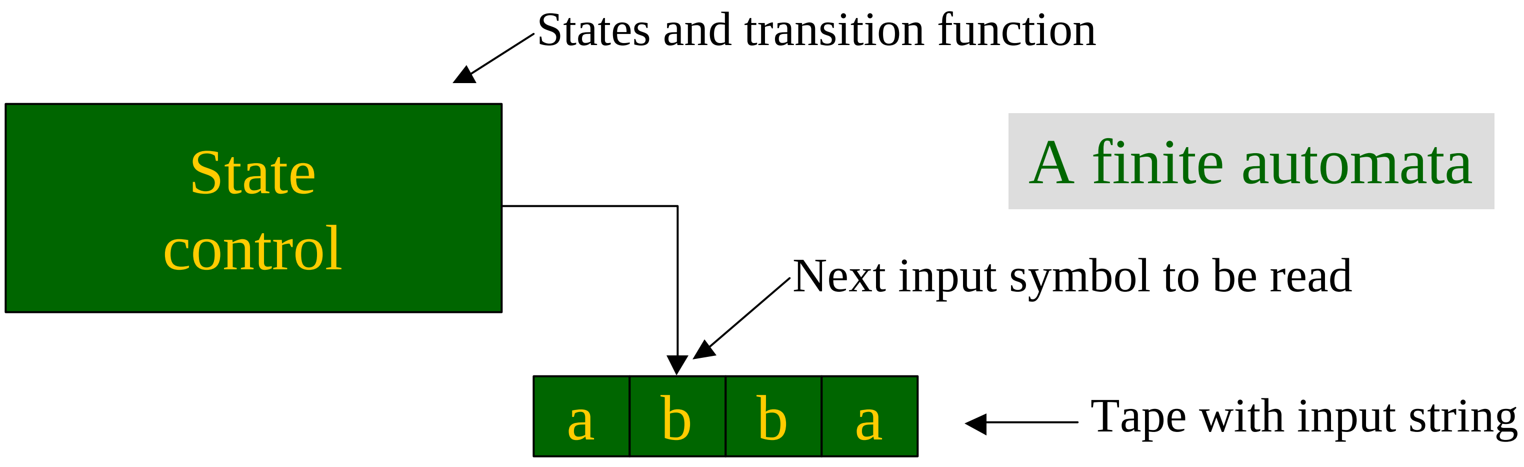
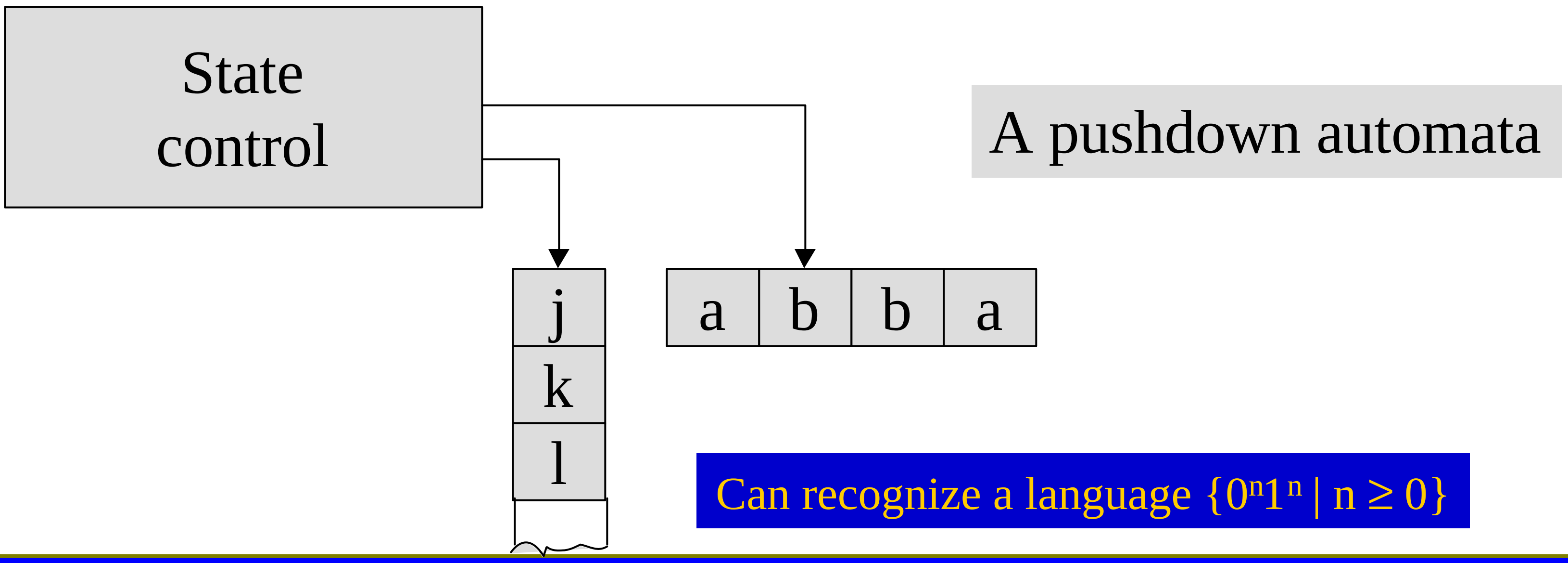
16.070

Introduction to Computers & Programming

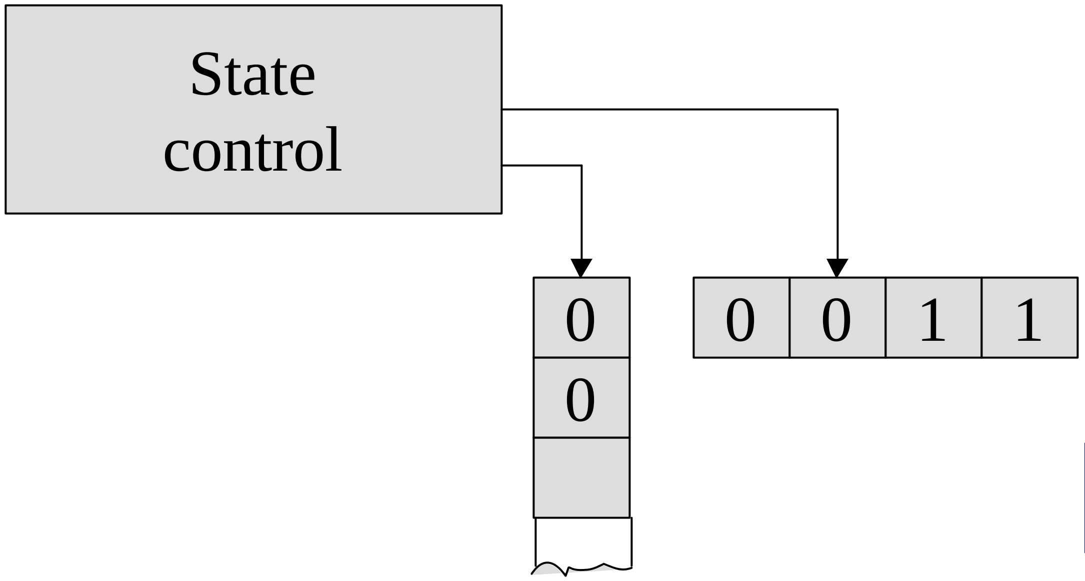
Theory of computation 5: Reducibility, Turing machines

Prof. Kristina Lundqvist
Dept. of Aero/Astro, MIT





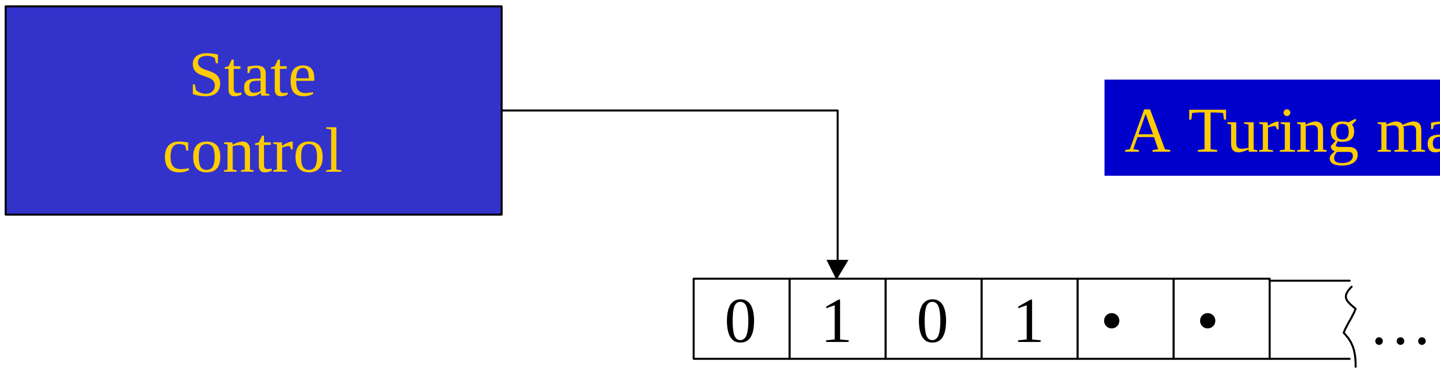
Can recognize a language $\{0^n1^n \mid n \geq 0\}$



$\{0^n 1^n \mid n \geq 0\}$

Read symbols from the input. As each 0 is read, push it onto the stack. As soon as 1s are read, pop a 0 from stack. If reading the input finishes at the same time as the stack becomes empty of 0s, the input is **accepted**. If the stack becomes empty while still reading 1s, or if the 1s are finished while there still are 0s on the stack, **reject** the input.

$\{a^n b^n c^n \mid n \geq 0\}$ is not context free



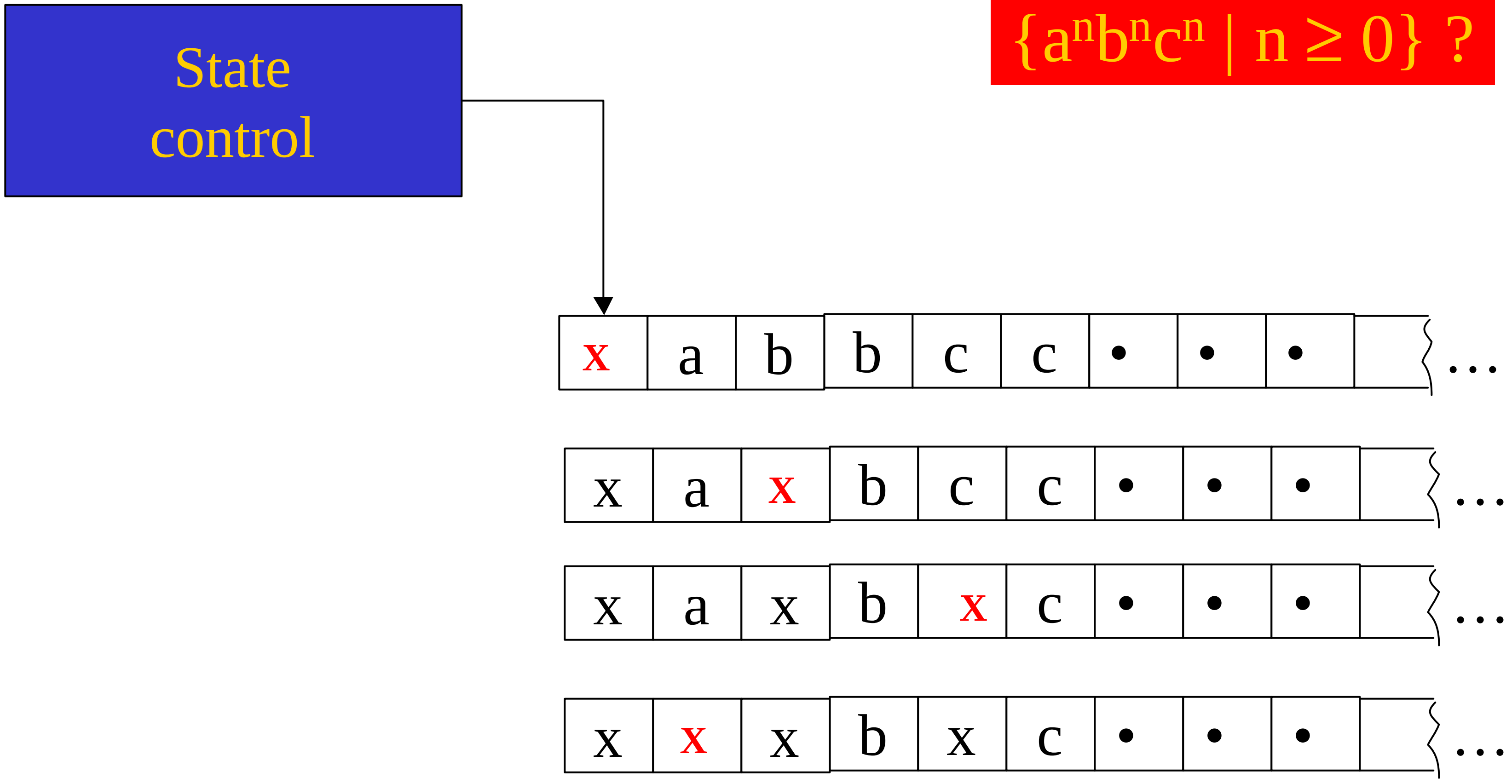
A Turing machine

Differences between finite automata and Turing machines:

1. A TM can both **write** on the tape and **read** from it
2. The read-write head can **move** both to the left and to the right
3. The tape is **infinite**
4. The **special states** for rejecting and accepting take immediate effects

Can a TM recognize: $\{a^n b^n c^n \mid n \geq 0\}$?

$\{a^nb^nc^n \mid n \geq 0\}$?



$\{a^nb^nc^n \mid n \geq 0\}$!!

Reducibility

- We use the ***Turing machine*** as our model of a general purpose computer.
- The primary method for proving that problems are computationally unsolvable is called: ***Reducibility***
- A ***reduction*** is a way of converting one problem into another problem in such a way that a solution to the second problem can be used to solve the first problem.

- **Example**:
  - Problem of finding your way around in a new city, can be reduced to the problem of obtaining a map of the city
  - Problem of measuring the area of a rectangle reduces to the problem of measuring its height and width

Problem Reducibility

- Once we have a single problem known to be *undecidable* we can determine that other problems are also undecidable by **reducing** a known undecidable problem to the new problem.

- To use this idea: take a problem known to be undecidable. Call this problem U. Given a new problem, P, if U can be *reduced* to P so that P can be used to solve U, **then** P must also be undecidable.

Problem Reducibility

- **Important** – we must show that U reduces to P, not vice versa
  - If we show that our P reduces to U then we have only shown that a new problem can be solved by the undecidable problem
  - It might still be possible to solve problem P by other means; e.g. we might be taking the tough path to solve P
- **But** if we can show the other direction, that P can solve U, then P must be at least as hard as U, which we already know to be undecidable.

Reducibility Example

- Does program **Q** ever call function *foo*?
  - This problem is also *undecidable*

- Just as we saw with the ‘hello world’ problem, it is easy to write a program that can determine if some programs call function *foo*.

- But we could have a program that contains lots of control logic to determine whether or not function *foo* is invoked. This **general case is much harder**, and in fact ***undecidable***

Reducibility Example

- Use the reduction technique for the Hello-World problem
  - **Rename** the function "foo" in program Q and all calls to that function.
  - **Add** a function "foo" that does nothing and is not called.
  - **Modify** the program to remember the first 12 characters that it prints, storing them in array A
  - **Modify** the program so that whenever it executes any output statement, it checks the array A to see if the 12 characters written are "hello, world" and if so, invokes function foo.
  - If the final program prints "hello, world" then it must also invoke function foo. Similarly, if the program does not print "hello, world" then it does not invoke foo.

Foo Caller

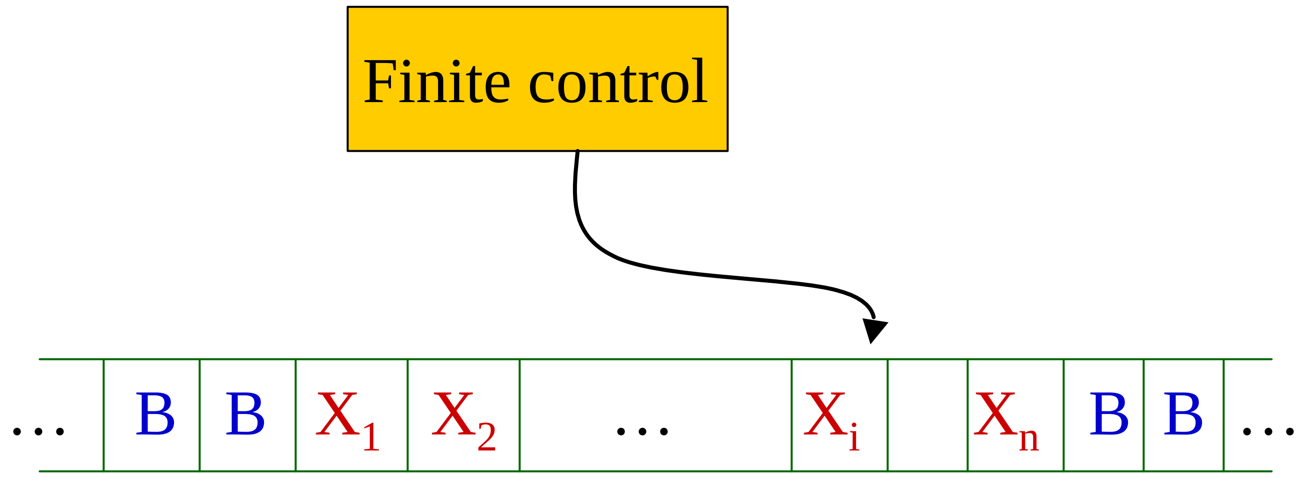
- Say that we have a program F-Test that can determine if a program calls foo.
- If we run F-Test on the modified program above, not only can it determine if a program calls foo, it can also determine if the program prints “hello, world”.
- But we would then be solving the “hello-world-tester” problem which we already know is undecidable, therefore our F-Test problem must be undecidable as well

Turing Machines

- TM’s described in 1936
  - Turing machines were first proposed by Alan Turing, in an attempt to give a mathematically precise definition of "algorithm" or "mechanical procedure".
  - Well before the days of modern computers but remains a popular model for what is possible to compute on today’s systems
  - Advances in computing still fall under the TM model, so even if they may run faster, they are still subject to the same limitations
- A TM consists of a **finite control** (i.e. a finite state automaton) that is connected to an **infinite tape**.

Turing Machine

- The **tape** consists of **cells** where each cell holds a **symbol from the tape alphabet**. Initially the input consists of a finite-length string of symbols and is placed on the tape. To the left of the input and to the right of the input, extending to infinity, are placed **blanks**. The **tape head** is initially positioned at the leftmost cell holding the input.



Turing Machine Details

- In one move the TM will:
  - **Change state**, which may be the same as the current state
  - **Write a tape symbol** in the current cell, which may be the same as the current symbol
  - **Move the tape head** left or right one cell
- Formally, the Turing Machine is denoted by the 7-tuple:
  - $M = (Q, \Sigma, G, d, q_0, B, F)$

Turing Machine Description

$$M = (Q, \Sigma, G, d, q_0, B, F)$$

- Q = finite states of the control
- Σ = finite set of **input symbols**, which is a subset of G below
- G = finite set of **tape symbols**
- d = transition function. $d(q,X)$ are a state and tape symbol X.
  - The output is the triple, (p, Y, D)
  - Where p = next state, Y = new symbol written on the tape, D = direction to move the tape head
- q_0= start state for finite control
- B = blank symbol. This symbol is in G but not in Σ.
- F = set of final or accepting states of Q.

TM Example

- Make a TM that recognizes the language

$$L = \{ w\#w \mid w \in (0,1)^* \}$$

That is, we have a language separated by a # symbol with the same string on both sides.

- Here is a strategy we can employ to create the TM:

1. Scan the input to make sure it **contains a single # symbol**. If not, reject.
2. **Starting with the leftmost symbol**, remember it and write an X into its cell. Move to the right, skipping over any 0's or 1's until we reach a #. Continue scanning to the first non-# symbol. If this symbol matches the original leftmost symbol, then write a # into the cell. Otherwise, reject.
3. **Move the head back** to the leftmost symbol that is not X.
4. If this symbol is not #, then repeat at step 2. Otherwise, scan to the right. If all symbols are # until we hit a blank, then accept. Otherwise, reject.

TM Example

- Typically we will describe TM's in this **informal fashion**. The formal description gets quite long and tedious. Nevertheless, we will give a formal description for this particular problem.

- We can use a table format or a **transition diagram** format. In the transition diagram format, a transition is denoted by:

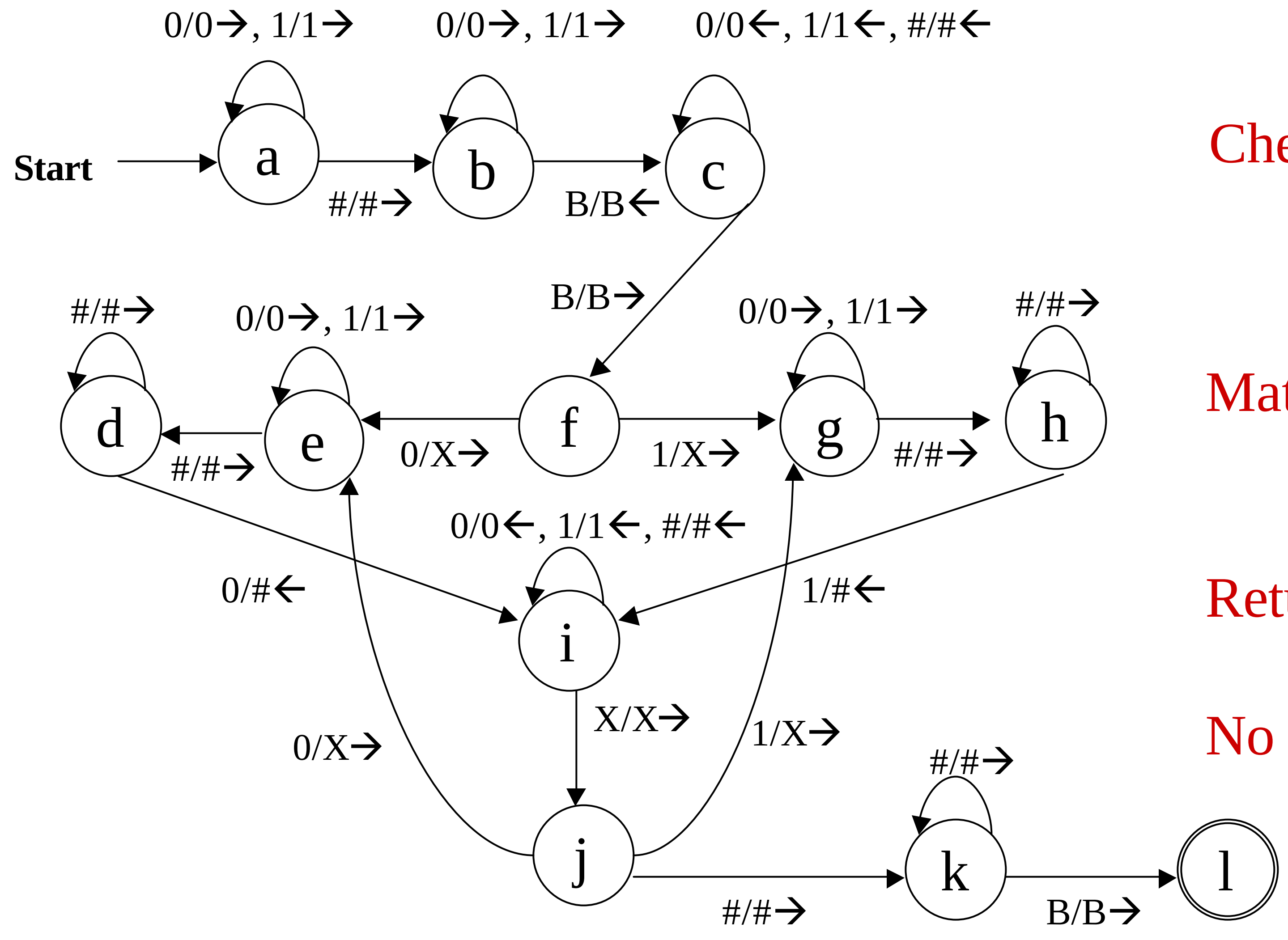
  Input symbol / Symbol-To-Write Direction to Move

  For example:

  0 / 1 →

- Means take this transition if the input is 0, and replace the cell with a 1 and then move to the right.

TM for L={w#w|w∈(0,1)*}



Check for #

Match symbols

Return to left

No input left

Instantaneous Description

- Sometimes it is useful to describe what a TM does in terms of its **ID** (instantaneous description), just as we did with the *PDA*.
- The **ID** shows all non-blank cells in the tape, pointer to the cell the head is over with the name of the current state
  - use the turnstile symbol + to denote the move.
  - As before, to denote zero or many moves, we can use + *.
- For example, for the above TM on the input 10#10 we can describe our processing as follows:

  Ba10#10B + B1a0#10B + B10a#10B + B10#b10B + B10#1b0B +
  B10#10bB + B10#1c0B + * cB10#10B + Bf10#10B + * BXX#XXBl

- In this example the blanks that border the input symbols are shown since they are used in the Turing machine to define the borders of our input.

Turing Machines and Halting

- One way for a TM to accept input is to end in a **final state**.
  - Another way is **acceptance by halting**. We say that a **TM halts** if it enters a state q, scanning a tape symbol X, and there is no move in this situation; i.e. **d(q,X) is undefined**.
- **Note** that this definition of halting was not used in the transition diagram for the TM we described earlier; instead *that TM died on unspecified input*!
- It is possible to modify the prior example so that there is no unspecified input except for our accepting state. An equivalent TM that halts exists for a TM that accepts input via final state.
- In general, we assume that **a TM always halts when it is in an accepting state**.
- Unfortunately, it is *not always possible to require that a TM halts even if it does not accept the input*. Such languages are called ***recursive***.
  Turing machines that **always halt**, regardless of accepting or not accepting, *are good models of algorithms for **decidable** problems.*

Turing Machine Variants

- There are many variations we can make to the basic TM
  - Extensions can make it useful to prove a theorem or perform some task
  - However, these extensions do **not** add anything extra the basic TM can't already compute

- **Example**: consider a variation to the Turing machine where we have the *option of staying put* instead of forcing the tape head to move left or right by one cell.
  - In the old model, we could replace each "stay put" move in the new machine with two transitions, one that moves right and one that moves left, to get the same behavior.

Multitape Turing Machines

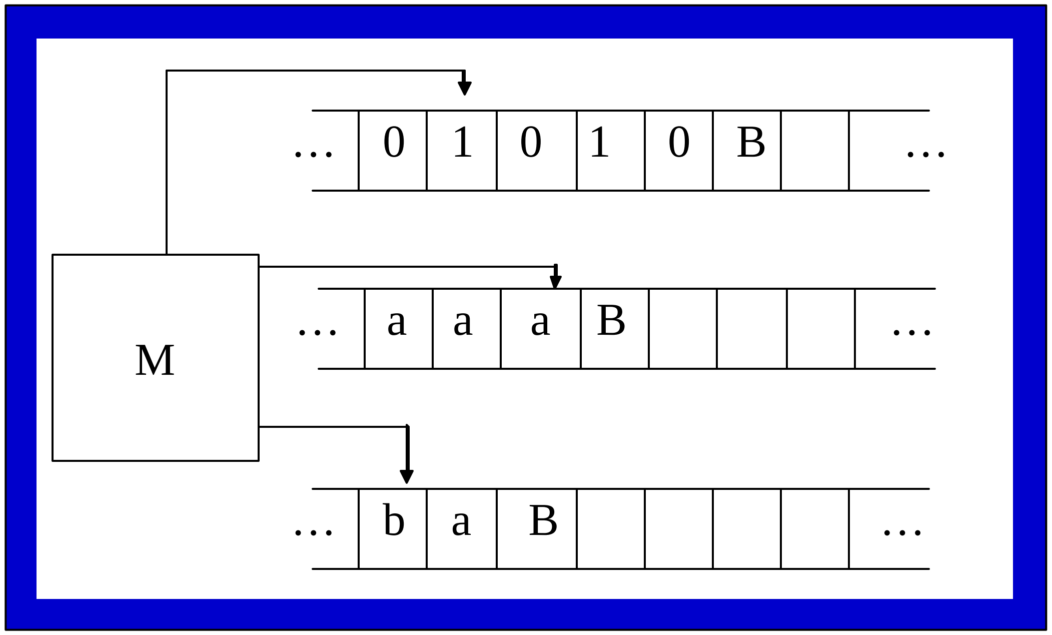
- A ***multitape*** Turing machine is like an ordinary TM but it has several tapes instead of one tape.
- Initially the **input starts on tape 1** and the other tapes are blank.
- The **transition function is changed** to allow for reading, writing, and moving the heads on all the tapes simultaneously.
  - This means we could read on multiple tape and move in different directions on each tape as well as write a different symbol on each tape, all in one move.

Multitape Turing Machine

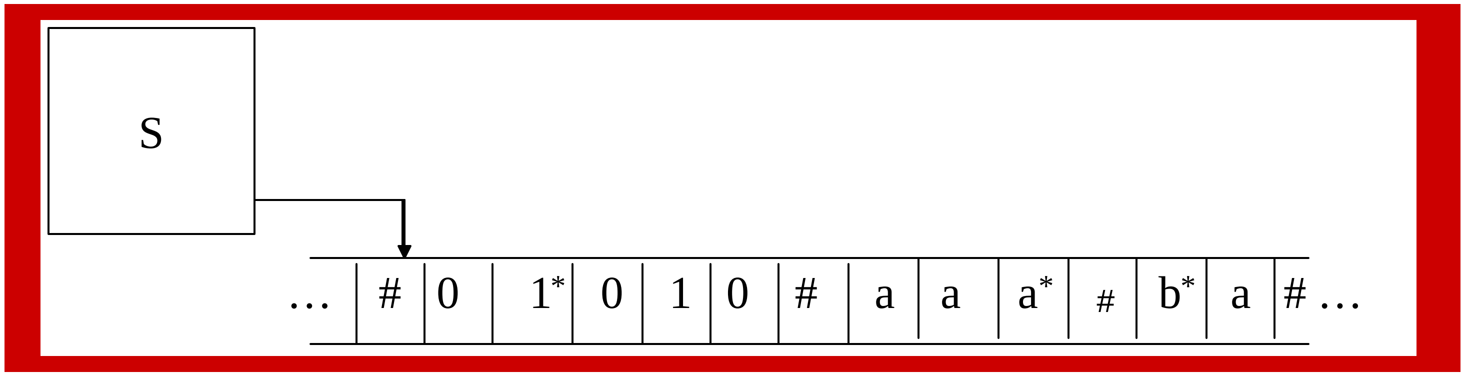
- Theorem: **A multitape TM is equivalent in power to an ordinary TM**. Recall that *two TM's are equivalent if they recognize the same language*. We can show how to convert a multitape TM, M, to a single tape TM, S:

- Say that M has k tapes.
  - Create the TM S to simulate having k tapes by interleaving the information on each of the k tapes on its single tape
  - Use a new symbol # as a delimiter to separate the contents of each tape
  - S must also keep track of the location on each of the simulated heads
    - Write a type symbol with a * to mark the place where the head on the tape would be
    - The * symbols are new tape symbols that don't exist with M
    - The finite control must have the proper logic to distinguish say, x* and x and realize both refer to the same thing, but one is the current tape symbol.

Multitape Machine



Equivalent Single Tape Machine:



Single Tape Equivalent

- One final detail
  - If at any point S moves one of the virtual tape heads onto a #, then this action signifies that M has moved the corresponding head onto the previously unread blank portion of that tape.
  - To accommodate this situation, S writes a blank symbol on this tape cell and shifts the tape contents to the rightmost # by one, adds a new #, and then continues back where it left off

Nondeterministic TM

- Replace the "DFA" part of the TM with an "NFA"
  - Each time we make a **nondeterministic** move, you can think of this as a **branch** or "fork" **to two simultaneously running machines**. Each machine gets a **copy of the entire tape**. If any one of these machines ends up in an accepting state, then the input is accepted.
- Although powerful, **nondeterminism does not affect the power of the TM model**
- **Theorem**: Every nondeterministic TM has an equivalent deterministic TM.
  - We can prove this theorem by simulating any nondeterministic TM, N, with a deterministic TM, D.

Equivalence of TM's and Computers

- In one sense, a real computer has a finite number of states, and thus is **weaker** than a TM.
- But, we can postulate an infinite supply of tapes, disks, or some peripheral storage device to **simulate an infinite TM tape**. Additionally, we can assume there is a human operator to mount disks, keep them stacked neatly on the sides of the computer, etc.
- Need to show both directions, **a TM can simulate a computer** and **that a computer can simulate a TM**

Computer Simulate a TM

- This direction is fairly easy - Given a computer with a modern programming language, certainly, we can write a computer program that emulates the finite control of the TM.

- The only issue remains the infinite tape. Our program must map cells in the tape to storage locations in a disk. When the disk becomes full, we must be able to map to a different disk in the stack of disks mounted by the human operator.

TM Simulate a Computer

- In this exercise the simulation is performed at the level of stored instructions and accessing words of main memory.
  - TM has one tape that holds all the used memory locations and their contents.
  - Other TM tapes hold the instruction counter, memory address, computer input file, and scratch data.
  - The computer's instruction cycle is simulated by:
    1. Find the word indicated by the instruction counter on the memory tape.
    2. Examine the instruction code (a finite set of options), and get the contents of any memory words mentioned in the instruction, using the scratch tape.
    3. Perform the instruction, changing any words' values as needed, and adding new address-value pairs to the memory tape, if needed.

TM/Computer Equivalence

- Anything a computer can do, a TM can do, and vice versa
- TM is much slower than the computer, though
  - But the difference in speed is polynomial
  - Each step done on the computer can be completed in O(n^2) steps on the TM.
- While slow, this is key information if we wish to make an analogy to modern computers. Anything that we can prove using Turing machines translates to modern computers with a polynomial time transformation.
- Whenever we talk about defining algorithms to solve problems, we can equivalently talk about how to construct a TM to solve the problem. If a TM cannot be built to solve a particular problem, then it means our modern computer cannot solve the problem either.