Lecture 30 - Notes

Goals

- Know the definition, computation, and termination of a Turing machine.
- Be able to design a Turing machine to accept a language.

Announcements

(see webpage)

Loose Ends from Last Time

- We built a PDA! It simulates the application of grammar rules.
- Is it deterministic? No - you may have multiple rules for the same variable.
    - This is fine - the theorem is that CFG ⇔ **N**PDA

A Brief Mention: The Pumping Lemma for Context-Free Languages

There is an equivalent lemma that can be used to show that a language is **not** context-free. The intuition is similar, except instead of "looping" journies through a state machine, we observe that subtrees of the parse tree must be repeated. We won't cover this in much detail, but here's the lemma:

Lemma (The Pumping Lemma for Context-Free Languages): Let L be a context-free language. Then there exists an integer $p \geq 1$, called the pumping length, such that every string s in L with $|s| \geq p$ can be written as $s = uvxyz$, where

- $|vy| \geq 1$ (i.e., v and y are not both empty)
- $|vxy| \leq p$, and

- $u\overset{\downarrow}{v}^{i}x\overset{\downarrow}{y}^{i}z \in L$ for all $i \geq 0$.

This can be used to prove, for example, that $A = \{a^n b^n c^n : n \geq 0\}$ is not context-free.

Proof Sketch: Consider the string $a^p b^p c^p$, and looks quite similar to the $a^n b^n$ regular proof, except that we have to address three cases for the location of vxy. Since u and/or w could be empty, we consider three cases:

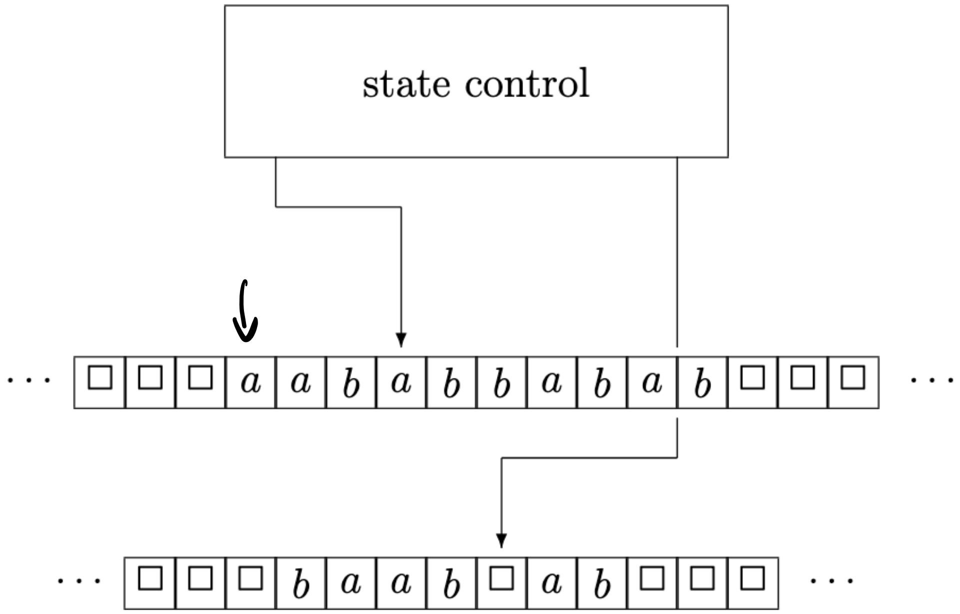
- vxy doesn't have any c; that is, it's entirely in the $a^p b^p$ part.
  - In this case, uv^2xy^2z adds a's or b's but not c's.
- vxy doesn't have any a; that is, it's entirely in the $b^p c^p$ part.
  - A symmetric argument can be made, pumping can add only b's or c's.
- vxy cannot contain a's, b's, and c's,' because its length is less than or equal to p

Thus there's no way to break s into $uvxyz$ that satisfies the CF pumping lemma criteria, so the language cannot be context-free.

The Turing Machine

Having studied finite automata and pushdown automata, we now turn to our next more powerful machine, which turns out to be the most powerful one in the Chomsky Hierarchy. This is the famous Turing Machine, which turns out to be powerful enough to model **any** computer we know how to build (any classical computer, anyway - quantum computing is a different thing).

Informally, a Turing machine is similar to a PDA except that instead of a separate (read-only) input tape and mutable stack, we now have 1 or more tapes that can be read and/or written. Each tape head is now free to move left or right along each tape, and a computation step can involve overwriting the contents of a tape cell. Here's a sketch from the textbook:



Definition

Formally, a deterministic k-tape turing machine is a 7-tuple $M = (\Sigma, \Gamma, Q, \delta, q, q_{accept}, q_{reject})$, where

1. Σ is an alphabet not containing the blank symbol $\square$
2. Γ is a tape alphabet where $\Sigma \subseteq \Gamma$ and $\square \in \Gamma$
3. Q is a finite set of states
4. $q \in Q$ is the start state
5. $q_{accept} \in Q$ is the accept
6. $q_{reject} \in Q$ is the reject state
7. δ is the transition function, which is a function

$$\delta : \underset{\substack{\swarrow \\ \text{State}}}{Q} \times \underset{\substack{\downarrow \\ \text{tape symbol(s)}}}{\Gamma^k} \to \underset{\substack{\downarrow \\ \text{new state}}}{Q} \times \underset{\substack{\downarrow \\ \text{tape write(s)}}}{\Gamma^k} \times \underbrace{\{L, R, N\}^k}_{\substack{\searrow \\ \text{move instruction(s)}}}. \tag{1}$$

Computation

Input: the transition function δ takes as input:

- the current state $r \in Q$
- the symbol currently at each of the k tape heads (let's call them $a_1, a_2, \ldots, a_k) \in \Gamma^k$

Output: the output of the transition function is:

- the new state $r' \in Q$
- the symbol to be written at each of the k tape heads $(a'_1, a'_2, \ldots, a'_k) \in \Gamma^k$
- a move instruction for each tape head $\sigma_1, \sigma_2, \ldots, \sigma_k \in \{L, R, N\}^k$, with L, R, and N representing "move left", "move right", and "don't move", respectively.

Initialization: The input string is stored on the first tape, with the first tape head on the leftmost symbol. The remainder of this and all other tapes is filled with the blank symbol $\square$. The machine is in state q, the start state.

Termination: The machine terminates immediately upon entering either the accept state q_{accept} or the reject state q_{reject}.

Acceptance: The machine accepts a string if the computation terminates in the q_{accept} state.

Rejection: The machine rejects a string (only) if the computation terminates in the q_{reject} state.

Language of the Machine: The language $L(M)$ accepted by Turing machine M is the set of strings accepted by M per the definition above. This means that a string is *not* in $L(M)$ if the string is rejected or if the machine does not terminate.

Example: $a^n b^n c^n$

The language $\{a^n b^n c^n : n \geq 0\}$ turns out not to be context-free; we can use the context-free pumping lemma to show this. Let's design a one-tape Turing machine to accept strings in this language.

Idea: The plan is to "mark" off the first a, then the first b, then the first c; then go back to the beginning and do it again. If we finish and there are no a's, b's, or c's left, then there have to have been the same number of each.

While "marking" symbols, we will need to be quite careful about the ordering while marking them: we don't want to accept acb, or perhaps a more tricky case $aabcbc$. To address this, we'll do the computation in two phases: first we'll check that the string looks like $a^*b^*c^*$; then we'll do the above marking process in a second phase.

Define the turing machine M as follows:

1. $\Sigma = \{a, b, c\}$
2. $\Gamma = \{a, b, c, x, \Box\}$

We will divide the states in Q into two groups, one for each phase. In Phase 1, we're only checking that the string looks like $a^*b^*c^*$:

$$\begin{aligned}
q_a &: \text{ start state, processing } a\text{'s} \\
q_b &: \text{ processing } b\text{'s} \\
q_c &: \text{ processing } c\text{'s} \\
q_L &: \text{ walking back to the start}
\end{aligned}$$

In Phase 2, we're replacing an a, then a b, then a c with x. Here are the states:

$$\begin{array}{lr}
q'_a : & \text{start of Stage 2; find the leftmost } a \\
q'_b : & \text{we have } x\text{'d the leftmost } a\text{; find a } b \\
q'_c : & \text{we have also } x\text{'d the leftmost } b\text{; find a } c \\
q'_L : & \text{all 3 have been } x\text{'d; walk back to the start}
\end{array}$$

We also, of course, have accept and reject states q_{accept} and q_{reject}. So in sum, $Q = \{q_a, q_b, q_c q_L, q'_a, q'_b, q'_c q'_L, q_{accept}, q_{reject}\}$.

Transition function:

We now need to encode the logic for the above states into the transition function. Each rule will have the form $q\sigma \to q'\sigma' M$, where q, q' are the input and output states, σ, σ' are the symbols read and written by the tape head, and M is a move instruction.

We can alternatively write the rules as a table, where each row is a state, each column is a tape symbol, and the cell contains the resulting state, output symbol, and move instruction; for a one-tape machine, I find this nicer:

	a	b	c	$\square$	x	**Comments**
q_a	$q_a aR$	$q_b bR$	$q_c cR$	$q_L \square L$	q_{reject}	Reading a^*
q_b	q_{reject}	$q_b bR$	$q_c cR$	$q_L \square L$	q_{reject}	Reading b^*
q_c	q_{reject}	q_{reject}	$q_c cR$	$q_L \square L$	q_{reject}	Reading c^*
q_L	$q_L aL$	$q_L bL$	$q_L cL$	$q'_a \square R$	q_{reject}	Move to beginning and start phase 2
q'_a	$q'_b xR$	q_{reject}	q_{reject}	q_{accept}	$q'_a xR$	Find leftmost a and x it, or accept if everything is x's.
q'_b	$q'_b aR$	$q'_c xR$	q_{reject}	q_{reject}	$q'_b xR$	Find leftmost b and x it. a or x is ok, but c and $\square$ are not.
q'_c	q_{reject}	$q'_c bR$	$q'_L xL$	q_{reject}	$q'_c xR$	Find leftmost c and x it. b or x is ok, but a and $\square$ are not.
q'_L	$q'_L aL$	$q'_L bL$	$q'_L cL$	$q'_a \square R$	$q'_L xL$	Move to beginning and continue phase 2

Informal Example: Palindromes in One Tape

1. Start at the left end.
2. Delete a character and rember what it was via state
3. Walk to the right end.
4. Make sure it matches the start character and delete it
5. Walk to the left end and start back at 1.