1$−OI: See [14].

Note that it follows from the above that $L = \{a^{n^2} | n \geq 1\} \in$ DNES, but $h^{-1}(L) \notin$ B (with h as in (1)). Thus all language families in between DNES and B are not closed under h^{-1} either.

The reader may wonder about the position of the class CF of context-free languages in Figure 8. Certainly none of the classes shown are contained in CF, because $\{a^n b^n c^n | n \geq 1\}$ is clearly in all of them. It is also easy to see that CF $\subseteq$ CS-PD and CF $\subseteq$ DS. One can show that CF is not included in NES as follows (cf. [22, 23]). Assume first that CF $\subseteq$ CS. Then, in particular, all parenthesis languages [29] would be in CS. Such languages do not contain infinite regular subsets. This property ensures that they must, in fact, be in DCS (since each square of the checking stack can only be visited a finite number of times; cf. [27]). Since DCS is closed under homomorphism it follows that CF $\subseteq$ DCS $\subseteq$ EDTOL,

which clearly contradicts the incomparability of CF and EDT0L shown in [7]. Hence CF is not included in CS. From [21, Lem. 4.1] it can now be concluded that CF is not included in NES either.

6. *Conclusion*

We have presented a detailed study of several language classes closely related to EB, the family of languages generated by macro grammars with set-parameters in which no macro calls within the parameters are allowed. We have presented a feasible protocol for implementing macro expansion for such grammars and proved that EB is the family of languages recognized by s-pd machines. Related characterizations for ELB and other families were obtained. One may view s-pd machines as nested stack automata [2], allowing only nested stacks of size 1, which are inserted "between" the symbols in the main stack as the stack-pointer moves down. The one-element stacks dissolve as the stack-pointer moves up, just as symbols are popped off the pushdown in the s-pd machine. As this protocol must be a strong curtailment of a nested stack automaton, it is supporting evidence that there must be a rich structure between EB and OI. We have explained some of the similarities and differences between features of stack machines on the one hand (push, pop, moveup, movedown) and properties of macro grammars on the other hand (set-parameters, linearity, nesting). It is often the case that a machine model for a family of languages is the easiest characterization to use in connection with intuitive reasoning, whereas the grammar model has its strength when dealing with formal proofs. In our opinion this is precisely the case for the class of languages discussed in this paper, and we hope that our results will prove helpful for a better understanding of the properties of stack machine and their correspondence to macro grammars.

ACKNOWLEDGMENTS. We thank Peter Asveld, Gilberto Filè, and Giora Slutzki for useful discussions and the referees for numerous helpful comments.

REFERENCES

1. AHO, A V Indexed grammars—an extension of context-free grammars *J ACM 15*, 4 (Oct 1968), 647–671
2. AHO, A V Nested stack automata *J ACM 16*, 3 (July 1969), 383–406
3. CHRISTENSEN, P A. Hyper-AFLs and ET0L systems In *L Systems*, G Rozenberg and A. Salomaa, Eds, Lecture Notes in Computer Science 15, Springer-Verlag, Berlin, 1974, pp 254–257
4 DOWNEY, P J Formal languages and recursion schemes Ph.D Th, Rep TR-16-74, Harvard U, Cambridge, Mass, 1974
5 DIJKSTRA, E W Guarded commands, nondeterminacy and formal derivation of programs *Comm. ACM 18*, 8 (Aug 1975), 453–457
6 DIJKSTRA, E W *A Discipline of Programming* PH Series in Automatic Programming, Prentice-Hall, Englewood Cliffs, N J, 1976
7 EHRENFEUCHT, A, AND ROZENBERG, G On some context-free languages that are not deterministic ET0L languages. *RAIRO (Informatique Theoretique) 11* (1977), 273–291.
8 EHRENFEUCHT, A, ROZENBERG, G, AND SKYUM, S A relationship between ET0L and EDT0L languages *Theoret Comptr Sci 1* (1976), 325–330
9 EHRICH, R W, AND YAU, S S Two-way sequential transductions and stack automata *Inform and Control 18* (1971), 404–446
10 ENGELFRIET, J Bottom-up and top-down tree transformations—a comparison *Math Systems Theory 9* (1975), 198–231.
11 ENGELFRIET, J Surface tree languages and parallel derivation trees *Theoret Comptr. Sci 2* (1976), 9–27
12. ENGELFRIET, J Macro grammars, Lindenmayer systems and other copying devices. In *Automata, Languages and Programming*, Fourth Colloquium, A Salomaa and M Steinby, Eds, Lecture Notes in Computer Science 52, Springer-Verlag, Berlin, 1977, pp 221–229
13 ENGELFRIET, J, AND MEINECHE SCHMIDT, E IO and OI Part I *J Comptr Syst. Sci 15* (1977), 328–353, Part II *J Comptr Syst. Sci 16* (1978), 67–99
14 ENGELFRIET, J, AND SKYUM, S Copying theorems *Inform. Proc Letters 4* (1976), 157–161
15 FILÈ, G The characterization of some language families by classes of indexed grammars M Sc Th, Dept of Comptr Sci, Pennsylvania State U, University Park, Pa, 1977
16 FISCHER, M J Grammars with macro-like productions, Ph D Th, Harvard U, Cambridge, Mass, 1968
17 GINSBURG, S *Algebraic and Automata-Theoretic Properties of Formal Languages* Fundamental Studies in

Computer Science 2, North-Holland/American Elsevier, Amsterdam, 1975
18 GINSBURG, S, GREIBACH, S A, AND HARRISON, M A Stack automata and compiling *J ACM 14*, 1 (Jan 1967), 172–201
19 GINSBURG, S, GREIBACH, S A, AND HARRISON, M A One-way stack automata *J ACM 14*, 2 (April 1967), 389–418
20 GINSBURG, S, GREIBACH, S A, AND HOPCROFT, J E *Studies in Abstract Families of Languages.* Memoirs of the AMS 87, Amer Math Society, Providence, R I, 1969
21 GREIBACH, S Checking automata and one-way stack languages *J Comptr Syst Sci 3* (1969), 196–217
22 GREIBACH, S.A One-way finite visit automata *Theoret Comptr Sci 6* (1978), 175–222
23 GREIBACH, S A Private communication
24 HARRISON, M A, AND SCHKOLNICK, M. A grammatical characterization of one-way nondeterministic stack languages *J ACM 18*, 2 (April 1971), 148–172
25 HERMAN, G T., AND ROZENBERG, G *Developmental Systems and Languages* North-Holland, Amsterdam, 1975
26 HOPCROFT, J E, AND ULLMAN, J D *Formal Languages and Their Relation to Automata* Addison-Wesley, Reading, Mass, 1969.
27 KIEL, D I Two-way a-transducers and AFL *J Comptr Syst Sci 10* (1975), 88–109
28 MANNA, Z *Mathematical Theory of Computation* McGraw-Hill, New York, 1974
29 MCNAUGHTON, R Parenthesis grammars *J ACM 14*, 3 (July 1967), 490–500
30 OGDEN, W Intercalation theorems for stack languages Proc First ACM Symp on Theory of Computing, Marina del Rey, Calif, 1969, pp 31–42.
31 PERRAULT, C R Intercalation lemmas for tree transducer languages *J Comptr Syst Sci 13* (1976), 246–277
32 ROUNDS, W C Mappings and grammars on trees *Math Systems Theory 4* (1970), 257–287
33 ROZENBERG, G Extension of tabled 0L-systems and languages *Int J Comput Inform Sci. 2* (1973), 311–336
34 ROZENBERG, G, AND VERMEIR, D On ET0L systems of finite index Rep 75-13, Dept. of Mathematics, U of Antwerp, Wilrijk, Belgium, 1975
35 SALOMAA, A *Formal Languages* Academic Press, New York, 1973
36 SKYUM, S Decomposition theorems for various kinds of languages parallel in nature *SIAM J Comput 5* (1976), 284–296
37 VAN LEEUWEN, J Variations of a new machine model Conf Record 17th Annual IEEE Symp on Foundations of Computer Science, Houston, Texas, 1976, pp 228–235

RECEIVED SEPTEMBER 1977, REVISED SEPTEMBER 1978