CSE396 Thu. Week 9: Non-CFLs and Turing Machines

We arrange the statement of the CFL Pumping Lemma to emphasize the sequence of dependent quantifiers:

Theorem: For any language L, **if** L is context-free **then**:
there exists a number $N > 0$ so that
for any $x \in L$ with $|x| \geq N$,
there exists a breakdown $x =: uvwxy$ such that $|vwx| \leq N$, $vx \neq \epsilon$, and
for all $i \geq 0$, the "pumped string" $x^{(i)} = uv^iwx^iy$ **also belongs to** L.

Call the part beginning "there exists..." X. So the theorem says, if L is a CFL then X. The contrapositive of this is, "If Not-X then L is not a CFL." Now to find out what Not-X is, we flip each quantifier and negate the body at the end. What we get is:

Theorem$'$: For any language L, **if**
for all $N > 0$
there exists a string $x \in L$ with $|x| \geq N$ such that
for all breakdowns $x =: uvwxy$ such that $|vwx| \leq N$ and $vx \neq \epsilon$,
there exists $i \geq 0$ such that $x^{(i)} = uv^iwx^iy$ **does not belong to** L,
then L is not a CFL.

Now we can convert this into a "proof script" to use on a language L we think is a non-CFL:

Let any $N > 0$ **be given**.
Take $x =$ _____ (depending only on N, often $|x| = 3N$ or $4N$ or so). **See** $x \in L$.
Let any breakdown $x =: uvwxy$ such that $|vwx| \leq N$ and $vx \neq \epsilon$ **be given**.
[Now what often happens is that we need to break into multiple cases of where inside x the vwx portion could possibly be. The one thing we know is that because $|vwx| \leq N$, while $|x|$ is up around $3N$ or more, the "wingspan" of vwx has to stay within a limited portion of x. But it could be anywhere, and we need to analyze separately an **exhaustive** list of cases of where it could be.]
Case I. vwx is _____ : **Take** $i =$ ___. Then $x^{(i)} \notin L$ because _____.
Case II. vwx is _____: **Take** $i =$ ___. Then $x^{(i)} \notin L$ because _____.
...

Do as many cases as you need. You may use different values of i between cases. (Usually the issue is just whether to "pump down" with $i = 0$ or "pump up" with $i = 2$.) When all possible cases are accounted for, you can conclude: **Then** L is not a CFL, by the CFL Pumping Lemma. ☒

Example 1: Prove that $L_0 = \{a^n b^n c^n : n \geq 0\}$ is not a CFL.

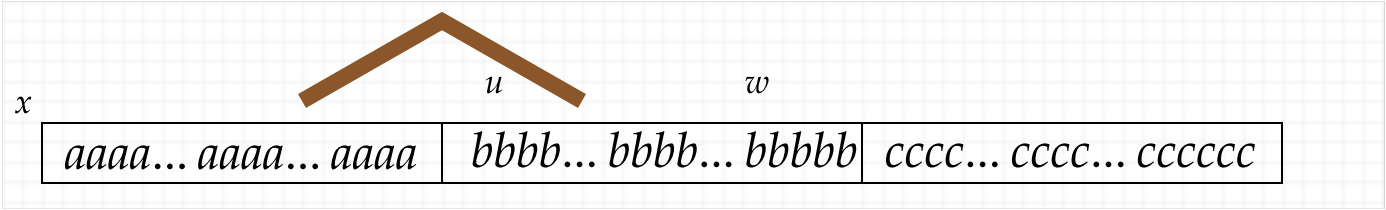
Proof: Let any $N > 0$ be given. Take $x = a^N b^N c^N$. Then $x \in L_0$. Let any breakdown $x =: uvwyz$ such that $|uvw| \leq N$ and $uw \neq \epsilon$ be given. The key point is that by $|uvw| \leq N$, the uvw part cannot touch both the a^N region and the c^N region.

- Case I: It does not touch the a^N region. By $uw \neq \epsilon$, u and w must collectively have at least one b or c. Thus the "pumped down" string $x^{(0)} = yvz$ begins with a^N but either has fewer than N b's or fewer than N c's (or both). So $x^{(0)}$ is not in L_0.
- Case II: It does not touch the c^N region. By $uw \neq \epsilon$, u and w must collectively have at least one a or b. Thus the "pumped down" string $x^{(0)} = yvz$ ends with c^N but either has fewer than N a's or fewer than N b's (or both). So $x^{(0)}$ is not in L_0.

Since these cases are exhaustive and we got $x^{(i)} \notin L_0$ in both, L_0 is not a CFL, by the CFL Pumping Lemma. ☒

Note that the cases do not have to be mutually exclusive---in fact, if uvw is wholly within the b^N part, either one could apply. But redundancy doesn't hurt here---that the cases cover all possibilities is what's needed. Another aspect is that u or w individually might include two different letters. Then the pumped-up string $yuuvwwz$ doesn't even preserve the $a^*b^*c^*$ "**format**" that x had. That can be ignored if we focus on what information in the cases is enough to conclude $x^{(i)} \notin L_0$ without overkill.

Here is a nifty visualization one can use for formulating the cases. The uvw part is like a **drawing compass** whose two points are u and w:



There is a limit to the width of the compass arms (else it would become unstable) and that represents N. Actually, the pumping up of u and w at the compass points is an act of creation of the grammar, and the compass---as used by architects---came to symbolize creation on the whole. The Masons put a **G** underneath the compass to stand for God and Geometry, but we can think of it as just meaning Grammar.

There is also a "more-personalized" way to do the proof in the form of an **adversary argument**. This treats the "let any ... be given" steps as being controlled by "**Adv**" who is trying to pass off the language as context-free. Your have to be prepared for any legal combination that Adv might try, especially in the "breakdown" step, and your goal is to refute Adv's assertion that his breakdown will stay in the language when pumped. This basically follows the "proof script" but words it more like a dialogue. We exemplify this on a trickier version of the first language.

Example 2: Prove that $L = \{a^i b^j c^k : i > j \wedge j > k\}$ is not a CFL.

Proof: We begin with Adv claiming, "I have a CFG G such that $L(G) = L$." This is the assertion for-sake-of-contradiction that you want to refute. You begin by asking,

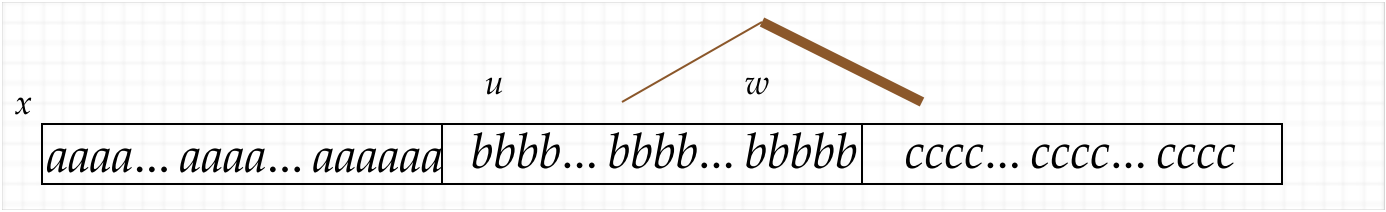
"What is the N of your grammar G?"

In an actual play of this "protocol", the Adversary would name a particular number N, such as $N = 65,536$. Yes, if the grammar has k variables when converted into Chomsky normal form then $N = 2^k$ but we need not care about this detail. We allow Adv to speak a general number N, but as with "m" and "n" in the Myhill Nerode proof script, the point is that it is fixed once Adv. says it. Now we itemize the steps like in the proof script.

1. Adv: names a particular number "N".
2. You take $x = a^{N+2}b^{N+1}c^N$ and say: "This x is in L. Give me a breakdown of x into $yuvwz$ such that $|uvw| \leq N$ and at least one of u and w is not the empty string." [Note: You could also take $x = a^{N+12}b^{N+11}c^{N+10}$. The point is to make the string x satisfy the condition $i > j > k$ just-barely.]

3. Adv. gives a breakdown x =: $yuvwz$. There are a zillion possible breakdowns Adv can give, subject to $|uvw| \leq N$. (but you may have to do this step multiple times for each case).
4. You take a value of $i \neq 1$ ($i = 0$ for "pump down" or $i = 2$ or more for "pump up") and your goal in each case is to get that $x^{(i)} = yu^i vw^i z$ is not in L. Note $x^{(0)} = yu^0 vw^0 z = yvz$.

You have to be prepared for every kind of case the adversary could come up with.



Claim: The following two cases are *exhaustive*: Either

- u or w includes at least one a and/or at least one b but does not include any c's, or
- u or w includes at least one b and/or at least one c but does not include any a's.

These cases are exhaustive because the "compass" cannot be wider than the b^{N+1} part.

In the first case, we "pump dpwn". If uw *includes at least one* b then pumping down leaves at most N b's while the number of c's remains at N, so the $j > k$ condition is violated. Else, uw includes only a's, but then pumping down violates the $i > j$ condition.

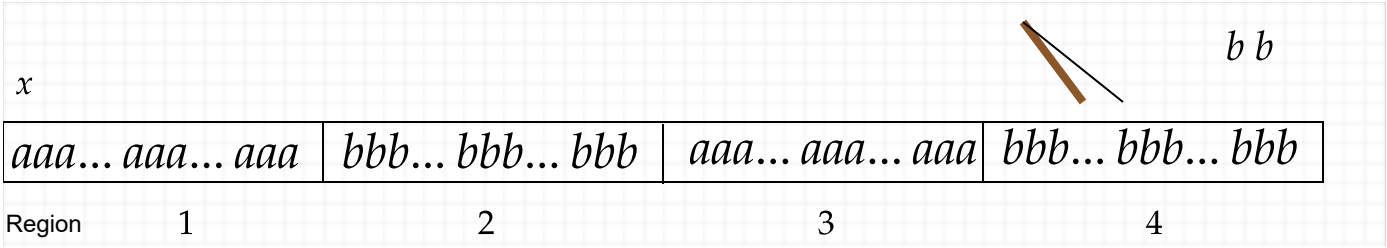
In the second case, we "pump up": Either this adds at least one b, thus violating $i > j$ since $i = N + 2$ is left alone in the a's while the number j of b's grows to at least $N + 2$, or this adds at least one c without changing the number of $b's$, thus violating $j > k$.

Since we handle all cases of what Adv could do, we refute Adv's claim that L is a CFL. This amounts to proving that L is not a CFL. ☒

Example 3: Now consider the following three languages:

1. $L_1 = \{a^m b^m \quad a^n b^n : m, n \geq 1\} \quad = \{a^m b^m : m \geq 1\} \bullet \{a^n b^n : n \geq 1\}$,
2. $L_2 = \{a^m b^n a^m b^n : m, n \geq 1\}$,
3. $L_3 = \{a^m b^n a^n b^m : m, n \geq 1\}$

Which one(s) are CFLs and which not? Consider strings of the form $x = a^N b^N a^N b^N$:



In L_1, if the "compass" straddles the first two regions, so that $u =$ some number of a's and $w =$ the same number of b's, then pumping changes the value of m in both regions but that's still fine for L_1. It is also OK for the compass to straddle regions 3 and 4. In L_3, the compass has to straddle regions 2 and 4, but that is AOK for the inner $b^n a^n$ part. In fact, they are CFLs:

$$S_1 \to CC, \qquad C \to aCb \mid ab\,.$$
$$S_3 \to aS_3b \mid aTb, \quad T \to bTa \mid ba\,.$$

L_2 is not a CFL. The reason intuitively is that the "compass" has to "touch" at least one of the regions, but it cannot simultaneously touch regions 1 and 3, nor 2 and 4. In the case where it touches region 2, say, pumping down crerates an imbalance with region 4. And so on.

Three closely related languages with alphabet $\Sigma = \{(,), [,]\}$ are:

1. $B_1 = \{(^m)^m[^n]^n : m, n \geq 1\}$
2. $B_2 = \{(^m[^n)^m]^n : m, n \geq 1\}$
3. $B_3 = \{(^m[^n]^n)^m : m, n \geq 1\}$.

The fact that B_1 and B_3 are CFLs goes with saying it is OK to nest different kinds of brackets side-by-side or nested within each other. But B_2 not being a CFL says it is wrong to "cross-nest" them.

Example 4: Prove that DOUBLEWORD = $\{ww : w \in \{a,b\}^*\}$ is not a CFL.

Note that L_2 is a subset of DOUBLEWORD. More important, the pumped-down strings never belong to DOUBLEWORD either. So you can "re-use" the same proof to conclude that DOUBLEWORD is not a CFL. (One intuitive ramification is that the grammar of a programming language cannot by-itself enforce that the parameters of a function match up between where the function is defined and whewre it is called. This is enforced in a type-checking stage of compilation that comes after the parsing stage.)

There is also a more-general way to prove that DOUBLEWORD is not a CFL. It uses the following theorem, whose proof we will defer until after we represent **Pushdown Automata** as a special-case of two-tape **Turing Machines**:

Theorem: Not only is every regular language a CFL, but also the intersection of a CFL and a regular language is always a CFL. ☒

Now, if DOUBLEWORD were a CFL, then its intersection with the regular language $a^+b^+a^+b^+$ would be a CFL. But that intersection is exactly the language $L_2 = \{a^m b^n a^m b^n : m,n \geq 1\}$ above, which we proved is not a CFL. So DOUBLEWORD is not a CFL.

What's interesting about this is that problem (2) on homework 5 basically showed that the *complement* E of DOUBLEWORD **is** a CFL. (Technically, E adds to the grammar the rules $S \to O, \quad O \to a \mid b \mid aaO \mid abO \mid baO \mid bbO$ to throw in all odd-length strings.) Looking at it from the point of view of E, E is a CFL whose complement is **not** a CFL. The upshot is:

Theorem: The class CFL of context-free languages is **not** closed under complements. Nor is it closed under intersection, as exemplified by the non-CFL L_0 in Example 1 equaling the intersection of $\{a^n b^n\} \cdot c^*$ with $a^* \cdot \{b^n c^n\}$, both of which are CFLs. (But the class of CFLs **is** closed under union, as follows from the idea of doing $S \to S_1 \mid S_2$ etc.)

The rule of thumb is that a CFG can handle **one** kind of counting dependency, but cannot handle **two** such dependencies that overlap or cross each other in some way. It will follow that Pushdown Automata cannot handle such dependencies either. What kind of simple machine *can* handle them? That turns the page to Chapter 3 and the final third of this course. [Note that the individual chapters of the Sipser text get much shorter from here on---even though we will do Chapters 3--5 and some of 7, the raw page count covered is about the same as in Chapter 1 and in Chapter 2.]

Turing Machines

We saw that DFAs M, nor even NFAs nor GNFAs, cannot recognize simple languages like $\{a^m\, b^n : m = n\}$. How can we augment the DFA *model* to give it the needed capability?

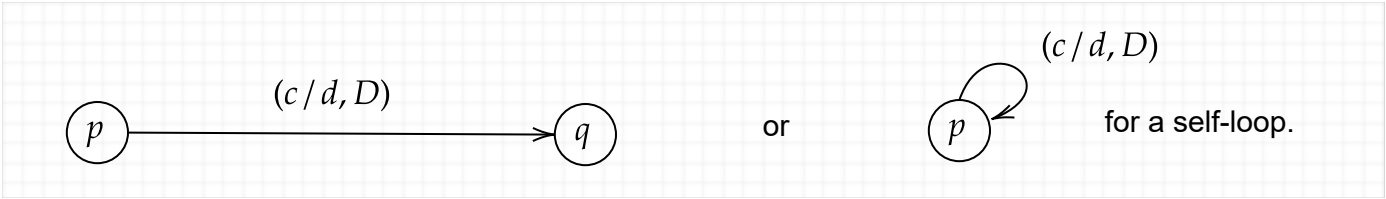
1. Allow M to change a character it reads, storing it on its tape.
2. Allow M to move its scanner left L as well as right R (or keep it stationary S).

Capability 1 by itself changes nothing: the DFA would still have to move R past the changed character. Capability 2 by itself also does not allow recognizing any nonregular languages. The proof, that every "two-way DFA" can be simulated by a simple 1-way DFA, is beyond our scope and involves another "exponential explosion" but we will cite it later to say that the class of regular languages equals "constant space" on a Turing machine.

But if we give both capabilities together, then we can do it---and lots more besides. The capabilities add two components to instructions in δ, making them 5-tuples:

$$(p, c\,/\,d, D, q) \quad \text{where } p \text{ and } q \text{ are states, } c \text{ and } d \text{ are chars, and } D \in \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}$$

The meaning is that if M is in state p and scans character c, then it can change it to d, move its scanning head one position left, right, or keep it stationary, and finally transit to state q. The case (p, c, c, R, q) is the same as an ordinary FA instruction (p, c, q) where moving right is automatic. I tend to like to write a slash for the second comma to emphasize that p, c are read and d, D, q are actions taken; it also visually suggests c being changed to d. Graphically the instruction looks like:



We also regard the blank as an explicit character. I will represent it as _ in MathCha but in full LaTeX you can get "\text{\textvisiblespace}" which turns up the corners to look like more than just an underscore. My other notes call the blank B. The blank belongs not to the *input alphabet* Σ but to the work alphabet Γ (capital Gamma) which always includes Σ too. We allow going past the right end of the input string $x \in \Sigma^*$ where successive *tape cells* each initially hold the blank. We *can* also allow moving leftward of the first char of x where there are likewise blanks on a "two-way infinite tape", *or* we can stipulate that x is initially left-justified on a "one-way infinite tape" and consider any left move from the first cell to be a "crash." The *Turing Kit* package shows a two-way infinite tape and this is the default. A compromise is to use a one-way infinite tape but place a special left-endmarker char $\wedge$ in cell 0 with x occupying cells $1, \ldots, n$ where $n = |x|$. If $x = \epsilon$ then the whole tape is initially blank except in the last case it has just $\wedge$ in cell 0. Then $\wedge$, as well as _, belongs to Γ but not to Σ. We will be free to put any other characters we want into Γ, but the blank (and $\wedge$ if used) are required. With all that said, the definition is crisp:

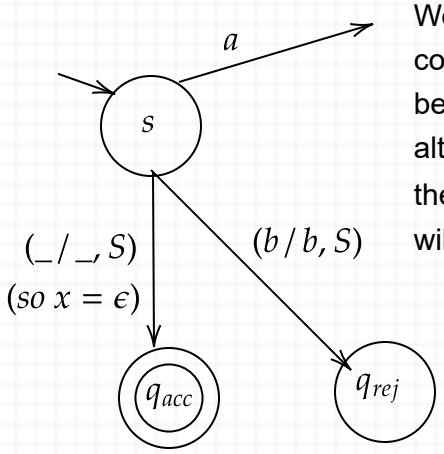
[Lecture got this far. I will pick up here on Tuesday.]

Definition: A *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

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

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 $M = (Q, \Sigma, \Gamma, \delta, \_, s, q_{acc}, q_{rej})$.

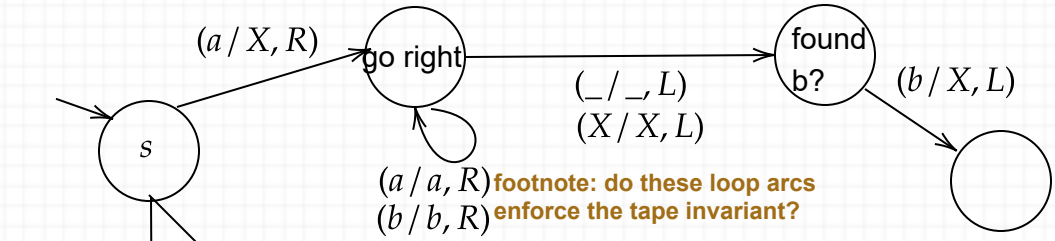
Now we can begin to design a Turing machine M such that $L(M) = \{a^m b^n : m = n\}$.



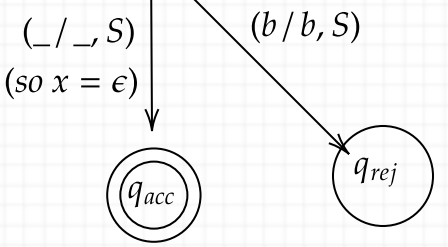
We've already been able to handle immediate accept and reject conditions in the start state. Now we decide strategy when x begins with a. The idea is to X-out a's and b's one-by-one in alternation. If we X-out always the leftmost a and the rightmost b then the string between (which after the first iteration 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.

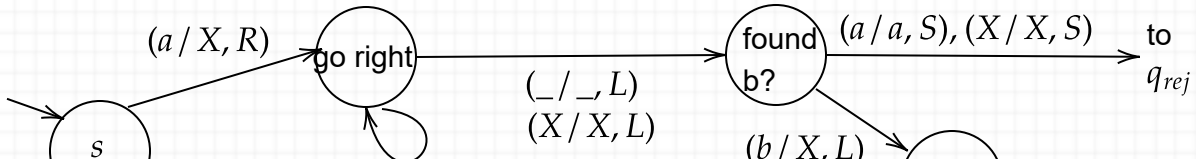
To perform the X-ing of one a then the rightmost b, add these states and instructions:

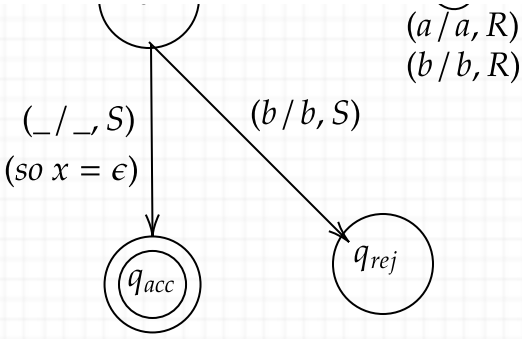


footnote: do these loop arcs enforce the tape invariant?

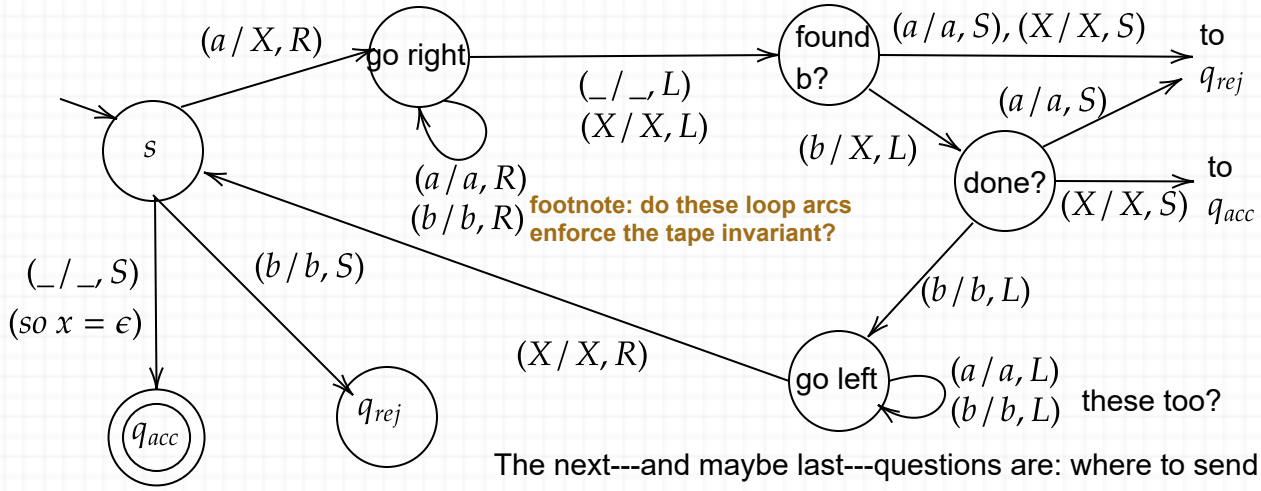


Note $\Gamma = \{a, b, \_, X\}$ so we need 4 arcs at each non-halting state. We added an arc on X at the "go right" state because on subsequent iterations the rightmost b will be next to an X not a blank. But what if there is no such b? Since we just X-ed an a, this means there were initially more a's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