MA/CSSE 474
Theory of Computation

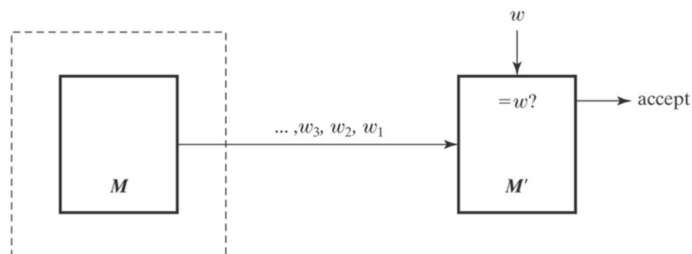
Reduction: Decidability and Undecidability Proofs

SD and Turing Enumerable

Theorem: A language is SD iff it is Turing-enumerable.

Proof that Turing-enumerable implies SD: Let *M* be the Turing machine that enumerates *L*. We use *M* as the basis for a machine *M'* that semidecides *L*.

1. Copy input *w on another tape*.
2. Using M', Begin enumerating *L*. Each time an element of *L* is enumerated, compare it to *w*. If they match, accept.



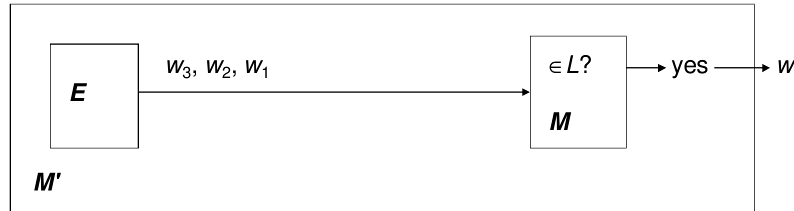
The Other Direction

Proof that SD implies Turing-enumerable:

If $L \subseteq \Sigma^*$ is in SD, then there is a Turing machine *M* that semidecides *L*.

A procedure *E* to enumerate all elements of *L*:

1. Enumerate all $w \in \Sigma^*$ lexicographically.
   e.g., ε, `a`, `b`, `aa`, `ab`, `ba`, `bb`, …
2. As each is enumerated, use *M* to check it.



But there is a problem with this …

Solution: "Dovetail" the computations

ε [1]					
ε [2]	a [1]				
ε [3]	a [2]	b [1]			
ε [4]	a [3]	b [2]	aa [1]		
ε [5]	a [4]	b [3]	aa [2]	ab [1]	
ε [6]	a [5]		aa [3]	ab [2]	ba [1]

Let L = L(M) for some TM M.
A procedure to enumerate all elements of *L*:

1. Enumerate all $w \in \Sigma^*$ lexicographically.
2. As each string w_i is enumerated:
   1. Start up a copy of *M* (call it M_i)with w_i as its input.
   2. Execute one step of each M_j (j < i),
      excluding those M_j that have previously halted.
   3. Whenever an M_i accepts, output w_i.

*

Lexicographic Enumeration

M ***lexicographically enumerates*** *L* iff *M* enumerates the elements of *L* in lexicographic order.

A language *L* is ***lexicographically Turing-enumerable*** iff there is a Turing machine that lexicographically enumerates it.

Example: $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$

Lexicographic enumeration:

*

Lexicographically Enumerable = D

Theorem: A language is in D iff it is lexicographically Turing-enumerable.

Proof that D implies lexicographically TE: Let *M* be a Turing machine that decides *L*.

Then *M'* lexicographically generates the strings in Σ^* and tests each using *M*.

Whenever M accepts w_i, M' outputs w_i.

Thus *M'* lexicographically enumerates *L*.

Proof, Continued

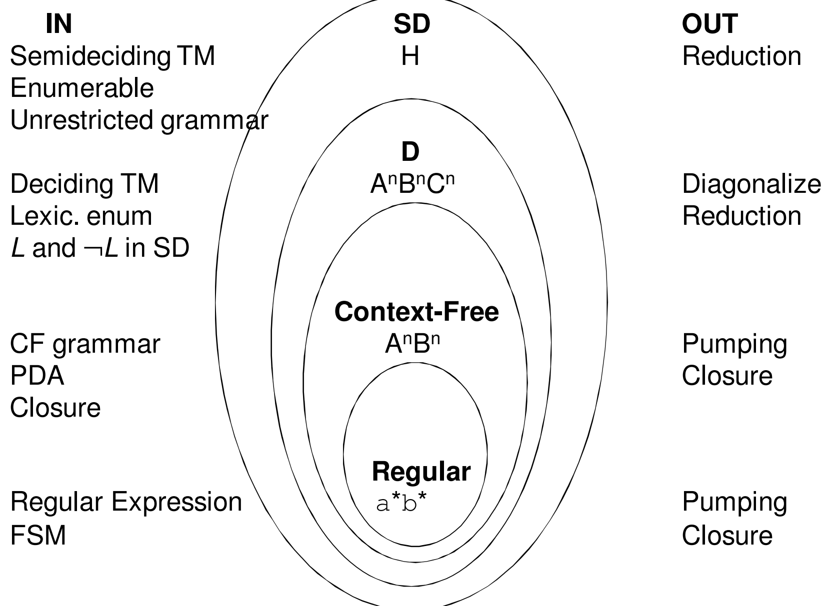
Proof that lexicographically Turing Enumerable implies D:

Let *M* be a Turing machine that lexicographically enumerates *L*. Then, on input *w*, *M′* starts up *M* and waits until:

- *M* generates *w* (then *M′* accepts),
- *M* generates a string that comes after *w* (*M′* rejects), or
- *M* halts (so *M′* rejects).

Thus *M′* decides *L*.

Language Summary



OVERVIEW OF REDUCTION

Reducing Decision Problem P_1 to another Decision Problem P_2

We say that P1 is **reducible** to P_2 (written $P_1 \leq P_2$) if

- there is a Turing-computable function f that finds, for an arbitrary instance I of P_1, an instance f(I) of P_2, and
- f is defined such that for every instance I of P_1,
  I is a yes-instance of P_1 if and only if
  f(I) is a yes-instance of P_2.

So $P_1 \leq P_2$ means "if we have a TM that decides P_2, then there is a TM that decides P_1.

Example of Turing Reducibility

Let

- $P_1(n)$ = "Is the decimal integer n divisible by 4?"
- $P_2(n)$ = "Is the decimal integer n divisible by 2?"
- $f(n) = n/2$ (integer division, which is clearly Turing computable)

Then $P_1(n)$ is "yes" iff
$P_2(n)$ is "yes" and $P_2(f(n))$ is "yes" .

Thus P_1 is reducible to P_2, and we write $P_1 \leq P_2$.

P_2 is clearly decidable (is the last digit an element of $\{0, 2, 4, 6, 8\}$?), so P_1 is decidable

Reducing *Language* L_1 to L_2

- L_1 (over alphabet Σ_1) is **reducible** to L_2 (over alphabet Σ_2) and we write $L_1 \leq L_2$ if

  there is a Turing-computable function
  $f : \Sigma_1^* \to \Sigma_2^*$ such that
  $\forall x \in \Sigma_1^*$, $x \in L_1$ if and only if $f(x) \in L_2$

Using reducibility

- If P_1 is reducible to P_2, then
  - If P_2 is decidable, so is P_1.
  - If P_1 is not decidable, neither is P_2.
- The second part is the one that we will use most.

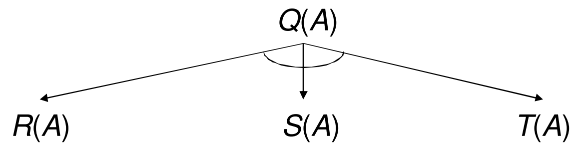
DETAILS OF REDUCTION

More Examples of Reduction

- Theorem proving

Suppose that we want to establish $Q(A)$ and that we have, as a theorem:

$$\forall x\,(R(x) \wedge S(x) \wedge T(x) \rightarrow Q(x)).$$



*

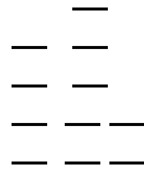
More Examples of Reduction

- Computing a function (where x and y are unary representations of integers)

multiply(x, y) =
1. *answer* := ε.
2. For i := 1 to $|y|$ do:
   answer = concat (*answer*, *x*) .
3. Return *answer.*

So we reduce multiplication to addition.

Nim

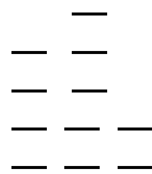


At each turn, a player chooses one pile and removes some sticks from it.

The player who takes the last stick wins.

Problem: Is there a move that guarantees a win for the current player?

Nim



- Obvious approach: search the space of possible moves.
- Reduction to an XOR computation problem:

100	1	10
101	1	01
010	0	11
011		

- XOR them together:
  - ♦ 0^+ means state is losing for current player
  - ♦ otherwise current player can win by making a move that makes the XOR 0.

Using Reduction for Undecidability

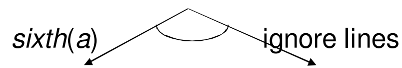
Theorem: There exists no general procedure to solve the following problem:

Given an angle A, divide A into sixths using only a straightedge and a compass.

Proof: Suppose that there were such a procedure, which we'll call *sixth*. Then we could trisect an arbitrary angle:

trisect(a: angle) =
1. Divide a into six equal parts by invoking *sixth*(a).
2. Ignore every other line, thus dividing a into thirds.

trisect(a)



sixth exists → *trisect* exists.

But we know that *trisect* does not exist. So:

http://en.wikipedia.org/wiki/Angle_trisection

Using Reduction for Undecidability

A ***reduction*** R from L_1 to L_2 is one or more Turing machines such that:

If there exists a Turing machine *Oracle* that decides (or semidecides) L_2,
then the TMs in R can be composed with *Oracle*
to build a deciding (or semideciding) TM for L_1.

$P \leq P'$ means that P is reducible to P'.

Using Reduction for Undecidability

(R is a reduction from L_1 to L_2) $\wedge$ (L_2 is in D) $\rightarrow$ (L_1 is in D)

If (L_1 is in D) is false, then at least one of the two antecedents of that implication must be false. So:

If (R is a reduction from L_1 to L_2) is true,
then (L_2 is in D) must be false.

Using Reduction for Undecidability

Showing that L_2 is not in D:

L_1 (known not to be in D)		L_1 in D	But L_1 not in D
R $\downarrow$		$\Uparrow$	$\Downarrow$
L_2 (a new language whose decidability we are trying to determine)		if L_2 in D	So L_2 not in D

To Use Reduction for Undecidability

1. Choose a language L_1:
   - that is already known not to be in D, and
   - that can be reduced to L_2.

2. Define the reduction *R*.

3. Describe the composition *C* of *R* with *Oracle*.

4. Show that *C* does correctly decide L_1 iff *Oracle* exists. We do this by showing:
   - *R* can be implemented by Turing machines,
   - *C* is correct:
     - If $x \in L_1$, then $C(x)$ accepts, and
     - If $x \notin L_1$, then $C(x)$ rejects.

Mapping Reductions

L_1 is *mapping reducible* to L_2 ($L_1 \leq_M L_2$) iff there exists some computable function *f* such that:

$$\forall x \in \Sigma^* \ (x \in L_1 \leftrightarrow f(x) \in L_2).$$

To decide whether *x* is in L_1, we transform it, using *f*, into a new object and ask whether that object is in L_2.

Example:

$DecideNIM(x) = XOR\text{-}solve(transform(x))$

Consider H_ε = {<*M*> : TM *M* halts on ε} *

1. **H_ε is in SD. *T* semidecides it:**

   T(<*M*>) =
   1. Run *M* on ε.
   2. Accept.

   T accepts <*M*> iff *M* halts on ε, so *T* semidecides H_ε.

* Recall: "M halts on w" is a short way of saying "M, when started with input w, eventually halts"

H_ε = {<*M*> : TM *M* halts on ε}

2. Theorem: **H_ε = {<*M*> : TM *M* halts on ε} is not in D.**

Proof: by reduction from H:

H = {<*M*, *w*> : TM *M* halts on input string *w*}

R ↓

(?*Oracle*) H_ε {<*M*> : TM *M* halts on ε}

R is a mapping reduction from H to H_ε:

R(<*M*, *w*>) =
1. Construct <*M#*>, where *M#*(*x*) operates as follows:
   - 1.1. Erase the tape.
   - 1.2. Write *w* on the tape and move the head to the left end.
   - 1.3. Run *M* on *w*.
2. Return <*M#*>.

*

Proof, Continued

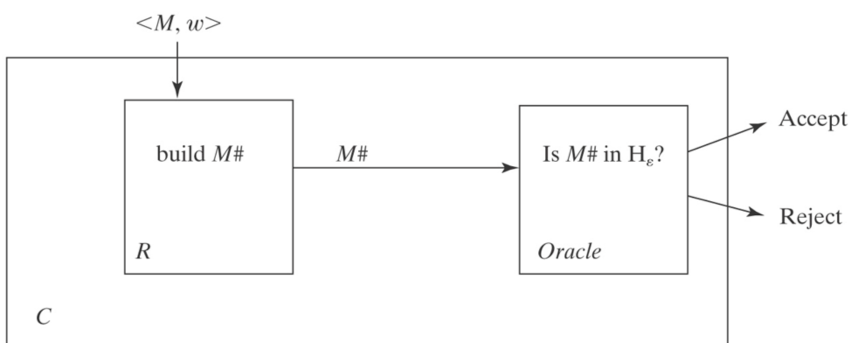
R(<*M*, *w*>) =

1. Construct <*M#*>, where *M#*(*x*) operates as follows:
    1.1. Erase the tape.
    1.2. Write *w* on the tape and move the head to the left end.
    1.3. Run *M* on *w*.
2. Return <*M#*>.

If *Oracle* exists, *C* = *Oracle*(*R*(<*M*, *w*>)) decides H:

- *C* is correct: *M#* ignores its own input. It halts on everything or nothing. So:
    - <*M*, *w*> $\in$ *H*: *M* halts on *w*, so *M#* halts on everything. In particular, it halts on ε. *Oracle* accepts.
    - <*M*, *w*> $\notin$ *H*: *M* does not halt on *w*, so *M#* halts on nothing and thus not on ε. *Oracle* rejects.

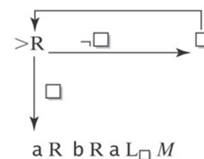
A Block Diagram of *C*



R Can Be Implemented as a Turing Machine

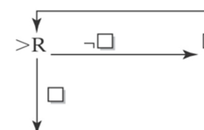
R must construct <*M#*> from <*M*, *w*>. Suppose *w* = `aba`.

M# will be:



So the procedure for constructing *M#* is:

1. Write:



2. For each character *x* in *w* do:
    2.1. Write *x.*
    2.2. If *x* is not the last character in *w*, write R.
3. Write L□ *M.*

Conclusion

R can be implemented as a Turing machine.

C is correct.

So, if *Oracle* exists:

C = *Oracle*(*R*(<*M*, *w*>)) decides H.

But no machine to decide H can exist.

So neither does *Oracle*.

This Result is Somewhat Surprising

If we could decide whether *M* halts on the specific string ε, we could solve the more general problem of deciding whether *M* halts on an arbitrary input.

Clearly, the other way around is true: If we could solve H we could decide whether *M* halts on any one particular string.

But we used reduction to show that H undecidable implies H_ε undecidable; this is not at all obvious.

How Many Languages Are We Dealing With?

H = {<*M*, *w*> : TM *M* halts on input string *w*}

R ↓

(?*Oracle*) H_ε {<*M*> : TM *M* halts on ε}

H contains strings of the form:
(*q*00,*a*00,*q*01,*a*10,←),(*q*00,*a*00,*q*01,*a*10,→),…,aaa

H_ε contains strings of the form:
(*q*00,*a*00,*q*01,*a*10,←),(*q*00,*a*00,*q*01,*a*10,→),…

The language on which some *M* halts contains strings of some arbitrary form, for example,

(letting Σ = {a, b}): aaaba

How Many Machines Are We Dealing With?

H = {<*M*, *w*> : TM *M* halts on input string *w*}

↓ *R*

(?*Oracle*) H_ε {<*M*> : TM *M* halts on ε}

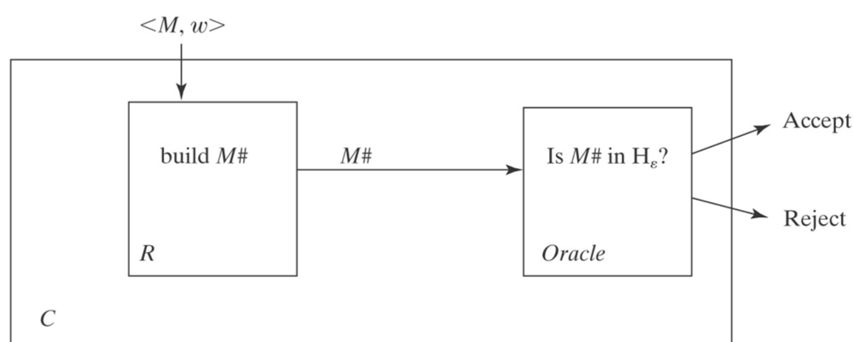
R is a reduction from H to Hε:

R(<*M*, *w*>) =

1. Construct <*M#*>, where *M#*(x) operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M* on *w*.
2. Return <*M#*>.

- *Oracle* (the hypothesized machine to decide H_ε).
- *R* (the machine that builds *M#*. Actually exists).
- *C* (the composition of *R* with *Oracle*).
- *M#* (the machine we will pass as input to *Oracle*). Note that we never run it.
- *M* (the machine whose membership in H we are interested in determining; thus also an input to *R*).

A Block Diagram of *C*



Important Elements in a Reduction Proof

- A clear declaration of the reduction “from” and “to” languages.
- A clear description of *R*.
- If *R* is doing anything nontrivial, argue that it can be implemented as a TM.
- Note that machine diagrams are not necessary or even sufficient in these proofs. Use them as thought devices, where needed.
- Run through the logic that demonstrates how the “from” language is being decided by the composition of *R* and *Oracle*. You must do both accepting and rejecting cases.
- Declare that the reduction proves that your “to” language is not in D.

Another Way to View the Reduction

```
// let L = {<M> | M is a TM that halts on epsilon}
// if L is decidable, let this function decide L:

bool HaltsOnEpsilon(TM M); // defined in magic.h

// HaltsOn decides H using HaltsOnEpsilon
// .: HaltsOn reduces to HaltsOnEpsilon as such:

bool HaltsOn(TM M, string w)
  {// a nested TM
   void Wrapper(string idontcare) {
                                    M(w);
                                    }
   return HaltsOnEpsilon(Wrapper);
   }
```

The Most Common Mistake: Doing the Reduction Backwards

The right way to use reduction to show that L_2 is not in D:

1. Given that L_1 is not in D,
2. Reduce L_1 to L_2, i.e., show how to solve L_1 (the known one) in terms of L_2 (the unknown one)

$$L_1 \downarrow L_2$$

Doing it wrong by reducing L_2 (the unknown one) to L_1:

If there exists a machine M_1 that solves *H*, then we could build a machine that solves L_2 as follows:

1. Return (M_1(<*M*, ε>)).

This proves nothing. It's an argument of the form:

If *False* then …

H_{ANY} = {<*M*> : there exists at least one string on which TM *M* halts}

Theorem: H_{ANY} is in SD.

Proof: by exhibiting a TM *T* that semidecides it.

What about simply trying all the strings in Σ* one at a time until one halts?

H_{ANY} is in SD

T(<*M*>) =

1. Use dovetailing to try *M* on all of the elements of Σ*:

ε [1]
ε [2] a [1]
ε [3] a [2] b [1]
ε [4] a [3] b [2] aa [1]
ε [5] a [4] b [3] aa [2] ab [1]

2. If any instance of *M* halts, halt and accept.

T will accept iff *M* halts on at least one string. So *T* semidecides H_{ANY}.

H_{ANY} is not in D

H = {<*M*, *w*> : TM M halts on input string *w*}

R ↓

(?*Oracle*) H_{ANY} = {<*M*> : there exists at least one string on which TM *M* halts}

R(<*M*, *w*>) =

1. Construct <*M#*>, where *M#*(*x*) operates as follows:
   1.1. Examine *x*.
   1.2. If *x* = *w*, run *M* on *w*, else loop.
2. Return <*M#*>.

If *Oracle* exists, then *C* = *Oracle*(*R*(<*M*, *w*>)) decides H:

- *R* can be implemented as a Turing machine.
- *C* is correct: The only string on which *M#* can halt is *w*. So:
  - <*M*, *w*> ∈ H: *M* halts on *w*. So *M#* halts on *w*. There exists at least one string on which *M#* halts. *Oracle* accepts.
  - <*M*, *w*> ∉ H: *M* does not halt on *w*, so neither does *M#*. So there exists no string on which *M#* halts. *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

(Another *R* That Works)

Proof: We show that H_{ANY} is not in D by reduction from H:

H = {<*M*, *w*> : TM M halts on input string *w*}

R ↓

(?*Oracle*) H_{ANY} = {<*M*> : there exists at least one string on which TM *M* halts}

R(<*M*, *w*>) =

1. Construct the description <*M#*>, where *M#*(*x*) operates as follows:
   1.1. Erase the tape.
   1.2. Write *w* on the tape.
   1.3. Run *M* on *w*.
2. Return <*M#*>.

If *Oracle* exists, then *C* = *Oracle*(*R*(<*M*, *w*>)) decides H:

- *C* is correct: *M#* ignores its own input. It halts on everything or nothing. So:
  - <*M*, *w*> ∈ H: *M* halts on *w*, so *M#* halts on everything. So it halts on at least one string. *Oracle* accepts.
  - <*M*, *w*> ∉ H: *M* does not halt on *w*, so *M#* halts on nothing. So it does not halt on at least one string. *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

The Steps in a Reduction Proof

1. ✪ Choose an undecidable language to reduce from.
2. ✪ Define the reduction *R*.
3. Show that *C* (the composition of *R* with *Oracle*) is correct.

✪ indicates where we make choices.

H_{ALL} = {<*M*> : TM *M* halts on all inputs}

We show that H_{ALL} is not in D by reduction from H_ε.

H_ε = {<*M*> : TM *M* halts on ε}

R ↓

(?*Oracle*) H_{ALL} = {<*M*> : TM *M* halts on all inputs }

R(<*M*>) =

1. Construct the description <*M#*>, where *M#*(*x*) operates as follows:
   1.1. Erase the tape.
   1.2. Run *M*.
2. Return <*M#*>.

If *Oracle* exists, then *C* = *Oracle*(*R*(<*M*>)) decides H_ε:

- *R* can be implemented as a Turing machine.
- *C* is correct: *M#* halts on everything or nothing, depending on whether *M* halts on ε. So:
  - <*M*> ∈ H_ε: *M* halts on ε, so *M#* halts on all inputs. *Oracle* accepts.
  - <*M*> ∉ H_ε: *M* does not halt on ε, so *M#* halts on nothing. *Oracle* rejects.

But no machine to decide H_ε can exist, so neither does *Oracle*.