CS 373: Theory of Computation

Manoj Prabhakaran   Mahesh Viswanathan

Fall 2008

Part I
Lecture 20

Turing Machines

1 Unrestricted Computation

General Computing Machines

- Machines so far: DFAs, NFAs, PDAs
  - Limitations on how they can use memory: fixed amount of memory, plus (for PDAs) a stack
  - Limitations on what they can compute/decide: only regular languages or context free languages
- The complete machine?
  - No limitations on memory usage? But maybe other ways to use computational resources that we haven't thought of...
    * Come up with a model that describes all "conceivable" computation
  - No limitation on what they can compute?
    * No! There are far too many languages over $\{0, 1\}$ than there are "machines" or programs (as long as machines can be represented digitally)

General Computing Machines

Alonzo Church, Emil Post, and Alan Turing (1936)

Figure 1: Alonzo Church

Figure 2: Emil Post

Figure 3: Alan Turing

- Church (λ-calculus), Post (Post's machine), Turing (Turing machine) independently came up with formal definitions of mechanical computation
- All equivalent!
- In this course: Turing Machines

---

2 Turing Machines

2.1 Definition

The 'aha' moment

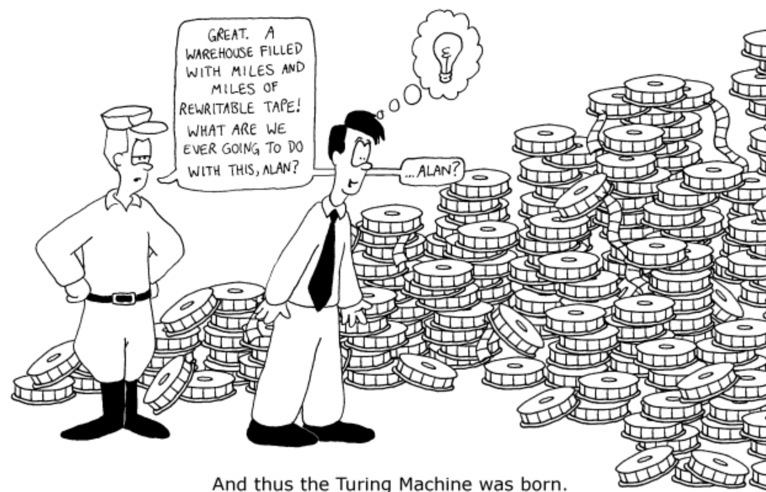
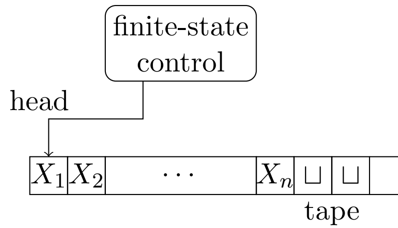


Figure 4:

Turing Machines



- Unrestricted memory: an infinite tape
  - A finite state machine that reads/writes symbols on the tape
  - Can read/write anywhere on the tape
  - Tape is infinite in one direction only (other variants possible)
- Initially, tape has input and the machine is reading (i.e., tape head is on) the leftmost input symbol.
- Transition (based on current state and symbol under head):
  - Change control state
  - Overwrite a new symbol on the tape cell under the head
  - Move the head left, or right.

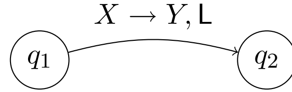
Turing Machines
Formal Definition

A Turing machine is $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}})$ where

- Q is a finite set of control states
- Σ is a finite set of input symbols
- $\Gamma \supseteq \Sigma$ is a finite set of tape symbols. Also, a blank symbol $\sqcup \in \Gamma \setminus \Sigma$
- $q_0 \in Q$ is the initial state
- $q_{\mathsf{acc}} \in Q$ is the accept state
- $q_{\mathsf{rej}} \in Q$ is the reject state, where $q_{\mathsf{rej}} \neq q_{\mathsf{acc}}$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathsf{L}, \mathsf{R}\}$ is the transition function.

  Given the current state and symbol being read, the transition function describes the next state, symbol to be written and direction (left or right) in which to move the tape head.

Transition Function



$\delta(q, X) = (p, Y, \mathsf{L})$: Read transition as "the machine when in state q_1, and reading symbol X under the tape head, will move to state q_2, overwrite X with Y, and move its tape head to the left"

- In fact $\delta : Q - \{q_{\mathsf{acc}}, q_{\mathsf{rej}}\} \times \Gamma \to Q \times \Gamma \times \{\mathsf{L}, \mathsf{R}\}$. No transition defined after reaching q_{acc} or q_{rej}
- Transitions are deterministic
- Convention: if $\delta(q, X)$ is not explicitly specified, it is taken as leading to q_{rej}, i.e., say $\delta(q, X) = (q_{\mathsf{rej}}, \sqcup, \mathsf{R})$

Configurations

The configuration (or "instantaneous description") contains all the information to exactly capture the "current state of the computation"

$$X_1 X_2 \cdots X_{i-1} q X_i \cdots X_n$$

- Includes the current state: q
- Position of the tape head: Scanning i^{th} symbol X_i
- Contents of all the tape cells till the rightmost nonblank symbol. This is will always be finitely many cells. Those symbols are $X_1 X_2 \cdots X_n$, where $X_n \neq \sqcup$ unless the tape head is on it.

Special Configurations

- Start configuration: $q_0 X_1 \cdots X_n$, where the input is $X_1 \cdots X_n$
- Accept and reject configurations: The state q is q_{acc} or q_{rej}, respectively . These configurations are *halting configurations*, because there are no transitions possible from them.

Single Step

Definition 2.1. We say one configuration (C_1) *yields* another (C_2), denoted as $\mathrm{C}_1 \vdash \mathrm{C}_2$, if one of the following holds.

- If $\delta(q, X_i) = (p, Y, \mathsf{L})$ then

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \vdash X_1 X_2 \cdots X_{i-2} p X_{i-1} Y X_{i+1} \cdots X_n$$

  Boundary Cases:

  - If $i = 1$ then $q X_1 X_2 \cdots X_n \vdash p Y X_2 \cdots X_n$
  - If $i = n$ and $Y = \sqcup$ then $X_1 \cdots X_{n-1} q X_n \vdash X_1 \cdots p X_{n-1}$

- If $\delta(q, X_i) = (p, Y, \mathsf{R})$ then

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \vdash X_1 X_2 \cdots X_{i-1} Y p X_{i+1} \cdots X_n$$

  Boundary Case:

  - If $i = n$ then $X_1 \cdots X_{n-1} q X_n \vdash X_1 \cdots X_{n-1} Y p \sqcup$

Computations

Definition 2.2. We say $\mathrm{C}_1 \vdash^* \mathrm{C}_2$ if the machine can move from C_1 to C_2 in zero or more steps. i.e., $\mathrm{C}_1 = \mathrm{C}_2$ or there exist $\mathrm{C}'_1, \ldots, \mathrm{C}'_n$ such that $\mathrm{C}_1 = \mathrm{C}'_1$, $\mathrm{C}_2 = \mathrm{C}'_n$ and $\mathrm{C}'_i \vdash \mathrm{C}'_{i+1}$

Acceptance and Recognition

Definition 2.3. A Turing machine M *accepts* w iff $q_0 w \vdash^* \alpha_1 q_{\mathsf{acc}} \alpha_2$, where α_1, α_2 are some strings. In other words, the machine M when started in its intial state and with w as input, reaches the accept state.

Note: The machine may not read all the symbols in w. It may pass back and forth over some symbols of w several times. Finally, w may have been completely overwritten.

Definition 2.4. For a Turing machine M, define $L(M) = \{w \mid M \textit{ accepts } w\}$. M is said to accept or *recognize* a language L if $L = L(M)$.

2.2 Examples

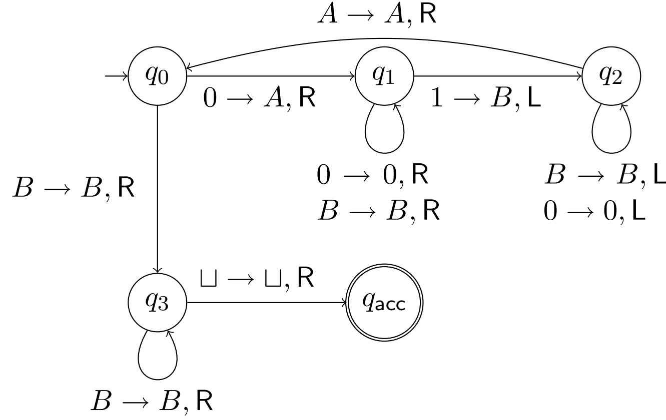
Example 1: TM for $\{0^n 1^n \mid n > 0\}$
Design a TM to accept the language $L = \{0^n 1^n \mid n > 0\}$

High level description

```
On input string w
    while there are unmarked 0s, do
        Mark the left most 0
        Scan right till the leftmost unmarked 1;
```

```
        if there is no such 1 then crash
    Mark the leftmost 1
done
Check to see that there are no unmarked 1s;
    if there are then crash
accept
```

Example 1: TM for $\{0^n1^n \mid n > 0\}$



- Accepts input 0011: $q_00011 \vdash Aq_1011 \vdash A0q_111 \vdash Aq_20B1 \vdash q_2A0B1 \vdash Aq_00B1 \vdash AAq_1B1 \vdash AABq_11 \vdash AAq_2BB \vdash Aq_2ABB \vdash AAq_0BB \vdash AABq_3B \vdash AABBq_3\sqcup \vdash AABB\sqcup q_{\mathsf{acc}}\sqcup$
- Rejects input 00: $q_000 \vdash Aq_10 \vdash A0q_1\sqcup \vdash A0 \sqcup q_{\mathsf{rej}}\sqcup$

Example: $\{0^n1^n \mid n > 0\}$
Formal Definition

The machine is $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}})$ where

- $Q = \{q_0, q_1, q_2, q_3, q_{\mathsf{acc}}, q_{\mathsf{rej}}\}$
- $\Sigma = \{0, 1\}$, and $\Gamma = \{0, 1, A, B, \sqcup\}$
- δ is given as follows

$$\begin{array}{ll}
\delta(q_0, 0) = (q_1, A, R) & \delta(q_0, B) = (q_3, B, R) \\
\delta(q_1, 0) = (q_1, 0, R) & \delta(q_1, B) = (q_1, B, R) \\
\delta(q_1, 1) = (q_2, B, L) & \delta(q_2, B) = (q_2, B, L) \\
\delta(q_2, 0) = (q_2, 0, L) & \delta(q_2, A) = (q_0, A, R) \\
\delta(q_3, B) = (q_3, B, R) & \delta(q_3, \sqcup) = (q_{\mathsf{acc}}, \sqcup, R)
\end{array}$$

In all other cases, $\delta(q, X) = (q_{\mathsf{rej}}, \sqcup, R)$. So for example, $\delta(q_0, 1) = (q_{\mathsf{rej}}, \sqcup, R)$.

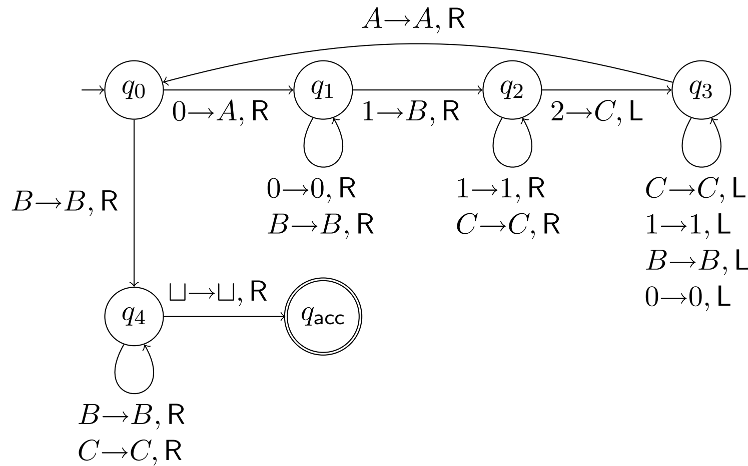
Example 2: TM for $\{0^n 1^n 2^n \mid n > 0\}$

Design a TM to accept the language $L = \{0^n 1^n 2^n \mid n > 0\}$

High level description

```
On input string w
    while there are unmarked 0s, do
        Mark the left most 0
        Scan right to reach the leftmost unmarked 1;
            if there is no such 1 then crash
        Mark the leftmost 1
        Scan right to reach the leftmost unmarked 2;
            if there is no such 2 then crash
        Mark the leftmost 2
    done
    Check to see that there are no unmarked 1s or 2s;
        if there are then crash
    accept
```

Example 2: TM for $\{0^n 1^n 2^n \mid n > 0\}$



e.g.: $q_0 001122 \vdash^* A0Bq_3 1C2 \vdash^* q_3 A0B1C2 \vdash Aq_0 0B1C2 \vdash^* AAq_0 BBCC \vdash^* AABBCCq_4 \sqcup \vdash AABBCC \sqcup q_{\mathsf{acc}} \sqcup$

Deciding a Language

- Only halting configurations are those with state q_{acc} or q_{rej}

- A Turing machine may keep running forever on some input
- Then the machine does not accept that input
- So two ways to not accept: reject or never halt

Definition 2.5. A Turing machine M is said to *decide* a language L if $L = L(M)$ and M halts on every input

Deciding a language is more than recognizing it. There are languages which are recognizable, but not decidable.

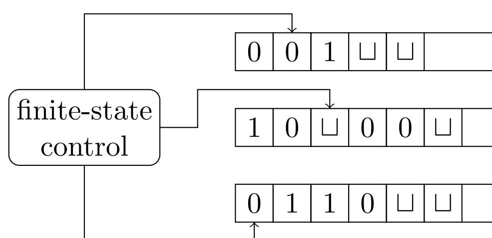
Part II
Lecture 21

Variants of Turing Machines and Church-Turing Thesis

3 Variants of Turing Machines

3.1 Multi-Tape TM

Multi-Tape Turing Machine



- Input on Tape 1
- Initially all heads scanning cell 1, and tapes 2 to k blank
- In one step: Read symbols under each of the k-heads, and depending on the current control state, write new symbols on the tapes, move the each tape head (possibly in different directions), and change state.

Multi-Tape Turing Machine
Formal Definition

A k-tape Turing Machine is $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}})$ where

- Q is a finite set of control states
- Σ is a finite set of input symbols
- $\Gamma \supseteq \Sigma$ is a finite set of tape symbols. Also, a blank symbol $\sqcup \in \Gamma \setminus \Sigma$
- $q_0 \in Q$ is the initial state
- $q_{\mathsf{acc}} \in Q$ is the accept state
- $q_{\mathsf{rej}} \in Q$ is the reject state, where $q_{\mathsf{rej}} \neq q_{\mathsf{acc}}$
- $\delta : (Q \setminus \{q_{\mathsf{acc}}, q_{\mathsf{rej}}\}) \times \Gamma^k \to Q \times (\Gamma \times \{\mathsf{L}, \mathsf{R}\})^k$ is the transition function.

Computation, Acceptance and Language

- A configuration of a multi-tape TM must describe the state, contents of all k-tapes, and positions of all k-heads. Thus, $\mathrm{C} \in Q \times (\Gamma^*\{*\}\Gamma\Gamma^*)^k$, where $*$ denotes the head position.
- Accepting configuration is one where the state is q_{acc}, and starting configuration on input w is $(q_0, *w, *\sqcup, \ldots, *\sqcup)$
- Formal definition of a single step is skipped.
- w is *accepted* by M, if from the starting configuration with w as input, M reaches an accepting configuration.
- $L(M) = \{w \mid w \text{ accepted by } M\}$

Expressive Power of multi-tape TM

Theorem 3.1. *For any k-tape Turing Machine M, there is a single tape TM single(M) such that $L(\textit{single}(M)) = L(M)$.*

Challenges

- How do we store k-tapes in one?
- How do we simulate the movement of k independent heads?

Storing Multiple Tapes

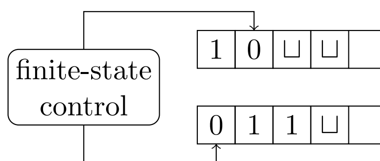


Figure 5: Multi-tape TM M

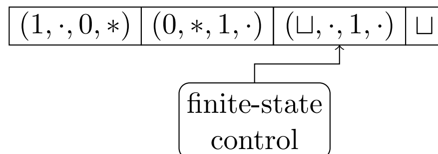


Figure 6: 1-tape equivalent single(M)

Store in cell i contents of cell i of all tapes. “Mark” head position of tape with $*$.

Simulating One Step

Challenge 1: Head of 1-Tape TM is pointing to one cell. How do we find out all the k symbols that are being read by the k heads, which maybe in different cells?

- Read the tape from left to right, storing the contents of the cells being scanned in the *state*, as we encounter them.

Challenge 2: After this scan, 1-tape TM knows the next step of k-tape TM. How do we change the contents and move the heads?

- Once again, scan the tape, change all relevant contents, “move” heads (i.e., move $*$s), and change state.

Overall Construction
First we outline the high-level algorithm for the 1-tape TM. On input w

1. First the machine will rewrite input w to be in “new” format.
2. To simulate one step
   - Read from left-to-right remembering symbols read on each tape, and move all the way back to leftmost position.
   - Read from left-to-right, changing symbols, and moving those heads that need to be moved right.
   - Scan back from right-to-left moving the heads that need to be moved left.

Formally, let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}})$. To define the machine single(M) it is useful to identify modes that single(M) could be in while simulating M. The modes are

$$\text{mode} = \{\text{init}, \text{back-init}, \text{read}, \text{back-read}, \text{fix-right}, \text{fix-left}\}$$

where

- init means that the machine is rewriting input in new format
- back-init means the machine is just going back all the way to the leftmost cell after “initiallizing” the tape
- read means the machine is scanning from left to right to read all symbols being read by k-tape machine
- back-read means the machine is going back to leftmost cell after the “read” phase
- fix-right means the machine is scanning from left to right and is going to make all tape changes and move those heads that need to be moved right

- fix-left means the machine is scanning right to left and moving all heads that need to be moved left

Now single(M) = $(Q', \Sigma', \Gamma', \delta', q'_0, q'_{\mathsf{acc}}, q'_{\mathsf{rej}})$ where

- Recall, based on the high-level description, single(M) needs to remember a few things in its state. It needs to keep track of the current "mode"; Ms state; during the read phase the symbols being scanned by each head of M; at the end of the read phase, the new symbols to be written and direction to move the heads. Thus,

$$Q' = \{q'_0, q'_{\mathsf{acc}}, q'_{\mathsf{rej}}\} \cup (\text{modes} \times Q \times (\Gamma \times \{\mathsf{L}, \mathsf{R}, *\})^k)$$

  where $q'_0, q'_{\mathsf{acc}}, q'_{\mathsf{rej}}$ are new initial, accept and reject states, respectively. "$*$" is new special symbol that we will use to when placing new head positions, and can be ignored for now. Intuitively, when the mode is "read" the directions don't mean anything, and symbols in Γ will be the symbols that M is scanning. During the "fix" phases the directions are the directions each head needs to be moved, and the symbols are the new symbols to be written.

- $\Sigma' = \Sigma$; the input alphabet does not change

- On the tape, we write the contents of one cell of each of the k-tapes and whether the head scans that position or not. Thus, $\Gamma' = \{\rhd, \sqcup\} \cup (\Gamma \times \{\cdot, *\})^k$, where $\rhd$ will be a new left-end-marker, as it will be useful for single(M) to know when it has finished scanning all the way to the left. $\sqcup$ as always is the blank symbol of the machine.

- The initial state, accept state and reject states are the new states q'_0, q'_{acc}, and q'_{rej}.

We will now formally define the transition function δ. We will describe δ for various cases below; for a case not covered below, we will assume our usual convention that the machine single(M) goes to the reject state q'_{rej} and moves the head left.

Initial State In the first step, single(M) will move to the "initialization phase", which will write a (new) left endmarker, and rewrite the tape in the correct format for the future. Thus, from initial state q'_0 you go to a state whose "mode" is init. Since we are going to insert a new left end-marker, we need to "shift" all symbols of the input one-space to right, which can be accomplished by remembering the next symbol to be written in the state. So $\delta'(q'_0, a) = (\langle \text{init}, q_0, a, *, 0, \mathsf{L}, 0, \mathsf{L}, \ldots, 0\mathsf{L}\rangle, \rhd)$; the symbols 0 and L don't mean anything (and so can be changed to whatever you please), and the $*$ remembers that when we write the next symbol all heads must be in that position.

Initialization In the initialization phase, we just read a symbol and write it in the "new format", which means writing blank symbols for all the other tape cells, and moving right. When we scan the entire input to go back left, i.e., change mode to back-init. There are two caveats to this. First we are shifting symbols of the input one position to the right because of the left endmarker $\rhd$; so we actually write what we remembered in our state, and remember what we read in the state. Also, in the first position, we need to "place" all the heads; this is remembered because of $*$. So we have

$$\delta'(\langle \text{init}, q_0, a, *, 0, \mathsf{L}, \ldots, 0, \mathsf{L}\rangle, b) = (\langle \text{init}, q_0, b, \mathsf{L}, 0, \mathsf{L}, \ldots, 0, \mathsf{L}\rangle, (a, *, \sqcup, *, \ldots, \sqcup, *), \mathsf{R})$$
$$\delta'(\langle \text{init}, q_0, a, \mathsf{L}, 0, \mathsf{L}, \ldots, 0, \mathsf{L}\rangle, b) = (\langle \text{init}, q_0, b, \mathsf{L}, 0, \mathsf{L}, \ldots, 0, \mathsf{L}\rangle, (a, \cdot, \sqcup, \cdot, \ldots, \sqcup, \cdot), \mathsf{R})$$
$$\delta'(\langle \text{init}, q_0, a, \mathsf{L}, 0, \mathsf{L}, \ldots, 0, \mathsf{L}\rangle, \sqcup) = (\langle \text{back-init}, q_0, 0, \mathsf{L}, \ldots, 0, \mathsf{L}\rangle, (a, \cdot, \sqcup, \cdot, \ldots, \sqcup, \cdot), \mathsf{L})$$

Ending Initialization After we have rewritten the tape, we move the head all the way back, and move to the next phase which is "reading". Here, the fact that we wrote a left end-marker will be useful in realizing, when we have gone all the way back. So formally,

$$\delta'(\langle \text{back-init}, q_0, 0, \mathsf{L}, \ldots, 0, \mathsf{L}\rangle, b) = (\langle \text{back-init}, q_0, 0, \mathsf{L}, \ldots, 0, \mathsf{L}\rangle, b, \mathsf{L})$$
$$\delta'(\langle \text{back-init}, q_0, 0, \mathsf{L}, \ldots, 0, \mathsf{L}\rangle, \triangleright) = (\langle \text{read}, q_0, 0, \mathsf{L}, \ldots, 0, \mathsf{L}\rangle, \triangleright, \mathsf{R})$$

where $b \neq \triangleright$

Reading Here we scan the tape to the right, and whenever we encounter a position, where there is a tape head (i.e., a $*$ in the appropriate position), we will remember that symbol in the state. When we reach the right end (i.e., read a $\sqcup$), we know all the information to determine the next step of M. We will remember the nexw symbols to right and directions of the head in the state, and move to the next phase back-read where we just go back all the way to the left end. These two cases are formally given as

- Suppose the current state is $P = \langle \text{read}, q, a_1, d_1, \ldots, a_i, d_i, \ldots, a_k, d_k\rangle$, and we read a symbol $X = (b_1, h_1, \ldots, b_i, h_i, \ldots, b_k, h_k)$, where if $h_i = *$ that means that the ith tape head is read this position, and if $h_i = \cdot$ then the ith tape head is not reading this position. Thus,
$$\delta'(P, X) = (\langle q, a'_1, d_1, \ldots, a'_i, d_i, \ldots, a'_k, d_k\rangle, X, \mathsf{R})$$
where $a'_i = a_i$ if $h_i = \cdot$ and $a'_i = b_i$ if $h_i = *$.
- Suppose the current state is $P = \langle \text{read}, q, a_1, d_1, \ldots, a_i, d_i, \ldots, a_k, d_k\rangle$, and we read $\sqcup$. This means we have finished scanning and all the symbols a_i are the symbols that are being read by M, and its state is q. So suppose M's transition function δ says
$$\delta(q, a_1, a_2, \ldots, a_k) = (q', b_1, d'_1, \ldots b_k, d'_k)$$
That is, it says "replace symbol a_i on tape i by b_i and move its head in direction d'_i". Then
$$\delta'(P, \sqcup) = (\langle \text{back-read}, q', b_1, d'_1, \ldots b_k, d'_k\rangle, \sqcup, \mathsf{L})$$

So the new state of M, symbols to be written and direction of heads in stored in the state and we go back.

Ending Reading After reading and determining the next step, we move the head all the way back, and move to the next phase which is fixing the tape to reflect the new situation. Here, the fact that we wrote a left end-marker will be useful in realizing when we have gone all the way back. So formally,

$$\delta'(\langle \text{back-read}, q, a_1, d_1, \ldots, a_k, d_k\rangle, b) = (\langle \text{back-read}, q, a_1, d_1, \ldots, a_k, d_k\rangle, b, \mathsf{L})$$
$$\delta'(\langle \text{back-read}, q, a_1, d_1, \ldots, a_k, d_k\rangle, \triangleright) = (\langle \text{fix-right}, q, a_1, d_1, \ldots, a_k, d_k\rangle, \triangleright, \mathsf{R})$$

where $b \neq \triangleright$.

Right Scan of Fixing In this phase we move all the way to the right. Along the way, we change the symbols to new symbols, wherever the old heads were, and move all heads that need to be moved right. To move a head right, we will use "$*$" in the state to remember that the old head position of the tape is in current cell, and it needs to be moved to the next cell. Finally, there is the boundary case of moving the head on some tape to right of the rightmost non-blank symbol on any tape. We will capture these cases formally.

- Let the current state of single(M) be $P = \langle \text{fix-right}, q, a_1, d_1, \ldots, a_k, d_k \rangle$ and let the symbol being read be $X = (b_1, h_1, \ldots b_k, h_k)$. In such a situation, single(M) will move to state $P' = \langle \text{fix-right}, q, a_1, d'_1, \ldots, a_k, d'_k \rangle$, move R and write $X' = (b'_1, h'_1, \ldots b'_k, h'_k)$. P' and X' are determined as follows. If $h_i = *$ (that is, tape i's head was here) and $d_i = \mathsf{R}$ then $b'_i = a_i$ (write new symbol), $h'_i = \cdot$ (new head is not here), and $d'_i = *$ (remember to put head in next cell). If $h_i = *$ and $d_i = \mathsf{L}$ then $b'_i = a_i$ (write new symbol), $h'_i = h_i$ (defer head movement to next phase), and $d'_i = d_i$. If $h_i = \cdot$ and $d_i = *$ (i.e., we remember new head position is here) then $b'_i = b_i$ (don't change symbol), $h'_i = *$ (new head is here), and $d'_i = \mathsf{R}$ (this was the original direction). If $h_i = \cdot$ and $d_i \neq *$ then $b'_i = b_i$ and $d'_i = d_i$.
- Consider the case when the state is $P = \langle \text{fix-right}, q, a_1, d_1, \ldots, a_k, d_k \rangle$, and $\sqcup$ is on the tape. There are two possibilities. If $d_i \neq *$ for every i then we move to state $P' = \langle \text{fix-left}, q, a_1, d_1, \ldots q_k, d_k \rangle$, write $\sqcup$ and move L. On the other hand, suppose there is some i such that $d_i = *$. Then we move to state $P' = \langle \text{fix-right}, q, a_1, d'_1, \ldots a_k, d'_k \rangle$, move R and write $X' = (b'_1, h'_1, \ldots b'_k, h'_k)$, where X' and P' are given as follows. First $b'_i = \sqcup$ for all i. Next, if $d_i = *$ then $h_i = *$ and $d'_i = \mathsf{R}$. On the other hand if $d_i \neq *$ then $h'_i = \cdot$ and $d'_i = d_i$.

Left Scan of Fixing In this phase we move all the way to the left, and along the way we move all the head positions that needed to be moved left. These changes are similar to the case of moving heads to the right, except for the case when moving a head left from the leftmost position.

- Let the current state of single(M) be $P = \langle \text{fix-left}, q, a_1, d_1, \ldots, a_k, d_k \rangle$ and let the symbol being read be $X = (b_1, h_1, \ldots b_k, h_k)$. In such a situation, single(M) will move to state $P' = \langle \text{fix-left}, q, a_1, d'_1, \ldots, a_k, d'_k \rangle$, move L and write $X' = (b_1, h'_1, \ldots b_k, h'_k)$. P' and X' are determined as follows. If $h_i = *$ and $d_i = \mathsf{L}$ (that is this tape's head needs to be moved left) then $h'_i = \cdot$ (new head is not here), and $d'_i = *$ (remember to put head in next cell). If $h_i = *$ and $d_i = \mathsf{R}$ then $h'_i = h_i$ and $d'_i = d_i$ (don't do anything since we already handled the right moves). If $h_i = \cdot$ and $d_i = *$ (i.e., we remember new head position is here) then $h'_i = *$ (new head is here), and $d'_i = \mathsf{L}$ (this was the original direction). If $h_i = \cdot$ and $d_i \neq *$ then $h'_i = h_i$ and $d'_i = d_i$.
- Now we consider the case when we finish the left scan, i.e., the symbol being read is $\triangleright$. Let the state be $P = \langle \text{fix-left}, q, a_1, d_1, \ldots, a_k, d_k \rangle$. Now there are two possibilities. Either there are no pending left moves, i.e., $d_i \neq *$ for all i, or there is a pending left move $d_i = *$ for some i. In the first case, we fixed all the moves correctly, and so we move to state $P' = \langle \text{read}, q, a_1, d_1, \ldots, a_k, d_k \rangle$, write $\triangleright$, and move R, and start simulating the next step. In the second case, we need to re-mark some head positions (since on left moves from the end of the tape, we stay there). We will do this in two steps. First single(M) will remain in state P, write $\triangleright$, and move R. In the next step, the transitions already defined, will place the heads correctly, move left in a state without pending head moves to read $\triangleright$ and will be handled by the previous case.

Acceptance/Rejection If M accepts/rejects (i.e., its state is q_{acc} or q_{rej}) then single(M) will

move to its accept/reject state.

$$\delta'(\langle \text{read}, q_{\mathsf{acc}}, a_1, d_1, \ldots, a_k, d_k \rangle, b) = (q'_{\mathsf{acc}}, b, \mathsf{L})$$
$$\delta'(\langle \text{read}, q_{\mathsf{rej}}, a_1, d_1, \ldots, a_k, d_k \rangle, b) = (q'_{\mathsf{rej}}, b, \mathsf{L})$$

3.2 Nondeterministic TM

Nondeterministic Turing Machine

Deterministic TM: At each step, there is one possible next state, symbols to be written and direction to move the head, or the TM may halt.

Nondeterministic TM: At each step, there are finitely many possibilities. So formally, $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}})$, where

- $Q, \Sigma, \Gamma, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}}$ are as before for 1-tape machine
- $\delta : (Q \setminus \{q_{\mathsf{acc}}, q_{\mathsf{rej}}\}) \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{\mathsf{L}, \mathsf{R}\})$

Computation, Acceptance and Language

- A configuration of a nondeterministic TM is exactly the same as that of a 1-tape TM. So are notions of starting configuration and accepting configuration.
- A single step $\vdash$ is defined similarly. $X_1X_2 \cdots X_{i-1}qX_i \cdots X_n \vdash X_1X_2 \cdots pX_{i-1}Y \cdots X_n$, if $(p, Y, L) \in \delta(q, X_i)$; case for right moves is analogous.
- w is *accepted* by M, if from the starting configuration with w as input, M reaches an accepting configuration, for some sequence of choices at each step.
- $L(M) = \{w \mid w \text{ accepted by } M\}$

Expressive Power of Nondeterministic TM

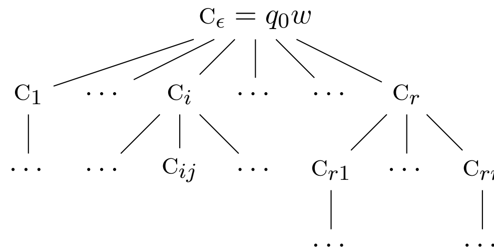
Theorem 3.2. *For any nondeterministic Turing Machine M, there is a (deterministic) TM $det(M)$ such that $L(det(M)) = L(M)$.*

Proof Idea

$\det(M)$ will simulate M on the input.

- Idea 1: $\det(M)$ tries to keep track of all possible "configurations" that M could possibly be after each step. Works for DFA simulation of NFA but not convenient here.
- Idea 2: $\det(M)$ will simulate M on each possible sequence of computation steps that M may try in each step.

Nondeterministic Computation



- If $r = \max_{q,X} |\delta(q, X)|$ then the runs of M can be organized as an r-branching tree.
- $c_{i_1 i_2 \cdots i_n}$ is the configuration of M after n-steps, where choice i_1 is taken in step 1, i_2 in step 2, and so on.
- Input w is accepted iff $\exists$ accepting configuration in tree.

Deterministic Simulation
Proof Idea

The machine det(M) will search for an accepting configuration in computation tree

- The configuration at any vertex can be obtained by simulating M on the appropriate sequence of nondeterministic choices
- det(M) will perform a BFS on the tree. Why not a DFS?

Observe that det(M) may not terminate if w is not accepted.

Proof Details

det(M) will use 3 tapes to simulate M (note, multitape TMs are equivalent to 1-tape TMs)

- Tape 1, called *input tape*, will always hold input w
- Tape 2, called *simulation tape*, will be used as M's tape when simulating M on a sequence of nondeterministic choices
- Tape 3, called *choice tape*, will store the current sequence of nondeterministic choices

---

Execution of det(M)

1. Initially: Input tape contains w, simulation tape and choice tape are blank
2. Copy contents of input tape onto simulation tape
3. Simulate M using simulation tape as its (only) tape
   (a) At the next step of M, if state is q, simulation tape head reads X, and choice tape head reads i, then simulate the ith possibility in $\delta(q, X)$; if i is not a valid choice, then goto step 4
   (b) After changing state, simulation tape contents, and head position on simulation tape, move choice tape's head to the right. If Tape 3 is now scanning $\sqcup$, then goto step 4
   (c) If M accepts then accept and halt, else goto step 3(1) to simulate the next step of M.
4. Write the lexicographically next choice sequence on choice tape, erase everything on simulation tape and goto step 2.

Deterministic Simulation
In a nutshell

- $\det(M)$ simulates M over and over again, for different sequences, and for different number of steps.
- If M accepts w then there is a sequence of choices that will lead to acceptance. $\det(M)$ will eventually have this sequence on choice tape, and then its simulation M will accept.
- If M does not accept w then no sequence of choices leads to acceptance. $\det(M)$ will therefore never halt!

3.3 Random Access Machine

Random Access Machines

This is an idealized model of modern computers. Have a finite number of "registers", an infinite number of available memory locations, and store a sequence of instructions or "program" in memory.

- Initially, the program instructions are stored in a contiguous block of memory locations starting at location 1. All registers and memory locations, other than those storing the program, are set to 0.

Instruction Set

- `add X, Y`: Add the contents of registers X and Y and store the result in X.
- `loadc X, I`: Place the constant I in register X.

- `load X, M`: Load the contents of memory location M into register X.
- `loadI X, M`: Load the contents of the location "pointed to" by the contents of M into register X.
- `store X, M`: store the contents of register X in memory location M.
- `jmp M`: The next instruction to be executed is in location M.
- `jmz X, M`: If register X is 0, then jump to instruction M.
- `halt`: Halt execution.

Expressive Power of RAMs

Theorem 3.3. *Anything computed on a RAM can be computed on a Turing machine.*

Proof. Proof sketch in the notes. □

Capturing State of RAM

In order to simulate the RAM, the TM stores contents of registers, memory etc., in different tapes as follows.

- Instruction Counter Tape: Stores the memory location where the next instrcution is stored; initially it is 1.
- Memory Address Tape: Stores the address of memeory location where a load/store operation is to be performed.
- Register Tape: Stores the contents of each of the registers.
  - Has register index followed by contents of each register as follows: $\#\langle\text{RegisterNumber}\rangle * \langle\text{RegisterValue}\rangle\# \cdots$. For example, if register 1 has 11, register 2 has 100, register 3 has 11011, etc, then tape contains $\#1 * 11\#10 * 100\#11 * 11011\# \cdots$
- Memory Tape: Like register tape, store $\#\langle\text{Address}\rangle * \langle\text{Contents}\rangle\#$
  - To store an instruction, have opcode, $\langle\text{arguments}\rangle$
- Work Tapes: Have 3 additional work tapes to simulate steps of the RAM

Simulating a RAM

- TM starts with the program stored in memory, and the instruction location tape initalized to 1.

- Each step of the RAM is simulated using many steps.
  - Read the instruction counter tape
  - Search for the relevant instruction in memory
  - Store the opcode of instruction and register address (of argument) in the finite control. Store the memory addres (of argument) in memory address tape.
  - Retrieve the values from register tape and/or memory tape and store them in work tape
  - Perform the operation using work tapes
  - Update instruction counter tape, register tape, and memory tape.

Example: ADD instruction

- Suppose instruction counter tape holds 101.
- TM searches memory tape for the pattern $\#101*$.
- Suppose the memory tape contains $\cdots\#101*\langle\texttt{add}\rangle, 11, 110\#\cdots$
- TM stores "add", 11 and 110 in its finite control. In other words, it moves to a state $q_{\texttt{add}\ 11,110}$ whose job it is to add the contents of register 11 and 110 and put the result in 11.
- Search the register tape for the pattern $\#11*$. Suppose the register tape contains $\cdots\#11*10110\#\cdots$; in other words, the contents of register 11 is 10110. Copy 10110 to one of the work-tapes.
- Search the register tape for pattern $\#101*$, and copy the contents of register onto work tape 2.
- Compute the sum of the contents of the work tapes
- Search the register tape for $\#11*$ and replace the string 10110 by the answer computed on the work tape. This may involve shifting contents of the register tape to the right (or left).
- Add 1 to the instruction counter tape.

---

4 Church-Turing Thesis

4.1 Universality of the Model

Robustness of the Class of TM Languages

Various efforts to capture mechanical computation have the same expressive power.

- Non-Turing Machine models: random access machines, λ-calculus, type 0 grammars, first-order reasoning, π-calculus, ...

- Enhanced Turing Machine models: TM with 2-way infinite tape, multi-tape TM, nondeterministic TM, probabilistic Turing Machines, quantum Turing Machines ...
- Restricted Turing Machine models: queue machines, 2-stack machines, 2-counter machines, ...

4.2 Church-Turing Thesis

Church-Turing Thesis

"Anything solvable via a mechanical procedure can be solved on a Turing Machine."

- Not a mathematical statement that can be proved or disproved!
- Strong evidence based on the fact that many attempts to define computation yield the same expressive power

---

Consequences

- In the course, we will use an informal pseudo-code to argue that a problem/language can be solved on Turing machines
- To show that something can be solved on Turing machines, you can use any programming language of choice, *unless the problem specifically asks you to design a Turing Machine*

Part III
Lecture 22

Decidability and Recognizability

5 High-Level Descriptions of Computation

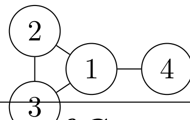
High-Level Descriptions of Computation

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

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
- 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, PDAs, regular expressions, grammars and Turing Machines
- e.g.: Convert a CFG to Chomsky Normal Form; Given a CFG G and a word w, decide if $w \in L(G)$; ...
- 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

High-Level Descriptions of Computation
Examples: Problems regarding Computation

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, PDA 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

---

6 Deciding vs. Recognizing

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 e.g., 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.

6.1 An Undecidable but Recognizable Language

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*!

  Proposition 6.1. *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)
```

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

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

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}.

6.2 Complementation

Deciding vs. Recognizing

Proposition 6.2. *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)
```

(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
- Is every language recognizable? *No!*

Proposition 6.3. $\overline{A_{\text{TM}}}$ *is unrecognizable*

Proof. If $\overline{A_{\text{TM}}}$ is recognizable, since A_{TM} is recognizable, the two languages will be decidable too! ☐

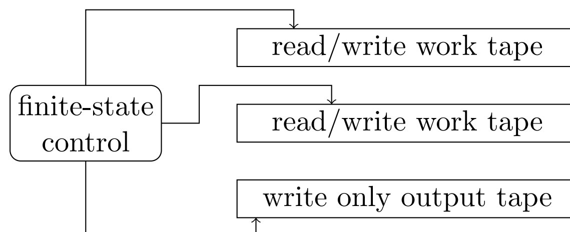
Note: Decidable languages are closed under complementation, but recognizable languages are not.

---

7 Recursive Enumeration

7.1 Enumerators

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 7.1. 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 7.2. L is *recursively enumerable (r.e.)* iff there is an enumerator M such that $L = E(M)$.

7.2 Equivalence of Enumerating and Recognizing a Language

Recursively Enumerable Languages and TMs

Theorem 7.3. *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 $L(M) = E(N)$. 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?

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. Will never reach step two in any execution!

□

Enumerators From Recognizers
Dovetailing

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! □

Part IV

Lecture 23

Undecidability and Reductions

8 Undecidability

8.1 Recap

Decision Problems and Languages

- A *decision problem* requires checking if an input (string) has some property. Thus, a decision problem is a function from `strings` to `boolean`.
- A decision problem is represented as a *formal language* consisting of those strings (inputs) on which the answer is "yes".

Recursive Enumerability

- A Turing Machine on an input w either (halts and) accepts, or (halts and) rejects, or never halts.
- The language of a Turing Machine M, denoted as $L(M)$, is the set of all strings w on which M accepts.
- A language L is *recursively enumerable/Turing recognizable* if there is a Turing Machine M such that $L(M) = L$.

Decidability

- A language L is *decidable* if there is a Turing machine M such that $L(M) = L$ and M halts on every input.
- Thus, if L is decidable then L is recursively enumerable.

Undecidability

Definition 8.1. A language L is *undecidable* if L is not decidable. Thus, there is no Turing machine M that halts on every input and $L(M) = L$.

- This means that either L is not recursively enumerable. That is there is no turing machine M such that $L(M) = L$, or
- L is recursively enumerable but not decidable. That is, every Turing machine M such that $L(M) = L$, M does not halt on some inputs.

Big Picture

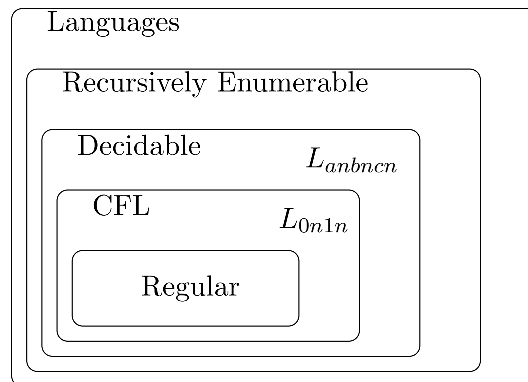


Figure 7: Relationship between classes of Languages

Machines as Strings

- For the rest of this lecture, let us fix the input alphabet to be $\{0, 1\}$; a string over any alphabet can be encoded in binary.
- Any Turing Machine/program M can itself be encoded as a binary string. Moreover every binary string can be thought of as encoding a TM/program. (If not the correct format, considered to be the encoding of a default TM.)
- We will consider a decision problem (language) whose input is a Turing Machine (encoded as a binary string)

8.2 Diagonalization

The Diagonal Language

Definition 8.2. Define $L_d = \{M \mid M \notin L(M)\}$. Thus, L_d is the collection of Turing machines (programs) M such that M does not halt and accept when given itself as input.

A non-Recursively Enumerable Language

Proposition 8.3. *L_d is not recursively enumerable.*

Proof. Recall that,

- Inputs are strings over $\{0,1\}$
- Every Turing Machine can be described by a binary string and every binary string can be viewed as Turing Machine
- In what follows, we will denote the ith binary string (in lexicographic order) as the number i. Thus, we can say $j \in L(i)$, which means that the Turing machine corresponding to ith binary string accepts the jth binary string. □

Completing the proof
Diagonalization: Cantor

Proof (contd). We can organize all programs and inputs as a (infinite) matrix, where the (i,j)th entry is Y if and only if $j \in L(i)$.

TMs ↓ \ Inputs ⟶	1	2	3	4	5	6	7	⋯
1	[N]	N	N	N	N	N	N	
2	N	[N]	N	N	N	N	N	
3	Y	N	[Y]	N	Y	Y	Y	
4	N	Y	N	[Y]	Y	N	N	
5	N	Y	N	Y	[Y]	N	N	
6	N	N	Y	N	Y	[N]	Y	

Suppose L_d is recognized by a Turing machine, which is the jth binary string. i.e., $L_d = L(j)$. But $j \in L_d$ iff $j \notin L(j)$! □

Acceptor for L_d?

Consider the following program

```
On input i
    Run program i on i
    Output ``yes'' if i does not accept i
    Output ``no'' if i accepts i
```

Does the above program recognize L_d? No, because it may never output "yes" if i does not halt on i.

Models for Decidable Languages

Question
Is there a machine model such that

- all programs in the model halt on all inputs, and
- for each problem decidable by a TM, there is a program in the model that decides it?

Models for Decidable Languages

Answer
There is no such model! Suppose a programming language in which all programs always halt. Programs in this language can be described by binary strings, and can be simulated by TMs. Consider the Turing Machine M_d

```
On input i
    Run program i on i
    Output ‘‘yes’’ if i does not accept i
    Output ‘‘no’’ if i accepts i
```

M_d always halts and solves a problem not solved by any program in our language! Inability to halt is *essential* to capture all computation.

8.3 The Universal Language

Recursively Enumerable but not Decidable

- L_d not recursively enumerable, and therefore not decidable. Are there languages that are recursively enumerable but not decidable?
- Yes, $A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$

The Universal Language

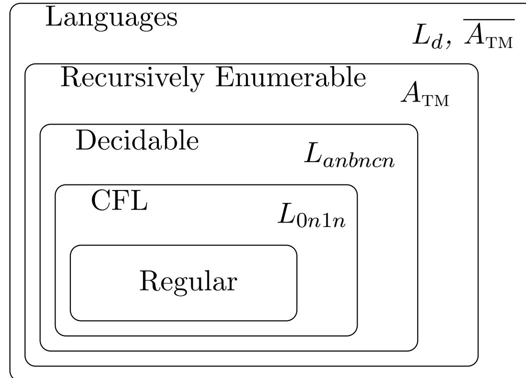
Proposition 8.4. *A_{TM} is r.e. but not decidable.*

Proof. We have already seen that A_{TM} is r.e. Suppose (for contradiction) A_{TM} is decidable. Then there is a TM M that always halts and $L(M) = A_{\text{TM}}$. Consider a TM D as follows:

```
On input i
    Run M on input ⟨i,i⟩
    Output ‘‘yes’’ if i rejects i
    Output ‘‘no’’ if i accepts i
```

Observe that $L(D) = L_d$! But, L_d is not r.e. which gives us the contradiction. □

A more complete Big Picture



---

9 Reductions

9.1 Informal Overview

Reductions

A *reduction* is a way of converting one problem into another problem such that a solution to the second problem can be used to solve the first problem. We say the first problem *reduces* to the second problem.

- Informal Examples: Measuring the area of rectangle reduces to measuring the length of the sides; Solving a system of linear equations reduces to inverting a matrix
- The problem L_d reduces to the problem A_{TM} as follows: "To see if $w \in L_d$ check if $\langle w, w\rangle \in A_{\text{TM}}$."

Undecidability using Reductions

Proposition 9.1. *Suppose L_1 reduces to L_2 and L_1 is undecidable. Then L_2 is undecidable.*

Proof Sketch.
Suppose for contradiction L_2 is decidable. Then there is a M that always halts and decides L_2. Then the following algorithm decides L_1

- On input w, apply reduction to transform w into an input w' for problem 2
- Run M on w', and use its answer.

Schematic View

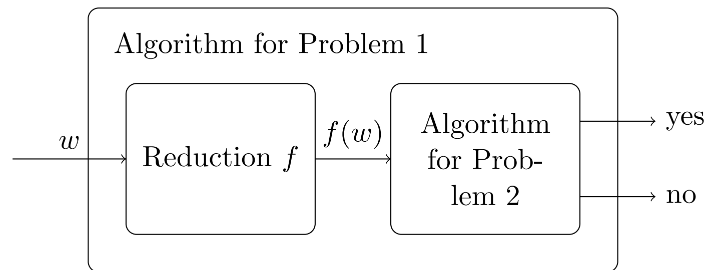


Figure 8: Reductions schematically

The Halting Problem

Proposition 9.2. *The language* $HALT = \{\langle M, w\rangle \mid M$ *halts on input* $w\}$ *is undecidable.*

Proof. We will reduce A_{TM} to HALT. Based on a machine M, let us consider a new machine $f(M)$ as follows:

```
On input x
    Run M on x
    If M accepts then halt and accept
    If M rejects then go into an infinite loop
```

Observe that $f(M)$ halts on input w if and only if M accepts w □

The Halting Problem

Completing the proof

Proof (contd). Suppose HALT is decidable. Then there is a Turing machine H that always halts and $L(H) = \text{HALT}$. Consider the following program T

```
On input ⟨M, w⟩
    Construct program f(M)
    Run H on ⟨f(M), w⟩
    Accept if H accepts and reject if H rejects
```

T decides A_{TM}. But, A_{TM} is undecidable, which gives us the contradiction. □

9.2 Definition and Properties

Mapping Reductions

Definition 9.3. A function $f : \Sigma^* \to \Sigma^*$ is *computable* if there is some Turing Machine M that on every input w halts with $f(w)$ on the tape.

Definition 9.4. A *mapping/many-one* reduction from A to B is a computable function $f : \Sigma^* \to \Sigma^*$ such that

$$w \in A \text{ if and only if } f(w) \in B$$

In this case, we say A is *mapping/many-one reducible* to B, and we denote it by $A \leq_m B$.

Convention

In this course, we will drop the adjective "mapping" or "many-one", and simply talk about reductions and reducibility.

Reductions and Recursive Enumerability

Proposition 9.5. *If $A \leq_m B$ and B is recursively enumerable then A is recursively enumerable.*

Proof. Let f be the reduction from A to B and let M_B be the Turing Machine recognizing B. Then the Turing machine recognizing A is

```
On input w
    Compute f(w)
    Run M_B on f(w)
    Accept if M_B does and reject if M_B rejects
```

□

Reductions and non-r.e.

Corollary 9.6. *If $A \leq_m B$ and A is not recursively enumerable then B is not recursively enumerable.*

Reductions and Decidability

Proposition 9.7. *If $A \leq_m B$ and B is decidable then A is decidable.*

Proof. Let M_B be the Turing machine deciding B and let f be the reduction. Then the algorithm deciding A, on input w, computes $f(w)$ and runs M_B on $f(w)$. □

Corollary 9.8. *If $A \leq_m B$ and A is undecidable then B is undecidable.*

9.3 Examples

Emptiness of Turing Machines

Proposition 9.9. *The language* $E_{\text{TM}} = \{M \mid L(M) = \emptyset\}$ *is undecidable.*

Proof. L_d is reducible to E_{TM} as follows. For machine M construct the following Turing machine $f(M)$

```
On input x
    Run M on M for |x| steps
    Accept x only if M accepts M within |x| steps
```

Observe that $L(f(M)) = \emptyset$ if and only if M does not accept M if and only if $M \in L_d$. □

Checking Regularity

Proposition 9.10. *The language* $REGULAR = \{M \mid L(M) \text{ is regular}\}$ *is undecidable.*

Proof. A_{TM} is reducible to REGULAR as follows. For $\langle M, w\rangle$ construct the following Turing machine $f(\langle M, w\rangle)$

```
On input x
    If x is of the form 0^n1^n then accept x
    else run M on w and accept x only if M does
```

If $w \in L(M)$ then $L(f(\langle M, w\rangle)) = \Sigma^*$. If $w \notin L(M)$ then $L(f(\langle M, w\rangle)) = \{0^n1^n \mid n \geq 0\}$. Thus, $f(\langle M, w\rangle) \in$ REGULAR if and only if $\langle M, w\rangle \in A_{\text{TM}}$ □

Part V
Lecture 24

Reductions and Rice's Theorem

10 Reductions

Mapping Reductions

Definition 10.1. A function $f : \Sigma^* \to \Sigma^*$ is *computable* if there is some Turing Machine M that on every input w halts with $f(w)$ on the tape.

Definition 10.2. A *reduction* (a.k.a mapping reduction/many-one reduction) from a language A to a language B is a computable function $f : \Sigma^* \to \Sigma^*$ such that

$$w \in A \text{ if and only if } f(w) \in B$$

In this case, we say A is *reducible* to B, and we denote it by $A \leq_m B$.

Reductions and Recursive Enumerability

Proposition 10.3. *If $A \leq_m B$ and B is r.e., then A is r.e.*

Proof. Let f be a reduction from A to B and let M_B be a Turing Machine recognizing B. Then the Turing machine recognizing A is

```
On input w
    Compute f(w)
    Run M_B on f(w)
    Accept if M_B accepts, and reject if M_B rejects
```

□

Corollary 10.4. *If $A \leq_m B$ and A is not r.e., then B is not r.e.*

Reductions and Decidability

Proposition 10.5. *If $A \leq_m B$ and B is decidable, then A is decidable.*

Proof. Let f be a reduction from A to B and let M_B be a Turing Machine *deciding* B. Then a Turing machine that decides A is

```
On input w
    Compute f(w)
    Run M_B on f(w)
    Accept if M_B accepts, and reject if M_B rejects
```

□

Corollary 10.6. *If $A \leq_m B$ and A is undecidable, then B is undecidable.*

10.1 Examples

The Halting Problem

Proposition 10.7. *The language HALT* $= \{\langle M, w\rangle \mid M$ *halts on input* $w\}$ *is undecidable.*

Proof. Recall $A_{\text{TM}} = \{\langle M, w\rangle \mid w \in L(M)\}$ is undecidable. Will give reduction f to show $A_{\text{TM}} \leq_m$ HALT $\implies$ HALT undecidable.

Let $f(\langle M, w\rangle) = \langle N, w\rangle$ where N is a TM that behaves as follows:

```
On input x
    Run M on x
    If M accepts then halt and accept
    If M rejects then go into an infinite loop
```

N halts on input w if and only if M accepts w. i.e., $\langle M, w\rangle \in A_{\text{TM}}$ iff $f(\langle M, w\rangle) \in$ HALT □

Emptiness of Turing Machines

Proposition 10.8. *The language* $E_{\text{TM}} = \{\langle M\rangle \mid L(M) = \emptyset\}$ *is not r.e.*

Proof. Recall $L_d = \{\langle M\rangle \mid \langle M\rangle \notin L(M)\}$ is not r.e.

L_d is reducible to E_{TM} as follows. Let $f(\langle M\rangle) = \langle N\rangle$ where N is a TM that behaves as follows:

```
On input x
    Run M on ⟨M⟩ for |x| steps
    Accept x only if M accepts ⟨M⟩ within |x| steps
```

Observe that $L(N) = \emptyset$ if and only if M does not accept $\langle M\rangle$ if and only if $\langle M\rangle \in L_d$. □

Checking Regularity

Proposition 10.9. *The language REGULAR* $= \{\langle M\rangle \mid L(M)$ *is regular*$\}$ *is undecidable.*

Proof. We give a reduction f from A_{TM} to REGULAR. Let $f(\langle M, w\rangle) = \langle N\rangle$, where N is a TM that works as follows:

```
On input x
    If x is of the form 0^n1^n then accept x
    else run M on w and accept x only if M does
```

If $w \in L(M)$ then $L(N) = \Sigma^*$. If $w \notin L(M)$ then $L(N) = \{0^n1^n \mid n \geq 0\}$. Thus, $\langle N\rangle \in$ REGULAR if and only if $\langle M, w\rangle \in A_{\text{TM}}$ □

---

11 Rice's Theorem

Checking Properties

Given $\langle M \rangle$

$$\left.\begin{array}{l}\text{Does } L(M) \text{ contain } \langle M \rangle? \\ \text{Is } L(M) \text{ non-empty?} \\ \text{Is } L(M) \text{ empty?}\end{array}\right\} \text{Undecidable}$$

$$\left.\begin{array}{l}\text{Is } L(M) \text{ infinite?} \\ \text{Is } L(M) \text{ finite?} \\ \text{Is } L(M) \text{ co-finite (i.e., is } \overline{L(M)} \text{ finite)?} \\ \text{Is } L(M) = \Sigma^*?\end{array}\right\} \text{Undecidable}$$

Which of these properties can be decided? None! By *Rice's Theorem*

Properties

Definition 11.1. A property of languages is simply a set of languages. We say L *satisfies* the property $\mathbb{P}$ if $L \in \mathbb{P}$.

Definition 11.2. For any property $\mathbb{P}$, define language $L_{\mathbb{P}}$ to consist of Turing Machines which accept a language in $\mathbb{P}$:

$$L_{\mathbb{P}} = \{\langle M \rangle \mid L(M) \in \mathbb{P}\}$$

Deciding $L_{\mathbb{P}}$: deciding if a language represented as a TM satisfies the property $\mathbb{P}$.

- *Example:* $\{\langle M \rangle \mid L(M) \text{ is infinite}\}$; $E_{\text{TM}} = \{\langle M \rangle \mid L(M) = \emptyset\}$
- *Non-example:* $\{\langle M \rangle \mid M \text{ has 15 states}\}$ $\longleftarrow$ This is a property of TMs, and not languages!

Trivial Properties

Definition 11.3. A property is trivial if either it is not satisfied by any r.e. language, or if it is satisfied by all r.e. languages. Otherwise it is non-trivial.

Example 11.4. Some trivial properties:

- $\mathbb{P}_{\text{ALL}}$ = set of all languages
- $\mathbb{P}_{\text{R.E.}}$ = set of all r.e. languages
- $\overline{\mathbb{P}}$ where $\mathbb{P}$ is trivial
- $\mathbb{P} = \{L \mid L \text{ is recognized by a TM with an even number of states}\} = \mathbb{P}_{\text{R.E.}}$

Observation. For any trivial property $\mathbb{P}$, $L_{\mathbb{P}}$ is decidable. (Why?) Then $L_{\mathbb{P}} = \Sigma^*$ or $L_{\mathbb{P}} = \emptyset$.

Rice's Theorem

Proposition 11.5. *If $\mathbb{P}$ is a non-trivial property, then $L_{\mathbb{P}}$ is undecidable.*

- Thus $\{\langle M \rangle \mid L(M) \in \mathbb{P}\}$ is not decidable (unless $\mathbb{P}$ is trivial)

We cannot algorithmically determine any interesting property of languages represented as Turing Machines!

Properties of TMs

Note. Properties of TMs, as opposed to those of languages they accept, may or may not be decidable.

Example 11.6.

$\{\langle M \rangle \mid M$ has 193 states$\}$
$\{\langle M \rangle \mid M$ uses at most 32 tape cells on blank input$\}$ — Decidable

$\{\langle M \rangle \mid M$ halts on blank input$\}$
$\{\langle M \rangle \mid$ on input 0011 M at some point writes the symbol \$ on its tape$\}$ — Undecidable

---

Proof of Rice's Theorem

Rice's Theorem
If $\mathbb{P}$ is a non-trivial property, then $L_{\mathbb{P}}$ is undecidable.

Proof.

- Suppose $\mathbb{P}$ non-trivial and $\emptyset \notin \mathbb{P}$.
  - (If $\emptyset \in \mathbb{P}$, then in the following we will be showing $L_{\overline{\mathbb{P}}}$ is undecidable. Then $L_{\mathbb{P}} = \overline{L_{\overline{\mathbb{P}}}}$ is also undecidable.)
- Recall $L_{\mathbb{P}} = \{\langle M \rangle \mid L(M)$ satisfies $\mathbb{P}\}$. *We'll reduce A_{TM} to $L_{\mathbb{P}}$.*
- Then, since A_{TM} is undecidable, $L_{\mathbb{P}}$ is also undecidable. □

Proof of Rice's Theorem

Proof (contd). Since $\mathbb{P}$ is non-trivial, at least one r.e. language satisfies $\mathbb{P}$. i.e., $L(M_0) \in \mathbb{P}$ for some TM M_0.

Will show a reduction f that maps an instance $\langle M, w \rangle$ for A_{TM}, to $\langle N \rangle$ such that

- If M accepts w then N accepts the same language as M_0.
  - Then $L(N) = L(M_0) \in \mathbb{P}$
- If M does not accept w then N accepts $\emptyset$.
  - Then $L(N) = \emptyset \notin \mathbb{P}$

Thus, $\langle M, w\rangle \in A_{\text{TM}}$ iff $\langle N\rangle \in L_{\mathbb{P}}$. □

Proof of Rice's Theorem

Proof (contd). The reduction f maps $\langle M, w\rangle$ to $\langle N\rangle$, where N is a TM that behaves as follows:

```
On input x
    Ignore the input and run M on w
    If M does not accept (or doesn't halt)
        then do not accept x (or do not halt)
    If M does accept w
        then run M0 on x and accept x iff M0 does.
```

Notice that indeed if M accepts w then $L(N) = L(M_0)$. Otherwise $L(N) = \emptyset$.

□

---

12 Closure Properties

12.1 Closure of Decidable Languages

Closure of Decidable Languages

Proposition 12.1. *Decidable languages are closed under union, intersection, complementation, concatenation, kleene star, and inverse homomorphism*

Proof. Given TMs M_1, M_2 that decide languages L_1, L_2

- A TM that decides $L_1 \cup L_2$: on input x, run M_1 and M_2 on x, and accept iff either accepts. (Similarly for intersection.)
- A TM that decides $\overline{L_1}$: On input x, run M_1 on x, and accept if M_1 rejects, and reject if M_1 accepts.

□

Closure of Decidable Languages

Proof (contd).

- A TM to decide L_1L_2: On input x, for each of the $|x| + 1$ ways to divide x as yz: run M_1 on y and M_2 on z, and accept if both accept. Else reject.
- A TM to decide L_1^*: On input x, if $x = \epsilon$ accept. Else, for each of the $2^{|x|-1}$ ways to divide x as $w_1 \ldots w_k$ ($w_i \neq \epsilon$): run M_1 on each w_i and accept if M_1 accepts all. Else reject.

- A TM to decide $h^{-1}(L_1)$: On input x, compute $h(x)$ and run M_1 on $h(x)$; accept iff M_1 accepts.

□

Closure of Decidable Languages

Proposition 12.2. *Decidable languages are not closed under homomorphism*

Proof. We will show a decidable language L and a homomorphism h such that $h(L)$ is undecidable

- Let $L = \{xy \mid x \in \{0,1\}^*, y \in \{a,b\}^*, x = \langle M, w\rangle$, and y encodes an integer n such that the TM M on input w will halt in n steps $\}$
- L is decidable: can simply simulate M on input w for n steps
- Consider homomorphism h: $h(0) = 0$, $h(1) = 1$, $h(a) = h(b) = \epsilon$.
- $h(L) =$ HALT which is undecidable.

□

12.2 Closure of Recursively Enumerable Languages

Closure of Recursively Enumerable Languages

Proposition 12.3. *R.e. languages are closed under union, intersection, concatenation, kleene star, homomorphism and inverse homomorphism.*

Proof. Given TMs M_1, M_2 that recognize languages L_1, L_2

- A TM that recognizes $L_1 \cup L_2$: on input x, run M_1 and M_2 on x *in parallel*, and accept iff either accepts. (Similarly for intersection; but no need for parallel simulation)

□

Closure of Recursively Enumerable Languages

Proof (contd).

- A TM to recognize L_1L_2: On input x, do *in parallel*, for each of the $|x| + 1$ ways to divide x as yz: run M_1 on y and M_2 on z, and accept if both accept. Else reject.
- A TM to recognize L_1^*: On input x, if $x = \epsilon$ accept. Else, do *in parallel*, for each of the $2^{|x|-1}$ ways to divide x as $w_1 \ldots w_k$ ($w_i \neq \epsilon$): run M_1 on each w_i and accept if M_1 accepts all. Else reject.

- A TM to recognize $h^{-1}(L_1)$: On input x, compute $h(x)$ and run M_1 on $h(x)$; accept iff M_1 accepts.
- A TM to recognize $h(L_1)$: On input x, start going through all strings w, and if $h(w) = x$, start executing M_1 on w, using *dovetailing* to interleave with other executions of M_1. Accept if any of the executions accepts.

□

Closure of Recursively Enumerable Languages

Proposition 12.4. *R.e. languages are not closed under complementation.*

Proof. We saw that A_{TM} is r.e. but $\overline{A_{\text{TM}}}$ is not. □

Also we saw the following:

Proposition 12.5. *L is decidable iff L and $\overline{L}$ are both r.e.*

The Big Picture