Even Simple Programs Are Hard To Analyze

NEIL D. JONES AND STEVEN S. MUCHNICK

The University of Kansas, Lawrence, Kansas

ABSTRACT. A simple programming language which corresponds in computational power to the class of generalized sequential machines with final states is defined. It is shown that a variety of questions of practical programming interest about the language are of nondeterministic linear space complexity Extensions to the language are defined (adding arithmetic and array data structures) and their complexity properties are explored. It is concluded that questions about halting, equivalence, optimization, and so on are intractable even for very simple programming languages

KEY WORDS AND PHRASES. low-level complexity, reducibility, programming languages, computational complexity

CR CATEGORIES· 5 23, 5 24, 5 25

1. *Introduction*

It has long been known that most questions of interest about the behavior of programs written in ordinary programming languages are recursively undecidable. These questions include whether a program will halt, whether two programs are equivalent, whether one is an optimized form of another, and so on. On the other hand, it is possible to make some or all of these questions decidable by suitably restricting the computational ability of the programming language under consideration. One way to do this is to abstract out the basic operations and so consider classes of schemata rather than programs [1, 9]. Another way is to restrict the statements which can be used to structure programs. The LOOP language of Meyer and Ritchie [6], for example, has a decidable halting problem, but undecidable equivalence. A third way is to restrict the range of data values and the operations which can act upon them. This approach is illustrated by finite automata and generalized sequential machines (or finite state transducers) for which virtually all of these questions are decidable (except that Griffiths [2] has shown equivalence undecidable for nondeterministic generalized sequential machines).

A natural question to ask is how hard it is to solve these problems for programming languages for which they are decidable, and it is with this area that we are concerned in this paper. In particular we describe a programming language modeled on current higher level languages which has exactly the computational power of deterministic finite state transducers with final states, and we analyze the space and time required to decide various questions of programming interest about the language. We find that questions about halting, equivalence, and optimization are already intractable for this very simple language. We also study extensions to the language such as simple arithmetic capabilities, arrays, and nondeterminism, some of which extend the capabilities of the language and/or increase the complexity of its decidable problems. In a related future paper we

A preliminary version of this work was presented at the Second ACM SIGACT-SIGPLAN Conference on Principles of Programming Languages, Palo Alto, Calif , Jan 1975.

The work of the second author was partially supported by University of Kansas General Research Grant 3758-5038.

Authors' address· Department of Computer Science, The University of Kansas, Lawrence, KS 66045

shall consider the effect of adding recursive subroutines with various parameter access mechanisms to the language.

2. *Finite Memory Programs: Definitions*

Let Σ be a fixed finite alphabet and \$ a symbol not in Σ (the *endmarker*). Let $s \geq 3$ denote the number of symbols in $\Sigma \cup \{\$\}$. Then a *finite memory program* (or FMP—pronounced "fump") P is a finite sequence of labeled instructions $1: I_1;\ 2: I_2;\ \ldots\ ;\ k: I_k$ such that

(1) each I_i is of one of the following forms:

read X_j	**if** $V_1 = V_2$ **goto** l
write V	**accept**
$X_j \leftarrow V$	**halt**

where $j \in N$, each V or V_i denotes either an element of $\Sigma \cup \{\$\}$ or a variable name X_j, and $1 \leq l \leq k$;

(2) I_k = **halt** or **accept**;

(3) there is an m such that each of $X_1, \ldots, X_m$ occurs in P and no X_j occurs in P for $j > m$.

We denote the length of the program by len(P), counting one for each symbol in $\Sigma \cup \{\$, \leftarrow, =, X\}$, one for each word (**read**, **if**, etc.), and the lengths of the binary representations of the subscripts which characterize the variable names and the statement numbers occurring in **goto** statements. Numbers which actually label statements are unnecessary and hence are neither counted in len(P) nor included in the representation of the FMPs used as input to Turing machines. While program length will be a major concern in this paper, linear changes in length will be of little or no significance.

A *configuration* of the FMP P is a triple $\alpha = \langle x\$, i, \mathbf{a}_m\rangle$ such that $x \in \Sigma^*$, $1 \leq i \leq k$, and $\mathbf{a}_m \in (\Sigma \cup \{\$\})^m$. (We use $\mathbf{a}_m$ to denote the vector $\langle a_1, a_2, \ldots, a_m\rangle$.) In what follows we use the symbol V to denote either a variable name or a symbol in $\Sigma \cup \{\$\}$ and so define the *content function* con(V, α) by

$$\operatorname{con}(V, \alpha) = \begin{cases} V \text{ if } V \in \Sigma \cup \{\$\}, \\ a_j \text{ if } V = X_j. \end{cases}$$

Let $\alpha = \langle x\$, i, \mathbf{a}_m\rangle$ and $\beta = \langle y\$, j, \mathbf{b}_m\rangle$ be configurations. Then α *yields* β *immediately*, denoted $\alpha \vdash \beta$, iff one of the following holds:

(1) I_i = **read** X_l for some l, $j = i + 1$, $b_k = a_k$ for $k \neq l$, and either $b_l \in \Sigma$ and $x\$ = b_l y\$$ or $x = y = \epsilon$ and $b_l = \$$;

(2) I_i = **write** V, $j = i + 1$, $y = x$, and $\mathbf{b}_m = \mathbf{a}_m$;

(3) $I_i = X_l \leftarrow V$, $j = i + 1$, $y = x$, $b_l = \operatorname{con}(V, \alpha)$, and $b_k = a_k$ for $k \neq l$;

(4) I_i = **if** $V_1 = V_2$ **goto** l, $y = x$, $b_k = a_k$ for all k, and either $j = l$ and $\operatorname{con}(V_1, \alpha) = \operatorname{con}(V_2, \alpha)$ or $j = i + 1$ and $\operatorname{con}(V_1, \alpha) \neq \operatorname{con}(V_2, \alpha)$.

The relation $\alpha \vdash \beta$ is false if I_i = **halt** or **accept**. As usual, we denote the reflexive transitive closure of $\vdash$ by $\vdash^*$ and define P *accepts* (*halts* for) $x \in \Sigma^*$ iff there exist $y \in \Sigma^*$, $\mathbf{a}_m \in (\Sigma \cup \{\$\})^m$, and $1 \leq i \leq k$, where k is the number of instructions in program P, such that

$$\langle x\$, 1, \$, \ldots, \$\rangle \vdash^* \langle y\$, i, \mathbf{a}_m\rangle$$

and I_i is **accept** (**halt**). Further, $\langle y\$, i, \mathbf{a}_m\rangle$ is called an *accepting* (*halting*) *configuration*. The *language* accepted by P is $L(P) = \{x \in \Sigma^* \mid P \text{ accepts } x\}$.

As noted above, a finite memory program is not simply an acceptor, but has output as well. The output is given by the unique (partial) function $P: \Sigma^* \rightarrow \Sigma^*$ which satisfies the condition that for $x \in \Sigma^*$, if $\alpha_1 \vdash \alpha_2 \vdash \cdots \vdash \alpha_p$, $\alpha_1 = \langle x\$, 1, \$, \ldots, \$\rangle$, and α_p is an accepting configuration, then $P(x) = b_1 b_2 \cdots b_p$, where for each i, if $\alpha_i = \langle y\$, j, \mathbf{a}_m\rangle$, then

$b_\iota = \text{con}(V, \alpha)$ if $I_\iota =$ **write** V and $\text{con}(V, \alpha) \in \Sigma$, and $b_\iota = \epsilon$ otherwise. If $x \notin L(P)$, then $P(x)$ is undefined. P_1 and P_2 are *equivalent* (written $P_1 \equiv P_2$) if $P_1(x)$ and $P_2(x)$ are either both defined and equal or both undefined for all $x \in \Sigma^*$. We also say P_1 and P_2 *have identical input-output behavior* for $P_1 \equiv P_2$.

The above describes the strict formal syntax of finite memory programs and their semantics. We shall not, however, adhere to the strict syntax in writing programs. We use a variety of informal constructs which may all be easily (i.e. in log space) translated into the strict syntax with a cost of not more than two additional instructions and no additional variables for each. These constructs include mnemonic labels and variable names, statements without labels, statement brackets (**begin**, **end**), **goto** l, **if** V_1 op V_2 **then** S_1, and **if** V_1 op V_2 **then** S_1 **else** S_2 (where S_1, S_2 represent statements and op is either "=" or "≠").

3. *Basic Lemmas*

The computational power of the class of FMPs is characterized as follows:

LEMMA 1. *If P is any* FMP, *then there is a deterministic generalized sequential machine* (DGSM) *M with final states* [3] *and input and output alphabet $\Sigma \cup \{\$\}$ such that P and M have identical input-output behavior* (*i.e.* $\forall x \in \Sigma^*$, $P(x) = M(x\$)$); *and conversely.*

PROOF. That a FMP can simulate a DGSM should be clear; so we concentrate on the other direction.

Note that we can modify a FMP so that it will never enter an infinite loop, but will halt instead (this is proved in case (3) of Lemma 2 below). The set of states of the DGSM can be

$$K = R \times (\Sigma \cup \{\$\})^m \cup \{q_A, q_H\},$$

where R is the set of labels of the **read** statements in the FMP, m is the number of variables, and q_A and q_H are distinct accepting and nonaccepting states. The initial state is $\langle \iota, a_1, \ldots, a_m \rangle$ where ι is the label of the first encountered **read** statement, and $a_1, \ldots, a_m$ are the values of the variables immediately before execution of that statement. The set of final states is $F = \{q_A\}$, and q_H is a looping state. Since the states record all of the current configurations of the FMP (exept the input), it follows that the transition function $\delta : K \times (\Sigma \cup \{\$\}) \to K \times \Sigma^*$ can be defined in a totally finite manner. □

A variety of modifications in the syntax of the language and the construction of programs can be made without altering the computational capabilities of the language. Some of these are summarized in Lemma 2. Note that, unlike the informal syntax extensions, some of these may have drastic effects on the size of programs and hence on the complexity of analyzing them.

LEMMA 2. *For any* FMP *P there is a* FMP *P' equivalent to P and satisfying any or all of the following conditions:*

(1) *P' has exactly one* **halt** *and exactly one* **accept** *instruction;*

(2) *P' halts or accepts only after reading the entire input;*

(3) *P' halts or accepts for every input* (*i.e. it never loops*) *and* $len(P') = O(len(P) \log len(P))$;

(4) *P' has no inaccessible instructions;*

(5) *P' has no assignment statements.*

PROOF. For (1), simply append to P the instructions

$k + 1$: **halt**; $k + 2$: **accept**

and change all other **halt** instructions to **if** \$ = \$ **goto** $k + 1$ and all other **accept** instructions to **if** \$ = \$ **goto** $k + 2$.

(2) Suppose P halts before reading the entire input. Then replace all **halt** instructions by

```
    l· read X_1,
l + 1· if X_1 ≠ $ goto l;
l + 2. halt
```

and relabel as necessary. A similar modification clearly works for accepting.

(3) The maximum number of instructions that can be executed between **read** instructions without P looping is bounded by $k \cdot s^m$. Let c be the least positive integer such that $2^c \geq k \cdot s^m$ and let $a \in \Sigma$. Add variables $Y_0, \ldots, Y_{c-1}$ to M to use as a binary counter, R to contain the last character read from the input, and $Z_1, \ldots, Z_k$ to guide the flow of control through M ($Z_i \neq \$$ iff I_i is about to be or is being executed). Perform the following steps on the program:

(a) Insert "$R \leftarrow a$;" at the beginning of the program.

(b) Replace each non-**read** instruction I_j by

```
    Z_j ← a;
    goto l,
l_j I_j
```

(c) Replace each **read** X_i instruction by

```
    Z_j ← a,
    goto l';
l_j X_i ← R
```

(d) Add the following to the end of P:

```
l': if R = $ then goto l,
    Y_0 ← $,    , Y_{c-1} ← $,
    read R,
    goto l'',
l   if Y_0 = $ then Y_0 ← a else
      begin Y_0 ← $,
        if Y_1 = $ then Y_1 ← a else
          begin

            Y_{c-2} ← $,
            if Y_{c-1} = $ then Y_{c-1} ← a

                        else halt
          end
      end,
  l'' if Z_1 = a then begin Z_1 ← $, goto l_1 end
else .
.
elseif Z_k = a then begin Z_k ← $, goto l_k end
elsehalt,
```

Note that

$$\begin{aligned} \text{len}(P') &\leq O(\text{len}(P) \cdot \log k) + O(c \log c + \log k) + O(c \log c + c \log k) + O(k \log k) \\ &\leq O(\text{len}(P) \cdot \log \text{len}(P)). \end{aligned}$$

(4) This follows by first constructing a DGSM as in Lemma 1, then removing the inaccessible states and translating the resulting DGSM back to a FMP. However, len(P') may be large in comparison to len(P).

(5) This also follows directly from Lemma 1 □

Although FMPs are not powerful enough to simulate all nondeterministic GSMs, there is an easy construction to prove the following lemma:

LEMMA 3. *Any nondeterministic finite automaton with n states and l transitions is equivalent to a* FMP *with not more than $2n + 1$ variables and length of order $l \cdot \log n$.*

PROOF. We use one variable Z to receive the input characters to the finite automaton. The other variables are partitioned into two sets $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_n$. The X_i are used to record the set of states Q in which the nondeterministic finite automaton is

currently active ($X_i = \$$ if state $q_i \notin Q$ and $X_i = a \in \Sigma$ if state $q_i \in Q$). The Y_i are used to construct the set of states $\delta(Q, b)$ as each symbol b is read and then the Y_i are copied into the X_i. □

As a corollary to Lemma 2, part (3), we have the following, which indicates one of the differences between FMPs and nondeterministic finite automata:

COROLLARY 4. *For any* FMP P *there is a* FMP P' *such that* $L(P') = \Sigma^* - L(P)$ *and* $len(P') = O(len(P) \cdot log\ len(P))$.

4. *Problems of Concern*

Among the questions about FMPs which we believe to be of interest are those expressed by membership in the following sets:

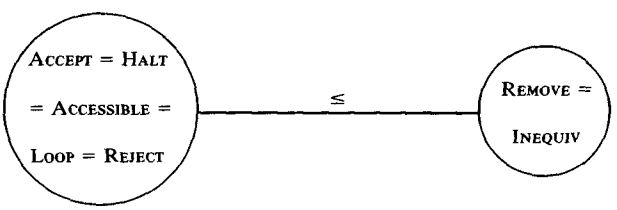
(1) ACCEPT = $\{P \in \text{FMP} \mid \exists x \in \Sigma^*$ such that P accepts $x\} = \{P \in \text{FMP} \mid L(P) \neq \phi\}$.
(2) HALT = $\{P \in \text{FMP} \mid \exists x \in \Sigma^*$ such that P *halts* (but does not accept) on input $x\}$.
(3) LOOP = $\{P \in \text{FMP} \mid \exists x \in \Sigma^*$ such that P enters an infinite loop on input $x\}$.
(4) REJECT = HALT ∪ LOOP = $\{P \in \text{FMP} \mid L(P) \neq \Sigma^*\}$.
(5) ACCESSIBLE = $\{\langle P, i\rangle \in \text{FMP} \times N \mid \exists x \in \Sigma^*$ such that P executes its ith instruction at least once on input $x\}$.
(6) INEQUIV = $\{\langle P, i\rangle \in \text{FMP}^2 \mid \exists x\ P_1(x) \neq P_2(x)\}$.
(7) REMOVE = $\{\langle P, i\rangle \in \text{FMP} \times N \mid \exists x\ P(x) \neq P^i(x)\}$ where P^i is the program obtained by replacing the ith instruction of P by $X_1 \leftarrow X_1$.

Questions about optimization other than those expressed by ACCESSIBLE and REMOVE are difficult to formulate without some measure of what it means for one FMP to be better or more efficient than another. Such measures might include program length, number of variables, execution time, and so on, but are difficult enough to pin down that for the present we limit our concern to consideration of the above questions.

Many of these sets are of essentially the same complexity in the following strong sense. Define $S \leq T$ to hold iff there is a function $f(\ )$ such that (1) for any x, $x \in S$ iff $f(x) \in T$; (2) for some constants c and d and all x, $|f(x)| \leq c \cdot |x| \cdot (\log |x|)^d$; and (3) $f(x)$ is log-space computable, where "$f(\ )$ is log-space computable" means that there is an offline deterministic Turing machine with an output tape which operates in space $\log(\ )$ and which, when given x on its input tape, writes $f(x)$ on its output tape and then halts [4].

Let $S = T$ mean $S \leq T$ and $T \leq S$. Then the sets are related as shown in the following theorem:

THEOREM 5. *The reducibility relationships among the problems are given by the following diagram*:



PROOF. Most of the reducibilities are easy to establish. We show that ACCEPT ≤ HALT ≤ REJECT ≤ LOOP ≤ ACCESSIBLE ≤ ACCEPT ≤ INEQUIV ≤ REMOVE ≤ INEQUIV.

(1) ACCEPT ≤ HALT: $f(P) = P'$ is identical to P except that all **accept** and **halt** instructions have been reversed.

(2) HALT ≤ REJECT: $f(P) = P'$ where P' is the result of applying the procedure given in the proof of Lemma 2, part (3), to P, except that all the **halt** instructions generated by the procedure are changed to **accept**s.

(3) REJECT ≤ LOOP: $f(P) = P'$, which is identical to P except that each instruction I_i = **halt** in P is replaced by "**goto** i" in P'.

(4) LOOP ≤ ACCESSIBLE: $f(P) = \langle P', l\rangle$ where P' is the result of applying the procedure

given in the proof of Lemma 2, part (3), to P, except that (a) all generated **halt** instructions are replaced by **goto** l and (b) the statement l: **goto** l is appended to the end of the program.

(5) ACCESSIBLE $\leq$ ACCEPT: $f(P, i) = P'$ where P' is P with all **accept** instructions changed to **halt** and the ith instruction changed to **accept**.

(6) ACCEPT $\leq$ INEQUIV: $f(P) = \langle P, P' \rangle$ where P' is identical to P except that all **accept** instructions are replaced by **halts**.

(7) INEQUIV $\leq$ REMOVE: Let $f(P_1, P_2) = \langle P, 1 \rangle$ where P is the program

goto l
P_1
l. P_2

with the appropriate renumbering to avoid statement number conflicts. Clearly $P \equiv P_2$ and $P^1 \equiv P_1$; so $\langle P_1, P_2 \rangle \in$ INEQUIV iff $\langle P, 1 \rangle \in$ REMOVE.

(8) REMOVE $\leq$ INEQUIV: $f(P, i) = \langle P, P^i \rangle$; so $\langle P, i \rangle \in$ REMOVE iff $P \not\equiv P^i$. □

It follows from Theorem 5 that we can restrict our attention to ACCEPT and INEQUIV in our consideration of the complexities of these problems. Note that a single reduction such as INEQUIV $\leq$ ACCEPT would render all of these problems equivalent. We conjecture that this is in fact true, but a reduction may be difficult to obtain in view of the difficulties encountered in the proof of Theorem 9.

5. *Complexity of the Basic Model*

The method embodied in the following definition and lemma is used to establish all our lower bounds for space and is due to Stockmeyer and Meyer [8]. Two varieties of Turing machines are used (see [3] for details), the one-tape Turing machine with a single two-way read-write tape, and the offline Turing machine with a read-only input tape with endmarkers and a separate read-write work tape. We abbreviate nondeterministic Turing machine by TM and deterministic Turing machine by DTM.

We say that a TM (or DTM) *operates in space* $S(\ )$ where $S: N \to N$ iff when given an input $x \in \Sigma^*$ it writes on not more than $S(|x|)$ tape cells and then halts. It *operates in time* $T(\ )$ for $T: N \to N$ iff when given an input $x \in \Sigma^*$ it performs at most $T(|x|)$ transitions and then halts. We shall consider only bounding functions satisfying $S(n) \geq n$ and $T(n) \geq n$ for one-tape TMS and $S(n) \geq \log n$ for offline TMS.

The class DSPACE(S) is defined to be the class of all subsets of Σ^* accepted by offline DTMS which operate in space $S(\ )$, and NSPACE(S) is defined similarly for offline (nondeterministic) TMS. DTIME(T) and NTIME(T) are defined analogously for time bounds on one-tape Turing machines. Note that for $S(n) \geq n$ there is no difference in recognition capability between one-tape and offline TMS operating in space $S(\ )$.

Definition. Let Z be a one-tape TM which operates in space $S(n) \geq n$. The *set of accepting computations* $\text{Comp}_{x,S}$ is

$$\text{Comp}_{x,S} = \{\#\alpha_1\#\alpha_2\# \cdots \#\alpha_m\# | m \geq 1,\ \alpha_1 \in q_0 x B^*,\ \alpha_m \in \Gamma^* q_f \Gamma^* \text{ and, for } i = 1, 2, \ldots, m-1,\ \alpha_i \vdash \alpha_{i+1} \text{ and, for each } i,\ |\alpha_i| = S(|x|) + 1\}.$$

It is assumed here that Σ and Γ are the input and work tape alphabets of Z, respectively, K is its state set, $B \in \Gamma - \Sigma$ is a "blank" symbol, and $q_0, q_f \in K$ are the initial and accepting states of Z, respectively. The α_i represent instantaneous descriptions (i.d.'s) of Z, and $\alpha \vdash \beta$ iff α and β are i.d.'s and α yields β in one step by the transition rules of Z.

LEMMA 6. *Let Z be a one-tape* TM *which operates in space $S(n) = n$. There is a function $f_Z: \Sigma^* \to$* FMP *such that (a) Z accepts $x \in \Sigma^*$ iff $L(f_Z(x)) \neq \phi$, (b) $|f_Z(x)| \leq c|x|\log|x|$ for some c and all $x \in \Sigma^*$, and (c) $f_Z(\ )$ is log-space computable.*

PROOF. Clearly Z accepts x iff $\text{Comp}_{x,S} \neq \phi$ since Z operates in space $S(\ )$. We show how to construct a FMP $g(x)$ with alphabet $\Gamma \cup K \cup \{\#\}$ such that $L(g(x)) = \text{Comp}_{x,S}$ and $g(\ )$ satisfies (a), (b), and (c). To obtain $f_Z(\ )$ as required, construct an invertible homomorphism $h: (\Gamma \cup K \cup \{\#\})^* \to \Sigma^*$ and construct FMP $f_Z(x)$ so that $L(f_Z(x)) =$

$h(L(g(x)))$. This can be done quite simply and so that $\text{len}(f_Z(x))$ is linear in $\text{len}(g(x))$, so that $f_Z(\ )$ will satisfy (a), (b), and (c) as required. It remains to show how to construct $g(\ )$.

Let $x = a_1 \cdots a_n$ be fixed, with each $a_i \in \Sigma$. Define the relation[1] $R \subseteq \Sigma^6$ by $R(b_1, b_2, b_3, b_4, b_5, b_6)$ iff there are two i.d.'s α and β such that (a) $\alpha \vdash \beta$ and $|\alpha| = |\beta|$, (b) b_1, b_2, b_3 occupy positions $i - 1$, i, and $i + 1$, respectively, in $\#\alpha\#$, and (c) b_4, b_5, b_6 occupy positions $i - 1$, i, and $i + 1$, respectively, in $\#\beta\#$ (where $1 \leq i \leq |\alpha|$).

Clearly $\alpha_1 \vdash \alpha_2 \vdash \cdots \vdash \alpha_m$ will hold iff $R(e_{i-1}, e_i, e_{i+1}, e_{i+n+1}, e_{i+n+2}, e_{i+n+3})$ is true for every position i in the string $\#\alpha_1\#\alpha_2\# \cdots \#\alpha_m\# = e_1 e_2 \cdots e_{m(n+2)+1}$, where $1 \leq i \leq (m - 1)(n + 2)$. Thus membership in $\text{Comp}_{x,S}$ may be determined by moving a window (i.e. a memory of $n + 5$ cells) along $e_1 e_2 \cdots e_{m(n+2)}$ to determine whether R holds at each position.

The following FMP is $g(x)$ and operates in the manner just described. We present it in an Algol-like notation to enhance readability, but the translation to a strict FMP is easy. We assume without loss of generality that $q_0 \neq q_f$.

```
begin symbol F, E1, E2,    , En+5,
  F ← $;
  read E1, read E2, ... , read En+5,
  if E1 ··· En+5 ≠ #α1#b1b2 then halt,
  while R(E1, E2, E3, En+3, En+4, En+5) do
    begin
      if En+4 = qf then F ← qf,
      if En+5 = # and F = qf then accept,
      E1 ← E2, E2 ← E3, . , En+4 ← En+5;
      read En+5
    end,
  halt
end
```

where b_1, b_2 are the first two symbols of α_2. This FMP clearly accepts $\text{Comp}_{x,S}$ and is log-space constructable from x. Further, the program uses $n + 6$ variables, each referred to at most a constant number of times in the strict FMP form of the program. Thus the total program length required for references to variable names is at most $c(n + 6)\log(n + 6) = O(n \log n)$, and so the total program length is also $O(n \log n)$.

Consequently (a), (b), and (c) are all satisfied by $g(\ )$, and hence by $f_Z(\ )$. □

The problem $P \in$ ACCEPT can be decided nondeterministically in the amount of space necessary to represent and manipulate a configuration without input $\alpha = \langle \iota, \mathbf{a}_m \rangle$ as follows:

THEOREM 7. ACCEPT $\in NSPACE(n)$.

Proof. For a FMP P we can decide $L(P) \neq \phi$ with a nondeterministic Turing machine Z by guessing an input string $x \in \Sigma^*$ one symbol at a time and simulating the behavior of P on input x. The control flow of P can be simulated by positioning the read head of Z to the instruction in P currently being executed. Thus only a configuration without input $\alpha = \langle \iota, \mathbf{a}_m \rangle$ need be stored on the tape along with P.

Clearly for some constant c and any program P with m variables and k instructions we have

$$\max(k, cm\lceil \log m \rceil) \leq \text{len}(P),$$

and consequently

$$|\alpha| \leq \lceil \log k \rceil + m \leq \lceil \log \text{len}(P) \rceil + \text{len}(P) \leq 2\,\text{len}(P).$$

Thus ACCEPT $\in$ NSPACE(n). □

From Lemma 6 we can obtain a lower bound for ACCEPT.

THEOREM 8. ACCEPT $\notin NSPACE(n^{1-\epsilon})$ *for any* $\epsilon > 0$.

[1] We use the symbol $\subseteq$ to indicate set inclusion and $\subset$ to indicate proper inclusion, i.e. $A \subset B$ iff $A \subseteq B$ and $A \neq B$.

PROOF. Let Z be an arbitrary Turing machine which operates in space $S(n) = n$. By Lemma 6, the question "Does Z accept x?" can be answered by computing $f_Z(x)$ and then deciding whether $L(f_Z(x)) \neq \phi$, i.e. whether $f_Z(x) \in$ ACCEPT.

Now suppose ACCEPT is recognized by a Turing machine Z^a which operates in space $n^{1-\epsilon}$ for some ϵ such that $0 < \epsilon \leq 1$. We show that this implies that $L(Z)$ can be recognized within a specific smaller space bound, contradicting a result of Seiferas, Fischer, and Meyer [7].

By Lemma 6, $f_Z(\ )$ is log-space computable. However, $f_Z(x)$ may exceed x in length, so we cannot store all of $f_Z(x)$ on tape. Instead we construct a TM Z' which applies Z^a to a simulated input tape containing $f_Z(x)$, only a single symbol of which is stored at any given moment. Z' will have an input tape containing x and a work tape composed of three tracks: track 1, the simulated work tape of Z^a on input $f_Z(x)$; track 2, the position i of Z^a's input head in $f_Z(x)$; and track 3, the work tape of the TM which computes $f_Z(\ )$. Z' operates in a three-step cycle:

(a) compute a_i, the ith symbol of $f_Z(x)$, using the TM whose existence is guaranteed by Lemma 6 and using track 3 as a work tape;

(b) simulate one step of Z^a on input a_i using track 1 as a work tape;

(c) update i (on track 2) to the new position of Z^a on its simulated input tape.

The number of tape cells used by Z' is bounded by

$$\max((\text{len}(f_Z(x)))^{1-\epsilon}, \log \text{len}(f_Z(x)), \log |x|) \leq \max((cn \log n)^{1-\epsilon}, \log(cn \log n), \log n) = O((cn \log n)^{1-\epsilon}),$$

so that any set recognizable in space n is also recognizable in space $(n \log n)^{1-\epsilon}$. However,

$$\lim_{n\to\infty} n/(n \log n)^{1-\epsilon} = \infty \quad \text{for } \epsilon > 0,$$

so, by Corollary 4 of [7], there are sets in NSPACE(n) − NSPACE($(n \log n)^{1-\epsilon}$), contradicting our result above. Thus we must have ACCEPT $\notin$ NSPACE($n^{1-\epsilon}$) for any $\epsilon > 0$. □

Clearly Theorems 7 and 8 apply to HALT, ACCESSIBLE, LOOP, and REJECT as well as ACCEPT.

Note that the set $\{\langle P, x\rangle | P \text{ accepts } x\}$ is in DSPACE(n) since we need not guess the x accepted by P as in the case of ACCEPT. It may also be shown that this set is not in DSPACE($n^{1-\epsilon}$) for any $\epsilon > 0$ by the following technique. Given a DTM Z which operates in space $S(n) = n$ and an input $x \in \Sigma^*$, construct a FMP P such that P accepts iff Z accepts x and $\text{len}(P) = O(|x| \log |x|)$. This is naturally done by assigning a memory cell of P for each work tape cell of Z and simulating Z step by step. Clearly $\langle P, \epsilon\rangle$ is in the given set iff Z accepts x (where ϵ denotes the empty string). A contradiction can then be obtained as was done in Theorem 8.

By a different technique we can show that the same bounds apply to INEQUIV and REMOVE, in itself a rather surprising result since inequivalence is harder to decide than acceptance for most classes of automata and languages. Our result is based on Theorem 12 of Jones, Lien, and Laaser [5], which states that inequivalence for DGSMs with final states is NSPACE($\log n$)-complete. We thus obtain the following theorem:

THEOREM 9. INEQUIV $\in$ *NSPACE*(n).

PROOF. For any FMP P, let M_p denote the naturally corresponding DGSM equivalent to P as constructed in Lemma 1. It is straightforward to build a Turing machine which, given $\langle P, i\rangle$ as input, will compute in linear space the ith symbol of a description of M_p.

Now given any two FMPS P and P', we can apply the Turing machine which decides inequivalence for DGSMs to M_p and M'_p. Clearly this can be done in linear space. □

As a corollary to Theorems 5 and 8, we have the following:

COROLLARY 10. INEQUIV $\notin$ *NSPACE*($n^{1-\epsilon}$) *for any* $\epsilon > 0$.

We might introduce an element of nondeterminism into FMPS by allowing the instruction

goto $l_1, l_2, \ldots, l_n$

whose execution would cause a nondeterministic selection of one of the statements labeled $l_1, l_2, \ldots, l_n$ to be the next statement executed. An input would then be accepted iff there were some sequence of choices which led to an accepting configuration. While not affecting the upper and lower space bounds for ACCEPT, ACCESSIBLE, and HALT, this change renders INEQUIV and REMOVE undecidable by Griffiths' result [2]. In addition the problem "Given $M \in$ FMP, is $L(M) \neq \Sigma^*$?" (i.e. the set REJECT) may be shown to require at least NSPACE($2^{cn^{1-\epsilon}}$) for some $c > 0$ and all $\epsilon > 0$, by use of Lemma 6.

Note that we began Section 2 by specifying that we were working over a fixed alphabet $\Sigma \cup \{\$\}$ of size s. If we were to allow the size of Σ to vary, this would affect our complexity results. We might then speak of the complexity of problems for machines of different word sizes, and would find that each of our upper bounds would have a multiplicative constant of $\log s$. As long as we stay within any particular word size (equals alphabet size) this is only a constant factor, but when we allow the size of Σ to vary arbitrarily we introduce a factor bounded by log(len(P)) into the upper bounds.

In summarizing this section, we see that all the questions of interest about deterministic FMPS require approximately nondeterministic linear space, even though the FMP is a particularly simple and restricted model of computation. We are thus led to conclude that the problems which are undecidable for general programming languages, though decidable in this limited context, are nevertheless extremely difficult for even such simple languages as we consider here. In essence they require the amount of space (and apparently time) necessary to do exhaustive search.

6. *Finite Memory Programs with Numbers*

In this section and Section 7 we discuss two natural extensions to finite memory programs which increase their ease of programming and/or computational ability and which make their decidable questions correspondingly harder. We shall be rather less formal in describing the extensions than in our presentation of the basic model in Section 2. We denote the extensions by descriptive superscripts on the name "FMP" and the problems about the extensions by applying the same superscripts to the names of the problems. Thus we have, for example, $\text{FMP}^{\text{ARRAY}}$ and $\text{ACCEPT}^{\text{ARRAY}}$ for FMPS with arrays.

FIXED WORD SIZE. The first extension we consider is the capacity to perform arithmetic. This is naturally done by interpreting the symbols in $\Sigma \cup \{\$\}$ as representing the integers $0, 1, \ldots, s-1$, with \$ representing 0. We then define a *finite memory program with numbers* (FMP^{NUM}) to be syntactically a FMP as before, with the possible addition of instructions of the forms $X \leftarrow Y$ op Z, where op is any of $+$, $-$, $*$, or $/$. Semantically these are interpreted in the usual way, except that all values are taken modulo s, and / denotes integer division (with truncation). In the case of an attempt to divide by zero we set the result to zero.

In spite of appearances, a FMP^{NUM} is not computationally more powerful than a GSM since its total number of configurations (without input) is still finite. In fact we have the following lemma:

LEMMA 11. *Any program P in FMP^{NUM} may be replaced by an equivalent program P' in* FMP; *further, $len(P') = O(len(P))$.*

PROOF. We first show how to replace each instruction of the form $X \leftarrow Y$ op Z by an equivalent sequence involving only $X \leftarrow X + 1$ and the usual FMP instructions. This is done in a way similar to the usual primitive recursive constructions of $+$, $*$, ... from successor. For example, $X \leftarrow Y + Z$ may be replaced by the sequence

$W \leftarrow 0$; $X \leftarrow Y$;
while $W \neq Z$ **do**
  $X \leftarrow X + 1$; $W \leftarrow W + 1$
end

Finally "$X \leftarrow X + 1$" may be replaced by:

```
if X = 0  then X ← 1
          else if X = 1 then X ← 2
                        else
                          else if X = s − 1 then X ← 0
```

□

Note that the number of instructions performed in executing P may be significantly less than in P'.

COROLLARY 12. ACCEPT^{NUM} *is in* $NSPACE(n)$ *and is not in* $NSPACE(n^{1-\epsilon})$ *for any* $\epsilon > 0$.

PROOF. Immediate from Lemma 11 and Theorems 7 and 8. □

VARIABLE WORD SIZE. The FMP version discussed above has the property that any value which is to be stored into a memory cell either must have originated in an input string or must occur as an explicity named constant in the program. This is of course not a restriction imposed by conventional programming languages; hence we introduce a new variant of the FMP in which the elements of Σ are interpreted arithmetically and which allows access to elements of Σ not explicitly named in the program.

Clearly altogether too much power would result from allowing Σ to be the natural numbers, since this would allow computation of arbitrary recursive functions. Hence we consider the problem of analyzing FMPs with a variable word size given in binary along with the program.

By definition a *finite memory program with variable word size* (FMP^{VWS}) is a pair $P_s = \langle P, \bar{s}\rangle$, where $\bar{s}$ is the binary representation of an integer s and P is to be described. We again interpret \$ as 0, and Σ as $\{1, 2, \ldots, s - 1\}$. P is a FMP as originally defined, with three exceptions:

(a) the elements of Σ appearing in a program are represented in binary notation, and their lengths are counted as part of len(P);

(b) P may contain instructions of the form $X \leftarrow X + 1$, with the addition performed modulo s,

(c) $\text{len}(P_s) = \max(\text{len}(P), |\bar{s}|)$, by definition.

Note that we could also allow instructions of the form "$X \leftarrow Y + Z$," etc., without increasing the computational abilities of a FMP^{VWS} since the same technique as in Lemma 11 could be used to reduce them to "$X \leftarrow X + 1$." However, this instruction cannot itself be eliminated as was done in Lemma 11. The reason is that s is variable; so each "$X \leftarrow X + 1$" would be replaced by $O(s) = O(2^{\text{len}(P_s)})$ instructions.

THEOREM 13. ACCEPT$^{VWS} \in NSPACE(n^2)$.

PROOF. The proof is as for Theorem 7, by simulating an arbitrary given FMP P on a nondeterministically guessed input string. The amount of space used is bounded by the size of a configuration:

$$\log k + m\lceil \log s\rceil = \log k + m|\bar{s}| \leq \text{len}(P_s) + \text{len}(P_s)\cdot\text{len}(P_s) = O(\text{len}^2(P_s)). \quad \square$$

THEOREM 14. ACCEPT^{VWS} *is not in* $NSPACE(n^{2-\epsilon})$ *for any* $\epsilon > 0$.

PROOF. The proof is very similar to that for ordinary FMPs (Theorem 8) but somewhat more complicated. The technique is to show that if Z is a one-tape TM which operates in space $S(n) = n^2$, then there is a function $f_Z: \Sigma^* \rightarrow$ FMP^{VWS} which satisfies conditions (a), (b), and (c) of Lemma 6. As in Lemma 6, we construct for each $x = a_1 \cdots a_n \in \Sigma^*$ a FMP^{VWS} $g(x)$ which accepts the set Comp_{x,n^2} and satisfies conditions (a), (b), and (c). We construct $g(x)$ so as to have $n + 1$ variables, each containing n symbols (except the last, which will contain five symbols) of the "window" used to scan an alleged accepting computation $\#\alpha_1\#\alpha_2\# \cdots \#\alpha_m\#$. The key to the construction is to let d be the least power of 2 such that $d \geq |\Gamma \cup K \cup \{\#, \$\}|$ and consider the n-symbol string $a_{n-1}\cdots a_0$ as representing the integer $a_0 + a_1 d + \cdots + a_{n-1}d^{n-1}$. The word size of the FMP^{VWS} will be $s = d^n$.

Thus $g(x)$ has $n + 1$ variables, each capable of holding a string of n symbols coded

arithmetically; consequently $g(x)$ has an effective memory of $n^2 + n$ symbols and so can store the entire "window" of an n^2-space-bounded TM Z.

We leave the detailed construction of $g(x)$ to the interested reader. It closely parallels that of Lemma 6, with the exception that the program variables denote strings of n symbols rather than single symbols. Thus it is necessary to perform "packing" and "unpacking" operations in order to test the R relation and to shift the window. These may be done arithmetically by multiplying, dividing, and taking remainders by d, as in Lemma 11.

The remainder of the proof is exactly parallel to that of Theorem 8; so we omit the details. The idea is to show that if ACCEPT^{VWS} could be accepted in NSPACE$(n^{2-\epsilon})$ for some $\epsilon > 0$, then any set in NSPACE(n^2) would be recognized in space $(n \log n)^{2-\epsilon}$, which is a contradiction, as in Theorem 8. □

7. *Finite Memory Programs with Arrays*

We next consider the extension of adding arrays to FMPs. In particular, the class of *finite memory programs with arrays* (or FMP^{ARRAY}) has an alphabet Σ, the endmarker \$, and $s = |\Sigma \cup \{\$\}|$, as before. In addition we have a fixed *subscript set* $\Theta \subseteq \Sigma \cup \{\$\}$ such that $|\Theta| \geq 2$.

A FMP^{ARRAY} P is then a program, as described in Section 2, except that (1) each occurrence of an X_i may now be followed by a subscript list of the form $[W_1, \ldots, W_l]$ where each W_j is either an unsubscripted variable X_p or a symbol from Θ; and (2) each V or V_i represents an X_i, an expression $X_i[W_1, \ldots, W_l]$, or a symbol from $\Sigma \cup \{\$\}$. The number of subscripts adhering to X_i must be the same for all occurrences of X_i throughout P. The semantics are a straightforward and obvious extension of the semantics of FMPs except for two points:

(1) In a configuration $\langle x\$, i, a_1, \ldots, a_m\rangle$, each a_i is a function $a_i: \Theta^l \to \Sigma \cup \{\$\}$, where l is the number of subscripts (possibly zero) adhering to the variable X_i.

(2) If in an expression $X_i[W_1, \ldots, W_l]$, one or more of the W_p has as its value a symbol not in Θ, then the entire expression evaluates to \$; if such an expression occurs on the left-hand side of an assignment, the statement has no effect.

By an argument similar to the one used for Theorem 7, we obtain the following theorem:

THEOREM 15. ACCEPT$^{ARRAY} \in$ *NSPACE*(2^{cn}) *for some* $c > 0$.

PROOF. Construct a Turing machine which, when given an arbitrary FMP^{ARRAY} P, simulates the behavior of P on a guessed input string x, and accept P just in case P accepts x. The storage required is the amount necessary to contain a configuration without input. Let $t = |\Theta| \geq 2$ and let $X_1, \ldots, X_m$ be the variables of P. Suppose X_i has l_i dimensions. Then the required space is given by

$$\begin{aligned} |\langle i, \mathbf{a}_m\rangle| &\leq \log \text{len}(P) + |a_1| + \cdots + |a_m| \\ &\leq \text{len}(P) + t^{l_1} + \cdots + t^{l_m} \\ &\leq \text{len}(P) + t^{l_1 + \cdots + l_m} \\ &\leq \text{len}(P) + t^{\text{len}(P)} \\ &= O(t^{\text{len}(P)}). \end{aligned}$$

□

A lower bound can be obtained by using techniques similar to those used in Theorem 8 and Lemma 6.

THEOREM 16. ACCEPT$^{ARRAY} \notin$ *NSPACE*$(2^{dn/\log n})$ *for some* $d > 0$.

PROOF. Let Z be a one-tape Turing machine which operates in space 2^n such that $L(Z) \notin$ NSPACE(1.5^n) (such a machine must exist by [7]). We show that there is a constant $b > 0$ such that, for any $d > 1$, ACCEPT$^{ARRAY} \in$ NSPACE$(d^{n/\log n})$ implies that $L(Z) \in$ NSPACE(d^{bn}). Hence if ACCEPT^{ARRAY} were in NSPACE$(d^{n/\log n})$ for all $d > 1$, then we could select d such that $d^b < 1.5$, implying that $L(Z) \in$ NSPACE$(d^{bn}) \subseteq$ NSPACE(1.5^n), a contradiction.

As in Lemma 6 we show that there is a function $f_Z: \Sigma^* \to$ FMP$^{\text{ARRAY}}$ which satisfies (a), (b), and (c) of that Lemma. This is done by constructing for each $x \in \Sigma^*$ a FMP$^{\text{ARRAY}}$ $g(x)$ with input alphabet $\Gamma \cup K \cup \{\$\}$ such that $L(g(x)) = \text{Comp}_{x,2^n}$ (where $x = a_1 \cdots a_n$ and each $a_i \in \Sigma$).

In order to present $g(\ )$ concisely, we introduce a notational device. Suppose that $t = |\Theta| = 2$, and identify the symbols in Θ with 0 (equals \$) and 1 (the general case $t > 2$ is entirely analogous). Then we use $\bar{I}$ to denote the $n + 3$-tuple of variables $I_1, \ldots, I_{n+3}$, so that $A[\bar{I}]$ denotes a single element of an $(n + 3)$-dimensional array which will be used to store the "window" used to scan an alleged member of $\text{Comp}_{x,2^n}$. The window in this instance will be composed of $2^n + 5$ symbols, and thus array A is sufficient to comprise the window since $2^n + 5 < 2^{n+3}$ for all $n \geq 0$. Further, we may treat $I_1 \cdots I_{n+3}$ as a binary number (I_1 is the most significant digit), and we use such notations as $\bar{I} \leftarrow \bar{0}$, $\bar{I} \leftarrow \bar{I} + 1$, and $\bar{I} \leftarrow \overline{2^n + 3}$ with their natural interpretations (modulo 2^{n+3}). It is easily seen that FMP$^{\text{ARRAY}}$ programs may be constructured to perform such statements and tests and that the constructions can be done in log space, while the resulting programs have length not exceeding $O(n \log n)$.

We now describe $g(\ )$. The scanning window is stored in an $(n + 3)$-dimensional array A which is used as a one-dimensional array of length 2^{n+3} as just described. Two pointers $\bar{L}$ and $\bar{U}$ into the array are maintained, one for each end of the window. The program is log-space constructable and is as follows:

begin
  comment read initial i.d and check its correctness,
  read $A[\bar{0}]$, **if** $A[\bar{0}] \neq$ # **then halt**,
  read $A[\bar{1}]$, **if** $A[\bar{1}] \neq q_0$ **then halt**,
  for $\bar{I} \leftarrow \bar{2}$ **to** $\overline{n + 1}$ **do**
    begin
      read $A[\bar{I}]$, **if** $A[\bar{I}] \neq a_{\bar{I}}$ **then halt**
    end,
  for $\bar{I} \leftarrow \overline{n + 2}$ **to** $\overline{2^n + 2}$ **do**
    begin
      read $A[\bar{I}]$, **if** $A[\bar{I}] \neq B$ **then halt**
    end,
  read $A[\overline{2^n + 3}]$, **if** $A[\overline{2^n + 3}] \neq$ # **then halt**,
  $\bar{L} \leftarrow \bar{0}$, $\bar{U} \leftarrow \overline{2^n + 4}$, **read** $A[\bar{U}]$,
  comment check the R relation across the string,
  while $R(A[\bar{L}], A[\overline{L + 1}], A[\overline{L + 2}], A[\overline{U - 2}], A[\overline{U - 1}], A[\bar{U}])$ **do**
    begin
      if $A[\overline{U - 1}] = q_f$ **then** $F \leftarrow q_f$,
      if $A[\bar{U}] =$ # **and** $F = q_f$ **then accept**;
      $\bar{L} \leftarrow \bar{L} + 1$,
      $\bar{U} \leftarrow \bar{U} + 1$,
      read $A[\bar{U}]$
    end,
  halt
end

It should be clear that for some b and all x,

$$|g(x)| \leq b\ |x|\ \log\ |x|$$

As in the proof of Theorem 8, if ACCEPT$^{\text{ARRAY}} \in$ NSPACE$(d^{n/\log n})$, it follows that $L(Z)$ is recognizable in space

$$d^{bn\ \log\ n/\log(bn\ \log\ n)} \leq d^{bn}.$$

Thus $L(Z) \in$ NSPACE(d^{bn}). But, as noted above, this cannot be the case for all $d > 1$; so ACCEPT$^{\text{ARRAY}} \in$ NSPACE$(d^{n/\log n})$ must be false for some > 1. □

Note that by varying the size of the subscript set or relaxing the requirement that $\Theta \subseteq \Sigma \cup \{\$\}$ we may radically alter the complexity bounds. The resulting nondeterministic

space bounds will, however, correspond to the number of cells addressable as a function of the size of the program.

8. *Summary of Results and Conclusions*

We have discussed a restricted model of computation based on current higher level languages and the computational difficulty of various solvable questions concerning it. We have then considered various natural extensions of the language and their contributions to increasing the complexity of these questions. We have provided strong evidence that in general the problems require essentially the amount of work needed to do an exhaustive search among the possible configurations of a program and thus are both extremely hard and unsusceptible of large improvement by the employment of clever techniques. The following table summarizes our results:

	Lower bound	Upper bound
ACCEPT	NSPACE($n^{1-\epsilon}$)	NSPACE(n)
ACCEPT^{NUM}	NSPACE($n^{1-\epsilon}$)	NSPACE(n)
ACCEPT^{VWS}	NSPACE($n^{2-\epsilon}$)	NSPACE($n^{2\epsilon}$)
ACCEPT^{ARRAY}	NSPACE($2^{dn/\log n}$)	NSPACE(2^{cn})

In a succeeding paper we shall discuss the extension of adding recursive subroutines to FMPS and the complexity bounds for their analysis.

We believe the kind of work reported here is novel and deserving of further attention. We suggest, in particular, the study of other extensions to the language, other problems of programming interest, definitions of complexity measures appropriate to the study of optimization questions, etc. A particular problem for which this appears to be a useful model is the study of time-space tradeoffs.

REFERENCES

1 CONSTABLE, R L , AND MUCHNICK, S S Subrecursive program schemata I, II. I Undecidable equivalence problems; II Decidable equivalence problems *J. Comptr System Sci 6*, 6 (Dec 1972), 480–537
2. GRIFFITHS, T V The unsolvability of the equivalence problem for Λ-free nondeterministic generalized machines *J ACM 15*, 3 (July 1968), 409–413
3 HOPCROFT, J E , AND ULLMAN, J D *Formal Languages and Their Relation to Automata* Addison-Wesley, Reading, Mass , 1969
4 JONES, N D Space-bounded reducibility among combinatorial problems *J Comptr. System Sci 11*, 1 (Aug. 1975), 68–85.
5 JONES, N D , LIEN, Y.E , AND LAASER, W T New problems complete for nondeterministic log space Comptr Sci. Tech Rep TR-75-1, U of Kansas, Lawrence, Kansas, April 1975
6. MEYER, A.R , AND RITCHIE, D M. Computational Complexity and Program Structure. Res Paper RC-1817, IBM T J Watson Res Ctr , Yorktown Heights, N Y , May 1967
7. SEIFERAS, J I , FISCHER, M J , AND MEYER, A R Refinements of the nondeterministic time and space hierarchies Conf. Rec 14th Ann IEEE Symp on Switching and Automata Theory, Oct 1973, pp. 130–137
8 STOCKMEYER, L J., AND MEYER, A R. Word problems requiring exponential time Preliminary report Proc Fifth Ann ACM Symp on Theory of Comptng , April–May 1973, pp. 1–9
9 YANOV, Y.I The logical schemes of algorithms *Problems Cybernet. (USSR) 1* (1960), 82–140 (English translation)

RECEIVED AUGUST 1975, REVISED APRIL 1976