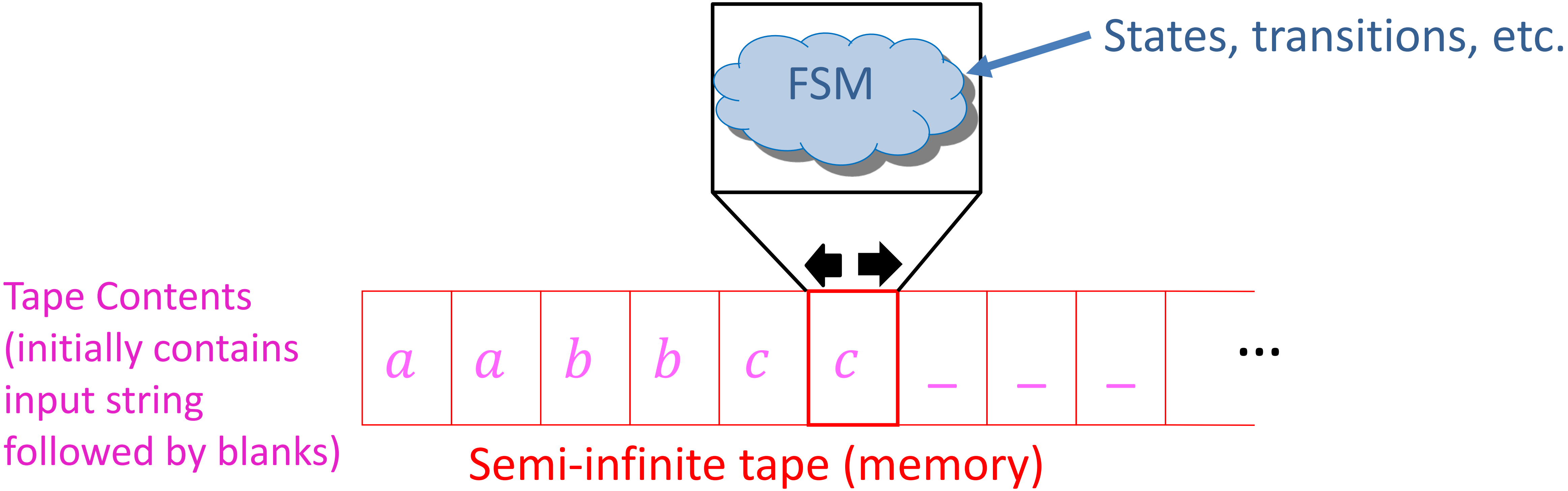
CS3102 Theory of Computation

Turing Machine



Operation: transitions outgoing from each state match on current character on the tape, when transitioning you can overwrite that character and move which cell you're reading

Turing Machines

- NFA/DFA:
  - Finite number of states,
  - read-once input,
  - transition using input character and state
- PDA:
  - Finite number of states,
  - read-once input,
  - stack (memory)
  - Transition using input character, state, and stack
  - Can push to the stack on transition

- Turing Machine:
  - Finite number of states,
  - Semi-infinite tape (memory)
  - Transition using state, and “current symbol” on tape
  - Can overwrite current symbol, move left/right on tape

Configurations

- DFA:
  - Which state is currently active
  - At start: Start state is active
- PDA:
  - Which state is currently active
  - What’s in the stack
  - At start: start state is active, stack is empty
- Turing Machine
  - Which state is currently active
  - What’s on the tape
  - Where the read head is on the tape
  - At start: start state is active, input is is on the tape (with blanks after), read head is at the first cell

Transition behavior

- DFA:
  - Input: State and input symbol
  - Result: state
- PDA:
  - Input: State, input symbol, symbol popped from the stack
  - Result: state, symbol pushed to the stack
- Turing Machine:
  - Input: State, symbol at current location in the tape
  - Result: State, symbol writtent to current location in the tape, current location moved 1 to the left/right

Acceptance Condition

- DFA
  - Input has all been read
  - We’re in a final/accepting state
- PDA
  - Input has all been read
  - We’re in a final/accepting state
  - The stack is empty
- TM
  - We’re in the accepting state

Rejection Condition

- DFA
  - Input has all been read
  - We’re not in a final/accepting state
- PDA
  - Input has all been read
  - We’re not in a final/accepting state
  - The stack isn’t empty
- TM
  - We’re in the rejecting state

Some Turing Machines never accept/reject

```
while(true){
   twiddle(thumbs);
}
```

- In this case they run forever
- 3 “reporting” behaviors
  - Accept and halt
  - Reject and halt
  - Run forever (implicit reject)
- This is necessary for computation

```
while(x != 1){
   if(x%2 == 0){
      x = x / 2;
   }
   else{
      x = 3x+1;
   }
}
Return true;
```

Running forever

- Is it a bad thing?

Programs we want to halt:

GCC

Your 2150 Homework

Programs we want to run forever:

Turing Machine Outcomes

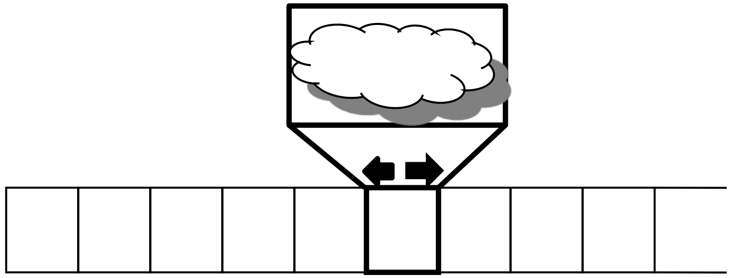
1. Running TM M on input w eventually leads to A
   - Result: Accept
2. Running TM M on input w eventually leads to R
   - Result: Reject
3. Running TM M on input w runs forever (never terminates).
   - Result: Reject

Recognizing Vs. Deciding

- **Turing-recognizable**: A language *L* is “Turing-recognizable” if there exists a TM *M* such that for all strings *w*:
  - If $w \in L$ eventually M enters A (the accept state)
  - If $w \notin L$ either *M* enters R *(the reject state)* ***or*** *M* never terminates
  - Intutively: You can write a program that always “yes” to the right answers
- **Turing-decidable**: A language *L* is “decidable” if there exists a TM *M* such that for all strings *w*:
  - If $w \in L$ eventually M enters A (the accept state)
  - If $w \notin L$ either *M* enters R *(the reject state)*
  - Intuitively: You can write a program that always answer correctly (yes or no)

Decidable

$w \rightarrow$
Input



$\Rightarrow$

✓ Accept & halt

✗ Reject & halt

Never runs forever

A language is decidable iff it is exactly the set of strings accepted by some always-halting TM.

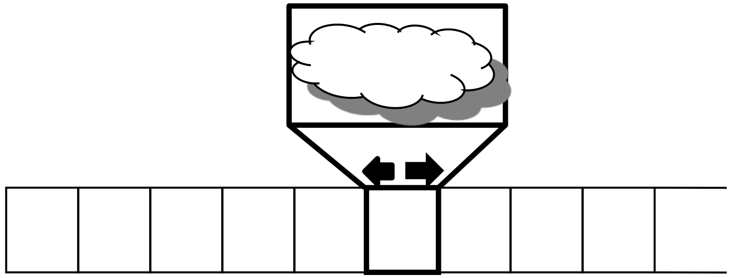
$w \in \Sigma^*$	a	b	aa	ab	ba	bb	aaa	aab	aba	abb	baa	bab	bba	bbb	aaaa	…
$M(w) \Rightarrow$	✓	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	…
$L(M) =$	$\{\ a,$		$aa,$				$aaa,$								$aaaa$	$\ldots\}$

M must always halt on every input.

Recognizable

$w \rightarrow$

Input



$\Rightarrow$ ✓ Accept & halt × Reject & halt $\equiv$ ∞ Run forever

A language is Turing-recognizable iff it is exactly the set of strings accepted by some Turing machine.

$w \in \Sigma^*$	a	b	aa	ab	ba	bb	aaa	aab	aba	abb	baa	bab	bba	bbb	$aaaa$	$\ldots$
$M(w) \Rightarrow$	✓	×	✓	∞	×	∞	✓	∞	∞	×	×	×	∞	×	✓	$\ldots$
$L(M) =$	$\{\ a,$		$aa,$				$aaa,$								$aaaa$	$\ldots\}$

M can run forever on an input, which is implicitly a reject (since it is not an accept).

Computability

- Generally: Computable = Decideable

An Undecidable Problem/Language

- Acceptance Problem
- Given a Turing Machine description M (e.g. a program, states+transitions, etc.) and a string w, does M accept the input w?
- $A_{TM}(M, w) = \{\langle M, w\rangle | M(w) \text{ accepts}\}$

Acceptance problem is undecidable

- Assume toward reaching a contradiction that M_{acc} is a Decider for A_{TM}.
- Consider a new Turing Machine M' which receives as input a turing machine description M.
- If $\langle M, M \rangle \in A_{TM}$ then M' will reject, else M' accepts

Pseudocode for M'

```
public static boolean mPrime(String m){
    return !accept(m,m);
}
```

What does **mPrime(source(mPrime))** return?

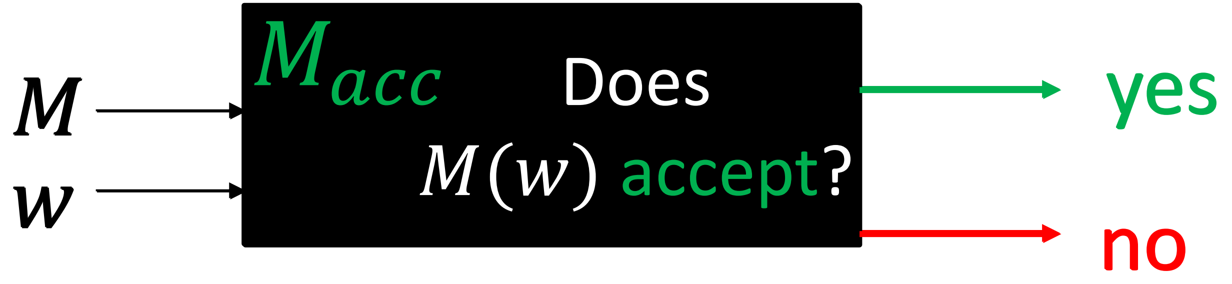
Acceptance problem is undecidable

- Assume toward reaching a contradiction that M_{acc} is a Decider for A_{TM}.
- Consider a new Turing Machine M' which receives as input a turing machine description M.
- If $\langle M, M \rangle \in A_{TM}$ then M' will reject, else M' accepts
- Consider $M'(M')$
  - If it accepts, then by definition $M'(M')$ will reject
  - If it rejects, then by definition $M'(M')$ with accept
  - Contradiction!
- Conclusion: M_{acc} cannot exist

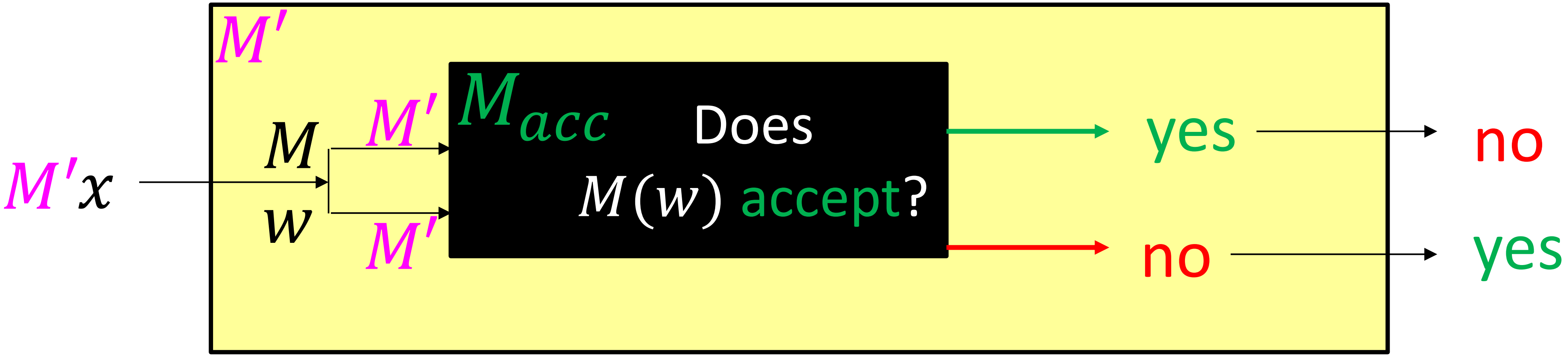
Theorem: the acceptance problem (A_{TM}) is not decidable

Proof: Assume some decider M_{acc} solves A_{TM}

always stops with the correct answer for any M and w



Use M_{acc}, construct a TM M':



$M'(M')$ accepts $\rightarrow$ $M'(M')$ does not accept

$M'(M')$ does not accept $\rightarrow$ $M'(M')$ accepts

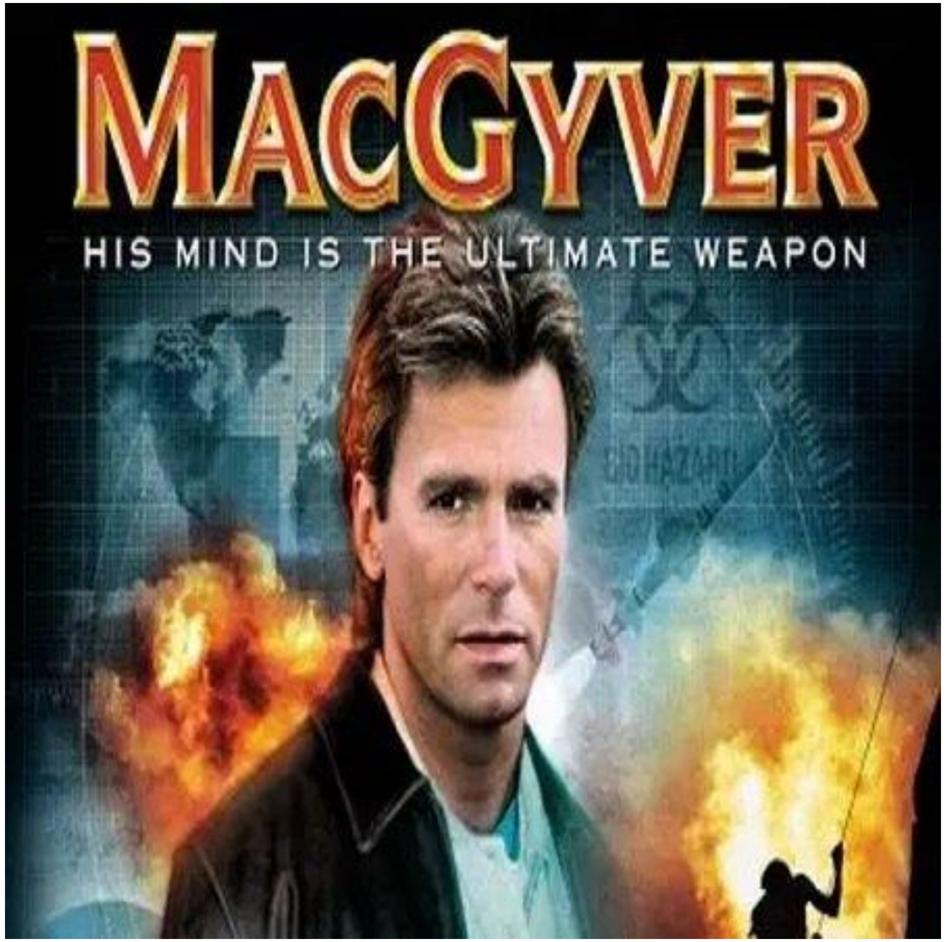
Contradiction!

$\rightarrow$ M_{acc} cannot exist! (at least as an algorithm / program / TM)

Proof by Reduction

Shows how two different problems relate to each other

MOVIE TIME!



Reduction Proofs

Opening a
do

A reduces to B

Lighting a
fir

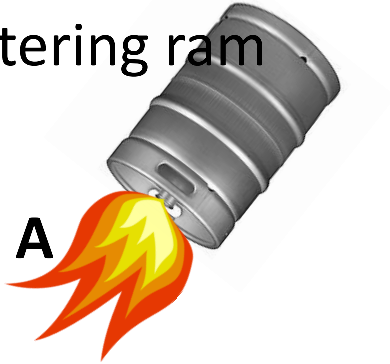
Alcohol, wood,
ma s

Y that can solve **B**

can be used to make

X that can solve **A**

Keg cannon
battering ram

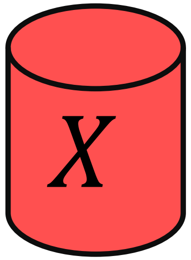


A is not a harder problem than B

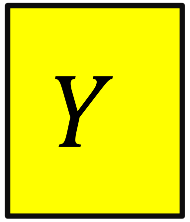
$$A \leq B$$

The name “reduces” is confusing: it is in the *opposite* direction of the making

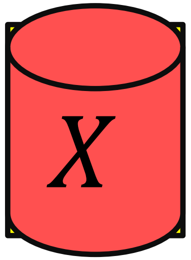
Proof of Impossibility by Reduction



1. X isn't possible
(e.g., X = some way to open the door)



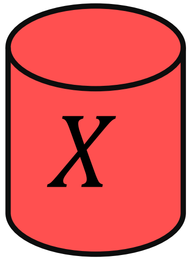
2. Assume Y is possible
(Y = some way to light a fire)



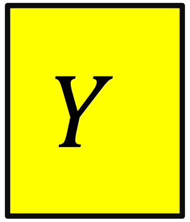
3. Show how to use Y to perform X.

4. X isn't possible, but Y could be used to perform X
conclusion: Y must not be possible either

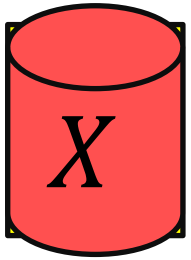
Proof of Impossibility by Reduction



1. X does not exist.
(e.g., X = some TM that decides A_{TM})



2. Assume Y exists.
(Y = some TM that decides B)



3. Show how to use Y to perform X.

4. X doesn't exist, but Y could be used to make X
conclusion: Y must not exist either

Converse?

A reduces to B

Y

can be used to make

X

that can solve **B**
(Lighting a fire)

that can solve **A**
(opening a door)

A is not harder to solve than B

$$A \leq B$$

Does this mean B is equally as hard as A? $A = B$

No!
Solving Y is only one way to solve X
There may be an easier way

Common Reduction Traps

- Be careful: the direction matters a great deal
  - Using a solver for B to solve A shows A is not harder than B

$\underline{\;A\;}$ Reduces to $\underline{\;B\;}$

- The transformation must use only things you ***can do***:
  - Otherwise it may be that B exists, but some other step doesn't!

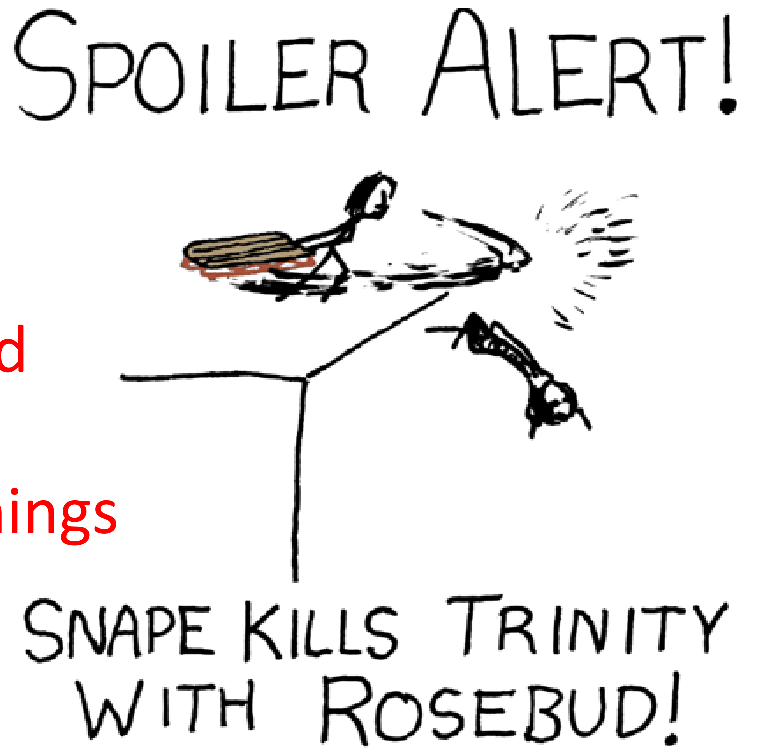
{ What does can do mean? }

What "Can Do" Means

- Tools used in a reduction are limited by what you are proving
- Undecidability:
  - You are proving something about all TMs:
  - The transformation "can do" things a terminating TM "can do"

Spoiler alert!

- Complexity:
  - You are proving something about time required:
  - The time it takes to do the transformation is limited
- Combinatorics:
  - You are proving something about the number of things
  - The transformation must be bijective



The Halting Language

$A_{TM} = \{ < M, w > \ | M$ is a TM description and M accepts input w $\}$
All machine description, input pairs in which the machine accepts that input

$HALT_{TM} = \{ < M, w > \ | M$ is a TM description and M halts on input w $\}$
All machine description, input pairs in which the machine halts on that input

Every $< M, w >$ which halts at all belongs to $HALT_{TM}$
$< M, w >$ belongs to A_{TM} if it both halts and accepts

$HALT_{TM}$ is Undecidable

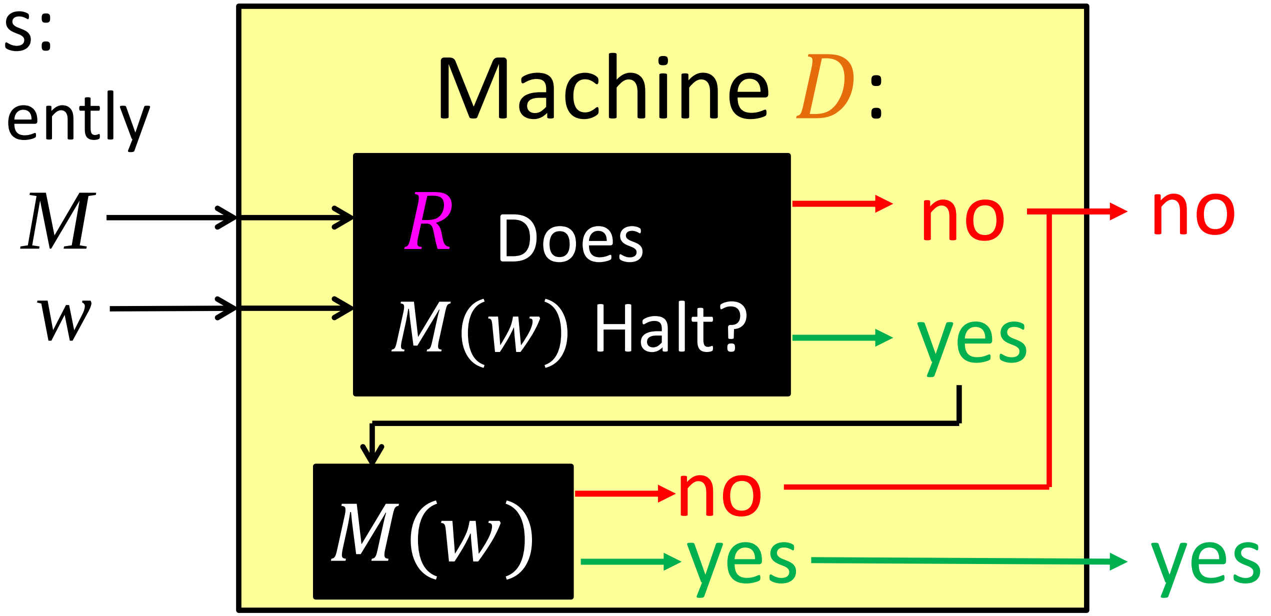
- $HALT_{TM} = \{< M, w > | M$ is a TM description and M halts on input $w\}$
  - All machine description input pairs in which the machine halts on input
- To show $HALT_{TM}$ is undecidable show A_{TM} isn't harder than $HALT_{TM}$
- Want to use a solver for $\underline{\quad HALT_{TM} \quad}$ to build a solver for $\underline{\quad A_{TM} \quad\quad}$
- $\underline{\quad A_{TM} \quad}$ reduces to $\underline{\quad HALT_{TM} \quad}$

A_{TM}

$HALT_{TM}$

Deciding A_{TM} with HALT_{TM}

- Assume $HALT_{TM}$ is decidable.
- Then some TM R can decide $HALT_{TM}$.
- We can use R to build a machine D that decides A_{TM}:
  - Call R on $< M , w >$
  - If R rejects, it means M doesn't halt; **reject**
  - If R accepts, it means M halts:
    - Call M on w, respond equivalently



Any TM that decides $HALT_{TM}$ could be used to build a TM that decides A_{TM} (which is impossible) thus no TM exists that can decide $HALT_{TM}$