LOGSPACE and PTIME Characterized by Programming Languages

Neil D. Jones¹

DIKU, University of Copenhagen, Denmark
E-mail: neil@diku.dk

A programming approach to computability and complexity theory yields more natural definitions and proofs of central results than the classical approach. Further, some new results can be obtained using this viewpoint. This paper contains new intrinsic characterizations of the well-known complexity classes PTIME and LOGSPACE, with no externally imposed resource bounds on time or space. LOGSPACE is proven identical with the decision problems solvable by read-only imperative programs on Lisp-like lists; and PTIME is proven identical with the problems solvable by recursive read-only programs.

Key words: Complexity, read-only or cons-free programs, PTIME, LOGSPACE.

1 Introduction

Thesis. We maintain that Computability and Complexity theory, and Programming Language and Semantics (henceforth CC and PL) have much to offer each other, in both directions². CC has a breadth, depth, and generality not often seen in PL, and a tradition for posing (and occasionally answering) sharply defined open problems of community-wide interest. PL has a firm grasp of algorithm design, presentation, and implementation, and several well-developed frameworks for making precise semantic concepts over a wide range of PL concepts (functional, imperative, control flow operators, communication/concurrency, object-orientation, and much more).

¹ This research was partially supported by the Danish Research Council (DART project), and the Esprit BRAs Semantique and Atlantique. An earlier version of this paper appeared in Electronic Notes in Theoretical Computer Science, proceedings of the 1995 meeting Mathematical Foundations of Programming Semantics.
² This theme is further developed in [12], which is an introduction to computability and complexity using programming-related models.
Some concrete connections. It is natural in PL to have efficient built-in 
data construction and decomposition operators; these are just examples of the 
"pairing functions" known in CC from the 1930s. We take them as primitives, 
as in the Lisp language. The main results of this paper are simple "intrinsic" 
characterizations of the well-known problem classes LOGSPACE and PTIME, in 
programming terms and without external imposition of space or time bounds; 
and new insights into the role of persistent (as opposed to evanescent) storage. 
One effect is that the use of PL concepts lead (at least for Computer Scientists) 
to more understandable statements of theorems and proofs in CC, and to 
stronger results.

Some interesting questions. Further, a number of old CC questions take 
on new life, and natural new questions arise. An important question category is: what is the effect of the programming styles we employ (functional, 
imperative, etc.) on the efficiency of the programs we can possibly write?

A puzzling tradeoff: we will see that a problem is solvable in polynomial time 
just in case it is solvable by recursive read-only program. Paradoxically, recursive 
read-only programs often run in exponential time (not a contradiction, 
since they can be simulated in polynomial time by memoization.) This trade- 
off indicates a tension between running time and memory space which seems 
worth further investigation.

2 Programming languages and complexity classes

2.1 Programming languages

Definition 2.1 A programming language L consists of two sets, L-programs 
and L-data, together with L's semantic function, which associates with every 
p \(\in\) L-programs a corresponding (partial) input-output function

\[ [p]^L(.) : L-data \rightarrow L-data \]

We are concerned with time and space bounded computations. A definition 
comprising both follows:

Definition 2.2 A resource-usage measure on L-programs is a partial function

\[ usage^L_p(.) : L-data \rightarrow \{\bot, 0, 1, 2, \ldots\} \]
This paper treats only decision problems, hence the following. Note that program \( p \) must terminate on all inputs.

**Definition 2.3** Let \( \text{true, false} \in L\text{-data} \) be two distinct data values. An L-program \( p \) **decides** a subset \( A \) of \( L\text{-data} \) if for any \( d \in L\text{-data} \)

\[
[p]^L(d) = \begin{cases} 
\text{true} & \text{if } d \in A \\
\text{false} & \text{if } d \in L\text{-data}\setminus A
\end{cases}
\]

\( \square \)

The requirements above are satisfied by any reasonable programming language. Additional naturality restrictions are often imposed, for example the following:

(i) **Turing completeness:** a partial function \( f : L\text{-data} \to L\text{-data} \) is computable if and only if \( f = [p]^L \) for some L-program \( p \).

(ii) **Universal function property:** \([p]^L(d)\), regarded as a two-argument partial function of \( p \) and \( d \), is a computable partial function.

(iii) Suppose L-programs may have multiple inputs. The \( s\text{-m-n} \) property, or program specialization: for any \( m, n \geq 0 \) there exists a computable total function

\[ s^m_n : \text{L-programs} \times L\text{-data}^m \to \text{L-programs} \]

such that for any \( m+n \)-input L-program \( p \) and inputs \( s_1, \ldots, s_m, d_1, \ldots, d_n \in L\text{-data} \), it holds that

\[ [p]^L(s_1, \ldots, s_m, d_1, \ldots, d_n) = [s^m_n(p, s_1, \ldots, s_m)]^L(d_1, \ldots, d_n) \]

(iv) For any \( p \in \text{L-programs} \) and \( d \in L\text{-data} \), \([p]^L(d) = \perp \) if and only if \( \text{usage}_p^L(d) = \perp \).

(v) This set is decidable for any \( p \in L\text{-data} \):

\[ \{ (p, d, n) \mid \text{usage}_p^L(d) \leq n \} \]

Properties (i)–(iii) state that language L is an **acceptable enumeration** of the partial recursive functions \([15]\), and properties (iv), (v) state that \text{usage} is a complexity measure acceptable in Blum’s sense \([2]\). These properties are easily seen to hold for the languages we will introduce; proofs are natural but omitted for brevity.

**Definition 2.4** Let languages \( L \) and \( M \) have the same data. Then \( L \) **can simulate** \( M \), written \( M \preceq L \), if for every \( M \)-program \( p \) there exists an \( L \)-program \( q \) such that \([p]^M = [q]^L\). Language \( L \) is **equivalent to** \( M \), written \( L \equiv M \), if \( L \preceq M \) and \( M \preceq L \). \( \square \)
2.2 Complexity classes

By Property i, any two languages satisfying the conditions above can simulate one another. Our concern, however, will be efficient mutual simulations. Given a bound on resources, e.g., \( n^3 \), complexity is concerned with the set of all problems solvable by programs \( p \) that run within that bound, when applied to any possible input.

We diverge slightly from custom, and define a complexity class to be a set of programs that run within a certain resource bound, rather than a set of problems that can be solved within that bound. Thus we can and will regard an \( L \) complexity class as a sublanguage of \( L \), restricted to all programs satisfying the given resource bounds, with programs having exactly the same semantics as they would in \( L \).

It is customary to measure resource usage as a function of the input size \( |d| \) (where \( |d| \) is length, number of symbols, etc. of input \( d \)), rather than the input itself. (Concrete size measures will be seen shortly.)

**Definition 2.5** Given programming language \( L \) with resource-usage measure usage, and a function \( f : \mathbb{N} \to \mathbb{N} \), define

\[
L^{usage(f)} = \{ p \in L\text{-program} \mid \text{usage}_p(d) \leq f(|d|) \text{ for all } d \in L\text{-data} \}
\]

This paper concerns two resource-usage measures: time and space (concrete definitions soon to come). The problems decidable in, respectively, polynomial time and logarithmic space, will be central:

\[
L^{time} = \bigcup_{a,b \geq 0} L^{time(\lambda n \cdot a + n^b)}
\]

\[
L^{space} = \bigcup_{k \geq 1} L^{space(\lambda n \cdot k \log n)}
\]

3 The WHILE language and Turing machines

3.1 The WHILE language

We introduce a simple programming language called WHILE, in essence a small subset of Pascal or Lisp. Why just this language? Because WHILE seems to

\footnote{The difference is inessential, e.g. the well-known class PTIME is exactly the set of problems solvable by programs in \( \text{WHILE}^{time} \) as defined below.}
have just the right mix of expressive power and simplicity for our purposes. *Expressive power* is important for carrying out constructions, e.g., or showing how one program can simulate another. *Simplicity* is also essential to prove theorems about programs and their behavior. This argues against larger, more powerful languages, since proofs about them would simply be too complex to be easily understood.

### 3.1.1 Syntax of WHILE data and programs

**Definition 3.1** Let \( A = \{a_1, \ldots, a_n\} \) be some finite set. Then

(i) \( \mathcal{D} \) is the smallest set satisfying \( \mathcal{D} = (\mathcal{D} \times \mathcal{D}) \cup A \). The *pairing* operation yields value \((d_1, d_2)\) when applied to values \(d_1, d_2\).

(ii) The *size* \( |d| \) of a value \( d \in \mathcal{D} \) is defined as follows: \( |a| = 1 \) if \( a \in A \), and \( 1 + |d_1| + |d_2| \) if \( d = (d_1, d_2) \).

\[ \square \]

Values in the set \( \mathcal{D} \) are built up from *atoms* in \( A \) by finitely many applications of the *pairing* operation “\( \text{cons} \)”\). A value \( d \in \mathcal{D} \) is thus a binary tree with atoms as leaf labels. An example, written in “fully parenthesized form”: \(((a.((b.nil).c)).nil)\). In formal constructions we will only use a single atom, called nil (so in fact \( A = \{\text{nil}\} \)), but will for readability’s sake use more atoms in examples. There is no loss of generality, as multiple-atom structures can be encoded into ones using only nil with no loss of information, the overhead being a multiplication of run time by a constant.

A compact linear notation for values: Unfortunately it is hard to read deeply parenthesized structures (one has to resort to counting), so we will use a more compact “list notation” taken from the Lisp and Scheme languages, in which

\[
\begin{align*}
() & \quad \text{stands for} \quad \text{nil} \\
(d_1 \cdots d_n) & \quad \text{stands for} \quad (d_1.(d_2.\cdots.(d_n.\text{nil})\cdots))
\end{align*}
\]

The syntax of WHILE programs is given by the “informal syntax” part of Figure 3.1. Programs manipulate tree structures built by \( \text{cons} \) from atoms. Operations \( \text{hd} \) and \( \text{tl} \) (head, tail) decompose such structures. In tests, nil serves as “false,” and anything else serves as “true.” For readability we will often write \text{false} in place of nil, and \text{true} in place of (nil.nil). For an example, the following program, \text{reverse}:
\begin{center}
\begin{tabular}{ | l | l |}
 \hline
 P : Program & ::= read X; C; write Y \\
 \hline
 C : Command & ::= Z := E \\
 & | C1; C2 \\
 & | if E then C1 else C2 \\
 & | while E do C \\
 \hline
 E : Expression & ::= Z (any variable) \\
 & | D \\
 & | cons E1 E2 \\
 & | hd E \\
 & | tl E \\
 \hline
 X, Y, Z : Variable & ::= X0 | X1 | ... \\
 D : Data-value & ::= nil | (D.D) \\
 \hline
\end{tabular}
\end{center}

\textit{Figure 3.1: \textsc{while} program syntax}

read X;
Y := nil;
while X do \{ Y := cons (hd X) Y; X := tl X \};
write Y

satisfies \([\text{reverse}] (a.(b.(c.nil))) = (c.(b.(a.nil)))\) or, in the compact notation, \([\text{reverse}] (a \ b \ c) = (c \ b \ a)\). In both cases the outermost parenthesis pair has been omitted.

\subsection{Semantics of \textsc{while} programs}

Informally, the net effect of \textit{running a program} \(p\) is to compute a partial function \([p]_{\text{\textsc{while}}} : \mathcal{D} \rightarrow \mathcal{D}_\perp\), where \(\mathcal{D}_\perp\) abbreviates \(\mathcal{D} \cup \{\perp\}\). Control structures are \textit{sequential composition} \(C1; C2\), the \textit{conditional} \(\text{if } E \text{ then } C1 \text{ else } C2\), and the \textit{while loop} \(\text{while } E \text{ do } C\). In tests, nil serves as “false,” and anything else serves as “true.” A formal definition of the semantics:

\textbf{Definition 3.2} Consider a \textsc{while} program \(p\) of form \(\text{read } X; \ C; \text{ write } Y\). Let \(\text{Vars}(p) = \{X, \ldots, Y\}\) be the set of all variables in \(p\).

\begin{enumerate}
\item A \textit{store} \(\sigma\) for \(p\) is by definition a function from \(\text{Vars}(p)\) to elements of \(\mathcal{D}\).
More generally, \(\text{Store}^p = \text{Vars}(p) \rightarrow \mathcal{D}\) is the set of all stores for \(p\).
\item The \textit{initial store} \(\sigma_0^p(d)\) for inputs \(d \in \mathcal{D}\) binds \(X\) to \(d\) and all else to nil:
\[\sigma_0^p(d) = [X \mapsto d, Z \mapsto \text{nil}, \ldots, Y \mapsto \text{nil}]\]
\end{enumerate}

Figure 3.2 contains the three semantic functions \(E, C,\) and \(P\) (\(C\) and \(P\) are partial) for \textsc{while} programs, with types

6
\[ \mathcal{E}[\mathbf{\times}]_\sigma = \sigma(\mathbf{x}) \]
\[ \mathcal{E}[\mathbf{d}]_\sigma = \mathbf{d} \quad \text{if } \mathbf{d} \in \mathcal{D} \]
\[ \mathcal{E}[[\text{cons } E1 \ E2]]_\sigma = (\mathbf{d}_1, \mathbf{d}_2) \quad \text{if } \mathcal{E}[\mathbf{E1}]_\sigma = \mathbf{d}_1 \text{ and } \mathcal{E}[\mathbf{E2}]_\sigma = \mathbf{d}_2 \]
\[ \mathcal{E}[[\text{hd } E]]_\sigma = \mathbf{d}_1 \quad \text{if } \mathcal{E}[\mathbf{E}]_\sigma = (\mathbf{d}_1, \mathbf{d}_2) \]
\[ \mathcal{E}[[\text{hd } E]]_\sigma = \text{nil} \quad \text{otherwise} \]
\[ \mathcal{E}[[\text{tl } E]]_\sigma = \mathbf{d}_2 \quad \text{if } \mathcal{E}[\mathbf{E}]_\sigma = (\mathbf{d}_1, \mathbf{d}_2) \]
\[ \mathcal{E}[[\text{tl } E]]_\sigma = \text{nil} \quad \text{otherwise} \]
\[ \mathcal{E}[[\mathbf{X} := E]]_\sigma = \sigma[\mathbf{X} \mapsto \mathbf{d}] \quad \text{if } \mathcal{E}[\mathbf{E}] = \mathbf{d} \]

\[ C[[\mathbf{C1 ; C2}]]_\sigma = \sigma' \quad \text{if } C[[\mathbf{C1}]]_\sigma = \sigma'' \text{ and } C[[\mathbf{C2}]] = \sigma'' \]
\[ C[[\text{if } E \text{ then } C1 \text{ else } C2]]_\sigma = \sigma' \quad \text{if } \mathcal{E}[\mathbf{E}]_\sigma \neq \text{nil} \text{ and } C[[\mathbf{C1}]]_\sigma = \sigma' \]
\[ C[[\text{if } E \text{ then } C1 \text{ else } C2]]_\sigma = \sigma' \quad \text{if } \mathcal{E}[\mathbf{E}]_\sigma = \text{nil} \text{ and } C[[\mathbf{C2}]]_\sigma = \sigma' \]
\[ C[[\text{while } E \text{ do } C]]_\sigma = \sigma \quad \text{if } \mathcal{E}[\mathbf{E}]_\sigma = \text{nil} \]
\[ C[[\text{while } E \text{ do } C]]_\sigma = \sigma' \quad \text{if } \mathcal{E}[\mathbf{E}]_\sigma = (\mathbf{d}_1, \mathbf{d}_2) \]
\[ \text{and } C[[\mathbf{C}]]_\sigma = \sigma'' \]
\[ \text{and } C[[\text{while } E \text{ do } C]]_\sigma = \sigma'' \]

\[ \mathcal{P}[[\text{read } \mathbf{X}; \ C; \ write \ \mathbf{Y}]](\mathbf{d}) = \mathbf{d}' \quad \text{if } C[[\mathbf{C}]](\sigma^P_\delta(\mathbf{d})) = \sigma(\mathbf{Y}) \]

\textbf{Figure 3.2:} Semantics of WHILE programs.

\[ \mathcal{E} : \text{Expression} \rightarrow (\text{Store}^P \rightarrow \mathcal{D}) \]
\[ C : \text{Command} \rightarrow (\text{Store}^P \rightarrow \text{Store}^P) \]
\[ \mathcal{P} : \text{Program} \rightarrow (\mathcal{D}^n \rightarrow \mathcal{D}_\bot) \]

Function \( \mathcal{E} \) evaluates expressions. Given a store \( \sigma \) containing the values of the variables in an expression \( E \), \( \mathcal{E} \) maps \( E \) and \( \sigma \) into the value \( \mathcal{E}[\mathbf{C}]_\sigma \) in \( \mathcal{D} \) that \( E \) denotes. Suppose command \( C \) has a number of assignments that alter the store. Given a store \( \sigma \), function \( C \) maps the command and the current store into a new store \( C[[\mathbf{C}]]_\sigma = \sigma' \in \text{Store}^P \). If command \( C \) does not terminate on the given store \( \sigma \), then \( C[[\mathbf{C}]]_\sigma \) is undefined, written: \( C[[\mathbf{C}]]_\sigma = \bot \). Finally, \( \mathcal{P} \) maps a program and given value \( \mathbf{d} \) for the input variables into a value \( \mathcal{P}[[\mathbf{p}]](\mathbf{d}) = C[[\mathbf{C}]](\sigma^P_\delta(\mathbf{d})) \) in \( \mathcal{D} \) if the program terminates, else \( \bot \). The meaning of a program is written as \( \mathcal{P}[[\mathbf{p}]] : \mathcal{D} \rightarrow \mathcal{D}_\bot \), and \( \mathcal{P}[[\mathbf{p}]] \) will also be written as \( [[\mathbf{p}]]^\text{WHILE} \), or even \( [[\mathbf{p}]] \) if the language is clear from context.

For brevity, running time is only informally defined. A later section on implementation justifies the reasonability of this definition—natural with the data-sharing implementation techniques used in Lisp and other functional languages.

\textbf{Definition 3.3} The running time \( \text{time}_\mathbf{p}(\mathbf{d}) \in \{\bot, 0, 1, 2, \ldots\} \) is obtained by counting 1 every time any of the following is performed while computing \( [[\mathbf{p}]](\mathbf{d}) \).
as defined in the semantics: a variable or constant reference; an operation hd, tl, cons, or := is applied; or a test in an if or while command. Its value is ⊥ if the computation does not terminate. □

3.1.3 The GOTO variant of WHILE

The WHILE language has both “structured” syntax and data. This is convenient for programming, but when constructing one program from another it will often be convenient to use a lower-level “flow chart” syntax in which a program is a sequence $p = 1:i1\ 2:i2\ \ldots\ m:im$ of labeled instructions, executed serially except as redirected by control transfers. Instructions are of the form $X:=n\!\!il, X:=Y, X:=\text{cons}\ Y\ Z, X:=\text{hd}\ Y, X:=\text{tl}\ Y$, or if $X$ goto $\ell$ else $\ell$. (Note that the syntax of GOTO expressions is significantly more limited than that of WHILE expressions.)

The semantics is natural and so not presented here. Following is the GOTO equivalent of the reverse program above.\footnote{Where goto 1 abbreviates if $X$ goto 1 else 1.}

0: read $X$;
1: if $X$ goto 2 else 6
2: $Z:=\text{hd}\ X$;
3: $Y:=\text{cons}\ Z\ Y$;
4: $X:=\text{tl}\ X$;
5: goto 1;
6: write $Y$

It is easy to see that any WHILE program can be translated into an equivalent GOTO program running at most a constant factor slower (measuring GOTO times by the number of instructions executed). Conversely, any GOTO program can be translated into an equivalent WHILE program running at most a constant factor slower (the factor may depend on the size of the GOTO program).

3.2 Off-line Turing machines

A TM-program is a traditional off-line Turing machine with a two-way read-only input tape, and a two-way read-write work tape. By definition TM-data = $\{0,1\}^*$. Since our aims concern only decision powers and not computation of functions, a Turing machine output will be a single bit 0 or 1. Extension to outputs in $\{0,1\}^*$ is routine, by a one-way write-only output tape.

A TM-program is a sequence $p = 1:i1\ 2:i2\ \ldots\ m:im$. The instructions I$\ell$
are as follows, where subscript 1 indicates that the input tape 1 is involved; or 2 indicates that work tape 2 is involved. Instruction syntax:

Tape 1: \( I := \text{right}_1 | \text{left}_1 | \text{if}_1 \ S \ \text{goto} \ \ell \ \text{else} \ \ell \)
Tape 2: \( I := \text{right}_2 | \text{left}_2 | \text{if}_2 \ S \ \text{goto} \ \ell \ \text{else} \ \ell | \text{write}_2 \ S \)
Symbols: \( S := 0 | 1 | \mathcal{B} \)
Strings: \( L,R := \varepsilon | L \ S \)

A tape together with its scanning position will be written as \( L_1 \underline{S}_1 R_1 \), where the underline indicates the scanned position. As usual the tape is extensible — a new blank appears when a move is made beyond the end of the tape. A \emph{total state} is a triple \( s = (\ell, L_1 \underline{S}_1 R_1, L_2 \underline{S}_2 R_2) \) whose first component \( \ell \) is the number of the instruction about to be executed, and whose second and third components describe both of the tapes, and underlines mark their scanning positions.

The semantics of the individual instructions is a state transition relation \( s \to s' \) (actually a function), as is usual for Turing machines. For example, instruction 1: \( \text{right}_2 \) causes transition from state \( (1, \underline{B}10, B0111B) \) to \( (2, \underline{B}10, B0111B) \), or from \( (1, \underline{B}10, B01) \) to \( (2, \underline{B}10, B01\underline{B}) \). We assume the program never attempts to move right or left beyond the blanks that delimit the input, unless a nonblank symbol has first been written.\(^5\)

A \emph{computation} on input \( d \in \{0,1\}^* \) is a sequence \( s_0 \to s_1 \to \ldots \to s_n \) where \( s_i \to s_{i+1} \) for \( 0 \leq i < n \), and \( s_0 \) equals \( (1, \underline{B}d, \underline{B}) \), and \( s_n \) has instruction \( m+1 \) as first component (the “program end”). The output for input \( d \) is defined by:

\[
[p](d) = \begin{cases} 
\bot & \text{if } p \text{ has no computation on input } d, \text{ else} \\
1 & \text{if } p\text{'s final state for input } d \text{ scans worktape symbol } 1 \\
0 & \text{otherwise}
\end{cases}
\]

We define the \emph{space usage} of a total state \( s = (\ell, L_1 \underline{S}_1 R_1, L_2 \underline{S}_2 R_2) \) by \( |s| = |L_2 \underline{S}_2 R_2| \), formally expressing that only the symbols on “work” tape 2 are counted, and not those on tape 1. Finally, we define

\[
\text{space}_p^w(d) = \begin{cases} 
\max\{|s_i|\} & \text{if } s_0 \to \ldots \to s_n \text{ is } p\text{'s computation on } d \\
\bot & \text{if } p \text{ does not terminate on input } d
\end{cases}
\]

\(^5\) This condition simplifies constructions, and causes no loss of generality in computational power, or increase in time beyond a constant factor.
The class LOGSPACE plays a central role in complexity theory, as it is the beginning of the much-studied hierarchy \( \text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PSPACE} \subseteq \ldots \). Further, all known reductions used to show familiar combinatorial problems complete for these classes can be carried out by logspace-bounded Turing machines.

LOGSPACE is in a sense the smallest natural complexity class, because a machine needs at least logarithmic storage in order to “remember” a position in its input string or to count a number of input symbols (for instance, to decide whether its input has the form \( 0^n1^n0^n \)).

Despite the naturality and centrality of the class LOGSPACE, its Turing machine definition seems artificial due to hardware restrictions on the number of tapes, their usage, and the external restriction on the run-time length of the work tape. A somewhat more natural definition comes from a “folklore theorem”: that logspace Turing machines have the same decision power as two-way multihead finite automata (read-only) — machines that can “see but not touch” their input.

In this section we give a simpler and still more natural “look but not touch” characterization: LOGSPACE is exactly the set of problems decidable by read-only WHILE programs. More formally, let \( \text{WHILE}^{\text{cons}} \) be the language identical to \( \text{WHILE} \) but restricted to programs not containing \text{cons}. We will prove the following:

**Theorem 4.1** \( \text{WHILE}^{\text{cons}} \equiv \text{TM}^{\text{logspace}} \)

**4.1 About the simulations**

**A bijection between \( \{0,1\}^* \) and a subset of \( \mathcal{L} \).** As defined, Turing machine and WHILE inputs are not the same. We resolve this by restricting \( \mathcal{L} \) to a subset \( \mathcal{L}_{01} \) isomorphic with \( \{0,1\}^* \). Define the coding \( c : \{0,1\}^* \rightarrow \mathcal{L} \) by \( c(a_1a_2\ldots a_n) = (a'_1 a'_2 \ldots a'_n) \) where \( 0' = \text{nil} \) and \( 1' = (\text{nil.nil}) \). Define \( \mathcal{L}_{01} \) to be the range of \( c \). An example using Lisp list notation:

\[
c(001) = (0'0'1') = (\text{nil nil (nil.nil)})
\]

**Relating WHILE and Turing machine states.** Given a WHILE program, a Turing machine simulating it will store on its work tape the pointer values of all the WHILE variables. Conversely, given a Turing program, a WHILE program
simulating it will encode, by means of pointer values, the current contents of
the Turing machine's work tape. We first analyze their numbers of states.

**Turing machine**: Let \( p \) have \( m \) instructions, work tape storage bound \( k \log n \),
and input \( a_1 a_2 \ldots a_n \in \{0, 1\}^* \). Then \( p \) can enter at most
\[
m(n + 2)3^{k \log n} = m(n + 2)n^{k \log 3}
\]
different total states. For fixed \( p \) this number is \( O(n^{1+k \log 3}) \).

**Read-only WHILE program**: Let \( q \) have \( m' \) instructions, \( k' \) variables, and
input list \( (a_1 a_2 \ldots a_n) \in \mathcal{I}_0 \). During the computation every \( q \)-variable \( X \) value
must be a pointer to one of:

(i) The root of a suffix \( (a_i a_{i+1} \ldots a_n) \) at a position \( i \) along the "spine"
    of the input list; or
(ii) The root of \( (\text{nil. nil}) \), the coding \( c(1) \) of some \( a_i = 1 \); or
(iii) The atom \( \text{nil.} \).

Thus each variable can take on at most \( n + 2 \) values, so the number of program
total states is bounded by: \( m'(n + 2)^{k'} \). This is also a polynomial in \( n \), giving
hope for Theorem 4.1 since the programs on either side of \( \equiv \) have comparable
numbers of states for a given input.

### 4.2 LOGSPACE simulation of read-only programs

**Lemma 4.2** \( \text{WHILE}^{\text{cons}} \preceq \text{TM}^{\text{logspace}} \)

**Proof** Let \( q = 1: \text{I1} \ 2: \text{I2} \ldots \ m: \text{Im} \) be the GOTO version of a read-only WHILE
program. Its instructions are of the form \( X := \text{nil}, X := Y, X := \text{hd} Y, X := \text{tl} Y, \) or if \( X \) goto \( \ell \) else \( \ell' \).

The input to \( q \) is a list \( (a_1 a_2 \ldots a_n) \in \mathcal{I}_0 \) corresponding to string \( a_1 a_2 \ldots a_n \)
in \( \{0, 1\}^* \). Possible variable values in any \( q \) state have been analyzed above.
Construct an off-line Turing machine \( p \) to simulate \( q \), as follows. The idea is
to represent each variable \( X \) of \( q \) by its position and its tag numbers \( (p_X, t_X) \)
with values \((i, 0)\) in case (i), and \((0, 1)\) in case (ii), and \((0, 0)\) in case (iii).
Turing machine program \( p \) stores each pair \( (p_X, t_X) \) on its work tape in binary,

---

6 A value of \( \text{nil} \) can arise 3 ways, but the effects while executing \( p \) are the same:
either it is the coding of some \( a_i = 0 \); or the \( \text{nil} \) at the end of the input list; or it
is the head or tail of \( c(1) = (\text{nil.nil}) \).

7 We assume \( n > 0 \); special case code gives the correct answer for \( n = 0 \).
thus using at most \(O(\log n)\) bits for all of \(q\)'s variables. \textsc{goto} command I is simulated by Turing commands achieving the following effects:

<table>
<thead>
<tr>
<th>\textsc{goto} command I</th>
<th>Effect of corresponding Turing code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X := \text{nil})</td>
<td>((p_X, t_X) := (0, 0))</td>
</tr>
<tr>
<td>(X := Y)</td>
<td>((p_X, t_X) := (p_Y, t_Y))</td>
</tr>
<tr>
<td>(X := \text{hd}\ Y)</td>
<td>((p_X, t_X) := \text{if } p_Y &gt; 0 \land a_{p_Y} = 1 \text{ then } (0, 1) \text{ else } (0, 0))</td>
</tr>
<tr>
<td>(X := \text{tl}\ Y)</td>
<td>((p_X, t_X) := \text{if } p_Y &gt; 0 \text{ then } (p_Y - 1, t_Y) \text{ else } (0, 0))</td>
</tr>
<tr>
<td>if (X) \text{ goto } \ell\ else \ell' if (t_X = 1) or ((p_X &gt; 0 \land a_{p_X} = 1)) then goto (\ell) else (\ell')</td>
<td></td>
</tr>
</tbody>
</table>

All is straightforward Turing programming: the test "\(a_{p_Y} = 1\)" is done by scanning the Turing input tape \(a_0 a_1 \ldots a_n\) left to right, until \(p_Y\) symbols have been seen. It is easy to see that this is a faithful simulation that preserves the representation of the variables in \textsc{goto} program \(q\). \(\square\)

### 4.3 Counter machines

The converse proof is a bit more subtle, as \(O(\log n)\) bits of Turing machine work storage must be encoded using \textsc{while} variables whose values are positions on the input string, and which can only be advanced forward (\(X := \text{tl}\ X\)). This will be done using the counter machine \(\mathcal{CM}\) as an intermediate computation model, in the following stages (\(\mathcal{CM}^{\text{poly}}\) and \(\mathcal{CM}^{\uparrow}\) are defined below):

\[
\text{\textsc{while}}^{\text{cons}} \preceq \text{TM}^{\text{logspace}} \preceq \mathcal{CM}^{\text{poly}} \preceq \mathcal{CM}^{\uparrow} \preceq \text{\textsc{while}}^{\text{cons}}
\]

**Definition 4.3** A program in the language \(\mathcal{CM}\) of counter machines is a sequence of labeled instructions \(p = 1: I_1 \ 2: I_2 \ldots m: I_m\) of the following forms for \(0 < i\) and \(0 \leq j\) (so counter \(C_0\) is never assigned).

\[
I \ := \begin{cases} C_i := C_i + 1 & | \text{ if } C_j = 0 \text{ goto } \ell \text{ else } \ell' \ \\
C_i := C_i - 1 & | \text{ if } I_{C_j} = 0 \text{ goto } \ell \text{ else } \ell'
\end{cases}
\]

Storage has form \((d, \sigma) \in \{0, 1\}^* \times (\mathbb{N} \rightarrow \mathbb{N})\) where \(d\) is the input data (read-only) and \(\sigma(i)\) is the current contents of counter \(C_i\) for any \(i \in \mathbb{N}\). Input to a counter machine is a string \(d\) in \(\{0, 1\}^*\). Data initialization sets counter \(C_0\) to \(n\), giving the program a way to "know" how long its input is. The counter values \(\sigma\) are initialized to zero except for \(C_0\): initially,

\[
\sigma_0(d) = [0 \mapsto |d|, 1 \mapsto 0, 2 \mapsto 0, \ldots]
\]

A state for input \(d\) has form \(s = (\ell, (d, \sigma))\), where \(\ell\) is the instruction counter. Input access is by instruction if \(I_{C_0} = 0 \text{ goto } \ell \text{ else } \ell'. Its effect: If \(1 \leq \sigma(i) \leq n\) and \(a_{\sigma(i)} = 0\) then control is transferred to \(I_\ell\), else to \(I_\nu\). The
remaining instructions behave as expected from the syntax, except that we define \( 0 - 1 = 0 \) to avoid negative integers.

Thus \( p \) defines a one-step transition relation \( s \rightarrow s' \). A computation on input \( d \in \{0,1\}^* \) is a sequence \( s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n \) where each \( s_i \) yields \( s_{i+1} \), and \( s_0 \) equals \((1, (d, \sigma_0(d)))\), and \( s_n \) has instruction \( m+1 \) as first component (the "program end"). The output for input \( d \) is defined by:

\[
[p](d) = \begin{cases} 
\bot & \text{if } p \text{ has no computation on input } d, \text{ else} \\
1 & \text{if } p's \text{ final state for input } d \text{ has 1 in counter } C_1 \\
0 & \text{otherwise}
\end{cases}
\]

Since our aims concern only decision powers and not computation of functions, a counter machine output will be a single bit 0 or 1. Extension to outputs in \( \{0,1\}^* \) is routine, by a one-way write-only output tape. \( \square \)

We now introduce some restrictions on counter machines.

**Definition 4.4** A \( \mathcal{CM} \) program \( p \) is \( f(n) \)-bounded if for any computation \( s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n \) on input \( d \), no counter in any state \( s_i \) exceeds \( f(|d|) \).

The language \( \mathcal{CM}^{poly} \) is identical to \( \mathcal{CM} \), except that \( \mathcal{CM}^{poly} \)-programs is the subset of all \( \mathcal{CM} \) programs \( p \) that are \( f(n) \)-bounded for some polynomial \( f(n) \).

The language \( \mathcal{CM}^+ \) is identical to \( \mathcal{CM} \), except that \( \mathcal{CM}^+ \) programs contains no instructions of form \( C_i := C_{i+1} \). \( \square \)

Note that \( \mathcal{CM}^+ \) program is a \( \mathcal{CM}^{poly} \) program. In fact, it is \( n \)-bounded, since all counters must remain less than or equal to the input length \( n \).

### 4.4 Read-only simulation of \( \text{LOGSPACE} \) programs

**Lemma 4.5** \( \text{TM}^{\text{logspace}} \preceq \mathcal{CM}^{poly} \) \( \square \)

**Proof** Let \( p \) be a \( \text{TM} \) program that uses space at most \( k \log n \) where \( n \) is the length of its input. We will show how to simulate \( p \) by a polynomially bounded counter machine. A total state of \( p \) is \( s = (\ell, L_1 \underline{S_1} R_1, L_2 \underline{S_2} R_2) \) where \( \ell \) is the instruction counter. The input tape is read-only and identical to the counter machine's input, so the contents of tape 1 need not be represented by counter \( s \); its scanning position is sufficient.

The simulation represents \( p \)'s work tape contents by two polynomially bounded counters, and simulates operations on either tape by corresponding counter operations. The scanning positions on tapes 1, 2 can be represented by counters.
no larger than \(n + 2\) or \(k \log n\), respectively. Both are certainly polynomially bounded. A work tape containing \(b_1 \ldots b_r \ldots b_m\) can be represented by a pair of numbers \(l, r\), where

- \(l\) is the value of \(b_1 \ldots b_i\) as a base 3 number (counting \(B\) as digit 0, 0 as digit 1, and 1 as digit 2), and
- \(r\) is the value of \(b_m b_{m-1} \ldots b_{i+1}\) (note the reversal), also as a base 3 number.

The work tapes are initially all blank, so \(l = r = 0\) at the simulated computation's start. Since \(m \leq k \log n\), we have \(l, r \leq 3^{k \log n} = n^{k \log 3}\). Thus altogether we have two counters to represent the input and work tape scanning positions, and two counters to represent the work tape contents. These counters are all bounded in size by \(n + 2\) or \(n^{k \log 3}\), and collectively represent the Turing machine's total state. Each Turing machine operation can be faithfully simulated by operations on counters. For example, the effect of moving a work tape head right one position is to replace \(l\) by \(3 \cdot l + (r \mod 3)\), and to divide \(r\) by 3. It is easy to see that these operations can be done by counters. Testing the scanned square's contents amounts to a test on \(l \mod 3\), also easily done. □

**Lemma 4.6** \(CM_{\text{poly}} \subseteq CM^{\text{\textbackslash}+}\)

**Proof**

**Counters bounded by** \(n\): Recall that counter \(C_0\) is initialized to the length \(n\) of the input, and never re-assigned. We first show that any \(n\)-bounded \(CM\) program \(p\) is equivalent to some program \(q\) without \(C := C + 1\). All counters are by assumption bounded by \(n\), so we need not account for “overflow.” First, the effect of \(D := n - C\) can be realized by the following:

\[
D := C_0; \ E := C; \ \text{while} \ E \neq 0 \ \text{do} \ \{E := E-1; \ D := D-1\};
\]

We can now simulate \(C := C + 1\) by:

\[
D := n - C; \ D := D - 1; \ C := n - D
\]

**Polynomial-bounded programs**: Now let \(p\) be an \(n^b\)-bounded \(CM\) program (extension to a general polynomial is straightforward). We show how to replace any variable \(C\) in \(p\) by a collection of \(b + 1\) variables, each bounded by \(n\). The idea is simple: consider the base \(n + 1\) representation \(k_b \ldots k_1 k_0\) of a number \(k \leq n^b\) and represent \(C\) by \(b + 1\) register variables \(C_{b}, \ldots, C_1, C_0\).

Using this, the familiar algorithms for base \(n + 1\) addition and subtraction are easy to program, using \(C_0\) to to recognize when “overflow” and “borrowing” occur. The test if \(inc = 0\) goto \(\ell\) else \(\ell'\) is realized by transferring to \(\ell'\) if any of \(C_b, \ldots, C_1\) are nonzero, and otherwise performing if \(inc = 0\) goto \(\ell\) else \(\ell'\). □
4.5 Read-only simulation of counter machines

Lemma 4.7 \( CM^{+} \preceq \text{WHILE}^{\text{cons}} \)

Proof An arbitrary \( CM^{+} \) program \( p \) must be shown equivalent to some read-only \( \text{WHILE} \) program. We do this by constructing a simulating program in \( GOTO \) form. Input to \( p \) is a string \( a_1a_2...a_n \), corresponding to \( \text{WHILE} \) input list \( (a'_1a'_2...a'_n) \in D_{01} \). Each \( a'_i \) is \( \text{nil} \) if \( a_i = 0 \), else \( (\text{nil.nil}) \).

The counters \( C_i \) of \( p \) can only assume values between 0 and \( n \). Represent each counter \( C_i \) by a corresponding \( GOTO \) program variable \( X_i \), and represent the value of \( C_i \) by the distance from \( X_i \) to the end of the \( \text{WHILE} \) program's input. This gives the following simulation invariant:

<table>
<thead>
<tr>
<th>Value of ( C_i )</th>
<th>Value of ( X_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \text{nil} = () )</td>
</tr>
<tr>
<td>( k &gt; 0 )</td>
<td>( (a'_{n-k+1}...a'_n) )</td>
</tr>
</tbody>
</table>

Counter command \( C_i := C_j \) can obviously be simulated by \( X_i := X_j \). Clearly \( C_i \) has value 0 if and only if \( X_i \) has value \( \text{nil} \). Thus command if \( C = 0 \) goto \( \ell \) else \( \ell' \) can be simulated by if \( X_i \) goto \( \ell \) else \( \ell' \). Further, instruction \( C_i := C_i - 1 \) can be realized by \( X := \text{tl} \ X \), reducing the distance to the end by 1. (This works for value 0 since the head of \( \text{nil} \) is \( \text{nil} \).)

If \( C_i \) has value \( k \) then command if \( \text{inc} \) \( C_i = 0 \) goto \( \ell \) else \( \ell' \) tests symbol \( a_k \). It can be simulated by the following (informal) code:

\[
\begin{align*}
\text{if } X_i = \text{nil } & \text{ goto } \ell; \quad \text{(* if } C_i = 0 \text{ goto } \ell \ast) \\
Z := \text{tl} X_i; \ Y := X_0; \quad \text{(* } Y := (a'_1...a'_n) \ast) \\
\text{while } Z \text{ do } & \quad \text{(* At the loop end, } Y = (a'_1...a'_n) \ast) \\
\{} & \quad Y := \text{tl} Y; \ Z := \text{tl} Z \}; \\
\text{if } \text{hd} \ Y \text{ goto } \ell' \ & \text{ else } \ell
\end{align*}
\]

If \( C_i \) has value \( k \) then \( X_i \) has value \( (a'_{n-k+1}...a'_n) \), with \( k \) items. The while loop removes \( k - 1 \) items from the start of input copy \( Y \) (initial value \( (a'_1a'_2...a'_n) \)), leaving \( Y \) with value \( (a'_ka'_{k+1}...a'_n) \). The head of this is \( \text{nil} \) if and only if \( a_k = 0 \). □

Proof of Theorem 4.1. Lemmas 4.2, 4.7, 4.6, and 4.5 establish:

\[
\text{WHILE}^{\text{cons}} \preceq \text{TM}^{\text{logspace}} \preceq \text{CM}^{\text{poly}} \preceq \text{CM}^{+} \preceq \text{WHILE}^{\text{cons}}
\] □
Simulating recursive read-only programs in polynomial time

The main result of the rest of the paper is the following:

**Theorem 5.1** $\text{WHILE}^{\text{rec}} \setminus \text{cons} \equiv \text{WHILE}^{\text{time}}$

This section establishes inclusion one way: that problems solvable by recursive read-only programs can be decided in polynomial time.

**Implicit storage: the recursion stack.**

**Definition 5.2** Suppose L is the programming language WHILE, CM, or one of these restricted to a subset of programs (for instance, $\text{CM}^{\text{poly}}$). Thus any L-program has form $p = 1: I_1 2: I_2 \ldots m: I_m$.

The *recursive extension* $L^{\text{rec}}$ is defined so $L^{\text{rec}}$-programs consists of all programs with syntax as in Figure 5.1, where each instruction In, Jn or Kn can be either:

- call Pr for some procedure Pr; or
- Any L-instruction, with two restrictions to avoid cross-procedural data references or gotos. First: in each procedure Pi, any referenced variable $X$ must satisfy $X \in \{U_1, \ldots, U_u, P_1, P_2, \ldots\}$, i.e. it must be either local or global. Second, the label $\ell$ in instruction goto $\ell$ refers to the procedure containing the instruction.

```
globalvariables U_1, \ldots, U_u;

procedure P_1; localvariables P_1, \ldots, P_{1v};
    1: I_1 2: I_2 \ldots i: I_i i+1:

procedure P_2; localvariables P_1, \ldots, P_{2w};
    1: J_1 2: J_2 \ldots j: J_j j+1:

procedure P_m; localvariables P_{m1}, \ldots, P_{mx};
    1: K_1 2: K_2 \ldots k: K_k k+1:

read U_1; 1: call P_1; 2: write U_1
```

*Figure 5.1: Recursive program syntax.*

A *store* $\sigma$ is a binding of variables to values. It is *global* if it binds $U_1, \ldots, U_u$, and local if it binds the local variables of some procedure $P_i$. A *total state* is
a stack (with the topmost element on the left) of form.  

\((\ell_0, \sigma_0, \ell_1, \sigma_1, \ldots, \ell_n, \sigma_n, \text{exit})\)

**Control:** each \(\ell_i\) is a label either in \(p\)'s “main program” or in one of its procedures. Label \(\ell_0\) labels the instruction about to be executed (perhaps a procedure return), \(\ell_1, \ldots, \ell_n\) are return addresses, and \text{exit} indicates program termination.

**Storage:** the last store \(\sigma_n\) always contains the global variable bindings (values of \(U_1, \ldots, U_n\)), and \(\sigma_0\) contains the local variable bindings of the most recently called procedure. The other \(\sigma_0, \ldots, \sigma_{n-1}\) contain local variable values of earlier procedures that have been called but not yet returned from. Variable fetches and assignments are done using only \(\sigma_n\) and \(\sigma_0\).

**Semantics:** this is as usual for imperative languages with recursion, and so is only briefly described. The initial state for input \(d\) is

\[(1, [U_1 \mapsto d, U_2 \mapsto DV, \ldots, U_n \mapsto DV], \text{exit})\]

where \(DV\) is an appropriate default value (nil for a tree value, or 0 for a counter value). Instruction \(\ell_0: \text{call} \ P\) pushes a new stack frame in place of \(\ell_0\), yielding

\[(1, \sigma_{\text{new}}, \ell_0 + 1, \sigma_0, \ell_1, \sigma_1, \ldots, \ell_n, \sigma_n, \text{exit})\]

Here \(1\) is the new procedure's initial control point, and \(\sigma_{\text{new}}\) assigns default values to all of \(P\)'s local variables. Thus label \(\ell_0 + 1\) plays the role of “return address” (or \text{exit} for the initial call.)

When a procedure's last instruction has been executed, the leftmost label and store \(\ell_0, \sigma_0\) are popped off, and control is transferred to the instruction whose label is on the stack top. This yields total state

\[(\ell_0 + 1, \sigma_0, \ell_1, \sigma_1, \ldots, \ell_n, \sigma_n, \text{exit})\]

\[\Box\]

\(^8\) In compiler jargon this is the familiar “call stack.” Each \(\sigma_i\) is a “stack frame” containing the local variable values of the currently called procedure. Label \(\ell_i\) for \(i > 1\) is the “return address,” to which control will be passed after the current procedure’s execution terminates.
Memoization: using explicit storage to simulate implicit storage.
Suppose we are given an L-recc program $p$ as in Figure 5.1. We outline informally a memoizing simulation of $p$ on input $d$. The idea is that it records results of procedure calls as they are simulated. The underlying principle (due to Cook [3]) is that the net effect of a call to procedure $P$ (if it terminates) is to transform the values in global store $\sigma_n$ into a new global store $\sigma_n'$.

If procedure $P$ is called a second time with the same global store $\sigma_n$, it will, of course, have the same effect; so the simulator can take a “short cut,” directly changing its global store $\sigma_n$ into $\sigma_n'$ without bothering to simulate $P$ again. This can lead to major time savings.

Construction. While the individual instructions of $p$ are being simulated, a table $TAB$ of triples of form $(P, \sigma, \sigma')$ will be built ($TAB$ stands for a “tabulation” of call effects). Entry $(P, \sigma, \sigma') \in TAB$ signifies that procedure $P$ has been called from a state with global store $\sigma$, and that it terminated with global store $\sigma'$.

Thus $TAB$ will be used in two ways: to “archive” a triple $(P, \sigma, \sigma')$ whenever a fact $P(\sigma) = \sigma'$ has been established; and to retrieve from the archive the final effect of a procedure call which has earlier been simulated.

Program $p$ is then simulated as follows: First, set $TAB = \emptyset$ and initialize the total state (as above) to $(\ell_0, \sigma_0, \textit{exit})$ where $\sigma_0 = [u_1 \mapsto d, u_2 \mapsto DV, \ldots, u_u \mapsto DV]$. Then repeat the following until termination occurs (if it does) on the basis of the current total state

$$(\ell_0, \sigma_0, \ell_1, \sigma_1, \ldots, \ell_n, \sigma_n, \textit{exit})$$

(i) Non-call: $\ell_0$ labels a non-call instruction $I$. Then perform $I$ as usual for L programs. This may increment or change $\ell_0$, and change the local or global stores $\sigma_0, \sigma_n$.

(ii) First call: $\ell_0$ labels an instruction call $P$ and $TAB$ contains no triple $(P, \sigma_n, \sigma)$. Then change the total state to the following.\footnote{This is exactly as for a normal call except for the components $1, \sigma_n$: an extra copy of the global store has been added, with dummy label 1 to preserve the format.} Note that execution of $P$'s body will be done exactly as in the standard semantics; the first difference will appear on returning from $P$.

$$(1, \sigma_{new}, 1, \sigma_n, \ell_0 + 1, \sigma_0, \ell_1, \sigma_1, \ldots, \ell_n, \sigma_n, \textit{exit})$$

(iii) Later call (short-cut): $\ell_0$ labels an instruction call $P$ and $(P, \sigma_n, \sigma_n') \in TAB$. Then change the total state to

$$(\ell_0 + 1, \sigma_0, \ell_1, \sigma_1, \ldots, \ell_n, \sigma_n', \textit{exit})$$

18
and continue the simulation.

(iv) Procedure return: \( \ell_0 \) is at the end of procedure \( P \), and the total state is of form

\[
(\ell_0, \sigma_{new}, 1, \sigma, \ell_0 + 1, \sigma_0, \ell_1, \sigma_1, \ldots, \ell_n, \sigma', \text{exit})
\]

Then add triple \((P, \sigma, \sigma')\) to \( TAB \), change the total state to the following, and continue the simulation. Note the the overall effect of a call is exactly the same in both the original and the simulated versions.

\[
(\ell_0 + 1, \sigma_0, \ell_1, \sigma_1, \ldots, \ell_n, \sigma_n, \text{exit})
\]

In summary: when simulating a first call to \( P \), the above saves a copy of the global store on the stack top. On return from a first call simulation, the previous global store \( \sigma \) and the final global store \( \sigma' \) after the call are both available (\( \sigma \) was saved in step ii above). Using these, the triple \((P, \sigma, \sigma')\) is tabulated.

The same technique can be applied to nondeterministic programs; but table \( TAB \) then becomes a relation rather than a function, and the proof below needs modification.

**Lemma 5.3** Suppose \( L^e \) program \( p \) terminates on all inputs, and enters at most \( f(|d|) \) different local or global stores during the computation of \( [p](d) \). Then the simulation described above of \( p \) on \( d \) can be carried out in time \( O(f(|d|)^2) \).

**Proof** First, the table \( TAB \) is initially empty, and an entry is added each time a procedure return is processed. This table can contain at most \( O(f(|d|)^2) \) entries, since each consists of a procedure name (constant with respect to \( d \)) and two stores.\(^{10}\)

Further, no entry \((P, \sigma, \sigma')\) is ever added twice to \( TAB \). To see this, consider the first time procedure \( P \) is called with global store \( \sigma \). We assumed program \( p \) terminates, so the call to \( P \) will exit; and \( P \) is not called again with the same store \( \sigma \) within this call (else it would have looped). Consequently a triple \((P, \sigma, \sigma')\) will be added to \( TAB \), for the first time, on the first exit from \( P \). If any subsequent call to \( P \) occurs with the same global store, the “short-cut” will be taken without effect on \( TAB \).

Consequently **no** \( p \) instruction is ever simulated twice with the same global store, so the total number of instruction simulations carried out is proportional to the number of global stores: \( O(f(|d|)) \). For each the maximum time needed is the time to search or update \( TAB \), which can be done in time \( O(f(|d|)) \).

\(^{10}\) The number is actually \( O(f(|d|)) \), since \( \sigma \) determines \( \sigma' \).
(in fact, in substantially less time). In any case, the total simulation time is $O(f(|d|)^2)$.  

**PTIME simulation of recursive read-only programs.**

**Lemma 5.4** \( \text{WHILE}^{\text{rec}\backslash \text{cons}} \preceq \text{WHILE}^{\text{ptime}} \)

**Proof** Let \( p \) be a \( \text{WHILE}^{\text{rec}\backslash \text{cons}} \) program, and \( d \) its input with length \( n \). Since \( p \) is \( \text{cons} \)-free, the value that any \( p \) store \( \sigma \) assigns to any variable \( x \) must be a pointer to some part of \( d \), and thus a number between 0 and \( n \). If \( p \) contains at most \( k \) variables, the number of stores on input \( d \) is at most \((n + 1)^k\).

By Lemma 5.3, \( p \) can be simulated in time \( O((n + 1)^{2k}) \), using the informal algorithm sketched there. Completion of the proof involves a programming of that algorithm in the \( \text{WHILE} \) language. This can clearly be done without increasing time by more than a small polynomial.

**6 Realizing polynomial time by recursive cons-free programs**

The remaining goal is to show that any polynomial-time decidable subset of \( \{0, 1\}^* \) can also be decided by a recursive cons-free \( \text{WHILE} \) program. This in effect shows that storage allocation by \( \text{cons} \) may be replaced by recursion. The proof will require a more specific storage model than seen earlier. Once this is set up, we will use a counter machine, as in the characterization of \( \text{LOGSPACE} \), but in a recursive version.

**Explicit storage: cons.** We have assumed every elementary operation \( \text{cons} \), \( \text{hd} \), etc. as well as every conditional to take one time unit. These costs may seem illogical and even unreasonable since, for example, the command \( X := \text{cons} \ Y \ Y \) binds to \( X \) a tree with more than twice as many nodes as the one bound to \( Y \). In fact, it is reasonable to assign constant time to a \( \text{cons} \) operation and the others using the data-sharing implementation techniques common to Lisp and other functional languages. In this section we give such a semantics for the flow chart version \( \text{GOTO} \), using a Pascal-like simulation.

First, an example: consider the \texttt{reverse} program of Section 3.1.3. The two DAGs (directed acyclic graph) of Figure 6.1 illustrate the storage state at the beginning and at the end of execution.
Figure 6.1: First and last DAG in execution of reverse

**Construction.** Let $p = 1: I_1; \ldots; m: I_m$ be a GOTO program, with variables $X, Z_1, \ldots, Z_k$ and with input and output through variable $X$. Construct a Pascal-like simulating program as follows:

```pascal
type Index = 1..infinity;
  Node = 0..infinity; (* 0 encodes nil *)
var X, Y, Z1, ..., Zk : Node;
  Hd, Tl : array Index of Node;
  Time : Index;       (* The current step number *)
Hd[0] := 0; Tl[0] := 0;     (* hd and tl of nil give nil *)
X := 0; \ Z1:=0; \ldots; Zn:=0;     (* Initialize all vars to nil *)
Time := 1;            (* Step number initially 1 *)
  1 : T11;           (* Code simulating p's instructions *)
  2 : T12;
... 
m : \ Tm;
m+1 : writeout;        (* Write answer X using Hd, Tl *)
```

The storage is regarded as a DAG (initially empty for simplicity; input will be treated shortly). Command $J_1$, which simulates command $I_l$ for $l \geq 0$, is defined in Figure 6.2.

The variable $\text{Time}$ will keep track of the number of steps executed since the computation started, and so is zero when computation begins. The two parallel arrays $\text{Hd}, \text{Tl}$ hold all pointers to $\text{hd}$ and $\text{tl}$ substructures. The values of variables $X_1$, etc. will always be pointers to nodes in this DAG. A variable has value $0$ if it is bound to $\text{nil}$, and otherwise points to a position in the $\text{Hd}$ and $\text{Tl}$ arrays.

For simplicity we handle allocation by using variable $\text{Time}$ to find an unused
<table>
<thead>
<tr>
<th>Form of ( I_i )</th>
<th>Simulating command 1:J1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X := \text{nil} )</td>
<td>( X := 0; \text{Time} := \text{Time} + 1 )</td>
</tr>
<tr>
<td>( X := Y )</td>
<td>( X := Y; \text{Time} := \text{Time} + 1 )</td>
</tr>
<tr>
<td>( X := \text{hd} Y )</td>
<td>( X := \text{Hd}[Y]; \text{Time} := \text{Time} + 1 )</td>
</tr>
<tr>
<td>( X := \text{tl} Y )</td>
<td>( X := \text{Tl}[Y]; \text{Time} := \text{Time} + 1 )</td>
</tr>
<tr>
<td>( X := \text{cons} Y Z )</td>
<td>( \text{Hd} (\text{Time}) := Y; \text{Tl} (\text{Time}) := Z; )</td>
</tr>
<tr>
<td>( \text{if} \ X = \text{nil} \text{ goto } r \text{ else } s )</td>
<td>( \text{if} \ X = 0 \text{ then goto } r \text{ else goto } s )</td>
</tr>
</tbody>
</table>

*Figure 6.2: Pascal-like program simulation code.*

index in these arrays, so every DAG node will be identified by *the time at which it was created*. Note that each of the simulation sequences in Figure 6.2 takes constant time, under the usual assumptions about Pascal program execution.

Suppose now that program \( p \) has input \( d = (a_1 a_2 \ldots a_n) \in D_{01} \). This data has to be stored into the Pascal data structures \( \text{Hd}, \text{Tl} \). One way to describe this is to assume that variable \( X \) has been initialized by the following sequence of \( n + 3 \) instructions, inserted at the start of \( p \). Here \( \text{Zero} \) indicates the atom \( \text{nil} \), modeled by the always-present cell 0:

\[
\text{One} := \text{cons} \text{Zero} \text{Zero}; X := \text{Zero}; \text{Init}_1; \ldots \text{Init}_n;
\]

Instruction \( \text{Init}_i \) for \( 1 \leq i \leq n \) is:

\[
X := \text{cons} \text{Zero} \ X \ \text{if} \ a_i = 1, \text{else} \\
X := \text{cons} \text{One} \ X \ \text{if} \ a_i = 1
\]

The following indicates the initial DAG built this way for input \( d = (1 0) \), coded as ((nil.nil) nil).

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{Hd}[0] &= \text{Tl}[0] = 0: \text{Head}, \text{tail of nil} = \text{nil} \\
\text{Hd}[1] &= \text{Tl}[1] = 0: \text{Cell for One} = (\text{nil.nil}) \\
X &= \text{nil} \text{ at start, no values in Hd}[2] \text{ or Tl}[2] \\
\text{Hd}[3] &= \text{Tl}[3] = 0: \text{cons 0 onto } X \\
X &= 4, \text{ Hd}[4] = 1, \text{ Tl}[4] = 3: \text{cons 1 onto } X
\end{align*}
\]

**Trace of an example execution.** Consider the reverse program seen before, and assume that it is given input \( X = (1 0) \) represented as above. This gives rise to the sequence of memory images in Figure 6.3, where

---

11 By this model, the number of DAG nodes created during execution is at most the program's running time. A more parsimonious implementation could, for example, maintain a “free list” of unused memory cells.
Instr

U

Hd

= the instruction about to be executed at time t
= the DAG cell variable U is bound at to time t
= the final values of Hd[t], Tl[t], respectively

Cons-free simulation of PTIME programs.

Lemma 6.1 Let p = 1:I_1 2:I_2 ... m:I_m be a GOTO-program, and let d ∈ D_0 be an input. Let (ℓ_1, σ_1) → ... → (ℓ_t, σ_t) → ... be the (finite or infinite) computation of p on d, where ℓ_1 = 1 and σ_1 is the initial DAG for input d. Then for any t ≥ |d| + 2 and variable X the equations in Figure 6.4 hold.

Proof A simple induction on t, using the code definitions from Figure 6.2.

<table>
<thead>
<tr>
<th>Time t</th>
<th>Instr</th>
<th>Hd</th>
<th>Tl</th>
<th>X_t</th>
<th>Y_t</th>
<th>Z_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Initialize data at</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>times</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>t = 1, ..., n + 2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1: Y := nil</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2: if X goto 4</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>4: Z := hd X</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>5: Y := cons Z Y</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>6: X := t1 X</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>7: goto 2</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>2: if X goto 4</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>4: Z := hd X</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>5: Y := cons Z Y</td>
<td>0</td>
<td>8</td>
<td>3</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>6: X := t1 X</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>7: goto 2</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>2: if X goto 4</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>3: goto 8</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>8: write Y</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>13</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 6.3: Execution trace for [[reverse]]((nil.nil) nil).

Lemma 6.2 WHILE^{time} \subseteq CM^{rec-poly}

Proof Suppose one is given a WHILE-program p that runs in time |d|^k on input d, and that n = |d|. Consider the values of Instr, Hd, Tl, X_t. If they can be computed then the value of output variable X is available through X_{n^k+n+3}. Thus WHILE^{time} \subseteq CM^{rec-poly} if we can show that all of the values Instr, etc. can be computed by a recursive counter program whose counters are polynomially bounded.
\[\begin{align*}
\text{Instr}_{t+1} &= \begin{cases}
1': I_{t'} & \text{if } \text{Instr}_t = 1: \text{ goto } 1' \\
1': I_{t'} & \text{if } \text{Instr}_t = 1: \text{ if } X \text{ goto } 1' \text{ else } 1'' \text{ and } X_t \neq 0 \\
1'': I_{t''} & \text{if } \text{Instr}_t = 1: \text{ if } X \text{ goto } 1' \text{ else } 1'' \text{ and } X_t = 0 \\
1 + 1: I_{t+1} & \text{if } \text{Instr}_t = 1: I_t \text{ otherwise}
\end{cases} \\
\text{Hd}_{t+1} &= \begin{cases}
Y_t & \text{if } \text{Instr}_t = 1: X := \text{ cons } Y \\
0 & \text{otherwise}
\end{cases} \\
\text{Tl}_{t+1} &= \begin{cases}
Z_t & \text{if } \text{Instr}_t = 1: X := \text{ cons } Y \\
0 & \text{otherwise}
\end{cases} \\
X_{t+1} &= \begin{cases}
X_t & \text{if } \text{Instr}_t \neq 1: X := \ldots \\
Y_t & \text{if } \text{Instr}_t = 1: X := Y \\
\text{Hd}_{(Y_t)} & \text{if } \text{Instr}_t = 1: X := \text{ hd } Y \\
\text{Tl}_{(Y_t)} & \text{if } \text{Instr}_t = 1: X := \text{ tl } Y \\
t + 1 & \text{if } \text{Instr}_t = 1: X := \text{ cons } Y Z
\end{cases}
\end{align*}\]

**Figure 6.4:** Relations among the values of \text{Hd}, \text{Tl}, \text{X} for \(t > |d| + 3\).

Regard each equation in Figure 6.4 as a definition of a function of one variable \(t\). This is always a nonnegative integer, between 0 and \(|d|^k + |d| + 3\), where the addend \(|d| + 3\) accounts for the time to initialize the \text{Hd}, \text{Tl} tables to represent input \(d\).

The various functions \text{Instr}, \text{Hd}, \text{Tl}, \text{X}, are computable by mutual recursion, at least down to \(t = |d|\). Further, the program input \(d\) uniquely (and simply) determines the values of \text{Hd}, \text{Tl} for \(t < |d|\) (and the other variables are irrelevant). The calls all terminate, since in each call the value of argument \(t\) decreases. Further, \(t\) is bounded by \(p\)'s running time on \(d\), at most \(|d|^k + |d| + 3\).

Combining these, it is straightforward to build a recursive counter machine program \(q\) to simulate \(p\); and it has polynomial size bounds on its counters.

**Lemma 6.3** \(\mathrm{CM}^{\text{rec}-\text{poly}} \preceq \mathrm{CM}^{\text{rec}\backslash +}\)

**Lemma 6.4** \(\mathrm{CM}^{\text{rec}\backslash +} \preceq \mathrm{WHILE}^{\text{rec}\backslash \text{cons}}\)

**Proof** Lemma 4.6 showed \(\mathrm{CM}^{\text{poly}} \preceq \mathrm{CM}^{\backslash +}\), and Lemma 4.7 showed \(\mathrm{CM}^{\backslash +} \preceq \mathrm{WHILE}^{\text{cons}}\). The constructions for these two results can be applied to recursive programs as well.

**Remark.** The recursive counter machine's computation is easily seen to take exponential time, due to recomputing values many times. For example, \text{Instr}_t
is recomputed again and again. Thus even though a polynomial-time problem is being solved, the solver is running in superpolynomial time.

**Proof** of Theorem 5.1. Lemmas 5.4, 6.2, 6.3, and 6.4 establish:

\[ \text{WHILE}^{\text{rec\cons}} \leq \text{WHILE}^{\text{ptime}} \leq \text{CM}^{\text{rec\poly}} \leq \text{CM}^{\text{rec\+}} \leq \text{WHILE}^{\text{rec\cons}} \]

**Further developments.** Stephen Cook [3] proved analogous results in the framework of "auxiliary push-down automata." Further developments involving efficient memoization led to the result that any 2DPDA (*two-way deterministic pushdown automaton*) can be simulated in linear time ([5,11,1]).

7 Conclusions and open questions

The programming approach yields quite simple intrinsic characterizations, by restrictions on program syntax, of the well-known problem classes \text{LOGSPACE} and \text{PTIME}, without external imposition of space or time bounds.

**An interesting open problem.** These two characterizations give new insight into the role of persistent (as opposed to evanescent) storage, and put questions about tradeoffs between computation time and space into sharp relief. The results above can be interpreted as saying that, in the absence of "cons," recursive programs are capable of simulating imperative ones; but at a formidable cost in computing time (exponentially larger), since results computed earlier cannot be stored but must be recomputed. In essence, we have shown that the heap can be replaced by the stack, but at a very high time cost.

It is not known, however, whether this cost is necessary. Proving that it is necessary (as seems likely) would require proving that there exist problems which can be solved in small time with general storage, but which require large time when computed functionally. A simple but typical example would be to establish a nonlinear lower bound on the time that a one-tape, no-memory *two-way pushdown automaton* [5], [11] requires to solve some decision problem. One instance would be to prove that string matching must take superlinear time. We conjecture that this is true.

**Related work.** The "treeless" programs of Wadler [16] also involve a restricted use of data constructors, and a treeless program of type \( \{0, 1\}^* \rightarrow \)}
\{0, 1\} is a “read-only” program in our sense. Consequently the programs output by the “treeless transformer” compute LOGSPACE functions, if restricted to this input-output type. Presumably those of more general type also have a simple characterization in terms of LOGSPACE.

Several other works have given exact characterizations of complexity classes in terms of programming languages or other formal languages. Most are, however, rather more technical than the LOGSPACE and PTIME characterizations above. Meyer and Ritchie’s “loop language” characterization of the elementary functions [14] has a similar approach to that of this paper, but at a (much) higher complexity level. Another early result was the discovery that spectra of first-order logic are identical with \textsc{nexptime} [9], further developed actively as finite model theory, see Immerman [8] and many others. Girard, Scedrov and Scott gave a linear logic characterization of PTIME in [6]. Bellantoni and Cook [4], Leivant and Marion [13], and Hillebrand and Kanellakis [7] have characterized several complexity classes by variants of the lambda-calculus.

References


12 Wadler’s requirement of right-hand side linearity probably implies it is a proper subset of LOGSPACE.


