MATHEMATICAL CENTRE TRACTS 108

FOUNDATIONS OF COMPUTER SCIENCE III

PART 1 : AUTOMATA, DATA STRUCTURES, COMPLEXITY

J.W. DE BAKKER (ed.)
J. VAN LEEUWEN (ed.)

MATHEMATISCH CENTRUM AMSTERDAM 1979

1980 Mathematics subject classifications: 68B15,68C25,68C40,68D05,68D025,68F05,68F10

ACM-Computing Review-categories 4.34, 5.22, 5.23, 5.25, 5.26

ISBN 90 6196 176 9

CONTENTS

PREFACE

The 3rd Advanced Course on the Foundations of Computer Science was held from August 21 to September 1, 1978, in Amsterdam as part of an international program of Advanced Courses sponsored by the CREST Subcommittee on Training in Data Processing of the Commission of the European Communities.

The Advanced Courses on the Foundations of Computer Science are organized to provide an opportunity for computer science graduates and professionals to learn about the modern developments in theoretical computer science at a high level. In 1978 semantics, analysis/complexity of algorithms and automata theory were chosen as major fields of attention. Eight distinguished lecturers were invited to present a series of six lectures each on leading issues and new results in their current field of specialty.

These volumes contain the (edited) text of most lectures given on the occasion of the 3rd Advanced Course. The material, written especially for the Course, usually presents an original view of an entire research area which is not available in this form yet from textbooks for classroom use. We believe that the chapters will serve as a valuable source of material for high-level seminars in theoretical computer science. The lectures of M.O. Rabin ("Probabilistic Algorithms") and of C-P. Schnorr ("Elementary Methods in Algebraic Complexity Theory") will be published elsewhere and are not contained in these volumes.

We thank the lecturers for their superb contributions and the participants for being a most receptive audience. We are very grateful to the Dutch Ministry of Education and the Commission of the European Communities, which together provided the necessary funds for organising the Course. Finally, we thank Mrs. S.J. Kuipers-Hoekstra for her invaluable assistance throughout the organization of the Course and the Publication Service of the Mathematical Centre for the technical realization of these volumes.

J.W. de Bakker - J. van Leeuwen
Directors of the Course

TWO-WAY AUTOMATA AND CHECKING AUTOMATA by J. ENGELFRIET

MATHEMATICAL CENTRE TRACTS 108 (1979), 1-69

TWO-WAY AUTOMATA AND CHECKING AUTOMATA

J. ENGELFRIET

Technological University, Twente, the Netherlands

INTRODUCTION

A 2-*way automaton*(*) has a two-way read-only input tape, i.e. its input reading head may move in all possible directions from one square to another (neighbouring) one. This facility to reinspect its input without having to store it in memory is what makes the 2-way automaton in general stronger than the corresponding 1-way automaton. Figure 1 shows a general 2-way transducer consisting of a finite control, a 2-way input tape, a 1-way output tape and a storage X (specified by a storage space and how to manipulate it). Apart from being investigated as an interesting subject on its own [RabSco; SteHarL; AhoUll 1], two-way automata have been defined to model certain parts of compilers [GinGreH; AhoUll 2], turned up in complexity theory [Coo 1; Coo 2; Iba] and were recently used in the study of bounded crossing Turing machines [Raj 1; Gre 1; Gre 2].

A *checking automaton* has a 2-way read-only tape which is part of its memory and should be filled before computation. Such a tape is called a checking stack and is therefore usually drawn vertically in pictures. Figure 2 shows a general checking acceptor with a finite control, an input tape, and a storage consisting of a checking stack and additional storage X. The automaton accepts the input if at least one choice of filling the checking stack leads to a successful computation. In other words, viewing the checking stack as a second input tape, the automaton accepts a binary relation of which only the domain (or range) is considered. Thus the checking stack facility incorporates a large amount of nondeterminism: the automaton can e.g. guess (or ask for) an encoding of a computation of some other

---

(*) We use "automaton" here to mean either an acceptor or a transducer; later it will also be used to mean acceptor only.

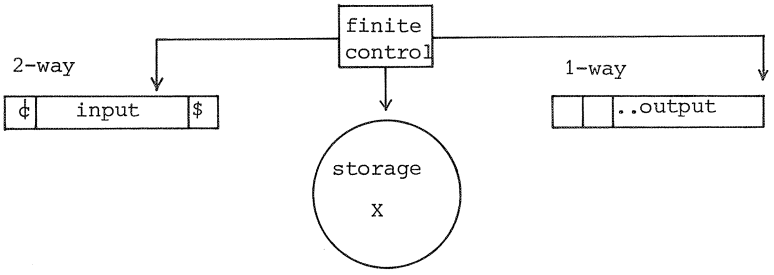


Figure 1. A general 2-way transducer.

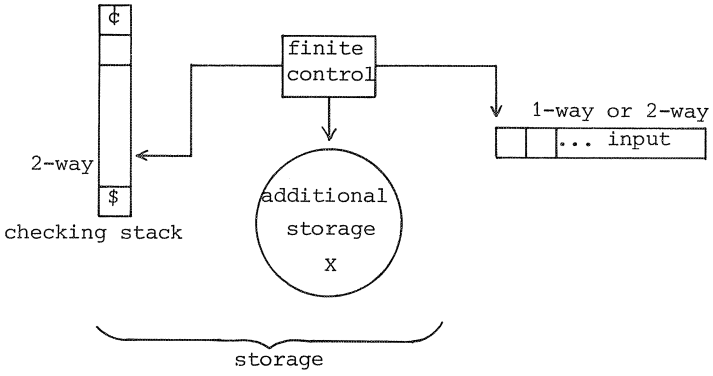


Figure 2. A general checking acceptor.

machine or grammar, check that it is indeed such an encoding, and check whether its guess was correct, i.e. whether the machine (or grammar) accepts (or produces) the input string according to this computation. Checking automata were defined as a useful tool in the analysis of 1-way stack automata [Gre 3], were used to characterize space complexity classes [Fis 1; Iba] and recently turned out to be strongly related to parallel rewriting systems [Raj 2; vLe].

Thus a 2-way read-only tape is the common feature of 2-way automata and checking automata. The aim of these notes is to provide the reader with some insight into the properties of such "automata with a 2-way read-only tape" and the theoretical tools to handle them. Both "classical" and more recent results will be discussed.

We will try to present most results from the point of view of two basic concepts: visiting (or crossing) sequences and transition tables. A visiting sequence contains all moves of the automaton on one particular square of the 2-way tape during some computation, whereas a (state) transition table contains for each given starting state the corresponding final state after the computation of the automaton on (part of) its 2-way tape. Thus they contain local and global information (respectively) concerning the 2-way computation of the automaton.

We will consider several specific types of automata, such as 2-way finite state transducers and 1-way checking-stack pushdown acceptors, but we will also be interested in obtaining general results on 2-way and checking automata, independent of their (additional) storage structure. An example of the latter, which should be obvious from the above discussion and Figures 1 and 2, is the following (where X is some storage structure): the class of languages generated as output by nondeterministic 2-way X transducers is equal to the class of languages accepted by nondeterministic 1-way [X + checking-stack] acceptors. This relationship will be used systematically in the sequel. (In the deterministic case there is no immediate connection between the two classes.)

These notes are divided into three parts. In the first part we consider the simplest 2-way device: the 2-way finite state transducer. The domains of these transducers are regular; we will therefore concentrate on the mappings and output languages they define. Note that the output languages are those accepted by the (ordinary) 1-way checking stack acceptor [Gre 3]. Since the finite control of any 2-way automaton consists essentially of a 2-way finite state transducer (with instructions to manipulate storage as output), one can expect to obtain general properties of 2-way automata from those of the 2-way finite state transducer: this is the philosophy of general automata theory in a nutshell [Gin; HopUll 1; Sco; AhoHopU 1].

In the second part we consider two types of grammars related to the 2-way finite state transducer: parallel grammars and register grammars. They represent two different ways of capturing the 2-way motion of the automaton in a 1-way generation process. Straightforward generalization of these grammars leads to syntax-directed translations (or, top-down tree transducers) and macro grammars. The recursion present in these generalizations can be handled by a stronger 2-way transducer: a restricted type of 2-way pushdown transducer (and the corresponding [checking-stack + pushdown] acceptor). Variations of this automaton provide a uniform description of classes of

languages generated by the above types of rewriting systems.

In the third part we discuss the relationship of 2-way and checking automata to space and time complexity of Turing machines.

1. TWO-WAY FINITE STATE TRANSDUCERS.

The simplest machine with a 2-way read-only tape is the 2-way finite state automaton [RabSco]. Addition of a 1-way output tape gives the 2-way finite state transducer [AhoUll 1], which is also called 2-way generalized sequential machine (2gsm) or 2-way a-transducer. Figure 1 shows a 2gsm when the storage X is disregarded. Informally, a 2gsm consists of

- an input tape which is an array input[0:n+1] of symbols with input[0] = ¢ and input[n+1] = \$, where n is the length of the input string,
- a read head (or input pointer) which is a variable x of type [0: n + 1],
- an output tape of the usual kind, and
- a finite control which is a nondeterministic flowchart build up from elementary operations x := x+1 (move right), x := x-1 (move left), print(σ) for each output symbol σ, and elementary tests input[x] = σ for each input symbol σ.

Formally a *two-way nondeterministic finite state transducer* or *2gsm* is a tuple $M = (Q,\Sigma,\Delta,\delta,q_0,F)$ of states, input symbols, output symbols, transition function, initial state and final states respectively. δ is a function from $Q \times (\Sigma \cup \{¢,\$\})$ into the finite subsets of $Q \times \{-1,0,+1\} \times \Delta^*$. The interpretation of $(q',d,u) \in \delta(q,\sigma)$ is that M in state q and reading input symbol σ may go into state q', move its read head d squares to the right and produce output u. A configuration of M is of the form (q,¢w\$,i,v) indicating the state, the content of the input tape, the position of the input head ($0 \leq i \leq |w| + 1$) and the content of the output tape. M realizes a translation (or mapping) from Σ^* to Δ^*, also denoted by M, viz.
$M = \{(w,v) \mid (q_0,¢w\$,0,\lambda) \vdash^* (q,¢w\$,i,v)$ for some $q \in F$ and $i \in \{0,n+1\}\}$, where $\vdash^*$ is the usual computation relation induced by δ. Thus the computation starts in the initial state with the read head on ¢ and ends in some final state with the read head on ¢ or \$. Alternatively we could require the computation to end on ¢ (or \$) only, or even to end by falling off the left (or right) of the tape, i.e. with i = -1 (or i = n+2); similarly the computation could start by climbing up the tape. For $w \in \Sigma^*$, M(w) denotes as usual the set of all $v \in \Delta^*$ such that $(w,v) \in M$; for a language $L \subseteq \Sigma^*$, $M(L) = \cup\{M(w) \mid w \in L\}$ i.e. the image of L under M. M is *deterministic*

(denoted by 2dgsm) if δ is a function from $Q \times (\Sigma \cup \{¢,\$\})$ into $Q \times \{-1,0,+1\} \times \Delta^*$. The class of mappings realized by a 2gsm (2dgsm) is denoted by 2GSM (2DGSM). For a class of languages L, 2GSM(L) is as usual $\{M(L) \mid M \in 2GSM \text{ and } L \in L\}$.

Dropping the endmarkers and restricting the set of directions to $\{0,+1\}$ and $\{+1\}$ gives the usual *a-transducer* and 1-way generalized sequential machine (*gsm*, dgsm) respectively. Dropping the output tape gives the corresponding acceptors (2-way and 1-way finite state automata) which accept the domains of the translations realized by the transducers.

In what follows we will always assume (in the 2-way case) that the set of directions is restricted to $\{-1,+1\}$!

The basic capability of the 2-way finite state transducer is to make copies of substrings and to reverse them. Thus it is easy to define e.g. an $M \in 2DGSM$ such that $M(w) = ww\tilde{w}$ for all $w \in \Sigma^*$, where $\tilde{w}$ denotes the reverse of w, and an $M \in 2GSM$ such that $M(a^n) = \{(a^n b)^m \mid m \geq 2\}$ for all n.

1. EXAMPLE. We define a 2dgsm M such that $M(a^{n_1}ba^{n_2}b\ldots ba^{n_k}) = a^{n_1}b^{n_1}a^{n_2}b^{n_2}\ldots a^{n_k}b^{n_k}$. The machine makes three sweeps over each a^{n_i} consecutively; it has the following Algol-like flowchart (initially x = 0 and finally x falls off the right)

```
while input[x] ≠ $ do
  begin x := x+1;
        while input[x] = a do begin x := x+1; print(a) end;
        x := x-1;
        while input[x] = a do x := x-1;
        x := x+1;
        while input[x] = a do begin x := x+1; print(b) end
  end;
x := x+1.
```

Formally M has the following transition function

$\delta(q_0,¢) = (q_1,+1,\lambda)$

$\delta(q_1,a) = (q_1,+1,a)$ $\qquad$ $\delta(q_1,b \text{ or } \$) = (q_2,-1,\lambda)$

$\delta(q_2,a) = (q_2,-1,\lambda)$ $\qquad$ $\delta(q_2,¢ \text{ or } b) = (q_3,+1,\lambda)$

$\delta(q_3,a) = (q_3,+1,b)$ $\qquad$ $\delta(q_3,b) = (q_1,+1,\lambda)$

$\delta(q_3,\$) = (q_3,+1,\lambda)$ $\qquad$ □

1.1. Transition tables and visiting sequences

The first main result on 2gsm (both here and historically) is that they accept regular languages, i.e. the 2-way finite state automaton has the same power as the 1-way finite state automaton [RabSco; She]. This might sound surprising in view of the following examples. Given a regular language R, a 2-way finite state automaton is able to cut R in half, i.e. to accept the language $H(R) = \{u \in \Sigma^* \mid$ there exists $v \in \Sigma^*$ such that $|u| = |v|$ and $uv \in R\}$. In fact, if M is a 1-way finite state automaton recognizing R, then a 2-way finite state automaton can be constructed which, on input u, first simulates M on input u (from left to right) and then continues simulation of M on some string v of the same length which it guesses nondeterministically symbol by symbol by walking from right to left over u. By a slightly different construction it is easy to see that a deterministic 2-way finite state automaton can take the root of R, i.e. accept the language $\{u \in \Sigma^* \mid uu \in R\}$. It is left as an exercise to the reader to show that the languages $\{u \in \Sigma^* \mid$ there exists $v \in \Sigma^*$ such that $uv \in R$ and $|v| = n|u|$ for some $n \geq 1\}$ and $\{u_1\#u_2\#\ldots\#u_n \mid$ there exists $v_1,\ldots,v_n$ such that $|u_i| = |v_i|$ and $u_1v_1\#u_2v_2\#\ldots\#u_nv_n \in R\}$ are accepted by a 2-way finite state automaton.

Thus 2-way finite state automata are capable of a few weird operations which nevertheless preserve the regular languages by the above mentioned result. We will give two proofs of this result in order to define and illustrate the use of two important general concepts: visiting sequences and transition tables. Let us start with the simplest proof, which uses (state) transition tables [She; HopUll 2], well-known from the 1-way finite automaton. The *transition table* of a 2gsm M for a nonempty string u, not necessarily surrounded by endmarkers, is a relation $R_u \subseteq Q \times Q$ (or a partial function $Q \to Q$ if M is deterministic) such that $(q_1,q_2) \in R_u$ if and only if there is a computation of M on u which starts on the first square of u in state q_1 and falls off the left end of u in state q_2. Clearly, just as in the 1-way case, if $R_u = R_v$ then $R_{\sigma u} = R_{\sigma v}$, where σ is a symbol. This means that the transition table for a string u can be computed by a deterministic 1-way finite state automaton A which walks from right to left on u and contains in its state the transition table of the suffix of u it has read. A accepts the same language as M (modulo endmarkers), assuming that M always starts at ¢ and terminates to the left of ¢, and that A accepts a string u iff u is of the form ¢w$ and R_u contains (q_0,q) for some final state q of M. If you do not like A's direction, then consider its "reversal" or define

transition tables for prefixes of ¢w$ as in [She].

2. THEOREM. [Rab 1]. *For each 2-way finite state automaton an equivalent 1-way finite state automaton can effectively be found. So 2-way finite state automata accept regular languages.*

PROOF. We have seen that a 1-way automaton A exists. To prove effectiveness, it suffices to see that $R_{\sigma u}$ can be computed from σ and R_u by the following rule (see Fig. 3):

(*) if there is a sequence of states $q_1, q_1', \ldots, q_{n-1}, q_{n-1}', q_n, q_n'$ of M such that $1 \leq n \leq \#(Q)$,

$$\delta(q_i, \sigma) \ni (q_i', +1, \ldots) \text{ for } 1 \leq i \leq n-1,$$

$$\delta(q_n, \sigma) \ni (q_n', -1, \ldots)$$

and $(q_i', q_{i+1}) \in R_u$ for $1 \leq i \leq n-1$,

then (q_i, q_n') is in $R_{\sigma u}$ for all i, $1 \leq i \leq n$.

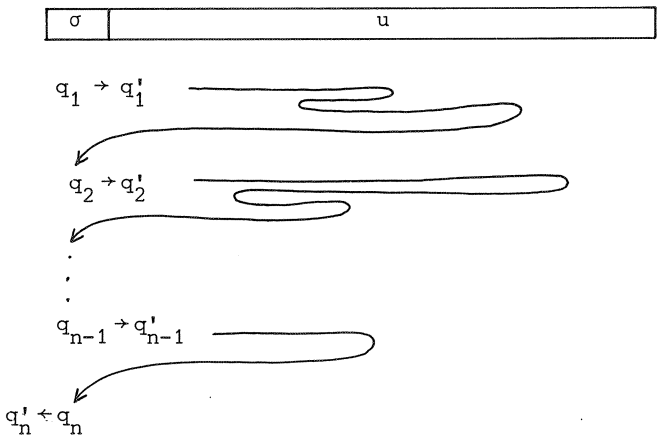


Figure 3. Behaviour of a 2gsm on a suffix.

Intuitively (*) means that we start M on σu in state q_1, let M work (using R_u whenever it enters u) and wait until it moves left of σ (or repeats itself). □

Note that, by this theorem, each 2dgsm is equivalent to an always halting one. Note also that it easily follows from Theorem 2 that the regular languages are closed under intersection. (In fact, the families of

languages accepted by 2-way acceptor models are nearly always closed under intersection.)

3. EXERCISE. Prove that the regular languages are closed under inverse 2gsm mappings. Show that the operations mentioned in the beginning of section 1.1 (such as cutting in half, taking the root) are examples of inverse 2gsm mappings. □

We now turn to the alternative proof of Theorem 2 based on visiting sequences (or crossing sequences [Rab 2; Hen; HopUll 3]). The proof is more complicated but easier to vary for other purposes.

A straightforward idea to prove Theorem 2 is the following. Consider the computation path of a 2gsm M on some input tape, see Fig. 4. Between two turns M behaves actually just as a 1-way automaton. It is clearly possible to simulate all these 1-way pieces of computation simultaneously by a 1-way automaton N (from left to right), alternatingly a forward and a backward simulation of M. N guesses M's state at ⊂ turns and checks whether the forward and backward computations fit at ⊃ turns. To keep track of this simulation N keeps a visiting sequence of M in its finite control.

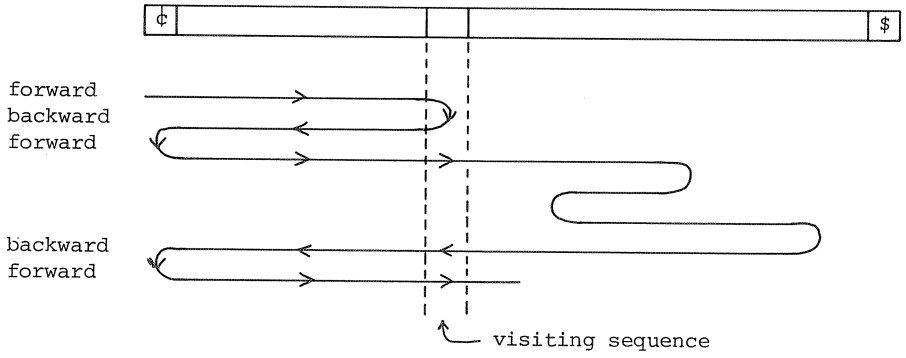


Figure 4. Computation path of a 2gsm.

A visiting sequence contains local information about the computation of a 2gsm, analogous to a triple $\langle q_1,\sigma,q_2\rangle$ in the 1-way case where q_1 (q_2) is the state of the 1-way automaton before (after) reading σ. A *visiting sequence* of a 2gsm M is a sequence $s = (\sigma,\langle d_1,q_1,d_1',q_1',v_1\rangle, \langle d_2,q_2,d_2',q_2',v_2\rangle,\ldots,\langle d_k,q_k,d_k',q_k',v_k\rangle)$ such that $(q_i',d_i',v_i) \in \delta(q_i,\sigma)$, where

$\sigma \in \Sigma$, $k \geq 0$, $d_i, d_i' \in \{-1,+1\}$, $q_i, q_i' \in Q$ and $v_i \in \Delta^*$. Intuitively s encodes all information about M's behaviour while visiting a particular input tape square during a given computation: the square contains σ, it is visited k times by M and during the i-th visit M enters the square in the d_i direction in state q_i and leaves it in the d_i' direction in state q_i', producing v_i as output. Clearly each (halting) computation of M on an input of length n can be described completely by a sequence of n+2 visiting sequences, one for each square. An example is given in Figure 5 for the 2dgsm defined in Example 1; in this picture the visiting sequences are put vertically, d_i is drawn as an arrow towards q_i and d_i' as an arrow away from q_i; q_i' and v_i are left out. (For a 2dgsm d_i', q_i' and v_i are actually superfluous because they are determined by δ.) The dots in Figure 5 are drawn to

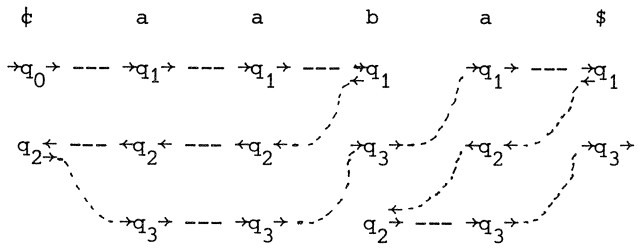


Figure 5. A sequence of visiting sequences.

indicate how the computation path of the 2dgsm can be recovered from the visiting sequences: just connect the right-going arrows "between" two visiting sequences in the obvious bijective way (in the order in which they appear in the sequences), and similarly for the left-going arrows.

Vice versa, given an arbitrary sequence of visiting sequences, it is easy to decide whether they encode a (halting) computation of M (on the input string spelled by their first elements): check for each pair s_1, s_2 of consecutive visiting sequences whether they *fit* in the sense that the arrows which cross the boundary between the two squares can be connected as suggested above and that the state transitions indicated by s_1 and s_2 are consistent; also check whether the first visiting sequence is *initial* in the sense that its symbol is ¢, its first 5-tuple is $<+1,q_0,+1,\ldots>$ and all others are of the form $<-1,q,+1,\ldots>$, and whether the last 5-tuple is

final in a symmetric sense (where we assume that M climbs up the left end and falls off the right end of the tape). If this has been checked, the visiting sequences have to encode a computation: just follow the path indicated by the (connected) arrows, starting at the first 5-tuple of the first visiting sequence; since no 5-tuple can be passed twice (which would give a loop), the path has to end at the only loose exiting arrow: the last 5-tuple of the last visiting sequence. Note however that there may be some superfluous information which encodes a circular path disconnected from the main path; as an example, change the visiting sequence of square b in Figure 5 into $(b,\rightarrow q_1 \rightarrow, \overset{\rightarrow}{\leftarrow} q_3, q_2 \overset{\leftarrow}{\rightarrow})$.

Thus a string of visiting sequences encodes a computation of M if and only if it starts (ends) with an initial (final) visiting sequence and consecutive visiting sequences fit. A formal proof is left to the reader; it can be found (for other purposes) in [Gre 2; Fis 1]. It should moreover be clear that (in case the length of the visiting sequences is bounded) the decision process can be handled by a 1-way finite state automaton, cf. the discussion following Exercise 3. For $k \geq 1$ we say that a computation of a 2gsm is k-*visit* if each square is visited at most k times. We can now formulate the following basic lemma.

4. LEMMA. *For each* 2gsm M *and* $k \geq 1$ *there is a* 1-*way* gsm V_k *such that* $V_k(\not\sigma_1\sigma_2\ldots\sigma_n\$) = \{s_0s_1s_2\ldots s_ns_{n+1} \mid s_0\ldots s_{n+1}$ *is a sequence of visiting sequences of* M *encoding a* k-*visit computation of* M *on input tape* $\not\sigma_1\ldots\sigma_n\$\}$.

PROOF. The states and output symbols of V_k are the visiting sequences of M with at most k 5-tuples. V_k starts by guessing an initial visiting sequence and for each new input symbol σ it guesses a visiting sequence with symbol σ which fits to the previous one; V_k accepts if a final visiting sequence can be reached. Observe that when V_k has read $\not\sigma_1\ldots\sigma_j$ it has guessed a computation of M on $\not\sigma_1\ldots\sigma_j$ in which there are still "holes" caused by 5-tuples with $d_i' = +1$ or $d_i = -1$ in the current visiting sequence. □

Note that V_k is actually a sequential machine (with accepting states), i.e. produces one output symbol for each input symbol. Note also that each halting computation of a 2dgsm M is $\#(Q)$-visit: if M visits the same square twice in the same state, then it is in a loop; however, the number of visits in halting computations of a 2gsm is in general not bounded. On the other hand, if the 2gsm M has an accepting computation on a given input, then it also has one which is k-visit for $k = \#(Q)$: skip the parts between two

visits to the same square in the same state. Hence the 1-way gsm V_k of Lemma 4 accepts the same language as M, which proves Theorem 2 again.

For future use we observe that Theorem 2 can also be proved using the concept of *visiting set* which is a visiting sequence without order (note that we may assume that no 5-tuple occurs twice in a visiting sequence). The definition of fitting is the same except that some order should be chosen in the visiting sets (which may be different with respect to the left and right neighbour of a visiting set!). Thus a sequence of fitting visiting sets encodes in general many (but at least one!) computations of the 2gsm, depending on the choice of the order.

We finally note that the bounded visit property even allows writing on the tape, as shown in the next exercise.

5. EXERCISE. [Hen]. Let T be a (nondeterministic) Turing machine with one tape (which is both input and working tape) and assume that there is an integer k such that each accepted input can be accepted with a k-visit computation. Show that the language accepted by T is regular. □

1.2. Ranges of 2gsm mappings

Everything said about domains, we now turn to ranges of 2gsm's and, more generally, to classes 2DGSM(L) and 2GSM(L) of output languages, where L is some class of input languages. In what follows we always assume that L is closed under 1-way (g)sm mappings and putting endmarkers (so that Lemma 4 is applicable). Note that 2GSM(REG) is the class of ranges of 2gsm's, where REG denotes the class of regular languages. As observed in the introduction and shown in [Raj 2], 2GSM(REG) equals the class CSA of languages accepted by 1-way checking stack automata [Grei 3]; similarly 2GSM(L) is equal to the class of "L-based preset" checking stack languages [Kie; Gre 1; Gre 2]. However, determinism does not carry over. We shall concentrate on the difference between 2dgsm and 2gsm, and on output languages obtained by compositions of 2gsm's. These results and many others can be found in [Gre 3; AhoUll 1; EhrYau; Raj 1; Raj 2; Kie] and the more recent [Gre 1; Gre 2; EngRozS; Gre 4; Eng 1] and (in a different formalism, cf. the next section) in [RozVer; Ver; Lat 1; Lat 2].

A basic property of the range of a 2dgsm is the "Parikh property", i.e. changing the order of the symbols in each of its strings produces a regular language.

6. THEOREM. *Each language in* 2DGSM(REG) *has the Parikh property.*

PROOF. Define, for a visiting sequence s, h(s) to be the output produced according to its visits, i.e. if $s = (\sigma,\langle d_1,q_1,d_1',q_1',v_1\rangle,\langle\ldots,v_2\rangle,\ldots,\langle\ldots,v_k\rangle)$ then $h(s) = v_1v_2\ldots v_k$. Change the gsm V_k of Lemma 4 (with $k = \#(Q)$) such that it outputs h(s) rather than s. V_k now produces the same output string as M but in a different order. Moreover the output language of V_k is regular, because REG is closed under gsm mappings. □

It should be clear from the proof that Theorem 6 holds for any 2DGSM(L) such that each language of L has the Parikh property. It is therefore in particular true for 2DGSM(CF), where CF denotes the class of context-free languages.

Since $\{a^n \mid n \text{ is not prime}\}$ is in 2GSM(REG) (translate a^k into $(a^k)^m$ for $k,m \geq 2$), Theorem 6 implies that 2DGSM(REG) $\not\subseteq$ 2GSM(REG). In this example the 2gsm makes an unbounded number of visits to the squares of its input tape. As shown next, this property is necessary for a 2gsm to produce a language not in 2DGSM(REG). A 2gsm is k-*visit* if each string of its output language can be produced by a k-visit computation, and *finite visit* if there is such a k. The corresponding classes of languages are denoted by $2GSM_k$(REG) and $2GSM_{FIN}$(REG). Note that these concepts are also natural for checking stack automata (CSA_k and CSA_{FIN}).

7. THEOREM. $2GSM_{FIN}(REG) = 2DGSM(REG)$.

PROOF. Let M be a k-visit 2gsm. The idea is to take the visiting sequences of M's computations on "regular" strings (with up to k visits per square) and to feed then as input into a 2dgsm N which simulates M by merely following the computation path encoded in each such sequence. Thus if L is the regular input language of M, then $V_k(L)$ is the (regular) input language of N (see Lemma 4). N follows M's computation by keeping an integer between 1 and k (to indicate which visit it simulates) in its finite control. □

This shows that 2dgsm's correspond to finite visit checking stack automata in a natural way. It is left to the reader to extend this result to Turing machines with a 1-way input (or output) tape and a working tape on which they are finite visit (cf. Exercise 5) [Raj 1; Gre 1]. Note that Theorem 7 can also be generalized to arbitrary L.

8. Exercise. Show that 2DGSM(2DGSM(REG)) = 2DGSM(REG), i.e. 2DGSM(REG) is closed under 2dgsm mappings. Show similarly that 2GSM(2DGSM(REG)) = = 2GSM(REG). Hint: the output string of the first 2dgsm lies "wrapped up" in the string of its visiting sequences. □

It follows from Theorem 7 that 2gsm output-languages which cannot be produced by a 2dgsm, contain strings which can be pumped.

9. COROLLARY. *If* $L \in 2GSM(REG) - 2DGSM(REG)$, *then* L *contains an infinite regular language.*

PROOF. The 2gsm which produces L is not $^{\#}(Q)$-visit by Theorem 7. Hence there exists an output string w which cannot be obtained from a $^{\#}(Q)$-visit computation. Thus, in the (shortest) computation with output w, some square is visited twice in the same state, and so the subcomputation between these visits (which produces $w_2 \neq \lambda$ with $w = w_1w_2w_3$) can be repeated any number of times. Hence L contains $w_1w_2^*w_3$. □

Theorem 6 and Corollary 9 together show that every infinite language in 2GSM(REG) over a one-letter alphabet contains an infinite regular language [Gre 3]. Hence the language $\{a^{n^2} \mid n \geq 1\}$, which is accepted by a 1-way (non-erasing) stack automaton, is not a checking stack language (every checking stack language is a stack language [Gre 3]). Another consequence of Theorem 7 and its Corollary 9 is that 2GSM(REG) is not closed under 2dgsm mappings (cf. Exercise 8) and in particular not under "copying". Define $c_2(L) = \{w\#w \mid w \in L\}$, where # is a "new" symbol.

10. THEOREM. *For every language* L, *if* $c_2(L) \in 2GSM(L)$ then $L \in 2DGSM(L)$.

PROOF. $c_2(L)$ contains no infinite regular language: if $w_1w_2^*w_3 \subseteq \{w\#w \mid w \in L\}$, then w_2 cannot contain # and it cannot be a substring of one of the w's. This forces the 2gsm to be finite visit and hence Corollary 9, generalized to L, implies $c_2(L) \in 2DGSM(L)$. A dgsm can be used to recover L from $c_2(L)$ (Exercise 8, generalized to L). □

Taking $L = \{a^n \mid n \text{ is not prime}\}$ or any other language in 2GSM(REG)-2DGSM(REG), $c_2(L)$ shows that

11. COROLLARY. $2GSM(REG) \not\subseteq 2DGSM(2GSM(REG))$. □

This shows that 2gsm mappings are not closed under composition! Intuitively this is because an output which is produced nondeterministically cannot be reproduced.

Theorem 10 is an example of an (upward) "translational method" [RubFis]. It enables one, by translating the "problem" L into the harder problem $c_2(L)$, to lift the proper inclusion $2DGSM(\mathcal{L}) \subsetneq 2GSM(\mathcal{L})$ to the proper inclusion $2GSM(\mathcal{L}) \subsetneq 2DGSM(2GSM(\mathcal{L}))$. Translational methods are widely used in complexity theory, where they are mostly called reduction techniques, and in formal language theory, e.g. [Gre3; Sky; EngSky].

12. EXERCISE. Show that $2GSM^2(\mathcal{L}) = 2GSM(\mathcal{L})$ if and only if $2DGSM(\mathcal{L}) = 2GSM(\mathcal{L})$. □

The natural question now arises whether the composition of n+1 2gsm's is more powerful than that of n. This is indeed the case as recently shown in [Gre 4; Eng 1]. We follow the proof in [Gre 4]; an alternative proof will be given in section 3, Exercise 46. What makes 2GSM intuitively more powerful than 2DGSM is the unbounded visit property which e.g. allows a 2gsm to make an unbounded number of copies of the input string. This can be expressed formally in the language operation c_* defined by $c_*(L) = \{(w\#)^n \mid w \in L, n \geq 1\}$, where # is a new symbol. Thus L can be translated by a 2gsm into $c_*(L)$. The following translational result shows that our intuition is true if the class of input languages is of the form 2GSM(...).

13. THEOREM. *If* $c_*(L) \in 2DGSM(2GSM(\mathcal{L}))$, *then* $L \in 2DGSM(\mathcal{L})$.

PROOF. Let $c_*(L) = M_2(M_1(L_0))$ for some $L_0 \in \mathcal{L}$, $M_1 \in 2GSM$ and $M_2 \in 2DGSM$. The idea of the proof is (similarly to that of Theorem 10) to show that the form of the language $c_*(L)$ forces M_1 to be finite visit and so $c_*(L) \in 2DGSM^2(\mathcal{L}) = 2DGSM(\mathcal{L})$ and $L \in 2DGSM(\mathcal{L})$. A quick formal proof goes as follows. Let k be the number of states of M_2. Then $c_*(L) \in 2DGSM_k(2GSM(\mathcal{L}))$ and so $c_{k+1}(L) = \{(w\#)^{k+1} \mid w \in L\}$ is also in $2DGSM_k(2GSM(\mathcal{L}))$ because this class is closed under intersection with regular languages. Hence $c_{k+1}(L) = N_2(N_1(K_0))$ with $K_0 \in \mathcal{L}$, $N_1 \in 2GSM$ and $N_2 \in 2DGSM_k$. Since $2GSM(\mathcal{L})$ is closed under 1-way gsm mappings and endmarking, we may assume by Lemma 4 that N_1 produces output strings on which the visiting sequences of N_2 are printed. Suppose now that N_1 is not finite visit. As in the proof of Corollary 9 this means that N_1 produces strings $w_1 w_2^* w_3$ with $w_2 \neq \lambda$. The k-visit 2dgsm N_2 translates these into strings $u_1 v_1^n u_2 v_2^n \ldots u_k v_k^n u_{k+1}$ $(n \geq 1, v_1 v_2 \ldots v_k \neq \lambda)$

in $c_{k+1}(L)$; this is easy to see if the subcomputations of N_1 on w_2 always start at one end and leave at the other end of w_2, for the general case see the "pumping lemma" in [Gre 1, Lemma 4.22]. Clearly $c_{k+1}(L)$ cannot contain these strings. □

This shows that the proper inclusion $2DGSM(L) \subsetneq 2GSM(L)$ can be lifted to the proper inclusion $2DGSM(2GSM(L)) \subsetneq 2GSM(2GSM(L))$ on the next level. Together with Theorem 10 we obtain the following hierarchy result.

14. THEOREM. *If* $2DGSM(L) \subsetneq 2GSM(L)$, *then for all* $n \geq 1$ $2GSM^n(L) \subsetneq 2DGSM(2GSM^n(L)) \subsetneq 2GSM^{n+1}(L)$. □

Note that the pumping argument in the proof of Theorem 13 also shows that k+1 visits are more powerful than k visits, at all levels (also for 2DGSM(REG) because every regular language contains some $w_1w_2^*w_3$).

Thus for L = REG and L = CF we obtain proper hierarchies $\{2GSM^n(L)\}_{n\geq 1}$. General conditions on L under which the hypothesis of Theorem 14 is true are given in [Gre 4]; see also Theorem 49 in section 3. One such condition is clearly the Parikh property. Another is that L is a full principal substitution-closed AFL, not closed under 2dgsm mappings (such as the indexed languages). It is open whether "L is a full semi-AFL and $L \subsetneq 2GSM(L)$" is such a condition.

Another recent result on ranges is the incomparability of CF and 2GSM(REG), i.e. of pushdown automata and checking stack automata [Gre 1; EngSch^{V}L]. A careful analysis of the closure properties of $2DGSM(L)$ with respect to the operation of substitution (similar to the one concerning stack automata in [Gre 3]) gives the following result [Gre 1; Lat 1].

15. THEOREM. *Let* L *be a full semi-*AFL *and* L_1 *a full principal substitution-closed* AFL. *If* $L_1 \subseteq 2DGSM(L)$, *then* $L_1 \subseteq L$. □

Taking e.g. L_1 = CF this means that if L does not contain all context-free languages, then neither does $2DGSM(L)$: 2dgsm mappings are of no use in producing all context-free languages. In particular L = REG shows that 2DGSM(REG) does not include CF. This can be extended to 2GSM(REG) as follows. Assume that CF $\subseteq 2GSM(L)$ and let L $\in$ CF. Let p(L) be the parenthesis language obtained by putting parentheses around right-hand sides of rules in the context-free grammar for L. Then p(L) $\in 2GSM(L)$. Since parenthesis languages do not contain an infinite regular language, Theorem 7 shows that

$p(L) \in 2DGSM(L)$ and, erasing parentheses, that $L \in 2DGSM(L)$. Together with Theorem 15 this proves that if $CF \subseteq 2GSM(L)$ then $CF \subseteq L$. Moreover, since CF is a full principal AFL, the next general theorem is obtained [Gre 4].

16. THEOREM. *Let L be a full semi-*AFL. *If* $CF \subseteq \bigcup_n 2GSM^n(L)$, *then* $CF \subseteq L$. □

This theorem actually holds not only for CF, but for any full principal substitution-closed AFL with an AFL-generator which contains no infinite regular set (e.g. the indexed languages).

Thus CF cannot be reached from REG using 2gsm mappings. By the well-known substitution result of [Gre 3] it follows that not all context-free languages can be accepted by nonerasing stack automata.

We note that the above "parenthesis argument" strongly suggests that there is no "natural" class of grammars corresponding to the checking stack automaton. Otherwise the same parenthesis argument would be applicable to 2GSM(REG), giving a contradiction.

1.3. Composition of 2dgsm mappings, general automata theory

Exercise 8 suggests that 2dgsm mappings are closed under composition. This has recently been proved in [ChyJak], although most of the steps needed for the proof are already contained in [HopUll 1; AhoUll 1]. The problem with the composition of two 2dgsm's M_1 and M_2 is that we would like to construct a 2dgsm M_3 which gets an input string of M_1 and knows the visiting sequences of M_1 on this input, cf. Exercise 8. Note that since M_1 is deterministic, there is at most one computation on each input and consequently at most one "correct" visiting sequence on each square. But it seems impossible for M_3 (which has only a finite memory) to leave the current square in order to compute M_1's visiting sequence by, say, simulating V_k of Lemma 4. Surprisingly, the following "*regular context lemma*" was proved in [HopUll 1, Lemma 3], see also [AhoHopU 1, p.212].

17. LEMMA. *A* 2dgsm *can keep track in its finite control of the state of a* 1*-way deterministic finite state automaton.*

PROOF. Let $A = (Q,\Sigma,-,\delta,q_0,-)$ be a 1-way deterministic finite state automaton (i.e. a dgsm without output). We extend δ as usual such that, for $q \in Q$ and $w \in \Sigma^*$, $\delta(q,w)$ is the state of A after reading w when starting in state q. We want to show that, on an input tape $\¢\sigma_1\sigma_2\ldots\sigma_i\ldots\sigma_n\$$, every

2dgsm M can keep track of $\delta(q_0,\sigma_1\sigma_2\ldots\sigma_i)$ where i is the position of M's input head. To prove this, suppose that M has $\delta(q_0,\sigma_1\sigma_2\ldots\sigma_i) = q$ in its finite control (initially M starts on ¢ with q_0 in its finite control). If M moves right, it just replaces q by $\delta(q,\sigma_{i+1})$. The problem arises when M moves left: what should M do to compute $\delta(q_0,\sigma_1\ldots\sigma_{i-1})$? Suppose $\{p \in Q \mid \delta(p,\sigma_i) = q\} = \{p_1,p_2,\ldots,p_r\}$, i.e. $p_1,\ldots,p_r$ are all possible states of A after simulating A one step backwards. If $r = 1$, then $\delta(q_0,\sigma_1\ldots\sigma_{i-1}) = p_1$ and there is no problem. If $r \geq 2$ then M continues to move left while simulating A backwards for each p_i as "final" state, simultaneously. Thus when M arrives on the j-th square of the input tape $¢\sigma_1\ldots\sigma_j\ldots\sigma_{i-1}\sigma_i\ldots\$$, it has computed $\gamma(\sigma_{j+1}\ldots\sigma_{i-1},p_k)$ for every k ($1 \leq k \leq r$), where $\gamma(v,p) = \{p' \in Q \mid \delta(p',v) = p\}$. Note that these γ-sets are disjoint. But M has to be careful: it should be able to return to the (i-1)-th square. Fortunately M can always do so as long as at least two of the $\gamma(\sigma_{j+1}\ldots\sigma_{i-1},p_k)$ are nonempty: in that case it picks a state out of each one of these two γ-sets and moves right simulating A for both of these states; as soon as they coincide M is back on the i-th square (they cannot coincide earlier because they should be different on the (i-1)-th square).

There are two cases. Case 1: at the j-th square all γ-sets become empty except one: $\gamma(\sigma_{j+1}\ldots\sigma_{i-1},p_k)$. Then p_k is the state of A to be computed. M moves one square to the right into the previous situation (which it should remember in its finite control) and goes home as indicated above.
Case 2: M arrives at ¢ with at least two of the γ-sets nonempty. If $q_0 \in \gamma(\sigma_1\ldots\sigma_{i-1},p_k)$, then p_k is the required state. □

This lemma clearly settles the composition result in case M_1 is a 1-way dgsm. It will now be used to prove the composition result of [ChyJak].

18. THEOREM. 2DGSM *is closed under composition.*

PROOF. To compose M_1 and M_2 it suffices to show that M_3 can be constructed so that it keeps track of the visiting sequence of M_1 at the current square of the input tape of M_1 (and M_3). Then M_3 can move along the output string of M_1 (which is wrapped up in the visiting sequences) simulating M_2 on this string. Let M_3 be on the i-th square of input tape $¢\sigma_1\ldots\sigma_i\ldots\sigma_n\$$. By Lemma 17, applied to the deterministic 1-way finite state automaton corresponding to the gsm V_k (without output) of Lemma 4 by the usual subset construction, M_3 can compute the finite set L of all visiting sequences s_i of M_1 for which

there is a sequence $s_0 s_1 \ldots s_i$ of fitting visiting sequences (on $\not\sigma_1 \ldots \sigma_i$) with s_0 initial. Since in both Lemma 4 and 17 we may as well assume that the 1-way device moves from right to left, M_3 can also compute the set R of all s_i for which there is a fitting sequence $s_i \ldots s_n s_{n+1}$ with s_{n+1} final. Since there is only one computation of M_1 on the input tape, $L \cap R$ is a singleton containing the required visiting sequence. □

We now observe that in this proof it is essential that M_1 is a 2dgsm, whereas M_2 might in fact be any other device which, if required, is able to work as a 2dgsm. In particular 2DGSM ∘ 2GSM ⊆ 2GSM (where $M_1 \circ M_2$ means "first M_1, then M_2"), cf. Exercise 8. If M_2 is any other two-way transducer, then M_3 will be of the same type. The reader can easily check that the only thing required of our transducer type is that it can move along its input tape without manipulating its storage. This independence between input and storage is exactly the basic assumption of general automata theory as described, independently, in [HopUll 1; Gin; Sco]. The definitions found in each of these references can serve (almost) the same purposes; unfortunately, the simplest [Sco] has been the least developed. Our definition is inspired by [Sco], cf. [Gol].

First the nondeterministic case. A *storage type* (or data type) X is specified by a set S of (storage) configurations, a set $S_0 \subseteq S$ of initial configurations, a set $S_\infty \subseteq S$ of final configurations, a set T of tests which are partial functions from S to {true,false}, and a set F of operations which are binary relations on S. As an example, pushdown storage is defined by taking $S = \Gamma^*$ for some countably infinite alphabet Γ (or equivalently $\Gamma = \{0,1\}$), $S_0 = \{\lambda\}$, $S_\infty = \{\lambda\}$ or $S_\infty = \Gamma^*$ depending on whether acceptance is by empty storage or not, $T = \{e\} \cup \{top_\gamma \mid \gamma \in \Gamma\}$ such that e(w) is true for $w \in \Gamma^*$ iff $w = \lambda$, and $top_\gamma(w)$ is true iff w's rightmost symbol is γ, and $F = \{pop\} \cup \{push_\gamma \mid \gamma \in \Gamma\}$ where $pop = \{(w\gamma,w) \mid w \in \Gamma^*, \gamma \in \Gamma\}$ and $push_\gamma = \{(w,w\gamma) \mid w \in \Gamma^*\}$ for all $\gamma \in \Gamma$. Alternatively (in the nondeterministic case) we could take $T = \emptyset$ and $F = \{pop_\gamma \mid \gamma \in \Gamma\} \cup \{push_\gamma \mid \gamma \in \Gamma\}$ with $pop_\gamma = \{(w\gamma,w) \mid w \in \Gamma^*\}$, and use a special symbol in Γ as bottom symbol. An *automaton type* consists of a storage type X together with a way of handling the input and output (such as 1-way, 2-way, multi-head, etc.). Informally an automaton type consists of a type of storage and a type of finite state automaton. This shows why finite state automata are so important in general automata theory. An *automaton* "of that type" is obtained by writing any nondeterministic flowchart which uses the tests and operations

allowed on X, the input and the output. The automaton starts with its storage in an initial configuration and accepts only in a final configuration. In the *deterministic* case we require that the operations in F are partial functions and that all flowcharts are deterministic (and, if you wish, that S_0 is a singleton). In the nondeterministic case it may always be assumed that $T = \emptyset$: each test can be simulated in the usual way by a guess between two partial (identity) functions.

As an example, for a given X without tests, a 2-way nondeterministic X transducer is a nondeterministic flowchart which uses the elementary instructions on X together with those on the input array and output tape as specified for the 2gsm in the beginning of this section. Formally it can be described in the usual fashion by a function δ from $Q \times (\Sigma \cup \{¢,\$\})$ to the finite subsets of $Q \times \{-1,0,+1\} \times \Delta^* \times F^*$. An element $f_1 f_2 \ldots f_n$ of F^* indicates that operations $f_1, f_2, \ldots, f_n$ should be applied to the current configuration in that order (nothing happings if $n = 0$). In the deterministic case (and an X with tests) the values of the tests appear in the domain of δ; a formalization is left to the reader.

For every storage type X let $T_2^D(X)$ and $T_2^N(X)$ denote the classes of translations defined by two-way deterministic and nondeterministic X transducers respectively.

19. THEOREM. *For every storage type* X,
$2DGSM \circ T_2^D(X) \subseteq T_2^D(X)$ *and* $2DGSM \circ T_2^N(X) \subseteq T_2^N(X)$. □

Let us now consider 2-way deterministic X acceptors and let $L_2^D(X)$ denote the class of accepted languages. Since $dom(f \circ g) = f^{-1}(dom(g))$, where dom = domain, the above composition theorem shows that $L_2^D(X)$ is closed under inverse 2dgsm mappings. In fact this property (almost) characterizes such classes, as shown in [AhoUll 1]. To prove this we need a basic lemma which formally states the division of an automaton into storage and finite state automaton. Consider a 2-way deterministic X acceptor M, where X has no tests. Then δ is a function from $Q \times (\Sigma \cup \{¢,\$\})$ to $Q \times \{-1,0,+1\} \times F^*$, i.e. M is a 2dgsm with output alphabet F! Actually the output alphabet is a finite subset of F. For $F' \subseteq F$, let $L_{F'}$ be the set of all $f_1 f_2 \ldots f_n \in (F')^*$ which (when interpreted) lead from some initial configuration to some final configuration. The next lemma is analogous to one for the 1-way case in [Gin].

20. LEMMA. *Let* X *be a storage type without tests.* $L_2^D(X)$ *consists of all languages* $M^{-1}(L_{F'})$, *where* M *is a* 2dgsm *with output alphabet* $F' \subseteq F$.

PROOF. Since no tests are made on storage, we can first compute the "output" $f_1 f_2 \ldots f_n$ and then check whether this sequence of operations leads to success. □

21. THEOREM. [AhoUll 1]. *A class of languages is* $L_2^D(X)$ *for some storage type* X *if and only if it is closed under inverse* 2dgsm *mappings and marked union (i.e.* $aL_1 \cup bL_2$ *where* a *and* b *are new symbols).*

PROOF. Closure of $L_2^D(X)$ under inverse 2dgsm mappings has been shown and closure under marked union is obvious. Now let $\mathcal{L}$ be a class closed under these operations. Storage type X is defined as follows. S consists of the empty string and all strings $\sigma_L w$ with $L \in \mathcal{L}$ (where σ_L is L viewed as a symbol) and $w \in \Sigma^*$ (where Σ is the set of all symbols appearing in $\mathcal{L}$). $S_0 = \{\lambda\}$ and $S_\infty = \{\sigma_L w \mid w \in L\}$. T is empty and F is the set of all partial functions

$$\begin{cases} f_L(\lambda) = \sigma_L \text{ and undefined otherwise, for all } L \in \mathcal{L} \\ f_\sigma(\sigma_L w) = \sigma_L w\sigma \text{ and } f_\sigma(\lambda) \text{ is undefined, for all } \sigma \in \Sigma \end{cases}$$

F as an alphabet can be identified with $\{\sigma_L \mid L \in \mathcal{L}\} \cup \Sigma$. Then L_F is clearly the union of all languages $\sigma_L L$, $L \in \mathcal{L}$. Hence, for a finite $F' \subseteq F$, $L_{F'}$ is a finite union of such languages and so $L_{F'} \in \mathcal{L}$ (by closure under marked union). Lemma 20 now implies that $L_2^D(X) \subseteq \mathcal{L}$. On the other hand, each $L \in \mathcal{L}$ can easily be recognized by an (even 1-way) acceptor which writes σ_L in storage followed by the input string.

Of course the constructed 2-way deterministic acceptor type is not very natural: it has an infinite number of instructions and accepts in a odd way. □

The result in [AhoUll 1] actually considers the more special case of automata which can be reset to their (unique) initial configuration. To characterize these in terms of operations, marked concatenation and marked star have to be added.

It is easy to see that Lemma 20 also holds for 2-way nondeterministic X acceptors ($L_2^N(X)$) for which $T = \emptyset$ may always be assumed. Let $L_1^N(X)$ be the corresponding 1-way class.

22. THEOREM. *For every storage type* X, $L_2^N(X) = 2GSM^{-1}(L_1^N(X))$.

PROOF. $\subseteq$: Use the nondeterministic version of Lemma 20. Clearly $L_{F'}$ is accepted by a 1-way X acceptor.
$\supseteq$: The 2-way acceptor simulates the 2gsm and feeds its output directly into the 1-way acceptor. □

As an example, $2GSM^{-1}(CF)$ is the class of languages accepted by 2-way nondeterministic pushdown automata.

Since $\mathcal{L}_1^N(X)$ classes can be characterized as full semi-AFLs (i.e. closed under a-transducers and union) [Gin; Gol], this shows that the $\mathcal{L}_2^N(X)$ classes are characterized as the images of full semi-AFLs under inverse 2gsm mappings. Thus the study of 2-way acceptors is dual to that of preset checking stack automata ($2GSM^{-1}(\mathcal{L})$ versus $2GSM(\mathcal{L})$). Recall that inverse 2gsm mappings are mappings such as cutting in half, taking the root, etc. (cf. Exercise 3).

Lemma 20 and Theorems 21 and 22 (see also Exercise 3) express the basic "inverse law" of the general theory of acceptors: the class of languages accepted by an acceptor type can be expressed in terms of the inverse mappings defined by the finite state transducers with the same type of input. (For the 1-way case [Gin], note that a-transducers are closed under inverse, and that quasi-realtime 1-way acceptors are characterized by inverse gsm mappings i.e. h^{-1}, $\cap$ R,λ-free h). Theorems 19 and 21 express the "composition law" of the theory: in order to obtain a characterization of $\mathcal{L}(X)$ by closure under language operations, the corresponding class of finite state transductions should be closed under composition. Thus the nonclosure of 2gsm mappings under composition explains that no such characterization result for 2-way nondeterministic acceptors exists. It is an open problem whether a different set of operations exists which characterizes $\mathcal{L}_2^N(X)$ as in Theorem 21.

We return to 2dgsm and 2gsm mappings. Although 2gsm's are clearly stronger than 2dgsm's due to nondeterminism, we will show that this gives no additional power apart from computing more outputs to the same input. More clearly: 2GSM $\cap$ PF = 2DGSM, where PF is the class of partial functions. To show this we need the concept of a 1-way dgsm with *regular look-ahead* (denoted $dgsm^R$). A $dgsm^R$ M decides its next move on the basis of regular properties of the rest of the input string (and, as usual, its state and input symbol). These properties are specified by a finite set of disjoint regular languages or, equivalently, by the states of a 1-way deterministic finite state automaton which works from right to left on the same input. Basic properties of $dgsm^R$ are

(i) DGSM $\not\subseteq$ DGSM^R $\subseteq$ GSM (clear),

(ii) DGSM^R $\subseteq$ 2DGSM (by the regular context lemma 17),

(iii) GSM $\cap$ PF = DGSM^R (using regular look-ahead, determine which next moves of the gsm lead to acceptance; do one of these moves).

(i) and (iii) show that the theorem to prove does not hold in the 1-way case! (iii) can be improved by replacing the dgsm^R by a conceptually simpler automaton: the bimachine [Sch; Eil].

We need the following "*slice lemma*".

23. LEMMA. *For every* 2gsm M *there is a* 2dgsm N *with the same domain and* N $\subseteq$ M (*as a relation*).

PROOF. We first note that this slice lemma holds for gsm and dgsm^R (in place of 2gsm and 2dgsm respectively): use the same proof as (iii) above. Let k be the number of states of M. If M has an accepting computation on ¢w$, then it also has a k-visit accepting computation on ¢w$. Let M_1 be a dgsm^R which "slices" the gsm V_k of Lemma 4. Let M_2 be the 2dgsm which simulates M on the sequences of visiting sequences. Let N = $M_1 \circ M_2$. Clearly N slices M. Since DGSM^R $\subseteq$ 2DGSM and 2DGSM is closed under composition, N is in 2DGSM. □

It easily follows from this lemma that 2GSM $\cap$ PF = 2DGSM. Now remember that {2GSM^n} is a proper hierarchy. Even these compositions cannot give new partial functions.

24. THEOREM. *For every* n $\geq$ 1, 2GSM^n $\cap$ PF = 2DGSM.

PROOF. Let $M_1 \circ M_2 \circ \ldots \circ M_n$ be a partial function, with $M_i \in$ 2GSM. Change M_i in such a way that it only produces strings in the domain of M_{i+1} (use Theorem 2 and start with i = n-1). Replace M_i by a slice N_i according to Lemma 23. Since 2DGSM is closed under composition, the result follows. □

2. GRAMMARS FOR TWO-WAY TRANSDUCERS

We would like to find grammars corresponding to 2-way finite state transducers, i.e. generating 2DGSM(REG) or 2GSM(REG). Very often it seems to be difficult to find grammars for 2-way devices, but we might have some chance because we are dealing with 1-way (checking stack) automata! The existence of grammars for 2DGSM(REG) was shown in [Raj 2].

A grammar G may be viewed as a 1-way transducer M and vice versa: a computation of M is a derivation of G, the output of M is the string generated by G, the input of M is the "control string" of G's derivation indicating which elementary steps have to be taken in which order, and the range of M is the language generated by G. Thus our problem is how to transform the 2-way translation process of a 2gsm into a 1-way process of some 1-way transducer. It turns out that the 1-way transducer has to make use of some kind of parallelism (such as that present in expressions or in parallel assignments): the 2-way computation is cut in several 1-way pieces and these pieces are simulated simultaneously. In fact we have already seen how to to do this in section 1: by computing transition tables or by guessing visiting sequences (the results of section 1 all depend on this possibility to transform 2-way into 1-way and then use well-known 1-way techniques).

Generalization of these two constructions in section 1 (used to prove Theorem 2) by incorporating the output of the 2dgsm will lead to two different types of grammar. We will first take the visiting sequence approach and then that of transition tables. In both approaches it is essential that (for a given symbol σ) the translations of a string σu can be expressed in terms of the translations of u. This was apparent for transition tables. For visiting sequences it can be seen from the behaviour of V_k of Lemma 4: after guessing a visiting sequence for σ it continues to work on u and need not return to σ. This essential property shows that the translation can be defined recursively on the structure of the string (the string is build up by prefixing symbols). As an example consider a 1-way gsm. It can be described by (rewriting) rules $q(\sigma x) \to wq'(x)$ where $(q',+1,w) \in \delta(q,\sigma)$ and x stands for any suffix of the input string. Such a rule expresses that the q-translation (i.e. the output when starting in state q) of σx is w followed by the q'-translation of x. A natural generalization of this idea is to allow any number of translations in the right-hand side of the rule. This leads to the following definition of a generalized gsm.

2.1. Generalized gsm mappings and restricted 2-way pushdown transducers

A generalized gsm or *parallel machine* (abbreviated pam) is a tuple $M = (Q,\Sigma,\Delta,q_0,R)$ where Q is a set of states or procedure names, q_0 is the initial state or main procedure name, Σ and Δ are the input and output alphabet respectively, and R is a set of rules or body definitions of the form

$$(*) \qquad q(\sigma x) \to w_1 q_1(x) w_2 q_2(x) \ldots w_n q_n(x) w_{n+1}$$

with $n \geq 0$, $q, q_i \in Q$, $\sigma \in \Sigma \cup \{¢,\$\}$ and $w_i \in \Delta^*$. A sentential form of M is a string $v_1 p_1(u_1) v_2 p_2(u_2) \ldots v_m p_m(u_m) v_{m+1}$ with $p_i \in Q$, $u_i \in \Sigma^*$ and $v_i \in \Delta^*$. Application of rule (*) to a sentential form consists of rewriting an occurence of $q(\sigma u)$, for some $u \in \Sigma^*$, by $w_1 q_1(u) w_2 q_2(u) \ldots w_n q_n(u) w_{n+1}$. The translation realized by M is $\{(w,v) \mid q_0(¢w\$) \overset{*}{\Rightarrow} v\}$ where $\overset{*}{\Rightarrow}$ denotes repeated application of rules. M is *deterministic* (dpam) if for each q and σ there is at most one rule with left-hand side $q(\sigma x)$.

There are three ways of viewing a pam. First as a set of recursive procedures (with parameter and result of type string); rule (*) is part of the body of procedure q and expresses the translation of σx recursively in those of x. Secondly as a (generalized) finite state transducer; (*) means that when M reads σ in state q it splits into n identical transducers which continue to process the rest of the string in parallel (each in its own state). Thirdly as a rewriting system as suggested in the definition. Without loss of generality, derivations may be restricted to be left most and also to be parallel (as in the case of context-free grammars). *Parallel derivation* means that a rule is applied to each "call" $p_i(u_i)$ in the sentential form simultaneously; during a parallel derivation each sentential form is of the form $v_1 p_1(u) v_2 p_2(u) \ldots v_m p_m(u) v_{m+1}$ and thus the parameter u may be kept globally, i.e. outside the sentential form; in that case the rule (*) can be written as σ: $q \to w_1 q_1 w_2 q_2 \ldots w_n q_n w_{n+1}$, i.e. the "control symbol" σ causes the "nonterminal" q to be rewritten by the string $w_1 q_1 \ldots q_n w_{n+1}$. From this parallel rewriting point of view M is called an ETOL system [HerRoz; Roz].

25. EXAMPLE. The translation of Example 1 can be realized by the dpam N with states q_0, q_1, q_3 and the following rules:

$$q_0(¢x) \to q_1(x)q_3(x),$$
$$q_1(ax) \to aq_1(x), \quad q_1(bx) \to \lambda, \quad q_1(\$x) \to \lambda,$$
$$q_3(ax) \to bq_3(x), \quad q_3(bx) \to q_1(x)q_3(x),$$
$$q_3(\$x) \to \lambda.$$

Clearly $q_1(a^{n_1}ba^{n_2}\ldots ba^{n_k}\$) \overset{*}{\Rightarrow} a^{n_1}$ and $q_3(a^{n_1}ba^{n_2}\ldots ba^{n_k}\$) \overset{*}{\Rightarrow} b^{n_1}a^{n_2}b^{n_2}\ldots a^{n_k}b^{n_k}$.

A left-most evaluation of the call $q_0(¢aaba\$)$ is
$q_0(¢aaba\$) \Rightarrow q_1(aaba\$)q_3(aaba\$) \Rightarrow aq_1(aba\$)q_3(aaba\$) \Rightarrow aaq_1(ba\$)q_3(aaba\$) \Rightarrow aaq_3(aaba\$) \Rightarrow aabq_3(aba\$) \Rightarrow aabbq_3(ba\$) \Rightarrow a^2b^2q_1(a\$)q_3(a\$) \Rightarrow a^2b^2aq_1(\$)q_3(a\$) \Rightarrow a^2b^2aq_3(a\$) \Rightarrow a^2b^2abq_3(\$) \Rightarrow a^2b^2ab$.

A parallel derivation is

$q_0(\text{¢aaba\$}) \Rightarrow q_1(\text{aaba\$})q_3(\text{aaba\$}) \Rightarrow aq_1(\text{aba\$})bq_3(\text{aba\$}) \Rightarrow aaq_1(\text{ba\$})bbq_3(\text{ba\$}) \Rightarrow aabbq_1(\text{a\$})q_3(\text{a\$}) \Rightarrow a^2b^2aq_1(\$)bq_3(\$) \Rightarrow a^2b^2ab.$

And an ETOL derivation is

$q_0 \overset{¢}{\Rightarrow} q_1q_3 \overset{a}{\Rightarrow} aq_1bq_3 \overset{a}{\Rightarrow} aaq_1bbq_3 \overset{b}{\Rightarrow} aabbq_1q_3 \overset{a}{\Rightarrow} aabbaq_1bq_3 \overset{\$}{\Rightarrow} a^2b^2ab.$ □

26. EXERCISE. Show that the language $\{w_1\#w_2\#\ldots\#w_n\#w \mid n \geq 1 \text{ and } w = w_i \text{ for some } 1 \leq i \leq n\}$ can be generated by a pam. □

We now want to show that 2DGSM $\subseteq$ PAM. Let $M = (Q,\Sigma,\Delta,\delta,q_0,\{q_f\})$ be a 2dgsm which starts on ¢ and falls off the left of ¢. The pam N which simulates M has state set $Q \times Q$, initial state $\langle q_0,q_f\rangle$ and the same input and output alphabet. The rules of N will be constructed in such a way that $\langle q_1,q_2\rangle(u)$ generates the output produced by M during a computation which starts on the first square of u in state q_1 and terminates by falling off the left of u in state q_2 (just as for transition tables). Moreover, if $\langle q_0,q_f\rangle(\text{¢}w_1w_2\$) \overset{*}{\Rightarrow} v_1\langle q_1,p_1\rangle(w_2\$)v_2\ldots v_m\langle q_m,p_m\rangle(w_2\$)v_{m+1}$ is a parallel derivation, then $v_1,v_2,\ldots,v_{m+1}$ are the pieces of output produced by M on $\text{¢}w_1$ during some computation and $q_1,p_1,\ldots,q_m,p_m$ is the "crossing sequence" of that computation at the boundary between w_1 and w_2, i.e. the sequence of states in which M crosses that boundary (q_i in the +1 and p_i in the -1 direction). Thus N guesses crossing sequences (as V_k in Lemma 4) and at the same time produces the corresponding output at the proper places. Note that each visiting sequence essentially consists of two crossing sequences, one for each boundary of the square. The rules of N are constructed such that each $\langle q,p\rangle$ guesses its own piece of the visiting sequence of the next square. As an example, $\langle q_1,q_3'\rangle$ might guess the piece of Figure 6 (cf. Fig. 3 in section 1), where σ is the next symbol and, e.g. $q_2 \overset{\leftarrow}{\underset{v_2}{\rightarrow}} q_2'$ denotes the 5-tuple $\langle -1,q_2,+1,q_2',v_2\rangle$. The corresponding rule of N is $\langle q_1,q_3'\rangle(\sigma x) \rightarrow v_1\langle q_1',q_2\rangle(x)v_2\langle q_2',q_3\rangle(x)v_3$. Note that q_2 and q_3 have to be guessed; if, however, the pam N has regular look-ahead (in the obvious sense), then it can compute q_2 from q_1' and q_3 from q_2' (because the domain of a 2dgsm is regular, Theorem 2). The above construction was applied in Example 25 to the 2dgsm M of

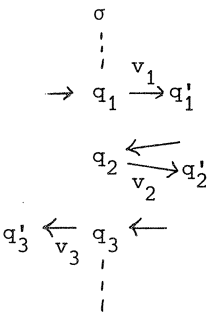


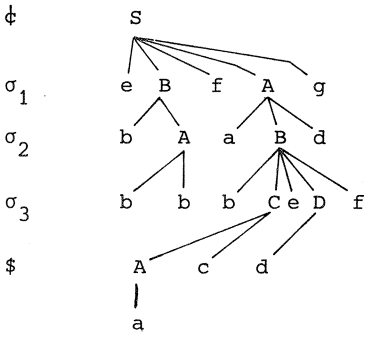
Figure 6. Piece of visiting sequence

Example 1 to which a state q_∞ is added which, at the end, walks from \$ to ¢; in Example 25, q_0 stands for $\langle q_0, q_\infty \rangle$, q_1 for $\langle q_1, q_2 \rangle$ and q_3 for $\langle q_3, q_\infty \rangle$.

Hopefully this has convinced the reader that 2DGSM $\subseteq$ PAM and 2DGSM $\subseteq$ $DPAM^R$. The reverse inclusions are false: the dpam with rules $q_0(¢x) \to q(x)$, $q(ax) \to q(x)q(x)$ and $q(\$x) \to a$ translates a^n into a^{2^n}, but $\{a^{2^n} | n \geq 0\}$ is not even in 2GSM(REG) by Theorem 6 and Corollary 9. Also $\{a^{n^2} | n \geq 1\}$ can be produced by a dpam. To characterize 2DGSM we note that, in the construction sketched above, a parallel sentential form contains at most k calls $\langle q_i, p_i \rangle(u)$, where $k = \#(Q)$, because each q_i is part of the same visiting sequence (of the first square of u). A parallel derivation of a pam is k-*copying* (or, of index k) if each sentential form contains at most k "calls". A pam is k-copying if each of its successful parallel derivations is k-copying (for ranges we only require that there exists such a derivation for each output), and *finite copying* (or, of finite index) if there is such a k (notation: subscripts k and FIN). Intuitively this means that of each suffix of the input string at most k translations appear in the output string. Thus 2DGSM $\subseteq$ $DPAM^R_{FIN}$. We also prove the reverse inclusion.

27. THEOREM. 2DGSM = $DPAM^R_{FIN}$.

PROOF. One inclusion has been shown above. For the other inclusion it suffices to show that $DPAM_{FIN} \subseteq$ 2DGSM because the regular look-ahead can be printed on the tape by a 2dgsm and a dpam can use this information instead (and 2DGSM is closed under composition, Theorem 18). We explain the idea of the proof by a (very simple) example. Let the dpam M have states S.A.B.C.D and (at least) rules $S(¢x) \to eB(x)fA(x)g$, $B(\sigma_1 x) \to bA(x)$, $A(\sigma_1 x) \to aB(x)d$, $A(\sigma_2 x) \to bb$, $B(\sigma_2 x) \to bC(x)eD(x)f$, $C(\sigma_3 x) \to A(x)c$, $D(\sigma_3 x) \to d$ and $A(\$x) \to a$. Let $¢\sigma_1\sigma_2\sigma_3\$$ be an input string and consider Fig. 7. This figure represents a derivation tree of the evaluation of the call $S(¢\sigma_1\sigma_2\sigma_3\$)$ in an obvious way. Since M is k-copying (for some k) the width of the derivation tree is bounded (by k times the maximal length of the right-hand sides of rules). Consequently, the derivation tree can be printed (in some suitable coding) on the input string with all nodes of one level at one square as indicated in Fig. 7. Since this can obviously be done by a 1-way dgsm, it now suffices to define a 2dgsm N which simulates M on input tapes of the form of Fig. 7 (again because 2DGSM is closed under composition). N just makes a left-most traversal through the derivation tree, producing its yield



as output (it keeps a pointer to a node at the current level in its finite control; the level itself is indicated by the input head). Thus N, in some sense, simulates a left-most derivation of M. □

Figure 7. Derivation tree of a pam.

Since the regular look-ahead can be computed in advance, finite copying dpam's generate precisely 2DGSM(REG). It should also be clear that the same holds for finite copying pam's.

28. COROLLARY. [Raj 2]. $2DGSM(REG) = DPAM_{FIN}(REG) = 2GSM_{FIN}(REG) = PAM_{FIN}(REG)$. □

Finite copying pam's are the same as the "absolutely parallel grammars" of [Ray 2] (modulo a slight definitional variation). They are also called ETOL systems of finite index and are investigated as such in [RozVer; Ver; Lat 1; Lat 2]. Corollary 28 also holds for arbitrary L (satisfying the restrictions of the previous section), i.e. $2DGSM(L)$ is generated by the "L-controlled" PAM_{FIN} grammars; see [Gre 1], where it is also shown that for all k $2GSM_{2k}(L) = PAM_k(L)$ which corresponds to the intuition that one pass (i.e. two visits) of the 2gsm over a string corresponds to one translation of the pam, cf. Example 25. Arbitrary L-controlled PAM (ETOL) grammars are investigated e.g. in [Asv].

It is unlikely (cf. the end of section 1.2) that there is a natural subclass of PAM which generates 2GSM(REG), i.e. the 1-way checking stack languages (we saw that a dpam can generate $\{a^{2^n} | n \geq 0\}$). We now turn to the reverse question and ask whether there is a natural class of automata corresponding to pam's. Since a pam is a set of recursive procedures, it can be implemented using a pushdown store in the usual fashion. Moreover, since the recursion closely follows the recursive structure of the string, it turns out that the parameters of the procedures (suffixes of the input

string) need not be stored on the pushdown. A global 2-way pointer to the input string can keep track of the parameter of the procedure call(s) on top of the pushdown (by pointing to the first square of the suffix), cf. the description of a parallel derivation from the ETOL point of view, and Fig. 7. Thus a pam M can be simulated by a 2-way pushdown transducer N! N's method is exactly the same as for the finite copying case (Fig. 7) except that now a pushdown is used to traverse M's derivation tree, simulating a leftmost derivation. N has suffixes of right-hand sides (without parameters) in its pushdown squares indicating which part of the tree still has to be traversed. Thus (taking the same example, see Fig. 8) N starts on ¢ with empty pushdown and moves one square to the right pushing the "symbol" eBfAg. N will evaluate eBfAg in such a way that after evaluation its input head has returned to the same square. N generates output e, changes eBfAg into BfAg and moves one square to the right pushing bA (in order to evaluate the call B). If N has evaluated bA (thereby changing it into the "symbol" λ), it moves one square to the left, popping λ, and changes BfAg *into fAg. Note that*

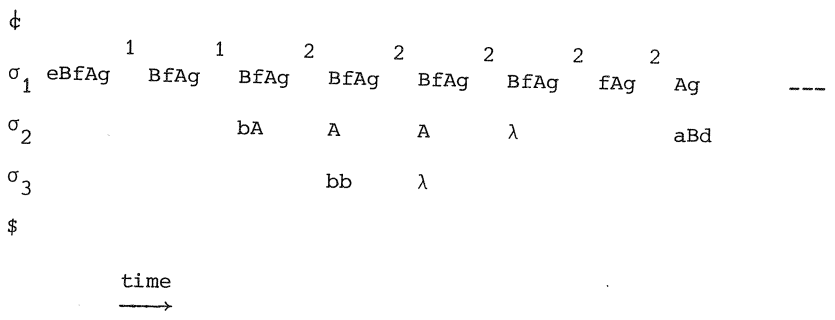


Figure 8. Implementation of a pam on a 2-way pushdown transducer

N always knows what to push because its input head scans the first element of the current parameter. Several stages in the computation of N are shown in Fig. 8 where the input string is put vertical, the pushdown is upside down and the input head scans the square at the same level as the top of the pushdown (the number of intermediate steps of N is also indicated). This shows that the languages in PAM(REG) can be generated by a 2-way pushdown transducer. This does not mean very much because all recursively enumerable languages can be so generated (check that the input string is the encoding of a Turing machine computation). Observe however that the above 2-way

pushdown transducer always moves right when it pushes and moves left when it pops; hence its pushdown always has the same length as the prefix of the input string to the left of the input head. This observation leads to the following automaton, introduced in [vLe].

A *checking stack pushdown transducer* (cs-pd *transducer*) is a 2-way pushdown transducer whose operations on input and storage may only be used coupled in the following way: move right one square and push one symbol, move left one square and pop one symbol (and eventually: do not move the input head and change the top symbol of the pushdown). Thus the input pointer and the pushdown pointer are tied together and forced to move simulta-

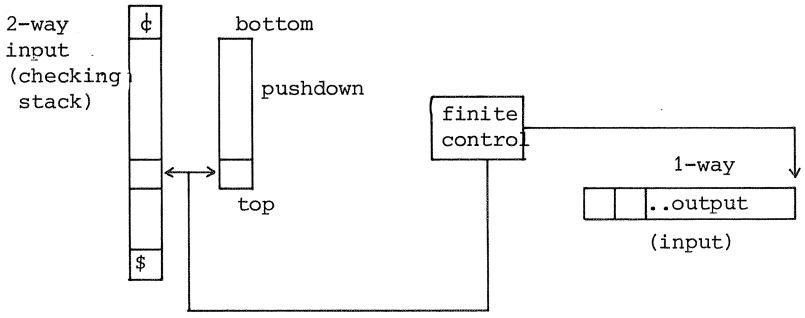


Figure 9. The cs-pd transducer (acceptor).

neously, as suggested in Fig. 9. A cs-pd *acceptor* is obtained in the usual way by viewing input and output of the cs-pd transducer as checking stack and input, respectively. Note that a cs-pd transducer does not satisfy the basic assumption in general automata theory as explained in section 1.3, whereas the cs-pd acceptor does. CS-PD(L) will denote the class of languages generated from L by the cs-pd transducer (or, accepted by the "L-based" cs-pd acceptor). A dcs-pd transducer is a deterministic cs-pd transducer (we will not consider deterministic cs-pd acceptors).

The above arguments show that PAM(REG) $\subseteq$ CS-PD(REG) and, obviously, DPAM(REG) $\subseteq$ DCS-PD(REG). The following characterization result is shown in [vLe]; note that PAM(REG) = ETOL and DPAM(REG) = EDTOL; for the second equality see [EngSch^{V}L; EngRozS].

29. THEOREM. PAM(REG) = CS-PD(REG) *and*
DPAM(REG) = DCS-PD(REG).

PROOF. To prove that DCS-PD(REG) $\subseteq$ DPAM(REG), one can view the pushdown of a dcs-pd transducer M as an additional read/write track on the input tape which has to be erased when moving to the left. Extend the notion of visiting sequence to include the current content of that track (i.e. the top of the pushdown) in the 5-tuples (cf. Exercise 5). Although M is of course not finite visit in general, the proof that 2DGSM $\subseteq$ DPAM^{R}_{FIN} (see Fig. 6) can be carried over to show DCS-PD $\subseteq$ DPAM^{R}: the states of the dpam N are triples $\langle q,\gamma,p\rangle$ meaning that M starts on the first square of a suffix of the input string in state q and with γ on top of the pushdown, and falls off the left of the suffix in state p, popping γ. The return states can be guessed by regular look-ahead because the domain of a cs-pd transducer is regular (Theorem 35). This implies that DCS-PD = DPAM^{R}, cf. Theorem 27. Note that this proof is based on the fact that the length of a piece of visiting sequence (as in Fig. 6) is bounded for a dcs-pd transducer (otherwise it would be in a loop). In the nondeterministic case this is not true any more (a pam translates each input into a finite number of outputs, whereas the number of outputs may be infinite for a cs-pd transducer). However, it is easy to see that the infinitely many pieces of visiting sequence to be guessed form a regular language (i.e. the guessing can be done by a finite state automaton). An element of this language may be guessed step-wise if arbitrarily many dummy symbols are interspersed with the input symbols; for details, see [vLe; EngRozS]. □

Due to the coupling of input and storage, the analogue of Theorem 19 is not true for DCS-PD: the use of the language $\{a^n b^n \mid n \geq 1\} \in$ 2DGSM(REG) as input to a dcs-pd transducer can produce an output language not in DCS-PD(REG). Nevertheless, Theorem 18 (composition closure of 2DGSM) can be generalized as follows.

30. THEOREM. DCS-PD $\circ$ 2DGSM $\subseteq$ DCS-PD.

PROOF. We will show that DPAM^{R} $\circ$ 2DGSM $\subseteq$ DCS-PD, cf. the previous proof. Let M be a dpam^{R} and N a 2dgsm. Let Fig. 7 be a derivation tree of M. The output of M for input $\dashv\sigma_1\sigma_2\sigma_3\$$ lies wrapped up as the yield of this tree. Analogous to the case of visiting sequences, the 2dgsm N walking on this output can be simulated by walking along the paths of the tree, as shown in Figure 10

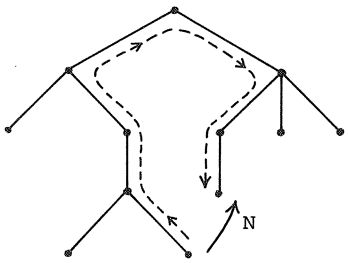


Figure 10.

N moves right on the yield.

(for a different tree). Although the derivation tree is not present on the input tape, the necessary nodes of the tree can be constructed using the pushdown facility as in the proof of PAM $\subseteq$ CS-PD (Fig. 8); since this time the tree is traversed to and fro, the dcs-pd transducer should not erase symbols of right-hand sides in its pushdown squares, but has instead a pointer in each right-hand side indicating the topnode of the subtree being processed (as in the proof of Theorem 27). Note finally that the regular look-ahead of M can easily be computed: the current input square can be marked on the pushdown and the input head can walk to the right (pushing dummy symbols) and walk left computing the regular look-ahead until it finds back the marked square. []

Hence DCS-PD(REG) is closed under 2dgsm mappings [EngRozS].

2.2. Tree transducers

The results of the previous subsection can be generalized straightforwardly to tree transducers, i.e. transducers which obtain a tree (preferably a derivation tree of some sort) as input and translate it into another tree or a string. Such transducers are studied because of their relevance to the syntax-directed translation of context-free languages. In fact, the model of (generalized) syntax-directed translation defined in [AhoUll 2,3] is equivalent to the top-down tree transducer of [Rou; Tha 1]. Both models formalize the recursive translation of a tree, using its recursive structure. Thus the top-down tree transducer generalizes the pam in a natural way. (In [Tha 1] tree transducers are called g^2sm mappings, here it would be appropriate to call them g^3sm mappings!). We will only sketch the generalization to trees, a detailed survey (also for the string case) can be found in [EngRozS].

A *top-down tree-to-string transducer* M is the same as a pam except that its rules are of the form

$$q(\sigma[x_1,\ldots,x_m]) \to w_1q_1(x_{i_1})w_2q_2(x_{i_2})\ldots w_nq_n(x_{i_n})w_{n+1}$$

meaning that the q-translation of a tree with topnode labeled σ and subtrees $x_1,\ldots,x_m$ (in that order) is expressed as a concatenation of output symbols and translations of the subtrees (for $1\leq s\leq n$: $1\leq i_s\leq m$). In case the right-hand side is a (coding of a) tree (with $q_s(x_{i_s})$ at the bottom) M is a top-down tree (to tree) transducer. The class of top-down tree transductions is denoted by T (rather than TTT), and the class of top-down tree-to-string transductions by yT (where y denotes the mapping which associates with each tree its yield; clearly yT = T∘y). Determinism is indicated by D. The role of REG is taken over by the class RECOG of recognizable tree languages, which is strongly related to derivation tree languages of context-free grammars: y(RECOG) = CF [Tha 2], whence the relationship to syntax-directed translation of CF.

The results of section 2.1 for strings are generalized to trees by putting the strings vertical (as we already did in Figures) and imagining they are trees. In particular the notion of suffix generalizes to that of subtree. Thus a tree transducer is finite copying if it makes a bounded number of translations of each subtree of the input tree. Also, a tree-to-string transducer can be implemented on a 2-way tree-to-string pushdown transducer, called *checking tree pushdown transducer* (ct-pd transducer), which keeps a pointer to the top of that subtree which is the current parameter.

A ct-pd *transducer* has an input head pointing to a node of the input tree, a usual pushdown and a usual 1-way output tape, and it works by moving down and up the tree (down to a specified son, up to the father) simultaneously pushing and popping symbols, respectively (see Fig. 11). Since the

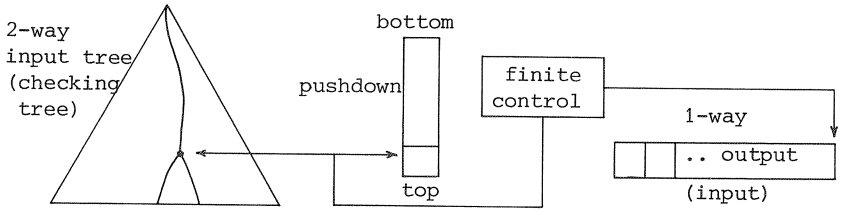


Figure 11. The ct-pd transducer (acceptor).

machine starts with empty pushdown, the pushdown always has the same length as the path from the root to the input pointer (thus one may also imagine, as before, that each node of the input tree has an additional read/write square which has to be erased when moving up to the father). A ct-pd transducer is finite visit if the number of visits of each node is bounded. The

proof of the next theorem is the same as in the string case (Theorems 7 and 29, Corollary 28).

31. THEOREM.

(i) yT(RECOG) = CT-PD(RECOG) *and*
yDT(RECOG) = DCT-PD(RECOG).

(ii) $yDT_{FIN}(RECOG) = DCT(RECOG) = CT_{FIN}(RECOG)$. □

yDT(RECOG) is the class of languages which can be obtained from the context-free languages by deterministic (generalized) syntax-directed translation [AhoUll 2]. DCT denotes all dct-pd transducers which do not make use of the pushdown; thus they are 2-way finite state tree transducers, the generalization to trees of the 2dgsm. These tree-walking dct transducers were introduced in [AhoUll 2], where also the relationship to finite copying was established (first equality of Theorem 31(ii)). The restriction of their proof to vertical strings gives the proof of Corollary 28 [Ray 2].

An inclusion diagram of classes of ranges of transducers is shown in Figure 12, where REG and RECOG are left out, CS and DCS stand for 2GSM and 2DGSM, a path means inclusion and no path incomparability. This figure shows that variations of the same machine model characterize several different defining mechanisms (such as the top-down tree-to-string transducer, the ETOL system, the 2-way finite state transducer).

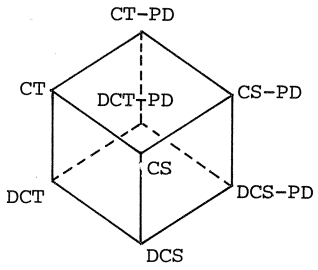


Figure 12.
CT-PD automata classes

The tree-walking trick used in the proof of Theorem 30 (Figure 10) can be realized by a dct transducer. Thus, a 2dgsm walking on the yield of a tree can be simulated by a dct transducer walking on that tree, and similarly in the nondeterministic case. Using Theorem 31 this can be expressed as follows, where L is a class of tree languages satisfying certain closure properties.

32. THEOREM. $2DGSM(y(L)) \subseteq yDT_{FIN}(L)$ *and*
$2GSM(y(L)) \subseteq yT(L)$. □

Taking L = RECOG implies $2DGSM(CF) \subseteq yDT_{FIN}(RECOG)$ and $2GSM(CF) \subseteq yT(RECOG)$. Thus 2gsm mappings of context-free languages can be defined by syntax-directed translation! Taking L = DT(RECOG) and using the fact that DT is (almost) closed under composition, it even shows that yDT(RECOG) is

closed under 2dgsm mappings.

Iteration of the second inclusion in Theorem 32 implies that $2GSM^n(CF) \subseteq yT^n(RECOG)$, where yT^n denotes the composition of n top-down tree transducers followed by yield. $\{yT^n(RECOG)\}_n$ is a proper hierarchy and the counter-examples can already be found in the sub-hierarchy $\{2GSM^n(REG)\}_n$ [Eng 1], cf. Theorem 14.

33. EXERCISE. Show that DPAM(REG) and yDT(RECOG) are closed under c_*. □

2.3. Register grammars and macro grammars

We now turn to the transition table approach to find grammars for 2DGSM(REG). We will try to incorporate the output of the 2dgsm M in the construction of the (first) proof of Theorem 2. Addition of output to the 1-way (right-to-left) finite state automaton A computing the transition tables then results in a 1-way transducer (grammar) for 2DGSM(REG). As discussed before, the existence of A is based on the fact that $R_{\sigma u}$ is determined (and can be computed from) σ and R_u; similarly, the output strings involved in computations of M on σu can be expressed in terms of those on u. If we provide A with a register $x[q_1,q_2]$ of type string, for each pair of states (q_1,q_2) of M, then it can use this to store the output of M resulting from a "(q_1,q_2)-computation" on u. A updates its registers by assignments, as follows (cf. (*) in the proof of Theorem 2, and Fig. 3):

(**) if there is a sequence of states $q_1,q_1',\ldots,q_{n-1},q_{n-1}',q_n,q_n'$ of M such that $1 \leq n \leq \#(Q)$,

$$\delta(q_i,\sigma) = (q_i',+1,v_i) \quad \text{for } 1 \leq i \leq n-1,$$
$$\delta(q_n,\sigma) = (q_n',-1,v_n), \quad \text{and}$$
$$(q_i',q_{i+1}) \in R_u \quad \text{for } 1 \leq i \leq n-1,$$

then (q_1,q_n') is in $R_{\sigma u}$ and the assignment

$$x[q_1,q_n'] := v_1x[q_1',q_2]v_2x[q_2',q_3]v_3\ldots v_{n-1}x[q_{n-1}',q_n]v_n$$

should be executed by A.

(Note that the sequence $q_1,\ldots,q_n'$ is unique for given q_1 and σ, because M is deterministic and R_u is a function.) Thus A, reading σ, changes its register contents by a parallel assignment of the form $(x_1 := w_1, x_2 := w_2,\ldots,x_m := w_m)$ with $w_i \in (\Delta \cup \{x_1,\ldots,x_m\})^*$, where $x_1,\ldots,x_m$ are all registers $x[q_1,q_2]$. At the end of A's computation register $x[q_0,q_f]$ contains the output of M.

34. EXERCISE. Define a "1-way register transducer" which can handle the above computation. Do this formally by specifying a storage type and a way of manipulating the input and producing output, as indicated in section 1.3. □

Observe that there is a direct correspondence between the assignments in (**) and the rules of the pam corresponding to M (see the argument before Theorem 27). Actually the register transducer can be viewed as an (iterative) bottom-up computation of the values of the recursive procedure calls of the pam. Thus the difference between the visiting sequence and transition table approaches corresponds, when generalized to trees, to the difference between top-down and bottom-up computation on trees (cf. [Eng 2]). Also, if the rules of the pam (or top-down tree transducer) are viewed as recursive functional equations, then the register machine may be viewed to perform an iterative computation of the least fixed point of these equations (in the usual way).

If we use the 1-way register transducer to generate languages, then the input can be replaced by nondeterminism. The corresponding flowcharts are a very simple class of nondeterministic register programs for the generation of languages: they only use parallel assignments. Let RP denote the class of generated languages. Thus 2DGSM(REG) $\subseteq$ RP. As an example, the language $\{a^{n^2} | n \geq 0\}$ is generated (in register y) by nondeterministically iterating the parallel assignment (x := ax, y := yxxa) starting with (x:= λ, y := λ). This example shows that RP is more than 2DGSM(REG).

The transition table approach can be extended to the dcs-pd transducer. Due to the forced dependence between input and pushdown, the behaviour of a dcs-pd transducer M on a suffix u of the input string is determined by the state q in which M enters the first square of u and the symbol γ it just pushed on the pushdown (cf. the proof of Theorem 29). Thus a transition table of M can be defined as a function R: $Q \times \Gamma \to Q$, where Γ is the set of pushdown symbols, such that $R(q,\gamma)$ is the state M is in when falling off the left of u, popping γ. In the nondeterministic case R is a relation $\subseteq (Q\times\Gamma)\times Q$. Similar to the pushdown-less case, this gives the following well-known result (extending Theorem 2).

35. THEOREM. *The domain of a* cs-pd *transducer is regular.* □

In terms of (ETOL) grammars this means that the set of control strings which lead to success, also called the Szilard language, is regular. Generalized to trees it can be shown similarly, using a bottom-up finite state

tree automaton [ThaWri], that the domain of a top-down tree transducer is in RECOG.

It is easy to check that, as for a 2dgsm, a register program can compute the translation of a dcs-pd transducer. We leave it as an exercise to the reader to show that, vice versa, each language generated by a register program is the range of a dcs-pd transducer (use the "duality" between assignments and pam rules), cf. [EngSch^{V}L].

36. THEOREM. RP = DCS-PD(REG). []

This result was proved in [Dow], for DPAM(REG) i.e. EDTOL.

It is rather straightforward to see that 2DGSM(REG) corresponds precisely to the "noncopying" register programs [EngRozS], which only use assignments $(x_1 := w_1,\ldots,x_m := w_m)$ such that each x_i occurs at most once in $w_1w_2\ldots w_m$. A more careful proof shows that 2k-visit 2dgsm correspond to noncopying register programs with at most k registers (and to k-copying pam).

Register programs are such a natural language-generating device that they form an additional motivation to study DCS-PD(REG). It follows from an extension of Theorem 15 that not all context-free languages are in RP [EhrRoz; EngRozS; Lat 2]. It is easy to see that all derivation-bounded context-free languages are in 2DGSM(REG). It has recently been shown [Lat 3] that no full AFL generator of CF is in RP.

The language generated by a register program can be recognized by simulating each register by two pointers to the string to be recognized, pointing to the substring which is the content of the register (if a register content is not a substring of that string, then it can be disregarded). Using some more pointers the parallel assignments can easily be simulated by pointer manipulation only. This shows that RP can be recognized by nondeterministic multi-head finite state automata, i.e. in nondeterministic logarithmic space [JonSky 1]. On the other hand, 2GSM(REG) contains an NP-complete language [Hun].

In the nondeterministic case, CS-PD(REG) can be characterized by "extended" register programs which have languages in their registers (each register containing the set of possible translations of a given suffix), see [Dow;EngSch^{V}L;AsvEng] and cf. the proof of Theorem 29. Note that CF $\subseteq$ CS-PD(REG): just disregard the checking stack (which should be "guessed" long enough); the corresponding extended register program computes the least fixed point of the context-free grammar viewed as a system of recursive equations.

Register programs are related to *macro grammars* [Fis 2]. If the state

of the finite control of the register program changes from F to G executing assignment $(x_1 := w_1,\ldots,x_m := w_m)$, then the corresponding macro grammar has a rewriting rule $F(x_1,\ldots,x_m) \rightarrow G(w_1,\ldots,w_m)$. For example, the macro grammar with rules $S \rightarrow F(\lambda,\lambda)$, $F(x,y) \rightarrow F(ax,yxxa)$ and $F(x,y) \rightarrow y$ will generate $\{a^{n^2} | n \geq 0\}$. In general a macro grammar is just a set of (nondeterministic) recursive procedures (such as F,G) with bodies defined by its rules. The difference with a pam is that a "macro grammar procedure" has any number of parameters, builds up its parameters by concatenation (instead of breaking them down) and can have calls nested in other calls. Macro grammars can also be viewed as context-free grammars in which the nonterminals have parameters. These parameters can be used to define non-contextfree features of the syntax of programming languages. The class of all macro grammars generates the indexed languages, which are characterized by the nested stack automaton [Aho]. Natural restrictions on macro grammars (making them look like register programs) define 2DGSM(REG), DCS-PD(REG) and CS-PD(REG).

A *stack* is a pushdown which may also be used as a 2-way read-only tape; it has only one pointer which is either inside the stack (for reading) or on top of the stack (for pushing and popping) [GinGreH]. A 1-way (nondeterministic) stack automaton can easily simulate a 1-way (nondeterministic) checking stack automaton [Gre 3]. In [EngSch^{V}L] the 1-way stack pushdown (s-pd) automaton is considered which, as the cs-pd automaton, has a read/write second track on its stack which should be erased when the stackpointer moves up. The s-pd automaton is a very restricted case of the nested stack automaton (each pushdown square is a nested stack of size one), and the cs-pd automaton even more. The diagram in Fig. 13 shows some classes of languages defined by stack-like machines (where S denotes "stack"), cf. Figure 12. A peculiar kind of determinism can be defined to name the two unlabeled nodes of Fig. 13. Natural classes of macro grammars have been found for all automata except stack and checking stack. Gluing Figures 12 and 13 together shows that tree transducer languages and macro languages are related via PAM(REG) or ETOL languages, and that this relationship can be "explained" by simple variations in the machine model. The addition of the "pushdown facility" to stack-like automata seems to improve the chance to find grammars (which are so helpful

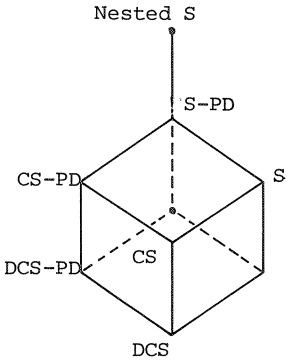


Figure 13.
Stack-like automata classes.

in formal proofs); another possibility is to impose the finite visit property (or some kind of determinism). Correctness of the diagrams of Figures 12 and 13 together is shown in [EngSch^{V}L ; EngRozS].

37. EXERCISE. [vLe]. Show that a nonerasing (i.e. nonpopping) stack automaton can be simulated by a cs-pd automaton. Show the same for a stack-counter automaton (i.e. a stack automaton whose pushdown alphabet is a singleton). Hint: the top of the stack can be marked on the pushdown. □

38. EXERCISE. [Gre 3]. Using transition tables, show that a 1-way nonerasing stack acceptor which does not read input when reading in its stack, accepts a regular language. Show that a 1-way stack automaton with the same property accepts a context-free language (Hint: store the transition tables on the stack). □

3. TWO-WAY AUTOMATA AND COMPLEXITY

Space-bounded Turing machines are typical examples of automata which do not satisfy the basic assumption of general automata theory (as discussed in section 1.3): the amount of storage depends on the input. Similarly time-bounded Turing machines are not acceptable because of the (time) restrictions imposed on the programs for the machine. Surprisingly, certain types of 2-way (stack) automata, which do satisfy the basic assumption, turn out to characterize some of the space and time complexity classes. Moreover, if these automata are allowed to have several input heads (i.e. are multi-head automata), then still more complexity classes are obtained. In general the advantage of an automaton characterization of a complexity class is that the automaton can be programmed freely without caring for space or time, and one can be sure that an equivalent efficient algorithm exists. The main known results can be found in [HopUll 2 or 3; Fis 1; Coo 1,2; Iba; Gre 2; vLe].

3.1. Two-way checking automata

The next simplest device with a 2-way read-only tape (after the 2-way finite state transducer or 1-way checking stack acceptor) is the 2-way checking stack acceptor. A nondeterministic 2-*way checking stack automaton* (2csa) has two 2-way read-only tapes: an input tape and a checking stack.

Thus it is essentially a 2-tape 2-way finite state automaton accepting a binary relation of which only the domain (or range) is of interest.

Formally a 2csa is as usual a tuple $M = (Q,\Sigma,\Gamma,\delta,q_0,F)$ of states, input symbols, stack symbols, transition function, initial state and final states respectively, and δ is a function from $Q \times (\Sigma \cup \{¢,\$\}) \times (\Gamma \cup \{¢,\$\})$ into the finite subsets of $Q \times \{-1,0,+1\} \times \{-1,0,+1\}$ with obvious meaning. The set of directions of the stack-head may be restricted to $\{-1,+1\}$. As usual, $2CSA(L)$ is the class of languages accepted by L-based 2csa, i.e. 2csa whose checking stack is preset by a string from a given language in L.

39. EXERCISE. Show that $2DGSM^{-1}(2CSA(L)) = 2CSA(L)$ and $2CSA(2DGSM(L)) = 2CSA(L)$. □

The notions of transition table and visiting sequence can be generalized from the 1-way to the 2-way csa (i.e. they pertain to the checking stack, not to the input tape): in the definitions, the states (and piece of input) of the 1-way csa should be replaced by the states of the 2csa together with the positions of the input head (and the movements of the input head should be dealt with). Thus the 5-tuples $\langle d,q,d',q',v\rangle$ in the visiting sequence of a 2gsm (1csa) are replaced by 6-tuples $\langle d,q,x,d',q',x'\rangle$ with $d,d' \in \{-1,+1\}$, $q,q' \in Q$ and $0 \leq x,x' \leq n+1$ where n is the length of the input. A *visiting sequence* of a 2csa M for a given input w is a sequence $s = (A,\langle...\rangle,...,\langle...\rangle)$ such that $A \in \Gamma$ and each 6-tuple $\langle d,q,x,d',q',x'\rangle$ in s satisfies $(q',x'-x,d') \in \delta(q, input[x],A)$, where "input" is the input array [0:n+1] containing ¢w$. Note that, since $|x'-x| \leq 1$, we can put an element of $\{-1,0,+1\}$ in the place of x' in all 6-tuples. As in the 2gsm case, a computation of M on some input ¢w$ and checking stack ¢v$ is represented by a fitting sequence of visiting sequences (for input w) whose stack symbols spell ¢v$. It may be assumed that all 6-tuples in a visiting sequence are different, because the computation between two identical 6-tuples may be skipped. Thus the same remarks hold for *visiting sets* (except that a sequence of visiting sets may represent more than one computation).

The relationship of the 2csa to space-bounded Turing machines is stated in the next theorem, where NSPACE(n) denotes the class of languages accepted by nondeterministic Turing machines in linear space: the context-sensitive languages.

40. THEOREM. [Fis 1]. 2CSA = NSPACE(n).

PROOF. We first show that 2CSA $\subseteq$ NSPACE(n). An input ¢w$ is accepted by a 2csa M if there exists a checking stack such that M makes some successful computation on ¢w$ and that checking stack. A linear bounded Turing machine N can be constructed which, with input w, guesses one by one the elements of a fitting sequence of visiting sets for that input (cf. Lemma 4); thus it guesses both a checking stack and a computation of M. The idea of the construction is that one visiting set may be represented on a tape of length n (or n+2) by storing the 6-tuple <d,q,x,d',q',x'> on the x-th square of that tape as a 5-tuple <d,q,d',q',e> where e = x'-x (there are only a bounded number of such 5-tuples). The checking stack symbol is kept in the finite control of N. Thus N has one tape of length n+2 with three tracks; the first track contains the input ¢w$, the second track contains the representation of some visiting set and the third track is used to guess the next (fitting) visiting set. It should be clear that N can realize such a guess by moving through the second track and "connect" each of its 6-tuples <d,q,x,d',q',x'> such that d = -1 or d' = +1 to a corresponding 6-tuple on the third track; note that the position of this new 6-tuple (on the track) differs at most one square from that of the old one; note also that N can check the consistency of the connection by reading input[x]. This proof makes an essential use of visiting sets; a visiting sequence would take nlogn space: logn space to store a 6-tuple ($0 \leq x \leq n+1$) and there may be n of them.

To show NSPACE(n) $\subseteq$ 2CSA, let M be a Turing machine with a tape of length n exactly. The 2csa N guesses a computation of M on input w by checking that its stack contains a string $w_1 \# w_2 \# \ldots \# w_m$ where $w_1, w_2, \ldots, w_m$ are the (usual) instantaneous descriptions of M's computation. N first checks that $w_1 = q_0 w$, where q_0 is M's initial state. It then checks for each i whether w_{i+1} is a possible successor of w_i: w_{i+1} should be the same as w_i except for a local change due to a move of M. N compares w_i and w_{i+1} symbol by symbol; since the distance between corresponding symbols is n+1, this can be done by counting to n+1 on the input: it takes the input head exactly n+1 steps to travel from ¢ to $. Clearly the possible local changes can be kept in N's finite control. The essential property of the 2csa in this proof is that it can measure off n squares using the input as a "yardstick". □

Note that the theorem explicitly concerns acceptors; if both automata

are given a (1-way) output tape, the construction (in one direction) does not work.

The proof of Theorem 40 in [Fis 1] is amusing. It uses the intermediate concept of a "bug-automaton" which is a 2-way finite state automaton with a read-only rectangle (divided into labeled squares) as input; 2-way (or better 4-way) means as usual that the input head ("bug") can move to all possible neighbors of a square; the automaton accepts the string on the first row of the square. The bug-automaton is equivalent to the 2csa because two one-dimensional pointers are equivalent to one two-dimensional pointer. The bug-automaton recognizes NSPACE(n) because a rectangle can contain a derivation of a context-sensitive grammar, and a linear bounded Turing machine can guess a rectangle (with bug visiting sets) line by line.

Theorem 40 characterizes NSPACE(n) by a very simple unrestricted 2-way automaton. Note that the 2csa needs 2^n space to have the Turing machine computation on its checking stack.

Note that by Theorem 22 (taking X to be the checking stack) 2CSA = $2GSM^{-1}(1CSA)$ and hence, by Theorem 40, NSPACE(n) = $2GSM^{-1}(2GSM(REG))$ which means that each context-sensitive language is characterized by two 2gsm mappings.

As observed above, each 2csa M is "linear visit", i.e. for each input of length n there is a computation of M which visits each square of the checking stack at most kn times for some constant k. Let us see what happens if the 2csa is required to be finite visit ($2CSA_{FIN}$). Let NSPACE(log n) = NLOG be the class of languages accepted by a nondeterministic Turing machine (with a 2-way read only input tape) in log n space (on its working tape). It is well-known (see e.g. [Har]) that NSPACE(log n) is the class of languages accepted by nondeterministic multi-head 2-way finite state automata. Note that this is already an automaton characterization of NLOG.

41. THEOREM. [Gre 2]. $2CSA_{FIN}$ = NSPACE(log n).

PROOF. The proof is a variant of the previous one.

(i) $2CSA_{FIN} \subseteq$ NSPACE(log n). Let the 2csa be k-visit. Since each visiting set contains at most k 6-tuples <d,q,x,d',q',x'>, k input heads can point to the x-coordinates of these 6-tuples and the other coördinates can be kept in the finite control (of a multi-head finite state automaton). A next visiting set can easily be guessed, moving each head at most one square.

(ii) NSPACE(log n) $\subseteq$ $2CSA_{FIN}$. Let M be a k-head 2-way fsa. An instantaneous description of M consists of the input string w with k markers to indicate the position of the heads on w (and M's state). As before a 2csa N checks a sequence $w_1 \# w_2 \# \ldots \# w_m$ of instantaneous descriptions of M. This time N first checks whether w is the "underlying" string of all w_i (in one sweep). Then it checks for all i the changes in the markers between w_i and w_{i+1}; to do this it makes one sweep over w_i to determine the symbols read by the heads and thus their next moves, and k sweeps over $w_i \# w_{i+1}$ to check these moves (using the input as a yardstick). This shows that N is finite visit. □

In the finite visit case we may allow writing on the checking stack as before (Exercise 5 and the remark following Theorem 7) [Gre 2]. Thus NSPACE(log n) is the class of languages accepted by Turing machines with a 2-way read-only input tape, and a working tape on which they are finite visit.

Note that, by Theorem 22, $2CSA_{FIN} = 2GSM^{-1}(1CSA_{FIN})$ and so, by Theorems 41 and 7, NSPACE(log n) = $2GSM^{-1}$(2DGSM(REG)); cf. [Kie] for the fact that a finite visit checking stack is a storage type. Since NSPACE(log n) $\not\subseteq$ NSPACE(n), this implies that $2GSM^{-1}$(2DGSM(REG)) $\not\subseteq$ $2GSM^{-1}$(2GSM(REG)). Note that this gives an alternative proof that 2DGSM(REG) $\not\subseteq$ 2GSM(REG), cf. section 1.2.

A space-bounded Turing machine may be viewed as a general automaton (from the point of view of section 1.3) if we consider the bounded amount of space as a way of handling the input (contemplate the sequence "1-way, 2-way, multi-head 2-way = logarithmic space, linear space,..."). Following [Coo 2] we can define the following automaton type for each storage type X and function S(n): an <u>NSPACE(S(n)) auxiliary X acceptor</u> is a nondeterministic acceptor which has storage type X and handles its input by way of a 2-way read-only input tape and an additional S(n) bounded Turing machine tape. A similar definition holds in the deterministic case (DSPACE(S(n))). Depending on your point of view, "auxiliary" refers to X [Coo 2] or to NSPACE(S(n)). Thus an S(n)-bounded Turing machine is an NSPACE(S(n)) auxiliary finite state automaton! An NSPACE(log n) auxiliary X acceptor is equivalent to a multi-head (2-way) X acceptor, and an NSPACE(n) auxiliary X acceptor to a "writing X acceptor" which has a read/write input tape (of fixed length). Theorems 39 and 40 are extended to arbitrary multi-head and writing acceptors in the next theorem of [Gre 2] (only (2) is actually

stated there).

42. THEOREM. *Let* X *be a storage type and* $\mathcal{L}_1^N(X)$ *the corresponding class of nondeterministic* 1*-way languages.*

(1) $2CSA(\mathcal{L}_1^N(X))$ *is the class of languages accepted by nondeterministic writing* X *acceptors.*

(2) $2CSA_{FIN}(\mathcal{L}_1^N(X))$ *is the class of languages accepted by nondeterministic multi-head* X *acceptors.*

PROOF. (i) $\subseteq$: The proof is exactly the same as in Theorems 40 and 41. Each time a new visiting set is guessed, the corresponding stack symbol is "fed" into X storage which is programmed to accept only strings from the preset-language. Note that the "feeding" is 1-way.
(ii) $\supseteq$: The proof is obtained by combining the previous proofs with an appropriate version of the basic Lemma 20. Thus the class of languages accepted by nondeterministic multi-head X acceptors consists of all languages $M^{-1}(L_{F'})$ where M is a nondeterministic multi-head finite state transducer and $L_{F'}$ is the language of all instruction sequences (over F') which lead from an initial to a final configuration of the storage X (F' is a finite set of operations on storage). To accept a string in $M^{-1}(L_{F'})$ the 2csa N guesses and checks a string $w_1\$v_1{}^{\#}w_2\$v_2\ldots{}^{\#}w_m\$v_m$ where $w_1,\ldots,w_m$ is a sequence of instantaneous descriptions of M as before and $v_1,\ldots,v_m$ are the corresponding pieces of output. If we now preset N's checking stack with the language L of all strings which are in $L_{F'}$ when symbols not in F' are disregarded, then N accepts $M^{-1}(L_{F'})$. Clearly $L_{F'}$ can be accepted by a 1-way X acceptor; thus N is an $\mathcal{L}_1^N(X)$-based finite visit 2csa. The same proof is valid for the writing case. □

It is easy to see that an $\mathcal{L}$-based 2csa which has a 1-way checking stack (notation $2CSA_{1\text{-way}}$) accepts $2GSM^{-1}(\mathcal{L})$. Thus, by Theorems 22 and 42, we obtain the following increasing sequence (where $\mathcal{L} = \mathcal{L}_1^N(X)$):

$$
\begin{aligned}
2CSA_{1\text{-way}}(\mathcal{L}) &= \text{2-way X acceptor} &&= 2GSM^{-1}(\mathcal{L}) \\
2CSA_{FIN}(\mathcal{L}) &= \text{multi-head X acceptor} &&= 2GSM^{-1}(2DGSM(\mathcal{L})), \\
2CSA(\mathcal{L}) &= \text{writing X acceptor} &&= 2GSM^{-1}(2GSM(\mathcal{L})).
\end{aligned}
$$

By adding heads to the 2csa higher complexity classes can be obtained. In fact the multi-head 2csa corresponds to polynomial space on Turing machines. To simplify the proof we will use transition tables. A *transition*

table of a k-head 2csa M for a given input ¢w$ and checking stack u is a binary relation R_u on $Q \times [0:n+1]^k$ such that $\langle p,x_1,\ldots,x_k,q,y_1,\ldots,y_k\rangle \in R_u$ iff M when started in state p with its stack-head on the first square of u and its i-th input-head at position x_i, terminates by falling off the left of u in state q with its i-th input-head at position y_i. Let P-SPACE denote $\bigcup_k$ NSPACE$(n^k) = \bigcup_k$ DSPACE(n^k).

43. THEOREM. [Iba]. Multi-head 2CSA = P-SPACE.

PROOF. It is easy to show that NSPACE(n^k) can be accepted by the k-head 2csa, as in the proof of Theorem 40 (note that k heads on an input of length n can count to n^k). Vice versa, a proof similar to the one of Theorem 40 can be given to show that a k-head 2csa can be simulated in space n^k, and hence k-head 2CSA = NSPACE(n^k) for all k. To prove the statement of this theorem it suffices to use the rougher method of transition tables. A given k-head 2csa on input ¢w$ can be simulated by a Turing machine which guesses a checking stack symbol by symbol and computes the corresponding transition tables (cf. the proof of Theorem 2). Since a transition table is an $n^k \times n^k$ boolean matrix (roughly), it can be stored in space n^{2k}. It is easy to see that this amount of space also suffices to compute a new transition table (see [HopUll 2]). The actual proof in [Iba] uses translational arguments which reduce the problem to the 1-head case. □

44. EXERCISE. Show that Multi-head 2CS-PD = P-SPACE, i.e. the multi-head 2-way cs-pd acceptor defines P-SPACE [vLe]. Show that the multi-head 2NESA (2-way nonerasing stack automaton) also defines P-SPACE [Iba], cf. Exercise 37. It is proved in [HopUll 2,3] that NSPACE$(n^2) \subseteq$ 2NESA (and vice versa); show that 2CS-PD = NSPACE(n^2) = 2NESA. □

Theorem 43 can again be generalized to show that the use of an auxiliary checking stack results in an exponential jump in space [Iba]: if $S(n) \geq \log n$, then a language is accepted by an NSPACE(S(n)) auxiliary 2csa if and only if it is in NSPACE$(2^{cS(n)})$ for some constant c. Combining this with the method of Theorem 42, we obtain the following general result.

45. THEOREM. *Let* X *be a storage type and* $S(n) \geq \log n$. *Then* NSPACE(S(n)) aux CSA(L_1^N(X)) *is the class of languages accepted by an* NSPACE$(2^{cS(n)})$ aux X *acceptor for some constant* c.

PROOF. Exercise (note that space S(n) can be used to count to $2^{cS(n)}$; note that a transition table for an NSPACE(S(n)) aux csa can be stored and updated in space $2^{cS(n)}$ for some c). □

46. EXERCISE. Using Theorem 45 and the remark following Theorem 42, prove that the classes $2GSM^{-1}(2GSM^{k}(REG))$ form increasingly larger space complexity classes $U_c NSPACE(f_k(cn))$ where $f_1(n) = n$ and $f_{k+1}(n) = 2^{f_k(n)}$. Prove a similar result for $2GSM^{-1}(2DGSM(2GSM^{k}(REG)))$. □

This exercise thus provides an alternative proof that $2GSM^{k}(REG)$ is a proper hierarchy (Theorem 14). It also shows, using a result of Ritchie, that $2GSM^{-1}(U_k 2GSM^{k}(REG))$ is the class of "elementary" languages (cf. e.g. [Arb, section 6.3]).

We now turn to time complexity. Let NTIME(f(n)) denote the class of languages accepted by (multi-tape) nondeterministic Turing machines in f(n) steps, and similar for DTIME(f(n)). Let NP-TIME = $U_k NTIME(n^k)$ and P-TIME = $U_k DTIME(n^k)$. Recall that DSPACE(log n) $\subseteq$ NSPACE(log n) $\subseteq$ P-TIME $\subseteq$ P-SPACE. P-TIME can be characterized by multi-head 2-way (deterministic) pushdown acceptors (2(D)PDA); in general the use of an auxiliary pushdown gives an exponential jump from space to time. We state this basic result without proof.

47. THEOREM. [Coo 2].

(1) Multi-head 2PDA = Multi-head 2DPDA = P-TIME.

(2) *For* S(n) $\geq$ log n, NSPACE(S(n)) aux PDA = DSPACE(S(n)) aux PDA = $U_c DTIME(2^{cS(n)})$. □

Using Theorem 47, time-complexity classes can be characterized also by checking automata. First, this can be done by CF-based checking stack acceptors [Gre 2]. For instance, by Theorem 42, P-TIME = $2CSA_{FIN}(CF)$. Note that consequently NLOG $\subsetneq$ P-TIME if and only if $2GSM^{-1}(2DGSM(REG)) \subsetneq 2GSM^{-1}(2DGSM(CF))$. The general effect of an auxiliary CF-based checking stack can easily be computed from Theorems 45 and 47.

Secondly, time-complexity classes are related to checking tree acceptors. Let CTA denote the storage type of a checking tree acceptor (without additional pd) as explained in section 2.2 (Figure 11). Thus 2CTA is the class of languages accepted by 2-way checking tree acceptors (2-way input tape, tree-walking storage).

48. THEOREM.

(1) $2CTA_{FIN}$ = Multi-head 2PDA = P-TIME.

(2) 2CTA = NSPACE(n) aux PDA = U_cDTIME(2^{cn}).

(3) Multi-head 2CTA = P-SPACE aux PDA = U {DTIME($2^{p(n)}$) | p(n) *is a polynomial*}.

PROOF. The second equalities follow from Theorem 47. The proof of 2CTA = NSPACE(n) aux PDA is similar to that of 2CSA = NSPACE(n) in Theorem 40, the checking tree corresponding to the behaviour of the pushdown.

(i) 2CTA $\subseteq$ NSPACE(n) aux PDA. A computation of a 2cta can be coded by a tree of visiting sets (rather than a sequence of visiting sets as for the 2csa) obtain from the checking tree by adding to each node a set of tuples which code the visits of the 2cta to that node. The visiting set of a node should moreover fit to the visiting sets of its sons in an obvious sense. An NSPACE(n) aux pda can guess such a tree by storing visiting sets of nodes on its pushdown, using a pre-order traversal. It replaces the visiting set of a node on top of the pushdown by a possible sequence of visiting sets belonging to its sons. The check that the visiting sets fit can be done in linear space (similar to the case of the 2csa).

(ii) NSPACE(n) aux PDA $\subseteq$ 2CTA. By Theorem 42(1), NSPACE(n) aux PDA = 2CSA(CF). The inclusion 2CSA(CF) $\subseteq$ 2CTA can be proved using the tree-walking trick in the proof of Theorem 30 (Figure 10), cf. also Theorem 32. The 2cta first checks that its checking tree is a derivation tree of the given context-free grammar and then simulates the 2-way movement of the 2csa on the yield of this tree by walking on the tree itself.

This proves (2). The proof of (1) is analogous, using Theorems 41 and 42(2). The proof of (3) uses in the same way Theorems 43 and 45. Note that transition-tables for subtrees have to be computed in a bottom-up fashion; the checking tree with transition-tables is guessed in this case by a post-order traversal. □

Thus, in general, an auxiliary checking tree gives a double exponential jump from space to time. The same is true for an auxiliary ct-pd.

Another automaton related to time-complexity is the (general) stack automaton. A 2-way stack automaton (2SA) can be turned into a pushdown automaton by storing the transition table of each suffix of the stack on top of that suffix: this ensures that the transition table of the whole stack is always present on the top of the stack, and hence stack-reading

can be skipped (cf. Exercise 38). Since a new transition table can be computed in space n^2 (cf. Exercise 44), this shows that 2SA $\subseteq$ NSPACE(n^2) aux PDA. Actually 2SA = NSPACE(n^2) aux PDA which equals the union of all DTIME(2^{cn^2}) by Theorem 47. More generalization, [Iba], shows that Multi-head 2SA = "2^P-TIME", and gives corresponding results for auxiliary stack automata. Thus an auxiliary stack is equal in power to an auxiliary checking tree (cf. Theorem 48). It is shown in [Bee] that an auxiliary nested stack also has the same power.

From these complexity considerations we can now prove the following result concerning 2-way gsm (cf. Theorem 14).

49. THEOREM. *If L is a full semi*-AFL *included in the class of indexed languages, then* 2DGSM(L) $\not\subseteq$ 2GSM(L).

PROOF. By Exercise 12 it suffices to show that 2DGSM(L) $\not\subseteq$ $2GSM^k(L)$ for some k. We will show that there even exists a language in $2GSM^k$(REG) for some k but not in 2DGSM(Indexed). To be able to use complexity arguments we apply the operation $2GSM^{-1}$. Consider the class $2GSM^{-1}$(2DGSM(Indexed)). Since the indexed languages are recognized by the 1-way nested stack automaton, $2GSM^{-1}$(2DGSM(Indexed)) is the class of languages accepted by the multi-head nested stack automata (cf. Theorem 42 and the remark following it) which equals the class 2^P-TIME = $\bigcup \{DTIME(2^{p(n)}) \mid p(n)$ is a polynomial$\}$ by [Bee]. Since 2^P-TIME $\subseteq$ 2^P-SPACE $\subseteq \bigcup_c DSPACE(2^{2^{cn}})$, Exercise 46 implies that 2^P-TIME $\subseteq$ $2GSM^{-1}(2GSM^3(REG))$, and hence we can conclude (using Exercise 46 again) that $2GSM^{-1}(2DGSM(Indexed)) \subseteq 2GSM^{-1}(2GSM^3(REG)) \not\subseteq 2GSM^{-1}(2GSM^4(REG))$. Therefore $2GSM^4$(REG) $\not\subseteq$ 2DGSM(Indexed) which finishes the proof. □

Note that, by Theorem 48(3) and its generalization to CT-PD, Theorem 49 also holds for the tree transformation languages yT(RECOG), cf. Theorem 31, instead of the indexed languages, and in fact for any class $L_1^N(X)$ such that the class of languages accepted by multi-head X acceptors is inside some $\bigcup_c NSPACE(f_k(cn))$ as defined in Exercise 46.

The relationship between multi-head automata and complexity classes has some consequence with respect to the complexity of certain problems concerning 1-way automata, cf. [Jon 1; JonLaa; Gal 1; Hun]. We will show that for certain storage types X the non-emptiness problem of 1-way X acceptors (i.e. the problem whether a given 1-way X acceptor accepts a non-empty language) is complete in the class of languages accepted by multi-

head (2-way) X acceptors, i.e. the problem can be handled by a multi-head X acceptor and (more important) each multi-head X language can be reduced (in log space) to the problem. We say that a storage type is *finitely encoded* [Gin] if it has only finitely many tests and operations. The proof of the next theorem is similar to those in e.g. [Jon 1; JonLaa; Gal 1; Hun].

50. THEOREM. *Let X be a finitely encoded storage type. The non-emptiness problem for nondeterministic 1-way X acceptors is complete in the class of languages accepted by nondeterministic multi-head X acceptors.*

PROOF. Let $A_1^N(X)$ denote the class of nondeterministic 1-way X acceptors and also, ambiguously, the set of strings which code these acceptors over a fixed alphabet in the usual way (note that the alphabet can be fixed because X is finitely encoded). We first show that the language $\{M \in A_1^N(X) \mid L(M) \neq \emptyset\}$ can be accepted by a 2-head (2-way) X acceptor. Given an input string M, the 2-head X acceptor N checks that $M \in A_1^N(X)$ and then simulates M, guessing nondeterministically some input string for M which hopefully is accepted by M. N uses one head to point to the current instruction of M, simulates this instruction (possible because X is finitely encoded) and uses the second head to find a new instruction; the correct state behaviour of M is ensured by N checking equality of (binary coded) states in the two instructions, using both heads.

Next, let N be a k-head X acceptor. We will show that L(N) is log-tape reducible to $\{M \in A_1^N(X) \mid L(M) \neq \emptyset\}$. Let w be a given input to N of length n. Construct $N_w \in A_1^N(X)$ such that, independent of its input, N_w simulates N on w by keeping track of N's head positions on w in its states; thus N_w has states $\langle q,i_1,\ldots,i_k\rangle$ such that q is a state of N and $0 \leq i_j \leq n+1$. Clearly $w \in L(N)$ iff $L(N_w) \neq \emptyset$. The translation from w to N_w can be realized by a log n space Turing machine with a 1-way output tape. In fact, N_w contains for each $\langle i_1,\ldots,i_k\rangle$ a set of instructions easily obtainable from those of N; thus N_w has cn^k instructions for some constant c. To write down a sequence $\langle i_1,\ldots,i_k\rangle$ takes log n space and it suffices to keep track of these sequences. Note that N_w is of length $cn^k \log n$. □

We note that this theorem holds (instead of non-emptiness) for every nontrivial property of 1-way X languages which can be decided by a multi-head X acceptor and which is false for $\emptyset$ (cf. [Hun]): after successful simulation of N on w, N_w should simulate (on its actual input) some machine

M whose language has the property. It is also easy to see that the theorem is true for the "general membership problem", i.e. the language $\{(M,v) \mid M \in A_1^N(X) \text{ and } v \in L(M)\}$. We finally note that the theorem also holds if the 1-way X acceptors are restricted to be deterministic (N_w uses its input to determine the nondeterminism of N).

Since most of the discussed storage types (excluding e.g. CSA_{FIN}) are finitely encoded or (what suffices) equivalent to some finitely encoded storage type, we can obtain a lot of completeness results using Theorem 50. Let us consider some of them.

First, since NLOG is the class of languages accepted by multi-head finite automata, $\{M \in 1FA \mid L(M) \neq \emptyset\}$ is complete in NLOG, where 1FA is the class of 1-way finite state automata [Jon 1].

By Theorem 43, $\{M \in 1CSA \mid L(M) \neq \emptyset\}$ is complete in P-SPACE. Since a 1-way csa is the same as a 2gsm and the range of a 2gsm is empty iff its domain is, this implies that $\{M \in 2GSM \mid dom(M) \neq \emptyset\}$ is complete in P-SPACE, cf. [Gal 1]. It is possible to determine the complexity of $\{M \in 1CSA \mid L(M) \neq \emptyset\}$ more precisely as follows. It is quite easy to see that $\{M \in 1CSA \mid L(M) \neq \emptyset\}$ can even be accepted by a 2csa (rather than a 2-head 2csa as in the proof of Theorem 50): for input M, check that the checking stack contains a string $\sigma_1 M \sigma_2 M \ldots \sigma_k M$, simulate M with checking stack $\sigma_1\sigma_2\ldots\sigma_k$ and use the stackhead as a second input head working on one of the M's. Hence, by Theorem 40, $\{M \in 1CSA \mid L(M) \neq \emptyset\}$ is in NSPACE(n). Moreover, since the log tape reduction used in the proof of Theorem 50 produces an n log n output, it is not in $NSPACE(n^{1-\varepsilon})$ for any $\varepsilon > 0$. A similar reasoning (cf. Exercise 44) shows that $\{M \in 1CS\text{-}PD \mid L(M) \neq \emptyset\}$, which is complete in P-SPACE, is in $NSPACE(n^2)$ but not in any $NSPACE(n^{2-\varepsilon})$; in [JonSky 2] this is shown for $\{M \in PAM \mid L(M) \neq \emptyset\}$ and NSPACE(n): note that the transition from a cs-pd machine to a pam requires nonterminals which are pairs of states and thus a factor n^2, cf. Theorem 29. A similar result holds for nonerasing stack automata [Hun].

51. THEOREM. *The language* $\{M \in 2GSM^2 \mid dom(M) \neq \emptyset\}$ *is complete in exponential space. In general, for every* $k \geq 1$, *the language* $\{M \in 2GSM^k \mid dom(M) \neq \emptyset\}$ *is complete in* k*-double exponential space, i.e.* $U\{NSPACE(f_k(p(n))) \mid p(n)$ *is a polynomial*$\}$ *where* f_k *is defined in Exercise* 46.

PROOF. By Theorem 45, $\{M \in 1CSA(L_1^N(X)) \mid L(M) \neq \emptyset\}$ is complete in the class of languages accepted by P-SPACE auxiliary X acceptors. From this the result follows by induction. □

It can be shown that this theorem also holds for $2GSM^{k-1} \circ 2DGSM$ rather than $2GSM^k$.

We now turn to time. By Theorem 47(1), $\{M \in 1(D)PDA \mid L(M) \neq \emptyset\}$ is complete in P-TIME [JonLaa; Gal 1; Hun].

52. THEOREM. *The language* $\{M \in CT \mid dom(M) \neq \emptyset\}$ *is complete in exponential time.*

PROOF. By Theorem 48(3). □

This theorem shows that the non-emptiness problem for the finite-state tree-walking automaton (i.e. the checking tree transducer without output, cf. section 2.2) is complete in exponential time. This problem is closely related to the circularity problem of attribute grammars [JazOgdR]: it is quite easy to simulate a tree-walking automaton by the dependency graph of an attribute grammar which has an attribute corresponding to each of its states.

The language $\{M \in 1SA \mid L(M) \neq \emptyset\}$ is also complete in 2^P-TIME, cf. [Hun].

This ends our discussion of complete problems. We finally consider NP-TIME.

It is an open problem whether NP-TIME can be characterized by an unrestricted class of automata. NP-TIME can be obtained by restricting time or space of the multi-head 2csa, see [vLe]. We say that a multi-head 2csa is *polynomial* either if it works in polynomial time or if it only needs a checking stack of polynomial length.

53. THEOREM. *Polynomial Multi-head* 2CSA = NP-TIME.

PROOF. As in Theorem 43. On the one hand, an NP-TIME Turing machine has computations $w_1\#w_2\#\ldots\#w_m$ where both $|w_i|$ and m are polynomial in n, and so the multi-head 2csa can work in polynomial time (and hence in polynomial space). On the other hand, transition tables of a multi-head 2csa M can be updated in polynomial time, and if M's checking stack has polynomial length, only polynomially many transition tables have to be computed. □

The same result (with the same proof) holds for polynomial multi-head 2cs-pd and for the general "auxiliary case" [vLe].

It would be nice if the sequence P-SPACE, NP-TIME, P-TIME would correspond to a sequence of multi-head automata obtained by increasing restrictions on the same storage type. Consider Theorems 43 and 47 and note that the checking stack and the pushdown are incomparable storage types (cf. Theorem 16). What about combining them into a cs-pd automaton! A cs-pd acceptor is said to be deterministic if it works deterministically and moreover has to start by deterministically building up (symbol by symbol) its own checking stack (i.e. it is viewed as a deterministic s-pd automaton). It is shown in [vLe] that in the deterministic (2-way multi-head) case the checking stack is of no use to the cs-pd automaton. Thus ([vLe]) the above sequence corresponds to multi-head 2cs-pd acceptors (Exercise 44) restricted to be polynomial (Theorem 48) and deterministic (Theorem 47) respectively (each deterministic multi-head cs-pd works in polynomial space).

3.2. Two-way deterministic pushdown automata

We have seen that restricted 2-way pushdown automata related to syntax-directed translation (the cs-pd transducer in section 2.1, Figure 9) and generalized 2-way pushdown automata related to time complexity (the multi-head 2pda in section 3.1). In this section we consider the normal 2-way pushdown automaton which has been studied extensively for both of the above reasons, see e.g. [SteHarL; GraHarI; Gal 2; Hun; Mon]. Apart from a few general properties we will concentrate on the linear time simulation of a 2-way deterministic pushdown automaton (on a random access machine with uniform cost) proved in [Coo 1]. This result can be used to obtain efficient pattern-matching algorithms [KnuMorP; cf. AhoHopU 3, Chapter 9].

A 2dpda consists of a 2-way read-only input tape which is an array input[0:n+1], an input head which is a variable x of type [0:n+1], an output tape and pushdown store of the usual kind, and a finite control which can use the operations x := x + 1, x := x - 1, push(A), pop, and the tests input[x] = σ and top = A (for each input symbol σ and pushdown symbol A). Formally, a *2-way deterministic pushdown automaton* (2dpda) is a tuple $M = (Q,\Sigma,\Gamma,\delta,q_0,F,Z_0)$ of states, input symbols, pushdown symbols, transition function, initial state, final states and bottom pushdown symbol respectively, where δ is a function

$$Q \times (\Sigma \cup \{¢,\$\}) \times \Gamma \to Q \times \{-1,0,+1\} \times (\{pop\} \cup \{push(A) \mid A \in \Gamma\}).$$

The interpretation of $\delta(q,\sigma,A) = (q',d,\beta)$ is that M in state q reading input symbol σ with top = A may go into state q', move the input head d squares to the right and either pop A (if β=pop) or push B on top of A (if β = push(B)). A *configuration* of M for a given input ¢w$ is of the form (q,i,γ) indicating the state, the position of the input head ($0 \le i \le |w|+1$) and the content of the pushdown store (with the top at the left). M accepts the language $\{w \in \Sigma^* \mid (q_0,0,Z_0) \vdash^* (q,i,\gamma)$ for some $q \in F$, $0 \le i \le |w|+1$, $\gamma \in \Gamma^*$, and all configurations are for ¢w$$\}$. We will also need the concept of a *surface configuration* of M for given input ¢w$: it is a triple (q,i,A) with $q \in Q$, $0 \le i \le |w|+1$ and $A \in \Gamma$, which only indicates the top of the pushdown rather than the complete pushdown. The basic property of surface configurations (as in the 1-way case) is that if $(q,i,A) \vdash^* (p,j,\lambda)$ then, for all $\gamma \in \Gamma^*$, $(q,i,A\gamma) \vdash^* (p,j,\gamma)$, expressing that we never look inside the pushdown. The notion of a *nondeterministic* 2pda is obtained in the obvious way.

54. EXAMPLES.

(1) The following program for the 2dpda recognizes all palindromes, i.e. strings w equal to their reverse $\tilde{w}$, by putting w on its pushdown and comparing it with the input. (We use a few more involved operations and tests which can easily be similated).

```
begin x := x + 1;
     while input[x] ≠ $ do begin push(input[x]); x := x + 1 end;
     while input[x] ≠ ¢ do x := x - 1;
     x := x + 1;
     while input[x] = top do begin x := x + 1; pop end;
     if input[x] = $ and top = Z0 then accept else reject
end.
```

(2) The following program shows the applicability of the 2dpda to *string matching*: it recognizes all strings u#v such that u is a substring of v. The program uses the most obvious method: it tests whether u is a prefix of a suffix of v; successive suffixes of v are kept on the pushdown. We assume that the pushdown contains already vZ_0 and that x = 1.

```
begin
     while top ≠ Z0 do
          begin {does u match a prefix of the pushdown?}
               while input[x] = top do begin x := x + 1; pop end;
```

```
            if input[x] = # then accept
            else if top = Z0 then reject
            else {restore the pushdown after a mismatch
                  and try the next suffix}
              begin x := x - 1; while input[x] ≠ ¢ do begin push(input[x]);
                                                            x := x - 1
                                                    end;
                    pop      ;  x := x + 1
              end
        end; reject
end. □
```

55. EXERCISE. Show that the following languages are in 2DPDA:
$\{a^{2^n} | n \geq 0\}$ (if $x = 2^m$, use the pushdown to make $x = 2^{m+1}$),
$\{a^n b^{n^2} | n \geq 1\}$, $\{a^n b^{n^3} | n \geq 1\}$,
$\{u_1 \# u_2 \# \ldots \# u_n \mid n \geq 2,\ u_i \neq u_j \text{ for } i \neq j\}$,
$\{u_1 \# u_2 \# \ldots \# u_n \#\# v_1 \# v_2 \# \ldots \# v_m \mid n,m \geq 2,\ u_i \neq u_j \text{ for } i \neq j,$
and for each i there exists j such that $v_j = u_i\}$ (declarations!),
$\{uv \mid u = \tilde{u}, |u| > 1\}$ (similar to Example 54(2), see [AhoHopU 3]),
$\{u \# v_1 \# v_2 \# \ldots \# v_n \mid n \geq 1,\ uv_i \in D_2 \text{ for some } 1 \leq i \leq n\}$,
$\{u_1 \# \ldots \# u_n \#\# v_1 \# \ldots \# v_m \mid n,m \geq 1,\ u_i v_j \in D_2 \text{ for some } i,j\}$,
where D_2 is the Dyck language generated by the context-free rules
$S \to \underset{a}{[}S\underset{a}{]}S$, $S \to \underset{b}{[}S\underset{b}{]}S$, $S \to \lambda$ (interpret $\underset{a}{[}$ as push(a) and $\underset{a}{]}$ as pop(a)). □

We first note that 2DPDA and 2PDA are closed under inverse 2dgsm mappings by Theorem 19, cf. [GraHarI] where this was not yet realized. Next we note that 2DPDA languages are context-sensitive.

56. THEOREM. [SteHarL; GraHarI] 2DPDA $\subseteq$ DSPACE(n).

PROOF. The result follows because the length of the pushdown during a successful computation of a 2dpda M is bounded linearly in the length of the input. To prove this, let $A_m A_{m-1} \ldots A_1$ be the content of the pushdown at some moment of time and let, for $1 \leq k \leq m$, (q_k, i_k, A_k) be the surface configuration of M at the last moment of time that the pushdown had height k. If $(q_s, i_s, A_s) = (q_t, i_t, A_t) = (q,i,A)$ for some $1 \leq s < t \leq m$, then $(q,i.A) \overset{*}{\vdash} (q,i,A\gamma A)$, where $A\gamma A = A_t \ldots A_s$, and the 2dpda is in a loop. Hence the length of the pushdown is at most the number of surface configurations which is proportional to n. □

It is an open problem whether the above inclusion is strict, but in [Gal 2] it is shown that 2DPDA = DSPACE(n) implies P-TIME = P-SPACE. Similarly it is open whether 2DPDA is properly included in 2PDA, and whether CF $\subseteq$ 2DPDA. It is shown in [GalSei], using combinatorial properties of palindromes, that the language $\{u\tilde{u}v\tilde{v} \mid u,v \in \{a,b\}^+\}$ is in 2DPDA (which was conjectured in [AhoHopU 2] not to be in it). The relation of 2PDA to NSPACE(n) is unknown. Similarly the relation of 2DPDA to DLOG = DSPACE(log n) is not clear. Using Theorem 47 [Coo 2] it is shown in [Gal 2] by a translational argument (translating k-head 2dpda down to 1-head 2dpda) that if 2DPDA $\subseteq$ DLOG then P-TIME = DLOG (the same result was shown in [Hun;Mon]). Since 2DPDA $\subseteq$ DTIME(n^4) [AhoHopU 2], this proves that 2DPDA $\neq$ DLOG. Thus several open problems in complexity theory "hide" in 2DPDA.

We now turn to the linear time simulation of the 2dpda [Coo 1]. The result is important because it allows one to program a 2dpda without caring about time bounds, and be sure that an efficient equivalent algorithm exists. In fact, a 2dpda can be made to count to 2^n in between its moves: store the position of the input head on the pushdown by moving left to ¢ pushing dummy symbols; simulate a dcs-pd transducer which translates a^n into a^{2^n}; find back the original input head position. Note also that the string-matching algorithm of Example 54(2) works in time n^2.

The proof in [Coo 1] uses essentially a transition table approach. In this approach "small" computations are used to build up "larger" computations (in the cases we have seen in sections 1.1 and 2.3 "small" and "large" refer to the length of the suffix on which the computations work, or just to the length of the computations). Such a method is also called "dynamic programming" and can be used e.g. to recognize context-free languages in time n^3, cf. [AhoHopU 3, section 2.8; AhoUll 3, section 4.2]. In general dynamic programming can be used to compute the values of recursive procedures (or pushdown algorithms) in a "bottom-up" fashion, i.e. by calculating the values for small parameters first (storing them in a table) and proceed to larger parameters (cf. the bottom-up computation by a register transducer of the recursive calls of a pam, section 2.3). The (time) advantage of the method is that identical calls are evalulated only once and stored in a table for further use. The disadvantage of the method is that the structure of the original recursive (or pushdown) algorithm is almost completely lost, and moreover that calls are computed which may never occur in the original algorithm giving rise to a large constant in the time bound.

A solution to these disadvantages is to take a mixed approach [Jon 2]: the recursive (or pushdown) algorithm is executed in the usual "top-down" fashion, but as soon as the value of a call F(x) has been computed, it is stored in a (transition) table; a subsequent call F(x) is just looked up in the table. Observe that a call of a recursive procedure in a recursive algorithm corresponds roughly to a ("surface") computation $(q,i,A) \vdash^* (p,j,\lambda)$ of a pushdown algorithm, where (q,i,A) is a surface configuration. In [Jon] this mixed strategy is applied to the simulation of the 2dpda [Coo 1], whereas in [Bir] it is applied to some specific examples of 2dpda algorithms.

We shall give an iterative variant of the algorithm of [Jon 2]. However, to illustrate the idea we start with a

57. "Mixed" proof of Theorem 35.

PROOF. We only consider the deterministic case. Let $M = (Q,\Sigma,\Gamma,\delta,q_0,F,Z_0)$ be a dcs-pd transducer without output (i.e. a restricted 2dpda). We will show that, using the mixed approach, M can be simulated by a finite visit Turing machine N with one read/write head of length n and one input head x (note that this implies regularity of M's domain by Exercise 5; note also that the finite visit property implies that N works in linear time). N's tape is divided into four tracks as shown in Fig. 14. Track 2 contains the input $\¢\sigma_1\sigma_2\ldots\sigma_n\$$ of M and track 3 contains the pushdown of M; N's input head simulates M's input head, i.e. it points to the top of the pushdown.

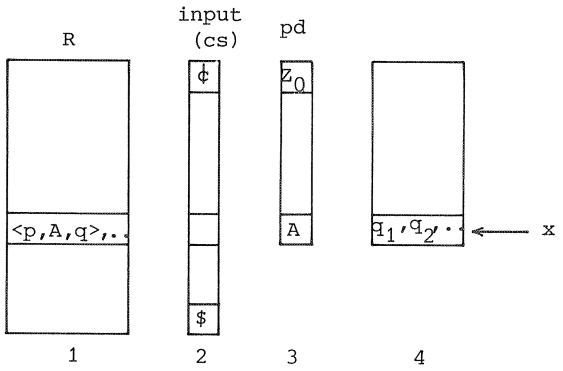


Figure 14. "Mixed" simulation of a cs-pd transducer (without output).

N simulates M step by step except that it skips subcomputations which it

has stored in a table. Track 1 contains the (partially filled) transition tables of the suffixes of the input string, i.e. (for each i) R[i] is the (partial) transition table of suffix $\sigma_i \ldots \sigma_n\$$; R[i] is a list of triples, each triple <p,A,q> representing the usual (surface!) computation of M on $\sigma_i \ldots \sigma_n\$$; moreover this computation has actually been executed before by M and the triple was stored in R[i] after its completion (except that, to simplify N, R[i] contains initially <p,A,q> if δ(p, input[i],A) = (q,-1,pop)). Whenever N simulates M in state p at position x with top = A, it consults R[x] to see whether it contains some triple <p,A,q>; if so, N moves left popping A and continues to simulate M in state q; if not, it simulates a (push) move of M. In order to fill R, N has to do some bookkeeping on track 4. L is an auxiliary pushdown which makes the same movements as M's pushdown; for $0 \leq i \leq x$, L[i] contains a list of states of M: they are the previous states of M at position i during the current subcomputation of M on $\sigma_i \ldots \sigma_n\$$ (they correspond to the states $q_1, \ldots, q_n$ in (*) of the proof of Theorem 2 and (**) in section 2.3; note that, in those statements, actually $(q_j, q_n') \in R_{\sigma u}$ for all q_j). Suppose as before that N simulates M in state p at position x with top = A; if there is no triple <p,A,q> in R[x], then N adds p to the list L[x] and simulates a (push) move of M (pushing the empty list on L); if there is a triple <p,A,q> in R[x], then N adds a triple $<q_j,A,q>$ to R[x] for each q_j in the list L[x], and moves left popping both pushdowns and continues with M in state q (in this way N has recorded all of M's subcomputations on the suffix it just left). This ends the description of N.

It remains to show that N is finite visit. This is mainly due to the following property of the algorithm which formally expresses that N never computes the same surface computation $(p,i,A) \overset{*}{\vdash} (q,i-1,\lambda)$ twice:

(1) for given i, each state q of M is added at most once to L[i]. (Proof: if q occurs twice in L[i] at some moment of time, then M got into a loop and simulation should stop; if there is only one occurrence of q in L[i] and L[i] is then popped, then a triple <q,-,-> is added to R[i] and q will never be added to L[i] again).

It remains to show that the number of visits is proportional to the number of state additions. This follows from the following observations, where $k = \#(Q)$.

(2) Each L[i] is a nonempty list except possibly L[x]. (Proof: before pushing, the current state of M has to be added to L[x]).

(3) The number of visits to a square x such that L[x] is nonempty is ≤ 2k. (Proof: with each such visit either a state is added to L[x] or a state is removed from L by popping L[x]; by (1), this happens at most 2k times).

(4) The number of visits to x with L[x] empty is ≤ k. (Proof: by the proof of (2), and (1)).

Since the initial filling of R (with pop-moves) takes two visits, (3) and (4) imply that the total number of visits is bounded by 3k+2. □

We now turn to the (unrestricted) 2dpda and show that the linear time property of the above construction is preserved. The simulating algorithm will be written in Pidgin ALGOL, cf. [AhoHopU 3, section 1.8].

58. THEOREM. [Coo 1]. *If* L *is accepted by a* 2dpda, *then* L *is accepted by a Pidgin ALGOL program in linear time.*

PROOF. [Jon 2]. Let $M = (Q,\Sigma,\Gamma,\delta,q_0,F,Z_0)$ be an arbitrary 2dpda. A *transition table* of M for input ¢w$ is a function $R: Q \times [0{:}n+1] \times \Gamma \to Q \times [0{:}n+1]$ such that $R(q,i,A) = (p,j)$ if and only if $(q,i,A) \overset{*}{\vdash} (p,j,\lambda)$ is a (surface) computation of M on ¢w$. We will construct a Pidgin ALGOL program N, simulating M, which uses an array R to store a (partially filled) transition table of M; each location R[q,i,A] of R will either contain a pair (p,j) or some indication that it is undefined. Moreover (see Figure 15 which should be compared to Fig. 14) N uses the input, input head x and pushdown of M, plus an auxiliary pushdown L moving simultaneously with M's pushdown. Each "square" of L contains a list of surface configurations <q,i,A> of M (where the A is actually superfluous because it will always be equal to the pushdown symbol on the same level); if the k-th square of M's pushdown contains symbol B and the corresponding list on L contains a triple <q,i,B>, it means that M was in surface configuration <q,i,B> at a previous moment in its computation with the k-th square on top, and in the meantime this particular B has not been popped.

In program N, a pop-instruction will pop both pushdowns, whereas a push(B)-instruction pushes B on M's pushdown and the empty list on L; "top" refers to M's pushdown, whereas "list" denotes the top of L; "state" contains the current state of M; and "conf" abbreviates (state, x, top), i.e. the current surface configuration of M. The program starts with Z_0 on M's pushdown, the empty list on L, and R completely undefined. We assume that the program halts automatically if M halts. Finally we assume that $\delta(q_0,¢,Z_0) \neq (-,-,\text{pop})$. Program N now follows.

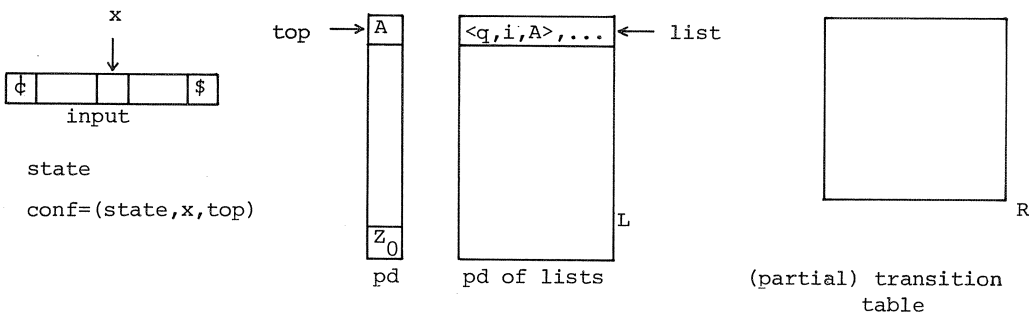


Figure 15. Data structure of the Pidgin ALGOL program.

```
1    begin for all (q,i,A) ∈ Q × [0:n+1] × Γ do
           if δ(q, input[i],A) = (p,d,pop) for some d ∈ {-1,0,+1}
              then R[q,i,A] := (p,i+d);
2      state := q0; x := 0;
3      add (q0,0,Z0) to list;
4      repeat
5        begin let δ(state,input[x],top) = (p,d,push(A)) in
                    begin state := p; x := x + d; push(A) end;
6                 while R[conf] is defined do
7                  begin for all c on list do R[c] := R[conf];
8                        (state,x) := R[conf];
9                         pop
                   end;
10                add conf to list
         end
     end of N.
```

COMMENTS:

— Lines 1,2,3 contain the initialization; in particular in line 1 R is filled with one-step surface computations (due to pop-moves).

— Line 5 simulates one push-move of M; pop-moves are dealt with when applying R (in line 9) due to the initialization of R with pop-moves.

— In the while-statement of line 6 use is made as much as possible of the subcomputations in the transition table R (each such subcomputation results in a pop). In line 7 R is updated using the information in list.

— Line 10: if R cannot be used, the current surface configuration is added

to list and a push move will be simulated (line 5).

It remains to show that N works in linear time. This follows from the following observations:

(1) A surface configuration is only added to list if R[c] is undefined (lines 6 and 10).

(2) Each surface configuration c is added at most once to list by line 10 (if c occurs twice in L at the same moment of time, then M is in a loop, cf. the proof of Theorem 56; but if c occurs once in L and is removed from L (line 9), then line 7 has defined R[c] and so, by (1), c cannot be added again to L).

(3) The number of surface configurations is O(n).

(4) Lines 3 and 10 now show that, by (2) and (3), the repeat statement (lines 5 to 10) is executed O(n) times.

(5) Since a push-move is always preceded by line 10 (or 3), list is nonempty after a pop (line 9).

(6) Since execution of the while-body (lines 7-9) is always preceded by a pop (except on each first entrance), (5) implies that at least one c is removed in line 7. Thus the total time used to execute the while-statement (lines 6-9) is O(n) (the number of first entrances, see (4)) plus O(n) (the time to execute line 7, see (3)), hence O(n).

Taking (4) and (6) and the time to execute line 1 together shows that the program takes O(n) time. □

If it costs log n time to store number n, then the algorithm takes time n log n. The algorithm can easily be generalized to show that a k-head 2dpda can be simulated in time n^k (or n^k log n), cf. [Coo 1] and Theorem 47.

59. EXAMPLE. Consider the string-matching 2dpda of Example 54(2). The only interesting state q of that program is the one on the 4-th line in which u is matched against the pushdown. If at the end a mismatch occurs, then the use of R will make it unnecessary to raise the pushdown. In fact this means that if $u = u_1u_2\ldots u_m$ ($u_i \in \Sigma$) and (when the mismatch occurs) x = i and top = A (thus $A \neq u_i$), then R[q,i,A] = (q,j) where j is the largest integer such that $u_1\ldots u_j$ is a suffix of $u_1\ldots u_{i-1}A$. Note that R depends only on u. This is almost exactly the function computed in the linear time algorithm for string matching described in [KnuMorP], see also [AhoHopU 3; Bir]. □

Finally we discuss the relevance of these ideas to limited backtrack (top-down) parsing of context-free languages [AhoUll 3, section 6.1].

Although a 2dpda has the ability to reconsider its input, it is not immediately clear how this can be used for backtracking. Top-down recognition with limited backtracking works as follows. Let A be a nonterminal (of a context-free grammar) to be recognized, beginning at position i of the input string. Let A have productions $A \to \alpha_1|\alpha_2|\ldots|\alpha_m$. If α_1 is recognized, then A is recognized; if α_1 is not recognized, then no other recognition of α_1 is tried, but the input-head is reset to position i and recognition of α_2 is started. Etcetera, for details see [AhoUll 3]. This recognition method can be implemented on a "*back-track* 2dpda" which is an extension of the 2dpda (and does not satisfy the basic assumption of general automata theory). Apart from the usual 2dpda-instructions a back-track 2dpda also has "jump" instructions $\delta(q,\sigma,A) = (q',jump,pop)$ meaning that A should be popped and the input head should be put at the position it had just after this particular A was pushed on the pushdown.

60. THEOREM. *Each backtrack* 2dpda *can be simulated in linear time by a Pidgin ALGOL program.*

PROOF. The program is a slight extension of the one in Theorem 58. The idea of the extension is that (i) the first item of list always contains the "back-track position" to which the input head should jump (see Fig. 15), and (ii) if, just after a push, R[conf] indicates a back-track computation, then the input head can stay where it is. Apart from the usual R[q,i,A] = (p,j) which expresses a "successful" surface computation $(q,i,A) \vdash^* (p,j,\lambda)$ of the backtrack 2dpda M, we now also have R[q,i,A] = (p,jump) expressing a "failing" surface computation of M resulting in a backtrack: $(q,i,A) \vdash^* (q',i',A) \vdash^* (q',i',A) \vdash (p,?,\lambda)$ where the last move is caused by a jump instruction.

We give the new program without much comment. It consists essentially of the old program together with (numbered) statements handling the jump case. The while statement has been unravelled once. The program is shown in Fig. 16.

COMMENTS:

— At line 1 the one-step backtrack computations are put in R.

— The if-statement starting at line 2 is executed just after a push. Therefore, if R indicates a backtrack computation, then (by the definition of the jump instruction) x is at the backtrack position and hence should

```
    begin for all (q,i,A) ∈ Q × [0:n+1] × Γ do
          if δ(q,input[i],A) = (p,d,pop) then R[q,i,A] := (p,i+d)
1         else if δ(q,input[i],A) = (p,jump,pop) then R[q,i,A] := (p,jump);
      state := q0; x := 0;
      add (q0,0,Z0) to list;
      repeat
        begin let δ(state,input[x],top) = (p,d,push(A)) in
                  begin state := q; x := x+d; push(A) end;
2         if R[conf] is defined then
            begin if R[conf] = (q,j) then (state,x) := (q,j)
3            else if R[conf] = (q,jump) then (state,x) := (q,x);
             pop
            end;
          while R[conf] is defined do
           begin if R[conf] = (q,j) then
                   begin for all c in list do R[c] := R[conf];
                         (state,x) := (q,j)
                   end
4            else if R[conf] = (q,jump) then
                   begin state := q; x := input-position of first item in
                                                 list;
                         for all c in list do R[c] := R[conf]
                   end;
                 pop
           end;
        add conf to list
      end of repeat
    end
```

Figure 16. Simulation of backtrack 2dpda.

not change (line 3), cf. remark (ii) above.

— The if-statement starting at line 4 is executed just after a pop (and hence list is nonempty); x is set to the input-head position of the first triple of list, cf. remark (i) above.

The time-analysis of the new program is exactly the same as that in the proof of Theorem 58. □

As a corollary we obtain that limited backtrack top-down recognition of context-free languages can be done in linear time (see [AhoUll 3, section 6.1.4], where a dynamic programming algorithm is given). Actually the backtrack 2dpda is (almost) the same as the "parsing machine" in [AhoUll 3, section 6.1.5].

REFERENCES

[Aho] AHO, A.V., *Nested stack automata*, JACM 15 (1968), 647-671.

[AhoHopU] AHO, A.V., J.E. HOPCROFT & J.D. ULLMAN,
(1) *A general theory of translation*, Math. Syst. Th.3 (1969), 193-221.
(2) *Time and tape complexity of pushdown automaton languages*, Inf. and Control 13 (1968), 186-206.
(3) *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass., 1974.

[AhoUll] AHO, A.V. & J.D. ULLMAN,
(1) *A characterization of two-way deterministic classes of languages*, JCSS 4 (1970), 523-538.
(2) *Translations on a context-free grammar*, Inf. and Control 19 (1971), 439-475.
(3) *The theory of parsing, translation and compiling*, Prentice-Hall, Englewood Cliffs, N.J., 1972 (two volumes).

[Arb] ARBIB, M.A., *Theories of abstract automata*, Prentice-Hall, Englewood Cliffs, N.J., 1969.

[Asv] ASVELD, P.R.J., *Controlled iteration grammars and full hyper*-AFL's, Inf. and Control 34 (1977), 248-269.

[AsvEng] ASVELD, P.R.J. & J. ENGELFRIET, *Extended linear macro grammars, iteration grammars, and register programs*, Memorandum 209, Twente University of Technology, 1978 (to appear in Acta Informatica).

[Bee] BEERI, C., *Two-way nested stack automata are equivalent to two-way stack automata*, JCSS 10 (1975), 317-339.

[Bir] BIRD, R.S., *Improving programs by the introduction of recursion*, CACM 20 (1977), 856-863.

[ChyJak] CHYTIL, M.P. & V. JÁKL, *Serial composition of 2-way finite-state transducers and simple programs on strings*, in: *Automata, Languages and Programming*, Fourth Colloquium (A. Salomaa, M. Steinby eds), Lecture Notes in Computer Science 52, p. 135-147, 1977. (Springer-Verlag, Berlin).

[Coo] COOK, S.A.,

(1) *Linear time simulation of deterministic two-way pushdown automata*, Information Processing 71 (Proceedings IFIP Congress, Ljubljana), North-Holland, p. 75-80, 1972.

(2) *Characterizations of pushdown machines in terms of time-bounded computers*, JACM 18 (1971), 4-18.

[Dow] DOWNEY, P., *Formal languages and recursion schemes*, Harvard University, Report TR 16-74, 1974.

[EhrRoz] EHRENFEUCHT, A. & G. ROZENBERG, *On some context-free languages that are not deterministic ETOL languages*, RAIRO (Informatique Theoretique) 11 (1977), 273-291.

[EhrYau] EHRICH, R.W. & S.S. YAU, *Two-way sequential transductions and stack automata*, Inf. and Control 18 (1971), 404-446.

[Eil] EILENBERG, S., *Automata, languages and machines*, Volume A; Academic Press, New York, 1974.

[Eng] ENGELFRIET, J.,

(1) *Three hierarchies of transducers*, Memorandum 217, Twente University of Technology, Enschede, 1978 (see also [EngRozS]).

(2) *Bottom-up and top-down tree transformations - a comparison*, Math. Syst. Th.9 (1975), 198-231.

[EngRozS] ENGELFRIET, J., G. ROZENBERG & G. SLUTZKI, *Tree transducers, L systems and two-way machines*, Memorandum 187, Twente University of Technology, Enschede, 1977, also in: Proc. 10-th Ann. ACM Symp. on Theory of Computing, San Diego, 1978, p. 66-74 (together with [Eng 1]).

[EngSch^{V}L] ENGELFRIET, J., E. MEINECHE SCHMIDT & J. VAN LEEUWEN, *Stack machines and classes of nonnested macro languages*, Report RUU-CS-77-2, University of Utrecht, 1977 (to appear in JACM).

[EngSky] ENGELFRIET, J. & S. SKYUM, *Copying theorems,* Inf. Proc. Letters 4 (1976), 157-161.

[Fis] FISCHER, M.J.,

(1) *Two characterizations of the context-sensitive languages,* IEEE Conf. Rec. of 10-th Ann. Symp. on Switching and Automata Theory, p. 149-156, Waterloo, Ontario, Canada, 1969.

(2) *Grammars with macro-like productions,* Ph.D. Thesis, Harvard University, 1968.

[Gal] GALIL, Z.,

(1) *Hierarchies of complete problems,* Acta Informatica 6 (1977), 77-88.

(2) *Some open problems in the theory of computation as questions about two-way deterministic pushdown automaton languages,* Math. Syst. Th. 10 (1977), 211-228.

[GalSei] GALIL, Z. & J. SEIFERAS, *A linear-time on-line recognition algorithm for "palstar",* JACM 25 (1978), 102-111.

[Gin] GINSBURG, S., *Algebraic and automata-theoretic properties of formal languages,* North-Holland/Elsevier, Amsterdam/New York, 1975.

[GinGreH] GINSBURG, S., S.A. GREIBACH & M.A. HARRISON, *Stack automata and compiling,* JACM 14 (1967), 172-201.

[Gol] GOLDSTINE, J., *Automata with data storage,* Proc. of A Conference on Theoretical Computer Science, Waterloo, Ont., Canada 1977, p. 239-246.

[GraHarI] GRAY, J.N., M.A. HARRISON & O.H. IBARRA, *Two-way pushdown automata,* Inf. and Control 11 (1967), 30-70.

[Gre] GREIBACH, S.A.,

(1) *One-way finite visit automata,* Theor. Comp. Sci. 6 (1978), 175-221.

(2) *Visits, crosses, and reversals for nondeterministic off-line machines,* Inf. and Control 36 (1978), 174-216.

(3) *Checking automata and one-way stack languages,* JCSS 3 (1969), 196-217.

(4) *Hierarchy theorems for two-way finite state transducers,* Report, University of California, Los Angeles, 1977.

[Har] HARTMANIS, J., *On non-determinacy in simple computing devices,* Acta Informatica 1 (1972), 336-344.

[Hen] HENNIE, F.C., *One-tape, off-line Turing machine computations,* Inf. and Control 8 (1965), 553-578.

[HerRoz] HERMAN, G.T. & G. ROZENBERG, *Developmental systems and languages,* North-Holland, Amsterdam, 1975.

[HopUll] HOPCROFT, J.E. & J.D. ULLMAN,

(1) *An approach to a unified theory of automata,* The Bell System Technical Journal 46 (1967), 1793-1829.

(2) *Nonerasing stack automata,* JCSS 1 (1967), 166-186.

(3) *Formal languages and their relation to automata,* Addison-Wesley, Reading, Mass., 1969.

[Hun] HUNT, H.B., III, *On the complexity of finite, pushdown, and stack automata,* Math. Syst. Th.10 (1976), 33-52.

[Iba] IBARRA, O.H., *Characterizations of some tape and time complexity classes of Turing machines in terms of multi-head and auxiliary stack automata,* JCSS 5 (1971), 88-117.

[JazOgdR] JAZAYERI, M., W.F. OGDEN & W.C. ROUNDS, *The intrinsically exponential complexity of the circularity problem for attribute grammars,* CACM 18 (1975), 697-706.

[Jon] JONES, N.D.,

(1) *Space-bounded reducibility among combinatorial problems,* JCSS 11 (1975), 68-85.

(2) *A note on linear time simulation of deterministic two-way pushdown automata,* Inf. Proc. Letters 6 (1977), 110-112.

[JonLaa] JONES, N.D. & W.T. LAASER, *Complete problems for deterministic polynomial time,* TCS 3 (1977), 105-117.

[JonSky] JONES, N.D. & S. SKYUM,

(1) *Recognition of deterministic ETOL languages in logarithmic space,* Inf. and Control 35 (1977), 177-181.

(2) *Complexity of some problems concerning* L *systems,* in: "Automata, Languages and Programming", Turku, 1977, p. 301-308, Lecture Notes in Computer Science 52, Springer-Verlag, Berlin.

[Kie] KIEL, D.I., *Two-way a-transducers and* AFL, JCSS 10 (1975), 88-109.

[KnuMorP] KNUTH, D.E., J.H. MORRIS JR. & V.R. PRATT, *Fast pattern matching in strings,* SIAM J. Computing 6 (1977), 323-350.

[Lat] LATTEUX, M.,

(1) *Substitutions dans les EDTOL-systèmes ultralinéaires,* Publication 77, Université de Lille, 1976.

(2) *EDTOL-systèmes ultralinéaires et opérateurs associés,* Publication 100, Université de Lille, 1977.

(3) *Generateurs algebriques et langages EDTOL,* Publication 109, Université de Lille, 1978.

[Mon] MONIEN, B., *Transformational methods and their application to complexity problems,* Acta Informatica 6 (1976), 95-108 (Acta Informatica 8 (1977), 383-384).

[Rab] RABIN, M.O.,

(1) *Two-way finite automata,* Proc. Summer Institute of Symbolic Logic, p. 366-369, Cornell University, 1957.

(2) *Real-time computation,* Israel J. Math. 1 (1963), 203-211.

[RabSco] RABIN, M.O. & D. SCOTT, *Finite automata and their decision problems,* IBM J. Res. Devel. 3 (1959), 115-125.

[Raj] RAJLICH, V.,

(1) *Bounded-crossing transducers,* Inf. and Control 27 (1975), 329-335.

(2) *Absolutely parallel grammars and two-way finite-state transducers,* JCSS 6 (1972), 324-342.

[Rou] ROUNDS, W.C., *Mappings and grammars on trees,* Math. Syst. Theory 4 (1970), 257-287.

[Roz] ROZENBERG, G., *Extensions of tabled OL-systems and languages,* Int. J. Comp. Inform. Sci. 2 (1973), 311-336.

[RozVer] ROZENBERG, G. & D. VERMEIR, *On ETOL systems of finite index,* Inf. and Control 38 (1978), 103-133.

[RubFis] RUBY, S. & P.C. FISCHER, *Translational methods and computational complexity,* Proc. 6-th Ann. IEEE Symp. on Switching circuit theory and Logical design (1965), p. 173-178.

[Sch] SCHÜTZENBERGER, M.P., *A remark on finite transducers,* Inf. and Control 4 (1961), 185-196.

[Sco] SCOTT, D., *Some definitional suggestions for automata theory,* JCSS 1 (1967), 187-212.

[She] SHEPHERDSON, J.C., *The reduction of two-way automata to one-way automata,* IBM J. Res. Devel. 3 (1959), 198-200.

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

[SteHarL] STEARNS, R.E., J. HARTMANIS & P.M. LEWIS II, *Hierarchies of memory limited computations,* Proc. 6-th Ann. IEEE Symp. on Sw. circuit Th. and Log. design (1965), 191-202.

[Tha] THATCHER, J.W.,

(1) *Generalized^2 sequential machine maps,* JCSS 4 (1970), 339-367.

(2) *Characterizing derivation trees of context-free grammars through a generalization of finite automata theory,* JCSS 1 (1967), 317-322.

[ThaWri] THATCHER, J.W. & J.B. WRIGHT, *Generalized finite automata theory with an application to a decision-problem of second-order logic,* Math. Syst. Th.2 (1968), 57-81.

[Ver] VERMEIR, D., *Over strukturele restrikties op ETOL systemen* (in English) Ph.D.Thesis, University of Antwerp, 1978.

[vLe] VAN LEEUWEN, J., *Variations of a new machine model,* 17-th Ann. IEEE Symp. on Foundations of Computer Science, Houston, 1976.

MATHEMATICAL CENTRE TRACTS 108 (1979), 71-96

DYNAMIC DATA STRUCTURES

K. MEHLHORN

University of Saarland, Saarbrücken, W. Germany

The organization and manipulation of large sets of data is one of the central problems of computer science. In commercial computing centers about 1/3 of the total computing time is spent on searching and sorting. Sets of data are often dynamic in a twofold sense:

1) the set itself changes by inserting elements into it and deleting elements from it.
2) the access behavior of the users changes, i.e. the points of interest in the file change.

Let us look at an example: the set of books in a library. This set changes by the acquisition of new books (INSERT) and by discarding obsolete books (DELETE). Also the books are charged out with different probabilities (and these probabilities differ drastically from book to book). Furthermore, the access probabilities vary over time, as reading habits change. This fact could easily be accomodated for by counting the number of accesses to each book. It is also conceivable that access probabilities can change drastically sometimes. Consider a university library. Whenever a new term starts there will be a rush for the standard text books. Also, it should be possible to treat newly acquired books in different ways: the librarian might want to make a guess at the importance of a new book. In conventional libraries this is done by putting some new releases at a special shelf near the entrance door.

We propose the following definitions to model this situation. Given is a subset $S = \{B_1,\ldots,B_n\}$ of an ordered universe U. With every element $B_i \in S$ we associate a weight (= access frequency) $p_i \in \mathbb{N}$. The basic operations are ($d \in \mathbb{Z}$, $p \in \mathbb{N}$)

Member (X,S,d)	if $X \in S$ then return the information associated with X and change the weight of X by d else say "no"
Insert (X,S,p)	$S \leftarrow S \cup \{X\}$; the initial weight of X is p
Delete (X,S)	$S \leftarrow S - \{X\}$.

REMARK. We do not assume that d is specified when the operation MEMBER (X,S,d) is initiated. The case that d is a function of the old weight is also conceivable.

In this series of lectures we shall study data structures which support the three operations above efficiently. Since this is a formidable task, we proceed in five stages.

STAGE 1. The uniform problem: (= first kind of dynamics), i.e. all weights are equal to 1. In particular, d = 0 in the Member instruction and p = 1 in the Insert instruction. This is the classic *dictionary* problem. Many solutions to this problem are known (AVL-trees, 2-3 trees, HB-trees,...). We will treat weight-balanced trees (Nievergelt and Reingold).

STAGE 2. The nonuniform static case; the initial weights p_i are nonequal, no Insert and Delete instructions are allowed, and d = 0 in Member instructions. We will discuss how to construct *Binary Search Trees* and how to estimate their behavior.

STAGE 3. The nonuniform dynamic case I; this is as in stage 2, but we allow d = ±1 in Member instructions. In stage 3 the second kind of dynamics comes into play. However, access frequencies may only change slowly. We propose Dynamic Binary Search Trees (D-trees) as a solution to this problem. D-trees will be applied to digital search trees (TRIES).

STAGE 4. The uniform problem with the additional instructions Concatenate and Split

Concatenate (S_1,S_2,S_3)	$S_3 \leftarrow S_1 \cup S_2$; this operation is only applicable if $\max S_1 < \min S_2$
Split (S,a,S_1,S_2)	$S_1 \leftarrow \{X \in S;\ X \leq a\}$ $S_2 \leftarrow \{X \in S;\ X > a\}$

The sets S_1, S_2 (resp. S) cease to exist after execution of Concatenate (resp. Split).

STAGE 5. The nonuniform dynamic case II. The full repertoire of Member, Insert and Delete operations is allowed. No restriction on d and p is placed. We will show how to extend D-trees in order to cope with the full problem.

Finally we describe an application to sorting presorted files. Stages 1 and 3 will be discussed in some detail, only brief accounts are given for the others.

STAGE 1: Weight Balanced Trees

In a binary tree a node has either two sons or no sons at all. Nodes with no sons are called leaves and are drawn as rectangular boxes in subsequent figures. Non-leaf nodes (internal nodes) are drawn as circles and subtrees are drawn as triangles.

DEFINITION. Let T be a binary tree. If T is a single leaf then the root-balance $\rho(T)$ is 1/2, otherwise we define $\rho(T) = |T_\ell|/|T|$, where $|T_\ell|$ is the number of leaves in the left subtree of T and $|T|$ is the number of leaves in tree T.

For the remainder of the paper α is a fixed constant, $0 \leq \alpha \leq 1/2$.

DEFINITION. A binary tree T is said to be of bounded balance α, or in the set BB[α], if and only if

1. $\alpha \leq \rho(T) \leq 1-\alpha$
2. T is a single leaf or both subtrees are of bounded balance α.

There are two ways of storing an ordered set $S = \{B_1, \ldots, B_n\}$ in a binary tree T.

Leaf-oriented storage organization. The tree T has $|S|$ leaves. The leaves are labelled from left to right by the elements of S. An internal node v is labelled by the largest element in the left subtree of v. Label B_i in internal node v corresponds to the query:

if $X \leq B_i$ then goto root of left subtree
else goto root of right subtree.

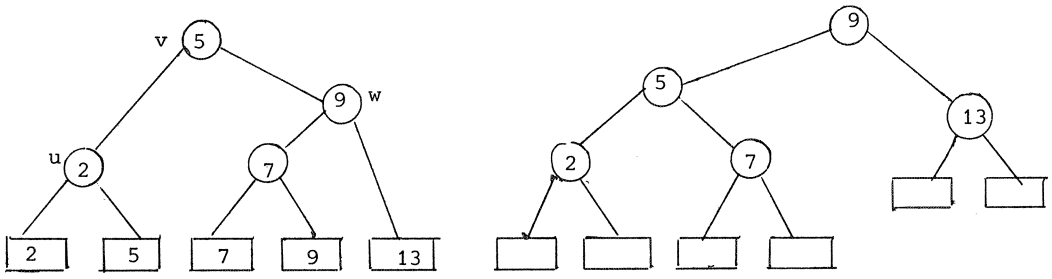
The leaf labelled B_i will be reached with all search arguments X such that $B_{i-1} < X \leq B_i$, if $i < n$, and $B_{n-1} < X$ if $i = n$.

Node-oriented storage organization. The tree T has $|S|$ internal nodes. The internal nodes are labelled from left to right by the elements of S. Label B_i in internal node v corresponds to the query:

```
case X ? B_i in
    < : goto root of left subtree
    = : found
    > : goto root of right subtree
```

Leaves correspond to unsuccessful searches in this case.

EXAMPLE. S = {2,5,7,9,13}



Leaf-oriented Node-oriented

We search in a binary tree by comparing the search argument X with the query in the root and then taking the appropiate action (= go to left subtree,...). In our example the second leaf of the node-oriented tree will be reached with all X such that $2 < X < 5$. In the leaf-oriented tree nodes u,v,w have balance 1/2, 2/5 and 2/3 respectively.

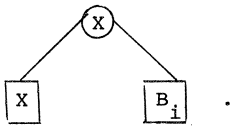
In the sequel we will always assume leaf-oriented storage organization except when explicitly stated otherwise.

LEMMA 1. *Let* $T \in BB[\alpha]$. *Then the depth of* T (= *length of longest path from root to leaf*) *is at most* $1 + (\log |T| - 1)/\log(1/(1-\alpha))$ *where* $|T|$ *is the number of leaves of* T.

PROOF. Let $v_0, v_1, \ldots, v_d$ be a longest path from the root v_0 to a leaf v_d. Let w_i be the number of leaves in the subtree with root v_i. Then $w_{i+1} \leq (1-\alpha)\, w_i$ by the definition of BB[α] tree and hence $2 \leq w_{d-1} \leq (1-\alpha)^{d-1} \cdot w_0 = (1-\alpha)^{d-1} |T|$. Taking logarithms finishes the proof. □

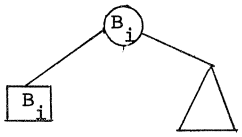
EXAMPLE. $\alpha = 1-\sqrt{2}/2 = 0.2928$. Then $1/\log(1/(1-\alpha)) = 2$.

As an immediate consequence of Lemma 1 we have that MEMBER (X,S,0) instructions take time $O(\log |S|)$. Next we turn to INSERT (X,1) and DELETE (X) instructions. We first perform a search for X; this search will end in the leaf labelled B_i with $B_{i-1} < X \leq B_i$. In the case of the INSERT instruction we are done if $X = B_i$. Otherwise we replace the leaf $\boxed{B_i}$ by the subtree

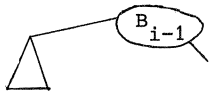


.

In the case of a DELETE instruction we are done if $X \neq B_i$. Otherwise $X = B_i$ and leaf $\boxed{B_i}$ is either the left of right son of its father. If it is the left son then we replace

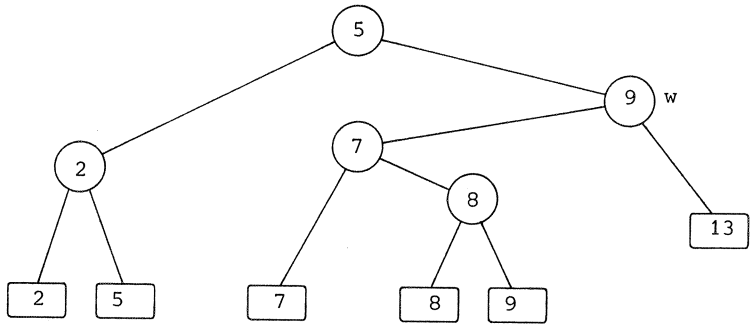


by  . If it is the right son then we replace

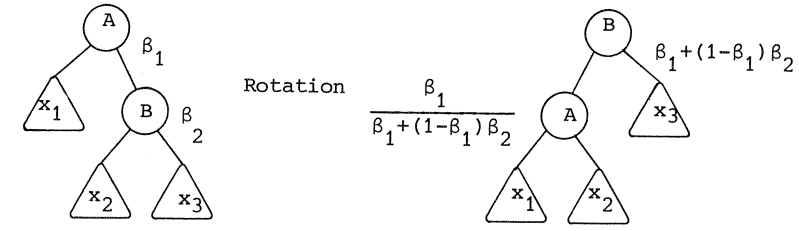


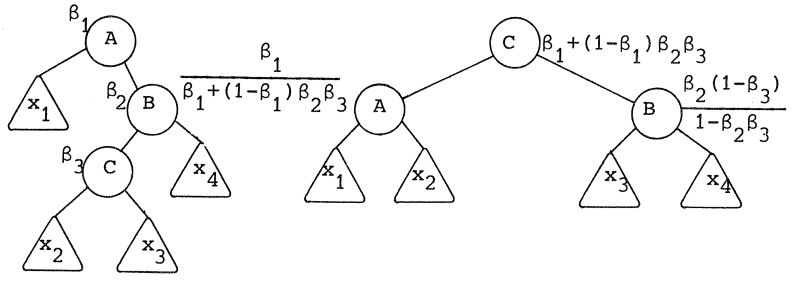
by  and furthermore we replace the interior node labelled B_i (which necessarily lies on the path from the root to leaf $\boxed{B_i}$) by an interior node labelled B_{i-1}. One problem remains. The new tree might not be in class BB[α].

EXAMPLE. Suppose we want to insert 8 into set S. Also suppose $\alpha = 1-\sqrt{2}/2 \approx 0.29$ (this choice of α becomes clear later on). We obtain



and node w is out of balance; $\rho(w) = 3/4 \notin [\alpha, 1-\alpha]$. In general some nodes on the path of search will be out of balance. Two operations (rotation and double-rotation) exist to restore balance. The figures show operations to the left about A. Symmetrical variants also exist.



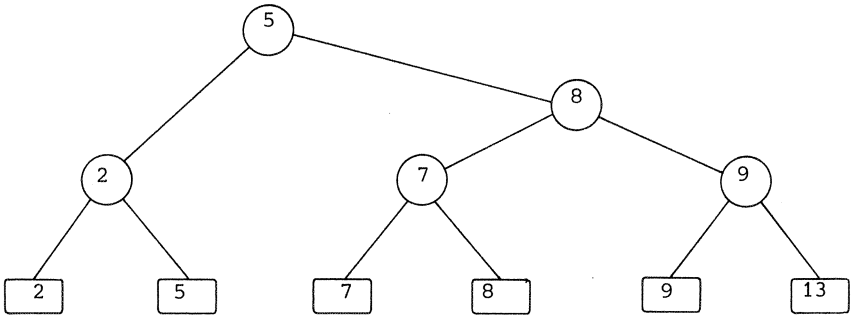


Let $\beta_1,\beta_2(\beta_1,\beta_2,\beta_3)$ be the root-balances of nodes A,B(A,B,C) before the rotation (double-rotation). Then the balances of these nodes after the rotation (double-rotation) are as given in the figure above. Consider the case of a rotation. Let x_1,x_2,x_3 denote the number of leaves of the various subtrees shown. Then $\beta_1 = x_1/(x_1+x_2+x_3)$ and $\beta_2 = x_2/(x_2+x_3)$. The balance of node B after the rotation is

$$(x_1+x_2)/(x_1+x_2+x_3) = \beta_1 + \beta_2(x_2+x_3)/(x_1+x_2+x_3)$$
$$= \beta_1 + \beta_2(1-\beta_1).$$

The other nodes are treated analogously.

EXAMPLE CONTINUED. A double rotation about w yields



and this tree is in BB[$1-\sqrt{2}/2$].

In general a tree is rebalanced by walking back from the (inserted or deleted) leaf to the root and performing rotations and double rotations as necessary. An exact statement can be found in

LEMMA 2. (Blum and Mehlhorn). *Let* $0 \leq \delta \leq 0.01$. *Then there exists a monotonically increasing function* c *with* $c(0) = 0$, $c(0.01) = 0.0045$ *such that: if* $1/4 < \alpha < 1-\sqrt{2}/2 - c(\delta)$ *then rotations and double rotations suffice to rebalance a* BB[α]*-tree upon the insertion or deletion of a leaf.*
More precisely: walk back from the inserted or deleted leaf to the root. Say we reach node A and all subtrees below A are restored to be in BB[α].

CASE 1. $\rho(A) \in [\alpha,1-\alpha]$; proceed to the father of A.

CASE 2. $\rho(A) < \alpha$; let B the right son of A. If $\rho(B) < 1/(2-\alpha) + \delta/([1+(1+\delta)(1-\alpha)](2-\alpha))$ then a rotation else a double rotation rebalances the tree, i.e.

$$\rho'(A),\rho'(B),\rho'(C) \in [(1+\delta)\alpha,1-(1-\delta)\alpha].$$

Here ρ' denotes the balance after the rotation or double-rotation. Proceed to the father of A.

CASE 3. $\rho(A) > 1 - \alpha$; symmetric to case 2.

PROOF. The lengthy but simple proof can be found in Blum and Mehlhorn. □

Lemma 2 with $\delta = 0$ shows that rotations and double-rotations along the path of search suffice to rebalance a BB[α] tree after an insertion or deletion. We obtain:

THEOREM 1 (Nievergelt and Reingold, Blum and Mehlhorn). *BB[α]-trees* ($2/11 < \alpha \leq 1 - \sqrt{2}/2$) *support the instructions* MEMBER, INSERT *and* DELETE *with processing time* $O(\log |S|)$ *per instruction.*

REMARK. Theorem 1 was first stated by Nievergelt and Reingold. The first complete proof is due to Blum and Mehlhorn.

Note that up to $O(\log |S|)$ rebalancing operations (= rotations, double rotations) may be required after a single insertion or deletion. Experiments show that on the average a constant number suffices. (Table 1 shows the findings of Baer and Schwab.) It has been a long-standing open problem whether this could actually be proven. The answer is theorem 2.

α	depth	average path length	rebalancing operations
$1 - \sqrt{2}/2$	12	9.26	426
0.25	14	9.46	206
0	22	12.14	0

TABLE I. (Baer and Schwab.) 1000 random insertions into an initially empty tree were performed for different values of α. The depth, average path length (see stage 2) of the resulting tree and total number of rebalancing operations are shown.

THEOREM 2 (Blum and Mehlhorn). *Let* $2/11 < \alpha < 1 - \sqrt{2}/2 - c(\delta)$, $0 \leq \delta \leq 0.01$ *and* c *be defined as in Lemma* 2. *Then there is a constant* d *such that:* d.m *rebalancing operation suffice to perform an arbitrary sequence of* m *insertions and deletions on an initially empty* BB[α]*-tree.*
($d \leq \min \{k - 1 + 3(1-\alpha)^k/\delta\alpha^2;\ k \in \mathbb{N},\ k \geq 12\}$).

REMARK. For $\alpha = 1/4$, $\delta = 0.01$ we obtain $d \leq 27$. There is certainly room for improvement.

The proof of theorem 2 relies upon lemma 2. The key observation is that the root-balances of nodes A,B,C after a rebalancing operation will be quite a distance (at least $\delta\alpha$) away from the critical values α and $1 - \alpha$. Hence a large number of searches (about $\delta\alpha n$ where n is the current number of leaves below such a node) can go through such a node without it being balanced again. Proper counting of rebalancing operations and of the number of insertions and deletions gives the desired result. The details can be found in Blum and Mehlhorn.

STAGE 2: BINARY SEARCH TREES.

In this section we treat the non-uniform static case; i.e. the initial weights p_i are non-equal, no Insert and Delete instructions are allowed and d = 0 in Member instruction. The weights p_i give rise to a probability distribution in a natural way: $\beta_i \leftarrow p_i/W$ where $W = \Sigma p_i$. In order to cope with leaf- and node-oriented storage organization we treat a slightly more general problem.

Given is a set $S = \{B_1,\ldots,B_n\}$ and 2n+1 probabilities $\alpha_0,\beta_1,\alpha_1,\ldots,\alpha_{n-1},\beta_n,\alpha_n$: $\alpha_j \geq 0$, $\beta_i \geq 0$ and $\Sigma\beta_i + \Sigma\alpha_j = 1$. Here β_i is the probability of accessing B_i and α_j is the probability of accessing elements X with $B_j < X < B_{j+1}$. Let T be a node-oriented search tree for set S. (If we want to talk about leaf-oriented search trees then we should set $\beta_i = 0$ for all i and α_j = probability of access of B_j). Let b_i be the depth of node B_i in T and let a_j be the depth of leaf (B_j,B_{j+1}) in tree T. Then

$$P = \sum_{i=1}^{n} \beta_i (b_i+1) + \sum_{j=0}^{n} \alpha_j a_j$$

is the weighted path length of tree T. It measures the average search time in tree T. Note that $b_i + 1$ are comparisons are required to find $X = B_i$ and a_j comparisons are required to find X with $B_j < X < B_{j+1}$.

THEOREM 3 (Mehlhorn 77b). *There is a tree T such that*

$$b_i \leq \log 1/\beta_i$$

$$a_j \leq \log 1/\alpha_j + 2$$

$$P \leq H(\alpha_0,\beta_1,\ldots,\beta_n,\alpha_n) + 2\Sigma\alpha_j$$

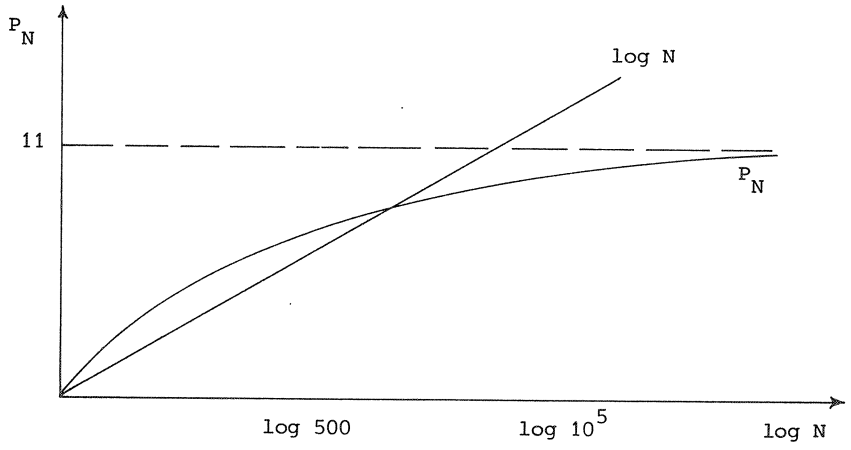
where $H = -\Sigma\beta_i \log \beta_i - \Sigma\alpha_j \log \alpha_j$ *is the entropy of the frequency distribution.*

THEOREM 4 (Bayer). *For every tree* T

$$H \leq P + \Sigma\beta_i[\log e - 1 + \log (P/\Sigma\beta_i)].$$

A tree satisfying the requirements of theorem 3 can be found in only linear time (Fredman). From theorem 4 we infer that this tree is close to optimal. Theorems 3 and 4 are alphabetic versions of Shannon's noiseless coding theorem. Algorithms for the construction of optimal trees are due to Hu/Tucker and Garsia/Wachs in the leaf-oriented case (Mehlhorn/Tsagarakis show that the two algorithms are actually the same) and due to Knuth in the node-oriented case. They have time complexity $0(n \log n)$ and $0(n^2)$ respectively. Extensions to non-binary trees can be found in Itai and Altenkamp/Mehlhorn.

Theorem 4 is the most important one for what follows. It shows that no tree can have weighted path length much less than the entropy of the distribution of access probabilities and provides us with a yardstick for near optimality. We close this section with an example. Gotlieb/Walker took a text of 10^6 words and counted word frequencies. Then they constructed (nearly) optimal binary search trees for the N most common words, N = 10, 100, 1000, 10 000, 100 000. Let P_N be the weighted



path length of the tree constructed for the N most common words. The figure (due to Gotlieb and Walker) suggests that $P_N \to 11$ for $N \to \infty$. In view of theorems 3 and 4 this observation may be explained analytically. Let β_i be

the probability of occurrence of the i-th most frequent word. Then (cf. Schwarz)

$$\beta_i \approx c/i^{1.12} \quad \text{where } c = 1/\sum_{i=1}^{\infty} 1.12$$

and (by a simple calculation)

$$H(\beta_1,\beta_2,\beta_3,\ldots) = -\sum_{i=1}^{\infty} \beta_i \log \beta_i \approx 10.2.$$

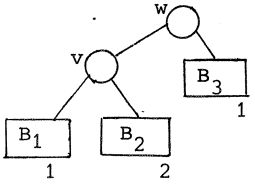
Since by theorems 3 and 4 entropy and weighted path length are closely related, this is an analytical explanation of the experiments.

STAGE 3. Dynamic Binary Search I (Mehlhorn 77c)

We are now ready to deal with the second kind of dynamics. We allow weight changes of size ±1. Let $S = \{B_1,\ldots,B_n\}$ and let $p_i \in \mathbb{N}$ be the weight of B_i. Strictly speaking, we should add a superscript t: p_i^t is then the weight of object B_i at time t.

Several solutions were proposed, notably Allan/Munro, Baer, Unterauer, Mehlhorn 77c. We describe the solution in Mehlhorn 77c, which is the only one with a proven worst case bound. The solutions proposed by Baer, Unterauer and Mehlhorn try to extend the BB[α] concept to weighted trees.

Suppose $\alpha = 1/5$ and $p_1 = 1$, $p_2 = 2$, $p_3 = 1$. Consider the following tree. We could define the



root-balances:

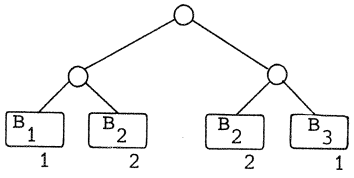
$\rho(v) = 1/(1+2) = 1/3$

$\rho(w) = (1+2)/(1+2+1) = 3/4$

Executing MEMBER (B_2,1) twice increases p_2 to 4 and we have $\rho(w) = (1+4)/(1+4+1) = 5/6$. Neither a rotation or a double-rotation (not even applicable) will help us. Even worse, there is no BB[1/5] tree for weights 1,4,1. What should we do?

Baer and Unterauer suggest to give up on the strict BB[α] idea but retain the balancing operations rotation and double-rotation. Baer expresses the hope and Unterauer proves (under reasonable probability assumptions) that this will keep the tree nearly optimal on the average.

We follow a different approach. We stick to the strict BB[α] idea but give up the concept that an object $B_i \in S$ has to be represented by a single leaf of the tree. Why not represent object B_2 by 2 leaves of weight 2 each in our example? (See also van Leeuwen.) Then a double rotation helps and gives



a tree in BB[1/5]. However, we have a new problem now. There are several leaves labeled by B_2. The way out of this dilemma is the following: one of these leaves "really" represents object B_2 (the active 2-node below), the others are only around to make rebalancing always possible (the non-active 2-nodes below). In other words, the non-active 2 nodes only serve for bookkeeping purposes. We show below that it is not necessary to store them explicitly (compact dynamic trees). Of course, there is no a-priori bound on the number of times a certain object has to split. However, our knowledge of the non-weighted case (= ordinary BB[α]-trees) tells us that parts won't have to have weight less than one. It is therefore reasonable to assume that an object of weight p consists of p "atoms" of weight 1. We are now ready for the formal definition of D-trees

D-trees are an extension of BB[α]-trees. We imagine an object B_i of weight p_i to consist of p_i leaves of weight 1. A D-tree for set S is then a BB[α]-tree T with $W = p_1 + p_2 + \ldots + p_n$ leaves. The leftmost p_1 leaves are labelled by B_1, the next p_2 leaves are labelled by $B_2, \ldots$.

DEFINITION.

a) A leaf labelled by B_j is a j-leaf.
b) A node v of T is a j-node iff all leaves in the subtree with root v are j-leaves and v's father does not have this property.
c) A node v of T is the j-joint iff all j-leaves are descendants of v and neither of v's sons has this property.
d) Consider the j-joint v. p_j' j-leaves are to the left of v and p_j'' j-leaves are to the right of v. If $p_j' \geq p_j''$ then the j-node of minimal depth to the left of v is active, otherwise the j-node of minimal depth to

the right of v is active.

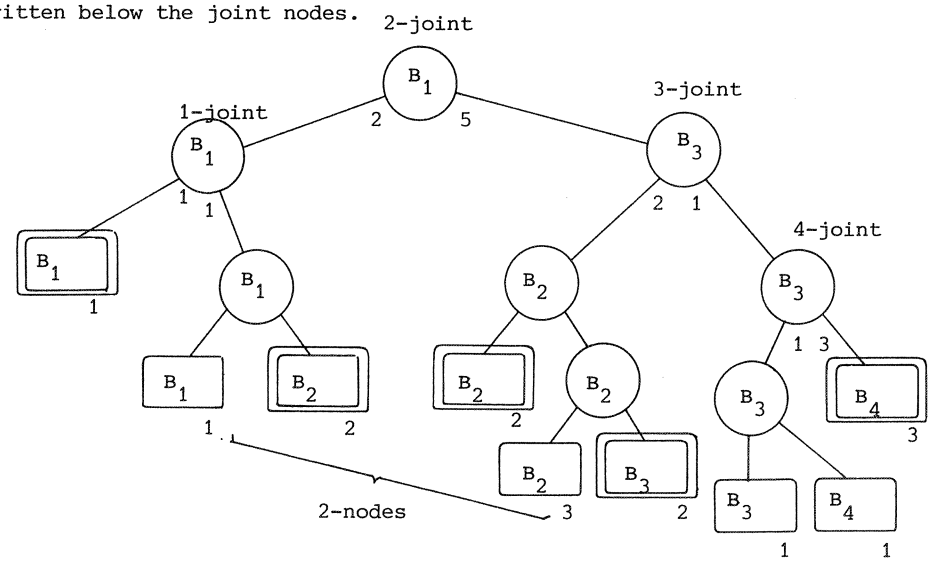
e) The thickness th(v) of a node v is the number of leaves in the subtree with root v.

Only parts of the underlying BB[α]-tree actually need to be stored, in particular all proper descendants of j-nodes can be pruned. Only their number is essential and is stored in the j-node. More precisely, a D-tree is obtained from the BB[α]-tree by

1) pruning all proper descendants of j-nodes
2) storing in each node.
   a) a query of the form "if X ≤ B then go left else go right"
   b) the type of the node: joint node, j-node or neither of above
   c) its thickness
   d) in the case of the j-joint the number of j-leaves in its left and right subtree.

The queries are assigned in such a way as to direct a search for B_i to the active i-node. More precisely, let v be any interior node of the D-tree and let the active 1-node,..., j-node be to the left of v. Then the query "if X ≤ B_j then go left else go right" is stored in v.

The next figure shows a D-tree for the distribution (p_1,p_2,p_3,p_4) = = (2,7,3,4) based on a tree in BB[1/4]. The j-nodes are indicated by squares, active j-nodes by double lines, the thickness of j-nodes is written below them and the distribution of j-leaves with respect to the j-joints is written below the joint nodes.



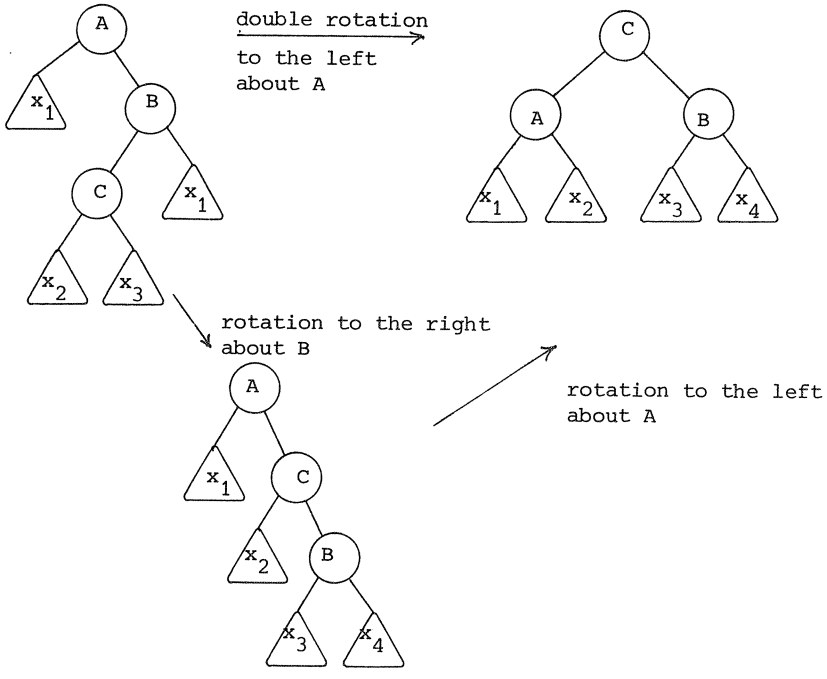
The following Lemma shows that D-trees are good search trees.

LEMMA 3. *Let* b_j *be the depth of the active* j*-node in tree* T. *Then* $b_j \leq c_1 \log W/p_j + c_2$ *where* $c_1 = 1/\log(1/(1-\alpha))$, $c_2 = 1 + c_1$.

EXAMPLE. For $\alpha = 1 - \sqrt{2}/2$ we have $c_1 = 2$ and $c_3 = 3$. In the light of Theorem 4 we have that search time in D-trees is at most twice the search time in optimum trees and usually much better (cf. experimental data below).

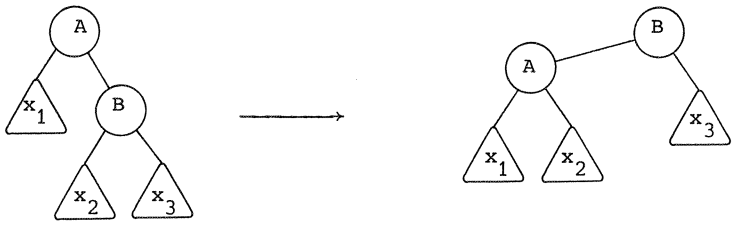
PROOF. Let v be the father of the active j-node. Then all j-leaves which are on the same side of the j-joint as the active j-node are descendants of v. Hence $th(v) \geq p_j/2$. The argument of Lemma 1 will finish the proof. □

Next we have to address the question of how to maintain D-trees. The answer is exactly as for BB[α]-trees, but be careful with the additional D-tree information. Suppose we execute a MEMBER (B_j,±1) instruction. The search will end in the active j-node. We have to update the thickness of all nodes on the path of search and the distribution of j-leaves with respect to the j-joint. The j-joint lies on the path of search and so this is easily done. Next we have to ascend the path of search from the active j-node to the root and perform rotations and double-rotations as required. Since a double-rotation is two rotations



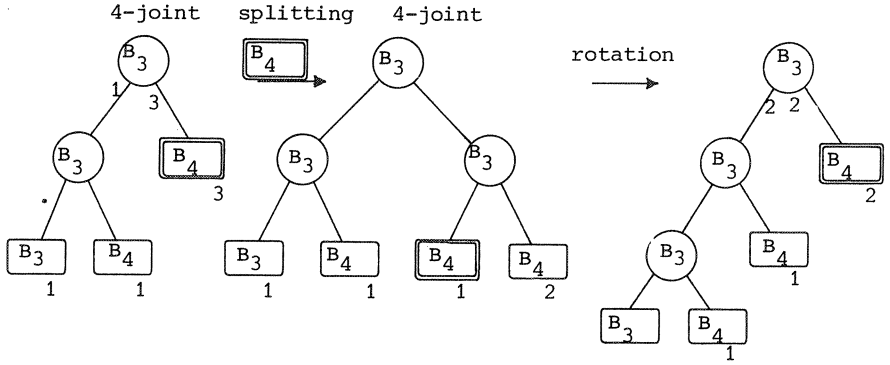
we only have to treat the case of a rotation. Let's call joint-nodes and j-nodes special nodes. If no special node is involved in the rotation then no additional actions are required. Suppose now, a special node is involved in the rotation.

CASE 1. A j-node is involved. Then we have the following picture



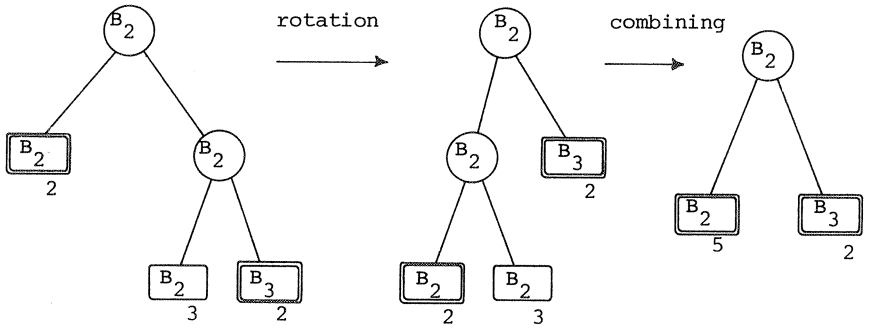
and node B is a j-node before the rotation, i.e. trees x_2 and x_3 do not exist explicitly. We create them by splitting B into two j-nodes of thickness $\lfloor th(B)/2 \rfloor$ and $\lceil th(B)/2 \rceil$ respectively. What query should we assign to B (Note that B is an interior node now)?. Suppose first that neither A nor B is the j-joint. Then A must be a left descendant of the j-joint. Otherwise x_1 can only contain j-leaves and hence A would be a j-node and hence B would not exist. So A must be a left descendant of the j-joint and hence the active j-node lies to the right of A. But then it also lies to the right of B (x_3 could be it) and thus we only have to copy the query from A into B. The discussion above also solves the case where B is the j-joint. Suppose next that A is the j-joint. Then the active j-node will be to the left of B after the split. Let Z be the nearest ancestor of A such that the left link was taken out of Z during theh search. Copy Z's query into B. Z can be found as follows: When the nodes on the path of search are stacked during the search, they are also entered into either one of two linear lists: the L-list or the R-list. The L-list contains all nodes which are left via their left links and the R-list contains all nodes which are left via their right links. Then Z is the first node on the L-list. This ends the discussion of B being a j-node.

EXAMPLE. Rotation to the left about the 4-joint.



The second possibility is that x_2 and x_3 are j-nodes and hence A is j-node after the rotation. In this case x_1 and x_2 are deleted after the rotation.

EXAMPLE. Rotation to the left about the father of the active 2-node



CASE 2. A joint node is involved, i.e. either A or B is a joint node or both. If B is a joint node then no additional actions are required. So let us consider the case that A is the j-joint. Let p'_j, p''_j be the distribution of j-leaves with respect to the j-joint A and let s be the thickness of the root of x_2. If $s \geq p''_j$ then x_3 contains no j-leaves and hence A will be the j-joint after the rotation. No action is required in this case.

If $s < p''_j$ then B will be the j-joint after the rotation. The distribution of j-leaves with respect to B is $p'_j + s$, $p''_j - s$.

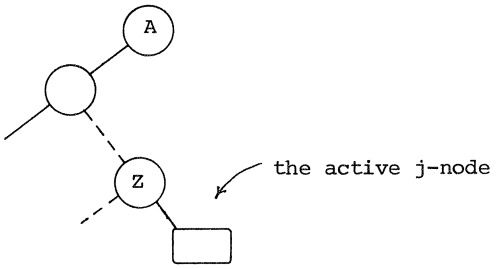
CASE 2.1. $p'_j + s \leq p''_j - s$. Then $p'_j \leq p''_j$ and the active j-node was to the right of A, in fact it was node x_2. Also the active j-node will be to the

right of B after the rotation and it still is to the right of A. Hence we only have to copy A's query into B.

CASE 2.2. $p'_j + s > p''_j - s$. Then the active j-node will be to the left of B after the rotation, and hence it will be node x_2.

CASE 2.2.1. $p'_j \leq p''_j$. Then x_2 also was the active j-node before the rotation. No additional action is required in this case.

CASE 2.2.2. $p'_j > p''_j$. Then the active j-node was to the left of A and hence to the left of B before the rotation. In this case B's query remains unchanged, but A's query has to be changed. Suppose first that A's left son is a j-node. Then A ceases to exist after the rotation and we are done. Suppose next that A's left son is not a j-node. The next figure shows a microscopic view of tree x_1.



We only have to copy Z's query into A. Z can be found by a brute force search. Note that $th(z) \geq p'_j \geq p_j/2$. Note also that the thickness s of x_3 is less than $p''_j \leq p_j/2$. Since $s = th(x_3) \geq \alpha \cdot th(B)$ (the underlying tree is in BB[α]) and $th(B) \geq (1-\alpha)th(A)$ (a rotation to the left about A is performed) we have $s \geq \alpha(1-\alpha)th(\alpha)$ and hence $th(A) \leq p_j/(2 \cdot \alpha(1-\alpha))$. The argument used in the proof of Lemma 1 shows that the depth of Z with respect to A is at most $\log(\alpha(1-\alpha))/\log(1-\alpha)$.

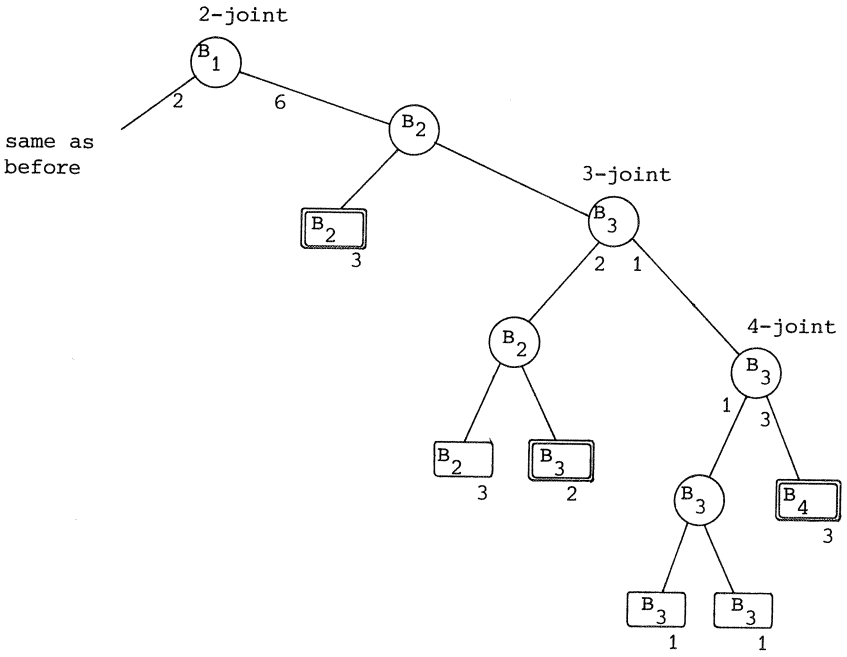
REMARKS. For $\alpha = 1 - \sqrt{2}/2$ we have $\log(\alpha \cdot (1-\alpha))/\log(1-\alpha) \approx 4.4$. Case 2.2.2 is not very likely to occur. In our simulations (several hundred thousand MEMBER (,+1) instructions) it never occurred.

We summarize the discussion in

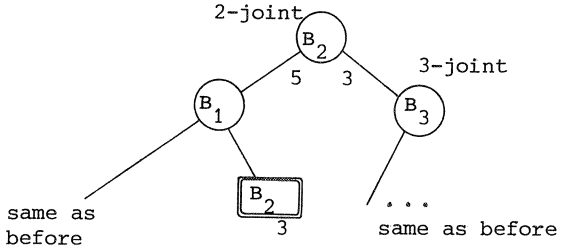
THEOREM 5 (Mehlhorn 77c). *Consider a* D-*tree based on a* BB[α]-*tree with* $2/11 < \alpha \leq 1 - \sqrt{2}/2$. *Let* p_i^t *be the weight of* B_i *at time* t, $1 \leq i \leq n$ *and let* $W^t = \sum_{i=1}^{n} p_i^t$. *A search for* B_i *at time* t *takes time* $c_1 \log W^t/p_i^t + c_2$.

Also a weight change by ± 1 *at time* t *takes time* $c_1 \log W^t/p_i^t + c_2$ *for some small constants* c_1 *and* c_2.

EXAMPLE. Suppose we want to execute a MEMBER (B_2,+1) instruction. This would increase the thickness of the active j-node from 2 to 3 and move the balance parameter of the root (the active 2-joint) out of the range [1/4,3/4]. A double-rotation (to the left) about the root is required. It is simulated by a rotation to the right about the 3-joint followed by a rotation to the left about the 2-joint. The rotation about the 3-joint requires no special action since s = th(father of active 3-node) = 5 > 2 = number of 3-leaves to the left of 3-joint. We obtain



Next we have to rotate about the 2-joint. We have $p_2' = 2$, $p_2'' = 6$ and s = 3 and hence case 2.2.1 of the discussion above applies. We obtain



Having described the theory of D-trees to some extent the reader might be interested in experimental data. H. Reinshagen and A. del Fabro programmed D-trees and carried out the following experiments. They took an arbitrary BB[$1-\sqrt{2}/2$] tree with 200 leaves and $p_1 = \ldots = p_{200} = 1$. Then they executed 30 000 MEMBER (,+1) instructions according to a fixed probability distribution (distribution I: $p_i = 100^i/(i!\ e^{100})$, distribution II: obtained by counting words starting with different 2 letter prefixes). The weighted path length of the actual D-tree and the total number of rotations and double rotations performed was recorded. The following table shows the

# of searches	$\frac{P_{actual}-P_{opt}}{P_{opt}} .100$	# ROT	$\frac{P_{actual}-P_{opt}}{P_{opt}}$	# ROT
0	48.6	0	22.9	0
100	35.7	34	14.5	52
500	19.8	51	9.9	148
1000	11.9	58	7.6	207
5000	1.7	84	5.8	370
10000	1.7	90	5.7	420
20000	1.8	93	5.7	440
30000	1.7	96		

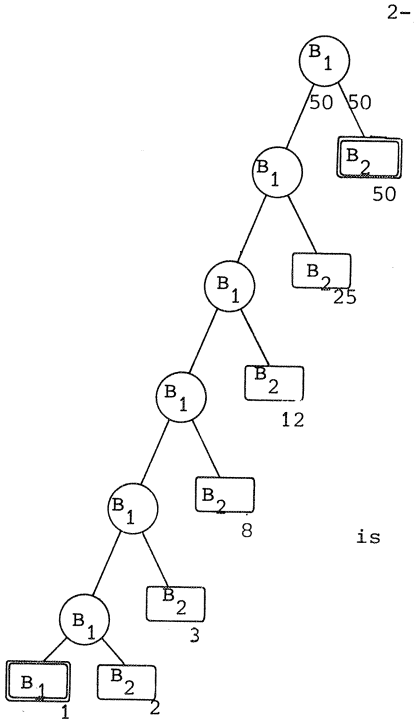
deviation (in percent) of the actual weighted path length from the weighted path length of the optimal tree for distribution I and II respectively. It also shows the number of rotations and double rotations required.

COMPACT D-TREES

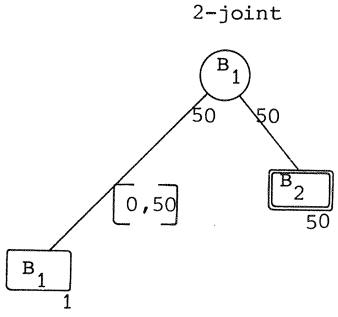
Non-active j-nodes only serve bookkeeping purposes; they permit a uniform treatment of rebalancing operations. In this section we indicate that they need not to be stored explicitly: compact D-trees. We introduce compact D-trees by way of example, the full theory can be found in [Mehlhorn 77 c].

In compact D-trees only those nodes are actually stored which are essential for the searches: the active j-nodes, the branch nodes (i.e. nodes having active descendants in both subtrees) and the joint-nodes. All other nodes are deleted, however their thickness is remembered.

Consider the following example; $p_1 = 1$, $p_2 = 100$. The compact version of the D-tree



is



The expression [0,50] on the right side of the edge from the 2-joint to the active 1-node denotes that right subtrees of that path containing a total number of 0 1-leaves and 50 2-leaves were deleted.

Compactification of (extended) D-trees to compact D-trees is a many-one mapping, i.e. in general many D-trees are represented by the same compact D-tree. The essential point is that one D-tree in the inverse image of a compact tree with respect to the compactification mapping is computed easily; in fact, reconstruction can be done locally.

Consider our example again. Say we want to expand the edge [0,50] again. The query of the top node of the edge being B_1, we know that [0,50] represents 0 1-leaves and 50 2-leaves. The thickness of the bottom node is 1. Hence we might partition the 50 2-leaves into pieces of size 1,2,4,8,16, 19 and obtain a tree in BB[1/4]. For full details we refer the reader to [Mehlhorn 77c]. Some further compactifications are possible: it is possible to approximately reconstruct the edge labels during the search and to use height-balanced trees instead of weight-balanced trees [Del Fabro/Mehlhorn].

STAGE 4. The uniform problem with the additional instructions: Concatenate and Split. The traditional name for data structures supporting the instruction Member, Insert, Delete, Concatenate and Split is Concatenable Queue. It has long been known that height-balanced trees support the full repertoire of Concatenable Queue operations with $O(\log|S|)$ processing time per instruction. We show that weight-balanced trees also support the full repertoire with the same time bound.

LEMMA (Mehlhorn 78). *BB[α]-trees support the full repertoire of concatenable queue instructions with* $O(\log|S|)$ *processing time per instruction.*

STAGE 5. The nonuniform dynamic case II. We finally treat the full problem: Member, Insert and Delete operations are allowed. No restriction on p and d is placed. Using the techniques developed in stage 4 we extend D-trees to cope with the full problem.

THEOREM 6 (Mehlhorn 78). *Let* $2/11 < \alpha \leq 1-\sqrt{2}/2$ *and let* T *be a* D*-tree for set* $S = \{B_1, B_2, \ldots, B_n\}$ *based on a* BB[α]*-tree. Let* p_i *be the weight of* B_i *and let* $W = \Sigma p_i$.

a) *The operation Member* (B_i, S, d) *takes time* $O(\min[\log W/\min(p_i, p_i+d), n])$

b) *The operations Insert* (X, S, p) *and Delete* (B_i, S) *take time* $O(\min(\log W, n))$.

Part a) of theorem 6 says that the execution time of the instruction MEMBER (B_i, S, d) is at most proportional to the logarithm of the old access probability W/p_i or the new access probability $W(p_i+d)$. (Since a compact D-tree has depth at most n, execution time is also $O(n)$). In view of theorem 4 this is optimal up to a constant factor. Part b) is almost a corollary of part a) if one observes that either the old weight (INSERT) or the new weight (DELETE) is 0 in this case.

We are now at the end of a long journey. We finally arrived at a solution to the problem posed in the introduction. We close with a brief discussion of two applications.

AN APPLICATION TO TRIES

An alternative to searching based on key comparion is digital searching. Here a key is identified by successive identification of its component characters. One such method is the TRIE. A set of strings over some alphabet Σ is represented by its tree of prefixes. So every node of a trie corresponds

to a word over $\Sigma = \{a_1,\ldots,a_p\}$.

Several implementations of tries were proposed.

1) Each node of the trie is represented by a vector of length $|\Sigma|$ (Fredkin). Identification of a character is done by indexing this vector. This method is very fast (one access per character) but it uses a large amount of storage.
2) Each node of the trie is represented by a linear list (Sussenguth). In a node w this list contains only those characters $a \in \Sigma$ such that wa is a prefix of some key. Identification of a character is done by a linear search through the list. This method is slow (up to $|\Sigma|$ comparisons per character) but it saves storage space.
3) Each node of the trie is represented by a binary search tree (Clampton). In a node w of the trie this tree contains those characters $a \in \Sigma$ such that wa is a prefix of some key. Identification of a character is by tree searching. This method is a compromise in speed and space requirement.

Let $S = \{B_1,\ldots,B_n\} \subseteq \Sigma^*$ be the set of keys and suppose that all keys are of equal length m (this is not essential but makes life easier), $|s| = n$. Clampton proposed to use a balanced binary search tree for each node.

EXAMPLE. $S = \{a_1^k a_j^{m-k};\ 0 \le k \le m,\ 1 \le j \le p\}$. Then $|S| = m \cdot p$ and a search for $X = a_1^{m-1}a_j$ takes time $O(m \log p) = O(|S| \cdot \log p/p)$.

We propose to use D-trees (or any other kind of nearly optimal search trees). More precisely, for $w \in \Sigma^*$ let

$$p_w = |\{B_i;\ w \text{ is prefix of } B_i\}|.$$

A node w of a TRIE is represented by a D-tree for the distribution $\{p_{wa};\ a \in \Sigma\}$. A key $B_i = a_{i1}a_{i2}\ldots a_{im}$ is identified by successively identifying the character a_{ik} in the tree corresponding to the node $a_{i1}\ldots a_{i(k-1)}$ of the tree. It takes time

$$O(c_1 \cdot \log p_{a_{i1}\ldots a_{i(k-1)}}/p_{a_{i1}\ldots a_{ik}} + c_2)$$

to identify a_{ik} where c_1, c_2 only depend on the balance parameter (cf. lemma 3). Hence B_i can be identified in time

$$O(c_1 \log p_\varepsilon/p_{a_{i1}\ldots a_{im}} + c_2 m) = O(c_1 \log n + c_2 m).$$

Since log n comparisons are required in any scheme based on comparisons with binary outcome and every character of the input has to be inspected we have nearly optimal tries under implementation 3.

We use D-trees to implement the nodes of a trie because we want to deal with updates, i.e. insertions and deletions of names. Suppose we want to insert a new name B into the set S. This amounts to increase p_w by 1 for all prefixes of B. Retaining near optimality is no problem since we used D-trees to implement the nodes of a trie. Conversely, suppose we want to delete a name B from the set S. This amounts to decrease p_w by 1 for all prefixes of B. Again retaining near optimality causes no problems. We thus proved

THEOREM 7. *Let S be a set of keys of* m *characters each. If a trie is used to represent the set* S *and every node of the trie is implemented as a* D-*tree then searching for a key in* S, *inserting a new key into* S *and deleting a key from* S *can be done in time* $O(\log|S| + m)$ *and this is uptimal up to a constant factor.*

EXAMPLE continued. In the example above, TRIES + D-trees guarantee that searches never take more than $O(m + \log(m \cdot p))$ time units.

Note that TRIES + D-trees give execution times which are independent of $|\Sigma|$. In database applications objects are often m-tuples (e.g. cities given by geographical altitude and latitude). In these applications $|\Sigma| = \infty$ is conceivable.

AN APPLICATION TO SORTING

Consider the problem of sorting a sequence $x_n x_{n-1} \ldots x_1$ by an insertion sort. Insertion sort proceeds by successively inserting x_i into its proper position. Let

$$f_i = |\{j;\ x_j < x_i \quad \text{and} \quad j < i\}|.$$

When x_i is inserted into the sorted version of sequence $x_{i-1} \ldots x_1$ then x_i has to be insorted after the f_i-th position of that sequence. If the sequence is presorted, i.e. $F = \Sigma f_i$ is small with respect to n^2, then the elements tend to be inserted near the front of the already sorted subsequence. Using the concepts developed in stage 3, Fredman has shown that it

should be possible to sort the sequence with $O(n \log(F/n))$ comparisons. Later-on practical versions of Fredman's algorithm were developed by Guibas et al., Brown/Tarjan and Mehlhorn 79. The algorithm described by Mehlhorn is based on AVL-trees and has running time $26\, n \log F/n + 40n$ on the machine described in Mehlhorn 77a (similar to MIX). Comparing this with Quicksort's running time of $9n \log n$ on the same machine gives

$$26\, n \log F/n + 40n \leq 9n \log n$$

iff

$$F \leq 0.314\, n^{1.375}.$$

For presorted files the new method will be superior.

SUMMARY. New approaches to searching and sorting were discussed which exploit the fact that in some applications search requests or sequences to be sorted are non-random. More specifically a tree structure (D-trees) was presented which supports searching in, inserting into, deleting from and changing weights in a weighted set S in time optimal up to a constant factor. Also a sorting method which sorts presorted input sequences in time strictly less than n log n was presented.

REFERENCES

ALLAN, A. & I. MUNRO, *Self organizing binary search trees*, JACM, 25 (1978), pp. 526-535.

ALTENKAMP, D. & K. MEHLHORN, Codes, *unequal letter costs, unequal probabilities*, 5th JCALP, 1978, Springer Lecture Notes in Computer Science vol. 62, pp. 15-25, to appear JACM.

BAER, J.L., *Weight-balanced trees*, Proc. AFIPS, vol. 44 (1975), pp. 467-472.

BAER, J.L. & B. SCHWAB, *A comparison of tree balancing algorithms*, CACM 20 (1977), 322-330.

BAYER, P., *Improved bounds on the cost of optimal and balanced binary search trees*, techn. report, Dept. of Computer Science, MIT, 1975.

BLUM, N. & K. MEHLHORN, *On the average number of balancing operations in weight-balanced trees*, 4th GI Conference on Theoretical Computer Science, Aachen, 1979 (to appear).

BROWN, M.R. & R.E. TARJAN, *A representation for linear lists with movable fingers,* 10th ACM STOC, 1978, p. 19-29.

CLAMPTON, H.A., *Randomized binary searching with tree structures,* CACM 7, 3 (March 1964), 163-165.

DEL FABRO, A. & K. MEHLHORN, *Further compactification of* D-*trees,* in preparation.

FREDKIN, E., *Trie memory,* CACM 3, 9 (sept. 60), 490-499.

FREDMAN, M.L., *Two applications of a probabilistic search technique: Sorting* X+Y *and building balanced search trees,* Proc. 7th Annual ACM Symp. on Theory of Computing, Albuquerque, 1975, pp.240-244.

GARSIA, A.M. & M.L. WACHS, *A new algorithm for minimum cost binary trees,* SICOMP 4 (1977), 622-642.

GOTLIEB, C.C. & W.A. WALKER, *A top-down algorithm for contructing nearly optimal lexicographical trees,* in: R.C. Read (ed.), Graph Theory and Computing, Academic Press, London, 1972, pp.303-323.

GUIBAS, L.J., E.M. MCCREIGHT, M.F. PLASS, J.R. ROBERTS, *A new representation for linear lists,* 9th ACM STOC, 1977, 49-60.

HU, T.C. & A.C. TUCKER, *Optimal computer search trees and variable length alphabetic codes,* SIAM J. Applied Math. 21, 1971.

ITAI, A., *Optimal alphabetic trees,* SICOMP 5 (1976), 9-18.

KNUTH, D.E., *The art of computer programming,* vol. 3, *Sorting and searching,* Addison Wesley, 1973.

MEHLHORN, K., 77a, *Effiziente Algorithmen,* Teubner Studienbücher Informatik, Stuttgart 1977.

MEHLHORN, K., 77b, *Best possible bounds on the weighted path length of optimum binary search trees,* SICOMP 6 (1977) pp. 235-239.

MEHLHORN, K., 77c, *Dynamic binary search,* 4th Colloquium on Automata, Languages and Programming Turku, 1977, Springer Lecture Notes in Computer Science, vol. 52, pp. 323-336.

MEHLHORN, K., 78, *Arbitrary weight changes in dynamic trees,* Techn. Bericht, Universität des Saarlandes, May 1978.

MEHLHORN, K., 79, *Sorting presorted files,* 4th GI conference on Theoretical Computer Science, Aachen, 1979 (to appear).

MEHLHORN, K. & M. TSAGARAKIS, *On the isomorphism of two algorithms,* Hu/Tucker & Garsia/Wachs, 4 ieme colloque de Lille, Feb. 1979, Lille, France.

NIEVERGELT, J. & E.M. REINGOLD, *Binary search trees of bounded balance,* SICOMP 2 (1973) 33-43.

SCHWARTZ, E.S., *A dictionary for minimum redundancy encoding,* JACM 10 (1963), 413-439.

SUSSENGUTH, E.H., *Use of tree structures for processing files,* CACM 6 (1963) 272-279.

UNTERAUER, K., *Optimierung gewichteter Binärbäume zur Organisation geordneter dynamischer Dateien, Doktor-arbeit,* TU München, 1977.

VAN LEEUWEN, J., *On the construction of Huffman-trees,* in: S. Michaelson & R. Milner (eds), Automata, Languages and Programming (Proc. of the 3rd Colloq.) Edinburgh Univ. Press, Edinburgh, 1976, pp. 382-410.

THE FUNDAMENTAL THEOREM OF COMPLEXITY THEORY by A. MEYER & K. WINKLMANN
(preliminary version)

MATHEMATICAL CENTRE TRACTS 108 (1979), 97-112

THE FUNDAMENTAL THEOREM OF COMPLEXITY THEORY

(Preliminary version)

A.R. MEYER & K. WINKLMANN

MIT, Cambridge, USA

1. INTRODUCTION

The amount of resources (such as time, space) used by an optimal program for a recursive function f describes the inherent computational complexity of the function. It is well-known, however, that there are recursive functions which do not have optimal programs ("Speed-up Theorem", [Bℓ67]). While *single* functions are therefore not always adequate for describing the complexity behavior of recursive functions, *sequences* of functions are: the complexity of any recursive function can be described by a recursive sequence of "honest" functions. Conversely, any recursive sequence of "honest" functions satisfying a few weak and simple properties does indeed describe the complexity behavior of some total recursive function. Combined we refer to these two results as the Fundamental Theorem of Complexity Theory. Both the Speed-up Theorem of BLUM [Bℓ67] and the Compression Theorem [Bℓ67,HS65] are corollaries to such a Fundamental Theorem. [2)]

Such a Fundamental Theorem can be proven in an axiomatic setting, only assuming that the use of resources satisfies the very general axioms of BLUM [MF72]. This generality is necessarily paid for by the introduction of "overhead" functions which somewhat obscure the statement of the Theorem. Several versions of the Fundamental Theorem which have been given for specific measures (time, space on various machine models [Ly75,SS75]) all either use overhead functions in the statement of the Theorem or use unnecessary assumptions (e.g. monotonicity of the sequences) when proving that all "reasonable" candidates indeed describe the complexity of some recursive function. In this paper we prove a version of the Fundamental Theorem for Turing machine space which uses no overhead functions and only very weak restrictions on what sequences of functions are considered "reasonable" candidates for describing the complexity of some recursive function. We assume uniform computability of the sequences plus a few simple properties

---

1) All footnotes appear at the end of the text.

which all complexity sequences (for Turing machine space) have.

In Section 2 we state the Fundamental Theorem for Turing machine space, show that both Speed-up and Compression Theorems are corollaries, and discuss the significance of such a Fundamental Theorem. In Section 3 we give a proof of the Fundamental Theorem for Turing machine space.

2. THE FUNDAMENTAL THEOREM FOR TURING MACHINE SPACE

We use Turing machines with a single two-way read-only input tape, a single one-way write-only output tape, and some constant number of worktapes. Both input and output are written in binary, with special markers separating multiple inputs. The worktape alphabet varies from machine to machine. Consider some fixed standard enumeration $M_0, M_1, \ldots$ of these machines. ϕ_i denotes the function computed by M_i, and $S_i(x)$ denotes the "space" used by M_i on input x, i.e. $S_i(x)$ is the number of different squares on worktape(s) of M_i visited by some read-write head during computation of M_i on input x. If M_i does not halt on input x, however, we consider $S_i(x)$ to be undefined, even if M_i "loops" within bounded space. Let P_n (respectively R_n) denote the partial (resp. total) recursive functions of n arguments.

DEFINITION 1 [MF72]. For $f \in R_1$, a sequence $p_0, p_1, \ldots$ of total functions is a *space-complexity sequence for* f if

(1) $\quad \forall i\, \exists j[\phi_j = f \wedge [S_j = p_i \text{ a.e.}]]$,

and

(2) $\quad \forall j[\phi_j = f \supset \exists i[S_j \geq p_i \text{ a.e.}]]$. [3]

DEFINITION 2. A sequence $p_0, p_1, \ldots$ of total recursive functions is a *space-candidate sequence* if

(3) each p_i is space-constructible, i.e. $\forall i\, \exists j[S_j = p_i]$;

(4) $p_0, p_1, \ldots$ accommodates linear tape reduction,
i.e. $\forall i\, \forall c > 0\, \exists j[p_j \leq \lceil p_i/c \rceil \text{ a.e.}]$; [4]

(5) $p_0, p_1, \ldots$ accommodates parallelism, i.e. $\forall i,j\, \exists k[p_k \leq \min(p_i, p_j)$ a.e.]; and

(6) $p_0, p_1, \ldots$ is uniformly computable; i.e. $\lambda i,x[p_i(x)] \in R_2$.

This definition of space-candidate sequences captures the intuitive notion of sequences which "can reasonably be expected to be space-complexity sequences". It is easy to verify from our definitions that every *uniformly computable* space-complexity sequence for a recursive function f is a space-candidate sequence: Property (3) of space-candidate sequences follows from Property (1) of space-complexity sequences, Property (4) follows from the fact that a linear reduction of the space used can always be achieved by increasing the size of the worktape alphabet (cf. [HU69,HS65]), and Property (5) is a consequence of the fact that given two machines, M_i and M_j, both computing the same function f, there is a third machine, M_k, which also computes f and does so in space $\min(S_i,S_j)$. A straightforward choice for M_k would be a machine which allocates increasing amounts of space for simulating both M_i and M_j (suppressing their output) until one of them is seen to terminate within the allocated space; M_k then would run M_i or M_j, whichever terminated, again, this time printing its output.

DEFINITION 3. Let $p_0,p_1,\ldots$ and $q_0,q_1,\ldots$ be two sequences of total functions. $|x|$ denotes the length of the binary representation of the number x and $\ell g\ell(x) = \log_2|x|$.

a. $p_0,p_1,\ldots$ is *decreasing* if $\forall i[p_{i+1} \leq p_i]$.

b. $p_0,p_1,\ldots$ is *of growth at least* $\ell g\ell$ if $\forall i\ \exists c[p_i \geq \lfloor \ell g\ell/c \rfloor]$.

DEFINITION 4. Two sequences $p_0,p_1,\ldots$ and $q_0,q_1,\ldots$ of total functions are *equivalent* if $\forall i\ \exists j[p_i \leq q_j$ a.e.] and $\forall i\ \exists j[q_i \leq p_j$ a.e.]. We write $p_0,p_1,\ldots \equiv q_0,q_1,\ldots$.

DEFINITION 5. A total recursive function f is of *space complexity at least* s if $\forall i[\phi_i=f \supset \exists c[S_i \geq \lfloor s/c \rfloor]]$.

The following Lemma 1 justifies the above definition of equivalence between sequences. It follows easily from our definitions; we do not give a proof.

LEMMA 1. *Let* $p_0,p_1,\ldots$ *and* $q_0,q_1,\ldots$ *be two sequences of total functions. If* $p_0,p_1,\ldots \equiv q_0,q_1,\ldots$ *then for all total recursive functions* f, $p_0,p_1,\ldots$ *is a space-complexity sequence for* f *if and only if* $q_0,q_1,\ldots$ *is.*

Clearly $\equiv$ is an equivalence relation among sequences of total functions. As a converse to Lemma 1 it is easy to verify that any two complexity

sequences of a single total recursive function are equivalent. This naturally yields a partition of all total recursive functions into classes of functions with the same "complexity", where "complexity" can be defined as an equivalence class of complexity sequences under $\equiv$. Informally speaking, Part 1 of the Fundamental Theorem below says that every imaginable kind of complexity of total recursive functions indeed does occur. Part 2 says that each such complexity can be described by a uniformly computable and decreasing complexity sequence.

FUNDAMENTAL THEOREM

1. *Every space-candidate sequence* $p_0,p_1,\ldots$ *of growth at least* $\ell g\ell$ *is a space-complexity sequence for some* 0-1 *valued total recursive function* f. *Moreover, a program for such an* f *can be found effectively from a program for* $p_0,p_1,\ldots$.
2. *Every total recursive function* f *that is of space-complexity at least* $\ell g\ell$ *has a decreasing space-complexity sequence* $p_0,p_1,\ldots$ *which is also a space-candidate sequence. Moreover, a program for such a* $p_0,p_1,\ldots$ *can be found effectively from a program for* f.

We give a proof of this Fundamental Theorem in the next Section. In the remainder of this Section we first show that both the Compression and the Operator Speed-up Theorem (from [Bℓ167] and [MF72] respectively) are corollaries to this Fundamental Theorem, and then illustrate the significance of such a Fundamental Theorem with two examples.

COROLLARY 1 (COMPRESSION THEOREM). *Let* p *be any space-constructible (i.e.* $\exists j[S_j=p]$*) total function of growth at least* $\ell g\ell$. *Then there is a total recursive function* f *of space complexity* p, *i.e. an* f *for which* $p_0,p_1,\ldots$ *with* $p_1 = \lceil p/(i+1) \rceil$ *is a complexity sequence.*

PROOF. Immediate from the first part of the Fundamental Theorem. □

The assumption that p be a space-constructible function is essential for the Compression Theorem. This is shown by the following Theorem, which is due to BORODIN [Bo72].

GAP THEOREM [Bo72]. *For any* $r \in R_2$ *with* $r(x,y) > y$ *there is a function* $t \in R_1$ *such that*

$$\forall i\ \forall x > i\ \neg[t(x) \leq S_i(x) \leq r(x,t(x))].$$

PROOF. The desired t can be defined by

$$t(x) = \min \{y: \forall i < x \ [y \leq S_i(x) \leq r(x,y)]\}.$$

It is not hard to verify that $t \in R_1$. We omit the details. □

We remark that the strong form of compression given in Corollary 1 actually holds for all space-constructible p, not just total ones of growth at least $\ell g \ell$. Corollary 1 was first observed by TRACHTENBROT [Tr70] and later independently by BORODIN *et.al* [BCH69] and MEYER [MC71].

DEFINITION 5. A mapping F from P_1 to P_1 is an *effective operator* if there is an $\alpha \in R_1$ such that $\forall i[F(\phi_i) = \phi_{\alpha(i)}]$. F is *total* if $\forall i[\phi_i$ total $\supset F(\phi_i)$ total].

COROLLARY 2 (OPERATOR SPEED-UP,[MF72]). *For any total effective operator* F, *there is a total recursive* 0-1 *valued function* f *that has* F*-speed-up, i.e.* $f \in R_1$ *is* 0-1 *valued and* $\forall i[\phi_i = f \supset \exists j[\phi_j = f \wedge [F(S_j) \leq S_i$ a.e.]]].

PROOF. The Corollary follows from the first part of the Fundamental Theorem if we can show how to construct a uniformly computable candidate sequence $p_0, p_1, \ldots$ of growth at least $\ell g \ell(x)$ with the property $\forall i[p_i \geq F(p_{i+1})$ a.e.]. We do this using a technique from [MF72] (see also [Yo73]).

Define a function ψ by

$$\psi(\ell,i,x) = \begin{cases} 0, \text{ if } \exists n < i[S_\ell(0,n) \geq x], \\ \max \{\psi(\ell,j,y), F(\lambda x \max(\phi_\ell(i+1,x), S_\ell(i+1,x)))(x): \\ \qquad\qquad x \geq j > i, \ x \geq y\}, \text{ otherwise.} \end{cases}$$

(As usual we think of undefined values as being infinite and use the obvious conventions about inequalities and maxima involving such values. Technically ϕ_i and S_i are in P_1, so our use of two arguments, e.g. $S_\ell(0,n)$, is to be regarded as an abbreviation for $S_\ell(pair(0,n))$ where $pair(x,y)$ is the integer which codes the binary representations of x and y concatenated with a separating symbol.) ψ is a partial recursive function. Therefore, by the Recursion Theorem (cf.[Ro67]), an ℓ_0 with $\phi_{\ell_0} = \lambda i,x \psi(\ell_0,i,x)$ can be found effectively. Define $p_i = \lambda x \max(\phi_{\ell_0}(i,x), S_{\ell_0}(i,x))$.

To prove that $p_0, p_1, \ldots$ is a candidate sequence with the desired properties consider the functions r_i defined by $r_i(x) = \phi_{\ell_0}(i,x) = \psi(\ell_0,i,x)$.

From the definition of ψ we have

$$(*) \qquad r_i(x) = \begin{cases} 0, \text{ if } \exists n < i[S_{\ell_0}(0,n) \geq x], \\ \max\{r_j(y), F(\lambda x \max(r_{i+1}(x), S_{\ell_0}(i+1,x)))(x): \\ \qquad\qquad x \geq j > i, x \geq y\}, \text{ otherwise.} \end{cases}$$

We first show that each r_i and therefore each p_i is total. Assume that r_0 is not total, i.e. assume that $r_0(n_0)$ is undefined for some n_0. Then all r_i with $i > n_0$ are defined by the first clause of (*) and are therefore total. But then r_{n_0} is also total because both clauses of (*) yield finite values for r_{n_0}. By induction r_0 is total. This implies that each r_i, including r_0 itself, is defined by the second clause of (*) on all but finitely many arguments. Hence, for r_0 to be total, $r_j(y)$ has to be defined for all j and y. Hence all the r_i are total.

Clearly $p_0, p_1, \ldots$ is uniformly computable, and each p_i is space-constructible by definition because $\max(\phi_j, S_j)$ is obviously space-constructible for any j.

Without loss of generality we may assume that $F(t) \geq \max(2 \cdot t, \ell g \ell)$ for all $t \in R_1$. Then both the fact that $p_0, p_1, \ldots$ is of growth at least $\ell g \ell$ and Properties (4) and (5) of space-candidate sequences follow from $\forall i[p_i \geq F(p_{i+1})$ a.e.], which is therefore all that is left to prove.

As pointed out above, each r_i is defined by the second clause of (*) on all but finitely many arguments. Hence $\forall i[r_i \geq F(\lambda x \max(r_{i+1}(x), S_{\ell_0}(x))$ a.e.]. By the definition of p_i, this yields $\forall i[p_i \geq r_i \geq F(p_{i+1})$ a.e.]. □

The next and final Corollary observes that restricting the computational tasks under consideration from computing arbitrary total recursive functions to deciding membership in recursive sets, i.e. to computing 0-1 valued total recursive functions, does not result in any loss of possible complexity behaviors.

COROLLARY 3. *For any total recursive function* f *that is of space complexity at least* $\ell g \ell$ *there is a* 0-1 *valued total recursive function* g *of the same complexity as* f, *i.e. a function* g *such that every complexity sequence for* f *is also a complexity sequence for* f *and vice versa. Moreover, a program for such an* g *can be found effectively from a program for* f.

PROOF. By Part 2 of the Fundamental Theorem there is a complexity sequence $p_0, p_1, \ldots$ for f which is a space-candidate sequence; by Part 1 there is a 0-1 valued total recursive function g which has $p_0, p_1, \ldots$ as a complexity sequence; and, by the remark after Lemma 1, if two functions share one complexity sequence then they share all their complexity sequences. □

We now illustrate the power of the Fundamental Theorem with two examples.

To get an example of a function with a rather "pathological" complexity behavior, define

$$t_i(x) = \begin{cases} 2^x, & \text{if } x \equiv 0 \bmod 2^i; \\ \ell g \ell(x), & \text{otherwise.} \end{cases}$$

$t_0(x) = 2^x$ on all arguments, but $t_i(x) = 2^x$ only on every (2^i)th argument x and is equal to $\ell g \ell(x)$ in between; t_{i+1} is obtained from t_i by "erasing" every other of these exponential "spikes." Informally, the first part of the Fundamental Theorem (applied with $p_i = \lceil t_i/(i+1) \rceil$) says that there is a total recursive function f which can be computed in logarithmic space (in the length of the input) just about everywhere, but that every program for f has infinitely many exponential "bursts" in complexity despite the fact that we can "thin out" those bursts as much as we like. The function f is an example of a function with "i.o.-speed-up" [5], that is, speed-up on infinitely many (but not necessarily almost all) arguments, as opposed to the "a.e.-speed-up" treated in Corollary 1.

In contrast to such pathological complexity properties, the Fundamental Theorem also yields examples of functions with "well-behaved" complexities. Applying the first part of the Fundamental Theorem with $p_i = \lceil |x|^2/(i+1) \rceil$ shows that there are total recursive functions whose tape complexity is exactly quadratic in the length of the input (up to the ever-present possibility of linear tape reduction).

3. PROOF OF THE FUNDAMENTAL THEOREM

Before we give a proof of the Fundamental Theorem we establish a lemma (Lemma 4) which allows us to assume that all candidate sequences are of an especially "nice" type. Lemmas 2 and 3 only serve to prove Lemma 4; their proofs are straightforward from the definitions and we omit them.

LEMMA 2. *Let* $p_0,p_1,\ldots$ *be a candidate sequence and let* $s \in R_1$ *be positive. Then* $r_0,r_1,\ldots$ *with* $r_i = \lceil \min\{s(j)\cdot p_j : j \le i\}/(i+1) \rceil$ *is a candidate sequence and* $p_0,p_1,\ldots \equiv r_0,r_1,\ldots$.

LEMMA 3. *Let* $p_0,p_1,\ldots$, $q_0,q_1,\ldots$ *and* $r_0,r_1,\ldots$ *be three sequences of total functions,* $p_0,p_1,\ldots \equiv q_0,q_1,\ldots$ *and* $p_i \le r_i \le q_i$ *a.e. for each* i. *Then* $p_0,p_1,\ldots \equiv r_0,r_1,\ldots \equiv q_0,q_1,\ldots$.

LEMMA 4. *Let* $q_0,q_1,\ldots$ *be a space-candidate sequence of at least* $\ell g \ell$ *growth. Then there is a sequence* $p_0,p_1,\ldots$ *of total functions with the following four properties*:

a. $q_0,q_1,\ldots \equiv p_0,p_1,\ldots$;
b. $p_0,p_1,\ldots$ *is decreasing*;
c. $\forall i,i'[i \le i'$ *implies* $(i'+1)\cdot p_{i'} \le (i+1)\cdot p_i + i']$; *and*
d. $\exists k\ \forall i,x[\phi_k(i,x) = p_i(x) \wedge S_k(i,x) \le (i+1)\cdot p_i(x) \wedge S_k(i+1,x) \le S_k(i,x)]$;
moreover, such a k *can be found effectively from a program for* $\lambda i,x[q_i(x)]$.

PROOF. Define $s(i) = \min\{j: \phi_j = S_j = q_i\}$. Such programs s(i) exist by Property (3) of candidate sequences. Consider the sequence $r_0,r_1,\ldots$ computed by the following program e with inputs i and x:

Program e:

STEP 1. Mark off $\ell g \ell(x)$ tape squares. Find the smallest j such that $\exists y\, \neg[\phi_j(y) = S_j(y) = q_i(y)]$ cannot be verified within the marked-off space. Write down this smallest j on a worktape. [6] Call it j_0.

STEP 2. Compute $\phi_{j_0}(x)$ or $q_i(x)$ (using the universal function for $q_0,q_1,\ldots$ to compute $q_i(x)$), whichever can be done in less space, writing the result on a worktape. Call the result s.

STEP 3. Output the larger of $s, \lceil \lg i \rceil$, and the space used so far by this program; halt.

End of program e.

We define the desired sequence $p_0,p_1,\ldots$ by

$$p_i = \lceil \min\{r_j : j \le i\}/(i+1) \rceil .$$

Proof of Property a. For every i it follows from the fact that q_i is total that the number j_0 computed in Step 1 of program e is ultimately, i.e. for all large enough x, equal to s(i). Hence for all such large enough x the value s computed in Step 2 of program e is $q_i(x)$, possibly computed as $\phi_{s(i)}(x)$ (or, more precisely, as $\phi_u(s(i),x)$ where u is some universal machine) and certainly computed in space no more than $S_u(s(i),x)$, which for any straightforward choice of u is no more than $s(i)\cdot S_{s(i)}(x) = s(i)\cdot q_i(x)$. So the sequence $r_0,r_1,\ldots$ computed by the above program e satisfies

$$q_i \leq r_i \leq s(i)\cdot q_i \text{ a.e.}$$

for each i and hence

$$\lceil \min\{q_j\colon j \leq i\}/(i+1)\rceil \leq \lceil \min\{r_j\colon j \leq i\}/(i+1)\rceil \leq$$
$$\leq \lceil \min\{s(j)\cdot q_j\colon j \leq i\}\rceil \text{ a.e.}$$

for each i. Lemmas 2 and 3 imply $p_0,p_1,\ldots \equiv q_0,q_1,\ldots$.

Property b. is an immediate consequence of the definition of $p_0,p_1,\ldots$.

Proof of c. From the definition of p_i we get $\min\{r_j\colon j \leq i\} \leq (i+1)\cdot p_i \leq \min\{r_j\colon j \leq i\} + i$ for all i.
Combining this with the obvious fact that $\min\{r_j\colon j \leq i\}$ is non-increasing in i yields $(i'+1)\cdot p_{i'} \leq \min\{r_j\colon j \leq i'\} + i' \leq \min\{r_j\colon j \leq i\} + i' \leq (i+1)\cdot p_i + i'$.

Proof of d. The following program d with inputs i and x computes $\min\{r_j(x)\colon j \leq i\}$.

Program d:

STEP 1. Set s to 1.

STEP 2. Use s tape squares to see if $\{j\colon j \leq \min(i,2^s) \wedge r_j(x) \leq s\} = \emptyset$. If so, then increase s by 1 and repeat Step 2; otherwise output s and halt.

End of program d.

It is easy to see that this program d computes $\min\{r_j(x)\colon j \leq i\}$ in those instances where the output is an s with $i \leq 2^s$. If the output is an

s with $i > 2^s$ then $s = \min\{r_j(x): j \le 2^s\}$ and since $r_j(x) \ge \lceil \log_2 j \rceil$ for all j, which is evident from the $\lceil \lg i \rceil$-term in Step 3 of program e, we have $s \le \min\{r_j(x): 2^s < j \le i\}$ and therefore again $s = \min\{r_j(x): j \le i\}$. Note that both programs e and d are (or can easily made to be) "honest," i.e. the number of tape squares they use is no more than their output value. Using this program d for $\min\{r_j: j \le i\}$, a program k for $p_i = \lceil \min\{r_j: j \le i\}/(i+1) \rceil$ with the properties given in part d, can be written in a straightforward way: Use program d to compute $z = \min\{r_j: j \le i\}$ in space z, then compute $\lceil z/(i+1) \rceil$, which can be done in no additional space. □

With Lemma 4 completed, both parts of the Fundamental Theorem can now be proven using standard techniques, the first part by a standard speed-up construction (cf. [Bℓ67,EB75,Ly75,MF72]) and the second part by exhibiting a straightforward computation of a complexity sequence for a given total recursive function f.

Proof of the Fundamental Theorem

1. We show how to construct a program for a total recursive 0-1 value function f from a given program for a candidate sequence $p_0, p_1, \ldots$ such that $p_0, p_1, \ldots$ is a complexity sequence for f. Applying Lemma 1, we may assume that $p_0, p_1, \ldots$ has properties b., c., and d. listed in Lemma 4.

Consider the following program t with input x:

Program t:

STEP 1. As far as possible within $\ell g \ell(x)$ tape squares execute this program recursively on inputs 0,1,2,3,... to find and write down as many elements as possible of the set $C = \{j: j \text{ gets } \textit{cancelled} \text{ by this program on some input}\}$.

STEP 2. Let $A = \{j: j \le \ell g \ell(x) \wedge S_j(x) < p_j(x) \wedge j \notin C\}$. If A is empty then output 0 and halt; otherwise output $1 \dot{-} \phi_{\min A}(x)$ and consider min A as *cancelled*. Halt.

End of program t.

Define $f = \phi_t$. f clearly is a 0-1 valued total recursive function.

We first show that $p_0, p_1, \ldots$ satisfies Property (1) of complexity sequences for f, i.e. we show that f can be computed in space p_i a.e. for each i. We achieve this by modifying, for each i, the program t so that the

resulting program t_i uses no more than p_i space. Notice that the only place where program t substantially exceeds the space bound p_i is in Step 2 when predicates $S_j(x) < p_j(x)$ are checked for $j < i$. (Remember that $p_0, p_1, \ldots$ is assumed to have properties b and c from Lemma 4.)

Let A(x) denote the set A defined in Step 2 on input x. For any j, $j \in A(x)$ for only finitely many x. To see this we consider two cases: If j gets cancelled by program t on some input, then this fact will be detected and j will be put into the set C in Step 1 on all large enough inputs x and therefore j will not be put into A anymore; if, on the other hand, j never gets cancelled then it cannot show up in A anymore after all those indices which are smaller than j and which do get cancelled on some input are in C: if j did still show up in A it would then be the minimum and therefore be cancelled, contradicting our assumption.

Hence for each i the predicate $\lambda j,x[j \in A(x) \wedge j < i]$ can be represented as a finite table and hence be computed *without using any worktape space.* (These finite tables cannot be found effectively from the indices i, cf. [Bℓ71].) To compute f in space p_i we use program t with the modification that in order to determine whether or not $j \in A(x)$ for some $j < i$ we consult this finite table, using no worktape space at all. Call this modified program t_i. t_i uses no more than space $\ell g \ell(x)$ in Step 1 and $(i+1) \cdot p_i + 2 \cdot \ell g \ell(x)$ in Step 2. This bound for Step 2 is derived by the following three observations, keeping in mind that the set A need not be computed, only its minimum needs to be found:

a. $\ell g \ell(x)$ space is (more than) enough to record values of $j \leq \ell g \ell(x)$ in the control of Step 2.

b. By Lemma 4, d, there is a program k which computes $p_j(x)$ in space no more than $(i+1) \cdot p_i(x)$ for all $j \geq i$.

c. With straightforward simulation techniques the predicate $S_j(x) < p_j(x)$ can be decided within space $j \cdot p_j(x)$ for all j. Using Lemma 4, c, we get

$$j \cdot p_j(x) \leq (j+1) \cdot p_j(x) \leq (i+1) \cdot p_i(x) + j \leq (i+1) \cdot p_i(x) + \ell g \ell(x)$$

for all $i \leq j \leq \ell g \ell(x)$.

Thus, altogether t_i uses no more than $c_i \cdot p_i(x)$ for some constant c_i since p_i is of growth at least $\ell g \ell$. Applying linear tape reduction and the fact that p_i is space-constructible shows that there is a program that computes f in space exactly p_i.

To prove that $p_0,p_1,\ldots$ satisfies Property (2) of complexity sequences for f we prove the statement that if $S_i < p_i$ i.o., then i gets *cancelled* by program t. Simply choose x such that $S_i(x) < p_i(x)$ and x is large enough that all $j < i$ which ever get *cancelled* by t are in C when Step 1 is performed with input x. Then i will be *cancelled* at step 2 of program t on input x, unless of course it already has been *cancelled* and is in C.

Finally, if i gets *cancelled* by program t on input x, then $\phi_i(x)$ is defined and $f(x) = 1 \dot{-} \phi_i(x) \neq \phi_i(x)$.

Thus $\phi_i = f$ implies that i never gets *cancelled*, which implies $S_i \geq p_i$ a.e. .

This finishes the proof of the first part of the Fundamental Theorem.

2. Given a program k for f, a complexity sequence $r_0,r_1,\ldots$ for f is computed by the following program h with inputs i and x:

Program h:

If $\phi_i \neq f$ can be detected within $\ell g \ell(x)$ tape squares, then output $S_k(x)$; otherwise output $\min(S_k(x),S_i(x))$.

End of program h.

Define $r_i = \lambda x \phi_h(i,x)$.

Since S_k is total, each r_i is total. We finish the proof by verifying that $r_0,r_1,\ldots$ is a complexity sequence for f.

Proof of Property (1). If $\phi_i = f$ then f can obviously be computed in space $r_i = \min(S_k,S_i)$ by running ϕ_k and ϕ_i in parallel; if $\phi_i \neq f$ then we distinguish between two cases: either there is an x with $\phi_i(x) \neq f(x)$ and both values defined, or there is no such x. In the first case, $\phi_i \neq f$ will be detected in $\ell g \ell(x)$ space for all large enough x, and from then on $r_i = S_k$, and again f can be computed in space r_i by patching program k with a finite table. In the second case, $r_i = \min(S_k,S_i)$ and f can be computed in that much space again by running ϕ_k and ϕ_i in parallel (which computes f because there is no x with $\phi_i(x)$ *defined* and $\phi_i(x) \neq f(x)$).

Proof of Property (2). If $\phi_j = f$ then $S_j \geq \min(S_k,S_j) = r_j$. $r_0,r_1,\ldots$ is a uniformly computable space-complexity sequence for f and hence, by the remarks following Definition 2, also a space-candidate sequence. Define $p_i = \min\{r_j : j \leq i\}$. Then $p_0,p_1,\ldots$ is a decreasing space-candidate sequence and equivalent to $r_0,r_1,\ldots$. □

The definitions of complexity and candidate sequences can be generalized to partial recursive functions by adding the requirement that the domains of all the p_i's are contained in the domain of f (cf. [Bℓ75,Ly75,Le73,Le74, Be76]).

Footnotes

1) Research sponsored by NSF Grants MCS77-19754 and MCS77-19754A02.

2) There is a long history of research resulting in the formulation and successive refinements of such a Fundamental Theorem. Early work was done by Blum [Bℓ67], Hartmanis and Stearns [HS65], and Rabin [Ra59,60]. Meyer and Fischer [MF72] show that the complexity of any recursive function can be described by a recursive sequence of "honest" functions. This observation is attributed independently to Kolmogorov by Levin [Le73,Le74] and is also implicit in Blum's results on "pseudo-speedup" [Bℓ71]. The results in the case of partial, as opposed to total, functions are considered in detail in [B 75,Be76]. Meyer and Fischer [MF72] also provide a partial converse to their observation mentioned above. This converse is strengthened by Schnorr and Stumpf [SS75], see also [Ly75]. Versions of the Fundamental Theorem for Turing machine space are given by Lynch [Ly75] and Schnorr [SS75], and for random-access machine time and space in [SS75]. A version of the Fundamental Theorem like the one given in the present paper has been proven previously by Levin [Le73,Le74].

3) "a.e." stands for "almost everywhere," meaning "on all but finitely many arguments."

4) $\lceil a \rceil$ denotes the smallest integer n with $n \geq a$. Similarly, $\lfloor a \rfloor$ denotes the largest integer n with $n \leq a$.

5) "i.o." stands for "infinitely often."

6) We assume that this minimal j can be written down in space $\ell g \ell(x)$. Otherwise we could write down $\min(j_0, \ell g \ell(x))$. This does not affect the proof at all.

REFERENCES

[Be76] BENNISON, V., *On the computational complexity of recursively enumerable sets,* PhD Thesis, University of Chicago (August 1976).

[BS78] BENNISON, V. & R.I. SOARE, *Some lowness properties and computational complexity sequences,* Theoretical Computer Science 6,3 (1978), 233-254.

[Bl67] BLUM, M., *A machine-independent theory of the complexity of recursive functions*, Journal of the ACM, 14, 2 (April 1967), 322-336.

[Bl71] BLUM, M., *On effective procedures for speeding up algorithms*, Journal of the ACM, 18, 2 (April 1971), 290-305.

[Bl75] BLUM, M., *On defining the complexity of partial recursive functions*, unpublished preprint (1975).

[Bo72] BORODIN, A., *Computational complexity and the existence of complexity gaps*, Journal of the ACM, 19, 1 (January 1972), 158-174.

[BCH69] BORODIN, A., R. CONSTABLE & J. HOPCROFT, *Dense and non-dense families of complexity classes*, IEEE Conf. Rec. on Switch. and Aut. Th. (1969), 7-19.

[EB75] VAN EMDE BOAS, P., *Ten years of speed-up*, Math. Foundations of Comp. Sci. 1975, Springer Lect. Notes in Comp. Sci., vol. 32, 13-29.

[HS65] HARTMANIS, J. & R.E. STEARNS, *On the computational complexity of algorithms*, Trans. AMS 117 (1965), 285-306.

[HU69] HOPCROFT, J.E. & J.D. ULLMAN, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading: 1969.

[Le73] LEVIN, L.A., *On storage capacity for algorithms*, Soviet Math. Dokl. 14, (1973), 1464-1466.

[Le74] LEVIN, L.A., *Complexity of Computation of Computable Functions*, in: Complexity of Computations and Algorithms, Kozmidiadi, Maslov, and Petri (eds). "Mir". Moscow (1974).

[Ly75] LYNCH, N., *Helping: Several formalizations*, Journal of Symbolic Logic, 40, 4 (December 1975), 555-566.

[MC71] MEYER, A.R. & E.M. MCCREIGHT, *Computationally complex and pseudo-random zero-one valued functions*, in: Theory of Machines and Computations, Z. Kohavi and A. Paz, eds. Academic Press, New York (1971), 19-42.

[MF72] MEYER, A.R. & P.C. FISCHER, *Computational speed-up by effective operators*, Journal of Symbolic Logic, 37, 1 (March 1972), 55-68.

[Ra59] RABIN, M.O., *Speed of computation and classification of recursive sets*, Third Conv. Scient. Societies Israel (1959), 1-2.

[Ra60] RABIN, M.O., *Degree of difficulty of computing a function and a partial ordering of recursive sets*, Tech. Rep. 2, Dept. of Math., Hebrew University, Jerusalem (1960).

[Ro67] ROGERS, H., *Theory of Recursive Functions and Effective Computability*, McGraw-Hill. New York: 1967.

[SS75] SCHNORR, C.P. & G. STUMPF, *A characterization of complexity sequences*, Zeitschrift für math. Logik und Grundlagen der Mathematik, 21 (1975), 47-56.

[Tr67] TRACHTENBROT, B.A., *Complexity of algorithms and computations*, (in Russian), Novosibirsk Univ. (1967).

[Tr70] TRACHTENBROT, B.A., *On autoreducibility*, Soviet Math. Dokl. 11,3 (1970), 814-817.

[Yo73] YOUNG, P.R., *Easy constructions in complexity theory: gap and speed-up theorems*, Proc. AMS. 37,2 (Feb. 1973), 555-563.

OTHER TITLES IN THE SERIES MATHEMATICAL CENTRE TRACTS

A leaflet containing an order-form and abstracts of all publications mentioned below is available at the Mathematisch Centrum, Tweede Boerhaavestraat 49, Amsterdam-1005, The Netherlands. Orders should be sent to the same address.

MCT 1 T. VAN DER WALT, *Fixed and almost fixed points*, 1963. ISBN 90 6196 002 9.

MCT 2 A.R. BLOEMENA, *Sampling from a graph*, 1964. ISBN 90 6196 003 7.

MCT 3 G. DE LEVE, *Generalized Markovian decision processes, part I: Model and method*, 1964. ISBN 90 6196 004 5.

MCT 4 G. DE LEVE, *Generalized Markovian decision processes, part II: Probabilistic background*, 1964. ISBN 90 6196 005 3.

MCT 5 G. DE LEVE, H.C. TIJMS & P.J. WEEDA, *Generalized Markovian decision processes, Applications*, 1970. ISBN 90 6196 051 7.

MCT 6 M.A. MAURICE, *Compact ordered spaces*, 1964. ISBN 90 6196 006 1.

MCT 7 W.R. VAN ZWET, *Convex transformations of random variables*, 1964. ISBN 90 6196 007 X.

MCT 8 J.A. ZONNEVELD, *Automatic numerical integration*, 1964. ISBN 90 6196 008 8.

MCT 9 P.C. BAAYEN, *Universal morphisms*, 1964. ISBN 90 6196 009 6.

MCT 10 E.M. DE JAGER, *Applications of distributions in mathematical physics*, 1964. ISBN 90 6196 010 X.

MCT 11 A.B. PAALMAN-DE MIRANDA, *Topological semigroups*, 1964. ISBN 90 6196 011 8.

MCT 12 J.A.TH.M. VAN BERCKEL, H. BRANDT CORSTIUS, R.J. MOKKEN & A. VAN WIJNGAARDEN, *Formal properties of newspaper Dutch*, 1965. ISBN 90 6196 013 4.

MCT 13 H.A. LAUWERIER, *Asymptotic expansions*, 1966, out of print; replaced by MCT 54 and 67.

MCT 14 H.A. LAUWERIER, *Calculus of variations in mathematical physics*, 1966. ISBN 90 6196 020 7.

MCT 15 R. DOORNBOS, *Slippage tests*, 1966. ISBN 90 6196 021 5.

MCT 16 J.W. DE BAKKER, *Formal definition of programming languages with an application to the definition of ALGOL 60*, 1967. ISBN 90 6196 022 3.

MCT 17 R.P. VAN DE RIET, *Formula manipulation in ALGOL 60, part 1*, 1968. ISBN 90 6196 025 8.

MCT 18 R.P. VAN DE RIET, *Formula manipulation in ALGOL 60, part 2*, 1968. ISBN 90 6196 038 X.

MCT 19 J. VAN DER SLOT, *Some properties related to compactness*, 1968. ISBN 90 6196 026 6.

MCT 20 P.J. VAN DER HOUWEN, *Finite difference methods for solving partial differential equations*, 1968. ISBN 90 6196 027 4.

MCT 21 E. WATTEL, *The compactness operator in set theory and topology*, 1968. ISBN 90 6196 028 2.

MCT 22 T.J. DEKKER, *ALGOL 60 procedures in numerical algebra, part 1*, 1968. ISBN 90 6196 029 0.

MCT 23 T.J. DEKKER & W. HOFFMANN, *ALGOL 60 procedures in numerical algebra, part 2*, 1968. ISBN 90 6196 030 4.

MCT 24 J.W. DE BAKKER, *Recursive procedures*, 1971. ISBN 90 6196 060 6.

MCT 25 E.R. PAERL, *Representations of the Lorentz group and projective geometry*, 1969. ISBN 90 6196 039 8.

MCT 26 EUROPEAN MEETING 1968, *Selected statistical papers, part I*, 1968. ISBN 90 6196 031 2.

MCT 27 EUROPEAN MEETING 1968, *Selected statistical papers, part II*, 1969. ISBN 90 6196 040 1.

MCT 28 J. OOSTERHOFF, *Combination of one-sided statistical tests*, 1969. ISBN 90 6196 041 X.

MCT 29 J. VERHOEFF, *Error detecting decimal codes*, 1969. ISBN 90 6196 042 8.

MCT 30 H. BRANDT CORSTIUS, *Excercises in computational linguistics*, 1970. ISBN 90 6196 052 5.

MCT 31 W. MOLENAAR, *Approximations to the Poisson, binomial and hypergeometric distribution functions*, 1970. ISBN 90 6196 053 3.

MCT 32 L. DE HAAN, *On regular variation and its application to the weak convergence of sample extremes*, 1970. ISBN 90 6196 054 1.

MCT 33 F.W. STEUTEL, *Preservation of infinite divisibility under mixing and related topics*, 1970. ISBN 90 6196 061 4.

MCT 34 I. JUHÁSZ, A. VERBEEK & N.S. KROONENBERG, *Cardinal functions in topology*, 1971. ISBN 90 6196 062 2.

MCT 35 M.H. VAN EMDEN, *An analysis of complexity*, 1971. ISBN 90 6196 063 0.

MCT 36 J. GRASMAN, *On the birth of boundary layers*, 1971. ISBN 90 6196 064 9.

MCT 37 J.W. DE BAKKER, G.A. BLAAUW, A.J.W. DUIJVESTIJN, E.W. DIJKSTRA, P.J. VAN DER HOUWEN, G.A.M. KAMSTEEG-KEMPER, F.E.J. KRUSEMAN ARETZ, W.L. VAN DER POEL, J.P. SCHAAP-KRUSEMAN, M.V. WILKES & G. ZOUTENDIJK, *MC-25 Informatica Symposium*, 1971. ISBN 90 6196 065 7.

MCT 38 W.A. VERLOREN VAN THEMAAT, *Automatic analysis of Dutch compound words*, 1971. ISBN 90 6196 073 8.

MCT 39 H. BAVINCK, *Jacobi series and approximation*, 1972. ISBN 90 6196 074 6.

MCT 40 H.C. TIJMS, *Analysis of (s,S) inventory models*, 1972. ISBN 90 6196 075 4.

MCT 41 A. VERBEEK, *Superextensions of topological spaces*, 1972. ISBN 90 6196 076 2.

MCT 42 W. VERVAAT, *Success epochs in Bernoulli trials (with applications in number theory)*, 1972. ISBN 90 6196 077 0.

MCT 43 F.H. RUYMGAART, *Asymptotic theory of rank tests for independence*, 1973. ISBN 90 6196 081 9.

MCT 44 H. BART, *Meromorphic operator valued functions*, 1973. ISBN 90 6196 082 7.

MCT 45 A.A. BALKEMA, *Monotone transformations and limit laws*, 1973. ISBN 90 6196 083 5.

MCT 46 R.P. VAN DE RIET, *ABC ALGOL, A portable language for formula manipulation systems, part 1: The language*, 1973. ISBN 90 6196 084 3.

MCT 47 R.P. VAN DE RIET, *ABC ALGOL, A portable language for formula manipulation systems, part 2: The compiler*, 1973. ISBN 90 6196 085 1.

MCT 48 F.E.J. KRUSEMAN ARETZ, P.J.W. TEN HAGEN & H.L. OUDSHOORN, *An ALGOL 60 compiler in ALGOL 60, Text of the MC-compiler for the EL-X8*, 1973. ISBN 90 6196 086 X.

MCT 49 H. KOK, *Connected orderable spaces*, 1974. ISBN 90 6196 088 6.

MCT 50 A. VAN WIJNGAARDEN, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER, M. SINTZOFF, C.H. LINDSEY, L.G.L.T. MEERTENS & R.G. FISKER (Eds), *Revised report on the algorithmic language ALGOL 68*, 1976. ISBN 90 6196 089 4.

MCT 51 A. HORDIJK, *Dynamic programming and Markov potential theory*, 1974. ISBN 90 6196 095 9.

MCT 52 P.C. BAAYEN (ed.), *Topological structures*, 1974. ISBN 90 6196 096 7.

MCT 53 M.J. FABER, *Metrizability in generalized ordered spaces*, 1974. ISBN 90 6196 097 5.

MCT 54 H.A. LAUWERIER, *Asymptotic analysis, part 1*, 1974. ISBN 90 6196 098 3.

MCT 55 M. HALL JR. & J.H. VAN LINT (Eds), *Combinatorics, part 1: Theory of designs, finite geometry and coding theory*, 1974. ISBN 90 6196 099 1.

MCT 56 M. HALL JR. & J.H. VAN LINT (Eds), *Combinatorics, part 2: graph theory, foundations, partitions and combinatorial geometry*, 1974. ISBN 90 6196 100 9.

MCT 57 M. HALL JR. & J.H. VAN LINT (Eds), *Combinatorics, part 3: Combinatorial group theory*, 1974. ISBN 90 6196 101 7.

MCT 58 W. ALBERS, *Asymptotic expansions and the deficiency concept in statistics*, 1975. ISBN 90 6196 102 5.

MCT 59 J.L. MIJNHEER, *Sample path properties of stable processes*, 1975. ISBN 90 6196 107 6.

MCT 60 F. GÖBEL, *Queueing models involving buffers*, 1975. ISBN 90 6196 108 4.

* MCT 61 P. VAN EMDE BOAS, *Abstract resource-bound classes, part 1*. ISBN 90 6196 109 2.

* MCT 62 P. VAN EMDE BOAS, *Abstract resource-bound classes, part 2*. ISBN 90 6196 110 6.

MCT 63 J.W. DE BAKKER (ed.), *Foundations of computer science*, 1975. ISBN 90 6196 111 4.

MCT 64 W.J. DE SCHIPPER, *Symmetric closed categories*, 1975. ISBN 90 6196 112 2.

MCT 65 J. DE VRIES, *Topological transformation groups 1 A categorical approach*, 1975. ISBN 90 6196 113 0.

MCT 66 H.G.J. PIJLS, *Locally convex algebras in spectral theory and eigenfunction expansions*, 1976. ISBN 90 6196 114 9.

* MCT 67 H.A. LAUWERIER, *Asymptotic analysis, part 2.* ISBN 90 6196 119 X.

MCT 68 P.P.N. DE GROEN, *Singularly perturbed differential operators of second order*, 1976. ISBN 90 6196 120 3.

MCT 69 J.K. LENSTRA, *Sequencing by enumerative methods*, 1977. ISBN 90 6196 125 4.

MCT 70 W.P. DE ROEVER JR., *Recursive program schemes: semantics and proof theory*, 1976. ISBN 90 6196 127 0.

MCT 71 J.A.E.E. VAN NUNEN, *Contracting Markov decision processes*, 1976. ISBN 90 6196 129 7.

MCT 72 J.K.M. JANSEN, *Simple periodic and nonperiodic Lamé functions and their applications in the theory of conical waveguides*, 1977. ISBN 90 6196 130 0.

MCT 73 D.M.R. LEIVANT, *Absoluteness of intuitionistic logic*, 1979. ISBN 90 6196 122 x.

MCT 74 H.J.J. TE RIELE, *A theoretical and computational study of generalized aliquot sequences*, 1976. ISBN 90 6196 131 9.

MCT 75 A.E. BROUWER, *Treelike spaces and related connected topological spaces*, 1977. ISBN 90 6196 132 7.

MCT 76 M. REM, *Associons and the closure statement*, 1976. ISBN 90 6196 135 1.

MCT 77 W.C.M. KALLENBERG, *Asymptotic optimality of likelihood ratio tests in exponential families*, 1977 ISBN 90 6196 134 3.

MCT 78 E. DE JONGE, A.C.M. VAN ROOIJ, *Introduction to Riesz spaces*, 1977. ISBN 90 6196 133 5.

MCT 79 M.C.A. VAN ZUIJLEN, *Empirical distributions and rankstatistics*, 1977. ISBN 90 6196 145 9.

MCT 80 P.W. HEMKER, *A numerical study of stiff two-point boundary problems*, 1977. ISBN 90 6196 146 7.

MCT 81 K.R. APT & J.W. DE BAKKER (eds), *Foundations of computer science II*, part I, 1976. ISBN 90 6196 140 8.

MCT 82 K.R. APT & J.W. DE BAKKER (eds), *Foundations of computer science II*, part II, 1976. ISBN 90 6196 141 6.

MCT 83 L.S. VAN BENTEM JUTTING, *Checking Landau's "Grundlagen" in the AUTOMATH system*, 1979 ISBN 90 6196 147 5.

MCT 84 H.L.L. BUSARD, *The translation of the elements of Euclid from the Arabic into Latin by Hermann of Carinthia (?) books vii-xii*, 1977. ISBN 90 6196 148 3.

MCT 85 J. VAN MILL, *Supercompactness and Wallman spaces*, 1977. ISBN 90 6196 151 3.

MCT 86 S.G. VAN DER MEULEN & M. VELDHORST, *Torrix I*, 1978. ISBN 90 6196 152 1.

* MCT 87 S.G. VAN DER MEULEN & M. VELDHORST, *Torrix II*, ISBN 90 6196 153 x.

MCT 88 A. SCHRIJVER, *Matroids and linking systems*, 1977. ISBN 90 6196 154 8.

MCT 89 J.W. DE ROEVER, *Complex Fourier transformation and analytic functionals with unbounded carriers*, 1978. ISBN 90 6196 155 6.

* MCT 90 L.P.J. GROENEWEGEN, *Characterization of optimal strategies in dynamic games*, . ISBN 90 6196 156 4.

* MCT 91 J.M. GEYSEL, *Transcendence in fields of positive characteristic*, . ISBN 90 6196 157 2.

MCT 92 P.J. WEEDA, *Finite generalized Markov programming*, 1979. ISBN 90 6196 158 0.

MCT 93 H.C. TIJMS (ed.) & J. WESSELS (ed.), *Markov decision theory*, 1977. ISBN 90 6196 160 2.

MCT 94 A. BIJLSMA, *Simultaneous approximations in transcendental number theory*, 1978 . ISBN 90 6196 162 9.

MCT 95 K.M. VAN HEE, *Bayesian control of Markov chains*, 1978. ISBN 90 6196 163 7.

* MCT 96 P.M.B. VITÁNYI, *Lindenmayer systems: structure, languages, and growth functions*, . ISBN 90 6196 164 5.

* MCT 97 A. FEDERGRUEN, *Markovian control problems; functional equations and algorithms*, . ISBN 90 6196 165 3.

MCT 98 R. GEEL, *Singular perturbations of hyperbolic type*, 1978. ISBN 90 6196 166 1

MCT 99 J.K. LENSTRA, A.H.G. RINNOOY KAN & P. VAN EMDE BOAS, *Interfaces between computer science and operations research*, 1978. ISBN 90 6196 170 X.

MCT 100 P.C. BAAYEN, D. VAN DULST & J. OOSTERHOFF (Eds), *Proceedings bicentennial congress of the Wiskundig Genootschap, part 1*, 1979. ISBN 90 6196 168 8.

MCT 101 P.C. BAAYEN, D. VAN DULST & J. OOSTERHOFF (Eds), *Proceedings bicentennial congress of the Wiskundig Genootschap, part 2*, 1979. ISBN 90 9196 169 6.

MCT 102 D. VAN DULST, *Reflexive and superreflexive Banach spaces*, 1978. ISBN 90 6196 171 8.

MCT 103 K. VAN HARN, *Classifying infinitely divisible distributions by functional equations*, 1978 . ISBN 90 6196 172 6.

MCT 104 J.M. VAN WOUWE, *Go-spaces and generalizations of metrizability*, 1979. ISBN 90 6196 173 4.

* MCT 105 R. HELMERS, *Edgeworth expansions for linear combinations of order statistics*, . ISBN 90 6196 174 2.

MCT 106 A. SCHRIJVER (Ed.), *Packing and covering in combinatories*, 1979. ISBN 90 6196 180 7.

MCT 107 C. DEN HEIJER, *The numerical solution of nonlinear operator equations by imbedding methods*, 1979. ISBN 90 6196 175 0.

* MCT 108 J.W. DE BAKKER & J. VAN LEEUWEN (Eds), *Foundations of computer science III*, part I, . ISBN 90 6196 176 9.

* MCT 109 J.W. DE BAKKER & J. VAN LEEUWEN (Eds), *Foundations of computer science III*, part II . ISBN 90 6196 177 7.

MCT 110 J.C. VAN VLIET, *ALGOL 68 transput*, part I,1979 . ISBN 90 6196 178 5.

MCT 111 J.C. VAN VLIET, *ALGOL 68 transput*, part II: *An implementation model*, 1979. ISBN 90 6196 179 3.

AN ASTERISK BEFORE THE NUMBER MEANS "TO APPEAR"