CIS 770: Formal Language Theory

Pavithra Prabhakar

Kansas State University

Spring 2016

High-Level Descriptions of Computation

High-Level Descriptions of Computation

- Instead of giving a Turing Machine, we shall often describe a program as code in some programming language

High-Level Descriptions of Computation

- Instead of giving a Turing Machine, we shall often describe a program as code in some programming language (or often "pseudo-code")

High-Level Descriptions of Computation

- Instead of giving a Turing Machine, we shall often describe a program as code in some programming language (or often "pseudo-code")
  - Possibly using high level data structures and subroutines

High-Level Descriptions of Computation

- Instead of giving a Turing Machine, we shall often describe a program as code in some programming language (or often "pseudo-code")
    - Possibly using high level data structures and subroutines
- Inputs and outputs are complex objects, encoded as strings

High-Level Descriptions of Computation

- Instead of giving a Turing Machine, we shall often describe a program as code in some programming language (or often "pseudo-code")
  - Possibly using high level data structures and subroutines
- Inputs and outputs are complex objects, encoded as strings
- Examples of objects:

High-Level Descriptions of Computation

- Instead of giving a Turing Machine, we shall often describe a program as code in some programming language (or often "pseudo-code")
  - Possibly using high level data structures and subroutines
- Inputs and outputs are complex objects, encoded as strings
- Examples of objects:
  - Matrices, graphs, geometric shapes, images, videos, ...

High-Level Descriptions of Computation

- Instead of giving a Turing Machine, we shall often describe a program as code in some programming language (or often "pseudo-code")
  - Possibly using high level data structures and subroutines
- Inputs and outputs are complex objects, encoded as strings
- Examples of objects:
  - Matrices, graphs, geometric shapes, images, videos, ...
  - DFAs, NFAs, Turing Machines, Algorithms, other machines ...

- "Everything" finite can be encoded as a (finite) string of symbols from a finite alphabet (e.g. ASCII)

- “Everything” finite can be encoded as a (finite) string of symbols from a finite alphabet (e.g. ASCII)
  - Can in turn be encoded in binary (as modern day computers do).

High-Level Descriptions of Computation

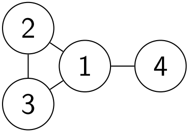
Encoding Complex Objects

- "Everything" finite can be encoded as a (finite) string of symbols from a finite alphabet (e.g. ASCII)
  - Can in turn be encoded in binary (as modern day computers do). No special $\sqcup$ symbol: use self-terminating representations

- "Everything" finite can be encoded as a (finite) string of symbols from a finite alphabet (e.g. ASCII)
  - Can in turn be encoded in binary (as modern day computers do). No special $\sqcup$ symbol: use self-terminating representations
- Example: encoding a "graph."

  `(1,2,3,4)((1,2)(2,3)(3,1)(1,4))`

  encodes the graph



High-Level Descriptions of Computation

- We have already seen several algorithms, for problems involving complex objects like DFAs, NFAs, regular expressions, and Turing Machines

High-Level Descriptions of Computation

- We have already seen several algorithms, for problems involving complex objects like DFAs, NFAs, regular expressions, and Turing Machines
  - For example, convert a NFA to DFA; Given a NFA N and a word w, decide if $w \in L(N)$; ...

High-Level Descriptions of Computation

- We have already seen several algorithms, for problems involving complex objects like DFAs, NFAs, regular expressions, and Turing Machines
  - For example, convert a NFA to DFA; Given a NFA N and a word w, decide if $w \in L(N)$; ...
- All these inputs can be encoded as strings and all these algorithms can be implemented as Turing Machines

High-Level Descriptions of Computation

- We have already seen several algorithms, for problems involving complex objects like DFAs, NFAs, regular expressions, and Turing Machines
  - For example, convert a NFA to DFA; Given a NFA N and a word w, decide if $w \in L(N)$; ...
- All these inputs can be encoded as strings and all these algorithms can be implemented as Turing Machines
- Some of these algorithms are for decision problems, while others are for computing more general functions

High-Level Descriptions of Computation

- We have already seen several algorithms, for problems involving complex objects like DFAs, NFAs, regular expressions, and Turing Machines
  - For example, convert a NFA to DFA; Given a NFA N and a word w, decide if $w \in L(N)$; ...
- All these inputs can be encoded as strings and all these algorithms can be implemented as Turing Machines
- Some of these algorithms are for decision problems, while others are for computing more general functions
- All these algorithms terminate on all inputs

Some more decision problems that have algorithms that always halt (sketched in the textbook)

Some more decision problems that have algorithms that always halt (sketched in the textbook)

- On input $\langle B, w \rangle$ where B is a DFA and w is a string, decide if B accepts w.

Some more decision problems that have algorithms that always halt (sketched in the textbook)

- On input $\langle B, w \rangle$ where B is a DFA and w is a string, decide if B accepts w. Algorithm: simulate B on w and accept iff simulated B accepts

Some more decision problems that have algorithms that always halt (sketched in the textbook)

- On input $\langle B, w \rangle$ where B is a DFA and w is a string, decide if B accepts w. Algorithm: simulate B on w and accept iff simulated B accepts
- On input $\langle B \rangle$ where B is a DFA, decide if $L(B) = \emptyset$.

Some more decision problems that have algorithms that always halt (sketched in the textbook)

- On input $\langle B, w\rangle$ where B is a DFA and w is a string, decide if B accepts w. Algorithm: simulate B on w and accept iff simulated B accepts
- On input $\langle B\rangle$ where B is a DFA, decide if $L(B) = \emptyset$. Algorithm: Use a fixed point algorithm to find all reachable states. See if any final state is reachable.

Some more decision problems that have algorithms that always halt (sketched in the textbook)

- On input $\langle B, w \rangle$ where B is a DFA and w is a string, decide if B accepts w. Algorithm: simulate B on w and accept iff simulated B accepts
- On input $\langle B \rangle$ where B is a DFA, decide if $L(B) = \emptyset$. Algorithm: Use a fixed point algorithm to find all reachable states. See if any final state is reachable.

Code is just data: A TM can take "the code of a program" (DFA, NFA or TM) as part of its input and analyze or even execute this code

Some more decision problems that have algorithms that always halt (sketched in the textbook)

- On input $\langle B, w \rangle$ where B is a DFA and w is a string, decide if B accepts w. Algorithm: simulate B on w and accept iff simulated B accepts
- On input $\langle B \rangle$ where B is a DFA, decide if $L(B) = \emptyset$. Algorithm: Use a fixed point algorithm to find all reachable states. See if any final state is reachable.

Code is just data: A TM can take "the code of a program" (DFA, NFA or TM) as part of its input and analyze or even execute this code

Universal Turing Machine (a simple "Operating System"): Takes a TM M and a string w and simulates the execution of M on w

Decidable and Recognizable Languages

Recall: Definition

A Turing machine M is said to recognize a language L if $L = L(M)$.

Decidable and Recognizable Languages

Recall: Definition

A Turing machine M is said to recognize a language L if $L = L(M)$. A Turing machine M is said to decide a language L if $L = L(M)$ and M halts on every input.

Decidable and Recognizable Languages

Recall: Definition

A Turing machine M is said to recognize a language L if $L = L(M)$.
A Turing machine M is said to decide a language L if $L = L(M)$ and M halts on every input.

L is said to be Turing-recognizable (or simply recognizable) if there exists a TM M which recognizes L.

Decidable and Recognizable Languages

Recall: Definition

A Turing machine M is said to recognize a language L if $L = L(M)$.
A Turing machine M is said to decide a language L if $L = L(M)$ and M halts on every input.

L is said to be Turing-recognizable (or simply recognizable) if there exists a TM M which recognizes L. L is said to be Turing-decidable (or simply decidable) if there exists a TM M which decides L.

Decidable and Recognizable Languages

Recall: Definition

A Turing machine M is said to recognize a language L if $L = L(M)$. A Turing machine M is said to decide a language L if $L = L(M)$ and M halts on every input.

L is said to be Turing-recognizable (or simply recognizable) if there exists a TM M which recognizes L. L is said to be Turing-decidable (or simply decidable) if there exists a TM M which decides L.

- Every finite language is decidable

Decidable and Recognizable Languages

Recall: Definition

A Turing machine M is said to recognize a language L if $L = L(M)$.
A Turing machine M is said to decide a language L if $L = L(M)$ and M halts on every input.

L is said to be Turing-recognizable (or simply recognizable) if there exists a TM M which recognizes L. L is said to be Turing-decidable (or simply decidable) if there exists a TM M which decides L.

- Every finite language is decidable: For example, by a TM that has all the strings in the language "hard-coded" into it

Decidable and Recognizable Languages

Recall: Definition

A Turing machine M is said to recognize a language L if $L = L(M)$. A Turing machine M is said to decide a language L if $L = L(M)$ and M halts on every input.

L is said to be Turing-recognizable (or simply recognizable) if there exists a TM M which recognizes L. L is said to be Turing-decidable (or simply decidable) if there exists a TM M which decides L.

- Every finite language is decidable: For example, by a TM that has all the strings in the language "hard-coded" into it
- We just saw some example algorithms all of which terminate in a finite number of steps, and output yes or no (accept or reject).

Decidable and Recognizable Languages

Recall: Definition

A Turing machine M is said to recognize a language L if $L = L(M)$. A Turing machine M is said to decide a language L if $L = L(M)$ and M halts on every input.

L is said to be Turing-recognizable (or simply recognizable) if there exists a TM M which recognizes L. L is said to be Turing-decidable (or simply decidable) if there exists a TM M which decides L.

- Every finite language is decidable: For example, by a TM that has all the strings in the language "hard-coded" into it
- We just saw some example algorithms all of which terminate in a finite number of steps, and output yes or no (accept or reject). i.e., They decide the corresponding languages.

Decidable and Recognizable Languages

- But not all languages are decidable!

Decidable and Recognizable Languages

- But not all languages are decidable! In the next class we will see an example:
  - $A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ is undecidable

Decidable and Recognizable Languages

- But not all languages are decidable! In the next class we will see an example:
  - $A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ is undecidable
- However A_{TM} is Turing-recognizable!

- But not all languages are decidable! In the next class we will see an example:
  - $A_{\mathrm{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$ is undecidable
- However A_{TM} is Turing-recognizable!

Proposition

There are languages which are recognizable, but not decidable

Recognizing A_{TM}

Program U for recognizing A_{TM}:

```
On input ⟨M, w⟩
    simulate M on w
    if simulated M accepts w, then accept
    else reject (by moving to q_rej)
```

Recognizing A_{TM}

Program U for recognizing A_{TM}:

```
On input ⟨M, w⟩
    simulate M on w
    if simulated M accepts w, then accept
    else reject (by moving to q_rej)
```

U (the Universal TM) accepts $\langle M, w\rangle$ iff M accepts w. i.e.,

$$L(U) = A_{\mathrm{TM}}$$

Recognizing A_{TM}

Program U for recognizing A_{TM}:

```
On input ⟨M, w⟩
    simulate M on w
    if simulated M accepts w, then accept
    else reject (by moving to q_rej)
```

U (the Universal TM) accepts $\langle M, w \rangle$ iff M accepts w. i.e.,

$$L(U) = A_{\mathrm{TM}}$$

But U does not decide A_{TM}

Recognizing A_{TM}

Program U for recognizing A_{TM}:

```
On input ⟨M, w⟩
    simulate M on w
    if simulated M accepts w, then accept
    else reject (by moving to q_rej)
```

U (the Universal TM) accepts $\langle M, w\rangle$ iff M accepts w. i.e.,

$$L(U) = A_{\mathrm{TM}}$$

But U does not decide A_{TM}: If M rejects w by not halting, U rejects $\langle M, w\rangle$ by not halting.

Recognizing A_{TM}

Program U for recognizing A_{TM}:

```
On input ⟨M, w⟩
    simulate M on w
    if simulated M accepts w, then accept
    else reject (by moving to q_rej)
```

U (the Universal TM) accepts $\langle M, w\rangle$ iff M accepts w. i.e.,

$$L(U) = A_{\mathrm{TM}}$$

But U does not decide A_{TM}: If M rejects w by not halting, U rejects $\langle M, w\rangle$ by not halting. Indeed (as we shall see) no TM decides A_{TM}.

Deciding vs. Recognizing

Proposition

If L and $\overline{L}$ are recognizable, then L is decidable

Proof.

Program P for deciding L, given programs P_L and $P_{\overline{L}}$ for recognizing L and $\overline{L}$:

Deciding vs. Recognizing

Proposition

If L and $\overline{L}$ are recognizable, then L is decidable

Proof.

Program P for deciding L, given programs P_L and $P_{\overline{L}}$ for recognizing L and $\overline{L}$:

- On input x, simulate P_L and $P_{\overline{L}}$ on input x.

Deciding vs. Recognizing

Proposition

If L and $\overline{L}$ are recognizable, then L is decidable

Proof.

Program P for deciding L, given programs P_L and $P_{\overline{L}}$ for recognizing L and $\overline{L}$:

- On input x, simulate P_L and $P_{\overline{L}}$ on input x. Whether $x \in L$ or $x \notin L$, one of P_L and $P_{\overline{L}}$ will halt in finite number of steps.
- Which one to simulate first?

Deciding vs. Recognizing

Proposition

If L and $\overline{L}$ are recognizable, then L is decidable

Proof.

Program P for deciding L, given programs P_L and $P_{\overline{L}}$ for recognizing L and $\overline{L}$:

- On input x, simulate P_L and $P_{\overline{L}}$ on input x. Whether $x \in L$ or $x \notin L$, one of P_L and $P_{\overline{L}}$ will halt in finite number of steps.
- Which one to simulate first? Either could go on forever.

Deciding vs. Recognizing

Proposition

If L and $\overline{L}$ are recognizable, then L is decidable

Proof.

Program P for deciding L, given programs P_L and $P_{\overline{L}}$ for recognizing L and $\overline{L}$:

- On input x, simulate P_L and $P_{\overline{L}}$ on input x. Whether $x \in L$ or $x \notin L$, one of P_L and $P_{\overline{L}}$ will halt in finite number of steps.
- Which one to simulate first? Either could go on forever.
- On input x, simulate in parallel P_L and $P_{\overline{L}}$ on input x until either P_L or $P_{\overline{L}}$ accepts

Deciding vs. Recognizing

Proposition

If L and $\overline{L}$ are recognizable, then L is decidable

Proof.

Program P for deciding L, given programs P_L and $P_{\overline{L}}$ for recognizing L and $\overline{L}$:

- On input x, simulate P_L and $P_{\overline{L}}$ on input x. Whether $x \in L$ or $x \notin L$, one of P_L and $P_{\overline{L}}$ will halt in finite number of steps.
- Which one to simulate first? Either could go on forever.
- On input x, simulate in parallel P_L and $P_{\overline{L}}$ on input x until either P_L or $P_{\overline{L}}$ accepts
- If P_L accepts, accept x and halt. If $P_{\overline{L}}$ accepts, reject x and halt.

Deciding vs. Recognizing

Proof (contd).

In more detail, P works as follows:

```
On input x
for i = 1, 2, 3, ...
    simulate P_L on input x for i steps
    simulate P_L̄ on input x for i steps
    if either simulation accepts, break
if P_L accepted, accept x (and halt)
if P_L̄ accepted, reject x (and halt)
```

Deciding vs. Recognizing

Proof (contd).

In more detail, P works as follows:

```
On input x
for i = 1, 2, 3, ...
    simulate P_L on input x for i steps
    simulate P_L̄ on input x for i steps
    if either simulation accepts, break
if P_L accepted, accept x (and halt)
if P_L̄ accepted, reject x (and halt)
```

(Alternately, maintain configurations of P_L and $P_{\overline{L}}$, and in each iteration of the loop advance both their simulations by one step.) □

Deciding vs. Recognizing

So far:

- A_{TM} is undecidable (next lecture)
- But it is recognizable

Deciding vs. Recognizing

So far:

- A_{TM} is undecidable (next lecture)
- But it is recognizable
- Is every language recognizable?

Deciding vs. Recognizing

So far:

- A_{TM} is undecidable (next lecture)
- But it is recognizable
- Is every language recognizable? No!

Deciding vs. Recognizing

So far:

- A_{TM} is undecidable (next lecture)
- But it is recognizable
- Is every language recognizable? No!

Proposition

$\overline{A_{\mathrm{TM}}}$ *is unrecognizable*

Deciding vs. Recognizing

So far:

- A_{TM} is undecidable (next lecture)
- But it is recognizable
- Is every language recognizable? No!

Proposition

$\overline{A_{\mathrm{TM}}}$ *is unrecognizable*

Proof.

If $\overline{A_{\mathrm{TM}}}$ is recognizable, since A_{TM} is recognizable, the two languages will be decidable too! $\square$

Deciding vs. Recognizing

So far:

- A_{TM} is undecidable (next lecture)
- But it is recognizable
- Is every language recognizable? No!

Proposition

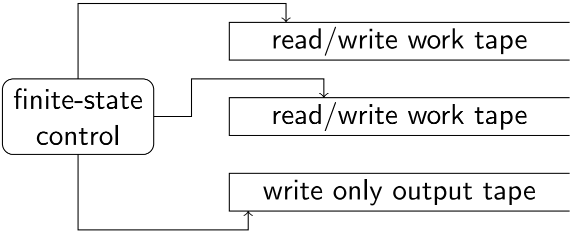
$\overline{A_{\mathrm{TM}}}$ *is unrecognizable*

Proof.

If $\overline{A_{\mathrm{TM}}}$ is recognizable, since A_{TM} is recognizable, the two languages will be decidable too! □

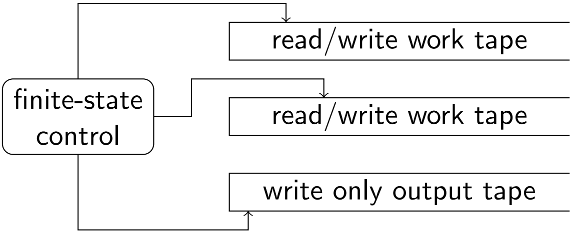
Note: Decidable languages are closed under complementation, but recognizable languages are not.

Enumerators



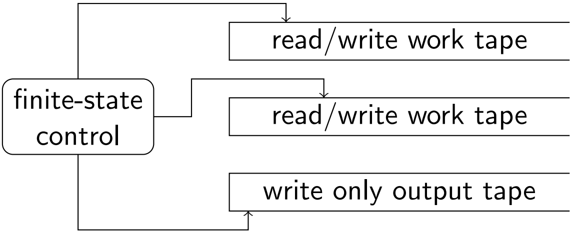
- An enumerator is multi-tape Turing Machine, with a special output tape which is write-only
  - Write-only means (a) symbol on output tape does not affect transitions, and (b) tape head only moves right.

Enumerators



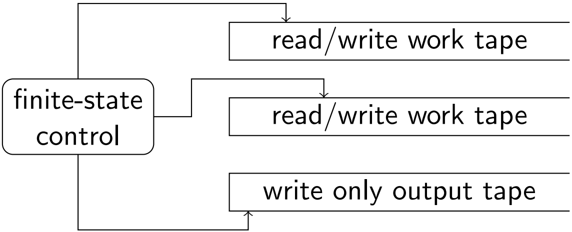
- An enumerator is multi-tape Turing Machine, with a special output tape which is write-only
  - Write-only means (a) symbol on output tape does not affect transitions, and (b) tape head only moves right.
- Intially all tapes blank (no input).

Enumerators



- An enumerator is multi-tape Turing Machine, with a special output tape which is write-only
  - Write-only means (a) symbol on output tape does not affect transitions, and (b) tape head only moves right.
- Intially all tapes blank (no input). During computation the machine adds symbols to the output tape.

Enumerators



- An enumerator is multi-tape Turing Machine, with a special output tape which is write-only
  - Write-only means (a) symbol on output tape does not affect transitions, and (b) tape head only moves right.
- Intially all tapes blank (no input). During computation the machine adds symbols to the output tape. Output considered to be a list of words (separated by special symbol #)

Recursively Enumerable Languages

Definition

An enumerator M is said to enumerate a string w if and only if at some point M writes a word w on the output tape.

Recursively Enumerable Languages

Definition

An enumerator M is said to enumerate a string w if and only if at some point M writes a word w on the output tape.
$E(M) = \{w \mid M \text{ enumerates } w\}$

Recursively Enumerable Languages

Definition

An enumerator M is said to enumerate a string w if and only if at some point M writes a word w on the output tape.
$E(M) = \{w \mid M \text{ enumerates } w\}$

Note

M need not enumerate strings in order. It is also possible that M lists some strings many times!

Recursively Enumerable Languages

Definition

An enumerator M is said to enumerate a string w if and only if at some point M writes a word w on the output tape.
$E(M) = \{w \mid M \text{ enumerates } w\}$

Note

M need not enumerate strings in order. It is also possible that M lists some strings many times!

Definition

L is recursively enumerable (r.e.) iff there is an enumerator M such that $L = E(M)$.

Recursively Enumerable Languages and TMs

Theorem

L is recursively enumerable if and only if L is Turing-recognizable.

Recursively Enumerable Languages and TMs

Theorem

L is recursively enumerable if and only if L is Turing-recognizable.

Note

Hence, when we say a language L is recursively enumerable (r.e.) then

- there is a TM that accepts L, and
- there is an enumerator that enumerates L.

Recognizers From Enumerators

Proof.

Suppose L is enumerated by N. Need to construct M such that $L(M) = E(N)$.

Recognizers From Enumerators

Proof.

Suppose L is enumerated by N. Need to construct M such that $L(M) = E(N)$. M is the following TM

On input w
    Run N. Every time N writes a word 'x'
    compare x with w.
    If $x = w$ then accept and halt
    else continue simulating N

Recognizers From Enumerators

Proof.

Suppose L is enumerated by N. Need to construct M such that $L(M) = E(N)$. M is the following TM

```
On input w
    Run N.  Every time N writes a word 'x'
    compare x with w.
    If x = w then accept and halt
    else continue simulating N
```

Clearly, if $w \in L$, M accepts w, and if $w \notin L$ then M never halts.

Enumerators From Recognizers

Proof (contd).

Let M be such that $L = L(M)$. Need to construct N such that $E(N) = L(M)$.

Enumerators From Recognizers

Proof (contd).

Let M be such that $L = L(M)$. Need to construct N such that $E(N) = L(M)$. N is the following enumerator

```
for w = ε, 0, 1, 00, 01, 10, 11, 000, . . . do
    simulate M on w
    if M accepts w then write the word ‘w’
        on output tape
```

Enumerators From Recognizers?

Proof (contd).

Let M be such that $L = L(M)$. Need to construct N such that $E(N) = L(M)$. N is the following enumerator

```
for w = ε, 0, 1, 00, 01, 10, 11, 000, ... do
    simulate M on w
    if M accepts w then write the word 'w'
        on output tape
```

Does N enumerate L?

Enumerators From Recognizers?

Proof (contd).

Let M be such that $L = L(M)$. Need to construct N such that $E(N) = L(M)$. N is the following enumerator

```
for w = ε, 0, 1, 00, 01, 10, 11, 000, ... do
    simulate M on w
    if M accepts w then write the word 'w'
        on output tape
```

Does N enumerate L? No!! M may not halt on a string $w \notin L$, in which case N will not output any more strings!

Enumerators From Recognizers?

Parallel simulation

Proof (contd).

Let M be such that $L = L(M)$. Need to construct N such that $E(N) = L(M)$. N is the following enumerator

```
for w = ϵ, 0, 1, 00, 01, 10, 11, 000, ... do
    simulate M on w
    if M accepts w then write the word ‘w’
        on output tape
```

Does N enumerate L? No!! M may not halt on a string $w \notin L$, in which case N will not output any more strings!
Must simulate M on all inputs in parallel.

Enumerators From Recognizers?

Parallel simulation?

Proof (contd).

Let M be such that $L = L(M)$. Need to construct N such that $E(N) = L(M)$. N is the following enumerator

```
for w = ϵ, 0, 1, 00, 01, 10, 11, 000, ... do
    simulate M on w
    if M accepts w then write the word 'w'
        on output tape
```

Does N enumerate L? No!! M may not halt on a string $w \notin L$, in which case N will not output any more strings!
Must simulate M on all inputs in parallel. But infinitely many parallel executions.

Proof (contd).

Let M be such that $L = L(M)$. Need to construct N such that $E(N) = L(M)$. N is the following enumerator

```
for w = ε, 0, 1, 00, 01, 10, 11, 000, ... do
    simulate M on w
    if M accepts w then write the word 'w'
        on output tape
```

Does N enumerate L? No!! M may not halt on a string $w \notin L$, in which case N will not output any more strings!
Must simulate M on all inputs in parallel. But infinitely many parallel executions. Will never reach step two in any execution!

Proof (contd).

Let M be such that $L = L(M)$. Need to construct N such that $E(N) = L(M)$.

Proof (contd).

Let M be such that $L = L(M)$. Need to construct N such that $E(N) = L(M)$. N is the following enumerator

```
for i = 1,2,3... do
    let w_1, w_2, ... w_i be the first i strings (in
        lexicographic order)
    simulate M on w_1 for i steps, then on w_2 for i
        steps and ...simulate M on w_i for i steps
    if M accepts w_j within i steps then write w_j
        (with separator) on output tape
```

Proof (contd).

Let M be such that $L = L(M)$. Need to construct N such that $E(N) = L(M)$. N is the following enumerator

for $i = 1, 2, 3 \ldots$ do
    let $w_1, w_2, \ldots w_i$ be the first i strings (in lexicographic order)
    simulate M on w_1 for i steps, then on w_2 for i steps and ...simulate M on w_i for i steps
    if M accepts w_j within i steps then write w_j (with separator) on output tape

Observe that $w \in L(M)$ iff N will enumerates w.

Proof (contd).

Let M be such that $L = L(M)$. Need to construct N such that $E(N) = L(M)$. N is the following enumerator

```
for i = 1,2,3... do
    let w_1, w_2, ... w_i be the first i strings (in
        lexicographic order)
    simulate M on w_1 for i steps, then on w_2 for i
        steps and ...simulate M on w_i for i steps
    if M accepts w_j within i steps then write w_j
        (with separator) on output tape
```

Observe that $w \in L(M)$ iff N will enumerates w. N will enumerate strings many times! $\square$