Undecidability

COMP 455 – 002, Spring 2019

Essential Definitions

Specific definitions of "problems" and "instances" are important in these slides:

- **Problem**: A yes/no question.
  - Example: Is G an ambiguous CFG?
- **Instance**: A list of arguments, one per parameter.
  - Example: a particular CFG, *e.g.* $S \rightarrow aSb \mid \varepsilon$.

We can encode *instances* of a *problem* as strings over some finite alphabet.

Equivalence of "Problem" and "Language"

- Consider the CFG $G = (V, T, P, S)$. We will encode this entire CFG as a string of 0s and 1s.
- Let $X_i, 1 \leq i \leq V$ denote the i^{th} variable in V.
- Let $X_{|V|+i}$ denote the i^{th} terminal in T.
- If ε appears in a production, then denote it as $X_{|V|+|T|+1}$.
- Encode (,), {, }, ,, and → by 1, 10, 100, 10^3, 10^4, and 10^5.
- Encode X_i as 10^{i+5}.

Represents a comma

With this definition, we can view any sequence of 0s and 1s as encoding some CFG. If the sequence is not of the right form, define it to mean a CFG with no productions.

Encoding a CFG as Binary: Example

- Let's encode the CFG $(\{S, A\}, \{0\}, \{S \rightarrow A, A \rightarrow 0\}, S)$ using the scheme from before.
- S is variable 1, denoted X_1, and A is X_2.
- 0 is the only terminal, denoted X_3.
- Now, just replace the (), {}, commas, $\rightarrow$'s, variables, and terminals with their "binary" representation:

1	100	1000000	10000	10000000	1000	10000	100	…
(	{	X_1	,	X_2	}	,	{	

"Problems" and "Languages", continued

- Now that we can represent a CFG as a string, we can treat yes/no questions about CFGs as *languages*.
- This same notion apples to any problem—if you can represent instances of your problem as strings, then the question of whether an algorithm exists to solve the problem can be posed as a question about whether a language is recursive.
- Example:
  - Let $L_{AMB} = \{w \mid \text{the CFG encoded by } w \text{ is ambiguous}\}$
  - The question "Does an algorithm exist to solve the ambiguity problem?" is equivalent to "Is L_{AMB} recursive?"

Recall: *Algorithm* = TM that always halts

Yes/No Questions

There are two main reasons we focus on yes/no questions:

- It fits our notion of a recursive language.
  - A TM that halts and accepts answers “yes”.
  - A TM that halts and rejects answers “no”.
- Many more general problems have an equivalent yes/no version, *i.e.*, the general version has an algorithm if and only if the yes/no version does.

Yes/No vs. "General" Algorithms

Say we have two problems:

- AMB: Takes a CFG and outputs "yes" if the CFG is ambiguous, otherwise outputs "no".
  - This is the yes/no version (in case it wasn't obvious)
- $FIND$: Takes a CFG and outputs some string w with at least two parse trees if the CFG is ambiguous, otherwise it outputs "no".
  - This is the "general" version.

Yes/No vs. "General" Algorithms

We will show that we have an algorithm for $FIND$ if and only if we have an algorithm for AMB.

First, we can easily solve AMB if we have an algorithm for $FIND$:

- Run the algorithm for $FIND$
- If it outputs any w then output "yes", otherwise output "no".

Yes/No vs. "General" Algorithms

Second, we have an algorithm for $FIND$ if we have one for AMB:

- Run the algorithm for AMB
- If it outputs "no", then $FIND$ should output no
- If it outputs yes, just systematically start checking every possible string for multiple parse trees until you find one with two parse trees.
  - You'll find one eventually, because you already know that the grammar is ambiguous.
  - Remember, an algorithm only needs to eventually finish—it doesn't need to finish quickly!

Definition of Decidability

A problem is **decidable** if its language is recursive, and **undecidable** otherwise.

Important Note: Finite Problems

- *Any* problem with a *finite* number of instances is decidable.

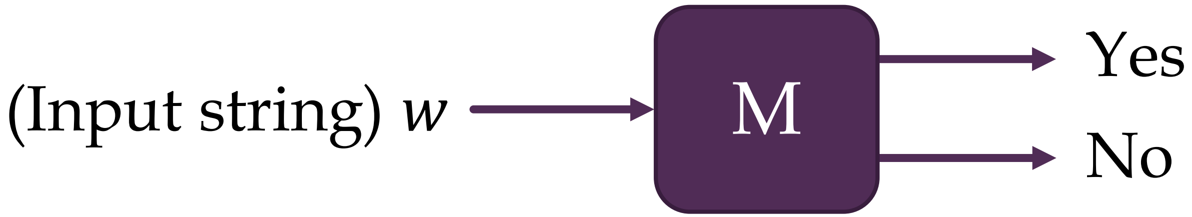
Example: Is there intelligent life on other planets?

- This problem only has a single instance, so it's decidable.
  - There exists a TM that accepts any input, and one that rejects any input. *One of these two TMs correctly answers this question*.

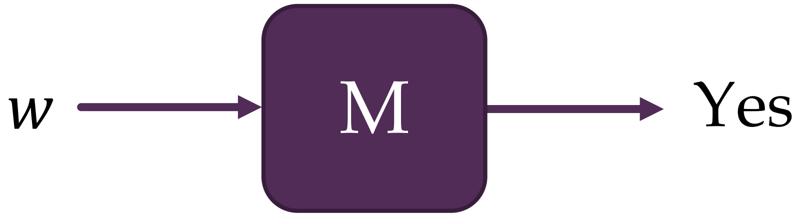
Recursive and RE Languages

We will simplify many of our proofs using pictures.

An algorithm (a TM that always halts) is represented like this:



An arbitrary TM (that may not halt) is represented like this:

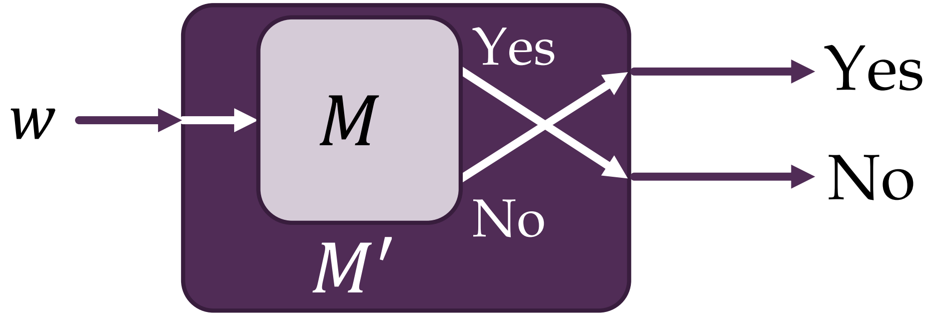


Complement of Recursive Languages

Theorem 9.3: The complement of a recursive language is recursive.

Proof:

Let L be a recursive language accepted by TM M. Construct M' to accept $\bar{L}$ as follows:

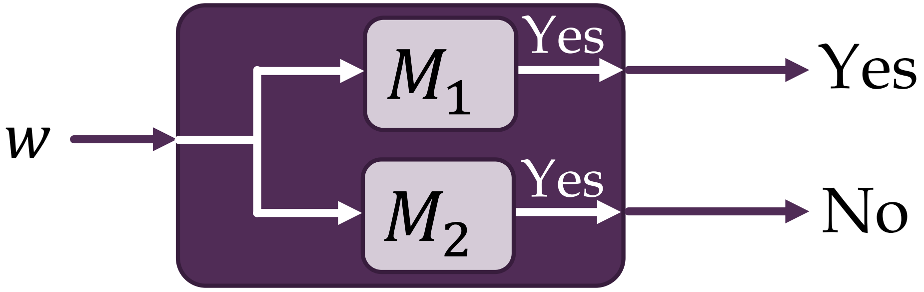


RE Languages and Complements

Theorem 9.4: If both a language and its complement are RE, then the language is recursive.

Proof:

Let $L = L(M_1)$ and $\bar{L} = L(M_2)$, for TMs M_1 and M_2. Construct an algorithm for L as follows:



The Possible Language Categories

Theorem: For any language L, we have three cases:

1. Both L and $\overline{L}$ are recursive.
2. Neither L nor $\overline{L}$ are recursively enumerable.
3. One of L and $\overline{L}$ is RE but not recursive, and the other is not RE.

Proof:

Suppose either L or $\overline{L}$ is RE but not recursive (not Case 1). Then, by Theorem 9.3, neither are recursive. Also, by Theorem 9.4, one is not RE.

An Undecidable Problem

Consider the problem:

- Does a Turing Machine M accept input w?

We want to show that that this problem is undecidable.

We will show that this is undecidable, even if we restrict w to be over the alphabet $\{0, 1\}$.

However, our first goal will be to encode the parameter M (a Turing Machine) as a string.

Encoding TMs as Binary Strings

- Consider TM $M = (Q, \{0,1\}, \Gamma, \delta, q_1, B, \{q_2\})$, where $Q = \{q_1, q_2, \dots, q_r\}$.
  - We are assuming q_2 is always the lone final state.
- Denote symbols in Γ as $X_1, X_2, X_3, \dots$, where 0, 1, and B are denoted X_1, X_2, and X_3, respectively.
- Denote L and R as D_1 and D_2.
- Encode the move $\delta(q_i, X_j) = (q_k, X_l, D_m)$ as $0^i 1 0^j 1 0^k 1 0^l 1 0^m$.
- Encode M as $\mathrm{code}_1 11 \mathrm{code}_2 11 \dots 11 \mathrm{code}_\mathrm{n}$, where n is the number of moves in M, and code_i is the encoding for the i^{th} move.

We only need one final state due to our assumption that a TM will halt when entering it, so it won't have any outgoing transitions.

Encoding TMs as Binary Strings

Notes on this encoding scheme:

- A given TM may have many possible encodings.
- Many strings of bits aren't "legal" TM encodings.
  - Define any such string to encode a TM with no moves (which accepts no strings).
- With these assumptions, *any string of 0s and 1s corresponds to a TM*.

For example, any string with three or more "1"s in a row.

Encoding Instances of the Problem

Remember our ultimate goal is to show "Does a Turing Machine M accept input w?" is an undecidable problem.

- We will want to give an instance of this problem as an input to some other TM.
- We need to, therefore, have a way to encode *both* a TM M and an input w in a single string.
  - Since a valid TM can't have three or more 1s in a row, encode instances of this problem (M, w) as the binary representation of M, followed by three 1s, followed by the input string w.

Remember that w is also over the alphabet $\{0, 1\}$.

An Example Encoding

- Say $M = (\{q_1, q_2, q_3\}, \{0,1\}, \{0,1,B\}, \delta, q_1, B, \{q_2\})$.
  - $\delta(q_1, 1) = (q_3, 0, R) \quad = \quad 0100100010100$
  - $\delta(q_3, 0) = (q_1, 1, R) \quad = \quad 0001010100100$
  - $\delta(q_3, 1) = (q_2, 0, R) \quad = \quad 00010010010100$
  - $\delta(q_3, B) = (q_3, 1, L) \quad = \quad 0001000100010010$
- The encoding of M: 010010001010011000101010010 0110001001001010011000100010010
- Encoding of $(M, 1011)$: 010010001010011000101010 010011000100100101001100010001000100101111011

A Non-RE Language

We need one more definition to help with the following undecidability proof:

- Define **canonical order** for strings of 0s and 1s by sorting strings first by increasing size, and then sorting strings of the same size in increasing "numerical" order.
  - E.g. $\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 100, 101, \ldots$ are in canonical order.
- Let w_i be the i^{th} string of 0s and 1s in canonical order.
- Let M_j be the TM whose binary representation is w_j.

The Diagonal Language

Imagine an infinite table, where each column corresponds to an input string w_i and each row corresponds to a TM M_j.

If M_j eventually accepts w_i, then put a 1 in cell (i, j), otherwise put a 0 in cell (i, j):

Input strings $w_0, w_1, \ldots w_\infty$

Turing Machines $M_0, M_1, \ldots M_\infty$

	0	**1**	**2**	**3**	**4**	**…**
0	0	1	1	0	1	…
1	1	0	0	0	1	…
2	0	1	1	1	0	…
3	0	1	0	1	0	…
4	0	0	1	0	1	…
…	…	…	…	…	…	…

Remember it's an example: The depicted portion of this table should actually be all 0s because, in our encoding, M_0 through M_4 won't be valid TMs.

The Diagonal Language

The **Diagonal Language**, L_d, is the set of strings w_i such that the (i, i) entry in the table is 0.

(In other words, L_d contains all strings that, when interpreted as a TM, do *not* accept themselves as input.)

Input strings $w_0, w_1, \ldots w_\infty$

Turing Machines $M_0, M_1, \ldots M_\infty$

	0	1	2	3	4	...
0	0	1	1	0	1	...
1	1	0	0	0	1	...
2	0	1	1	1	0	...
3	0	1	0	1	0	...
4	0	0	1	0	1	...
...	...	...	...	...	...	...

L_d (if based on this example table) *would* contain w_0 and w_1, and *would not* contain w_2, w_3, and w_4.

Proving L_d is not RE

- We can call each row in this table the *characteristic vector* for that row's TM: it represents how the TM will behave given any possible input.
- If L_d *is* RE, then it would have a corresponding TM, meaning that its characteristic vector must appear as a row of this table.
  - However, it turns out that this is impossible!

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

TM M_4's characteristic vector (from this example table) is $[0, 0, 1, 0, 1, \ldots]$

Proving L_d is not RE

- No row of this table can possibly contain the characteristic vector for L_d. Why is this?
- Let's start by thinking about what the characteristic vector for L_d should be, using our example table:

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

- The characteristic vector for $\overline{L_d}$ is along the diagonal of the table: it contains a 1 if M_i *does* accept w_i.
  - ❖ This is $[0, 0, 1, 1, 1, \ldots]$ in the example.
- The characteristic vector for L_d must contain the opposite of this.
  - ❖ It would start $[1, 1, 0, 0, 0, \ldots]$

Proving L_d is not RE

- $\overline{L_d}$'s characteristic vector is $[0, 0, 1, 1, 1, \ldots]$ in the example.
- L_d's characteristic vector is the opposite: $[1, 1, 0, 0, 0, \ldots]$.
- Why can't this be in the table? Let's think about what happens if we try to find rows containing this vector.

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

$[1,1,0,0,0,\ldots]$ can't be on row 0; the first value must be different.

$[1,1,0,0,0,\ldots]$ can't be on row 1; the second value must be different.

$[1,1,0,0,0,\ldots]$ can't be on row 2; the third value must be different.

Every row in the table will differ in at least one position from L_d's $[1,1,0,0,0,\ldots]$!

Proving L_d is not RE

Theorem 9.2: L_d is not RE.

Proof: (We will state the proof from the previous slides in more formal terms.)

- Assume L_d is RE. That means $L_d = L(M_j)$ for some j.
  - If entry (j, j) is 0, then $w_j \notin L(M_j)$. But, $w_j \in L_d$. This contradicts the assumption that $L_d = L(M_j)$.
  - If entry (j, j) is 1, then $w_j \in L(M_j)$. But $w_j \notin L_d$. This also contradicts the assumption that $L_d = L(M_j)$.

The Universal Language

The **Universal Language** L_u:

$L_u = \{(M, w) \mid TM\ M\ accepts\ input\ w\}$.

- Given a TM M and input w (both encoded as 0s and 1s), we can use a TM accepting L_u to determine if M accepts w.
  - In other words, M accepts w if and only if $(M, w) \in L_u$.
- A TM that accepts the universal language is called a **universal TM**.

The Universal Language, L_u

Theorem: L_u is RE.

Proof:

We can construct a TM M_1 accepting L_u as follows:

- M_1 has three tapes:
  - Tape 1: The encoding of (M, w) as described before $(\mathrm{code}_1 11 \mathrm{code}_2 11 \ldots 111 \mathrm{code}_\mathrm{n} 111 w)$.
  - Tape 2: M's tape
  - Tape 3: M's state

Store state q_i as 0^i.

The Universal Language, L_u

Here's how M_1 behaves:

1. Check the format of the TM encoding on tape 1.
   If it's wrong, halt without accepting.
2. Copy the input w to tape 2 (M's tape)
3. Initialize tape 3 to $0BBB$ .... (Set M's start state to q_0).
4. After every step, halt and accept if tape 3 contains $00BBB$ ....
   - Recall that q_2 was the only accepting state in our binary TM encoding.
5. If head 2 points to symbol X_j and 0^i is on tape 3, scan Tape 1 for a move $0^i 1 0^j 1 0^k 1 0^l 1 0^m$.
   - If such a move is not found, halt without accepting.
   - If the move is found, put 0^k on tape 3, replace X_i with X_l on tape 2, and move tape head 2 in the direction D_m.

The Universal Language, L_u

Theorem 9.6: L_u is RE but not recursive.

Proof:

We've already shown L_u is RE (we constructed a TM for it), so we only need to show it's not recursive.

Since L_d is not RE, L_u is not recursive.

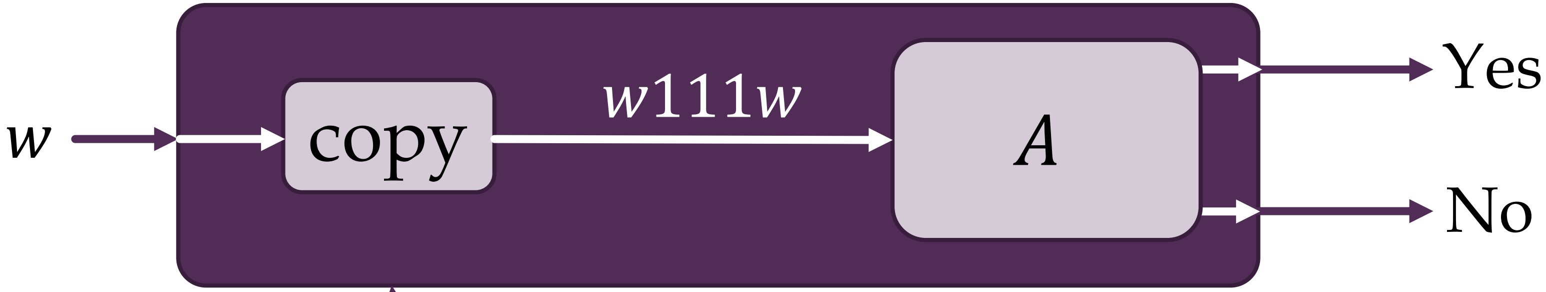
- L_u can be used to accept $\overline{L_d}$…
  - So, if L_u were recursive, we'd have an algorithm for $\overline{L_d}$…
  - But, by Theorem 9.3, if $\overline{L_d}$ is recursive, then L_d must also be recursive.

Theorem 9.3: The complement of a recursive language is recursive.

Using an Algorithm for L_u for $\overline{L_d}$

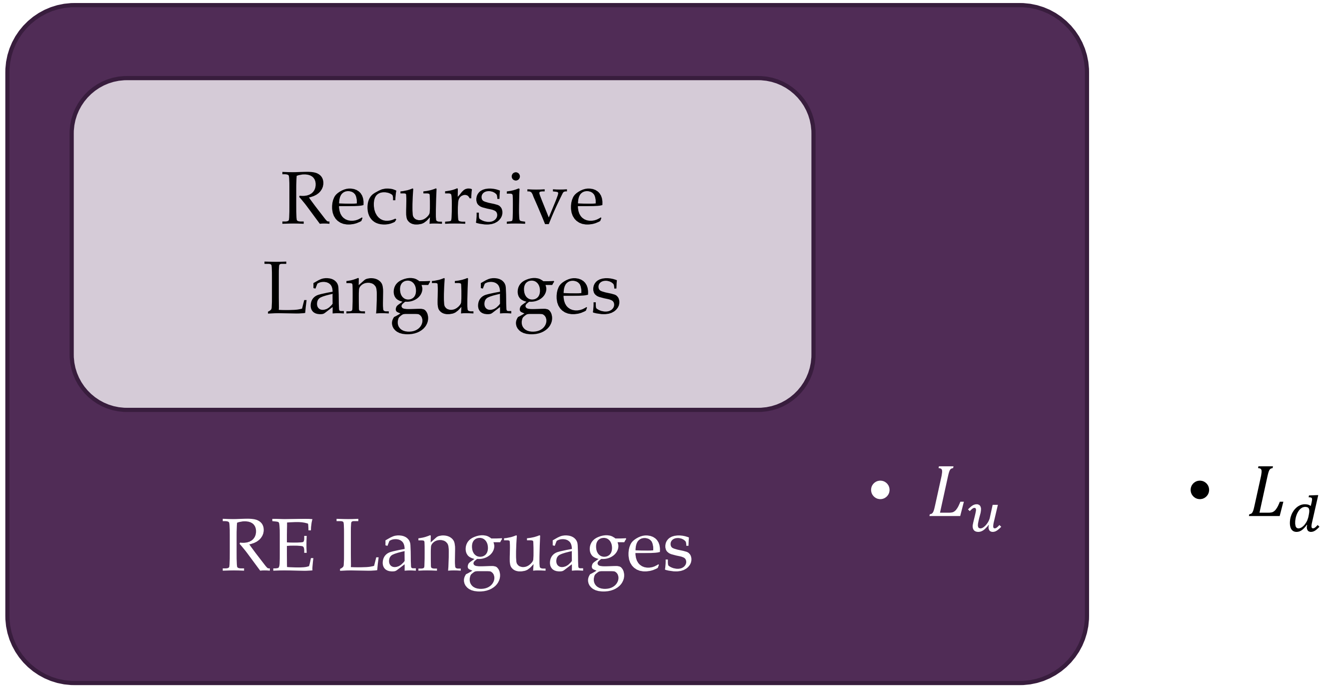
The previous slide claimed that if L_u were recursive, we could use it as an algorithm to accept $\overline{L_d}$, but didn't say how (it's fairly intuitive, though).

Say we have an algorithm A accepting L_u. Construct an algorithm accepting $\overline{L_d}$ from A as follows:



The "copy" routine here just converts a single input string into the input format A expects (a TM followed by its input).

Results So Far



Results So Far

- L_d is not RE.
  - And therefore, $\overline{L_d}$ is not recursive.
- L_u is RE but not recursive.
  - And therefore, $\overline{L_u}$ is not RE.

We will use these results to show that some other problems are undecidable.

Emptiness vs. Nonemptiness

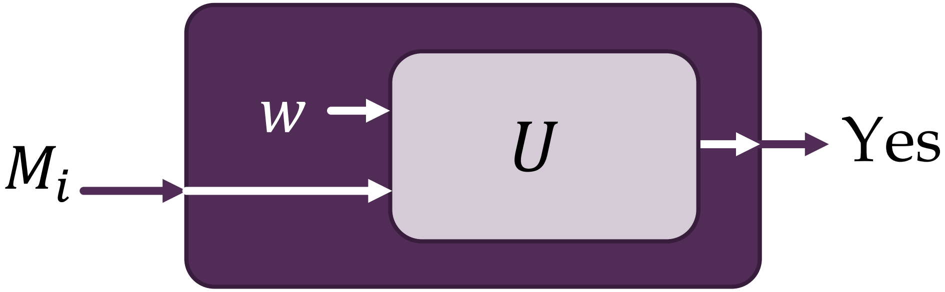
Problem: Is $L(M) \neq \emptyset$?

- This question takes a single TM and answers yes if the TM accepts any string, and no otherwise.
- We'll write "non-empty" language L_{ne} as follows:
  - $L_{ne} = \{M \mid L(M) \neq \emptyset\}$.
- The complement is $L_e = \{M \mid L(M) = \emptyset\}$.

The Non-Emptiness Language: L_{ne}

Theorem 9.8: L_{ne} is RE.

Proof: We will use L_u to accept L_{ne} as follows. Let U be a universal TM accepting L_u.



This TM will non-deterministically "guess" an input string w, feed U the input TM M_i with input w, and accept if U accepts.

The Non-Emptiness Language: L_{ne}

Theorem 9.9: L_{ne} is not recursive.

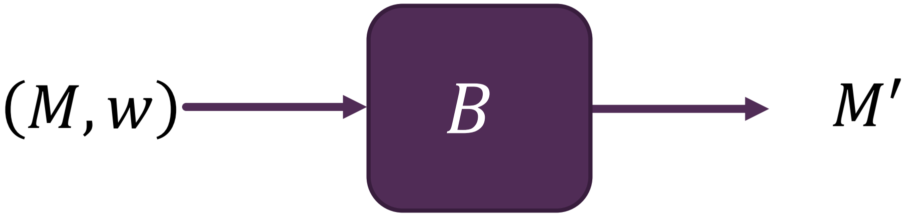
Proof:

Suppose L_{ne} is recursive.

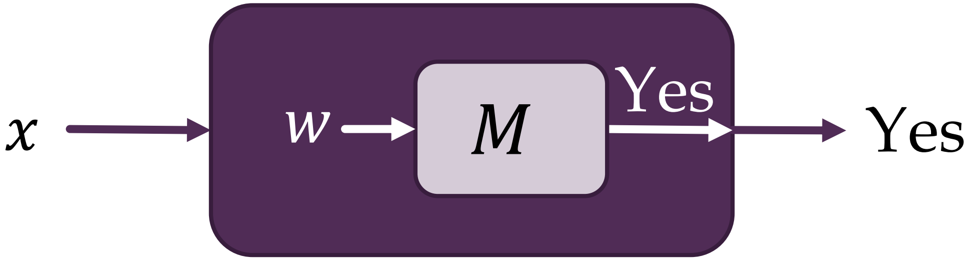
- Let A be an algorithm accepting L_{ne}.
- We will use algorithm A and another algorithm B to construct an algorithm for L_u (which we already proved doesn't exist).
  - The algorithm B is described next…

Proof that L_{ne} is not Recursive, continued

Think of the second algorithm, B, like this:



Where M' is the following TM:



M' is a TM that ignores whatever input it's provided and instead always carries out the same behavior that M did with input w.

B is similar to a "compiler": it takes a "source" TM and outputs a different TM.

Proof that L_{ne} is not Recursive, continued

This is how algorithm B works:

1. Scan the input (M, w) to find the input string w.
   - Let $w = a_1 a_2 \dots a_n$.
2. Create codes for the following TM moves:
   - $\delta(q_1, X) = (q_2, \$, R)$ For all X
   - $\delta(q_i, X) = (q_{i+1}, a_{i-1}, R)$ For all X and i such that $2 \le i \le n+1$
   - $\delta(q_{n+2}, X) = (q_{n+2}, B, R)$ For all $X \neq B$
   - $\delta(q_{n+2}, B) = (q_{n+3}, B, L)$
   - $\delta(q_{n+3}, X) = (q_{n+3}, X, L)$ For all $X \neq \$$
   - (These are moves in M' that erase the input x and replace it with w.)

Create n new states in M', each of which is responsible for writing one of the n original input symbols in w to the tape.

Proof that L_{ne} is not Recursive, continued

How algorithm B works, continued:

3. Modify the codes for the moves of M by adding $n+3$ to each state's index.
   - M's original first state becomes q_{n+4}, etc.
4. Create a code for the following move of M':
$$\delta(q_{n+3}, \$) = (q_{n+4}, B, R)$$
5. Modify the codes so that M's original q_2 is still the accepting state.

In future proofs, we will not describe these "compiler" algorithms in so much detail; the point is that you can implement them using a TM.

Proof that L_{ne} is not Recursive, continued

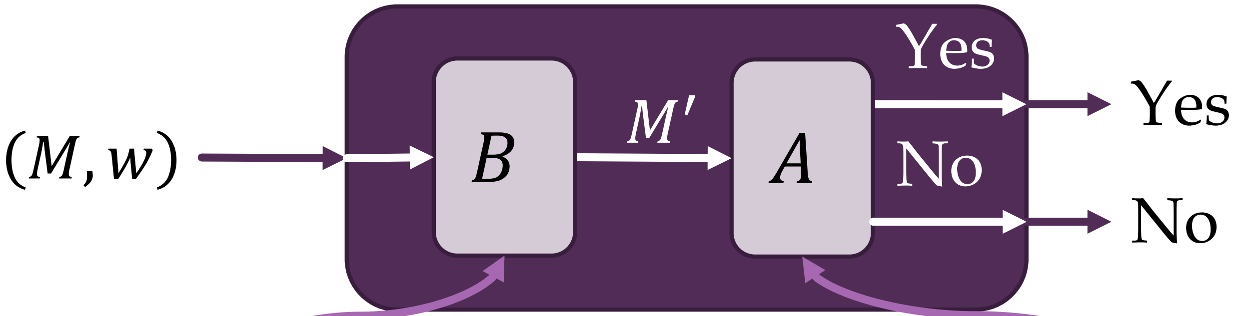
Consider M':

$$L(M') = \begin{cases} \emptyset & \text{If } M \text{ does not accept } w \\ (\mathbf{0}+\mathbf{1})^* & \text{If } M \text{ accepts } w \end{cases}$$

Using the algorithms A and B, we now construct an algorithm for L_u.

Proof that L_{ne} is not Recursive, continued

We can construct an algorithm for L_u using algorithms A and B as follows:

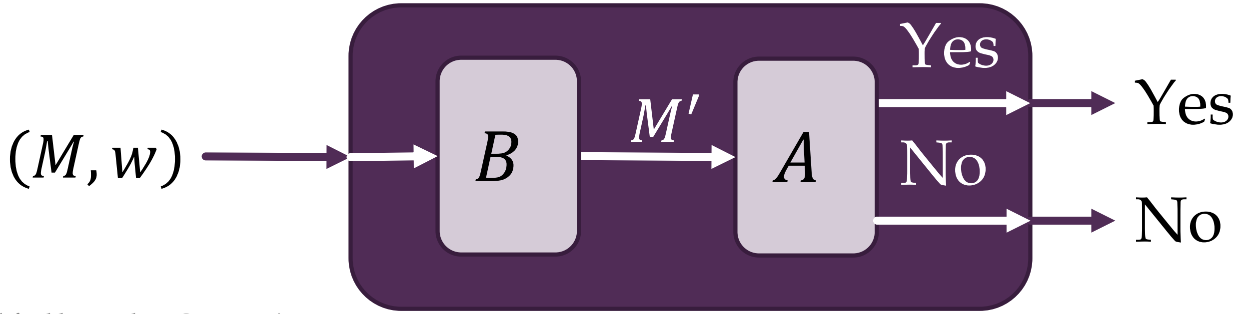


B is the algorithm for creating a TM that always behaves like M on input w, while ignoring actual input.

A is the hypothetical algorithm for L_{ne}, which returns "yes" if and only if a given TM accepts nothing.

Proof that L_{ne} is not Recursive, continued

- If M accepts w, then M' accepts anything, so $L(M')$ is *nonempty*. (So the hypothetical algorithm for L_{ne}, A, answers "yes" and so does the TM below.)
- If M does not accept w, then M' accepts nothing, so $L(M')$ is *empty*. So, A answers "no" and so does the TM below.
- So, the TM below is an algorithm for L_u, which we proved can't exist. This contradicts the assumption that algorithm A, for L_{ne}, exists.



Proof that L_e is not RE

Theorem 9.10: The language of all empty TMs, L_e, is not RE.

Proof:

This follows from the proof that L_{ne} is RE but not recursive, and the theorem that a language and its complement can't both be RE but not recursive.

Other Properties of RE Languages

We will now generalize the previous proofs by considering languages that represent properties of RE languages.

- A **property** P of RE languages is simply *a set of RE languages*.
- For example, *emptiness* is a property, and $L_e = \{M \mid L(M) = \emptyset\}$.
- We say that a language L has the property P if and only if $L \in P$.

Notice how this definition of "property" is the same as a yes/no question about a language.

Other Properties of RE Languages

(Continued from the previous slide)

- A property P is a set of RE languages.
- We say that a language L has the property P if and only if $L \in P$.
- P is a **trivial** property if P is either empty or consists of all RE languages.
- For the sake of notation, let $L_P = \{M \mid L(M) \in P\}$.

Rice's Theorem

Theorem 9.11 (Rice's Theorem): Any nontrivial property P of RE languages is undecidable.

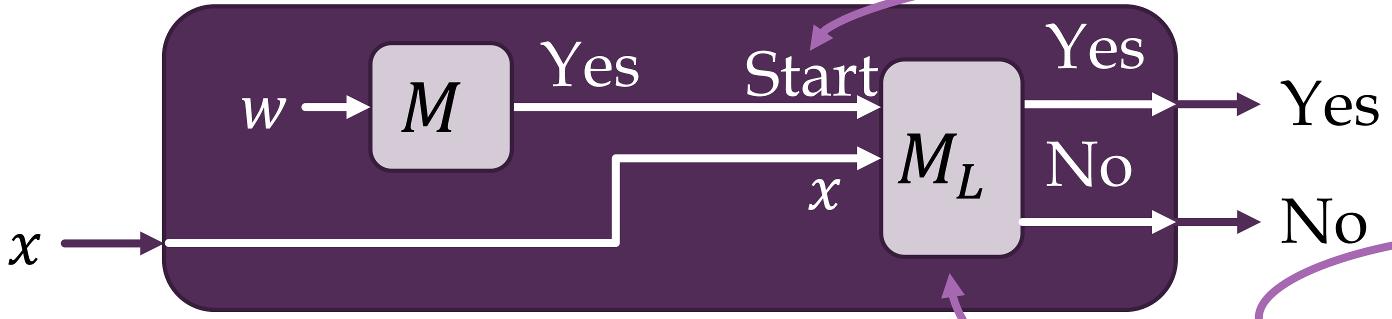
Proof:

Without loss of generality, assume $\emptyset$ (the empty language) is not in P. (If $\emptyset \in P$, we can simply consider $\overline{P}$ instead.)

Because we already said P is nontrivial, there exists some language L in P.

Proof of Rice's Theorem, continued

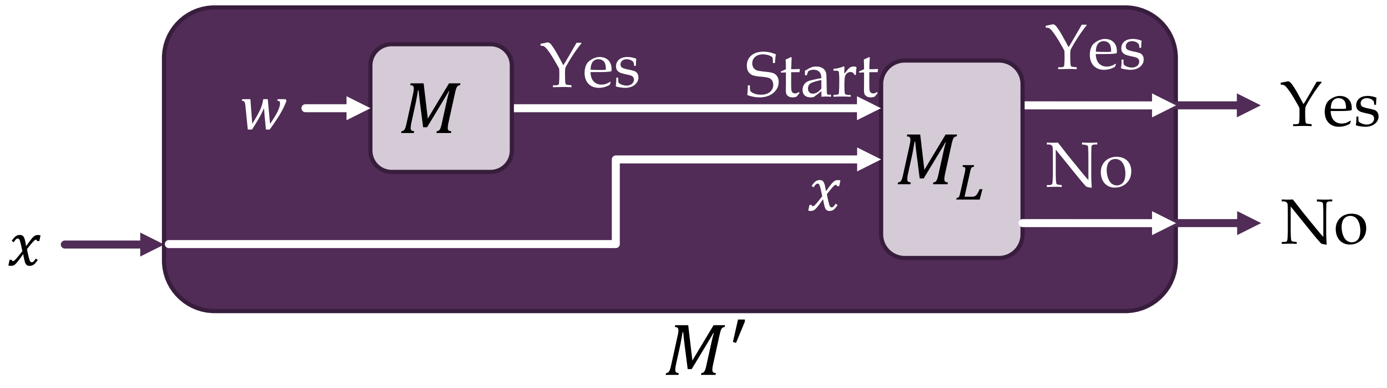
- Suppose any arbitrary property P is decidable. This means there must be an algorithm M_P accepting L_P.
- We will once again construct a "compiler" algorithm A that takes a (M, w) as input and produces M', where M' is as follows:



Only start running M_L after getting a "yes" answer from M.

M_L can just be a TM for any language L with property P.

Proof of Rice's Theorem, continued



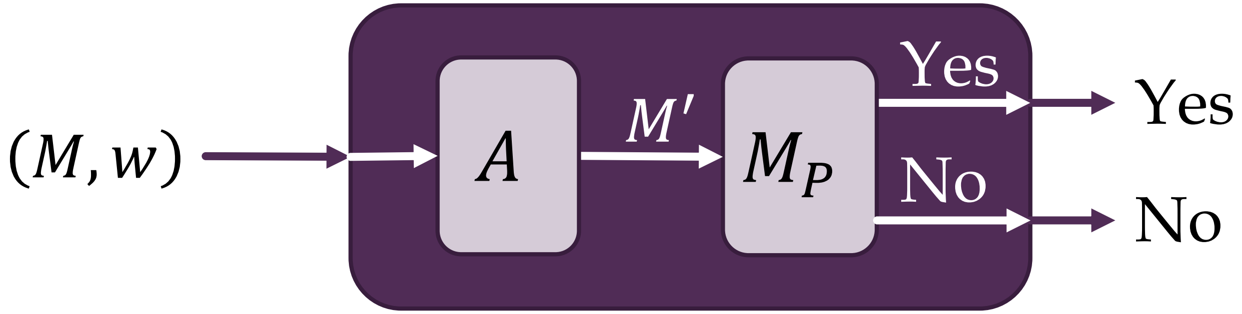
Note that:

$$L(M') = \begin{cases} \emptyset & \text{If } M \text{ does not accept } w \\ L & \text{If } M \text{ accepts } w \end{cases}$$

We can now construct an algorithm for L_u using M'.
(This is very similar to the proof that L_{ne} is undecidable).

Proof of Rice's Theorem, continued

- If M accepts w: Then $L(M') = L$, so M' will have property P. So, M_P answers "yes".
- If M does not accept w: Then $L(M') = \emptyset$, so M' will not have property P. So, M_P answers "no".
- But, we already proved that there is no algorithm for L_u, so M_P can't exist and P is undecidable.



Some Results of Rice's Theorem

The following properties of RE languages are not decidable:

- Emptiness
- Finiteness
- Context-freedom (does a given TM accept a CFL?)
- Regularity

Example Problem About TMs

(These problems may require some ingenuity.)

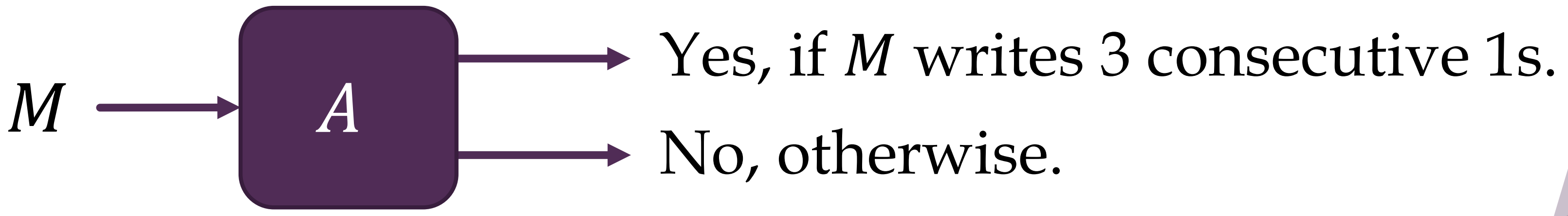
Example 1: Does a TM M with the alphabet $\{0, 1, B\}$ ever print three consecutive $1'$s on its tape?

Claim: This is undecidable.

Proof: On the following slides…

Proof of the "Three 1s" Problem

- Let $L_\varepsilon = \{M \mid \varepsilon \text{ is in } L(M)\}$
- By Rice's Theorem, L_ε is not recursive.
- If the problem from the previous slide is decidable, then there exists an algorithm A as follows:



Proof of the "Three 1s" Problem, continued

- Once again, we will construct a "compiler" algorithm B:

$$M \longrightarrow \boxed{B} \longrightarrow M'$$

- M' simulates M on a blank tape.
- M' uses 10 to encode a 1, and 01 to encode a 0.
  - This prevents "accidentally" printing a 111.
- M' prints 111 if M accepts.
- So, M' prints 111 if and only if ε is in $L(M)$.

Proof of the "Three 1s" Problem, continued

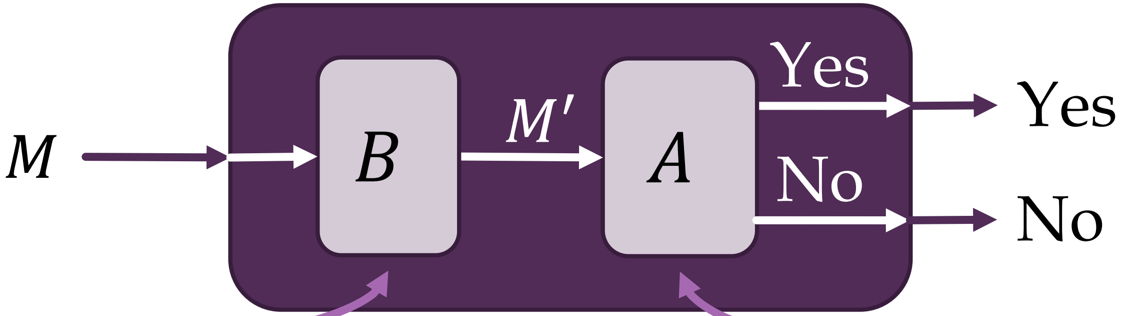
- Once again, we will construct a "compiler" algorithm B:

$$M \longrightarrow \boxed{B} \longrightarrow M'$$

- M' simulates M on a blank tape.
- M' uses 10 to encode a 1, and 01 to encode a 0.
  - This prevents "accidentally" printing a 111.
- M' prints 111 if M accepts.
- So, M' prints 111 if and only if ε is in $L(M)$.

Proof of the "Three 1s" Problem, continued

- We can now construct an algorithm for L_ε, which we know is undecidable.



B is the algorithm that outputs a TM M', where M' prints 3 1s if and only if M accepts ε.

A is the hypothetical algorithm for the "three 1s" problem.

Second Example TM Problem

Example 2: Does a TM M, with a single semi-infinite tape, scan any cell more than once when run on a blank tape?

Claim: This is decidable.

Proof:

Simulate M. If M moves left, answer "yes".

If M continues to move right and repeats a state, it is in a cycle will forever move right. So answer "no".