CSE396 Spr26 Lecture Tue. Mar. 31: Turing Machines and Pushdown Automata

The two extra components that enlarge the 5-tuple of a finite automation to the 7-tuple of a Turing machine are the work alphabet Γ and the blank $\sqcup$. Instruction "tuples" also have the same read-components, a current state p and current char c, but they have two more 'action" fields: a character d that c can change to ($d = c$ means no change) and a "direction" D that can be a left move (L) or keeping stationary (S) in addition to moving to the next character on the right (R). These concepts actually came earlier than finite automata, in Alan Turing's seminal 1936 paper titled, "On Computable Numbers, With an Application To the *Entscheidungsproblem*."

Definition: A **Turing machine** is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, \sqcup, s, F)$ where Q, s, F and Σ are as with a DFA, the *work alphabet* Γ includes Σ and the *blank* _, and

$$\delta \subseteq (Q \times \Gamma) \times \left(\Gamma \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}\} \times Q\right).$$

It is **deterministic** (a **DTM**) if no two instructions share the same first two components. Otherwise it is (properly) a **nondeterministic Turing machine** (**NTM**). If we just say "Turing Machine" (**TM**) we mean a DTM unless otherwise stated.

A DTM is in the Sipser textbook's "normal form" if F consists of one state q_{acc} and there is only one other state q_{rej} in which it can halt, so that δ is a *function* from $(Q \setminus \{q_{acc}, q_{rej}\}) \times \Gamma$ to $(\Gamma \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}\} \times Q)$. The notation then becomes $M = (Q, \Sigma, \Gamma, \delta, \_, s, q_{acc}, q_{rej})$.

We show the design a Turing machine M such that $L(M) = \left\{a^m b^n : m = n\right\}$. The main ulterior point is that we lead with the *logic* of the design, "writing comments before writing code," and then find that the machine is able to express the logic quite efficiently. The "5-tuple notation" as shown in the ***Turing Kit*** software, instead of the text's "4-tuple notation", helps with this.

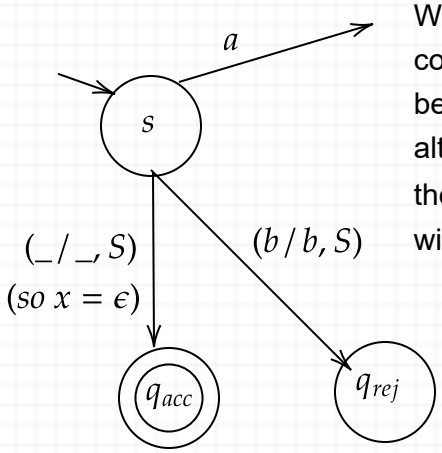
The language allows $m = n = 0$, so it contains ϵ. Our machine doesn't see 'ϵ' as a character, though. What it sees instead is a blank tape. "Blank tape" means that every **tape cell** is filled with the blank character $\sqcup$. So the input being ϵ is the same as M in its start state s reading $\sqcup$. We can handle this by the single instruction

$$(s, \sqcup \ / \ \sqcup, S, q_{acc}).$$

On the other hand, if the given **input string** x begins with a b, then x cannot be in the language. We can handle this case simply by the instruction

$$(s, b \ / \ b, S, q_{rej}).$$

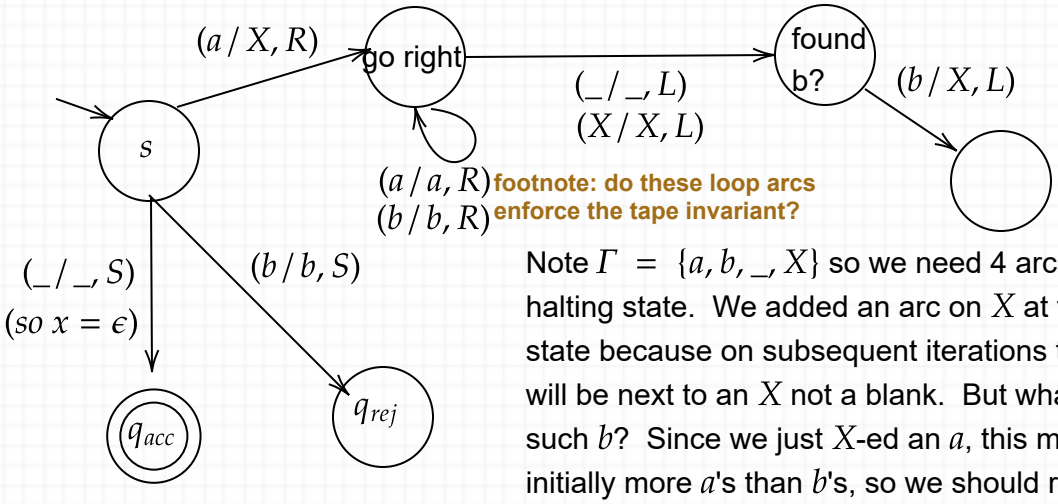
If x begins with an a, then we have to begin the "real work." But actually, handling the initial acceptance and rejection logic already gives us a nice chunk of the design of the machine:



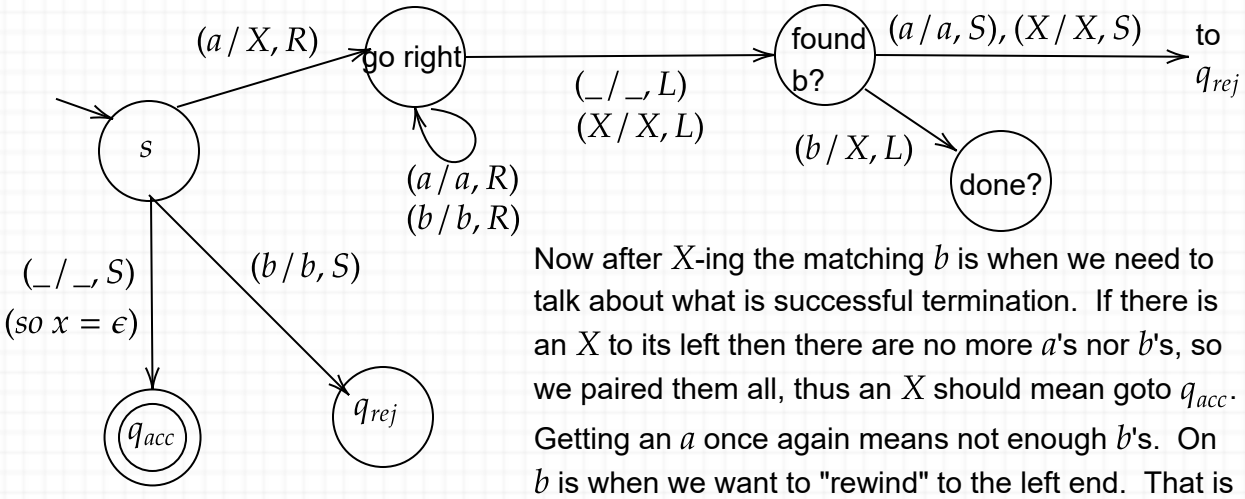
We've already been able to handle immediate accept and reject conditions in the start state. Now we decide strategy when x begins with a. The idea is to X-out a's and b's one-by-one in alternation. If we X-out always the leftmost a and the rightmost b then the string between (which after the first iteration is $a^{m-1}b^{n-1}$) will belong to L if and only if x does. So we can recurse and keep:

Tape Invariant: $X^*\ a^*\ b^*\ X^*$ and after X-ing a b the numbers of Xes on left and right are the same, so the string between them belongs to L if and only if the original x does.

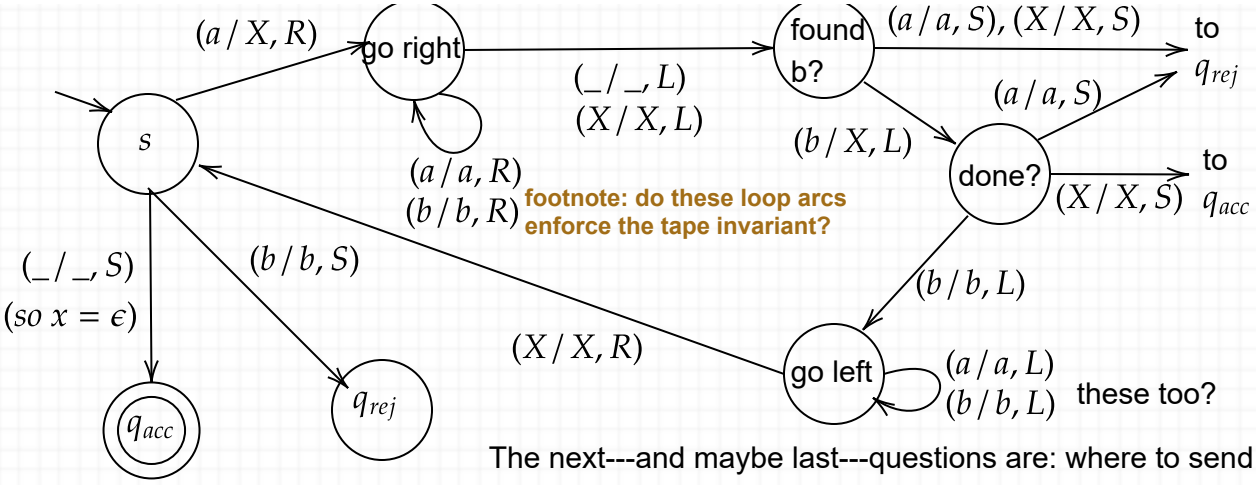
To perform the X-ing of one a then the rightmost b, add these states and instructions:



Note $\Gamma = \{a, b, \_, X\}$ so we need 4 arcs at each non-halting state. We added an arc on X at the "go right" state because on subsequent iterations the rightmost b will be next to an X not a blank. But what if there is no such b? Since we just X-ed an a, this means there were initially more a's than b's, so we should reject.



Now after X-ing the matching b is when we need to talk about what is successful termination. If there is an X to its left then there are no more a's nor b's, so we paired them all, thus an X should mean goto q_{acc}. Getting an a once again means not enough b's. On b is when we want to "rewind" to the left end. That is when we need X to stop a leftward loop. So we cannot loop at the "done?" state itself but need another state:



The next---and maybe last---questions are: where to send the arc on X, and what actions to do? Most in particular:

Can we complete the loop and the machine by making it be $(X / X, R)$ going back to start?

One thing to note is that if the char seen after executing $(X / X, R)$ is a b, then by the tape invariant it means there are no more a's but still at least one b since we went from "done" to "go left", so this is the case $m < n$. Well, in that case we should reject, and the arc on b going to q_{rej} is already there from the initial design. So: *this is OK and M is complete.*

Note that the input x can belong to $a^* b^*$ without belonging to L. Those strings abide by the tape invariant initially, and we can already see that M works correctly on those strings. But what if x is something like $aababb$? Will our M accept when it shouldn't? **That's what the footnote is about.**

Multitape Turing Machines

Assuming M is correct---or quickly fixable if not---we can ask, how long does it take to accept a good $x = a^n b^n$ in terms of n? The answer is, it takes $\Theta(n^2)$ steps, owing to lots of backing-and-forthing. Can we make it run faster? There is a way to make it run much faster on one tape, in $O(n \log n)$ time, but we can get an optimal $O(n)$ running time by using a second tape,

Definition: A *k-tape Turing machine* is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, \_, s, F)$ where Q, s, F and Σ are as with a DFA, the *work alphabet* Γ includes Σ and the *blank* _, and

$$\delta \subseteq \left(Q \times \Gamma^k\right) \times \left(\Gamma^k \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}\}^k \times Q\right).$$

It is *deterministic* (a DTM) if no two instructions share the same first two components. A DTM is "in normal form" if F consists of one state q_{acc} and there is only one other state q_{rej} in which it can halt, so that δ is a function from $(Q \setminus \{q_{acc}, q_{rej}\}) \times \Gamma$ to $(\Gamma \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}\} \times Q)$. The notation then becomes $M = (Q, \Sigma, \Gamma, \delta, \_, s, q_{acc}, q_{rej})$. All **instructions** (still also called **5-tuples** or just **tuples**) have the form

$$(p, [c_1, c_2, \ldots, c_k] \ / \ [d_1, \ldots, d_k], [D_1, \ldots, D_k], q)$$

with $p, q \in Q$, $c_j, d_j \in \Gamma$, and $D_j \in \{\mathsf{L}, \mathsf{R}, \mathsf{S}\}$ ($j = 1$ to k).

The machine is *deterministic* (a DTM) if no two instructions share the same first two components. A DTM is "in normal form" if F consists of one state q_{acc} and there is only one other state q_{rej} in which it can halt, so that

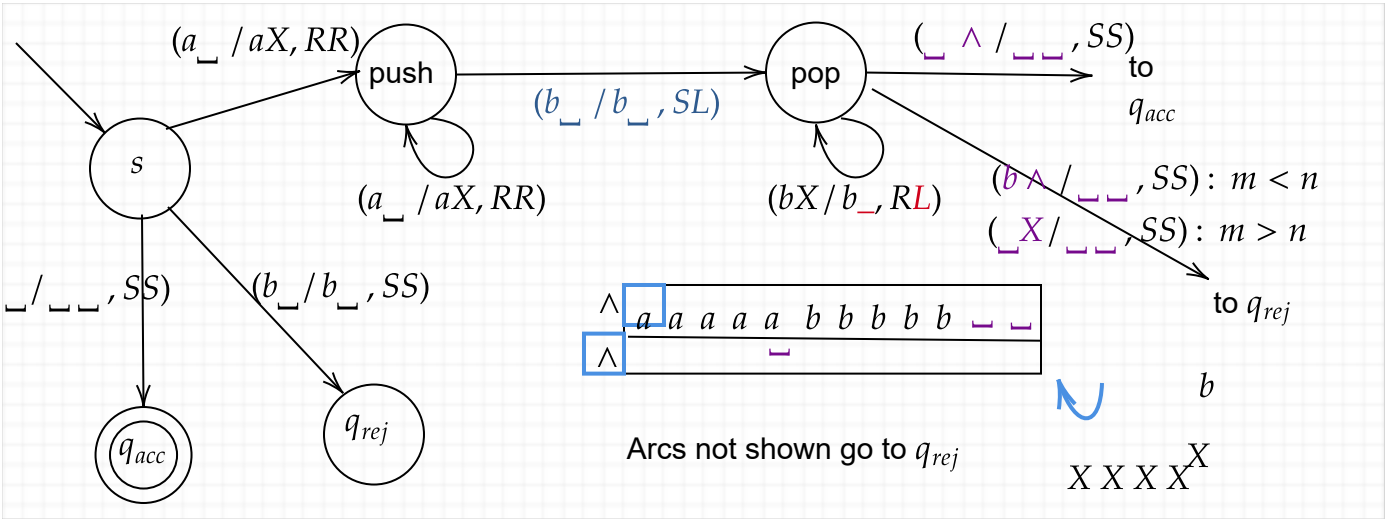
$$\delta : (Q \setminus \{q_{acc}, q_{rej}\}) \times \Gamma^k \to \left(\Gamma^k \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}\}^k \times Q\right)$$

This is read: "δ is a function from $(Q \setminus \{q_{acc}, q_{rej}\}) \times \Gamma^k$ to $\left(\Gamma^k \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}\}^k \times Q\right)$." The notation then becomes $M = (Q, \Sigma, \Gamma, \delta, \_, s, q_{acc}, q_{rej})$. An option I will show with $k = 2$ or 3 is to write them vertically like so:

$$\left(p, \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} / \begin{bmatrix} d_1 & D_1 \\ d_2 & D_2 \end{bmatrix}, q\right)$$

The Turing Kit allows both options, calling the latter "stacked"---IMHO it is easier to visualize.

Example 2-tape TM "on paper" for $L_1 = \{a^m b^n : n = m\}$:



Note the straightforwardness of the design as well as the running time efficiency. Also note the usefulness of having the second tape be two-way infinite with a blank to the left of the "column" initially holding the first a in x (if any). An alternative convention is to make both tapes one-way infinite but with a special char $\wedge$ in cell 0 at the left end on tape 1---so that the *initial configuration* I_0 has $\wedge x_1 \cdots x_n$ on tape 1 and just $\wedge$ on tape 2 "underneath" the $\wedge$ on tape 1. We can still start with the tape heads scanning the cells in "column 1" even if both are blank (so $x = \epsilon$). Then the final accepting instruction in the "pop" state becomes (␣ ∧ / ␣ ∧ , SS).

Pushdown Automata

This two-tape DTM has the properties that:

- the input tape head never moves L and never changes a character;
- whenever the second tape moves L, it writes a blank in the cell it just left.

The second condition forces the second tape to behave like a ***stack*** (except for some "flex" in how top-of-stack is treated). This enables us to make the following handy definition:

Definition: A **pushdown automaton** (PDA) is (equivalent to) a 2-tape Turing machine M in which every instruction has:

- $d_1 = c_1$, so that the input tape is **read-only**;
- $D_1 \neq \mathsf{L}$, so that the input tape is **one-way**; and
- $D_2 = \mathsf{L}$ only if $d_2 = \sqcup$, so that the tape-2 head is always on or right of the rightmost char.

The PDA is deterministic (a **DPDA**) or nondeterministic (an **NPDA**) according as M is deterministic or nondeterministic.

The third condition makes the second tape behave like a **stack**. Its head can only read the rightmost non-blank char, which is the "top" of a stack that "grows" to the right by "pushing" new chars. If the head wants to read the char to its left, the third condition makes it have to blank-out the top char, which is a "pop" move. The one cosmetic difference of using the TM notation is the need for an extra "stutter-step" to switch between "push mode" (which is when scanning the blank to the right of the topmost char) and "pop mode" (when scanning the topmost char).

Having the stay option S on the input tape is a handy coding convenience and replaces the use (IMHO, overuse) of ϵ's in the text's PDA notation in section 2.2. Another convenience is to introduce $\wedge$ as a bottom-of-stack marker on tape 2 in the first step

$$\left(s, \begin{bmatrix} \sqcup \\ \sqcup \end{bmatrix} / \begin{bmatrix} \sqcup & \mathsf{R} \\ \wedge & \mathsf{R} \end{bmatrix}, q\right)$$

(and optionally, for general TMs, to introduce $\wedge$ on tape 1 as well). That way you can't confuse blank meaning "the stack is empty" with the blank to the right of the stack saying where you can push new chars.

Theorem 1: A language A is context-free if and only if there is an NPDA N such that $L(N) = A$.

Proof: Skipped in 2026---a sketch is browned out below.

Definition: A language A is a **deterministic context-free language** (**DCFL**) if there is a DPDA M such that $L(M) = A$. The class of all DCFLs (over any given alphabet Σ) is denoted by **DCFL**.

Theorem 2: The complement of a DCFL is always a DCFL. That is, the class **DCFL** is closed under complementation.

Proof: Given any DPDA $M = (Q, \Sigma, \Gamma, \delta, \sqcup, s, q_{acc}, q_{rej})$, the idea is simply to do the same trick we did with DFAs: to interchange the accepting and rejecting states to make $M' = (Q, \Sigma, \Gamma, \delta, \sqcup, s, q_{rej}, q_{acc})$. We can get away with this, however, only if we first guarantee that M cannot "loop forever" with stay moves on Tape 1 and repeatedly thrashing the stack on tape 2. The details of doing so are FYI. (They are gritty even with the text's PDA notation. The text goes on in section 2.4 to develop kinds of grammars that produce DCFLs. But none of them captures the whole class **DCFL** as defined by machines---and no one has found a truly natural and fully general notion of "deterministic CFG" that does so.) ☒

The basic point made here becomes accentuated for general Turing machines, where we *cannot* always modify them so that they always halt. The demonstration of this is one of two main themes for the rest of the course (the other is **mapping reductions**). But before we get there, we should address details of **computations** and define them formally. [That will pick up the next lecture.] To summarize:

- Every regular language is a DCFL
- $\{a^n\ b^n\}$ is a DCFL that is not regular.
- $\{a^n\ b^n\ c^n\}$ is accepted by a DTM but is not a CFL, so it cannot be accepted by a PDA.
- The complement of $\{a^n\ b^n\ c^n\}$ is a CFL, but it is not a DCFL (by Theorem 2).

[Lecture finished by showing examples using the Turing Kit, also of a non-PDA for the $\{ww\}$ language.]

PDAs and CFGs [proof skipped in 2026]

Theorem: A language A is a CFL if and only if there is an NPDA N such that $L(N) = A$.

Proof Idea: Take a CFG $G = (V, \Sigma, \mathcal{R}, S)$ in Chomsky NF such that $L(G) = A \setminus \{\epsilon\}$. (As with the CFL Pumping Lemma, ChNF is not necessary and the text does without it, but IMHO it improves the visual understanding. If $\epsilon \in A$ we can handle that by a later patch to the code of N.) The NPDA N has just one "hub state" q (together with a helper state p for pushing) besides s and q_{acc} (and q_{rej} just to comply with the text's TM syntax). It begins with the instructions

$$\left(s, \begin{bmatrix} \sqcup \\ \sqcup \end{bmatrix} / \begin{bmatrix} \sqcup & \mathrm{R} \\ \wedge & \mathrm{R} \end{bmatrix}, p\right), \left(p, \begin{bmatrix} c \\ \sqcup \end{bmatrix} / \begin{bmatrix} c & \mathrm{S} \\ S & \mathrm{S} \end{bmatrix}, q\right)$$

which initialize the stack to hold just $\wedge$ and the grammar's start symbol S. Note that the second step

leaves whatever char $c \in \Sigma$ is on the input tape alone (and if $c = \text{␣}$ so that $x = \epsilon$ and we want to accept ϵ, this is where we can). A grammar rule $A \to BC$ becomes the instructions:

$$\left(q, \begin{bmatrix} c \\ A \end{bmatrix} / \begin{bmatrix} c & \mathsf{S} \\ C & \mathsf{R} \end{bmatrix}, p\right), \left(p, \begin{bmatrix} c \\ \text{␣} \end{bmatrix} / \begin{bmatrix} c & \mathsf{S} \\ B & \mathsf{S} \end{bmatrix}, q\right)$$

Notice that C is pushed first, because B will be expanded next in a *leftmost* derivation and so needs to be top-of-stack. Also note that if there are multiple rules for A, they all have the same q and A, and they automatically have c for all $c \in \Sigma$ because the input tape is left alone in these moves---so N is nondeterministic. And a terminal rule $A \to c$ simply becomes the "pop" move

$$\left(q, \begin{bmatrix} c \\ A \end{bmatrix} / \begin{bmatrix} c & \mathsf{R} \\ \text{␣} & \mathsf{L} \end{bmatrix}, q\right),$$

which also moves on to the next char on the input tape. If there is no next char, then we want to accept if and only if we just popped the last variable on the stack, which finishes off a leftmost derivation of the input x in G. So the computation should accept iff it now sees the $\wedge$ on tape 2, and this is handled by the single instruction

$$\left(q, \begin{bmatrix} \text{␣} \\ \wedge \end{bmatrix} / \begin{bmatrix} \text{␣} & \mathsf{S} \\ \text{␣} & \mathsf{S} \end{bmatrix}, q_{acc}\right).$$

The **invariant** that explains why $L(N) = L(G)$ is that at any point X in a leftmost derivation $S \implies \ldots \implies X \implies \ldots$ of G in ChNF, the sentential form X belongs to $\Sigma^* V^*$, that is, has the form $X = uW$ where u has only terminals and W has only variables. At the same point, N has read u so far on its input tape and its stack has W in reverse---so that the first variable in W is the top-of-stack element. When the W part disappears, this means u was the whole input string x and N sees the $\wedge$, so we simultaneously have $S \implies^* x$ and N accepts x.

The proof in the other direction---from machine to equivalent grammar---is more complicated and is: *FYI, skim/skip* in the text. ☒

The above also contains the essence of proving the *equivalence* of two criteria for PDAs that one can find discussed in other sources: *acceptance by empty stack* and *acceptance by final state*. We can always make a PDA---nondeterministic or deterministic---do both simultaneously. One final definition and fact to wrap up the material regarded as covered in chapter 2 (it is in the first few pages of section 2.4 but just take it from here---that section is otherwise *skipped*):