Undecidability

Everything is an Integer

Countable and Uncountable Sets

Turing Machines

Recursive and Recursively Enumerable Languages

Integers, Strings, and Other Things

- Data types have become very important as a programming tool.
- But at another level, there is only one type, which you may think of as integers or strings.
- Key point: Strings that are programs are just another way to think about the same one data type.

Example: Text

- Strings of ASCII or Unicode characters can be thought of as binary strings, with 8 or 16 bits/character.
- Binary strings can be thought of as integers.
- It makes sense to talk about “the i-th string.”

Binary Strings to Integers

- There's a small glitch:
  - If you think simply of binary integers, then strings like 101, 0101, 00101,… all appear to be "the fifth string."
- Fix by prepending a "1" to the string before converting to an integer.
  - Thus, 101, 0101, and 00101 are the 13th, 21st, and 37th strings, respectively.

Example: Images

- Represent an image in (say) GIF.
- The GIF file is an ASCII string.
- Convert string to binary.
- Convert binary string to integer.
- Now we have a notion of “the i-th image.”

Example: Proofs

- A formal proof is a sequence of logical expressions, each of which follows from the ones before it.
- Encode mathematical expressions of any kind in Unicode.
- Convert expression to a binary string and then an integer.

Proofs – (2)

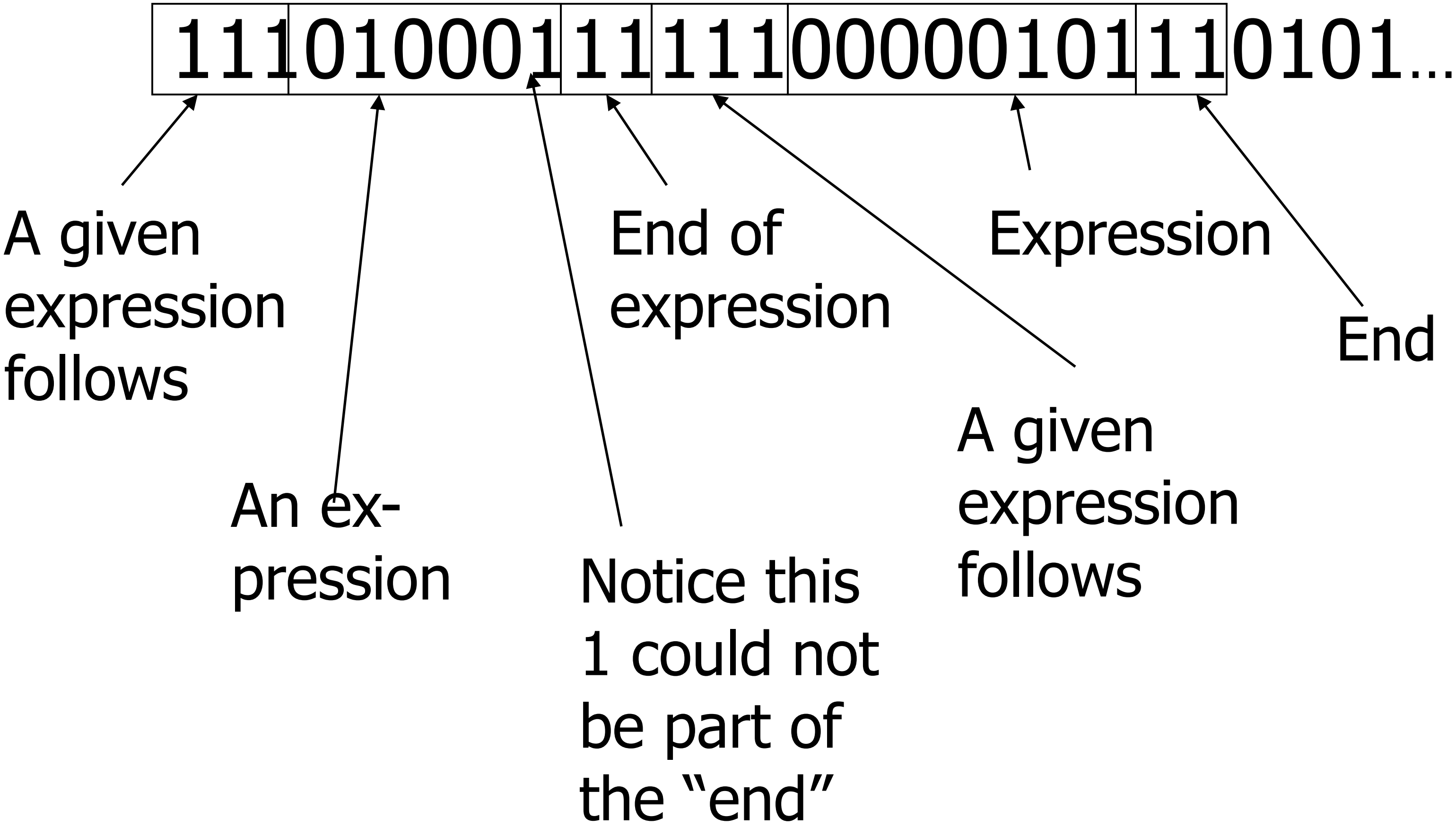
- But a proof is a sequence of expressions, so we need a way to separate them.
- Also, we need to indicate which expressions are given and which follow from previous expressions.

Proofs – (3)

- Quick-and-dirty way to introduce new symbols into binary strings:

1. Given a binary string, precede each bit by 0.
   - Example: 101 becomes 010001.
2. Use strings of two or more 1's as the special symbols.
   - Example: 111 = "the following expression is given"; 11 = "end of expression."

Example: Encoding Proofs



111 010001 11 111 00000101 11 0101…

A given expression follows

An ex-pression

Notice this 1 could not be part of the "end"

End of expression

A given expression follows

Expression

End

Example: Programs

- Programs are just another kind of data.
- Represent a program in ASCII.
- Convert to a binary string, then to an integer.
- Thus, it makes sense to talk about "the i-th program."
- Hmm…There aren't all that many programs.

Finite Sets

- A *finite set* has a particular integer that is the count of the number of members.
- Example: {a, b, c} is a finite set; its *cardinality* is 3.
- It is impossible to find a 1-1 mapping between a finite set and a proper subset of itself.

Infinite Sets

- Formally, an *infinite set* is a set for which there is a 1-1 correspondence between itself and a proper subset of itself.
- Example: the positive integers {1, 2, 3,…} is an infinite set.
  - There is a 1-1 correspondence 1<->2, 2<->4, 3<->6,… between this set and a proper subset (the set of even integers).

Countable Sets

- A *countable set* is a set with a 1-1 correspondence with the positive integers.
  - Hence, all countable sets are infinite.
- Example: All integers.
  - 0<->1; -i <-> 2i; +i <-> 2i+1.
  - Thus, order is 0, -1, 1, -2, 2, -3, 3,…
- Examples: set of binary strings, set of Java programs.

Example: Pairs of Integers

- Order the pairs of positive integers first by sum, then by first component:
- [1,1], [2,1], [1,2], [3,1], [2,2], [1,3], [4,1], [3,2],…, [1,4], [5,1],…
- Interesting exercise: figure out the function f(i,j) such that the pair [i,j] corresponds to the integer f(i,j) in this order.

Enumerations

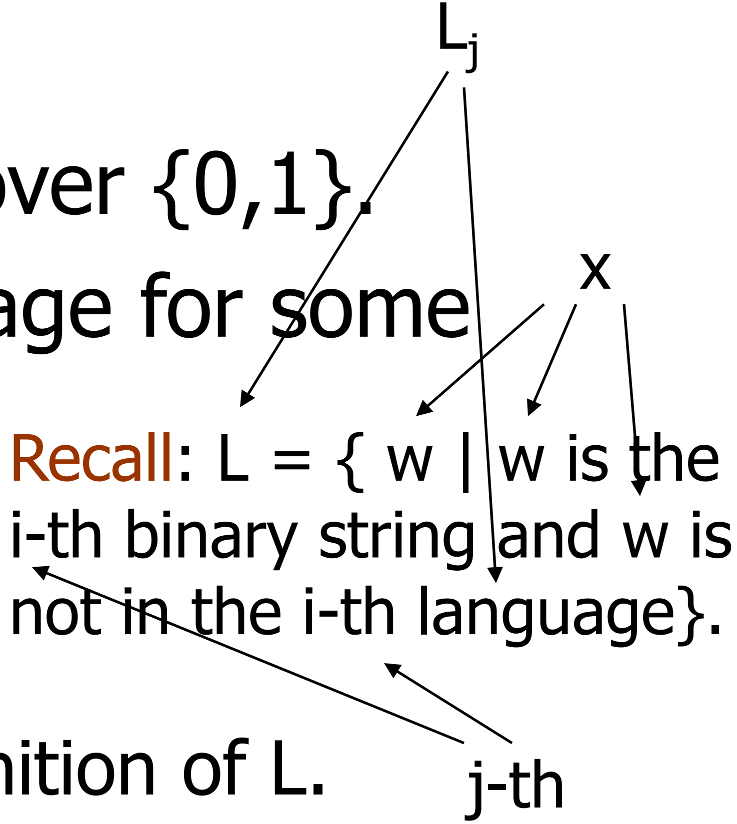
- An *enumeration* of a set is a 1-1 correspondence between the set and the positive integers.
- Thus, we have seen enumerations for strings, programs, proofs, and pairs of integers.

How Many Languages?

- Are the languages over {0,1} countable?
- No; here's a proof.
- Suppose we could enumerate all languages over {0,1} and talk about "the i-th language."
- Consider the language L = { w | w is the i-th binary string and w is not in the i-th language}.

Proof – Continued

- Clearly, L is a language over {0,1}.
- Thus, it is the j-th language for some particular j.
- Let x be the j-th string.
- Is x in L?
  - If so, x is not in L by definition of L.
  - If not, then x is in L by definition of L.



Recall: L = { w | w is the i-th binary string and w is not in the i-th language}.

Proof – Concluded

- We have a contradiction: x is neither in L nor not in L, so our sole assumption (that there was an enumeration of the languages) is wrong.
- Comment: This is really bad; there are more languages than programs.
- E.g., there are languages with no membership algorithm.

Diagonalization Picture

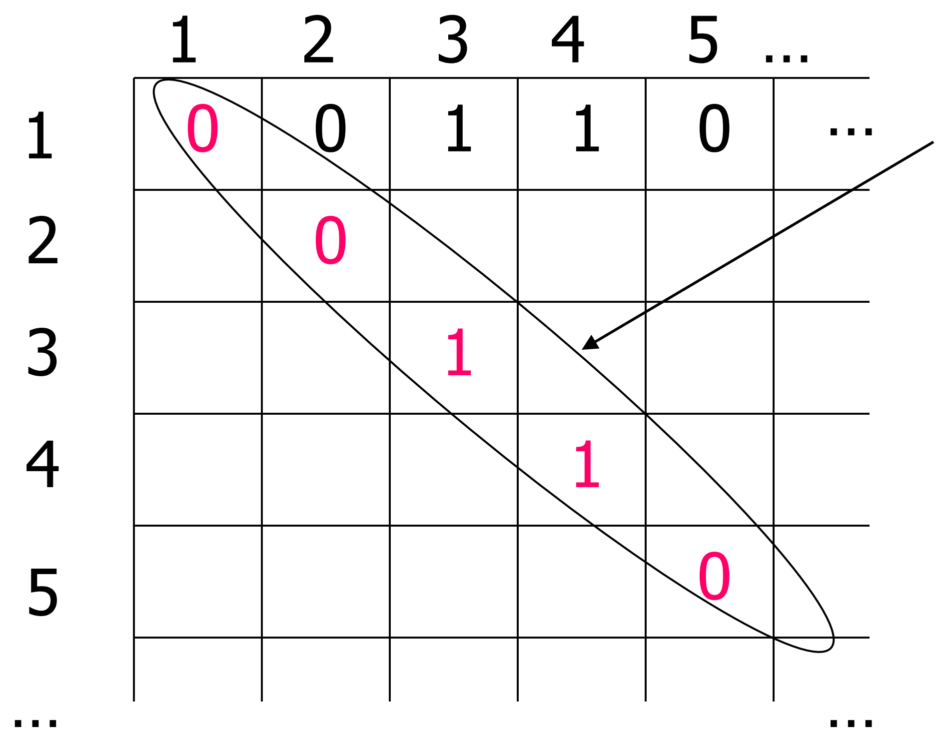
Strings

Languages

	1	2	3	4	5	…
1	1	0	1	1	0	…
2		1				
3			0			
4				0		
5					1	
…						…

Diagonalization Picture

Flip each diagonal entry



Strings

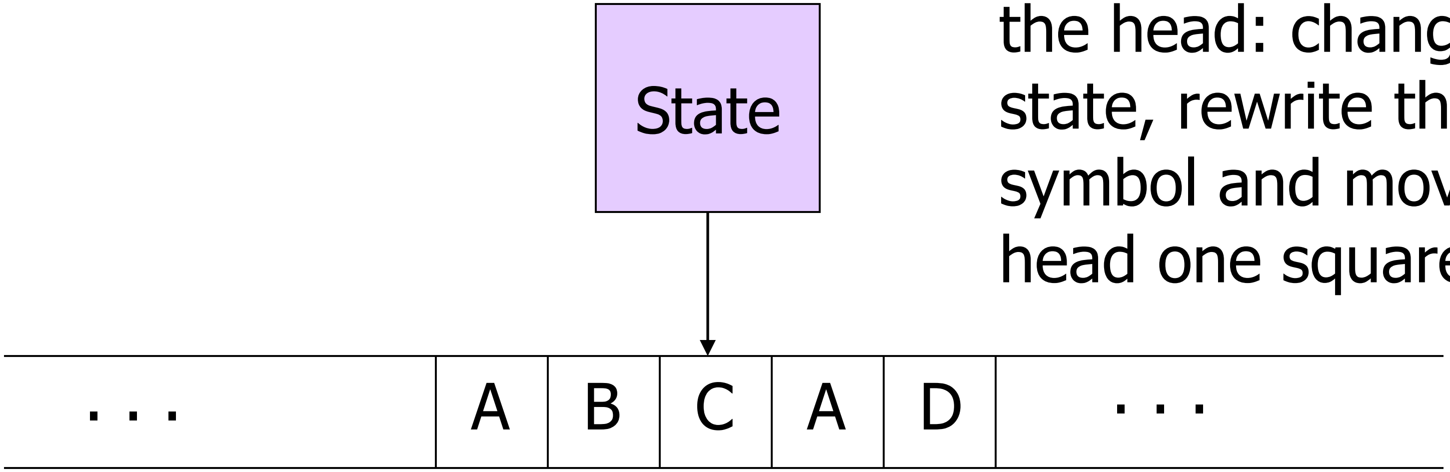
Languages	1	2	3	4	5	…
1	0	0	1	1	0	…
2		0				
3			1			
4				1		
5					0	
…						…

Can’t be a row – it disagrees in an entry of each row.

Turing-Machine Theory

- The purpose of the theory of Turing machines is to prove that certain specific languages have no algorithm.
- Start with a language about Turing machines themselves.
- Reductions are used to prove more common questions undecidable.

Picture of a Turing Machine



Action: based on the state and the tape symbol under the head: change state, rewrite the symbol and move the head one square.

Infinite tape with squares containing tape symbols chosen from a finite alphabet

Why Turing Machines?

- Why not deal with C programs or something like that?
- Answer: You can, but it is easier to prove things about TM's, because they are so simple.
  - And yet they are as powerful as any computer.
    - More so, in fact, since they have infinite memory.

Turing-Machine Formalism

- A TM is described by:
  1. A finite set of *states* (Q, typically).
  2. An *input alphabet* (Σ, typically).
  3. A *tape alphabet* (Γ, typically; contains Σ).
  4. A *transition function* (δ, typically).
  5. A *start state* (q_0, in Q, typically).
  6. A *blank symbol* (B, in Γ– Σ, typically).
     - All tape except for the input is blank initially.
  7. A set of *final states* ($F \subseteq Q$, typically).

Conventions

- a, b, … are input symbols.
- …, X, Y, Z are tape symbols.
- …, w, x, y, z are strings of input symbols.
- α, β,… are strings of tape symbols.

The Transition Function

- Takes two arguments:
  1. A state, in Q.
  2. A tape symbol in Γ.
- δ(q, Z) is either undefined or a triple of the form (p, Y, D).
  - p is a state.
  - Y is the new tape symbol.
  - D is a *direction*, L or R.

Example: Turing Machine

- This TM scans its input right, looking for a 1.
- If it finds one, it changes it to a 0, goes to final state f, and halts.
- If it reaches a blank, it changes it to a 1 and moves left.

Example: Turing Machine – (2)

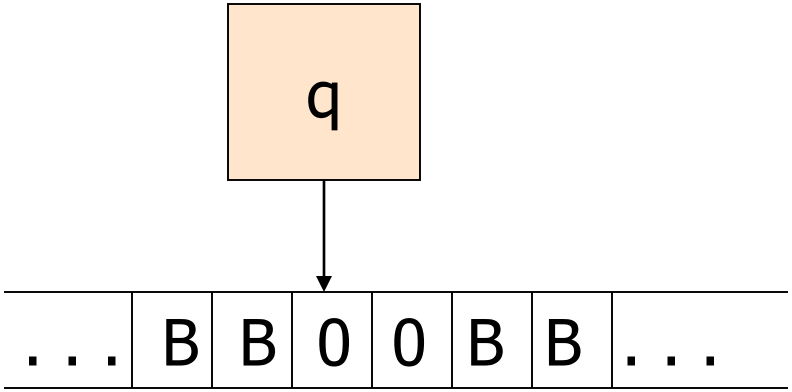
- States = {q (start), f (final)}.
- Input symbols = {0, 1}.
- Tape symbols = {0, 1, B}.
- δ(q, 0) = (q, 0, R).
- δ(q, 1) = (f, 0, R).
- δ(q, B) = (q, 1, L).

Simulation of TM

δ(q, 0) = (q, 0, R)

δ(q, 1) = (f, 0, R)

δ(q, B) = (q, 1, L)

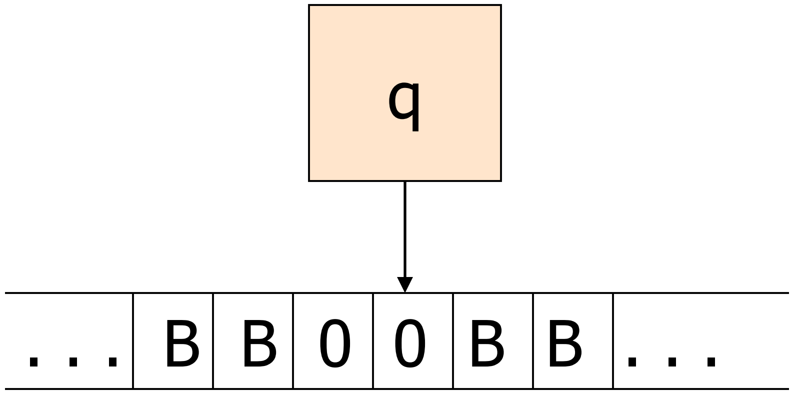


Simulation of TM

δ(q, 0) = (q, 0, R)

δ(q, 1) = (f, 0, R)

δ(q, B) = (q, 1, L)

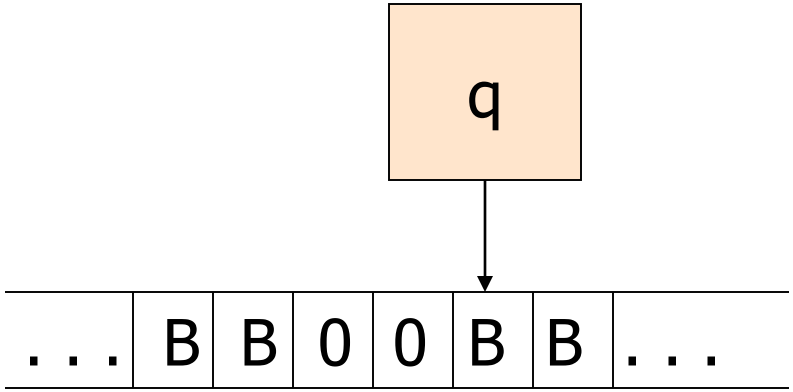


Simulation of TM

δ(q, 0) = (q, 0, R)

δ(q, 1) = (f, 0, R)

δ(q, B) = (q, 1, L)

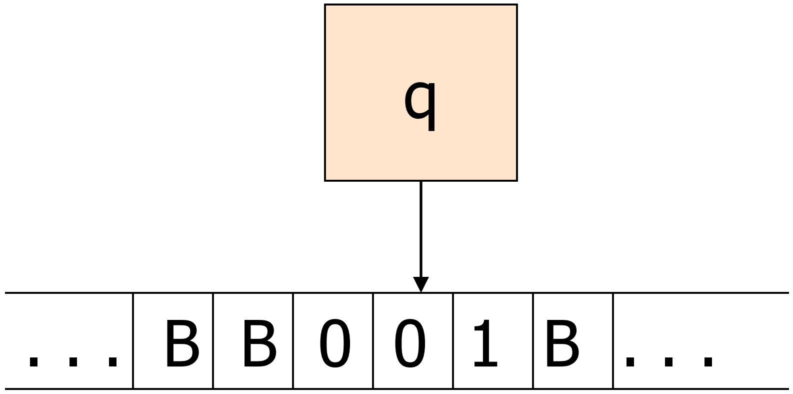


Simulation of TM

$\delta(q, 0) = (q, 0, R)$

$\delta(q, 1) = (f, 0, R)$

$\delta(q, B) = (q, 1, L)$

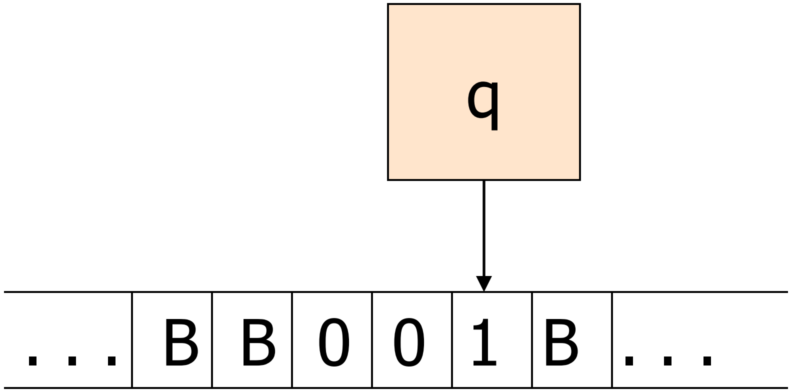


Simulation of TM

$\delta(q, 0) = (q, 0, R)$

$\delta(q, 1) = (f, 0, R)$

$\delta(q, B) = (q, 1, L)$



q

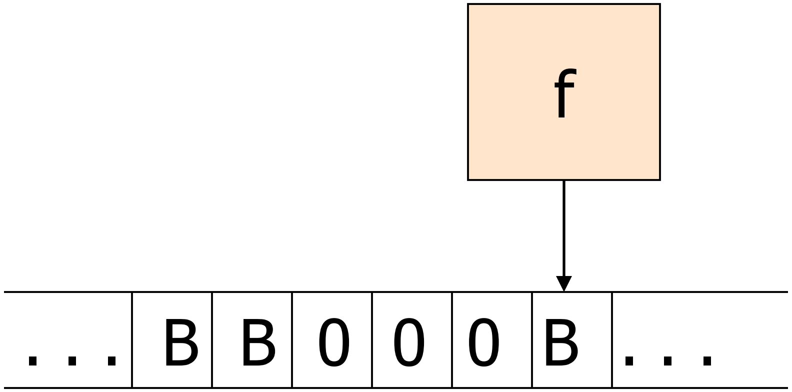
. . . | B | B | 0 | 0 | 1 | B | . . .

Simulation of TM

$\delta(q, 0) = (q, 0, R)$

$\delta(q, 1) = (f, 0, R)$

$\delta(q, B) = (q, 1, L)$



No move is possible. The TM halts and accepts.

Instantaneous Descriptions of a Turing Machine

- Initially, a TM has a tape consisting of a string of input symbols surrounded by an infinity of blanks in both directions.
- The TM is in the start state, and the head is at the leftmost input symbol.

TM ID's – (2)

- An ID is a string $\alpha q \beta$, where $\alpha\beta$ includes the tape between the leftmost and rightmost nonblanks.
- The state q is immediately to the left of the tape symbol scanned.
- If q is at the right end, it is scanning B.
  - If q is scanning a B at the left end, then consecutive B's at and to the right of q are part of α.

TM ID's – (3)

- As for PDA's we may use symbols $\vdash$ and $\vdash$* to represent "becomes in one move" and "becomes in zero or more moves," respectively, on ID's.
- Example: The moves of the previous TM are $q00 \vdash 0q0 \vdash 00q \vdash 0q01 \vdash 00q1 \vdash 000f$

Formal Definition of Moves

1. If $\delta(q, Z) = (p, Y, R)$, then
   - $\alpha qZ\beta \vdash \alpha Yp\beta$
   - If Z is the blank B, then also $\alpha q \vdash \alpha Yp$
2. If $\delta(q, Z) = (p, Y, L)$, then
   - For any X, $\alpha XqZ\beta \vdash \alpha pXY\beta$
   - In addition, $qZ\beta \vdash pBY\beta$

Languages of a TM

- A TM defines a language by final state, as usual.
- $L(M) = \{w \mid q_0w \vdash^* I$, where I is an ID with a final state$\}$.
- Or, a TM can accept a language by halting.
- $H(M) = \{w \mid q_0w \vdash^* I$, and there is no move possible from ID $I\}$.

Equivalence of Accepting and Halting

1. If L = L(M), then there is a TM M’ such that L = H(M’).
2. If L = H(M), then there is a TM M” such that L = L(M”).

Proof of 1: Final State -> Halting

- Modify M to become M′ as follows:
  1. For each final state of M, remove any moves, so M′ halts in that state.
  2. Avoid having M′ accidentally halt.
     - Introduce a new state s, which runs to the right forever; that is δ(s, X) = (s, X, R) for all symbols X.
     - If q is not a final state, and δ(q, X) is undefined, let δ(q, X) = (s, X, R).

Proof of 2: Halting -> Final State

- Modify M to become M” as follows:
  1. Introduce a new state f, the only final state of M”.
  2. f has no moves.
  3. If δ(q, X) is undefined for any state q and symbol X, define it by δ(q, X) = (f, X, R).

Recursively Enumerable Languages

- We now see that the classes of languages defined by TM's using final state and halting are the same.
- This class of languages is called the *recursively enumerable languages*.
  - Why? The term actually predates the Turing machine and refers to another notion of computation of functions.

Recursive Languages

- An *algorithm* is a TM, accepting by final state, that is guaranteed to halt whether or not it accepts.
- If L = L(M) for some TM M that is an algorithm, we say L is a *recursive language*.
  - Why? Again, don't ask; it is a term with a history.

Example: Recursive Languages

- Every CFL is a recursive language.
  - Use the CYK algorithm.
- Almost anything you can think of is recursive.

More About Turing Machines

“Programming Tricks”

Restrictions

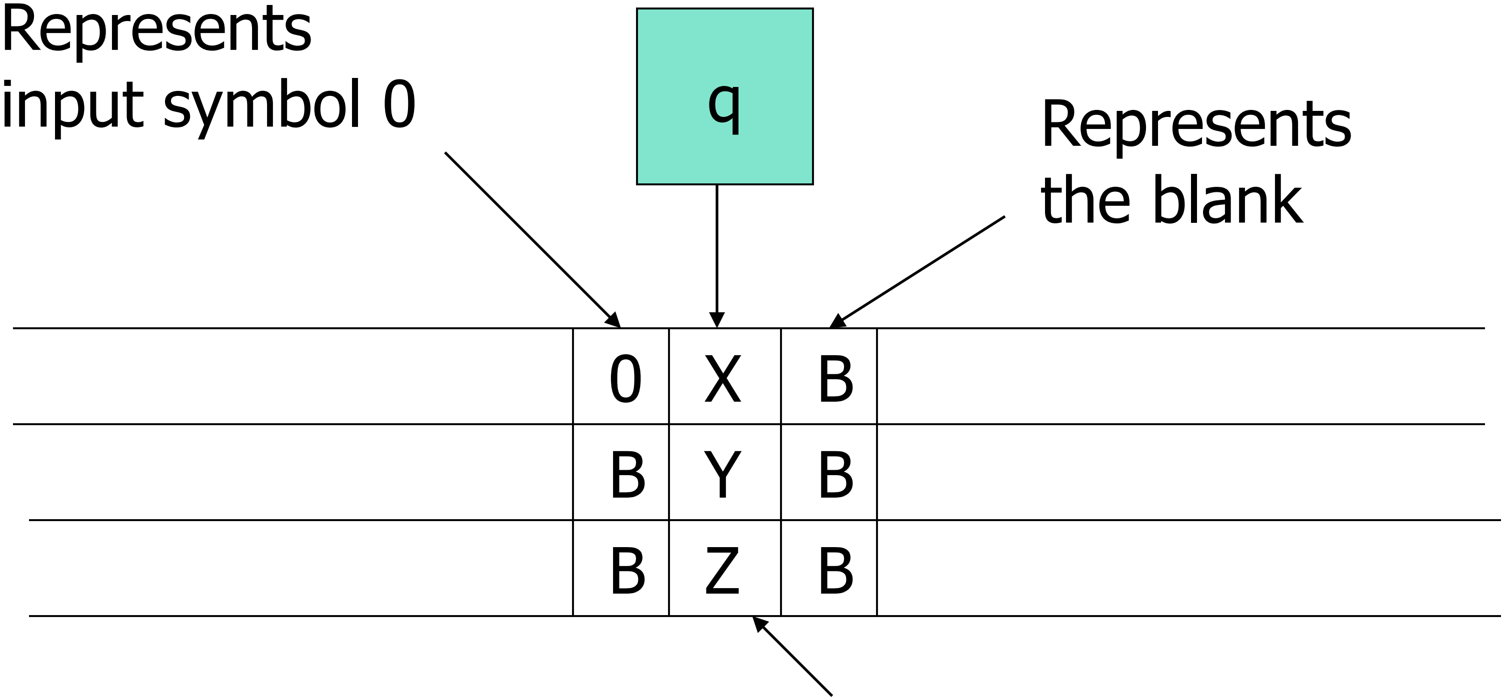
Extensions

Closure Properties

Programming Trick: Multiple Tracks

- Think of tape symbols as vectors with k components, each chosen from a finite alphabet.
- Makes the tape appear to have k tracks.
- Let input symbols be blank in all but one track.

Picture of Multiple Tracks



Represents input symbol 0

Represents the blank

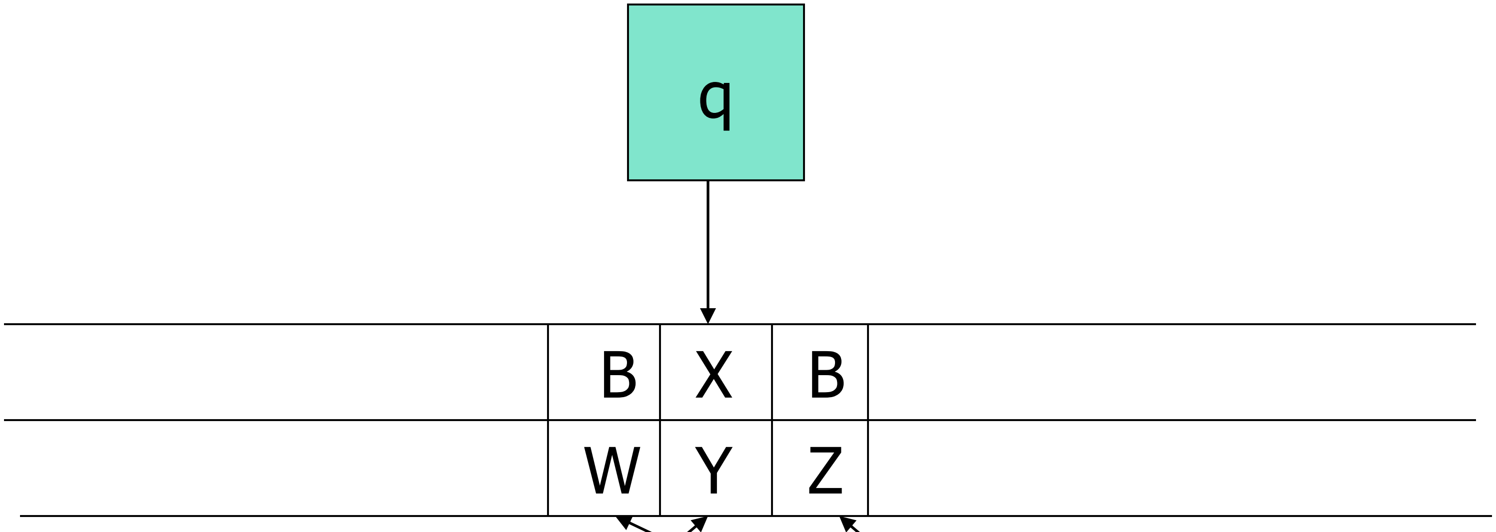
0	X	B
B	Y	B
B	Z	B

Represents one symbol [X,Y,Z]

Programming Trick: Marking

- A common use for an extra track is to *mark* certain positions.
- Almost all tape squares hold B (blank) in this track, but several hold special symbols (marks) that allow the TM to find particular places on the tape.

Marking



Marked Y

Unmarked
W and Z

Programming Trick: Caching in the State

- The state can also be a vector.
- First component is the “control state.”
- Other components hold data from a finite alphabet.

Example: Using These Tricks

- This TM doesn't do anything terribly useful; it copies its input w infinitely.
- Control states:
  - q: Mark your position and remember the input symbol seen.
  - p: Run right, remembering the symbol and looking for a blank. Deposit symbol.
  - r: Run left, looking for the mark.

Example – (2)

- States have the form [x, Y], where x is q, p, or r and Y is 0, 1, or B.
  - Only p uses 0 and 1.
- Tape symbols have the form [U, V].
  - U is either X (the "mark") or B.
  - V is 0, 1 (the input symbols) or B.
  - [B, B] is the TM blank; [B, 0] and [B, 1] are the inputs.

The Transition Function

- **Convention**: *a* and *b* each stand for "either 0 or 1."
- δ([q,B], [B,a]) = ([p,a], [X,a], R).
  - In state q, copy the input symbol under the head (i.e., *a*) into the state.
  - Mark the position read.
  - Go to state p and move right.

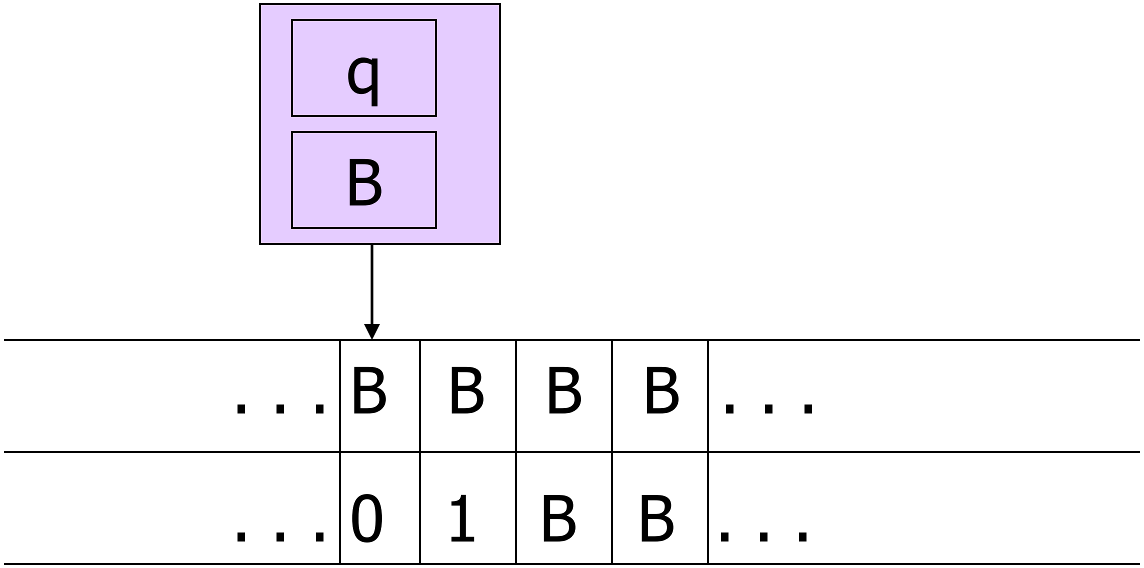
Transition Function – (2)

- δ([p,a], [B,b]) = ([p,a], [B,b], R).
  - In state p, search right, looking for a blank symbol (not just B in the mark track).
- δ([p,a], [B,B]) = ([r,B], [B,a], L).
  - When you find a B, replace it by the symbol (*a*) carried in the "cache."
  - Go to state r and move left.

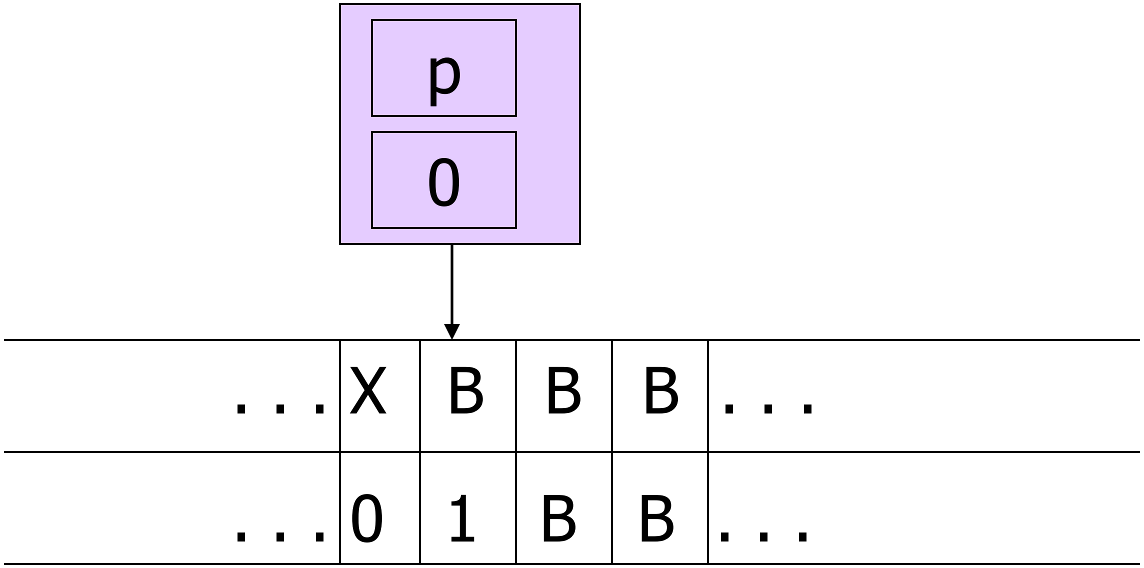
Transition Function – (3)

- δ([r,B], [B,a]) = ([r,B], [B,a], L).
  - In state r, move left, looking for the mark.
- δ([r,B], [X,a]) = ([q,B], [B,a], R).
  - When the mark is found, go to state q and move right.
  - But remove the mark from where it was.
  - q will place a new mark and the cycle repeats.

Simulation of the TM

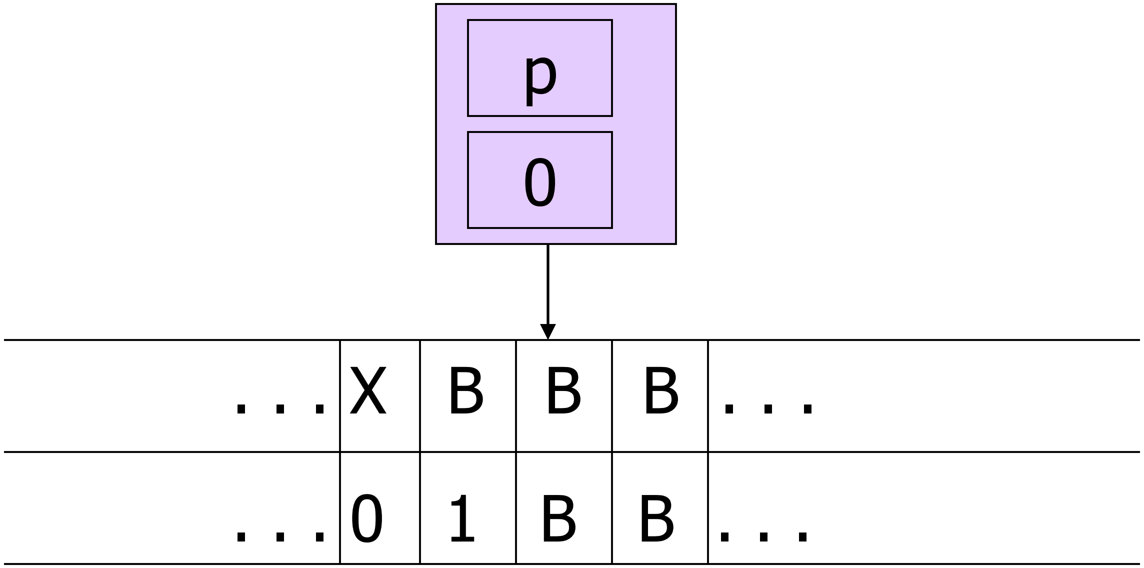


Simulation of the TM

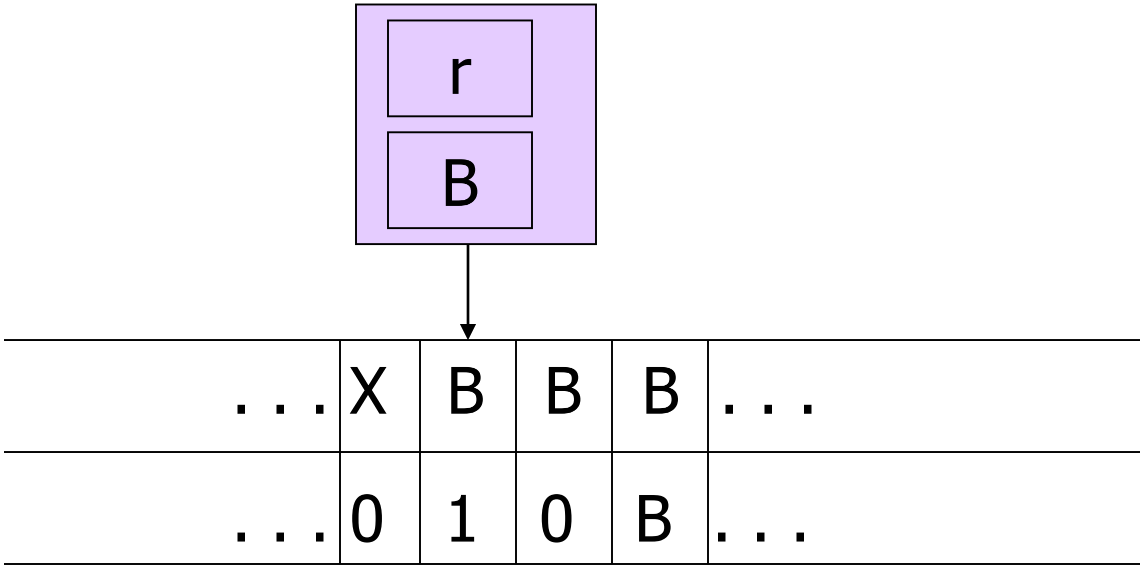


. . .	X	B	B	B	. . .
. . .	0	1	B	B	. . .

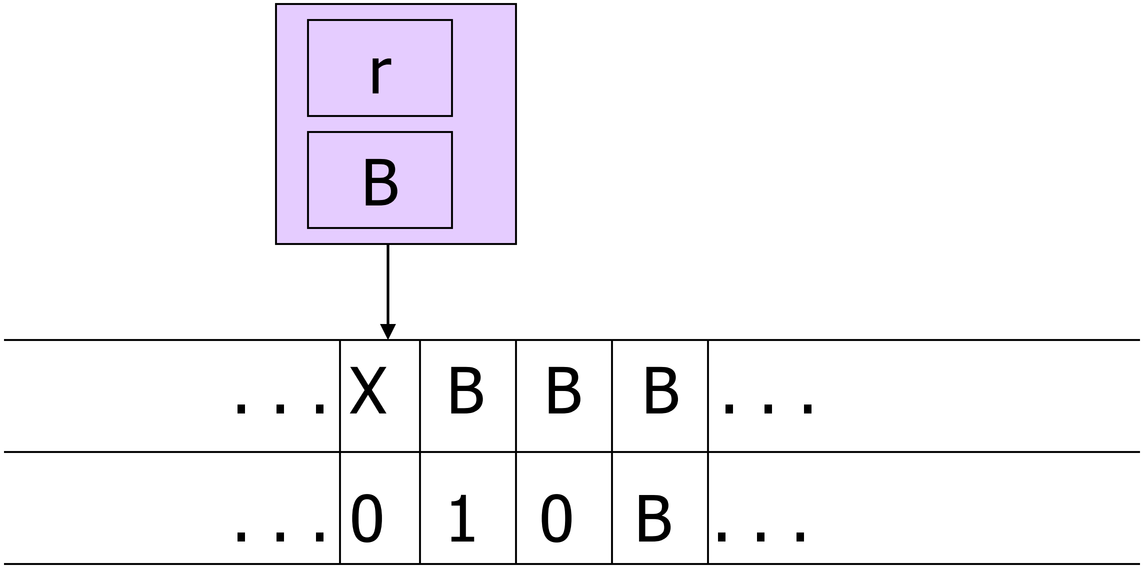
Simulation of the TM



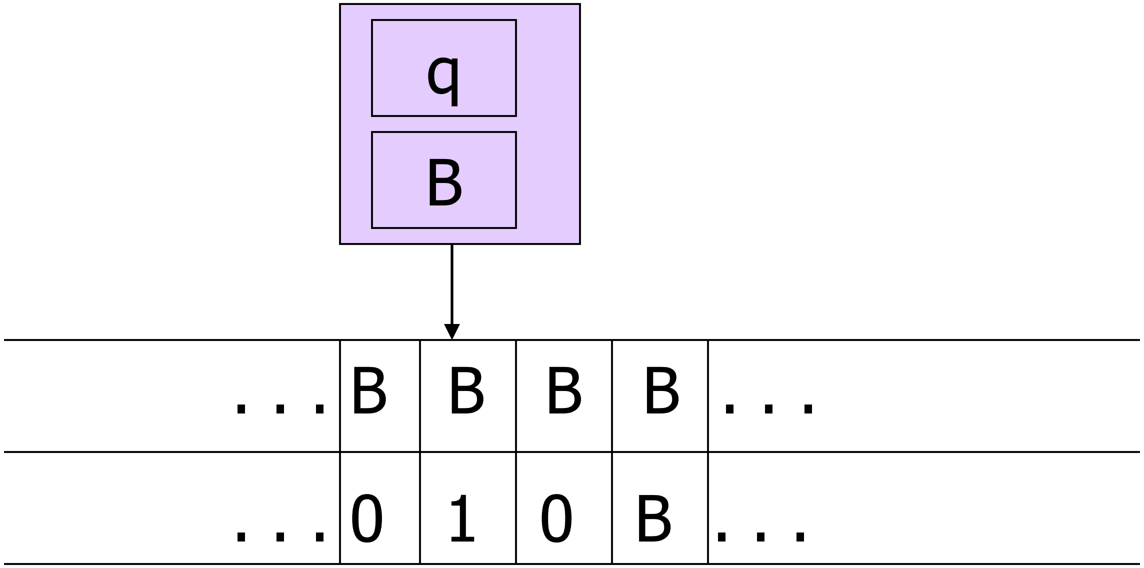
Simulation of the TM



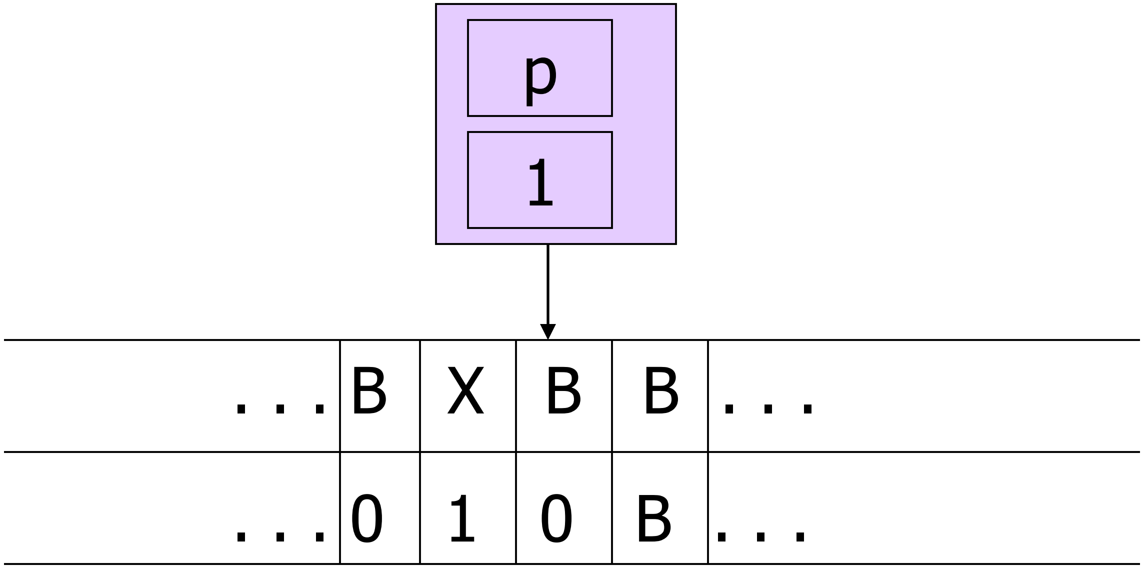
Simulation of the TM



Simulation of the TM



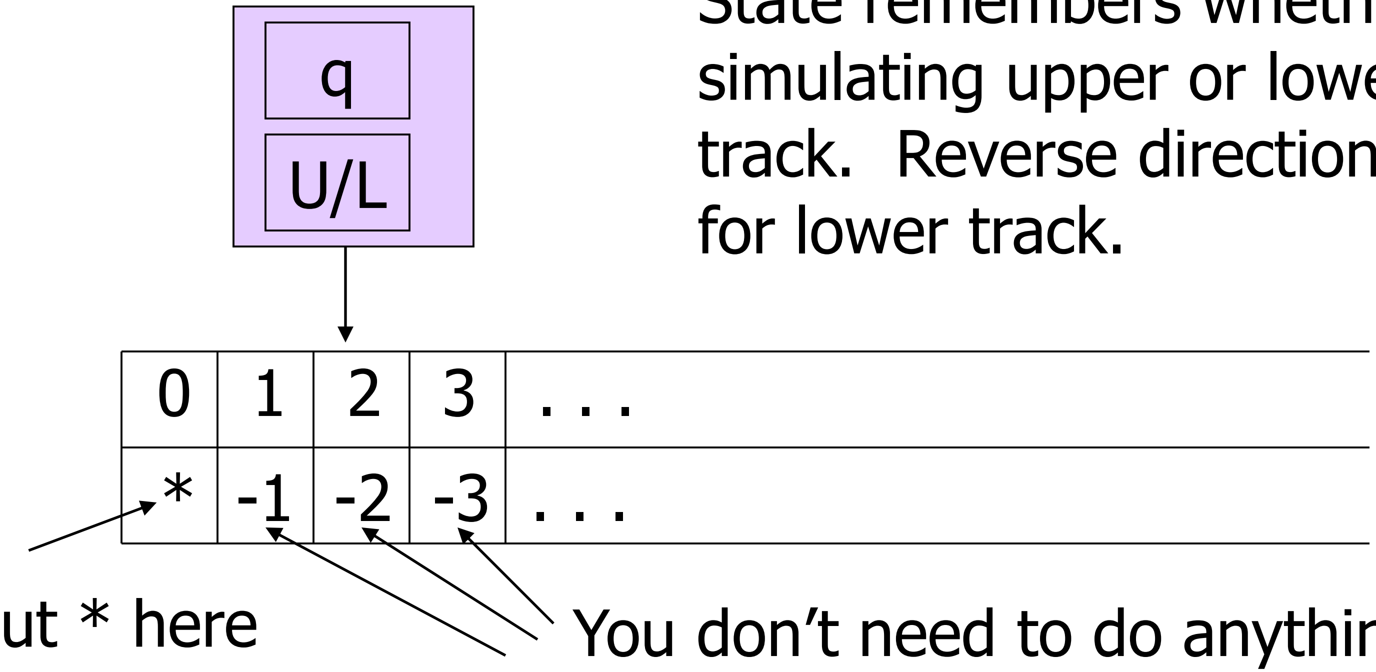
Simulation of the TM



Semi-infinite Tape

- We can assume the TM never moves left from the initial position of the head.
- Let this position be 0; positions to the right are 1, 2, … and positions to the left are –1, –2, …
- New TM has two tracks.
  - Top holds positions 0, 1, 2, …
  - Bottom holds a marker, positions –1, –2, …

Simulating Infinite Tape by Semi-infinite Tape



State remembers whether simulating upper or lower track. Reverse directions for lower track.

Put * here at the first move

You don’t need to do anything, because these are initially B.

More Restrictions

- Two stacks can simulate one tape.
  - One holds positions to the left of the head; the other holds positions to the right.
- In fact, by a clever construction, the two stacks to be *counters* = only two stack symbols, one of which can only appear at the bottom.

Factoid: Invented by Pat Fischer, whose main claim to fame is that he was a victim of the Unabomber.

Extensions

- More general than the standard TM.
- But still only able to define the RE languages.
  1. Multitape TM.
  2. Nondeterministic TM.
  3. Store for name-value pairs.

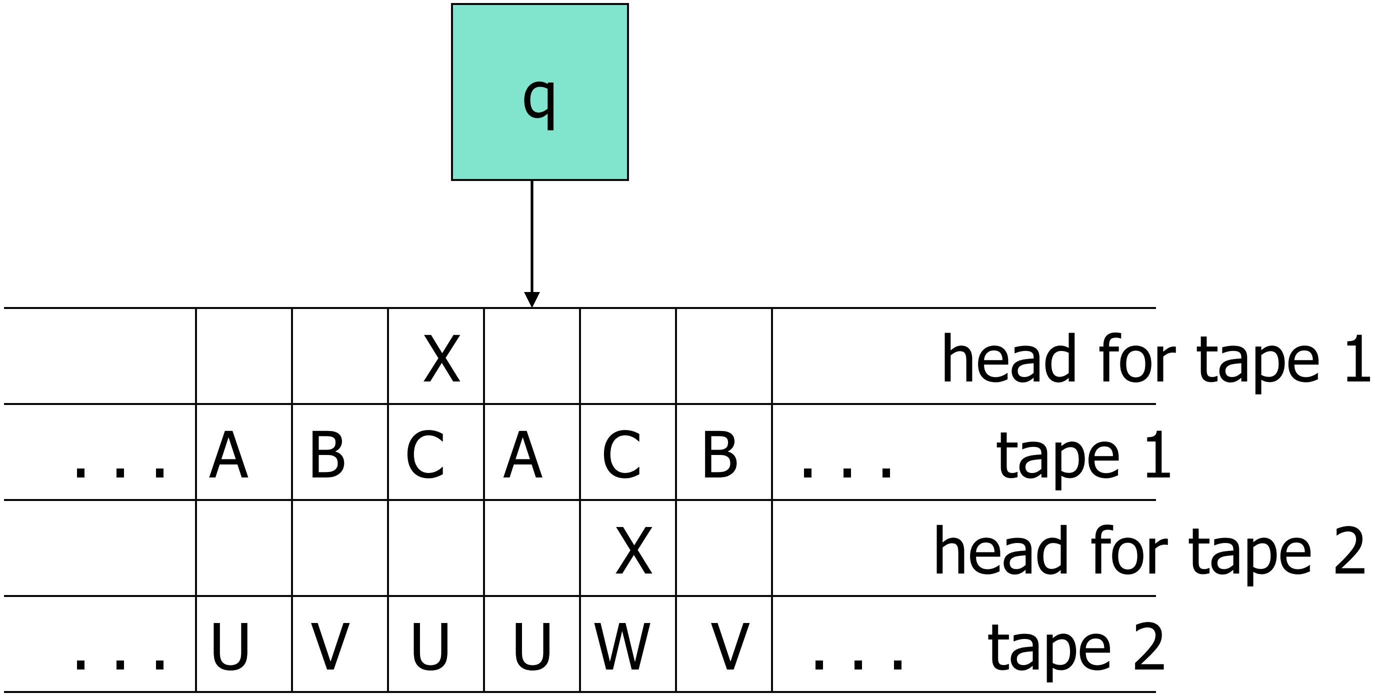
Multitape Turing Machines

- Allow a TM to have k tapes for any fixed k.
- Move of the TM depends on the state and the symbols under the head for each tape.
- In one move, the TM can change state, write symbols under each head, and move each head independently.

Simulating k Tapes by One

- Use 2k tracks.
- Each tape of the k-tape machine is represented by a track.
- The head position for each track is represented by a mark on an additional track.

Picture of Multitape Simulation



			X					head for tape 1
. . .	A	B	C	A	C	B	. . .	tape 1
					X			head for tape 2
. . .	U	V	U	U	W	V	. . .	tape 2

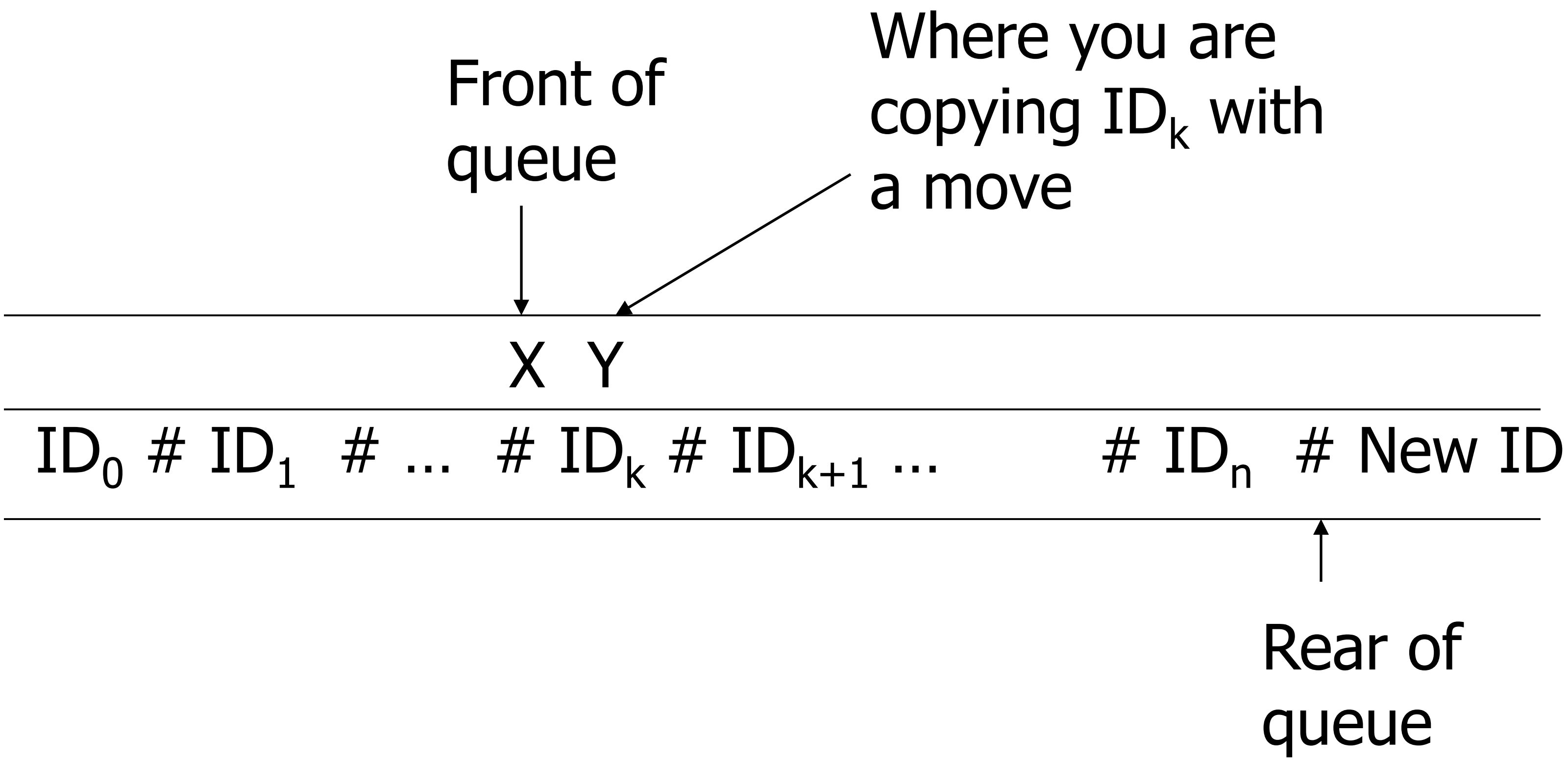
Nondeterministic TM's

- Allow the TM to have a choice of move at each step.
  - Each choice is a state-symbol-direction triple, as for the deterministic TM.
- The TM accepts its input if any sequence of choices leads to an accepting state.

Simulating a NTM by a DTM

- The DTM maintains on its tape a queue of ID's of the NTM.
- A second track is used to mark certain positions:
  1. A mark for the ID at the head of the queue.
  2. A mark to help copy the ID at the head and make a one-move change.

Picture of the DTM Tape



Front of queue

Where you are copying ID_k with a move

X Y

ID_0 # ID_1 # … # ID_k # ID_{k+1} … # ID_n # New ID

Rear of queue

Operation of the Simulating DTM

- The DTM finds the ID at the current front of the queue.
- It looks for the state in that ID so it can determine the moves permitted from that ID.
- If there are m possible moves, it creates m new ID's, one for each move, at the rear of the queue.

Operation of the DTM – (2)

- The m new ID's are created one at a time.
- After all are created, the marker for the front of the queue is moved one ID toward the rear of the queue.
- However, if a created ID has an accepting state, the DTM instead accepts and halts.

Why the NTM -> DTM Construction Works

- There is an upper bound, say k, on the number of choices of move of the NTM for any state/symbol combination.
- Thus, any ID reachable from the initial ID by n moves of the NTM will be constructed by the DTM after constructing at most $(k^{n+1}-k)/(k-1)$ID's.

$\uparrow$

Sum of $k+k^2+\ldots+k^n$

Why? – (2)

- If the NTM accepts, it does so in some sequence of n choices of move.
- Thus the ID with an accepting state will be constructed by the DTM in some large number of its own moves.
- If the NTM does not accept, there is no way for the DTM to accept.

Taking Advantage of Extensions

- We now have a really good situation.
- When we discuss construction of particular TM's that take other TM's as input, we can assume the input TM is as simple as possible.
  - E.g., one, semi-infinite tape, deterministic.
- But the simulating TM can have many tapes, be nondeterministic, etc.

Simulating a Name-Value Store by a TM

- The TM uses one of several tapes to hold an arbitrarily large sequence of name-value pairs in the format #name*value#…
- Mark, using a second track, the left end of the sequence.
- A second tape can hold a name whose value we want to look up.

Lookup

- Starting at the left end of the store, compare the lookup name with each name in the store.
- When we find a match, take what follows between the * and the next # as the value.

Insertion

- Suppose we want to insert name-value pair (n, v), or replace the current value associated with name n by v.
- Perform lookup for name n.
- If not found, add n*v# at the end of the store.

Insertion – (2)

- If we find #n*v’#, we need to replace v’ by v.
- If v is shorter than v’, you can leave blanks to fill out the replacement.
- But if v is longer than v’, you need to make room.

Insertion – (3)

- Use a third tape to copy everything from the first tape to the right of v′.
- Mark the position of the * to the left of v′ before you do.
- On the first tape, write v just to the left of that star.
- Copy from the third tape to the first, leaving enough room for v.

Picture of Shifting Right

Tape 1

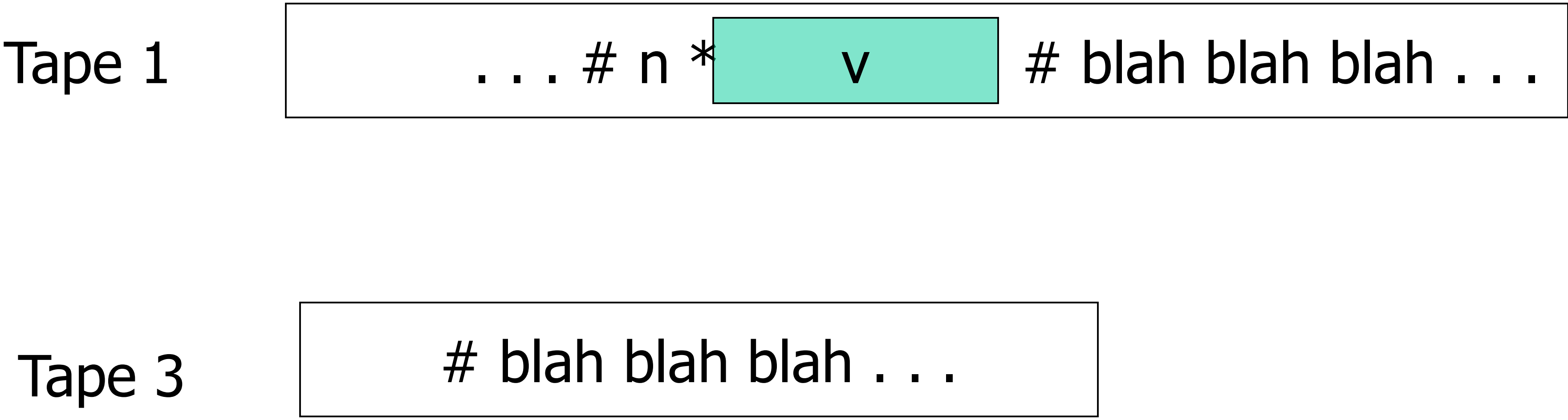
. . . # n * v’ # blah blah blah . . .

v

Tape 3

blah blah blah . . .

Picture of Shifting Right



Tape 1 . . . # n * v # blah blah blah . . .

Tape 3 # blah blah blah . . .

Closure Properties of Recursive and RE Languages

- Both closed under union, concatenation, star, reversal, intersection, inverse homomorphism.
- Recursive closed under difference, complementation.
- RE closed under homomorphism.

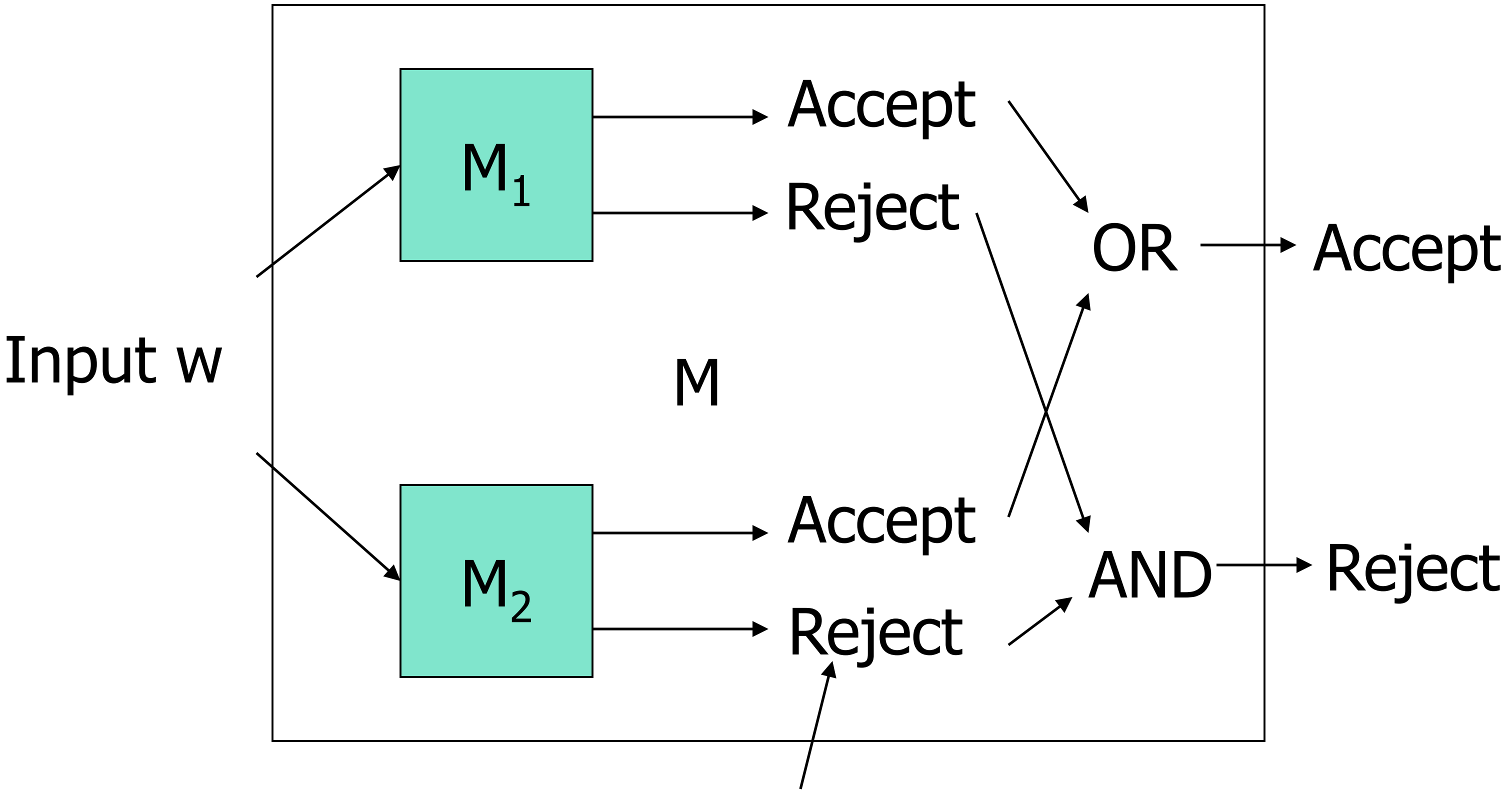
Union

- Let L_1 = $L(M_1)$ and L_2 = $L(M_2)$.
- Assume M_1 and M_2 are single-semi-infinite-tape TM's.
- Construct 2-tape TM M to copy its input onto the second tape and simulate the two TM's M_1 and M_2 each on one of the two tapes, "in parallel."

Union – (2)

- Recursive languages: If M_1 and M_2 are both algorithms, then M will always halt in both simulations.
- RE languages: accept if either accepts, but you may find both TM's run forever without halting or accepting.

Picture of Union/Recursive



Input w

M

M$_1$

M$_2$

Accept

Reject

Accept

Reject

OR

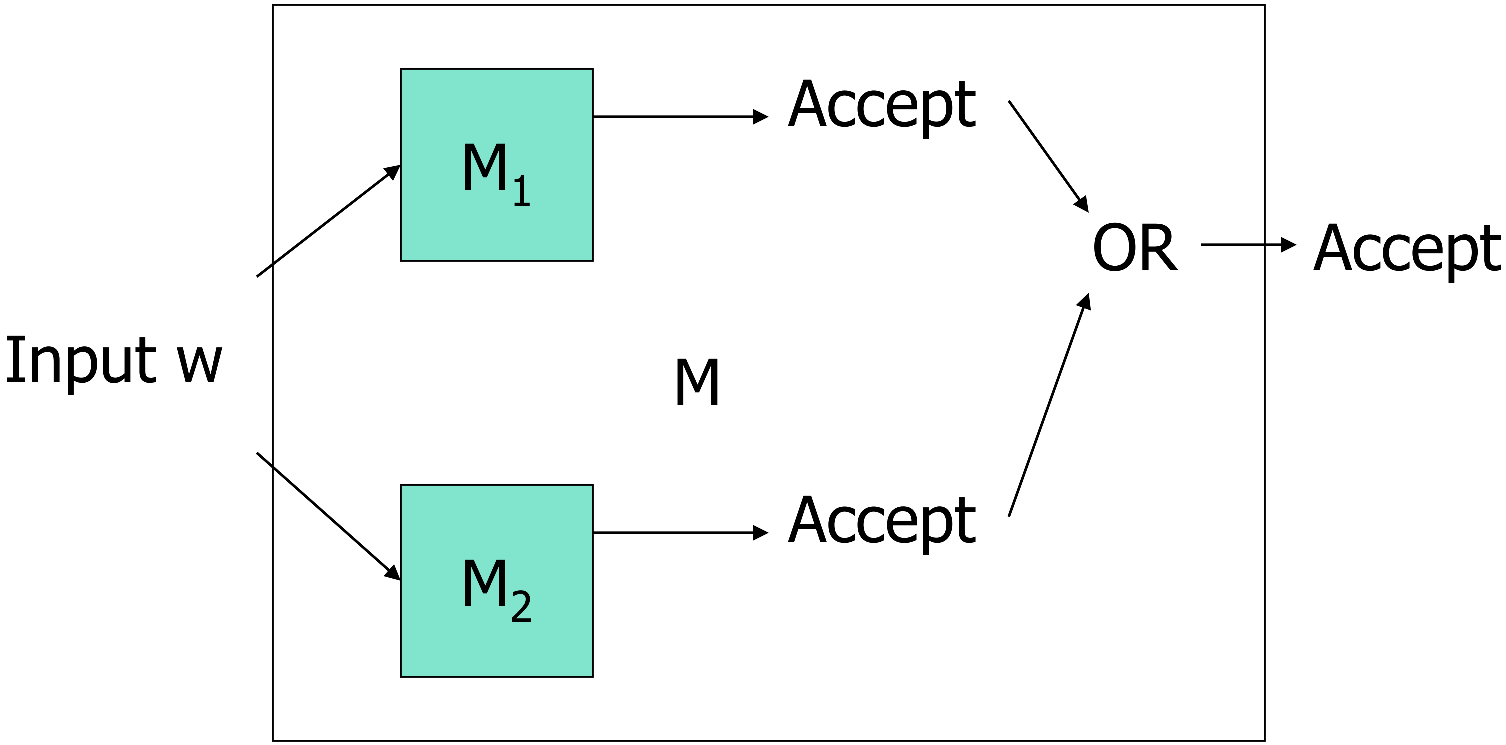
AND

Accept

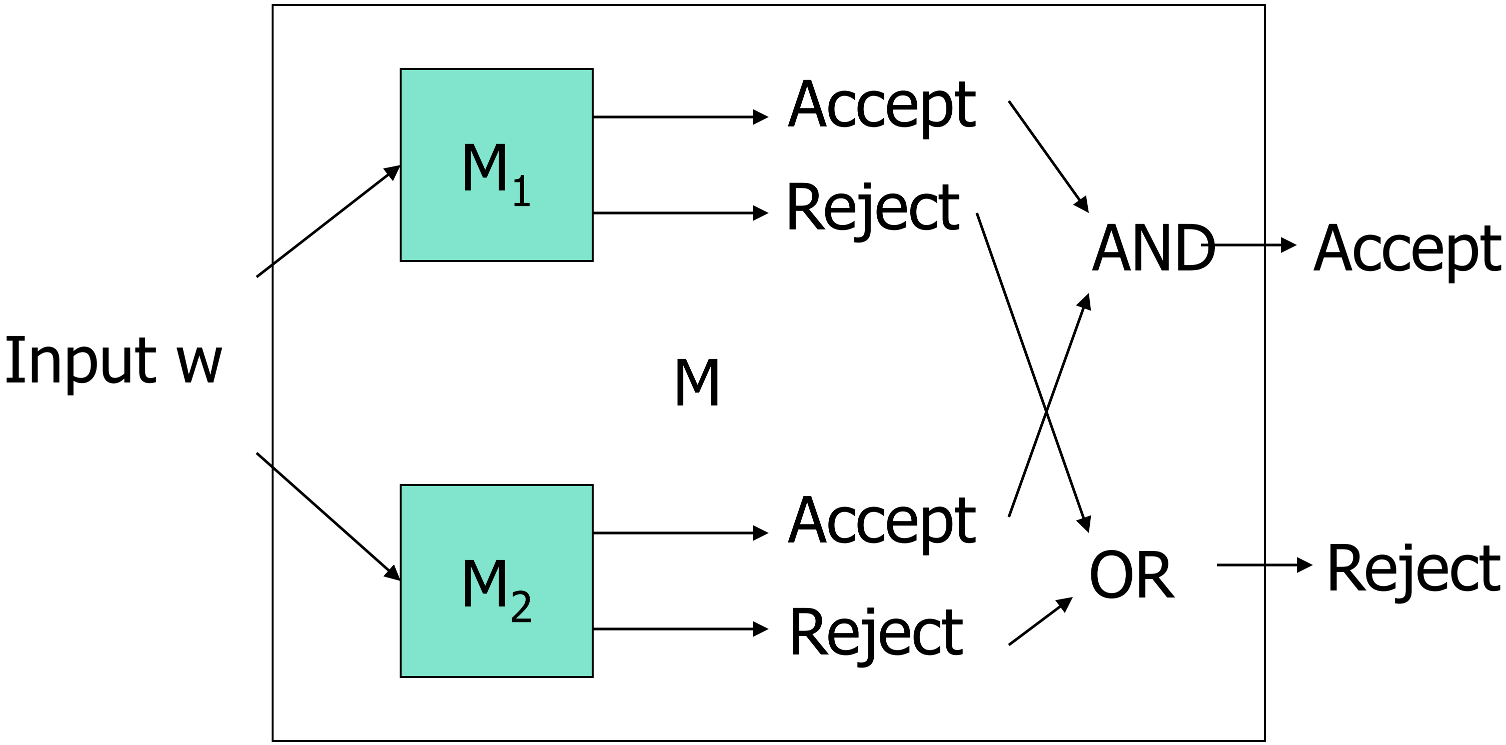
Reject

Remember: = "halt without accepting

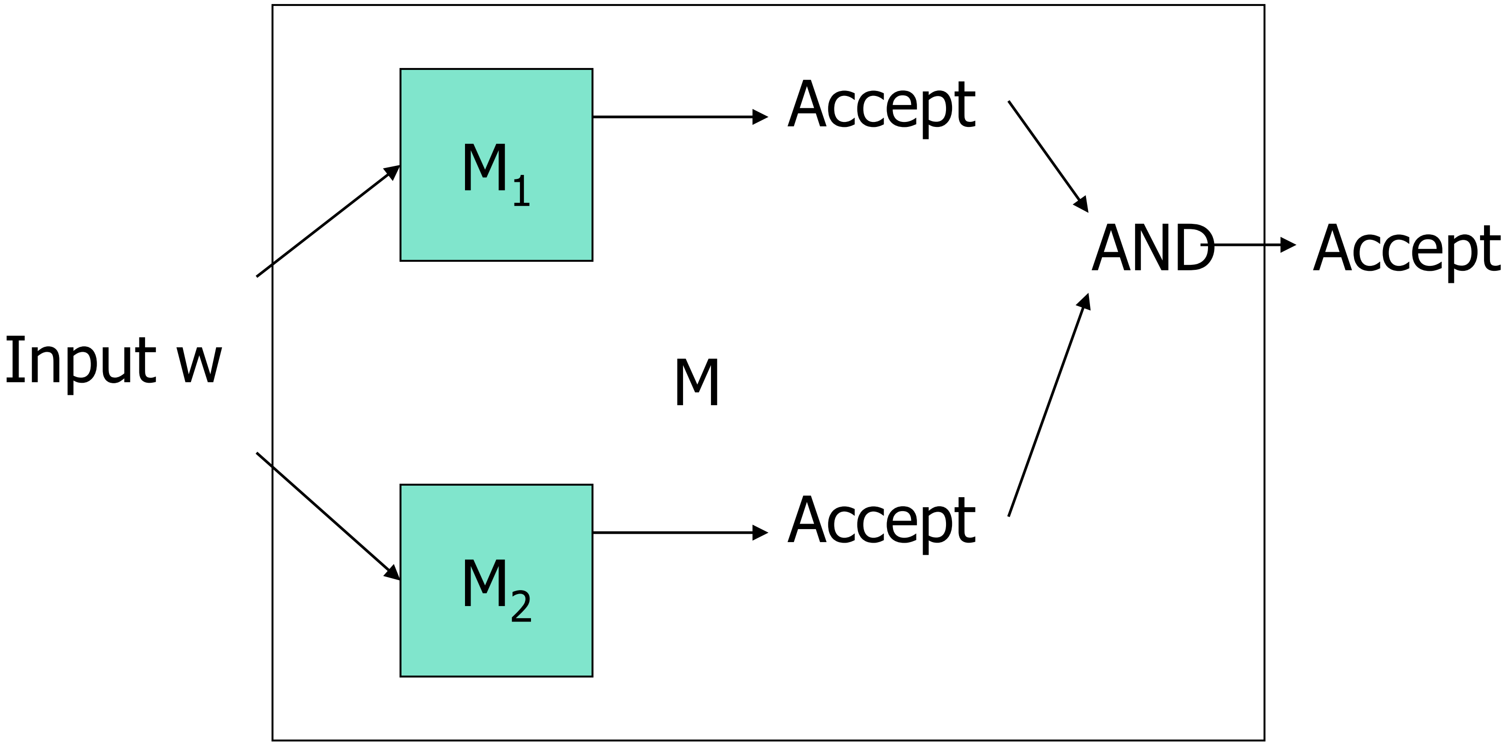
Picture of Union/RE



Intersection/Recursive – Same Idea



Intersection/RE



Input w

M_1

Accept

M_2

Accept

M

AND

Accept

Difference, Complement

- Recursive languages: both TM's will eventually halt.
- Accept if M_1 accepts and M_2 does not.
  - Corollary: Recursive languages are closed under complementation.
- RE Languages: can't do it; M_2 may never halt, so you can't be sure input is in the difference.

Concatenation/RE

- Let L_1 = $L(M_1)$ and L_2 = $L(M_2)$.
- Assume M_1 and M_2 are single-semi-infinite-tape TM's.
- Construct 2-tape Nondeterministic TM M:

1. Guess a break in input w = xy.
2. Move y to second tape.
3. Simulate M_1 on x, M_2 on y.
4. Accept if both accept.

Concatenation/Recursive

- Can’t use a NTM.
- Systematically try each break w = xy.
- M_1 and M_2 will eventually halt for each break.
- Accept if both accept for any one break.
- Reject if all breaks tried and none lead to acceptance.

Star

- Same ideas work for each case.
- RE: guess many breaks, accept if M_1 accepts each piece.
- Recursive: systematically try all ways to break input into some number of pieces.

Reversal

- Start by reversing the input.
- Then simulate TM for L to accept w if and only w^R is in L.
- Works for either Recursive or RE languages.

Inverse Homomorphism

- Apply h to input w.
- Simulate TM for L on h(w).
- Accept w iff h(w) is in L.
- Works for Recursive or RE.

Homomorphism/RE

- Let L = L(M_1).
- Design NTM M to take input w and guess an x such that h(x) = w.
- M accepts whenever M_1 accepts x.
- Note: won't work for Recursive languages.