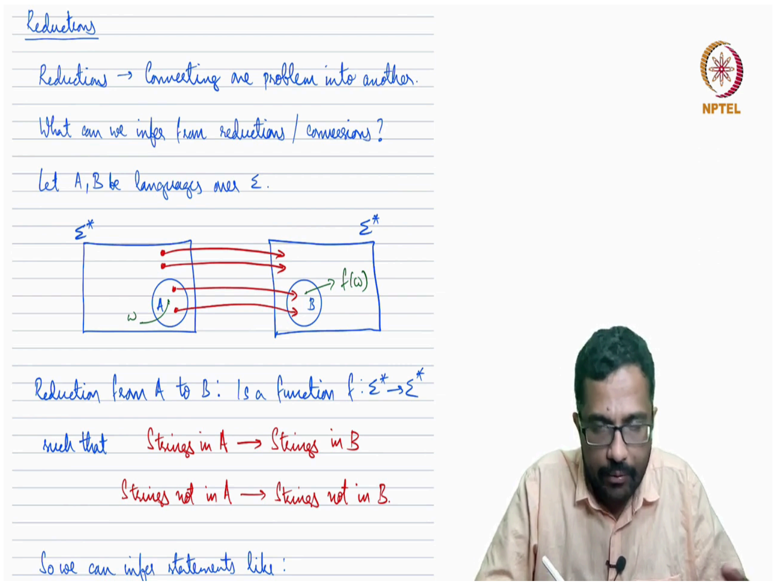
Theory of Computation
Professor Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
Lecture 40
An Introduction to Mapping Reducibility

(Refer Slide Time: 0:21)



Reductions

Reductions → Converting one problem into another.

What can we infer from reductions / conversions?

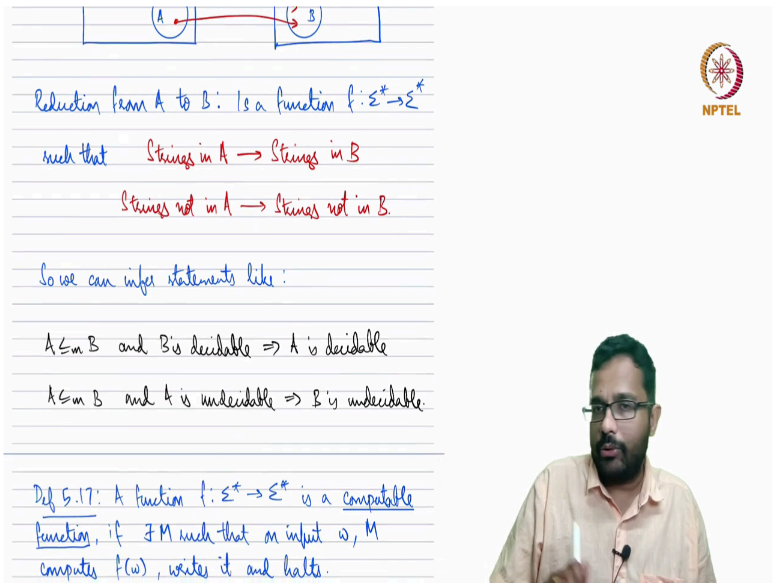
Let A, B be languages over Σ.

Reduction from A to B: Is a function $f : \Sigma^* \to \Sigma^*$

such that Strings in $A \to$ Strings in B

Strings not in $A \to$ Strings not in B.

So we can infer statements like:



Reduction from A to B: Is a function $f : \Sigma^* \to \Sigma^*$

such that Strings in $A \to$ Strings in B

Strings not in $A \to$ Strings not in B.

So we can infer statements like:

$A \leq_m B$ and B is decidable $\Rightarrow$ A is decidable

$A \leq_m B$ and A is undecidable $\Rightarrow$ B is undecidable.

Def 5.17: A function $f : \Sigma^* \to \Sigma^*$ is a computable function, if $\exists M$ such that on input w, M computes $f(w)$, writes it and halts.

Hello and welcome to lecture 39 of the course Theory of Computation, this is also the beginning of week 8. In week 7, we saw languages that are decidable and then we saw one undecidable language which was the language ATM, which we also referred to as the halting problem.

So, the undecidable language ATM was given a specific Turing Machine M and string W does so, we have to determine whether this Turing Machine M accepts the string W. And this was shown to be undecidable. And now, after a lot of hard work and with the theory of countable and uncountable numbers uncountable sets, we were able to show that this language is undecidable. Now, the question is now, how do we show are there other undecidable languages? How do we show the undecidability of other languages?

So, do we have to do all this like from scratch like do we have to prove the undecidability of other languages in a similar way to how we prove the undecidability of ATM. So, it turns out that we do not have to do the hard work all over again, we do not have to do all of the hard work, because we could use the notion of reductions. So, and use the fact that some known language is undecidable for now, the one known undecidable language is ATM, but soon we will figure out other undecidable languages as well.

So, for that the notion of reductions are going to be very useful. So, the reduction in very simple terms means that you convert one problem into another. So, let us say you take a problem A and you are able to convert that into a problem B. So, whenever you need to solve problem A, so since you have converted into problem B you can insist just solve problem B. So, this is a very very high-level idea of what reductions are.

So, just to give an idea, the figure here gives a pictorial depiction of this. So, here we have a Σ^* in the rectangle on the left side, and A as the small circle over here. So, instead of a problem, I am calling it like, you can consider it as a set. So, for instance, maybe the question is, does this given string have an odd number of ones? So, A will be the set of all strings that have an odd number of ones and Σ^* is the set of all strings. So, now, instead of asking does the given string has an odd number of ones we can equivalently asked the is the given string in A.

Similarly, maybe B is the given string a palindrome, so B is set of all palindromes. So, now, what I mean by a reduction or a conversion is that it is just a mapping. So, where if the one string that is in A is mapped to one string in B, so an A string a is mapped to a string in B, a string that is not in A is mapped to string not in B. So, that is what I mean by reduction. So, it is very simple. So, a S instance in the left side is mapped to a S instance in the right side. And a no instance in the left side is mapped to a no instance in the right side.

By S instance I mean something in A or B, by no instance, I mean something not in A or B. So, we want to map strings in A to strings in B and strings not in A to strings not in B. So, that is a meaning of a reduction. So, now what is the goal of this? So, the goal is that now suppose you want to, you want to answer the question, a given string, let us say W you want to answer whether this is in A, so we do not know if we do not have a way of testing whether this is an A.

So, now we compute the mapping of w. Suppose the mapping of w was easily computable. So, then we will map w, let us say for instance, that W was in A, so W was in A and the mapping is computed so, you get some $f(w)$ somewhere over here, write it a bit better $f(w)$ somewhere over here. So, w was in A and f sorry, $f(w)$, the w and $f(w)$ is in B.

So, now, we know how to like given $f(w)$. So, just for the sake of understanding I am saying that $f(w)$ is in A and $f(w)$ in a is in B, but we do not know this we do not know whether w is in A. So, we map and compute $f(w)$ and then check whether $f(w)$ is in B. So, if w is in A $f(w)$ is indeed in B and then when we check when we ask where is $f(w)$ in B we will know that it is in B and that will help us infer that f of that will help us infer that w was in A.

If w was not in A if w was outside suppose w was outside A, then $f(w)$ would also suppose w was here then $f(w)$ would have been somewhere here. So, then when we check whether $f(w)$ is in B then we would get the answer no and that would help us infer that w is also not in A.

So, now notice both of these are important we need yes instance to yes instance mapping that is anything in A should map to anything in B and anything not in A should map to anything not in B. Because if the second thing is also important because suppose w not in A, just for the sake of argument, suppose w not in A got mapped to B, suppose such a mapping like the one that I am drawing now, suppose that was there.

Now, let us say we check the map instance and we say we get that result B, but we do not we cannot infer anything about where it came from because it could have been a w that is in A and got mapped to B or it could have been a w that was not in A and got mapped to B. So, it is important that this kind of mapping is not there every w that is not in A should get mapped to something not in B. So, this kind of thing is not it should not be there.

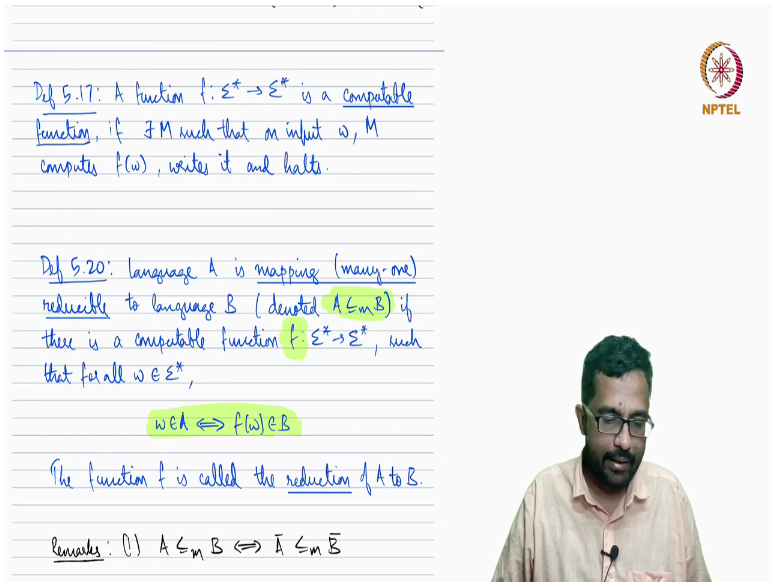
So, the goal of the reduction process is to make judgments about something some string being in A using something that is already known, like suppose we have a way to check whether a given string is in B, then instead of checking whether the given string is in A being converted to the map instance, and then check whether the map and instance is in B.

So, that is the general game plan here. So, now suppose we can reduce. So, now this this leads to some statements. So, I just stated now and we will see it a bit more formally after the formally defining the reduction. So, we can say things like this, suppose so, this notation means that A is reducible to B and A so, the notation is A less than or equal to with a subscript m.

$$A \leq_m B$$

And B is decidable meaning we can check whether a string is in B or not, then A is also decidable because we can convert an A instance to a B instance and then use the decidability of B. Suppose A is reducible to B and A is undecidable, suppose there is no way to tell whether A decide A then this also implies that B is undecidable. This is because if A was undecidable, sorry, if B was decidable, suppose if B was decidable, then the reduction then the if B was decidable, then A reduces to B and B is decidability gives a decidability for A. So, B cannot be decidable. So, these are the two main inferences that we get.

(Refer Slide Time: 9:29)



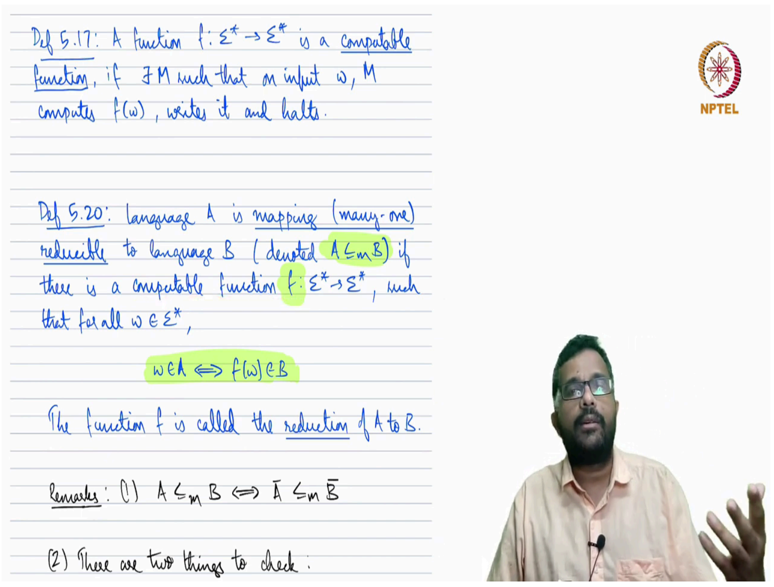
Def 5.17: A function $f: \Sigma^* \to \Sigma^*$ is a computable function, if $\exists$ M such that on input w, M computes $f(w)$, writes it and halts.

Def 5.20: language A is mapping (many-one) reducible to language B (denoted $A \leq_m B$) if there is a computable function $f: \Sigma^* \to \Sigma^*$, such that for all $w \in \Sigma^*$,

$$w \in A \iff f(w) \in B$$

The function f is called the reduction of A to B.

Remarks: (1) $A \leq_m B \iff \bar{A} \leq_m \bar{B}$



Def 5.17: A function $f: \Sigma^* \to \Sigma^*$ is a computable function, if $\exists$ M such that on input w, M computes $f(w)$, writes it and halts.

Def 5.20: Language A is mapping (many-one) reducible to language B (denoted $A \leq_m B$) if there is a computable function $f: \Sigma^* \to \Sigma^*$, such that for all $w \in \Sigma^*$,

$$w \in A \iff f(w) \in B$$

The function f is called the reduction of A to B.

Remarks: (1) $A \leq_m B \iff \bar{A} \leq_m \bar{B}$

(2) There are two things to check:

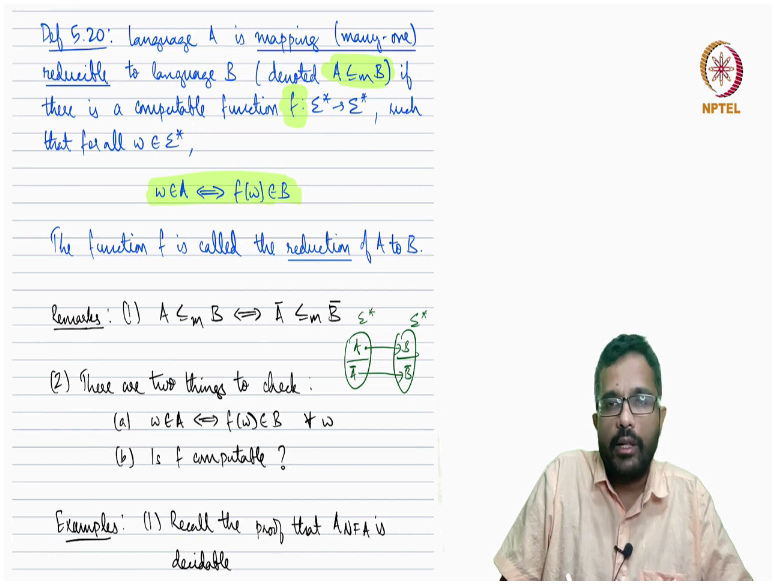
So, we will come back to this right. So, now let us formally define what is the reduction. So, before that, we need to define what is a computable function? A computable function is a function from $\Sigma^* \to \Sigma^*$ such that it can be computed by a Turing machine. So, what do I mean by computed by a Turing machine? So, suppose, the Turing machine gets w in the input, it is able to compute $f(w)$ and it halts. It writes $f(w)$ on the tape and stops.

This is what I mean by Turing Machine computing a function. So, a function is computable If it can be computed by a Turing machine. So, not everything is computable like just like not every function every language is decidable we say that a language A is mapping reducible, so far, I will be just saying reducible so, the formal word is mapping reducible or many one reducible, so this is another terminology many one reducible.

So, we say mapping reducible or many one reducible to language B and denoted by this notation $A \leq_m B$, if there is a computable function f, so, f is a reduction such that for all w we have this, this relation, what so what is this relation? Is same thing that I have been repeating right from the beginning of this lecture, w is in A if and only if $f(w)$ is in B. So, all the strings, so, notice that this is if and only if all the strings in A. So, both ways, all the strings in A should get mapped to strings in B and all the strings not in A should get mapped to strings not in B.

So, or in other words, if $f(w)$ was in B, then w must necessarily be in A. So, if $f(w)$ was in A, it should not be the case that w was not in A. So, it is if and only if w in A if and only if $f(w)$ in B. In other words, w not in A also means $f(w)$ is not in B. So, such a function that satisfies these properties is called the reduction. So, two things one is this relation, this correspondence w in A if and only if $f(w)$ is in B, and second is that the function f should be computable; it cannot be some arbitrary function that exists. So, these two together make the reduction.

(Refer Slide Time: 12:02)



Def 5.20: Language A is mapping (many-one) reducible to language B (denoted $A \le_m B$) if there is a computable function $f : \Sigma^* \to \Sigma^*$, such that for all $w \in \Sigma^*$,

$$w \in A \iff f(w) \in B$$

The function f is called the reduction of A to B.

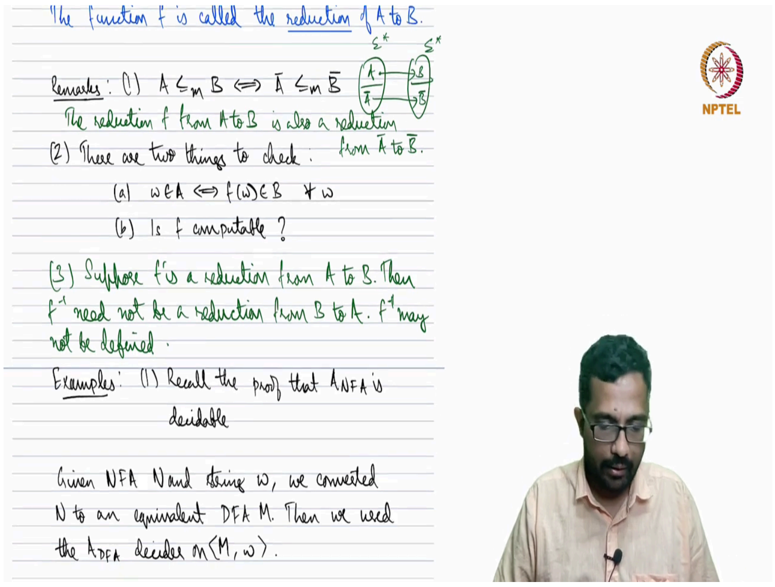
Remarks: (1) $A \le_m B \iff \overline{A} \le_m \overline{B}$

(2) There are two things to check:

(a) $w \in A \iff f(w) \in B \quad \forall w$

(b) Is f computable?

Examples: (1) Recall the proof that A_{NFA} is decidable



The function f is called the reduction of A to B.

Remarks: (1) $A \le_m B \iff \overline{A} \le_m \overline{B}$

The reduction f from A to B is also a reduction from $\overline{A}$ to $\overline{B}$.

(2) There are two things to check:

(a) $w \in A \iff f(w) \in B \quad \forall w$

(b) Is f computable?

(3) Suppose f is a reduction from A to B. Then f^{-1} need not be a reduction from B to A. f^{-1} may not be defined.

Examples: (1) Recall the proof that A_{NFA} is decidable

Given NFA N and string w, we converted N to an equivalent DFA M. Then we used the A_{DFA} decider on $\langle M, w \rangle$.

So, some quick points, first is that a reduction from A to B is also a reduction from A complement to B complement. Why is that maybe we will very quickly draw it on the side.

$$A \le_m B \Leftrightarrow \overline{A} \le_m \overline{B}$$

So, suppose this is Σ^* and this is A this is B suppose there is a mapping right? So, everything in A goes to B and everything not in A goes to not in B. Now, we can view the same function by not in A is A complement and not in B is B complement this also it maps everything in A complement to something in B complement.

And everything not in A complement to something not in B complement. Something not in A complement the double negative not in A complement means it is in A to something not in B complement means it is in B so we can view the same function as also a function from as a reduction from A complement to B complement. So, maybe I will just write it here the reduction f from A to B is also a reduction from A complement to B complement.

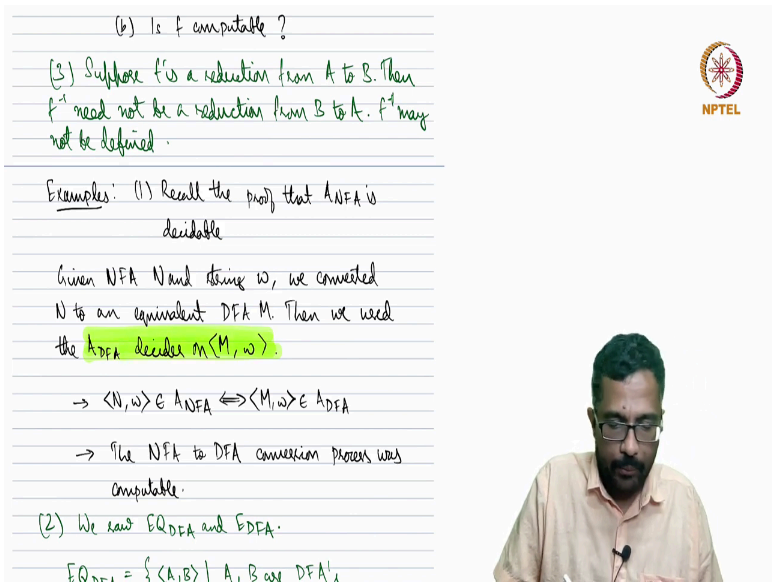
So, a reduction from A to B is also a reduction from A complement to B complement. At this point, I want to say another thing, a reduction from A to B need not be a reduction from B to A maybe I will just note it down somewhere where do I have to move some things around, so one thing a reduction from A to B maybe I will just rewrite it suppose f is a reduction from A to B, then the inverse of f not be reduction from B to A.

So, we may not be able to the main reason is that we may not be able to just invert the function perhaps, so the only thing that we are saying that everything in A should map to everything something in B everything in A should map to something in B, perhaps everything does not go to it does not go to all of B, the image of A maybe only a subset of B. So, that is the reason so there may be so we can we cannot invert the function, there may be other elements in w that, sorry there may be elements of B that are not images of something in A. So, f inverse may not be defined for that.

So, f inverse may not be defined, that is one point. One other thing is that I just want to stress when you when we are checking whether something is a reduction, we need to check two things, first is this correspondence w in A if and only if f(w) is and B and second is, is the function f computable the goal is you transform an instance of A into an instance of B and then use the decider for B.

So, to get a decider for A we need to first transform the instance of A to an instance of B and then use the decider. Now, if the instance of if the transformation is not computable means, we do not know how to you get this transformation, then it is not really useful to just have the decider of B. So, it is important that f be computable, otherwise we do not have a computable way to get to the B instance. So, both of these are important whenever we are talking about reduction both of these needs to be checked.

(Refer Slide Time: 17:03)



(b) Is f computable ?

(3) Suppose f is a reduction from A to B. Then f^{-1} need not be a reduction from B to A. f^{-1} may not be defined.

Examples: (1) Recall the proof that A_{NFA} is decidable

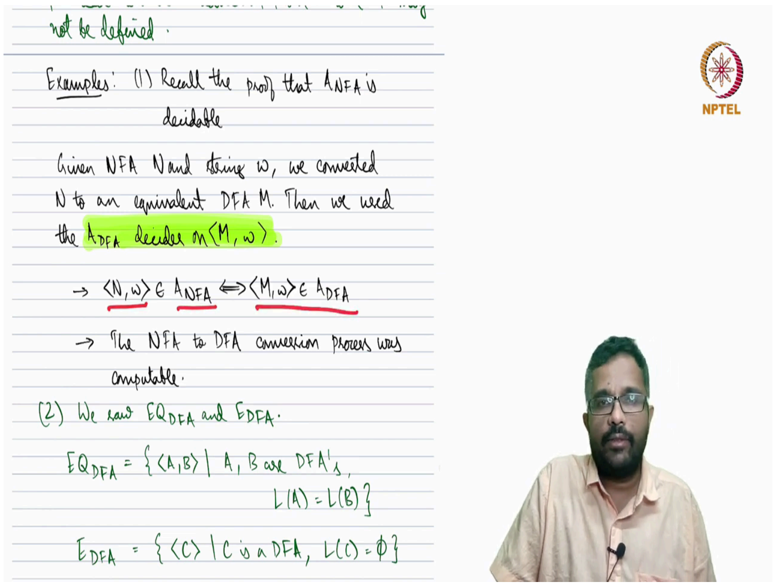
Given NFA N and string w, we converted N to an equivalent DFA M. Then we used the A_{DFA} decider on $\langle M, w \rangle$.

→ $\langle N, w \rangle \in A_{NFA} \Longleftrightarrow \langle M, w \rangle \in A_{DFA}$

→ The NFA to DFA conversion process was computable.

(2) We saw EQ_{DFA} and E_{DFA}.

$EQ_{DFA} = \{ \langle A, B \rangle \mid A, B$ are DFA's,



not be defined.

Examples: (1) Recall the proof that A_{NFA} is decidable

Given NFA N and string w, we converted N to an equivalent DFA M. Then we used the A_{DFA} decider on $\langle M, w \rangle$.

→ $\underline{\langle N, w \rangle} \in \underline{A_{NFA}} \Longleftrightarrow \underline{\langle M, w \rangle \in A_{DFA}}$

→ The NFA to DFA conversion process was computable.

(2) We saw EQ_{DFA} and E_{DFA}.

$EQ_{DFA} = \{ \langle A, B \rangle \mid A, B$ are DFA's, $L(A) = L(B) \}$

$E_{DFA} = \{ \langle C \rangle \mid C$ is a DFA, $L(C) = \phi \}$

Examples, so just to give examples, so the first example is we have in fact already seen some examples in the previous week. So, if you remember, we showed that A_{NFA} is decidable. So, this is the acceptance problem of an NFA. So, given an NFA we want to and a string w the goal was to show the goal was to check whether this string is the string accepted by the given NFA.

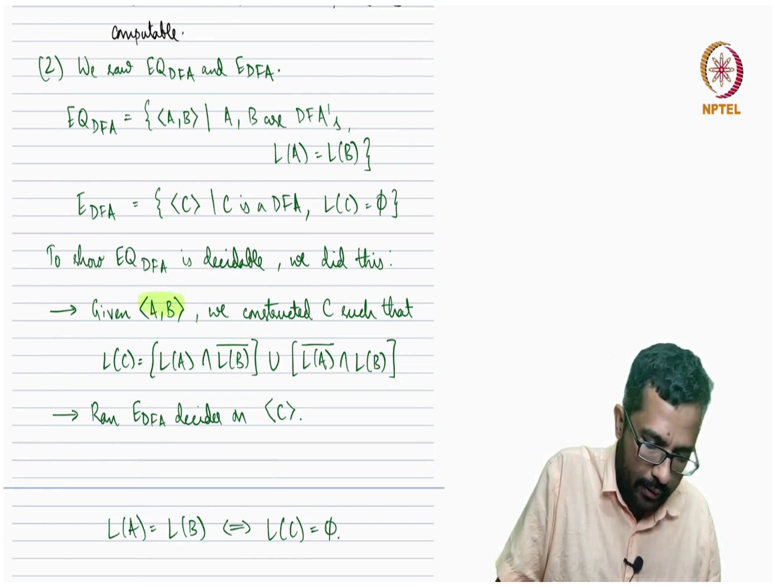
So, what we did was, we recalled that every NFA has an equivalent DFA and we were able to be constructed this equivalent DFA using the equivalence proof and then so since so given NFA and a string w, we converted the NFA N to an equivalent DFA M, and then we use the A_{DFA} decider. So, A_{DFA} is the problem of given a DFA M and a string w is a problem of determining whether the DFA M accepts system w.

So, now, since N and M are equivalent, it is good enough to answer A_{DFA} on it is good enough to answer A_{DFA} on M W, because N and M are equivalent. So, N accepts w if and only if M accepts w, so we could just run the A_{DFA} decider. So, we need to construct the equivalent DFA and then run the A_{DFA} decider, so this is what I am saying. N accepts w if and only if M accepts w, because N and M are equivalent. In other words, <N, w> is in A_{NFA} if and only if <M, w> is in A_{DFA} because A_{NFA} just is asking whether N accepts w and similarly A_{DFA}.

So, <N, w> is in A_{DFA}, sorry A_{NFA} if and only if <M, w> is in A_{DFA}. So, and this is what we want, this is what we want. So, what we want is w is in A if and only if f(w) is in B. Here our w is instead of w we have <N, w> in A_{NFA} if and only if <M, w> is in A_{DFA} so our w is in <N, w> and f(w) is <M, w>. So, the convert and also the conversion process was clearly defined what is a process etc. that is computable.

So, this is an example of something that we have already done, there what we were doing was a reduction. Another example, also something that we have already done is the reduction from EQ_{DFA} to E_{DFA}. So, EQ_{DFA} is the problem of given two DFAs. Are these two equivalent? Do they both recognize the same language E_{DFA} is a problem of given one DFA asking whether this accepts the empty language or this recognizes the empty language.

(Refer Slide Time: 20:18)



computable.

(2) We saw EQ_{DFA} and E_{DFA}.

$EQ_{DFA} = \{\langle A,B\rangle \mid A, B$ are DFA's, $L(A) = L(B)\}$

$E_{DFA} = \{\langle C\rangle \mid C$ is a DFA, $L(C) = \phi\}$

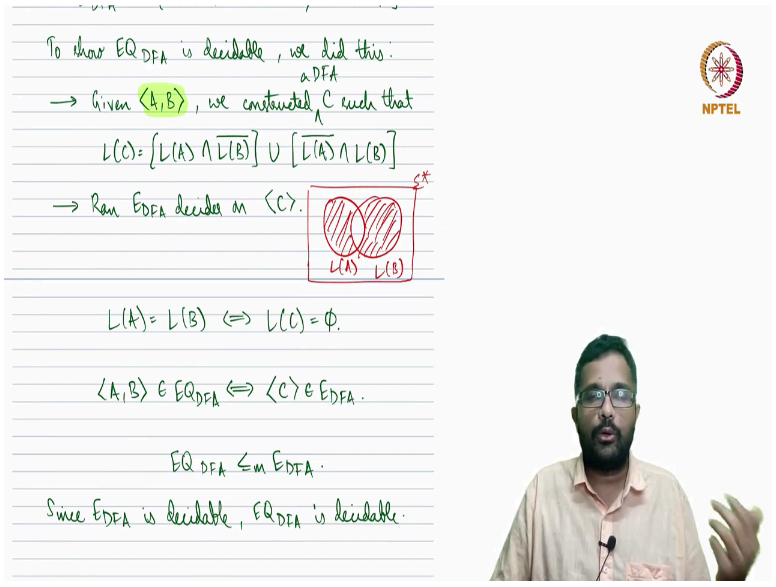
To show EQ_{DFA} is decidable, we did this:

→ Given $\langle A,B\rangle$, we constructed C such that

$L(C) = [L(A) \cap \overline{L(B)}] \cup [\overline{L(A)} \cap L(B)]$

→ Ran E_{DFA} decider on $\langle C\rangle$.

$L(A) = L(B) \iff L(C) = \phi$.



To show EQ_{DFA} is decidable, we did this:

→ Given $\langle A, B\rangle$, we constructed a DFA C such that

$$L(C) = [L(A) \cap \overline{L(B)}] \cup [\overline{L(A)} \cap L(B)]$$

→ Run E_{DFA} decider on $\langle C\rangle$.

$$L(A) = L(B) \iff L(C) = \phi.$$

$$\langle A, B\rangle \in EQ_{DFA} \iff \langle C\rangle \in E_{DFA}.$$

$$EQ_{DFA} \leq_m E_{DFA}.$$

Since E_{DFA} is decidable, EQ_{DFA} is decidable.

So, what we did, if you remember, if you recall is that given an EQ $_{DFA}$ instance, so EQ $_{DFA}$ instance is <A, B>, we constructed a DFA C maybe I will just write that we constructed A $_{DFA}$ C such that C accepts this language. This language, what is this? This is the symmetric difference of a $L(A)$ and $L(B)$. So, this accepts all the strings, so maybe I will just draw it just for the sake of convenience. So, this is $L(A)$ and this is $L(B)$.

So, $L(C)$ is everything that is in $L(A)$, but not in $L(B)$ and everything that is an $L(B)$ but not in $L(A)$. So, this shaded portion, so this is Σ^* the shaded portion is what $L(C)$ should be, so everything that is in A but not in B and everything that is in B but not in A so this is called a symmetric difference.

Now, if the language is in $L(A)$ and $L(B)$ are the same, then the symmetric difference will be empty. So, now after constructing this language, the DFA C we can just ask whether so we have to determine whether A and B are equal. So, instead what we will do is we will construct this DFA C which recognizes the symmetric difference and then just ask whether this DFA does it accept the empty language.

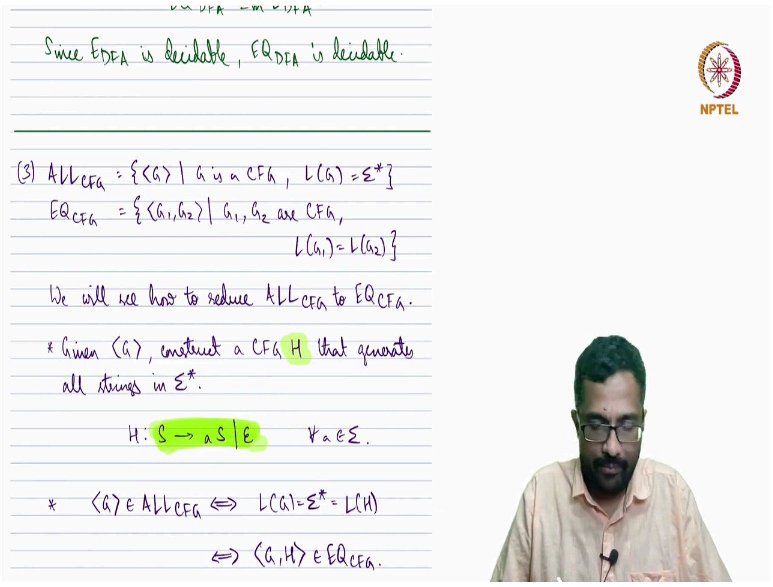
So, once again $L(A) = L(B)$ if and only if the language of C is empty. In other words, A and B is a S instance for EQ $_{DFA}$ if and only if C is a S instance for E $_{DFA}$. So, all of this so notice that if and only if, so if this is an yes instance of EQ $_{DFA}$ this is yes instance and if this is a no instance of EQ $_{DFA}$, there is a no instance as well. So, this is if and only if, and this is important B it is not enough if one direction is satisfied for a reduction we want both of these arrows both of these should be checked, it is not enough if w in A implies f(w) in B both directions have to be checked.

So, now and also the conversion process or the construction process of C is well. It is a procedural process, you may recall from the closure properties of DFAs that we constructed. Hence, we can construct a DFA C such that this is satisfied and hence we can say that EQ $_{\text{DFA}}$ reduces to or mapping reducible to E $_{\text{DFA}}$. And since, if you remember, we had a decidability process for a decidability algorithm for E $_{\text{DFA}}$ you can use this to decide whether EQ $_{\text{DFA}}$ whether the given EQ $_{\text{DFA}}$ instances are decidable.

So, here in the first case we used A $_{\text{DFA}}$ to decide A $_{\text{NFA}}$ here we are using E $_{\text{DFA}}$ is decider to decide EQ $_{\text{DFA}}$. So, now, let us see a slightly different example to the different example, the example is different because, so far, we were saying that A reduces to B if you can convert A instance to B instance and then solve B instance and this helps us solve A instance.

So, B decidable implies A is decidable. So, we reduce EQ $_{\text{DFA}}$ to E $_{\text{DFA}}$, E $_{\text{DFA}}$ is decidable hence EQ $_{\text{DFA}}$ is decidable. Now, we are going to say the opposite we are converting A to B. We know that A is not decidable this implies that B is also not decidable because if B were decidable then we could decide A by converting it to B and then solving B. But we know that is not decidable so this route should not be available to us. We should not be able to convert A to B and then decide B. So, we know the conversion process exists, so hence B should not be decidable. That is the only possible conclusion that we can arrive at.

(Refer Slide Time: 24:35)



Since E_{DFA} is decidable, EQ_{DFA} is decidable.

(3) $ALL_{CFG} = \{\langle G \rangle \mid G \text{ is a CFG},\ L(G) = \Sigma^*\}$

$EQ_{CFG} = \{\langle G_1, G_2 \rangle \mid G_1, G_2 \text{ are CFG},\ L(G_1) = L(G_2)\}$

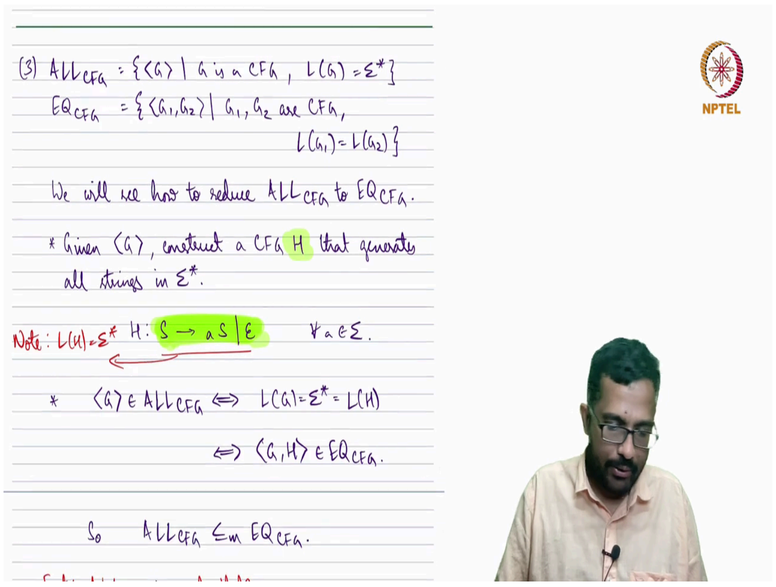
We will see how to reduce ALL_{CFG} to EQ_{CFG}.

* Given $\langle G \rangle$, construct a CFG H that generates all strings in Σ^*.

$H: S \rightarrow aS \mid \varepsilon \quad \forall a \in \Sigma.$

* $\langle G \rangle \in ALL_{CFG} \iff L(G) = \Sigma^* = L(H)$

$\iff \langle G, H \rangle \in EQ_{CFG}.$



(3) $ALL_{CFG} = \{\langle G\rangle \mid G \text{ is a CFG}, L(G) = \Sigma^*\}$

$EQ_{CFG} = \{\langle G_1, G_2\rangle \mid G_1, G_2 \text{ are CFG}, L(G_1) = L(G_2)\}$

We will see how to reduce ALL_{CFG} to EQ_{CFG}.

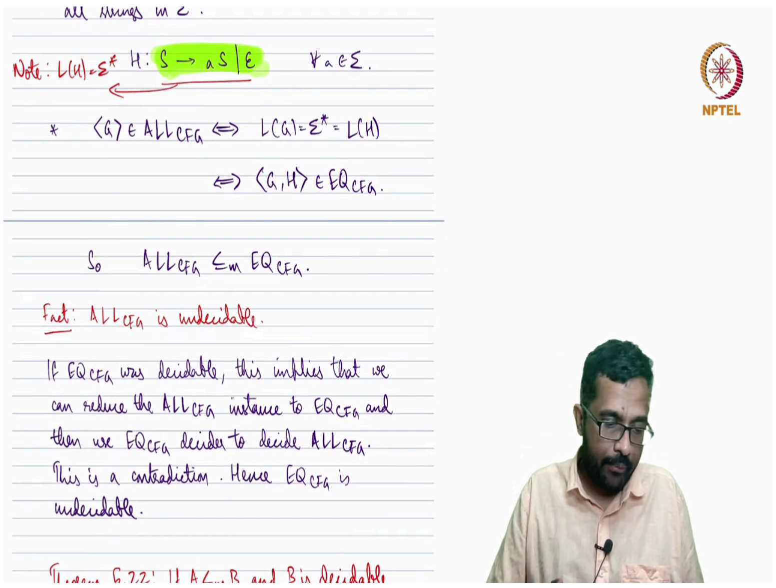
* Given $\langle G\rangle$, construct a CFG H that generates all strings in Σ^*.

Note: $L(H) = \Sigma^*$ H: $S \rightarrow aS \mid \varepsilon$ $\forall a \in \Sigma$.

* $\langle G\rangle \in ALL_{CFG} \iff L(G) = \Sigma^* = L(H)$

$\iff \langle G, H\rangle \in EQ_{CFG}$.

So $ALL_{CFG} \leq_m EQ_{CFG}$.



all strings in Σ^*.

Note: $L(H) = \Sigma^*$ H: $S \rightarrow aS \mid \varepsilon$ $\forall a \in \Sigma$.

* $\langle G\rangle \in ALL_{CFG} \iff L(G) = \Sigma^* = L(H)$

$\iff \langle G, H\rangle \in EQ_{CFG}$.

So $ALL_{CFG} \leq_m EQ_{CFG}$.

Fact: ALL_{CFG} is undecidable.

If EQ_{CFG} was decidable, this implies that we can reduce the ALL_{CFG} instance to EQ_{CFG} and then use EQ_{CFG} decider to decide ALL_{CFG}. This is a contradiction. Hence EQ_{CFG} is undecidable.

Theorem 5.22: If $A \leq_m B$ and B is decidable

So, this is such an example. So, we will reduce ALL_{CFG} to EQ_{CFG}. So, what is ALL_{CFG}? All_{CFG} say is a language that consists of description of context free grammars which accept all the strings or we generate all the strings. So, $L(G)$ must be Σ^* so all the strings should be generated seems like a very easy thing to check. The grammar should generate all possible strings. EQ_{CFG} is the same as EQ_{DFA} or similar to EQ_{DFA} given two grammars G1 and G2 we are asking whether they both generate the same language G $L(G1)$ equal to $L(G2)$.

So, now we will see how to reduce ALL_{CFG} to EQ_{CFG}. So, ALL_{CFG} is asking whether this grammar generates everything EQ_{CFG} giving two grammars and asking whether these two grammars are equivalent. So, this is not that difficult actually. So, given a grammar G and as an ALL_{CFG} instance, we want to know whether G generates everything. So, it is very easy to convert it to an EQ_{CFG} instance, what we will do is we will construct a CFG H. It is a very simple CFG, so H is this, H has only two types of rules.

The starting one is that the starting variable is, one second. So, H is this, what is that? There will be one variable which is starting variable S, S generates A S where a for any terminal A and two is S generates empty string. So, basically any string in the using this Σ , the alphabet Σ can be produced using this, so this is very easy to see. Maybe I will just note it here.

$$H\colon\ S \to aS|\epsilon$$

Note, L(H) is Σ^*. So, L of H is Σ^* and so now we can just ask is G and H equivalent? Instead of asking does G generate everything, we have constructed a grammar that generates everything which and H is is the fact that it generates everything is very easy to say. So, now, instead of asking whether G generates everything, we can ask that G and H are equivalent because if G and H are equivalent G will also generate everything. So, that is it so G is an all CFG, meaning G generates everything if and only if G generates everything. But then everything the Σ^* is also the language of H.

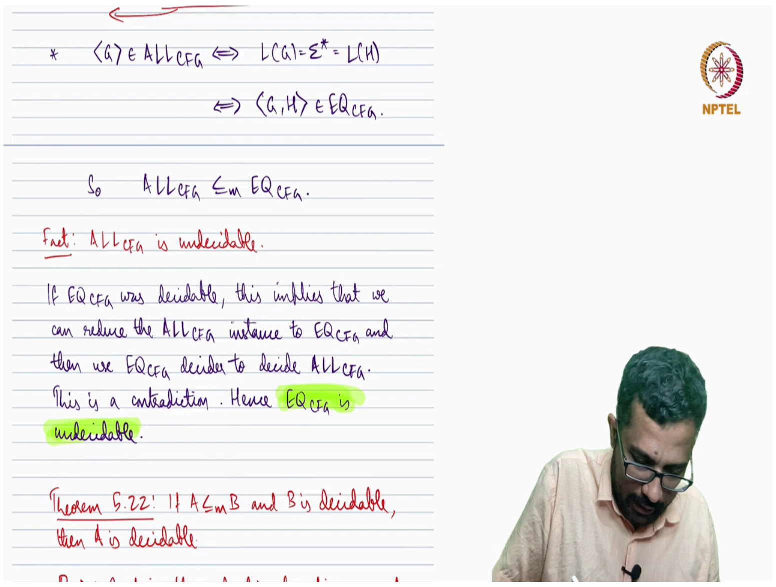
So, now it is the same as asking is G and H equivalent? Or G and H equivalent. So, G is an all CFG if and only if G H, the pair <G ,H> is an EQ_{CFG}, and that is what we want and the construction of this EQ_{CFG} instance is also easy, all we need to do is so you have given G some grammar G, all we need to do is to construct H and H is this, we just write down this grammar and then give it to the next machine, that's it.

So, constructing <G,H> is easy and this if and only if condition is also satisfied. Hence, we can say that ALL_{CFG} reduces to EQ_{CFG}. So, this tells is that ALL_{CFG} is mapping reducible to EQ_{CFG} we can convert an ALL_{CFG} instance to an EQ_{CFG} instance. Now, suppose I am telling now that ALL_{CFG} is undecidable, meaning there is no way to decide given a grammar whether it generates all the strings.

Now, the point that I want to make is that this implies that EQ_{CFG} is also undecidable. Why? Because if EQ_{CFG} was decidable we can decide ALL_{CFG} by converting it into equals EQ_{CFG} and then deciding EQ_{CFG}. If EQ_{CFG} was decidable we could do this but EQ_{CFG} but we know that ALL_{CFG} is not decidable, hence EQ_{CFG} should not be decidable. Otherwise we could do this thing that I said.

Hence, it follows that so just reading through this again, if EQ $_{CFG}$ was decidable, we can reduce the ALL$_{CFG}$ instance to EQ $_{CFC}$ and then use the EQ $_{CFG}$ decider to decide ALL$_{CFG}$. But then ALL$_{CFG}$ is undecidable so I am just stating I am just using this without proof. So, ALL$_{CFG}$ is undecidable. So, this contradicts that, that implies that our assumption that the EQ $_{CFG}$ was decidable was wrong. Hence it is undecidable.

(Refer Slide Time: 29:39)



* $\langle G\rangle \in ALL_{CFG} \iff L(G)=\Sigma^* = L(H)$

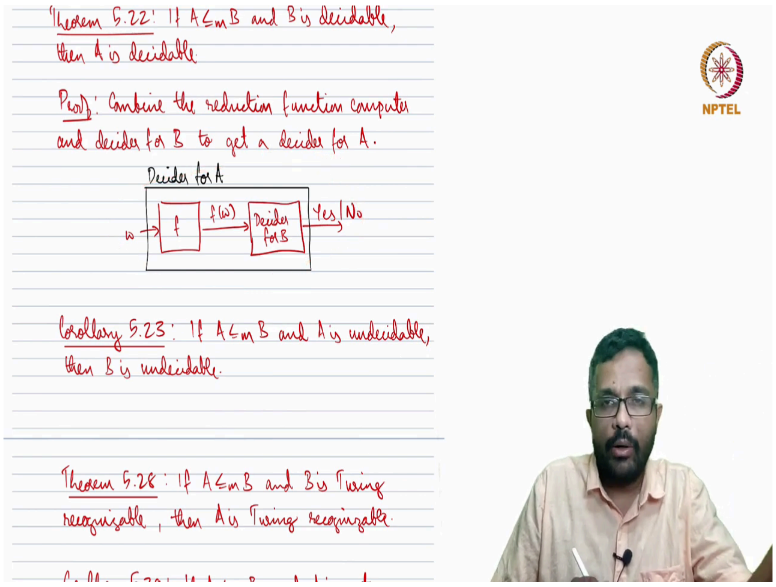
$\iff \langle G,H\rangle \in EQ_{CFG}$.

So $ALL_{CFG} \leq_m EQ_{CFG}$.

Fact: ALL_{CFG} is undecidable.

If EQ_{CFG} was decidable, this implies that we can reduce the ALL_{CFG} instance to EQ_{CFG} and then use EQ_{CFG} decider to decide ALL_{CFG}. This is a contradiction. Hence EQ_{CFG} is undecidable.

Theorem 5.22: If $A \leq_m B$ and B is decidable, then A is decidable



Theorem 5.22: If $A \leq_m B$ and B is decidable, then A is decidable

Proof: Combine the reduction function computer and decider for B to get a decider for A.

Corollary 5.23: If $A \leq_m B$ and A is undecidable, then B is undecidable.

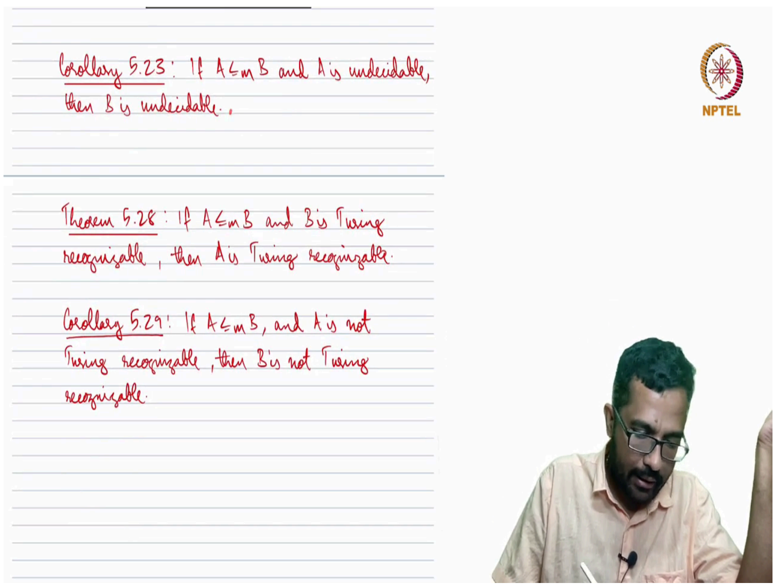
Theorem 5.28: If $A \leq_m B$ and B is Turing recognizable, then A is Turing recognizable.

So, ALL$_{CFG}$ undecidable implies that EQ $_{CFG}$ is undecidable. So, now we have seen decidability and undecidability using reductions. Now, let me just formally state it, if A reduces to B, and B is decidable, then A is decidable. So, just formally stating this theorem 5.22 in the book. So, what do we do we take, we first compute a given a string w and we want to check whether it is an A or not. We just compute or we just compute the function f, which is a reduction from A to B.

So, this is a reduction machine from A to B and then input f to the decider for B. So, the assumption is that B is decidable so there is a decider for B and whatever if B says f(w) is in B yes, then we say w is in A, if B says f(w) is not in B then w is not in A. So, the black thing is the decider for A. So, using a decider for B and reduction function, we get a decider for A.

And the same thing implies that if A reduces to B and A is undecidable, then B is undecidable is exactly what we set here in the context of EQ_{CFG} and ALL_{CFG} if A is if B was decidable by this construction over here, we could not we can construct a decider for A but if we know A is undecidable, then we cannot have a decider.

(Refer Slide Time: 31:13)

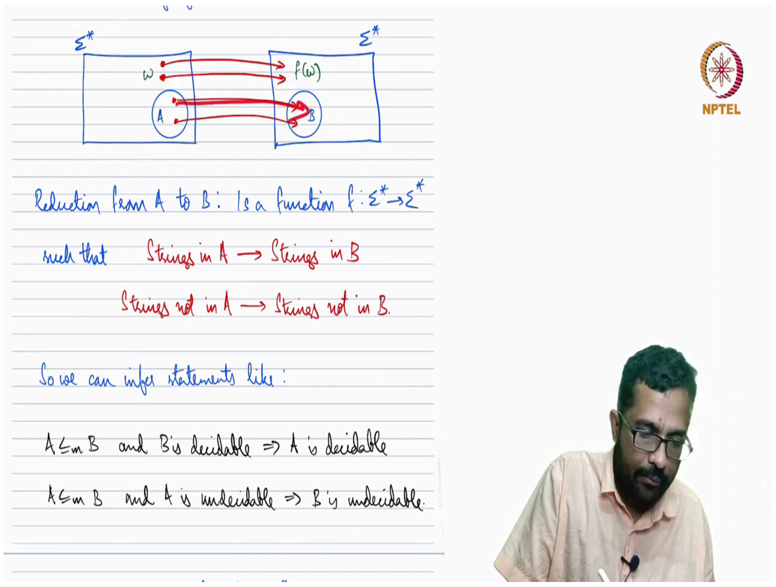

Corollary 5.23: If $A \leq_m B$ and A is undecidable, then B is undecidable.

Theorem 5.28: If $A \leq_m B$ and B is Turing recognizable, then A is Turing recognizable.

Corollary 5.29: If $A \leq_m B$, and A is not Turing recognizable, then B is not Turing recognizable.

And the next two things are theorem 5.28 and corollary 5.29 is the same thing, except that we have replaced decidability by recognisability, that is the only thing if A reduces to B and B is Turing recognizable, then A is Turing recognizable, and if A reduces to B and A is not recognizable, B is also not recognizable. So, that is pretty much it for as far as this lecture 39 is concerned.

(Refer Slide Time: 31:44)



Reduction from A to B: Is a function $f: \Sigma^* \to \Sigma^*$

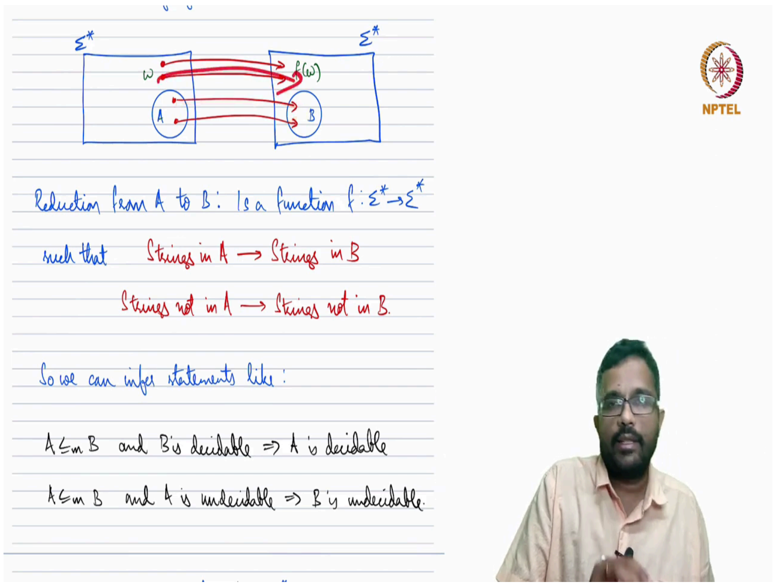
such that Strings in A $\to$ Strings in B

Strings not in A $\to$ Strings not in B.

So we can infer statements like:

$A \leq_m B$ and B is decidable $\Rightarrow$ A is decidable

$A \leq_m B$ and A is undecidable $\Rightarrow$ B is undecidable.


Reduction from A to B: Is a function $f: \Sigma^* \to \Sigma^*$

such that Strings in A $\to$ Strings in B

Strings not in A $\to$ Strings not in B.

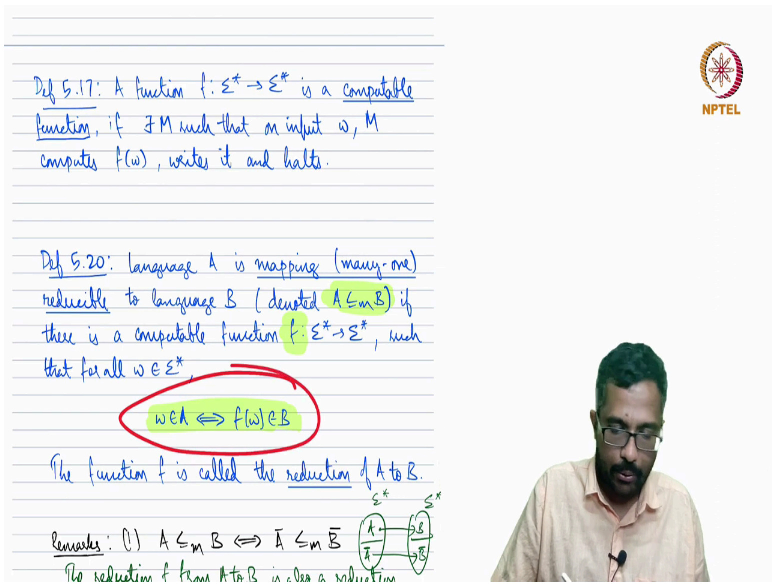
So we can infer statements like:

$A \leq_m B$ and B is decidable $\Rightarrow$ A is decidable

$A \leq_m B$ and A is undecidable $\Rightarrow$ B is undecidable.

So, we saw reductions and what are reductions? Reduction is the way to transform one problem into one language into another. So, the main thing is that the reduction should be able computable . The function should be computable; there should be a Turing machine that computes this function. And the main thing is that all the yes instances of A should go to yes instance B and all the no instances of A should go to the no instance of B, this is the most important two things to check.

(Refer Slide Time: 32:12)



Def 5.17: A function $f : \Sigma^* \to \Sigma^*$ is a computable function, if $\exists$ M such that on input w, M computes $f(w)$, writes it and halts.

Def 5.20: language A is mapping (many-one) reducible to language B (denoted $A \leq_m B$) if there is a computable function $f : \Sigma^* \to \Sigma^*$, such that for all $w \in \Sigma^*$,

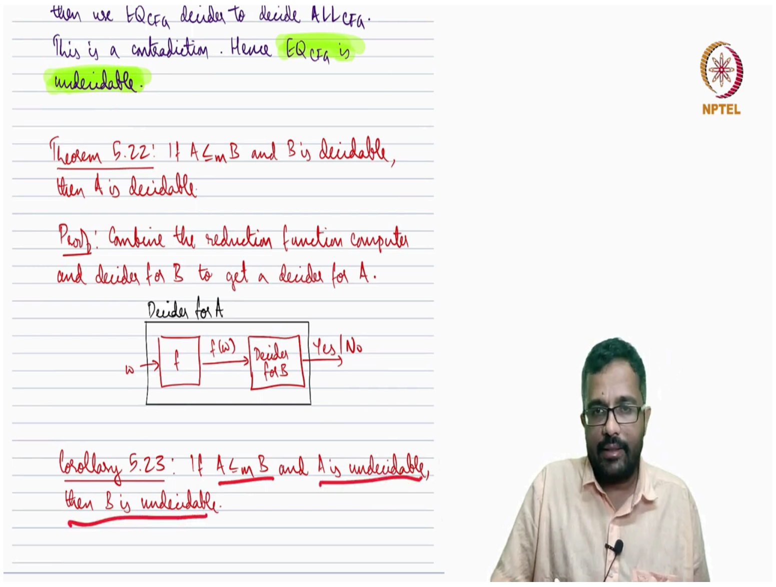
$$\boxed{w \in A \iff f(w) \in B}$$

The function f is called the reduction of A to B.

Remarks: (1) $A \leq_m B \iff \bar{A} \leq_m \bar{B}$

One is that the function is computable, two is that the sequence that is or this correspondence, if w is in A, then how w is in B, if w is not in A, then f(w) is not in B.

(Refer Slide Time: 32:34)

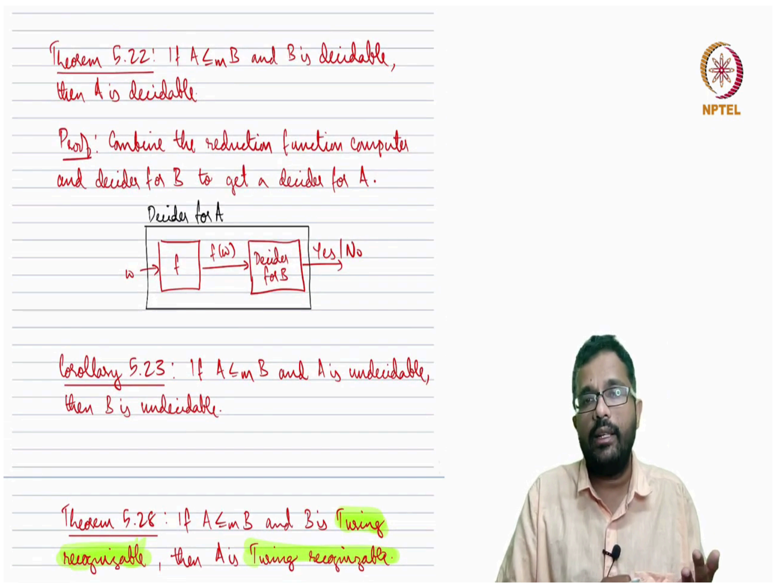


then use EQ_{CFG} decider to decide ALL_{CFG}.
This is a contradiction. Hence EQ_{CFG} is undecidable.

Theorem 5.22: If $A \leq_m B$ and B is decidable, then A is decidable.

Proof: Combine the reduction function computer and decider for B to get a decider for A.

Corollary 5.23: If $A \leq_m B$ and A is undecidable, then B is undecidable.



Theorem 5.22: If $A \leq_m B$ and B is decidable, then A is decidable.

Proof: Combine the reduction function computer and decider for B to get a decider for A.

Corollary 5.23: If $A \leq_m B$ and A is undecidable, then B is undecidable.

Theorem 5.28: If $A \leq_m B$ and B is Turing recognizable, then A is Turing recognizable.

And this if A reduces to B and B is decidable this implies A is decidable if A reduces to B and A is undecidable, this implies B is undecidable. So, using this, we saw that assuming ALL_{CFG} was undecidable, we saw that EQ_{CFG} was undecidable and we recalled two decidability proves that we have already seen and in fact, we had used reductions there. But at that time, we did not have the framework for reductions.

Now, we are kind of setting it in the framework for reductions and viewing them as reductions, the A_{NFA} and EQ_{DFA} proofs. So, that completes lecture 39. So, now we know how to get through something undecidable using reductions. So, in lecture 40 we will show we will start seeing we will start proving that certain languages are undecidable by using this sorry, by using this corollary 5.23. The corollary that A reduces to B and A is undecidable implies B is undecidable. So, that is in lecture 40 and that is all from me in lecture 39. Thank you.