You Are Here!

Reg = Regular Languages
- Deterministic Finite Automaton \(0^*1^*\)
- Nondeterministic Finite Automaton \((01)^*\)
- Regular Expression \(0^*1^*+(01)^*\)
- Right-Linear Grammar

CFL = Context-Free Language
- Context-Free Grammar \(0^n1^n\)
- Nondeterministic Pushdown Automaton \(ww^R\)

Dec = Recursive (Turing-Decidable) Language
- Turing Machine \(0^n1^n2^n\)
- Unrestricted Grammar \(ww\)

RE = Recursively Enumerable (Turing-Recognizable/Acceptable) Language

Lan = All Languages
Early Theory of Computation

- In the 1920s - 1940s, before the advent of modern computing machines, mathematicians were wrestling with the notion of effective computation: formalisms for expressing algorithms.

- Many formalisms evolved:
  - Turing Machines (Turing);
  - \(\lambda\)-calculus (Church, Kleene);
  - combinatory logic (Schönfinkel, Curry);
  - Post systems (Post);
  - \(\mu\)-recursive functions (Gödel, Herbrand).

- All of these formalisms were proven to be equivalent to each other!

The Church-Turing Thesis

- Computability is the common spirit embodied by this collection of formalisms.

- This thesis is a claim that is widely believed about the intuitive notions of algorithm and effective computation. It is not a theorem that can be proved.

- Because of their similarity to later computer hardware, Turing machines have become the gold standard for effectively computable.

- We’ll see in CS251 that the \(\lambda\)-calculus formalism is the foundation of modern programming languages.

- A consequence: programming languages all have the "same" computational "power" in term of what they can express. (But it may be "easier" or more efficient to use one than another.)
Endgame for this Course

- Lec #33 (today): Define Turing Machines and give examples of them (including the non-CFLs $ww$ and $a^nb^nc^n$).
- Lec #34: Parser generators (out-of-order wrap-up of parsing material).
- Lec #35: Show that adding various features to Turing Machines does not increase their power.
- Lec #36: Uncomputability: show that some languages cannot be decided by Turing Machines.
- Lec #37: Reduction techniques for proving languages uncomputable.
- Lec #38: Some classic examples of uncomputability involving context-free languages and programming languages.
- Lec #39: Course wrap-up.

What Is A Turing Machine?

Model of computation proposed by Alan Turing in 1936:
- A one-way infinite tape of cells holding symbols or blanks.
- A tape head that can read/write the symbol in the cell under it.
- A deterministic finite-state controller that, based on the current state and symbol under the tape head, writes a symbol under the tape head and moves it left or right.

Execution begins in the initial state with a string written on the tape starting at the leftmost cell:
- The string is accepted if the controller enters the accept state.
- The string is rejected if the controller enters the reject state.
- The machine may also loop -- continue processing the string forever without accepting or rejecting it.
Informal TM Example: \{w\#w \mid w \in \{a,b\}^*\}

1. If there is no unprocessed symbol in the left substring:
   a. If there is no unprocessed symbol in the right-hand substring, accept.
   b. Otherwise, reject.

   Otherwise, cross off first unprocessed symbol in the left substring and remember it.

2. Move to first unprocessed symbol in the right substring.
   a. If there is no such symbol (all the right-hand symbols have been crossed off) or the symbol does not match the remembered one, reject.
   b. If the symbol matches the remembered one, cross it off.

3. Move back to the first unprocessed symbol in the left substring (if there is one, or back to \#, otherwise) and go to step 1.

---

TM Transition Function

The core of a Turing Machine specification is the definition of a deterministic transition function \( \delta \):

\[(current-state, current-symbol) \rightarrow (next-state, new-symbol, left-or-right)\]

The transition \((q,\sigma) \rightarrow (q', \sigma', d)\) is typically represent as either:

- An entry in a 2D table keyed by \(q\) and \(\sigma\);
- A state transition in a diagram:

Abbreviations:

\[q \xrightarrow{\sigma \rightarrow d} q'\] means \[q \xrightarrow{\sigma \rightarrow \sigma', d} q'\]

\[q \xrightarrow{\sigma_1, \sigma_2 \rightarrow d} q'\] means \[q \xrightarrow{\sigma_1 \rightarrow \sigma_1, d} q'\]

\[q \xrightarrow{\sigma_2 \rightarrow \sigma_2, d} q'\]
TM State Transitions for \( \{w\#w \mid w \in \{a,b\}^*\} \)

This picture is adapted from p. 145 of Sipser.

TM State Transitions w/Implicit Reject State

This picture is adapted from p. 145 of Sipser.
**Formal Definition of a Turing Machine**

A deterministic one-tape Turing machine is a septuple

\[ \text{TM} = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}}) \]

where

1. \( Q \) is a finite set of states;
2. \( \Sigma \) is a finite input alphabet, which does not include the blank symbol, \( \Box \);
3. \( \Gamma \) is a finite tape alphabet, where \( (\Sigma \cup \{ \Box \}) \subseteq \Gamma \);
4. \( \delta \in (Q \times \Gamma) \to (Q \times \Gamma \times \{L, R\}) \) is the transition function;
5. \( q_{\text{start}} \in Q \) is the start state;
6. \( q_{\text{accept}} \in Q \) is the accept state;
7. \( q_{\text{reject}} \in Q \) is the reject state, \( q_{\text{reject}} \neq q_{\text{accept}} \).

(This is Sipser's formalism for a Turing machine. There are many variants.)

---

**Turing Machine Configurations**

A configuration specifies the state and tape head position of a TM.

E.g. \( \text{xbb}\#q_{4}\text{abb} \)

The transition function defines a relation \( \Rightarrow \) between configurations:

E.g. \( \text{xbb}\#q_{4}\text{abb} \Rightarrow \text{xbb}q_{6}\#\text{abb} \)

A Turing Machine accepts a string \( w \) iff \( q_{\text{start}}w \Rightarrow^{*} \ldots \) \( q_{\text{accept}} \ldots \)

A Turing Machine rejects a string \( w \) iff \( q_{\text{start}}w \Rightarrow^{*} \ldots \) \( q_{\text{reject}} \ldots \)

The language of TM, \( L(\text{TM}) = \) all strings accepted by TM.
Comparison: TMs vs. FAs and PDAs

- TM is necessarily deterministic.
- TM can both read and write input cells.
- The tape head can move both left and right.
- The tape is infinite.
- Accept/reject states take effect immediately (don’t have to read/consume all input).
- In all three kinds of machines, the controller is finite-state!

What about \{ww | w \in \{a,b\}^*\}?

One idea: first transform a b b a b b to a b b a b b

where there are 4 new symbols:

\{a, b, a, b\}

what to do now?
Subtlety: Detecting Left End of Tape

An attempt to move left from the leftmost tape cell leaves the position of the tape head unchanged.

How can the left end of the tape be detected? Two ways:

1. Put a special left-end marker symbol (often written $\vdash$) in the leftmost cell at the beginning of the computation. Sometimes another symbol (such as $x$ in the $w\#w$ example) can serve this purpose.

2. When moving leftward, make a change to the symbol in the current cell. If that change is still present after the left move, you're in the leftmost cell.

Note: some TM formalizations (e.g. Kozen) require a special "left-end marker" in leftmost tape cell.

Turing Machine for $\{a^n b^n c^n \mid n \in \text{Nat}\}$?
High-Level Algorithm Descriptions are OK!

We can give a formal TM description for any algorithm, but it would be extremely tedious to do so.

Instead, we will usually describe algorithms at a higher-level:

• A high-level English description of a Turing Machine or some other step-by-step process. (E.g., description on slide 33-7.)

• Pseudocode, such as in CLRS (CS231)

• A program in (or high-level description of a program in) your favorite programming language (SML, Python, Java, Scheme, etc.), assuming it has infinite memory.

In high-level algorithms, we often want to manipulate various kinds of complex values, such as lists, trees, graphs, grammars, and descriptions of various kinds of automata themselves. We can represent these via appropriate strings that are checked for well-formedness.

High-Level Algorithm for \{a^n b^n c^n \mid n \in \text{Nat}\}?