EVEN SIMPLE PROGRAMS ARE HARD TO ANALYZE

by Neil D. Jones & Steven S. Muchnick*
Department of Computer Science
The University of Kansas
Lawrence, Kansas 66045

I. Introduction

It has long been known that most questions of interest about the behavior of programs 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. The Loop language of Meyer and Ritchie [MR], for example, has a decidable halting problem, but undecidable equivalence. Restricting the computational ability still further, virtually all of these questions are decidable for finite automata and generalized sequential machines (except that Griffiths [Gri] has shown equivalence undecidable for nondeterministic gsms).

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 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 recursive subroutines with both call-by-value and call-by-name parameter passing mechanisms, some of which extend the capabilities of the language and/or increase the complexity of its decidable problems. In one case, that of recursion with call-by-name, the previously decidable questions are seen to become undecidable.

II. Finite Memory Programs

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) P is a finite sequence of labeled instructions

$$1{:}I_1;\ \ 2{:}I_2;\ \ \ldots;\ \ k{:}I_k$$

such that

1) each I_j is of one of the following forms

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

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

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 number of variables in P by var(P) = m, and the number of instructions in P by prog(P) = k. We denote the length of the program by len(P), counting each word (*read*, *if*, $\leftarrow$, etc.) as a single symbol and representing the subscripts which characterize the variable names in binary notation.

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

Labels are not counted in len(P). 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 $\alpha = \langle x\$, i, \vec{a}_m \rangle$ such that $x \in \Sigma^*$, $1 \leq i \leq k$, and $\vec{a}_m \in (\Sigma \cup \{\$\})^m$. (We use $\vec{a}_m$ to denote the vector $\langle a_1, a_2, \ldots, a_m \rangle$.) In the sequel we use the symbol V to denote either a variable name or a symbol in $\Sigma \cup \{\$\}$ and so define the *content function* $\overline{V}$ (with respect to the configuration α) by

$$\overline{V} = \begin{cases} V \text{ if } V \in \Sigma \cup \{\$\} \\ a_j \text{ if } V = X_j \end{cases}$$

Let $\alpha = \langle x\$, i, \vec{a}_m \rangle$ and $\beta = \langle y\$, j, \vec{b}_m \rangle$ 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 $k \neq \ell$, and either there is a $b_\ell \in \Sigma$ such that $x\$ = b_\ell y\$$, or $x = y = \varepsilon$ and $b_\ell = \$$;
2) I_i = *write* V, $j = i + 1$, $y = x$ and $\vec{b}_m = \vec{a}_m$;
3) $I_i = X_\ell \leftarrow V$, $j = i + 1$, $y = x$, $b_\ell = \overline{V}$, and $b_k = a_k$ for $k \neq \ell$;
4) I_i = *if* $V_1 = V_2$ *go to* ℓ, $y = x$, $b_k = a_k$ for all k, and either $j = \ell$ and $\overline{V}_1 = \overline{V}_2$, or $j = i + 1$ and $\overline{V}_1 \neq \overline{V}_2$.

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 $y \in \Sigma^*$, $\vec{a}_m \in (\Sigma \cup \{\$\})^m$ and $1 \leq i \leq \text{prog}(P)$ such that

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

and I_i is *accept* (*halt*). Further, $\langle y\$, i, \vec{a}_m \rangle$ 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 defined by a partial function $P: \Sigma^* \to \Sigma^*$ 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

$$P(x) = b_1 b_2 \ldots b_p$$

where for each i, if $\alpha_i = \langle y\$, i, \vec{a}_m \rangle$ then $b_i = \overline{V}$ if I_i = *write* V and $\overline{V} \in \Sigma$, and $b_i = \varepsilon$ 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*), *go to* ℓ, *if* $V_1 \overset{=}{\neq} V_2$ *then* S_1, and *if* $V_1 \overset{=}{\neq} V_2$ *then* S_1 *else* S_2 (where S_1, S_2 represent statements).

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 and identical input and output alphabets 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. Notice that we can modify a fmp to determine whether it has entered an infinite loop and halt if so (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 non-accepting states. The initial state is $\langle i, a_1, \ldots, a_m \rangle$ where i is the label of the first encountered *read* statement, and $a_1, \ldots a_m$ are the values of the variable 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 configuration of the fmp (except 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 the following lemma. Note that, unlike the informal syntax extensions, some of these may have drastic effects

on the size of programs and hence on their complexity.

Lemma 2 For any fmp 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);

4) P' has no inaccessible instructions;

5) P' has no assignment statements.

Proof For (1), simply append to P the instructions

$$k + 1\text{: } \mathit{halt}\text{; } k + 2\text{: } \mathit{accept}$$

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

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

$$\ell\text{: } \mathit{read}\ X_1;$$
$$\ell + 1\text{: } \mathit{if}\ X_1 \neq \$\ \mathit{go\ to}\ \ell;$$
$$\ell + 2\text{: } \mathit{halt}$$

and relabel as necessary.

(3) The maximum number of instructions that can be executed between *read* instructions without P looping is $\leq k \cdot s^m$. Let c be 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.

Replace each *read* X_i instruction by

$$\mathit{read}\ X_i;\ \mathit{if}\ X_i = \$\ \mathit{then\ halt};\ Y_0 \leftarrow \$;\ \ldots;\ Y_{c-1} \leftarrow \$$$

Replace any other instruction I by

```
I;
if Y0 = $ then Y0 ← a else
    begin Y0 ← $; if Y1 = $ then Y1 ← a else
          begin
                ...
                Yc-2 ← $; if Yc-1 = $ then Yc-1 ← a else halt
          end
    . . .
    end
```

(4) This follows by first constructing a gsm as in Lemma 1, then removing the inaccessible states and translating the resulting gsm back to a fmp.

(5) This also follows directly from Lemma 1. □

While fmps are not powerful enough to simulate all nondeterministic gsms, there is an easy construction to prove that

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 $\ell \cdot \log n$.

Proof We use one variable Z to receive each input character 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 P of states such that $\delta(Q,b) = P$ as each symbol b is read and then the Y_i are copied into the X_i. □

III. Basic Results

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

1) $\mathrm{Accept} = \{P \in \mathrm{fmp} \mid \exists x \in \Sigma^* \text{ such that P accepts x}\}$
$= \{P \in \mathrm{fmp} \mid L(P) \neq \emptyset\}$

2) $\mathrm{Halt} = \{P \in \mathrm{fmp} \mid \exists x \in \Sigma^* \text{ such that P halts (but does not accept) on input x}\}$

3) Loop = {P ∈ fmp | ∃x ∈ Σ* such that P enters an infinite loop on input x}

4) Reject = Halt ∪ Loop

5) Accessible = {⟨P, i⟩ ∈ fmp x N | ∃x ∈ Σ* such that P executes its i^{th} instruction at least once on input x}

6) Inequiv = {$\langle P_1, P_2 \rangle \in fmp^2$ | ∃x $P_1(x) \neq P_2(x)$}

7) Remove = {⟨P,i⟩ ∈ fmp x N | ∃x $P(x) \neq P^i(x)$} where P^i is the program obtained by removing the i^{th} instruction of P and relabelling as necessary.

Other questions about optimization 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.

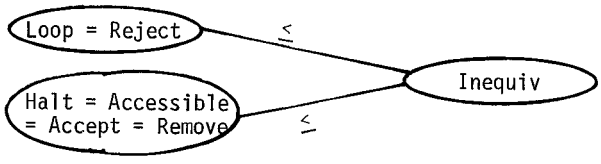
Many of these sets are of essentially the same complexity, in the following sense. Define $S \leq T$ to be true iff there is a function f( ) such that

1) for any x, $x \in S$ iff $f(x) \in T$;

2) for some c and all x, $|f(x)| \leq c|x|$; and

3) f(x) is computable in $\log|x|$ space.

Let $S = T$ mean $S \leq T$ and $T \leq S$. Then the following relations hold among these problems:



For example, Accept ≤ Inequiv may be shown as follows. Given a fmp P, we construct f(P) = ⟨P,P'⟩ where P' is identical to P except that each "*accept*" instruction in P is replaced by "*write* a; *accept*" and relabelling as appropriate, for some fixed a ∈ Σ. It is easily checked that f( ) satisfies 1), 2), and 3).

Thus we can restrict our attention to Accept, Loop, and Inequiv in our consideration of the complexities of these problems. The following definition and lemma (due in concept to Meyer and Stockmeyer [MS]) are essential to our lower bound results for space complexity.

Our model for measuring the complexity of problems is the one-tape Turing machine, that is a Turing machine with a single two-way read-write tape. We say that a Turing machine Z operates in space S( ) for $S(n) \geq n$ iff on input x it scans not more than $S(|x|)$ tape squares, and in time T( ) iff on input x it makes not more than $T(|x|)$ transitions. The class $DSPACE_S$ is defined to be the class of all sets which can be recognized by deterministic Turing machines operating in space S( ), and $NSPACE_S$ is defined analogously for nondeterministic Turing machines. $DTIME_T$ and $NTIME_T$ are defined analogously for time bounds.

Let* $R \subseteq \Delta^4$ be an arbitrary 4-ary relation defined on an alphabet Δ satisfying $\Sigma \cup \{\#, B, q_f\} \subseteq \Delta$, where $\#, B, q_f$ are distinct symbols not in Σ and let S( ) be a function such that $S(n) \geq n$. For any $x \in \Sigma^*$ (letting $n = |x|$) we define $Comp_{x,S}$ to be the set of all strings $y = b_1 \ldots b_m$ satisfying

1) each $b_i \in \Delta$,

2) $\#xB^{S(n)-n}\#$ is an initial substring of y,

3) $b_i = q_f$ for some $i \leq m$, and

4) for all i, $1 < i < m - S(n)$, $\langle b_{i-1}, b_i, b_{i+1}, b_{i+S(n)+1} \rangle \in R$.

A Σ-*encoding* of a set $L \subseteq \Delta^*$ is a set h(L), where $h: \Delta^* \to \Sigma^*$ is a one-to-one homomorphism such that $|h(a)| = |h(b)|$ for any $a,b \in \Delta$.

In the following M denotes a class of machine descriptions (or programs). We will initially, in our applications of Lemma 4 (below), have the class of fmp's as M, but later will apply it to various extensions of fmp's. Note that M could just as well be the class of Turing machines or another class of automata. We denote $\{M | M \in M \text{ and } L(M) \neq \emptyset\}$ by $Accept^M$ in general. In specific cases, we simplify this notation, so that $Accept^{fmp}$ is simply Accept, $Accept^{fmp^{num}}$ is $Accept^{num}$, and so on.

We shall use the clause "Q requires at least $NSPACE_{S(\ )}$" (and similar locutions for the other time

* We use the symbol ⊆ to indicate set inclusion and ⊂ to indicate proper inclusion, i.e., $A \subset B$ iff $A \subseteq B$ and $A \neq B$.

and space hierarchies) to mean that $Q \notin \text{NSPACE}_{S'(\ )}$ for any bound $S'(\ )$ such that

$$\bigcup_{c>0} \text{NSPACE}_{S'(cn)} \subset \text{NSPACE}_{S(n)}.$$

Note that this condition, while probably unfamiliar, is not particularly stringent. For most natural space bounds that do not exceed polynomials, we have $\text{NSPACE}_{S'(cn)} = \text{NSPACE}_{S'(n)}$ for all $c > 0$. In addition, if $0 < c < d$ implies $\text{NSPACE}_{S(cn)} \subset \text{NSPACE}_{S(dn)}$ (which is true for exponential bounds), then the conditions of Lemma 4 will imply that $\text{Accept}^M \notin \text{NSPACE}_{S(cn)}$ for some $c > 0$.

Lemma 4 Suppose that for each Δ and $R \subseteq \Delta^4$ there is a function $f^R: \Sigma^* \to M$ such that

1) $f^R(\ )$ is computable in log( ) space;
2) $|f^R(x)| = c|x|$ for some c and all $x \in \Sigma^+$, and
3) $f^R(x)$ accepts some Σ-encoding of $\text{Comp}_{x,S}$.

Then Accept^M requires at least $\text{NSPACE}_{S(\ )}$.

Proof Let Z be an arbitrary nondeterministic one-tape Turing machine operating in space S(n). Then there exist Δ and R such that $\forall x \in \Sigma^*$ Z accepts x iff $L(M) \neq \emptyset$, where $M = f^R(x) \in M$. This can be seen as follows: Let $\Delta = \Gamma \cup K \cup \{\#, B\}$, where Γ is the tape alphabet of Z and K is its set of states, and let q_f be the (unique) final state of A. Let $R = \{\langle a_{i-1}, a_i, a_{i+1}, b_i\rangle \mid \#a_1 \cdots a_{S(n)}\# \vdash_Z \#b_1 \cdots b_{S(n)}\#$ for $1 \leq i \leq S(n)\}$, (letting $a_0 = a_{S(n)+1} = \#$) where $\vdash_Z$ is the transition relation of Z. Then it is not difficult to see that $\text{Comp}_{x,S} = \{\alpha \mid \alpha$ is an accepting computation on input x by Z$\}$. Clearly $x \in L(Z)$ iff $\text{Comp}_{x,S} \neq \emptyset$.

Now suppose Accept^M can be recognized in space $\text{NSPACE}_{S'(\ )}$, where $\bigcup_{c>0} \text{NSPACE}_{S'(cn)} \subset \text{NSPACE}_{S(n)}$. Let Z be a Turing machine which operates in S(n) space and such that $L(Z) \in \text{NSPACE}_{S(n)} - \bigcup_{c>0} \text{NSPACE}_{S'(cn)}$. Then $x \in L(Z)$ can be determined by first calculating $f^R(x)$ and then deciding whether $f^R(x) \in \text{Accept}^M$. Clearly a nondeterministic Turing machine can do this in at most $S'(c|x|)$ space, contradicting the supposition that $L(Z) \in \text{NSPACE}_{S(n)} - \bigcup_{c>0} \text{NSPACE}_{S'(cn)}$. □

The above lemma also holds for off-line Turing machines with space bounds down to log n by a similar but more complex construction, which we omit for brevity's sake.

The problem $P \in \text{Accept}$ can be decided nondeterministically in the amount of space necessary to represent and manipulate a configuration without input $\alpha = \langle i, \vec{a}_m \rangle$. Comparing the size of a configuration without input and the size of the corresponding program P we conclude

Theorem 5 $\text{Accept} \in \text{NSPACE}_n$.

Proof We can decide $L(P) \neq \emptyset$ with a nondeterministic Turing machine by guessing an input x and simulating the operation of P on x\$. We simulate the control flow of P in the transition function of the Turing machine and so require space for the label of the current instruction and the contents of the variable, i.e., a configuration without input. We estimate the size of a fmp P with m variables and k instructions as

$$\max(k, m\lceil \log m \rceil, s) \leq \text{len}(P) \leq k(1 + \lceil \log k \rceil + 2\lceil \log m \rceil)$$

and the size of a configuration α as

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

Thus $\text{Accept} \in \text{NSPACE}_n$. □

From Lemma 4 we can obtain a lower bound for Accept, namely

Theorem 6 Accept requires at least $\text{NSPACE}_{n/\log n}$.

Proof We construct a fmp P which accepts $\text{Comp}_{x,n/\log n}$ for any R and x (where $n = |x|$) and which is of size linear in n. To do so we use $m = n/\log n$ variables $X_1, \ldots, X_m$ each of which ranges over $\Sigma \cup \{\$\}$. The variables and accompanying program structure may be thought of as a window which moves across the string y (see the definitions immediately preceding Lemma 4) checking whether the relation R is satisfied, whether $\#xB^{n/\log n - n}\#$ is an initial substring, and whether q_f occurs in y. Since we must name each of the m variables, we must have $m \log m \leq n$, which is the case for $m = n/\log n$. □

From our comments before Lemma 4, it is immediate that Theorems 5 and 6 apply to Halt, Remove, and Accessible as well as Accept. Theorem 5 also applies to Loop and Reject and a small modification of

Lemma 4 causes Theorem 6 to apply to them as well.

Note that the set $\{\langle P,x\rangle | P \text{ accepts } x\}$ is in $DSPACE_n$ and requires at least $DSPACE_{n/\log n}$, since we now do not need to guess the x accepted by P. The same comment, of course, applies to the other problems mentioned above.

By another technique we can show that the same bounds apply to Inequiv. In particular, we can show that if M_1, M_2 are deterministic gsms with final states then $M_1 \neq M_2$ is decidable in nondeterminstic logarithmic space. This result appears to be new and while not difficult, is not simply an easy corollary to the corresponding result for gsms without final states. Note the rather surprising fact that equivalence is of the same order of difficulty as halting. This also holds for most of the extensions of fmps, so we restrict our attention to the Accept problem in considering extensions.

Theorem 7 For gsms with final states M_1 and M_2, $M_1 \neq M_2 \in NSPACE_{\log n}$.

Since this result is of only subsidiary importance here, we reserve its proof for a fuller version of the paper.

For any $P \in$ fmp, let P' denote the naturally corresponding gsm equivalent to P. It is straightforward to build a Turing machine which, given $\langle P,i\rangle$ as input, will construct in linear space the i^{th} symbol of a description of P'. Now given any two fmps P_1 and P_2, we can apply the Turing machine which decides inequivalence for gsms to P_1' and P_2'. Analysis of the space requirements for this construction shows

Theorem 8 Inequiv $\in NSPACE_n$.

As a corollary to Theorem 6, we have

Corollary 9 Inequiv requires at least $NSPACE_{n/\log n}$.

Note that we began section II 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 (= 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.

We might introduce an element of non-determinism into fmps by allowing the instruction

$$go\ to\ \ell_1,\ell_2,\ldots,\ell_n$$

whose execution would cause a nondeterministic selection of one of the statements labeled $\ell_1,\ell_2,\ldots,\ell_n$ 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 and equivalent problems (Accessible, etc.), this change renders Inequiv undecidable by Griffiths' result [Gri]. In addition the problem "Given $M \in$ fmp, is $L(M) \neq \Sigma^*$" may be shown to require at least $NSPACE_{2^{cn/\log n}}$ for some $c > 0$, by use of Lemma 4.

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 are, though decidable in this limited context, 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.

IV. Finite Memory Programs with Numbers

In this and the succeeding two sections we discuss 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 II. 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 fmp^{rec} and $Accept^{rec}$ for fmps with recursion.

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,...,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: +, -, *, /. Semantically these are interpreted in the usual way, except that all values are taken modulo s, and / denotes integer division (with truncation).

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

Lemma 10 Any program P in fmp^{num} may be replaced by an equivalent program P' in fmp; further, the length of P' will be linearly bounded by the length of P, as P ranges over fmp^{num}.

Proof We first show how to replace each instruction of the form X ← Y op Z by an equivalent sequence involving only X ← 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 ← Y + Z may be replaced by the sequence

```
      W ← 0; X ← Y;
L:  if W = Z then go to NEXT;
      X ← X + 1; W ← W + 1; go to L;
NEXT:
```

and "X ← 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 11 $\text{Accept}^{\text{num}}$ is in NSPACE_n and requires at least $\text{NSPACE}_{n/\log n}$.

Proof Immediate from Theorems 5 and 6. □

Variable Word Size

In this variation we define a *finite memory program with variable word size* (fmp^{vws}) to be 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 0 as \$, and Σ as {1,2,...,s-1}; however, elements of Σ may be accessed implicitly by the use of addition modulo s. P is a fmp as defined before, with two exceptions:

a) the elements of Σ are represented in binary notation; and

b) P may contain instructions of the form X ← X + 1, with the addition performed modulo s.

Note that we could also allow instructions of the form "X ← Y + Z", etc., without increasing the computational abilities of a fmp^{vws}, since the same technique as in Lemma 10 could be used to reduce them to "X ← X + 1". However this instruction cannot itself be eliminated as was done in Lemma 10. The reason is that s is variable so each "X ← X + 1" would be replaced by $0(s) = 0(2^{\text{len}(P_s)})$ instructions.

Theorem 12 $\text{Accept}^{\text{vws}} \in \text{NSPACE}_{n^2}$.

Proof The proof is as for Theorem 5, 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:

$$\log k + m\lceil \log s \rceil = \log k + m|\bar{s}| \leq \text{len}(P_s) + \text{len}(P_s)\cdot\text{len}(P_s)$$

$$\leq 0(\text{len}^2(P_s)).$$

□

Theorem 13 $\text{Accept}^{\text{vws}}$ requires at least $\text{NSPACE}_{n^2/\log^2 n}$.

Proof This lower bound is obtained in a way similar to the corresponding result for ordinary fmps (Theorem 6), by constructing a fmp^{vws} of size proportional to n which accepts $\text{Comp}_{x,n^2/\log^2 n}$, for any fixed R and x, where $n = |x|$. The key to the construction is to consider m variables, each containing a number between 0 and 2^m-1. Each may be viewed as a string of m bits, so a configuration specifies m^2 bits of memory which can be used to store the most-recently-read string of m^2 symbols from the string y which is defined just before Lemma 4. The $1/\log^2 n$ factor enters because of the fact that each of the m variables must have a distinct name, so m must satisfy $m \log m \leq n$. Further details will be given in the final version of this paper. □

V. 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 $\text{fmp}^{\text{array}}$) has an alphabet Σ, the endmarker \$, and $s = |\Sigma \cup \{\$\}|$, as before. In addition we have a fixed *subscript set* $\Theta \subseteq \Sigma$. A $\text{fmp}^{\text{array}}$ P is ther a program, as described in section II, except that (1) for each occurrence of an X_i there we may now have either an X_i or an expression of the form $X_i[W_1,\ldots,W_\ell]$ 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 $X_i[W_1,\ldots,W_\ell]$, or a symbol from $\Sigma \cup \{\$\}$. The number of

subscripts adhering to X_i must be the same throughout P. The semantics are a straightforward and obvious extension of the semantics of fmps except for two points:

1) if in an expression $X_i[W_1,\ldots,W_\ell]$, one or more of the W_p's has as its value a symbol not in Θ, then the entire expression evaluates to 0; and
2) in a configuration $\langle x\$,i,a_1,\ldots,a_m\rangle$, each a_i is a function $a_i:\Theta^\ell \to \Sigma \cup \{\$\}$, where ℓ is the number of subscripts (possibly zero) adhering to the variable X_i.

By a similar argument to many of those above, we obtain the following

Theorem 14 The complexity of the problem $\text{Accept}^{\text{array}}$ is exactly NSPACE_{t^n}, where $t = |\Theta|$ (provided $t \geq 2$).

The lower bound is obtained by showing how to construct, given x, a $\text{fmp}^{\text{array}}$ which accepts Comp_{x,t^n} in space linear in $n = |x|$. The key is to construct an n-dimensional array whose elements contain a part (of length t^n) of the string y referred to immediately ahead of Lemma 4.

Note that by varying the size of the subscript set or relaxing the requirement that $\Theta \subseteq \Sigma$ we may radically alter the complexity bounds. The resulting bounds will, however, correspond to nondeterministic space proportional to the number of cells addressable as a function of the size of the program.

VI. Finite Memory Programs with Recursion

We next consider fmps with recursive subroutines. We discuss two forms, one with no explicit parameter passing mechanism but which turns out to be equivalent to call-by-value and the other which is explicitly call-by-name.

Recusion Without Parameters

A *finite memory program with recursion* (or fmp^{rec}) is a (2p + 2)-tuple $P = \langle P_0,X_0,P_1,X_1,\ldots,P_p,X_p\rangle$ where

1) $X_0,\ldots,X_p$ are disjoint sets of variable names (X_0 is the set of *global variables*, and X_i is the set of *local variables* for P_i for $i \geq 1$);
2) for $i = 0,1,\ldots,p$, P_i is an ordinary fmp except that
   a) P_i may refer only to variables in $X_0 \cup X_i$;
   b) P_i may contain instructions of the form "*call* P_j" (for $j \geq 1$) and "*return*";
3) the instructions of $P_0,P_1,\ldots,P_p$ have distinct labels, and
4) no *go to* instruction may cross a procedure boundary.

If expressed in the style of ALGOL, the syntax would take the following form:

```
begin symbol all variables in X0;
procedure P1;
   begin symbol all variables in X1;
         body of P1
   end;
. . .
procedure Pp;
   begin symbol all variables in Xp;
         body of Pp
   end;
body of main program (P0)
end
```

The semantics are a bit different from those of an ordinary fmp. A *configuration* is a variable length tuple (the length is three more than twice the current depth of calls) of the form

$$\alpha = \langle x\$,i_0,y_0,i_1,y_1,\ldots,i_q,y_q\rangle$$

where for each i_j, if I_{i_j} is an instruction in P_ℓ, then $y_{i_j} \in (\Sigma \cup \{\$\})^{|X_\ell|}$. The initial configuration is $\langle x\$,1,\$^{|X_0|}\rangle$. Intuitively, for $j < q$, i_j is a return address and y_{i_j} denotes the values of the local variables of the corresponding P_ℓ. The current instruction is that numbered i_q.

The *immediately yields* relation $\alpha \vdash\!\!\!- \beta$ is defined nearly as before. The effects of *read* X_i, *write* V, $X_i \leftarrow V$, *if* $V_1 = V_2$ *go to* ℓ, *accept* and *halt* are all essentially as for an ordinary fmp. The effect of "*call* P_j" is to add the pair $\langle i_j, \$^{|X_j|} \rangle$ to the end of α, where i_j is the number of the first instruction of P_j, and to change i_q to $i_q + 1$. The effect of "*return*" is to remove $\langle i_q, y_q \rangle$ from the end of α and to continue execution with the instruction numbered i_{q-1}.

Note that the semantics are quite consistent with those of the ALGOL 60 skeleton presented above. The effect of "call-by-value" parameters may clearly be achieved by setting global variables before a *call*, and then copying them into local variables immediately after procedure entry.

Clearly fmp^{rec} is computationally more powerful than fmp, since nonregular sets such as $\{a^n b^n | n \geq 0\}$ may be accepted. In particular, we have

Theorem 15 For any $P \in \text{fmp}^{\text{rec}}$ there is a deterministic pushdown transducer (dpdt) M equivalent to P (i.e., $P(x) = M(x\$)$ for all $x \in \Sigma^*$) and conversely.

Proof Given a $P \in \text{fmp}^{\text{rec}}$, the construction of an equivalent dpdt M, with final states and endmarker is a straightforward extension of the proof of the corresponding part of Lemma 1. The state set of the dpdt corresponds to the set of configurations of P, excluding input and the stack. The stack alphabet of the dpdt is the set of instruction labels $\{1,\ldots,k\}$ of P, along with symbols from Σ to hold current current memory contents. The effect of i:*call* P_j is simulated by pushing i + 1 onto the stack and transferring to the state corresponding to the first instruction of P_j and the current memory contents; *return* is simulated by branching to the state corresponding to the instruction on top of the stack (along with the current memory contents), and popping the stack.

Now suppose we are given a dpdt M. We may without loss of generality assume M is presented in the form of a program similar to that of a fmp, with the following instruction types:

read X, *write* C, *push* C, *pop* X, *if* X = C *go to* ℓ, *accept* and *halt*

with the obvious semantics. Only a single variable, X, is needed; C denotes any symbol from $\Sigma \cup \{\$\}$. Let the instructions of M be labelled 1,2,...,k.

We now define $P = \langle \{X, F_1, \ldots, F_k\}, P_0, \emptyset, \ldots, P_r, \emptyset \rangle$, where P_0 is defined below, and each P_i is indexed by a constant C in $\Sigma \cup \{\$\}$; P_i may also be denoted by P^C. The overall structure of P, expressed in ALGOL-like notation, is as follows ("a" is some symbol from Σ):

```
begin symbol X,F1,...,Fk;
. . .  comment one procedure for each C ∈ Σ ∪ {$};
procedure P^C;
   begin
L^C:if F1 ≠ $ then begin F1 ← $; go to L^C_1 end else
   if F2 ≠ $ then begin F2 ← $  go to L^C_2  end else
                        . . .                  else
   if Fk ≠ $ then begin Fk ← $; go to L^C_k  end;
   L^C_1:J^C_1;
   . . .
   L^C_k:J^C_k;
   end of procedure P^C;
F1 ← a;
call P^$; halt
end
```

If instruction i of M is:	then J^C_i in P^C is:
read X, *write* D, *accept*, *halt*	the same instruction
if X = D *go to* ℓ	*if* X = D *go to* L^C_ℓ
push D	$F_{i+1} \leftarrow a$; *call* P^D; *go to* L^C;
pop X	X ← C; $F_{i+1} \leftarrow a$; *return*;

The effect is to replace each *push* by a *call*, and each *pop* by a *return*; the variants P^C, one for each symbol, are used to keep track of which symbols were pushed. The flags $F_1,\ldots,F_k$ are used to preserve the point of control across a *call* or *return*; a similar technique will be used for the simulation of a queue by recursion with call-by-name. □

Theorem 16 There is a constant $c > 0$ such that $\text{Accept}^{rec} \in \text{DTIME}_{2^{cn}}$.

Proof Detailed examination of the construction of the first part of the proof of Theorem 15 reveals that M may be constructed from P in time $2^{a\ \text{len}(P)}$, for fixed a and all P. Now $L(M) \neq \emptyset$ iff $L(P) \neq \emptyset$, and the test "$L(M) \neq \emptyset$" may be carried out in time polynomial in len(M). Thus "$L(P) \neq \emptyset$" may be determined by constructing M and then testing to see whether $L(M) \neq \emptyset$. The total time used is clearly bounded by 2^{cn} for some $c > 0$. □

In the following we shall denote the binary representation of a nonnegative integer i by $\bar{i}$. Further, 0 and 1 will be identified with two symbols from Σ.

Definition 17 Let $b > 0$, and let OP and S be deterministic offline Turing machines with one work tape such that, if given $x,\bar{i},\bar{j}$ (where $x \in \Sigma^*$ and $i,j \in X_x \overset{\text{def}}{=} \{0,1,\ldots,T(bn) - 1\}$ are presented in binary notation), then

a) OP computes a binary operation $OP_x(i,j)$ on X_x,

b) S decides whether i is in a set $S_x \subseteq X_x$, and

c) OP and S operate in space $\log|x|$.

The GEN problem determined by x, OP and S is the following:

Given $\text{GEN}_S^{OP}(x) \overset{\text{def}}{=} \langle X_x, OP_x, S_x, 0\rangle$

To determine Whether 0 is in the smallest subset of X_x which contains S_x and is closed under OP_x (i.e., whether 0 is generated by S_x under OP_x). □

It was shown in [JL] that the GEN problem is complete for P, i.e., $\bigcup_k \text{DTIME}_{n^k}$. We now state without proof a closely related result for exponential time bounds. The proof depends on a careful analysis of the proof of Theorem 8 and Corollary 9 of [JL].

Theorem 18 Let Z be a Turing machine which operates in time T(n), where $T(n) \leq 2^{an}$ for some $a > 0$ and all n. Then there are offline Turing machines OP and S (as in Definition 17) such that for all $x \in \Sigma^4$:

$$Z \text{ accepts } x \text{ iff } \text{GEN}_S^{OP}(x) \text{ has a solution.}$$

The following is similar in purpose and use to Lemma 4.

Theorem 19 Let $\mathcal{M}$ be a class of machine descriptions (e.g., programs), let $T(n) \leq 2^{an}$ for some $a > 0$ and all n, and suppose further that $0 < c < d$ implies $\text{DTIME}_{T(cn)} \subset \text{DTIME}_{T(dn)}$. If, for every b,OP,S as in Definition 17, there is a function $f:\Sigma^* \to \mathcal{M}$ such that

a) f is computable in time $T(n/2)$;

b) $|f(x)| = c|x|$ for some c and all x; and

c) for all x, $f(x) \in \text{Accept}^{\mathcal{M}}$ iff $\text{GEN}_S^{OP}(x)$ has a solution.

Then $\text{Accept}^{\mathcal{M}} \notin \text{DTIME}_{T(dn)}$ for some $d > 0$.

Proof Suppose $\text{Accept}^{\mathcal{M}} \in \text{DTIME}_{T(dn)}$ for all $d > 0$, and let Z be any Turing machine which operates in time T such that $L(Z) \in \text{DTIME}_{T(n)} - \text{DTIME}_{T(n/2)}$. By Theorem 18 appropriate OP and S Turing machines exist. Let f be as in the premise. Combining the conclusion of Theorem 18 and premise c) above, we see that for all x, $x \in L(Z)$ iff $f(x) \in \text{Accept}^{\mathcal{M}}$. Now by assumption (with $d = 1/2c$), $\text{Accept}^{\mathcal{M}} \in \text{DTIME}_{T(n/2c)}$. Thus membership in L(Z) may be determined by calculating f(x) (of length $c|x|$) and testing for membership in $\text{Accept}^{\mathcal{M}}$, which requires at most time $2T(|f(x)|/2c) = 2T(|x|/2)$. But this contradicts the fact that $L(Z) \notin \text{DTIME}_{T(n/2)}$. Thus the assumption that $\text{Accept}^{\mathcal{M}} \in \text{DTIME}_{T(dn)}$ for all $d > 0$ must be false. □

Corollary 20 $\text{Accept}^{rec} \notin \text{DTIME}_{2^{dn/\log n}}$ for some $d > 0$.

Proof We show how the premises of Theorem 19 can be satisfied for $T(n) = 2^{n/\log n}$. First, for any OP, S and x consider the context-free grammar $G_x = (\{A_0,A_1,\ldots,A_{T(bn)-1}\},\{0,1,+\},P,A_0)$, where P contains:

$$A_k \to \bar{k} \text{ for each } k \in S_x, \text{ and}$$

$$A_k \to +A_iA_j \text{ whenever } OP_x(i,j) = k.$$

Clearly if $A_i \overset{*}{\Rightarrow} w \in \{0,1,+\}^*$, w may be viewed as a Polish prefix expression, with the symbol "+"

replacing OP_x. Further it is easily seen that $A_i \overset{*}{\Rightarrow} w$ holds only if i is the value denoted by w; thus $L(G_x) \neq \emptyset$ iff 0 is generated from S_x.

We now define f so that for each x, f(x) is a fmp^{rec} which accepts w iff $A_0 \overset{*}{\Rightarrow} w$, and satisfies conditions (a), (b) and (c) of Theorem 18. The form of f is as follows, again in a pseudo-ALGOL notation. We let $m = \lceil n/\log n \rceil$, where, as usual, $n = |x|$.

begin symbol $IN, K_1, \ldots, K_m, I_1, \ldots, I_m$;
procedure OP; *comment* OP calculates $OP_x(i,j)$ and stores the result into $K_1, \ldots, K_m$,
taking $\bar{i}$ from $I_1, \ldots, I_m$ and $\bar{j}$ from $K_1, \ldots, K_m$;
[body of OP];
procedure S; *comment* S takes $\bar{k}$ from cells $K_1, \ldots, K_m$ and sets IN to 1 if $k \in S_x$, and 0 if not;
[body of S];
procedure P; *comment* P reads an expression and returns its binary value in $K_1, \ldots, K_m$;
begin symbol $L_1, \ldots, L_m$;
read K_1;
if $K_1 \neq +$ *then begin read* K_2;...*read* K_m; *call* S;
if IN = 1 *then return else halt*
end;
call P; $L_1 \leftarrow K_1$;..;$L_m \leftarrow K_m$; *call* P; $I_1 \leftarrow L_1$;..;$I_m \leftarrow L_m$;
call OP;
end of P;
call P; *if* $K_1 = \ldots = K_m = 0$ *then accept else halt*
end of f(x);

All three of the conditions of Theorem 19 will be satisfied, provided it is possible to construct procedures OP and S to behave as specified, in time polynomial in n, and to have size which is linear in n. Condition (a) then follows from the simplicity of f(x) and the fact that T(n/2) dominates any polynomial in this case. Condition (b) also follows (with the addition of padding symbols if necessary), since $m \log m \leq n$. Condition (c) is immediate, since it is clear that when given a Polish prefix expression w, machine f(x) will evaluate it (using OP_x for +), and accept w iff w is in $L(G_x)$; thus $f(x) \in Accept^{rec}$ iff $GEN_S^{OP}(x)$ has a solution.

Finally, OP and S are obtained from the Turing machines of the same names from Defition 17, by constructing simulation programs in fmp. Each work-tape square is represented by a separate variable, and the state transitions are encoded as instruction sequences in a straightforward way. The number of variables needed is bounded by log n, which is certainly much smaller than n. The only complication arises from representation of the input tape (which corresponds to x along with variables $K_1, \ldots, K_m$ and $I_1, \ldots, I_m$), since a separate variable can not be assigned to each symbol of x without exceeding linearity in n. One solution is to define a "table lookup" procedure which, when given $i \leq n$ as a binary number (via log n variables), will return the i^{th} symbol of x. This can easily be done in space linear in n. □

Case 2: Parameters Called by Name

We now show that at least one interpretation of "call-by-name" parameter passing gives rise to an undecidable halting problem. Consequently emptiness, equivalence and the other problems are also undecidable.

In order to see this we show that any of a family of programs which is known to have an undecidable halting problem may be faithfully simulated by fmp-like programs which are completely finite, except for an unbounded push-down stack which contains return addresses and parameter information.

Consider a simple programming language Q with instructions of the form:

1) *Enqueue* X--that is, add the value of X to the left end of a queue
2) *Dequeue* X--that is, remove the value which is at the right end of the queue, and store it into X
3) *if* $V_1 = V_2$ *go to* ℓ
4) *halt*

} as for fmps.

The semantics should be clear. (We assume the queue symbols are positive integers.) Now it is known (e.g., via Minsky's results on Tag systems [Min]) that the following problem is undecidable:

Given an arbitrary program P in Q,
To determine whether or not P will eventually halt (by performing a *halt* instruction or by emptying its queue).

Theorem 21 The halting problem is undecidable for finite memory programs with recursion and call-by-name.

Proof Let P = 1:I_1;...;k:I_k be any program in Q, which uses variables $X_1,\ldots,X_m$. Consider the following pseudo-ALGOL program:

```
        begin integer I,X1,...,Xm;
        procedure P(A,B,C); value C; integer A,B,C;
           begin if I = 1 then go to L1 else
                 if I = 2 then go to L2 else...else
                 if I = k then go to Lk else go to Lk+1;
L1:        encoding of I1;
              . . .
Lk:        encoding of Ik;
Lk+1:      end;
        I:= 1;
        call P(0,-1,-1)
        end
```

Instruction Encoding:

a) I_i = "*Enqueue* X" is encoded as

```
I:= i+1; P(if B < 0 then C else A,C,X); go to Lk+1;
```

b) I_i = "*Dequeue* X" is encoded as

```
if B < 0 then go to Lk+1; X:= A; A:= -A;
```

c) I_i = "*if* $V_1 = V_2$ *go to* L;" appears unchanged.

Notes:

1) This program differs slightly from ALGOL 60 in that "A:= -A" becomes, in effect after parameter substitution:

   if B < 0 *then* C *else* A:= -(*if* B < 0 *then* C *else* A)".

   This is not legal in ALGOL 60, but its import should be clear.

2) At any point during execution the runtime stack will contain (among other things) a list of all previous values of C, resulting from earlier calls. If this list has the form -1,-3,-2,1,4,3,2, it signifies that the queue now consists of 1,4,3,2, and that elements 1,3,2 have been removed at earlier steps.

3) Note that each variable only assumes a finite range of values or (in the case of parameters A,B,C) a pointer to part of the original program. Thus all memory is strictly finite, except for the stacking implied by recursive calls.

4) It may be verified that this program will halt iff the original Q program P halts. Halting of Q programs, however, is undecidable. Thus the halting problem for any class of finite memory programs large enough to contain programs of the form above is also undecidable.

5) Representation of this pseudo-ALGOL program as a type of fmp, with call-by-name parameter passing is straightforward, although a bit complex. It is necessary to store parameter access information (e.g., "thunk" addresses) in the stack in addition to the return addresses. In addition the semantics would have to ensure the proper evaluation of formal parameters.

□

VII. 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 shown that in general the problems require essentially the work 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.

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, and so on.

REFERENCES

[Coo] Cook, Stephen A., Characterization of Pushdown Machines in Terms of Time-Bounded Computers, *Journal of the Association for Computing Machinery*, vol. 18, no. 1, January 1971, pp. 4-18.

[Gri] Griffiths, T. V., The unsolvability of the equivalence problem for Λ-free nondeterministic generalized machines, *Journal of the Association for Computing Machinery*, vol. 15, no. 3, July 1968, pp. 409-413.

[HU] Hopcroft, John E., and Jeffrey D. Ullman, *Formal Languages and Their Relation to Automata*, Addison-Wesley, 1969.

[JL] Jones, Neil D., and Laaser, William T., Complete problems for deterministic polynomial time, *Proceedings of Sixth Annual ACM Symposium on Theory of Computing*, April-May 1974, pp. 40-46.

[MR] Meyer, Albert R., and Dennis M. Ritchie, *Computational Complexity and Program Structure*, IBM Research Paper RC-1817, May 1967.

[MS] Meyer, Albert R., and L. J. Stockmeyer, The equivalence problem for regular expressions with squaring requires exponential space, *Thirteenth Annual Symposium on Switching and Automata Theory*, October 1972, pp. 125-129.

[Min] Minsky, Marvin L., *Computation: Finite and Infinite Machines*, Prentice-Hall, 1967, pp. 267-273.

[Ros] Rosenberg, Arnold, On multi-head finite automata, *Proceedings of the Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, 1963, pp. 221-228.