It’s possible to prove that some languages are not regular.

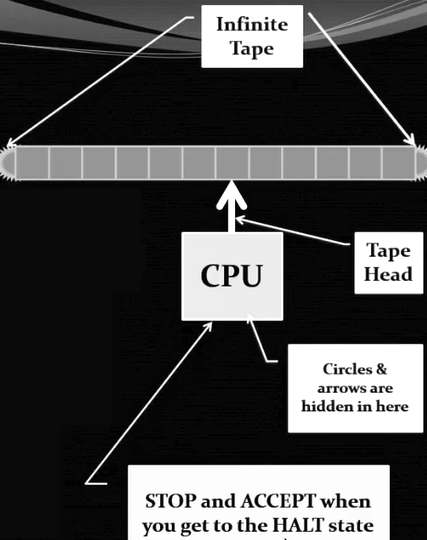
It’s possible to prove that some languages are not context-free.

# **Section 12.1 Turing Machine**

Machines for regular languages: DFA, NFA

Machine for context-free languages: PDA

Machine for everything else: Turing Machine

The Turing Machine is an equivalent to the PC on the desk (but it’s not very efficient). There are programs that cannot be written on a Turing Machine, which proves that you can’t write that program anywhere.

Each TM “instruction” contains:

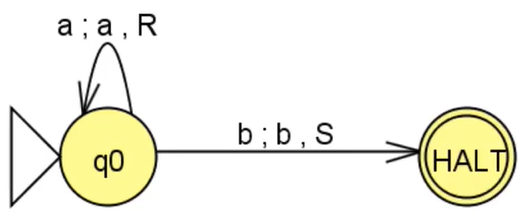
* Under what circumstances you can run the instruction (i.e. current “weather conditions”)
  + Current machine state
  + Symbol written on tape (pointed to by tape head)
* How to run the instruction:
  + What to write on the tape
  + Direction to move the tape head
  + Next machine state

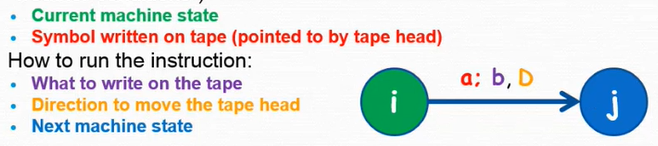
## **JFLAP TM Setup**

Before doing anything, go under Preferences, Turing Machine Preferences, and make sure that “Enable Transitions from Turing Machine Final States” is unchecked, and make sure that “Accept by Final State” and “Allow stay for tape head on transition” are checked.

When making Turing Machines in JFLAP, select “Turing Machine with Building Blocks”.

Example: Build a TM over {a,b}: Halt at b. (*This accepts any string of a’s and b’s that has a b in it. You can have as many a’s as you want, but once you see a b, you stop, signalled by the halt.*)

* Make 2 states.
* Make the first state the initial state.
* Make the second state the final state. (*This is supposed to be the halt state.*)
* Use a transition creator and click on the initial state to make the loop.
  + 1st spot: a
  + 2nd spot: a
  + 3rd spot: R
* These three spots mean if you see an a, replace it with an a (aka don’t change it), and move the tape head to the right
* Go from the initial state to the final with a transition
  + 1st spot: b
  + 2nd spot: b
  + 3rd spot: S (stay put)
* The final look is to the right:



Possible moves: L (left), R (right), S (stay - don’t move)

To be consistent with the textbook, we’ll have only one final state called “HALT”.

Thus, STOP and ACCEPT immediately when you get to ANY final state.

**Turing Machine Assumptions**

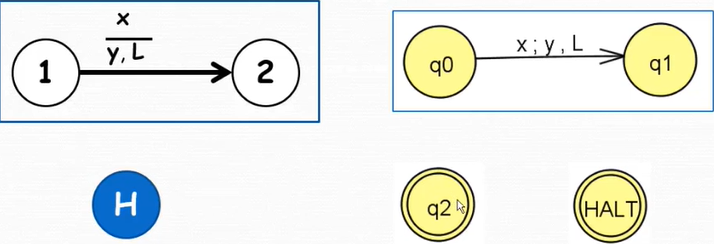
* The input string is written on the tape and all other tape cells contain ^
* The tape head starts at the leftmost symbol of the input string
* There is one start state and one halt state
* The TM stop if there is no valid move or if it arrives at the halt state
* A string is accepted by the TM if the machine enters the halt state

Example: TM over {a,b}: a\*b (2nd version of the previous example)

JFLAP uses ☐ while the textbook uses ^ for special symbol blank.

* Make 3 states
* Make the first state initial
* Make the last one the final state
* Make a loop at the initial state (a ; a, R)
* The transition from the first to the second state: (b ; b, R)
* The transition from the second to the third state: (☐ ; ☐ , S)

**JFLAP vs Textbook**

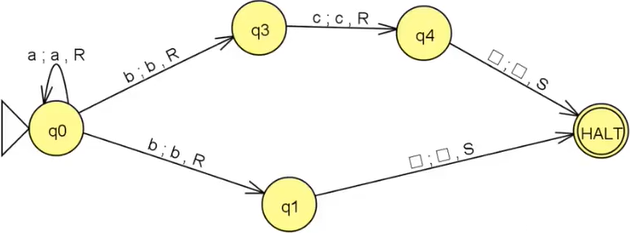
****

## **Determinism**

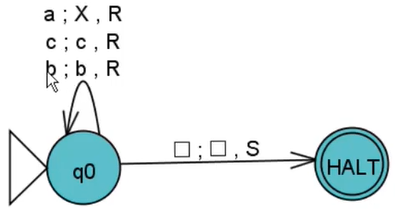
If a Turing Machine has at least 2 instructions with the same state and input letter (“weather conditions”), then it is non-deterministic; otherwise, it is deterministic.

You can test in JFLAP with “Multiple Run”.

Example: Nondeterministic TM for a\*b + a\*bc



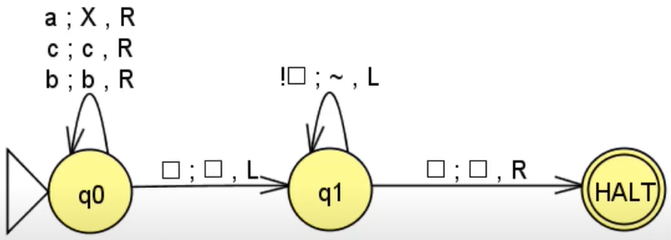
Example: TM over {a,b,c} that “crosses out” a’s (*Replace all the a’s with X’s*)



Example: TM over {a,b,c} that “crosses out” a’s and then goes back to the start of string

* Replace all the a’s with X’s
* Go left until the tape head is pointing at the first char in the string
* Special symbol anything but blank !☐ (type ! and it’ll work)
* Special symbol ~ (meaning “last thing you saw”, but can be buggy)

So, the first initial state is the same. Create a second state where it is (☐ ; ☐ , L) transition to it from the first one. Then, make a loop on that second state (meaning if you see a not-empty space, leave it alone, whatever the last thing you saw, and go left). Then, make a transition to the final state (☐ ; ☐ , R) since the tape head has to be at the start of the string.



Example: TM that “crosses out” a’s and capitalizes b’s and returns to the start

To do this is easy. For the b loop transition, just have it as (b ; B ; R) instead.

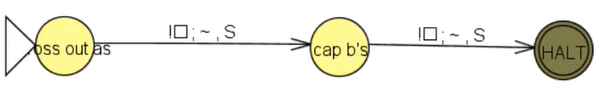
You could also do the following:

* Click on the “Building Block Creator” and click where you want to put the new building block in the white space
* For this problem, put the previous example twice in the white space
* Right click, “Edit Block”, and you can change it to take out the “a” transition and replace it with (X ; X , R) and put the (b ; B , R) loop on top.
* Make the first building block an initial state
* Make a transition between the blocks (!☐ ; ~ , S)
* Create another state and make it final with the same transition as the previous step

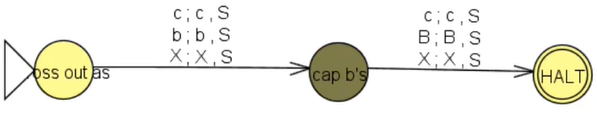
You can choose “Step by Building Block” to go in depth in the building block.

Choosing “Step” won’t show you what’s going on.

If it halts in a building block, it will find another out transition (outside the block).

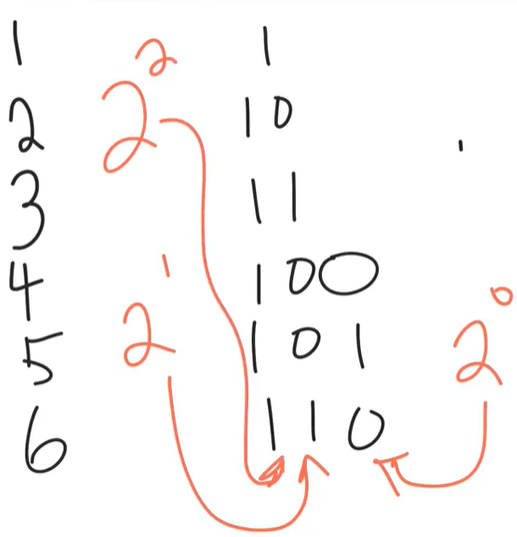


*HOWEVER*, the problem with this is that it accepts characters NOT in the alphabet. Thus, the transitions outside of the building blocks should look like so:



DO NOT use “Block Transition Creator”.

Example: TM that adds 1 to a binary number. (*Remember how to add binary numbers! Review v*)

Remember, 1 + 0 = 1 and 1 + 1 = 0. You keep carrying the 1 when it’s 1 + 1 in a particularly long number until you get to the 0.

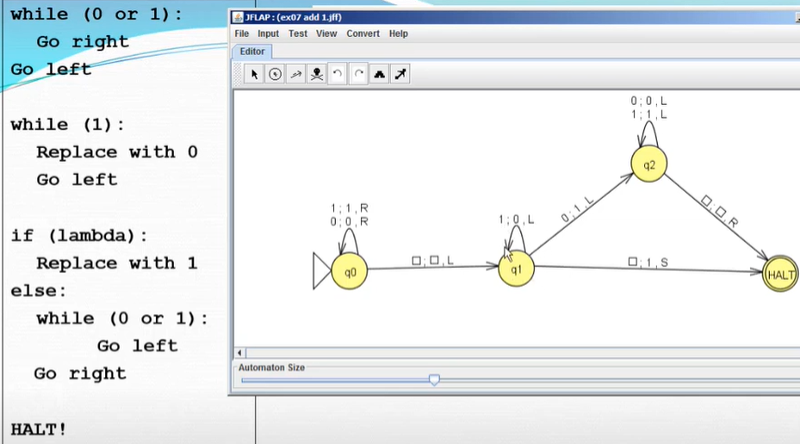
Algorithm:

* Move to the right of string, making sure it’s just 1’s

and 0’s until you see ^

* Move left one cell
* Repeat: If current cell contains 1, write 0 and left
* Until current cell contains 0 or ^
* Write 1
* Move to left of string and halt

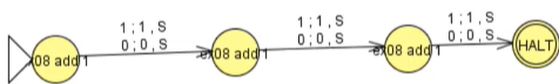
How this looks like:



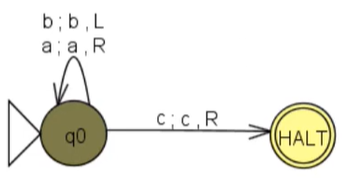
Example: TM that adds 3 to a binary #

Make building blocks of the +1 three times (because add three 1’s = 3). Now you need to make the initial state the first block, and make another state for the HALT final state.

Then, just add (1 ; 1 , S) and (0 ; 0 , S) for all the transitions between the building blocks and HALT states.



Example: Infinite loop example



If you input the string “aaaabbb” in it, it will keep going back and forth between a and b (the ones next to each other) infinitely. The a tells you to go to the right, but the b tells you to go to the left.

Accepts: a\*c(a + b + c)\*

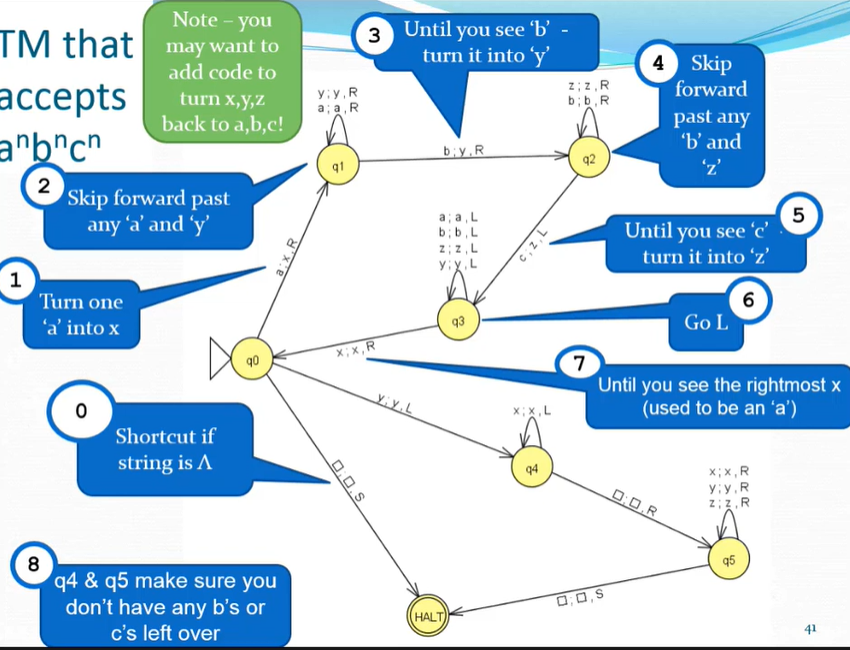
Loops on: a\*b(a + b + c)\*

## **Facts to Memorize**

The Turing Machine we use is equivalent to:

* TM with a tape that is only infinite in one direction
* TM with multiple heads
* TM with multiple heads *and* multiple tapes

Example: TM that accepts a*n*b*n*c*n*



Only if it reaches the HALT, it accepts, even if it hasn’t processed the entire string.

# **Definitions**

A problem is a yes or no question that we ask about a particular input. Examples are as follows:

* Given one input *x*, is *x* a prime number?
* Given 3 inputs *x*, *y*, and *z*, does (*x* + *y*) \* *y* = *z*?
* Given two inputs: a TM *Q* and a string *s*, does *Q* accept *s*?

An algorithm is a solution to a problem if it:

1. Gives the correct answer AND
2. It is guaranteed to run in a finite amount of time

# **The Halting Problem**

Given two inputs:

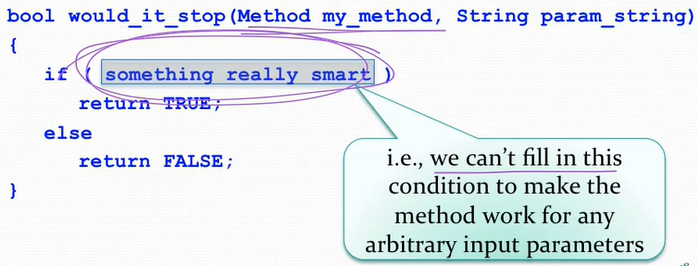
* my\_method: a Java method
* param\_string: a string of parameters that could be given to my\_method

Write another method (called would\_it\_stop) that:

* Returns false if my\_method would get stuck in an infinite loop given parameters param\_string
* Returns true if my\_method would eventually finish running and give you an answer given parameters param\_string

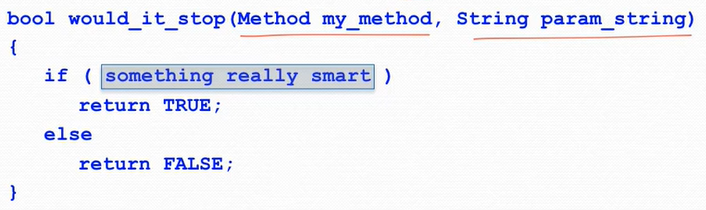
Stopper is a method that always stops. would\_stopper\_stop would always return true.

Loopy is a method that never stops running (stuck in an infinite loop). would\_loopy\_stop would always return false.



It is *impossible* to write a generic would\_it\_stop method!

**Proof:** Assume it is possible to write would\_it\_stop.



(It’s not possible to fill up the parameter for if to make this work)

Then, you could check any method with any param string, so:

would\_it\_stop(some\_method, “abc”)

would\_it\_stop(cool\_method, <text of US Constitution>)

would\_it\_stop(cool\_method, <text of cool\_method>)

would\_it\_stop(another\_method, “3.14, 7”)

Well, if you can write would\_it\_stop for any parameters, then you can write a new method called stops\_on\_self:

boolean stops\_on\_self(Method m)

{

boolean answer = would\_it\_stop(m, <text of m>);

return answer;

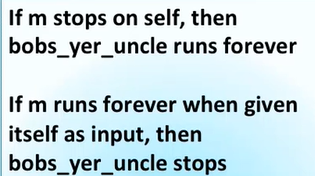
}

So, for example,

stops\_on\_self(stopper) returns TRUE

stops\_on\_self(looper) returns FALSE

If you can write stops\_on\_self, then you can write bobs\_yer\_uncle:



boolean bobs\_yer\_uncle(Method m)

{

if (stops\_on\_self(m)) {

while (TRUE)

{ // Infinite loop

}

}

else {

return TRUE;

}

}

What happens if you run bobs\_yer\_uncle on itself?

If bobs\_yer\_uncle(bobs\_yer\_uncle) loops forever,

stops\_on\_self(bobs\_yer\_uncle) returns FALSE,

meaning bobs\_yer\_uncle(bobs\_yer\_uncle) stops.

We’re not sure if that’s true, it could be wrong. There’s no contradiction as long as the other case is okay.

If bobs\_yer\_uncle(bobs\_yer\_uncle) stops,

stops\_on\_self(bobs\_yer\_uncle) returns TRUE,

meaning bobs\_yer\_uncle(bobs\_yer\_uncle) loops forever.

Now, both cases don’t work. No matter what is done, you end up with a contradiction. Thus, the assumption must be false. So, you can’t write would\_it\_stop. So, there exists a program that is impossible to write!

QED.