STACK MACHINES AND CLASSES OF NONNESTED MACRO-LANGUAGES

Joost Engelfriet

Erik Meineche Schmidt

Jan van Leeuwen

RUU - CS - 77 - 2

August 1977

Rijksuniversiteit Utrecht

Vakgroep Informatica

Budapestlaan 6
Utrecht 2506
Telefoon 030-53 1454

STACK MACHINES AND CLASSES OF NONNESTED MACRO-LANGUAGES

Joost Engelfriet
Dept. of Applied Mathematics
Twente University of Technology
Enschede/the Netherlands

Erik Meineche Schmidt
Dept. of Computer Science
Cornell University
Ithaca, NY 14853/USA

Jan van Leeuwen
Dept. of Computer Science
The Pennsylvania State University
University Park, PA 16802/USA

Technical Report RUU-CS-77-2
August 1977

Department of Computer Science
University of Utrecht
3508 TA UTRECHT
the Netherlands

A preliminary version of this report appeared earlier as Technical Report 221 (Spring 1977), Department of Computer Science, the Pennsylvania State University, University Park, PA 16802 (USA).

proposed running head:

STACK MACHINES AND MACRO LANGUAGES

all correspondence to:

Dr. Jan van Leeuwen
Department of Computer Science
University of UTRECHT
3508 TA UTRECHT
the NETHERLANDS

STACK MACHINES AND CLASSES OF NONNESTED MACRO LANGUAGES

Joost Engelfriet*, Erik Meineche Schmidt**, Jan van Leeuwen***

ABSTRACT. We investigate a new class of generalized 1-way stack automata called s-pd machines. 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. There are various motivations for the model, including 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" macro grammars). If we demand the stack to be nonerasing or checking, then we obtain a useful machine characterization for the ETOL languages together with the known characterization of this language family by means of extended "linear" basic macro grammars. It follows that the nonerasing 1-way stack languages are (strictly) included in ETOL, but we prove that ETOL and the family of unrestricted 1-way stack languages are incomparable. Certain deterministic restrictions of s-pd machines lead to machine models for the linear basic macro (or, EDTOL) languages and for M. Fischer's original basic macro languages.

*Author's address: Dept. of Applied Math., Twente University of Technology, Enschede, the Netherlands.

**Author's address: Dept. of Computer Science, University of Aarhus, Aarhus C, Denmark. This work was carried out while the author was at the Dept. of Computer Science, Cornell University, Ithaca, NY 14853 (USA), with partial support from Aarhus University, Aarhus, Denmark.

***Author's address: Dept. of Computer Science, University of Utrecht, 3508 TA Utrecht, the Netherlands. This work was carried out while the author was affiliated with the Dept. of Computer Science, The Pennsylvania State University, University Park, PA 16802 (USA).

1. Introduction

The theoretical models of stack automata have a traditional motivation in the compilation of higher-level programming languages [18,19] and in the implementation of recursive procedures with parameters. A (1-way) stack automaton is a pushdown automaton which may enter its store in a read-only mode, whereas writing and erasing is still permitted to occur at the top of the stack only. There have been several studies to analyze the complexity of stack languages (see [26]) or to find a syntactic characterization of such languages [24,9].

About 1968 Aho [2] proposed a generalized stack automaton (the "nested stack automaton") 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 beyond its top back up the old stack again. The model is rather complex, and was designed specifically for implementing the grammatical mechanism of indexed languages (Aho [1]).

About the same time M. 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 made into 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 the original call by one of the strings in the corresponding macro body, after suitable passing of actual parameters.

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 and therefore model call-by-name as parameter passing mechanism. The relation between OI and IO ("inside-out") macro grammars was studied in [16,13]. M. 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.

M. Fischer [16; section 7] observed as a corollary to his analysis that OI macro grammars which permit nested macro calls 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 such grammars). It is one of the main objectives of this paper to investigate precisely what the generative capacity is of macro grammars without nested calls (the "basic" macro grammars) and to characterize a class of simple stack machines which naturally corresponds to this class of languages. The results suggest an interesting protocol for the expansion of macros with intermittent parameter evaluations, and we obtain a natural machine model for nondeterministic recursive program schemes with parameters but without 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 also 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 the defining body. In such "extended" basic macro grammars we shall 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 EDTOL, and that the family of extended linear basic macro languages (ELB) corresponds exactly to the family ETOL (for EDTOL and ETOL, see [25]). In this paper we shall study the general family of extended basic macro languages (EB), and establish the relation of this family to several classes of stack languages. Another 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 programming languages syntax which were originally described by general OI macro grammars.

The machine-approach in studying extended basic macro grammars was inspired by a machine characterization of ETOL (or: extended linear basic macro languages) presented in [37]. The original model of a cs-pd machine described in [37] (see Fig. 1.a) has a checking stack (left) and a pushdown store (right) operating in synchronous order. (For the notion of a checking stack, see [21]). The machine emerged from a theoretical model

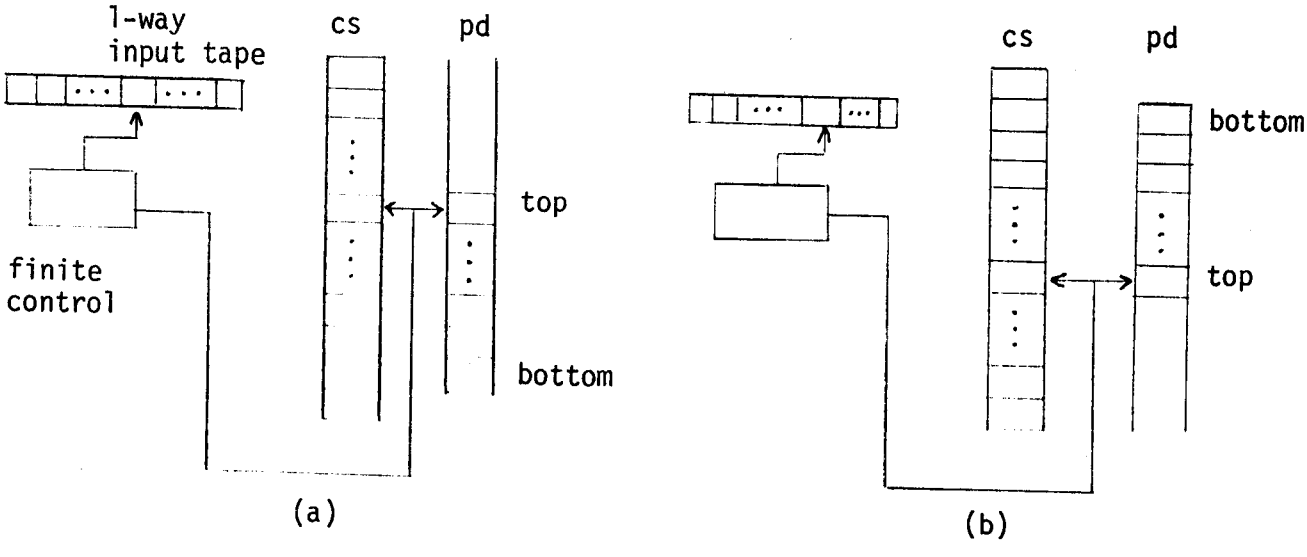


Fig. 1

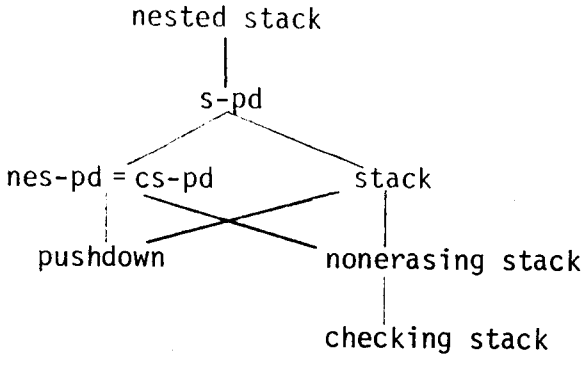
for studying Dijkstra's DO-construct (see [5,6]) as a single control-structure in programming, and it proved useful in obtaining a uniform characterization of certain hierarchies of complexity classes. It can be shown (by simulating a Post-machine [28]) that a similar machine model with a nonerasing stack replacing the checking stack can accept all recursively enumerable languages. Hence there seems to be no way of meaningfully generalizing this model in this direction. If we turn the pushdown store "upside down" (Fig. 1.b) and attach its bottom to the top of the checking stack, then we have a natural equivalent model which can be generalized. With this alternative description it has become easier to see how the machine is a (very) restricted version of the nested stack automaton.

The main object of this paper will be the study of the generalized model: the s-pd machine. 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 still require that the moves on the pushdown are synchronized with the stackpointer, whenever the machine enters a stack-reading mode (with the pushdown growing "downward"). We shall prove that the s-pd machines accept exactly the EB languages, generated by extended basic macro grammars.

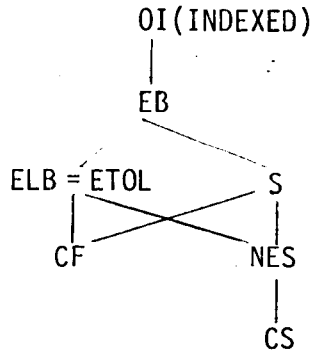
Using the machines Filé and van Leeuwen [14] have recently obtained elegant characterizations of ETOL and EB by classes of indexed grammars. It turns out that EB is precisely the family of languages generated by "restricted indexed grammars" (Aho [1,2]), which were originally unidentified in terms of macrogrammars.

From the machine characterization it follows further that the nonerasing 1-way stack languages are included in ETOL [37] and that the general 1-way stack languages are in EB (where obviously ETOL $\subseteq$ EB). We prove that ETOL and the 1-way stack languages 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 setparameters. (See [12] for indications that EB is a rather narrow strict subfamily of the OI macro languages). The relationships are summarized in Fig. 2.

If we classify the allowable operations on a stack in the following way: top operations (push,pop) and interior operations (movedown, moveup), then we can observe the following from Fig. 2. The incomparability of ETOL



(a) storage structure



(b) language families (if possible by grammar name)

Fig. 2

with S shows that it is impossible to separate the top operations from the interior operations by dividing the stack into two separate tracks, 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 (even when 'push' is added to the latter). Both results together illustrate the power of the pop operation in stack machines.

The remaining part of this paper consists of sections 2-5 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. In section 5 we put certain deterministic restrictions on the s-pd machine and obtain 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

Our notations and terminology will follow standard texts in automata - and language-theory [26,35]. The reader is assumed to be familiar with the main concepts concerning AFL-theory [17,20], stack automata [26], and the theory of parallel rewriting [25].

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

An OI macro grammar G (for a formal definition see [16]; cf. [13]) consists of an alphabet N of nonterminals (macro names, each with a specific number of arguments or rank), an alphabet Σ of terminals, an initial nonterminal S (the initial macro of rank 0), and a finite set P of rules (or macro definitions). Each macro definition is of the form $F(x_1,\ldots,x_n)\rightarrow\theta$, where θ is a well-formed term composed of elements from $\{x_1,\ldots,x_n\}\cup\Sigma$ and macro names by substitution and concatenation. Formally, a term is either (1) an atomic term, i.e. an element of $\{x_1,x_2,\ldots\}\cup\Sigma\cup\{\lambda\}$, or (2) of the form $H(t_1,\ldots,t_m)$, where H is any macro name of rank m and $t_1,\ldots,t_m$ are terms, or (3) of the form t_1t_2, where t_1 and t_2 are terms. Macros are expanded with outermost calls first, 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 any rule can have macro calls inside other macro calls (i.e. they are "nonnesting"), in linear basic macro grammars each term θ can have at most one macro call. The classes of OI, basic and linear basic macro languages are denoted by OI, B and LB respectively.

Macro grammars are a powerful language-generating device. Through proper parameter-passing one can copy (sub)strings and build several substrings at the same time in a controlled manner, thus largely extending the

original power of context-free grammars. An extended macro grammar (cf. [4]) permits the use of $\emptyset$ (denoting the empty set) and finite unions (denoted by +) in the macro definitions. Parameters become set-parameters, which can denote arbitrary finite sets of stringvalues (as opposed to singletons) during derivations. Macro expansion proceeds as usual by substituting the actual parameters for the formal ones in one of the righthand sides of a macro definition (with the standard notion of substitution of languages). Formally, an extended macro grammar is obtained by extending the former definition of terms as follows:
(1') each element of $\{x_1,x_2,\ldots\} \cup \Sigma \cup \{\lambda,\emptyset\}$ is an atomic term, and
(3') if t_1 and t_2 are terms, then so are $(t_1 + t_2)$ and t_1t_2.
Instead of formalizing the notion of derivation for these extended grammars explicitly we consider them as normal macro grammars in which + is viewed as a macro of rank 2 (written infix) with rules $+(x,y) \to x$ and $+(x,y) \to y$, and $\emptyset$ as a macro of rank 0 without rules.

Whereas the extension by set-parameters does not increase the generating power of arbitrary OI macro grammars, it does increase the power of nonnesting macro grammars. We define the extended basic (abbreviated as EB) and the extended linear basic (ELB) macro grammars to be those extended macro grammars that are basic and linear basic respectively when viewed as nonextended macro grammars with "terminals" + and $\emptyset$.

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 the EB grammar to store the declared identifiers of a same type. This can be seen from the following ELB grammar for the language $\{u_1 \# u_2 \# \ldots \# u_n \# u \mid n \geq 1$ and $u \in \{u_1,\ldots,u_n\}\}$:

$S \to F(\emptyset)$,

$F(x) \to G(x,\lambda)$,

$F(x) \to x$,

$G(x,y) \to aG(x,ya)$ for all $a \in \Sigma$,

$G(x,y) \to \#F(x+y)$.

A derivation of the string ab#a#ab is

$S \Rightarrow F(\emptyset) \Rightarrow G(\emptyset,\lambda) \Rightarrow aG(\emptyset,a) \Rightarrow abG(\emptyset,ab) \Rightarrow$

$\Rightarrow ab\#F(\emptyset + ab) \Rightarrow ab\#G(\emptyset + ab,\lambda) \Rightarrow$

$\Rightarrow ab\#aG(\emptyset + ab,a) \Rightarrow ab\#a\#F((\emptyset + ab) + a) \Rightarrow$

$\Rightarrow ab\#a\#((\emptyset + ab) + a) \Rightarrow ab\#a\#(\emptyset + ab) \Rightarrow$

$\Rightarrow ab\#a\#ab.$ □

A further extension of 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 also (provided t is), and a derivation is defined 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 of this notational variant in later proofs. The following lemma shows that one can eliminate the * and reformulate regular extended basic (REB) and regular extended linear basic (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 $\theta_1,\theta_2,E_1,\ldots,E_m$ are regular expressions over $\{x_1,\ldots,x_n\} \cup \Sigma$. We may assume without loss of generality that $\theta_1 = \theta_2 = \lambda$ (see the construction in the beginning of the proof of Lemma 4.1).

We shall replace this 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 only a finite number of strings from the 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 grammar and the continuity of the operation of language substitution [13].

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

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

and define new ELB macros $G_i^p(y,x_1,\ldots,x_n,y_1,\ldots,y_m)$, for $p \in Q$ and $1 \le i \le m$, as follows. The macro 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.

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

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], see Fig. 1. In our present treatment we shall assume that the pushdown store is initiated at the top of the checking stack, growing and shrinking in synchronous order with the movements of the stack-pointer, downwards and upwards respectively.

The model of an <u>s-pd machine</u> (Fig. 3) is obtained from a stack automaton by adding a pushdown store which is rooted at the top of the stack and whose top follows the movements of the stack-pointer. This synchronization means that the machine can change the contents of the stack only when the pushdown is empty and, conversely, if the pushdown is active then the stack-contents can be read only and not altered. The basic model of an s-pd machine is nondeterministic, with acceptance by final state (as usual). The machine starts with empty stack and pushdown. Particular s-pd machines will be specified later by writing (nondeterministic) programs in a symbolic language of instructions, tests, and standard identifiers which can be

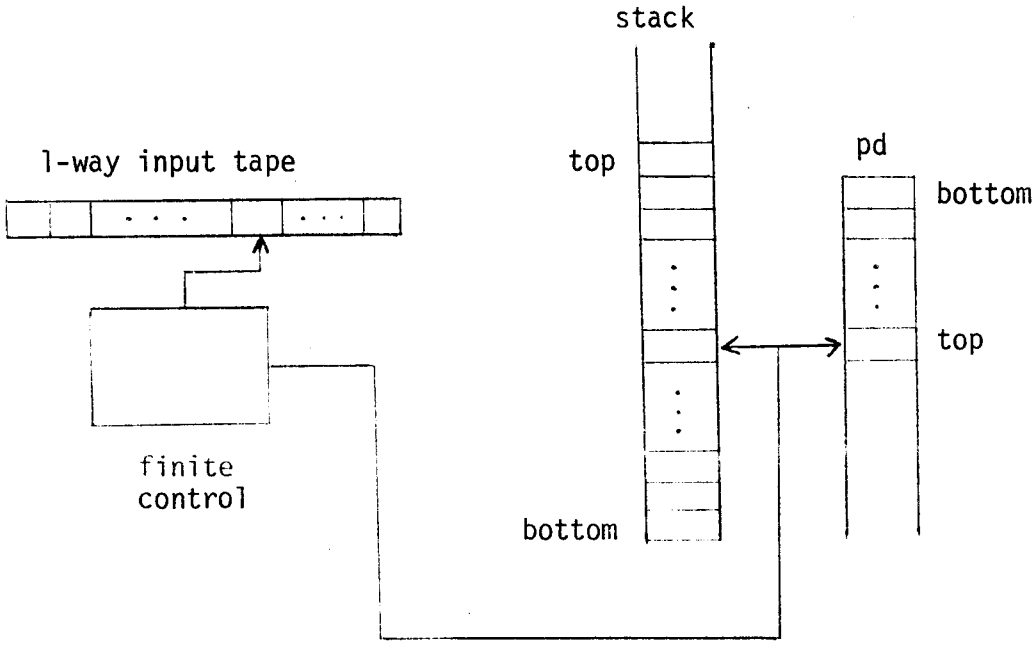


Fig. 3

implemented for s-pd (or cs-pd) machines in a straightforward manner (Section 4). The definition of s-pd machines as an x-tuple would not aid in the understanding of the model and is left to the interested reader.

By restricting the stack to be nonerasing or checking respectively we obtain the nes-pd and cs-pd machine models. A "program" for the nes-pd machine is not allowed to use the pop-instruction for its stack. A "program" for the cs-pd machine has to start execution with a number of push-instructions for its stack, but afterwards is not allowed to use any push- or pop-instruction anymore. The 1-way stack, nonerasing stack, and checking stack automata are obtained by dropping the pushdown facility from the extended machines. We shall use capital letters to denote the class of languages accepted by a machine whose "type" is written in equivalent small letters.

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; Theorem 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 a nes-pd machine M on a cs-pd machine M_1. 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.

When M_1 switches to checking mode, it returns to the bottom of the stack to begin a simulation of M while filling up the pushdown with dummy ¢'s. Simulating M, M_1 now verifies that its stack contains the symbols M would have written (meanwhile popping ¢'s off the pushdown) until M wants to stop pushing temporarily. If M is about to enter its stack for a read-excursion, then M_1 marks the current pushdown top with a \$ and uses the part of the pushdown from the \$-marked square on downwards to simulate M's instant 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. If M continues pushing, then M_1 removes the marker and verifies the next symbols on its stack. This repeats until M stops. □

Finally we give a brief description of ETOL grammars (see [33] or [25] for all further details). An ETOL-grammar [33] is a structure $G = \langle V, \Sigma, \{\tau_1, \ldots, \tau_n\}, S\rangle$ with V, Σ and S as usual 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 ETOL language. If the $\tau_1, \ldots, \tau_n$ are homomorphisms, then the resulting language is called an EDTOL language.

The relevance of ETOL languages for our discussion follows from the equalities ETOL = CS-PD [37], ETOL = ELB, and EDTOL = LB [4]. An alternative proof of the first two equalities will be given in section 4.

3. ETOL and one-way stack automata

It immediately follows from Theorem 2.3 and the equation ETOL = CS-PD [37] that the nonerasing stack languages are included in ETOL [37]. Strict inclusion follows because $\{a^n b^{n^2} c^n \mid n \geq 1\} \in$ ETOL but $\notin$ S (by a result of [30]). In other words, each nonerasing stack language can be defined by an extended linear basic macro grammar. We shall prove in this section that this is not true for all one-way stack languages (cf. [12]). From the results in the next section it will follow that each stack language can be defined by an extended basic macro grammar.

We need some additional preliminaries. Let L be a language over 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 [32,10]. 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 [36,14].

Lemma 3.1. (i) If $L \in$ ETOL has property P_3, then $L \in$ EDTOL.

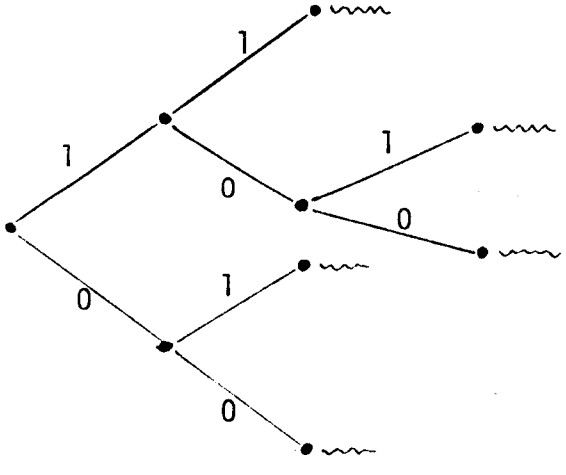
(ii) If $L \in yD_1$ has property P_3, then $L \in ydetD_1$.

We shall 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 (Fig. 4a). 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 Fig. 4a is given in Fig. 4b. 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 $1 \le i \le n$.



$\langle 00,01,100,101,11\rangle$

the infinite binary tree (a)

a cut (b)

Fig. 4

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

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

Note that for a string $s = aw_1 0 b w_1 1 \ldots a w_n 0 b w_n 1 \in L_0$ 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:

$S \to F(\lambda)$,

$F(x) \to F(x0)F(x1)$,

$F(x) \to axObx1$.

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

Lemma 3.3. $L_0 \in S$.

Proof: We shall program a one-way stack automaton which generates the consecutive nodes of a cut in its stack (one after the other). The stack automaton starts off with an arbitrary number of 0's in its stack. To read in its stack it will use a simple subroutine VERIFY, which will be called only when the stackpointer 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 the next part of the input. After a successful match the stackpointer is back at the top and the inputhead is pointing to the code 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 to read an input symbol x and move the inputhead one square to the right, rejecting if no symbol x was read otherwise.

The "routine text" for VERIFY is:

```
begin
    while not bottom stack do move stackpointer down od;
    while not top stack
        do read (stacksymbol); moveup od;
    read (stacksymbol)
end VERIFY;
```

We also need a subroutine PUSHZEROS which (nondeterministically) pushes zero or more 0's on top of the stack once it is called. It is given by

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

The complete program starts with an empty stack and can be described as follows (it uses the fact that two nodes w_1 and w_2 can be consecutive nodes in a cut iff there is a w such that $w_1 \in w01^*$ and $w_2 \in w10^*$):

```
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 the correctness of this program. ||

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 $L_0 \notin \text{ydet}D_1$, which will be shown using a pumping lemma for $\text{ydet}D_1$ 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 $u = v$ (similarly as in case (1)). This proves that L_0 has property P_3.

We now show that $L_0 \notin \text{ydet}D_1$. In [31] an intercalation lemma for tree transducer languages is proved that in a straightforward way gives rise to the following intercalation lemma for $\text{ydet}D_1$: for each $L \in \text{ydet}D_1$ there is an integer p such that every u in L longer than p can be written as $u = u_1 u_2 \ldots u_k$ 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 $|v_1 \ldots v_k| > N$, $v_1 \ldots v_k \in L$ and, for $1 \leq i \leq k$, $\text{alph}(v_i) = \text{alph}(u_i)$ (where, for a string s, $\text{alph}(s)$ denotes the set of symbols occurring in s).

Assume that $L_0 \in ydetD_1$. By the above result every long string in L_0 has small substrings which can be pumped up while staying on the same alph without leaving L_0. Consider a string $u = aw_1 0 bw_1 1 \ldots aw_n 0 bw_n 1$ in L_0 with $|w_i| \geq p$ for $1 \leq i \leq n$, and let $u = u_1 \ldots u_k$ as in the above lemma. No u_i contains both symbols a and b because of the length restriction. If we pump any u_i to 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 (viz. 2n). □

We conclude from Lemmas 3.3 and 3.4 the following (using that we know $0I - yD_1 \neq 0$ for the second part of the result)

Theorem 3.5. ETOL and S are incomparable, and so are yD_1 and 0I.

The last part of this result settles an open problem in [14], where the existence of a language in $ydetD_1 - 0I$ was shown.

4. Extended basic macro grammars and stack machines

In this section we obtain a machine characterization of the family of languages definable by macro grammars with set-parameters without nested macro calls. We shall prove that these extended basic macro languages coincide with the family of languages accepted by s-pd machines. The result immediately shows that $S \subseteq EB$, and it demonstrates what power is cut off from arbitrary OI macro grammars by the constraint of no nested calls (the s-pd machine is much simpler than the 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].

All proofs will make use of the new machine-models. In order to describe particular 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 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 $F(x_1,\ldots,x_n)\rightarrow F'(\theta_1,\ldots,\theta_m)$ or $F(x_1,\ldots,x_n)\rightarrow\theta$, where $\theta,\theta_1,\ldots,\theta_m$ are atomic terms and θ does not contain + or $\emptyset$. Otherwise (cf. [4]), replace rules $F(\ldots)\rightarrow\psi_1F'(\ldots)\psi_2$ and $F(\ldots)\rightarrow\psi$ by $F(x,\ldots,y)\rightarrow F'(x\psi_1,\ldots,\psi_2y)$ and $F(x,\ldots,y)\rightarrow Z(x\psi y)$ 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 $Z(x)\rightarrow x$.

We shall first write a recursive program to recognize G's language using a checking stack for storage, and then show how to implement the recursion using the extra pd-facility of the cs-pd machine.

The cs machine nondeterministically generates a complete symbolic expansion of successive macros (see Fig. 5): $F_0\rightarrow F_1(\theta_1^1,\ldots,\theta_{k_1}^1)$, $F_1(x_1,\ldots,x_{k_1})\rightarrow F_2(\ldots),\ldots,F_i(x_1,\ldots,x_{k_i})\rightarrow F_{i+1}(\ldots),\ldots,F_n(x_1,\ldots,x_{k_n})\rightarrow\theta$ (which are rules of G, with F_0 the initial nonterminal). Thus, the checking stack symbols are (codes for) the righthand 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

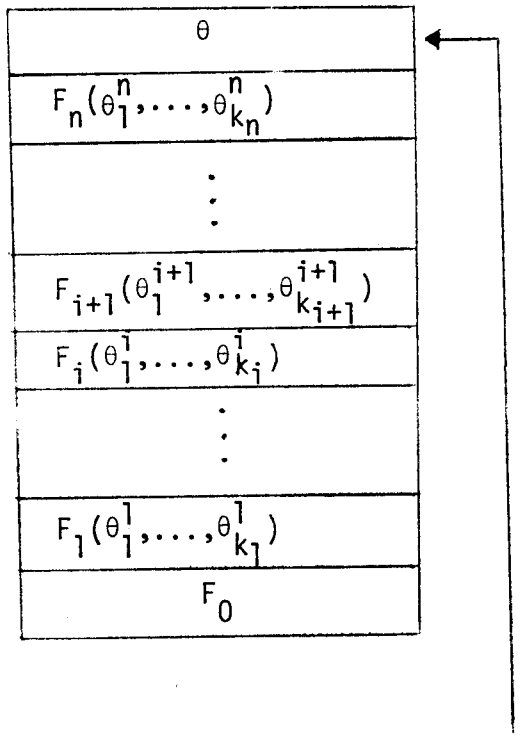


Fig. 5

the macro expansion. The (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 recursive 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 level in the stack, which is the current level of macro expansion.

We program EVAL as follows. By head(θ) and tail(θ) we denote the first symbol of θ and the string obtained from θ by erasing head(θ), respectively; by a_i we denote the i^{th} terminal symbol.

```
procedure EVAL(θ);
if  θ ≠ λ  then
     case head(θ) of
          a_i: read(a_i);
          x_j: begin ψ:= any element of the set denoted by the j^th
                         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 this program it is understood that the machine rejects if there is no value for ψ, i.e., if the set denoted by the j^{th} 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 synchronized pushdown store in addition to 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 its argument is longer than the righthand side of a rule. Hence the pd 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 righthand side of a rule for F) od;
        movedown (stacksymbol);
        EVAL
end;
```

where EVAL denotes the following routine text (without a parameter this time because we keep an explicit pushdown where the argument is stored):

```
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 set denoted by the j^th
                            argument of stacksymbol; movedown(ψ)
                    end
              esac fi;
          goto loop
end EVAL;
```

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

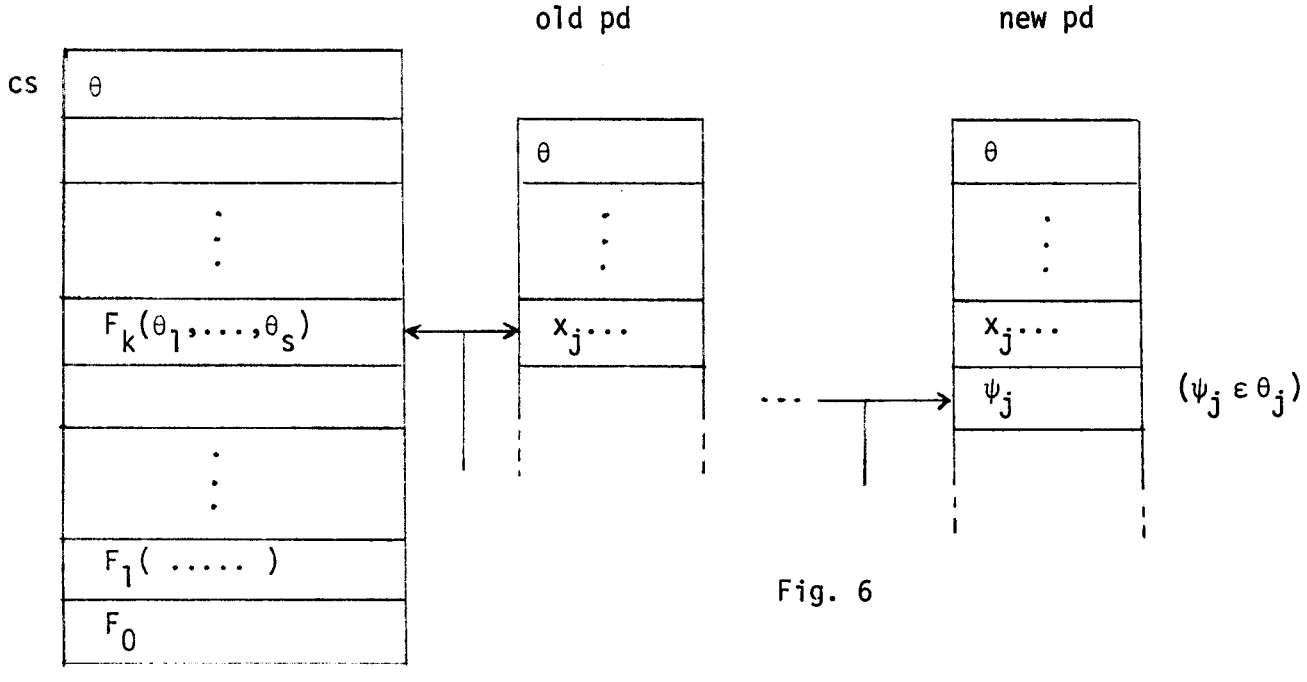


Fig. 6

As an example we consider the grammar with rules

$S \to F(\lambda,\emptyset)$,

$F(z,x) \to G(z,x,\lambda)$,

$F(z,x) \to zx$,

$G(z,x,y) \to G(za,x,ya)$ for all $a \in \Sigma$,

$G(z,x,y) \to F(z\#,x+y)$,

which generates the language of Example 2.1. Snapshots from the recognition of the string a#b#a on a cs-pd machine are given in Fig. 7.

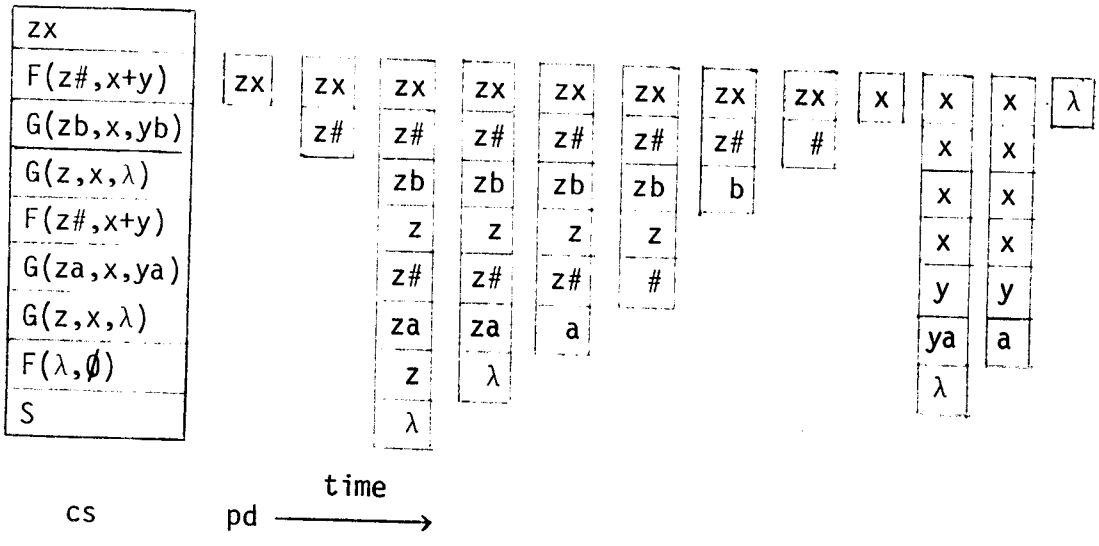


Fig. 7

We continue our proof 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 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 righthand sides of rules and their suffixes.

We may assume that the symbol + occurs only in the arguments of macro calls. Thus each righthand side θ of a rule is of the form $\theta = \theta_1\theta_2 \ldots \theta_k$ 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 $\theta_2 \ldots \theta_n$ 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:

"ψ:= any element of the set denoted by the j^{th} 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 above program indeed recognizes the EB language on the 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; chapter 5]): each family of languages defined by a class of "well-behaving" 1-way nondeterministic acceptors (of the 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.

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 show closure under regular substitutions f one replaces in a given EB grammar each terminal symbol "a" by some regular expression for f(a). This gives an REB grammar which can be transformed into an EB grammar by Lemma 2.2(1). Closure under intersection with regular sets can be shown as follows (cf. [16,4]).

Let an EB grammar G be given with terminal alphabet Σ and rules of the form $F(x_1,\ldots,x_n) \to G_1(\ldots)G_2(\ldots)\ldots G_k(\ldots)$ 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 $f \in H$, we 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 $F(x_1,\ldots,x_n) \to G_1(\ldots)\ldots G_k(\ldots)$ is changed into all rules of the form

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

such that $f_1 f_2 \ldots f_k = f$ 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 i^{th} argument of G_m in the righthand 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 $[F,f](\ldots) \to \theta'$, 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 ⊆ ETOL

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 are 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 is a full AFL generator of S-PD. By a standard equal-length coding we may assume that the stack/pushdown alphabet of the s-pd machine is $\{0,1\}$. In order to obtain a manageable L we reformulate the s-pd machine to have 'stack empty' as the only basic test, and the following basic instructions (apart from the read instruction):

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 : move up to the stacksymbol a and pop γ off the pushdown

for $a,\gamma \in \{0,1\}$.

It should be obvious that the new instructions can be simulated by the old ones, and vice versa. Checking the complicated definitions in [17; sections 5.2, 5.3] shows that s-pd machines form a reduced, finitely encoded AFA satisfying [17; Theorem 5.3.2]. Thus S-PD is indeed 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.

1,2. $S \to aF(\lambda)a^ES, \quad a = 0,a = 1$

3. $S \to \lambda,$

4,5. $F(x) \to xaF((a^D_0xa^U_0 + a^D_1xa^U_1)^*)a^EF(x), \quad a = 0,1$

6. $F(x) \to x.$

The set-parameter x stands for the set of all sequences of a^D_γ and a^U_γ instructions that 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 righthand 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 correctness of the grammar. By Lemma 2.2(1) one may convert the REB grammar into an 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 $\subseteq$ ETOL is quite 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 a certain stack s, w can be executed starting and ending at the top of s. This language is generated by the following RELB grammar:

1. $S \to F(\lambda)$,

2,3. $F(x) \to F((a_0^D x a_0^U + a_1^D x a_1^U)^*)$, $a = 0,1$,

4. $F(x) \to x$.

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

[]

Combining the lemmas we obtain the main result of this section.

Theorem 4.4. EB = S-PD. ||

We also conclude the following (known) characterizations of ELB.

Theorem 4.5. ELB = ETOL = CS-PD = NES-PD.

Proof: CS-PD = NES-PD was shown in Theorem 2.3. ELB $\subseteq$ CS-PD $\subseteq$ ETOL was shown in the previous two lemmas. The proof of ETOL $\subseteq$ ELB is straightforward [4]: the ELB grammar has, in addition to the initial macro S, only one macro F; a finite substitution f of the ETOL grammar is simulated by a

rule $F(x_1,\ldots,x_n) \to F(f(x_1),\ldots,f(x_n))$ of the ELB grammar, where $x_1,\ldots,x_n$ are renamings of the symbols $a_1,\ldots,a_n$ of the ETOL grammar ($f(x_i)$ denotes the renaming of $f(a_i)$); the final rule is $F(x_1,\ldots,x_n) \to x_1$ (if a_1 is the initial symbol of the ETOL 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 $\delta_i = \emptyset$ 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, and 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) $ETOL \not\subseteq EB$

4.6 (i) shows that each stack language can be defined by an extended basic macro grammar, but not vice versa. It is open whether there exists a natural restriction on EB grammars which characterizes S.

4.6 (ii) seems to be at present the strongest ramification of the hard result that ETOL $\subsetneq$ INDEXED [8]. We recall that EB $\subsetneq$ OI [12].

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 generator for S-PD was defined by the grammar

$S \to aF(\lambda)a^E S,$

$S \to \lambda,$

$F(x) \to xaF((a_0^D x a_0^U + a_1^D x a_1^U)^*)a^E F(x),$

$F(x) \to x,\ a \in \{0,1\}.$

A generator for S is obtained by dropping the pd-facility, i.e., changing $(a_0^D x a_0^U + a_1^D x a_1^U)^*$ into $(a^D x a^U)^*$, and 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:

$S \to aF(\lambda),$

$S \to \lambda,$

$F(x) \to xaF((a_0^D x a_0^U + a_1^D x a_1^U)^*),$

$F(x) \to x.$

When we drop the pd-facility from this grammar (as for 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:

$S \to F(\lambda),$

$F(x) \to F((a^D x a^U)^*),\quad a \in \{0,1\}.$

$F(x) \to x.$

Digressing on EB (and remembering that OI = INDEXED [16,1]), we may ask how the restriction of nonnested nonterminals in EB macrogrammars perhaps corresponds to an equally natural restriction on indexed grammars. Filé 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 proper class of indexed grammars corresponding to EB is Aho's class of "restricted indexed grammars" (RIG, see [1, last page]), although in [15] a more attractive equivalent form is derived. An RIG is like an ordinary indexed grammar, except that nonterminals produced by flag-consuming

productions cannot themselves introduce new flags ever. (Such nonterminals were called "intermediates"). It is remarkable that the class of restricted indexed grammars appears to be "natural" after all, from the point of view of macro grammars.

We can show (cf. [15])

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 an immediate consequence of our machine characterization of EB. Cor. 4.6. confirms the conjecture in [2] about strict inclusion of S 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. It continues the work of M. Fischer [16] to give further useful characterizations of the family BASIC. The same restriction for cs-pd (equivalently, nes-pd) machines gives a machine model for the linear basic macro languages (or EDTOL languages). At the end of this section we position all families of this paper in a diagram and show the correctness of the incomparabilities and proper inclusions.

In order to explain the particular deterministic restriction for s-pd machines it is convenient to view s-pd machines as generators (i.e., machines with output) rather than as acceptors (i.e., 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. We say that an s-pd machine is stack-deterministic (as a generator) iff it is deterministic in the stack-reading mode, i.e., whenever it moves up and down the stack using the pd-facility or when it is on top of the stack and must choose between staying at the top or moving down into the stack it acts completely deterministically. Thus, actual nondeterminism is only allowed in the stack-writing mode. From the acceptor point of view it means that we put restrictions on the program (or state-transition function) 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.

We shall abbreviate "stack-deterministic s-pd" by ds-pd. The same restriction can be put on restricted versions of the s-pd generator (with

a similiar notation). In particular, a dcs-pd generator first builds a checking stack nondeterministically, then generates output while moving in its stack deterministically.

We now show that stack-determinism provides a characterization of the (linear) basic macro languages. Intuitively, deterministic handling of the stack corresponds to "deterministic arguments" in the macro 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 "ψ:= any element ..." into "ψ:= the element ...". Thus, by changing "read(a_i)" into "write(a_i)" throughout the parsing program, a stack-deterministic program for the s-pd or cs-pd generator 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 is known. The basic grammar corresponding to a ds-pd generator, however, will be somewhat similar to the EB grammar in the proof of Lemma 4.3.

Let a ds-pd generator be given in the usual way by a set of states Q, a state transition function, output alphabet Σ, pushdown alphabet Γ, etcetera. We assume that the machine accepts by final state and empty stack. Let \$ be a new symbol, used to indicate an empty pd. For each $q \in Q$ and $\gamma \in \Gamma \cup \{\$\}$ we introduce a formal parameter $x(q,\gamma)$, denoting 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 $Q \times (\Gamma \cup \{\$\}) \to Q$.

Suppose that for a given stack s with top symbol A the generator, starting in state r at the top of s with pd symbol γ in the "opposite" pd square (empty pd if $\gamma = \$$) and empty output tape, moves into the stack and returns after a number of steps to the top of s in state $f(r,\gamma)$ with output $w(r,\gamma) \in \Sigma^*$, such that it cannot move again into the stack. (Note that due to the stack-deterministic restriction $f(r,\gamma)$ and $w(r,\gamma)$ are unique).

Then we want $[A; p,q; f](\vec{w})$ to generate $v \in \Sigma^*$ if and only if the generator, when starting in state p at the top of s (with empty pd), can return in the same situation (in some other state) and then pop A and go into state q, producing output v (where we assume that s all the time remains as lower part of the current stack).

This idea can be implemented with the following rules.

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

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

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

(b) $u(r,\gamma)$ and $g(r,\gamma)$ are obtained by writing down the (symbolic) output of the machine, starting in state r at the top symbol

B of the stack with γ in the opposite pd square (empty pd if $\gamma = \$$).

For instance, if the machine moves down (to A) in state r_1 pushing $\gamma_1 \in \Gamma$ on the pd and producing output $v_1 \in \Sigma^*$, then we write $u(r,\gamma) = v_1 x(r_1,\gamma_1)\ldots$ and continue in state $f(r_1,\gamma_1)$. If the machine now moves up (to B) and down again into state r_2 pushing γ_2 on the pd and producing output v_2, then we write $u(r,\gamma) = v_1 x(r_1,\gamma_1) v_2 x(r_2,\gamma_2)\ldots$ and continue in state $f(r_2,\gamma_2)$. If, finally, the machine moves up to B in state r_n producing output v_n, and it cannot move down again, then we set $u(r,\gamma) = v_1 x(r_1,\gamma_1) v_2 x(r_2,\gamma_2)\ldots v_n$ and $g(r,\gamma) = r_n$. Note that each $x(r_i,\gamma_i)$ occurs at most once in $u(r,\gamma)$ since otherwise the machine loops. 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 details, and to prove the correctness of the construction formally. It shows that DS-PD $\subseteq$ B.

A dnes-pd generator can be viewed as a ds-pd generator 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 rules:

(1) $[A;\ p;\ f](\vec{x}) \to x(p,\$)$ if the machine, 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. ||

Both lemmas together give the machine characterization of the non-nesting macro grammars.

Theorem 5.3.

(i) B = DS-PD.

(ii) EDTOL = LB = DCS-PD = DNES-PD. □

The families of languages discussed in this paper can now be put together in the following inclusion diagram (Fig. 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"; downwards: add "D"; from the reader away: add "pd".

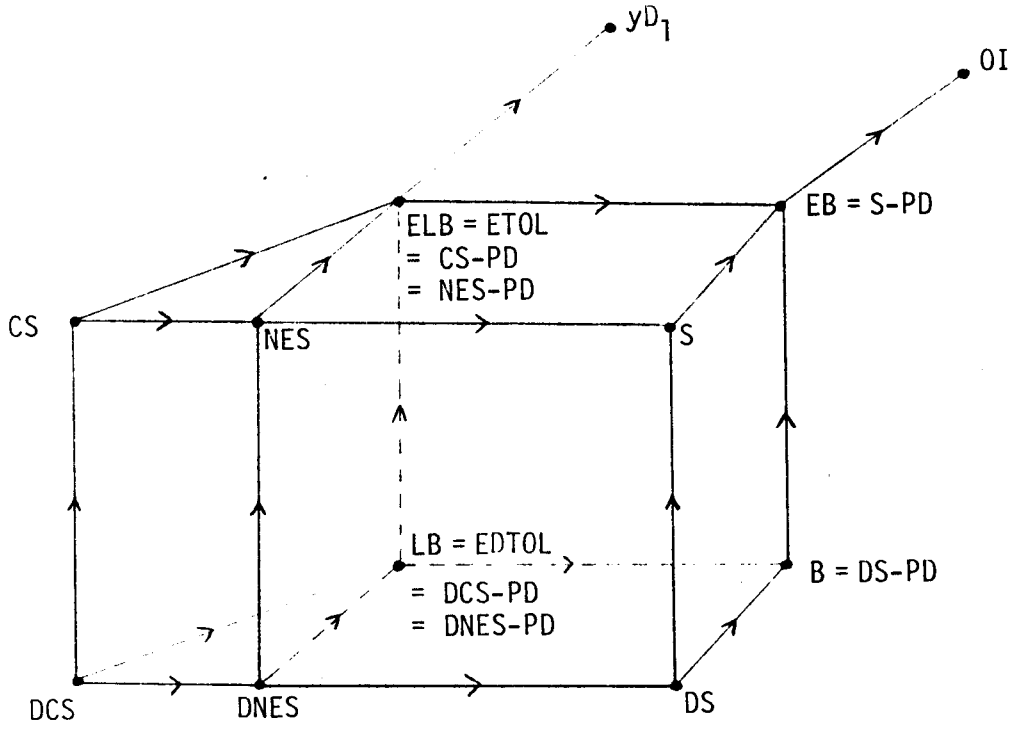


Fig. 8

We note that the family DCS is known from the literature. Since a checking stack (or cs-pd) machine, viewed as a generator, can be viewed as a transducer (with the checking stack contents as input), DCS is easily seen to be the image of the regular languages under two-way deter-

*Some of the lines are dotted only to improve perspective. All inclusions shown are proper.

ministic finite state transducers [27]. Note that a stack-deterministic cs-pd generator 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 fig. 8 and DCS are full AFL's. The other 4 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 Fig. 8 follows from the existence of languages in the following classes:

(1) CS-B: the language $\{w \in \{a,b\}^* | \text{the number of b's in } w \text{ is not prime}\}$ 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; section 3.4] showing the existence of a language in OI-IO proves in fact that if $L \subseteq b^*$ and $h^{-1}(L) \in IO$ (where $h(a) = \lambda$ and $h(b) = b$), 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 generator 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 $L = \{a^{n^2} | n \geq 1\}$ is in DNES, but $h^{-1}(L)$ is not in B. Thus, all families of languages in between these

two are not closed under h^{-1}.

The reader may wonder about the position of the class CF of context-free languages in fig. 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 $\subseteq$ S. Recently the (difficult) proof was given that CF is not included in EDTOL [7]. From this it can be shown as follows that CF is not included in NES (this was known to Greibach since the paper on CS [21] and has recently been proved by her [23]; she uses the argument below together with a direct proof that CF is not included in DCS (cf. [22])). Assume first that CF $\subseteq$ CS. Then in particular all parentheses languages [29] are in CS. Such languages do not contain infinite regular subsets. This property ensures that they are even 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 contradicts the result of [7]. Hence CF is not included in CS. From [21; Lemma 4.1] it can now be concluded that CF is not included in NES.

6. Conclusion

In this paper we presented a detailed study of several language classes closely related to EB, the family of languages generated by macrogrammars (with set-parameters) in which no macro calls within the parameters are allowed. We have presented a feasible protocol for implementing macro-expansions 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 restricted nested stack automata [2] in the following way. The pushdown store corresponds to a sequence of one-element stacks in the nested stack automaton, which it inserts "between" the symbols in the main stack as it moves down. The one-element stacks dissolve as the stack-pointer moves up, just like 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.

In this paper we have explained some of the similarities and differences between features of stack-like 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-like machines and the corresponding macro grammars.

Acknowledgements. We thank Peter Asveld, Gilberto Filè, and Giora Slutzki for useful discussions.

References

[1] A. V. Aho; Indexed grammars--an extension of context-free grammars; JACM 15 (1968), 647-671.

[2] A. V. Aho; Nested stack automata; JACM 16 (1969), 383-406.

[3] P. A. Christensen; Hyper-AFLs and ETOL systems; in "L Systems" (G. Rozenberg and A. Salomaa, eds), LNCS 15, 254-257, Springer-Verlag, Berlin, 1974.

[4] P. J. Downey; Formal languages and recursion schemes; Thesis, Harvard University, Report TR-16-74, 1974.

[5] E. W. Dijkstra; Guarded commands, nondeterminacy, and the formal derivation of programs; CACM 18 (1975), 453-457.

[6] E. W. Dijkstra; "A discipline of programming"; PH Series in Automatic Programming, Prentice-Hall, Englewood Cliffs, N. J., 1976.

[7] A. Ehrenfeucht and G. Rozenberg; On some context-free languages that are not deterministic ETOL languages; Report 77-03, Dept. of Mathematics, University of Antwerp, 1977.

[8] A. Ehrenfeucht, G. Rozenberg and S. Skyum; A relationship between ETOL and EDTOL languages; TCS 1 (1976), 325-330.

[9] R. W. Ehrich and S. S. Yau; Two-way sequential transductions and stack automata; Inf. and Control 18 (1971), 404-446.

[10] J. Engelfriet; Bottom-up and top-down tree transformations--a comparison; Math. Syst. Th. 9 (1975), 198-231.

[11] J. Engelfriet; Surface tree languages and parallel derivation trees; TCS 2 (1976), 9-27.

[12] J. Engelfriet; Macro grammars, Lindenmayer systems and other copying devices; 4th Int. Coll. on Automata, Languages and Programming, Turku, Finland, 1977.

[13] J. Engelfriet and E. Meineche Schmidt; IO and OI; DAIMI PB-47, Datalogisk Afdeling, Aarhus University, Denmark (to appear in JCSS).

[14] J. Engelfriet and S. Skyum; Copying theorems; Inf. Proc. Letters 4 (1976), 157-161.

[15] G. Filé and J. van Leeuwen; The characterization of some language families by classes of indexed grammars, Dept. of Computer Sci., The Pennsylvania State Univ., 1977 (forthcoming).

[16] M. J. Fischer; Grammars with macro-like productions; Ph.D. Thesis, Harvard University, 1968.

[17] S. Ginsburg; "Algebraic and automata-theoretic properties of formal languages"; Fundamental Studies in Computer Science 2, North-Holland/ American Elsevier, Amsterdam, 1975.

[18] S. Ginsburg, S. A. Greibach and M. A. Harrison; Stack automata and compiling; JACM 14 (1967), 172-201.

[19] S. Ginsburg, S. A. Greibach and M. A. Harrison; One-way stack automata; JACM 14 (1967), 389-418.

[20] S. Ginsburg, S. A. Greibach and J. E. Hopcroft; "Studies in Abstract Families of Languages", Memoirs of the AMS 87, 1969.

[21] S. Greibach; Checking automata and one-way stack languages; JCSS 3 (1969), 196-217.

[22] S. A. Greibach; One-way finite visit automata; University of California, unpublished report, 1977.

[23] S. A. Greibach; private communication.

[24] M. A. Harrison and M. Schkolnick; A grammatical characterization of one-way nondeterministic stack languages; JACM 18 (1971), 148-172.

[25] G. T. Herman and G. Rozenberg; "Developmental Systems and Languages"; North-Holland, Amsterdam, 1975.

[26] J. E. Hopcroft and J. D. Ullman; "Formal languages and their relation to automata", Addison-Wesley, Reading, Mass., USA, 1969.

[27] D. I. Kiel; Two-way a-transducers and AFL; JCSS 10 (1975), 88-109.

[28] Z. Manna; "Mathematical theory of computation", McGraw-Hill, New York, 1974.

[29] R. McNaughton; Parenthesis grammars; JACM 14 (1967), 490-500.

[30] W. Ogden; Intercalation theorems for stack languages; 1st ACM Symp. on Theory of Computing, 1969, pp. 31-42.

[31] C. R. Perrault; Intercalation lemmas for tree transducer languages; JCSS 13 (1976), 246-277.

[32] W. C. Rounds; Mappings and grammars on trees; Math. Syst. Th. 4 (1970), 257-287.

[33] G. Rozenberg; Extension of tabled OL-systems and languages; Int. J. Comp. Inform, Sci. 2 (1973), 311-336.

[34] G. Rozenberg and D. Vermeir; On ETOL systems of finite index; Report 75-13, University of Antwerp, 1975.

[35] A. Salomaa; "Formal languages", Academic Press, New York, 1973.

[36] S. Skyum; Decomposition theorems for various kinds of languages parallel in nature; SIAM J. Comp. 5 (1976), 284-296.

[37] J. van Leeuwen; Variations of a new machine model; 17th Ann. IEEE Symp. on Foundations of Computer Science, Houston, 1976.