MA/CSSE 474
Theory of Computation

Enumerability
Reduction

More on Dovetailing

Dovetailing: Run multiple (possibly an infinite number) of computations "in parallel".

S[i, j] represents step j of computation i.

S[1, 1]
S[2, 1]   S[1, 2]
S[3, 1]   S[2, 2]   S[1, 3]
S[4, 1]   S[3, 2]   S[2, 3]   S[1, 4]
. . .
For every i and j, step S[i, j] will eventually happen.

Enumeration

Enumerate means "list, in such a way that for any element, it appears in the list within a finite amount of time."

We say that Turing machine *M* ***enumerates*** the language *L* iff, for some fixed state *p* of *M*:

$$L = \{w : (s, \varepsilon) \vdash_M^* (p, w)\}.$$

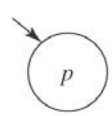
"p" stands for "print"

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

Another term that is often used is **recursively enumerable**.

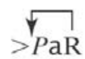
A Printing Subroutine

Let *P* be a Turing machine that enters state *p* and then halts:



Examples of Enumeration

M_1:



M_2:

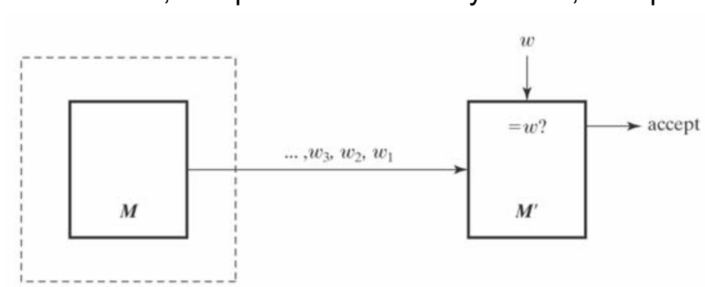
What languages do M_1 and M_2 enumerate?

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 convert *M* to a machine *M'* that semidecides *L*:

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



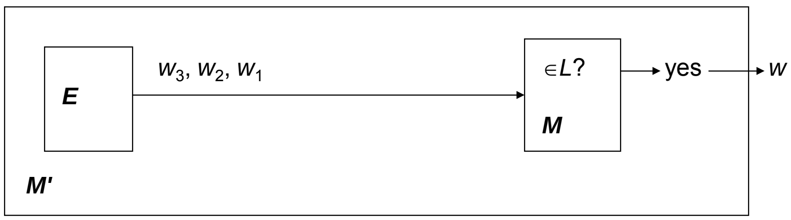
The Other Way

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.



Problem with this?

The Other Way

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 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 with w_i as its input.
   2. Execute one step of each M_i initiated so far, excluding those 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.

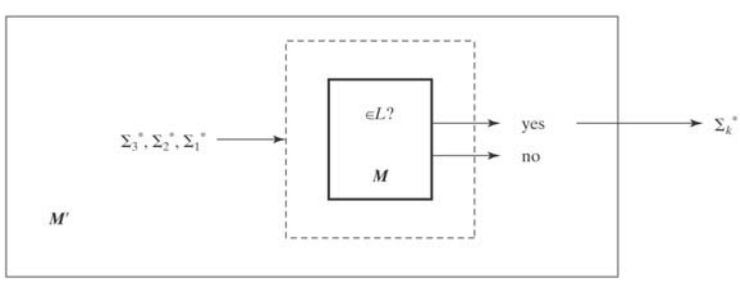
Example: $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$

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*. *M'* lexicographically generates the strings in Σ* and tests each using *M*. It outputs those that are accepted by *M*. Thus *M'* lexicographically enumerates *L*.



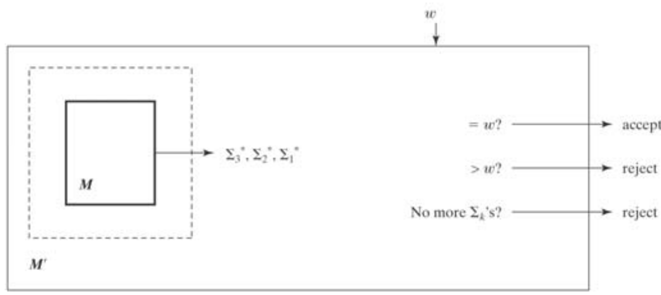
Proof, Continued

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

Thus *M′* decides *L*.



Language Summary

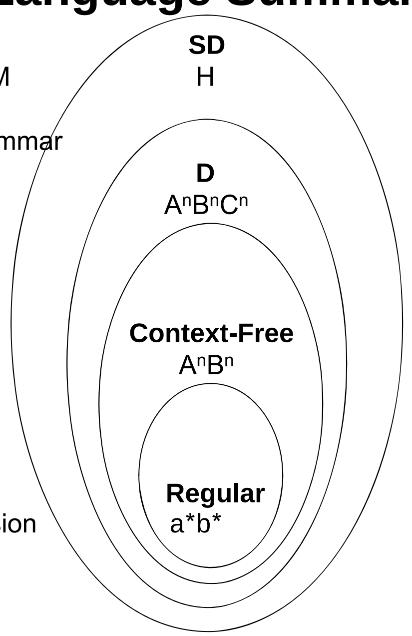
IN

Semideciding TM
Enumerable
Unrestricted grammar

Deciding TM
Lexic. enum
L and ¬*L* in SD

CF grammar
PDA
Closure

Regular Expression
FSM



OUT

Reduction

Diagonalize
Reduction

Pumping
Closure

Pumping
Closure

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

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

  there is a Turing-computable function
  $f : \Sigma_1^* \rightarrow \Sigma_2^*$ 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.

Example of Reduction

- Compute 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. (concatenation)

Using Reduction for Undecidability

A ***reduction*** *R* from language L_1 to language 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
and (L1 is in D) is false,
then (L_2 is in D) must be false.

Application: If L1 is a language that is known to not be in D, and we can find a reduction from L1 to L2, then L2 is also not in D.

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 ↓ — ↑ — ↓

L_2 (a new language whose decidability we are trying to determine) — if L_2 in D — 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
   - show that L_1 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.

Example: $H_\varepsilon = \{<M> : \text{TM } M \text{ halts on } \varepsilon\}$

Follow this outline in proofs that you submit.. We will see many examples in the next few sessions.

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-solve*(*transform*(*x*))

show H_ε in SD but not in D

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:

$H_\varepsilon \leq$ H **is intuitive, the other direction is not so obvious.**

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*> ∈ *H*: *M* halts on *w*, so *M#* halts on everything. In particular, it halts on ε. *Oracle* accepts.
  - <*M*, *w*> ∉ *H*: *M* does not halt on *w*, so *M#* halts on nothing and thus not on ε. *Oracle* rejects.