

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

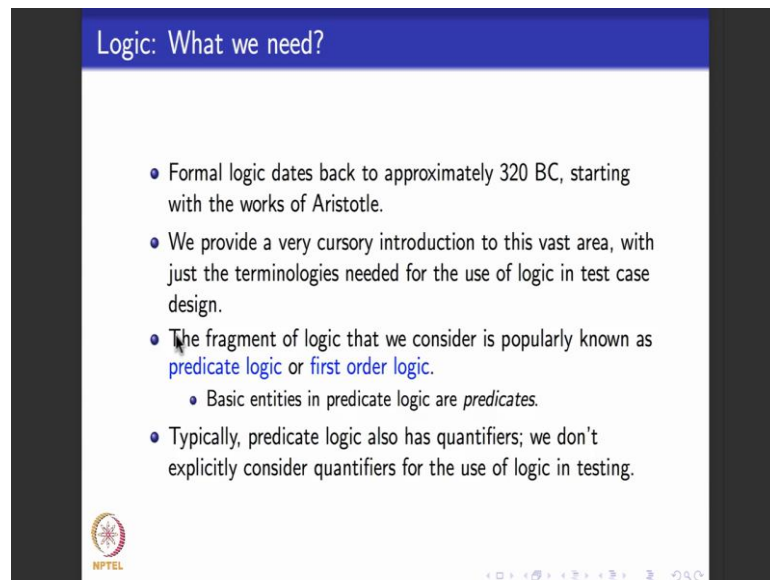
Lecture - 21
Logics: Basics needed for Software Testing

Hello again. We are in week 5: finally, done with graph coverage criteria. What we are going to do now is to move on and see algorithms for test case design based on predicates and logic. You might ask why logic? Why are we looking at logic? So the reason is, if you see a typical program, every kind of decision statement in the program-- - if, while, for loops, do while loops and so on; they have a predicate in them and the predicate is an expression that is meant to evaluate to true or false, and based on whether it evaluates to true or false, different execution paths are taken by the program.

So, logic is the very important part of the program and this week, what we are going to see is how to design test cases based on the logical predicates that occur in programs and later based on the logical predicates that occur as a part of specifications. So, like we did for graph coverage criteria, we will not look at programs and predicates that occur in them first; instead what we will spend time on, is directly looking at logic.

In this module, I will give you a basic of logic, basic introduction to logic as we would need it for design of test cases. Next module, we will introduce coverage criteria based on logical predicates without really seeing where these logical predicates come from. And then after seeing the coverage criteria, the different kinds, their subsumption from a purely theoretical basis, we will go and look at how we can apply to do the coverage criteria on source code and then on specification.

(Refer Slide Time: 01:57)



Logic: What we need?

- Formal logic dates back to approximately 320 BC, starting with the works of Aristotle.
- We provide a very cursory introduction to this vast area, with just the terminologies needed for the use of logic in test case design.
- The fragment of logic that we consider is popularly known as **predicate logic** or **first order logic**.
 - Basic entities in predicate logic are *predicates*.
- Typically, predicate logic also has quantifiers; we don't explicitly consider quantifiers for the use of logic in testing.

NPTEL

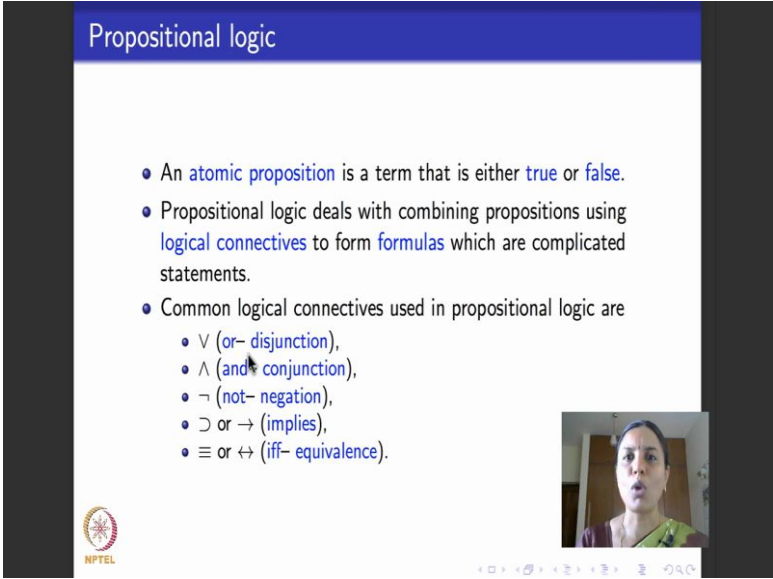
So in this module, I will introduce you to the basics of logic as we would need it for software testing. Formal logic is a very old area, as old as Mathematics is. Philosophers, mathematicians and astronomers several people have used logic. It approximately dates back to 320 BC starting with the work of Aristotle. By no means I can do justification to introducing you to any kind of decent fragment of logic in this course. The goal of this lecture is to be able to look at logic, just about as it is enough for designing test cases as we would need in this course in software testing.

So, the fragment of logic that we need to work with when we do software testing is what is called predicate logic or first order logic. Here you assume that there are functions, relations, put together called as predicates and then they are meant to eventually assume true or false values. But to be able to get to predicate logic, I need to introduce what is called propositional logic which most of you; if you have done a course in discrete maths or a course in logic, you would know about propositional logic. We need propositional logic which is a basic building block of every other logic including predicate logic and then we will do predicate logic.

Typically, another thing to note is that predicate logic or first order logic also has quantifiers. You might have heard two quantifiers for all and there exists, its very important part of first order logic or predicate logic, but as far as its use in testing is concerned we do not really need these quantifiers. So, I will introduce predicate logic

assuming that they are not going to use these quantifiers explicitly. Before we get on to predicate logic, I would like to spend some time recapping propositional logic. As I am not really sure if each of you have been through a course on discrete maths and really know propositional logic. This assumes that you have not seen it and we will do it from the basics. I will introduce you to the basics of propositional logic as we need it and then move on to predicate logic.

(Refer Slide Time: 03:58)



Propositional logic

- An **atomic proposition** is a term that is either **true** or **false**.
- Propositional logic deals with combining propositions using **logical connectives** to form **formulas** which are complicated statements.
- Common logical connectives used in propositional logic are
 - \vee (**or**- disjunction),
 - \wedge (**and**- conjunction),
 - \neg (**not**- negation),
 - \supset or \rightarrow (**implies**),
 - \equiv or \leftrightarrow (**iff**- equivalence).

NPTEL

So, what is propositional logic? Propositional logic can be thought of as a absolute basic logic that occurs as a subset of several different kinds of logic that are used in mathematics and philosophy. The building blocks of propositional logic are what are called atomic propositions. An atomic proposition can be thought of as a Boolean entity or a Boolean variable. It is always meant to be either true or false. and propositional logic tells you how to take an entity that is true or false, that is how to take an atomic proposition and how to combine it with the Boolean connectives.

So, here are the various Boolean connectives that we would be using or written like this, written like a \vee and a conjunction written like this; read it as; and not or negation written like this and implies could be represented in two different ways, this is right arrow symbol and this could be thought of as a superset symbol. Different books use these symbols interchangeably. So, that is an implication operator and then finally, you have

an equivalence operator which is again written in two different ways, you have these three lines or you have a double headed arrow.

So propositional logic, building blocks are propositions then uses these logical connectors to combine and talk about combinations of propositions.

(Refer Slide Time: 05:22)

Atomic propositions

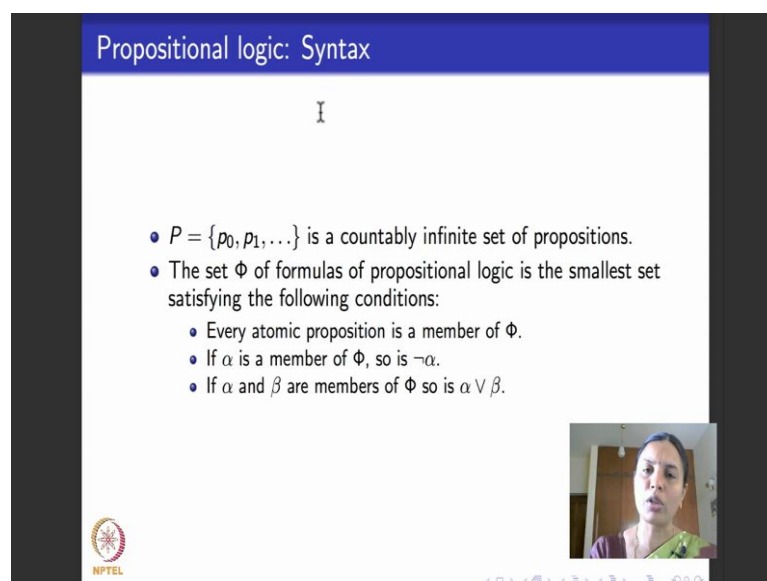
- Propositions are the basic building blocks of logic.
- An atomic proposition or just a proposition is a declarative sentence that is either true or false but not both.
- Examples of propositions:
 - New Delhi is the capital of India.
 - $2 + 3 = 5$.
 - $3 + 3 = 8$.
 - Today is Friday.

NPTEL

So, what are propositions? As I told you they are basic building blocks of propositional logic. An atomic proposition or a proposition is just an entity that is either true or false and there is clarity about when it is true and when it is false. But at no point in time, it can both be true or false and at no point in time, it can neither be true nor be false. So, here are some simple examples of atomic propositions, so the sentence which says; New Delhi is the capital of India, is an atomic proposition and we know that because Delhi is the capital of India; this proposition is true. And similarly, this statement which says $2 + 3 = 5$; that is if you add 2 and 3 you get 5 is an atomic proposition because it always evaluates to be true or false. In this case it evaluates to be true.

The next statement $3 + 3 = 8$; another statement about addition is another atomic proposition, but in this case the atomic proposition is false because we know that in the world of numbers that we deal with $3 + 3$ is not equal to 8. And the last one, today is Friday is another atomic proposition. It is true every Friday and on days that are not Fridays, it is false.

(Refer Slide Time: 06:33)



The slide is titled "Propositional logic: Syntax" in a blue header. Below the title, the letter 'I' is centered. A bulleted list follows, defining the syntax of propositional logic. The first bullet states that $P = \{p_0, p_1, \dots\}$ is a countably infinite set of propositions. The second bullet states that the set Φ of formulas of propositional logic is the smallest set satisfying three conditions: every atomic proposition is a member of Φ ; if α is a member of Φ , then $\neg\alpha$ is also a member; and if α and β are members of Φ , then $\alpha \vee \beta$ is also a member. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner.

- $P = \{p_0, p_1, \dots\}$ is a countably infinite set of propositions.
- The set Φ of formulas of propositional logic is the smallest set satisfying the following conditions:
 - Every atomic proposition is a member of Φ .
 - If α is a member of Φ , so is $\neg\alpha$.
 - If α and β are members of Φ so is $\alpha \vee \beta$.

So, in propositional logic, we begin with what is called as syntax of propositional logic. So, what is syntax of propositional logic? It assumes that we have a set of propositions written like this P as p_0, p_1, p_2 and so on. What sort of a set is this? We assume that the number of propositions that are available to this set for us is infinite and countably infinite. Countably infinite means it can take the propositions and enumerate them one after the other.

So, I have as many propositions as I need for use. So we begin with set of countably infinite propositions, typically represented by p naught, p_0, p_1, q, r and so on and then I define the set of formulas of propositional logic. What are the set of formulas for propositional logic? It is a smallest set that contains every atomic proposition and inductively, if it contains a formula α ; then it also contains negation of α , read this as $\neg \alpha$. If it contains formulas α and β , then it also contains $\alpha \vee \beta$. So, this is how I define all the formulas of propositional logic. It so turns out that these three entities; the atomic propositions, negation operation and disjunction operation or “or” are enough to define all the formulas of propositional logic.

So, you might wonder that I gave all these as also connectors and implies if and only if, but when I defined the syntax here, we use only not and or. What happened to the rest? The rest are what can be derived from using not and or operators.

(Refer Slide Time: 08:17)

Derived operators

- And: $\alpha \wedge \beta: \neg(\neg\alpha \vee \neg\beta)$
- Implies: $\alpha \supset \beta: \neg\alpha \vee \beta$
- Lff: $\alpha \equiv \beta: (\alpha \supset \beta) \wedge (\beta \supset \alpha)$.

The slide includes the NPTEL logo in the bottom left corner and a small video feed of a presenter in the bottom right corner.

So, and can be written as $\alpha \wedge \beta$ is nothing but negation of not negation alpha or negation beta. $\alpha \wedge \beta$ is nothing but $\neg(\neg\alpha \vee \neg\beta)$; alpha implies beta can be defined as not alpha or beta. Alpha if and only if beta or alpha equivalent to beta can be defined as alpha implies beta and beta implies alpha.

(Refer Slide Time: 08:48)

Propositional logic formulas: Examples

- You cannot ride the roller coaster if you are under 4 feet tall unless you are older than 16 years.
 $(r \wedge \neg s) \supset \neg q$, where q , r and s represent "You can ride the roller coaster", "You are under 4 feet tall" and "You are older than 16 years" respectively.
- Maria will find a good job when she learns Software Testing.
 $p \supset q$, where p and q represent "Maria learns Software Testing" and "Maria will find a good job" respectively.

The slide includes the NPTEL logo in the bottom left corner and a small video feed of a presenter in the bottom right corner.

So, moving on, here are some examples of propositional logic formulas. So here is a simple sentence which says that you cannot ride a roller coaster if you are under 4 feet tall or unless you are older than 16 years. So, how do I represent it as a propositional

logic formula? First I have to figure out what the atomic propositions in these entities are. So if you see there are three statements. The first statement can be thought of as this; you cannot ride the roller coaster, second statement is you are under 4 feet tall, third statement is you are older than 16 years.

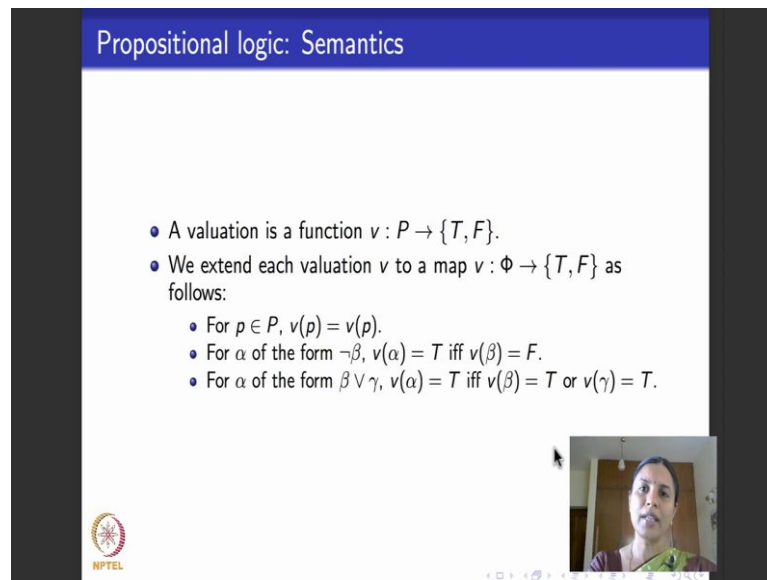
So, give them names, let us say q represents you can ride the roller coaster, you could say $\neg q$ represents you cannot ride the roller coaster. We just change the negation appropriately, there will be no difference. Let us say r represents you are under 4 feet tall; the second phrase and s represents you are older than 16 years; the third phrase. So, now what does this formula say? Take the first phrase which is $\neg q$ which is here; it says you cannot ride the roller coaster if; if is there and you cannot. So, there is an if and a not; if and a not can be this, α implies β , read it as if α then β .

So, if α then β is same as α implies β , so to be able to express you cannot ride the roller coaster if you are under 4 feet tall, I need to use an implication sentence. But then there is also one more atomic proposition here which is being older than 16 years and then there is an unless connective here. Unless means that you have to be older than 16 years, if you are under 4 feet tall to be able to ride a roller coaster. So, if I try to use the connectors that we learnt so far and try to write it as a propositional logic formula, this is the formula that I will get. What does it say? It says that you are under 4 feet tall or and you are older than 16 years and when you are not older than 16 years sorry implies you cannot ride the roller coaster.

So, I will repeat it again you are under 4 feet tall, so, r is true and you are not older than 16 years, which means you are younger than 16 years. These two implies that $\neg q$ is true. What does $\neg q$ say? $\neg q$ says, you cannot ride the roller coaster because we have instantiated q to be you can ride the roller coaster.

So, here is one more example. This example says Maria will find a good job when she learns software testing. So, how many atomic propositions can be created out of this sentence: one atomic proposition is which says Maria will find a good job, the second is if she learns software testing, some Maria learns software testing. So, I say p represents Maria learns software testing, q represents Maria will find a good job, then the sentence p implies q will mean Maria learns software testing if Maria finds a good job.

(Refer Slide Time: 11:44)



Propositional logic: Semantics

- A valuation is a function $v : P \rightarrow \{T, F\}$.
- We extend each valuation v to a map $v : \Phi \rightarrow \{T, F\}$ as follows:
 - For $p \in P$, $v(p) = v(p)$.
 - For α of the form $\neg\beta$, $v(\alpha) = T$ iff $v(\beta) = F$.
 - For α of the form $\beta \vee \gamma$, $v(\alpha) = T$ iff $v(\beta) = T$ or $v(\gamma) = T$.

NPTEL

So, now syntax tells you how to write formulas, how to build formulas. Now, what is the semantics of propositional logic. What does semantics mean? Semantics mean what is the meaning of these formulas. For example, if I go back and look at these two formulas r and not s implies not q and I look at p implies q , how do I know what they mean? Are the formulas true, are the formulas false, how will I know? How will I infer that? That is what semantics is all about.

So, semantics, we begin with a valuation function which takes every atomic proposition P and tells me whether it is true or false. So, valuation is the function from the set P of atomic propositions to the sets T and F , where T stands for true and F stands for false. Throughout this lecture, in the lecture for logic, I will use the term T for true and F for false. These are Boolean constants. Now, we extend valuation what other parts were there in the syntax of propositional logic? Every atomic proposition was there and then all these operators--- negation, disjunction and then derived operators conjunction, implication and equivalence.

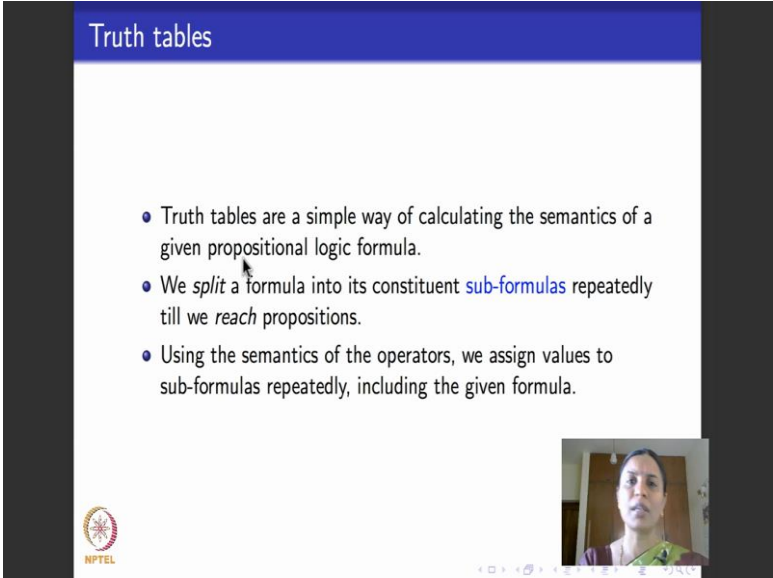
So, once I define, once I start with the valuation function that tells me whether every atomic proposition is true or false, I can go ahead and extend the valuation function to the set of all formulas Φ and it make each formula of Φ become true or false. So, extension of valuation is a map v that takes a set of all formulas Φ and each formula ϕ

whether it tells true or false by extending the valuation function v that takes the proposition and tells me whether each proposition is true or false. How is that extended?

So, if α is of the form not β , then v of α is true if and only if v of β is false. So, it says that if α is of the form negation β , then α is true if and only if β is false. If β is false then not β will be true so that is why α is true. Now if α is of the form β or γ then, α is true if and only if β is true or γ is true. Please note that this is not an either or statement, it is just a plain or. That means, that if both β and γ turn out to be true, then α can be true also. The only time when α is false is when both β and γ become false.

So, I have not given you the semantics of other three operators. But it is easy to infer what they are because we have given semantics of not and or and other three operators can be defined using them. Or, you could directly write the semantics of these operators. For example, when will α and β be true in under valuation function? If both α and β , both individually evaluate to true.

(Refer Slide Time: 14:33)



The slide is titled "Truth tables" in a blue header. It contains three bullet points:

- Truth tables are a simple way of calculating the semantics of a given propositional logic formula.
- We *split* a formula into its constituent *sub-formulas* repeatedly till we *reach* propositions.
- Using the semantics of the operators, we assign values to sub-formulas repeatedly, including the given formula.

In the bottom right corner, there is a small video inset showing a woman with dark hair, wearing a purple top, speaking. The NPTEL logo is visible in the bottom left corner of the slide.

So, later very soon I will tell you what truth tables are and then we will see what each of these semantics mean in terms of truth tables. Some of you might find this notation a bit cumbersome: to read valuation as a function from the set of propositions to true and false. But most of you might be familiar with the notion of truth tables. What are truth

tables? Truth tables are a simple way of calculating the semantics of a propositional logic formula.

So, what we do is we take a formula and we split the formula or break the formula into its constituent sub formulas repeatedly till we reach what are called atomic propositions, and when I reach atomic propositions, I have my valuation function which tells me when each proposition will become true or false. Then, I use the semantics of the Boolean operators and of the negation operators and work my way up till I know whether the entire formula is true or false. I have put these words split and reach in italics because if you have to do it properly, then you need to be able to parse the formula, generate the parse tree and the truth table works inductively bottom up beginning from the leaf all the way till the root of the parse tree which contains the formulas.

(Refer Slide Time: 15:46)

Truth tables

Truth table for \neg :

p	$\neg p$
T	F
F	T

Truth table for \wedge :

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

NPTEL

So, what are the various truth tables for the elementary connectors of Boolean logic? So, here are the truth tables. How does the truth table for not look like? Remember the valuation for the negation function not, if p is true, not p is false and if p is false, not p becomes true; that is what this table says. How does the truth table for and look like? For and it is a binary connective, so it takes two operands p and q. When is p and q true? p and q is true, if both p and q are true. That is what this first row says, if p is true and q is true then p and q is true. In all other places, p could be true, q could be false, p could be

false, q could be true, both p and q could be false, in all the other three cases p and q evaluates to be false because one of p or q is false.

(Refer Slide Time: 16:37)

Truth tables		
Truth table for \vee :	p	$p \vee q$
	q	
	T	T
	T	T
Truth table for \supset :	p	$p \supset q$
	q	
	T	T
	T	F
Truth table for \equiv :	p	$p \equiv q$
	q	
	T	T
	T	F

So, for or, we saw the semantics of valuation function p or q is true; if one of them is true or if both of them are true. So, if p is true, q is true p or q is true that is this first row; if p is true; q is false p or q is true that is second row; if p is false and q is true, p or q is again true because in all three cases one of p or q is true. But, when both p and q are false, p or q becomes false again. What is the truth table for implication? If you go back to the slide which defines the meaning of implication, it says α implies β is defined as not α or β . Or if you want to understand it in English; read α implies β as if α then β .

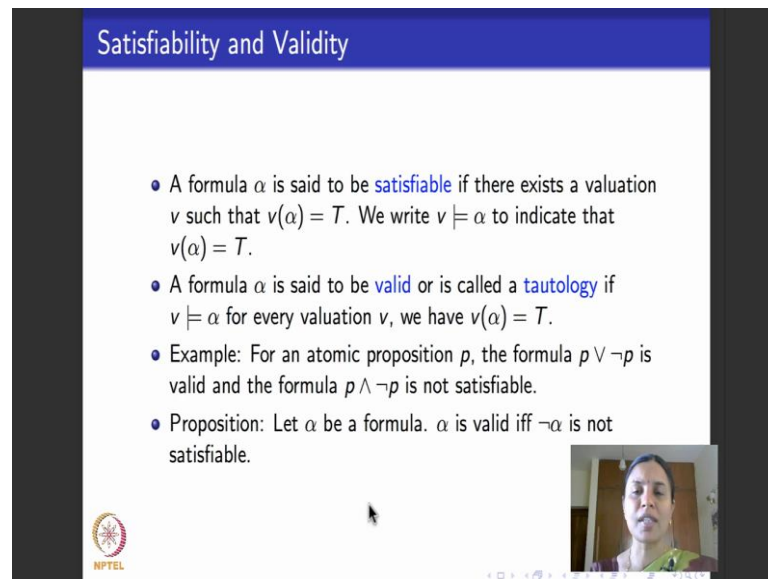
So, if I use that meaning then this is the truth table for implication. If p is true then and q is true then, obviously p implies q is true, because p is true q is also true, so this is true. So, if p is true and q is false; then p implies q will be false because it cannot be the case that p is true and p implies q will be true when q is false. If p is false then we do not really worry whether q is true or false, we say p implies q is true in a trivial sense. So, when both the cases when p is false and q is true or false, we say p implies q is true trivially.

What is the truth table for equivalence? The truth table for equivalence says that p equivalent to q is true if p and q have the same truth value. That is they both are true

together, which is this first row of the table or they both are false together, which is the last row of the table.

The second row and the third row, one of p or q turn out to be false, so p is not equivalent to q . In other words p equivalent to q itself turns out to be false.

(Refer Slide Time: 18:43)



Satisfiability and Validity

- A formula α is said to be **satisfiable** if there exists a valuation v such that $v(\alpha) = T$. We write $v \models \alpha$ to indicate that $v(\alpha) = T$.
- A formula α is said to be **valid** or is called a **tautology** if $v \models \alpha$ for every valuation v , we have $v(\alpha) = T$.
- Example: For an atomic proposition p , the formula $p \vee \neg p$ is valid and the formula $p \wedge \neg p$ is not satisfiable.
- Proposition: Let α be a formula. α is valid iff $\neg\alpha$ is not satisfiable.

The slide also features a small video inset of a person in the bottom right corner and a logo in the bottom left corner.

Now, we move on to the notions of satisfiability and validity. Why are these important? These are important because I told you that propositional logic formulas or predicate logic formulas are going to come as labels of decision statements in programs and when I have a predicate as a label of a decision statement in a program, I am evaluating the predicate by substituting some values for the variables that occur in the predicate and I am checking whether the predicate becomes true or false. If it becomes true, then the decision statement takes one path, if it becomes false then the corresponding decision statement takes another path.

So, in logic we call this as the problem of satisfiability. The problem of satisfiability involves checking whether a given logical formula evaluates to true or not. So, for propositional logic also, the problem of satisfiability checks whether given a formula α , is there a valuation function v such that v makes α true? That is v of α is true; in logic we write like this, we write $v \models \alpha$ read this entity as satisfies α to indicate that v of α is true. Please read this notation as v satisfies α .

The notion contrapositive to satisfiability is what is called validity you might have heard about it. We say that a formula is valid, if for every valuation v it becomes true. So we say α is valid if no matter what truth values you assign to the atomic propositions in α , α always becomes true. Another term for valid formula is what is called a tautology. The exact opposite of valid formulae are what are called contradictions, which mean that no matter what truth values you assign to atomic formulas, the formula will never be satisfiable; that is it will always be false.

So, here is a simple example of a formula that is valid and of a formula that is a contradiction. Consider an atomic proposition p . A formula of the form p or not p is always valid, why? Because if p is true then not p will be false, but this is a or, so the whole thing will be true. On the other hand, if p is false then not p will become true, again because this is a or, the whole thing becomes true. So, the p or not p is a formula that will always become true irrespective of whether p is true or false.

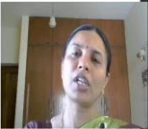

Now, consider a formula of the form p and not p . If you see here p and the not p will never be satisfiable, no matter what p is because if p is true then not p will become false and because it is an and here; the whole thing will be false. Similarly, if p itself is false then because it is an and here p and not p will be false. So, no matter what p is whether it is true or whether it is false; p and never p ; p and not p will always be a contradiction.

Here is a small result. It says that consider a formula α ; α is valid if and only if not α is not satisfiable. So, it says validity and satisfiability are what are called contrapositives of each other. The proof is very simple, but we would not really need it for this course, so I am leaving the proof without doing it. So, the only concept that you need to remember mainly from this slide for the rest of the course is to understand what satisfiability is. We say a formula is satisfiable if there is at least one valuation for the atomic propositions in the formula that make the entire formula true. Typically, the most common way of checking satisfiability is to be able to use truth tables.

(Refer Slide Time: 22:24)

Satisfiability problem of propositional logic

- To check if a formula α is satisfiable, construct the truth table for α and check if there is an entry (valuations of propositions) that makes α true.
- This algorithm takes time exponential in the length (number of symbols) of α .
- Satisfiability problem for propositional logic is NP-complete.





(Refer Slide Time: 22:29)

Satisfiability through truth tables: Example

Truth table for the formula $(r \wedge \neg s) \supset \neg q$.

r	s	q	$r \vee s$	$(r \vee s) \vee \neg q$
T	T	T	T	T
T	T	F	T	T
T	F	T	T	T
T	F	F	T	T
F	T	T	T	T
F	T	F	T	T
F	F	T	F	F
F	F	F	F	T



So, here is an example of how to check satisfiability using truth tables. So, you consider this formula r or s or not q , so here is a truth table for this formula. So, what I have done is, I have given true, false assignment to all these values. So, there are three atomic propositions here. So if I consider the possible combinations of true, false values to each of them, there will be 8 different combinations 2^3 is 8. So, r , s and q all three of them take true or r and s become true, q becomes false; r is true, s is false, q is true and so on. Now, I will first evaluate r or s which is, I use the semantics only for these two

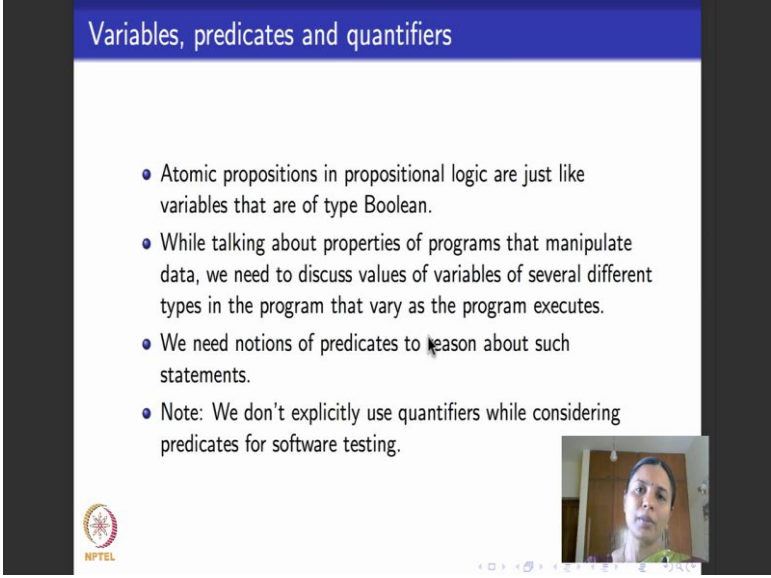
rows and the semantics of r to fill up this row with true false values. If you see in the first six rows, one of r or s will always be true here, so r or s throughout becomes true.

The last two rows both r and s are false, so r or s is false. Now, I do this r or s or not q ; I take r or s , then I take the row the column for q ; negate the column which I have not shown in the truth table here. So, if you want to be perfect, you could negate this column and then I apply or again. If I do that; then I will get all these values to be true and this value to be false, so this is how truth tables work.

Now, to check if a formula α is satisfiable, what I do is I generate the truth table and I check if there is at least one row in the last column of the truth table which corresponds to the formula, where the formula evaluates to be true. If there is one such row, then it means there is an assignment of true-false values to the variables of the formula that make the formula true. So in which case the formula is satisfiable. So if you see what is the running time of such an algorithm? I have not really described the algorithm.

But what is the running time of such an algorithm? The running time of this algorithm is going to be exponential in the number of symbols of α because I told you; suppose there are three different variables, then each of them will take two different truth values and the number of combinations is going to be 2^n . So, the number of rows in the truth table in the worst case is going to be 2^n and so the algorithm runs in exponential in the number of symbols or the number of atomic propositions within. In general, it is known that the satisfiability problem for propositional logic is NP-complete. So, NP-complete means we do not know of a polynomial time algorithm till date that we can use to solve satisfiability.

(Refer Slide Time: 25:12)



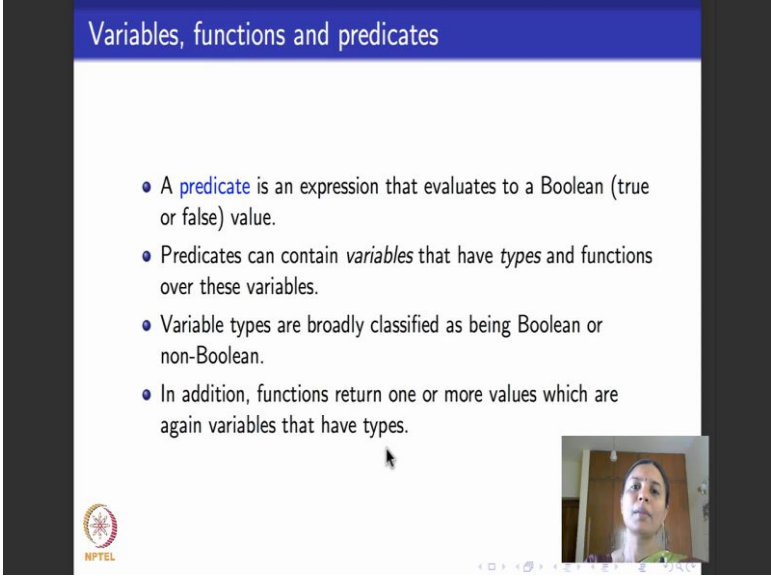
The slide has a blue header with the title "Variables, predicates and quantifiers". Below the header, there is a list of four bullet points. In the bottom right corner, there is a small video inset showing a woman with dark hair, wearing a green top, speaking. The NPTEL logo is visible in the bottom left corner of the slide area.

- Atomic propositions in propositional logic are just like variables that are of type Boolean.
- While talking about properties of programs that manipulate data, we need to discuss values of variables of several different types in the program that vary as the program executes.
- We need notions of predicates to reason about such statements.
- Note: We don't explicitly use quantifiers while considering predicates for software testing.

Now, that was a basic introduction to propositional logic. But as I told you in the beginning of this module, what we would need is what is called predicate logic. What are predicate logics? Predicate logics are used to define predicates which come as labels of decision statements and programs. Predicates have variables of all different kinds. There could be integer variables, they could be floating point variables, there could be functions evaluating certain things. All of them need not be just Boolean variables like in propositional logic.

So, we need to be able to move on. So, atomic propositions and propositional logic just define Boolean entities. But, when we talk about programs that manipulate data, we encounter other kinds of variables, so we need notions of predicates to reason about such statements. As I told you in the beginning of this lecture, strictly speaking, predicate logic also deals with quantifiers for all and there exists, but we will not need that for testing; so I am staying away from introducing them to you.

(Refer Slide Time: 26:13)



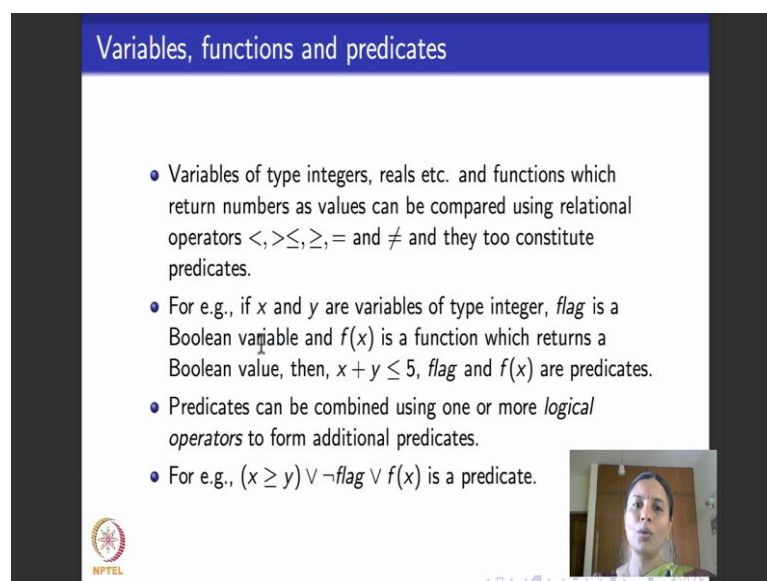
The slide has a blue header with the text "Variables, functions and predicates". Below the header, there is a list of four bullet points. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner.

- A **predicate** is an expression that evaluates to a Boolean (true or false) value.
- Predicates can contain *variables* that have *types* and functions over these variables.
- Variable types are broadly classified as being Boolean or non-Boolean.
- In addition, functions return one or more values which are again variables that have types.

So, what is a predicate? For our purposes a predicate can be thought of as an expression that always evaluates to true or false; because that is what we need as far as the course is concerned. Now, predicates can contain variables like we find them in programs and each variable could have different type. There could be integer type variables, there could be strings, there could be floating point numbers. Predicates also can contain functions. Like for example, you would agree with me that it is not very uncommon to see a statement which says that if \log of x is less than 0.1 then you do something.

So, what is \log of x ? \log of x is a function that takes x and evaluates \log of x and returns a number. So, predicates can contain function that return values of a certain type. We broadly classify variable types as Boolean and non Boolean because all non Boolean entities will be of one type as far as our semantics of predicates are concerned and functions return one or more values, which are again variables that have types.

(Refer Slide Time: 27:16)



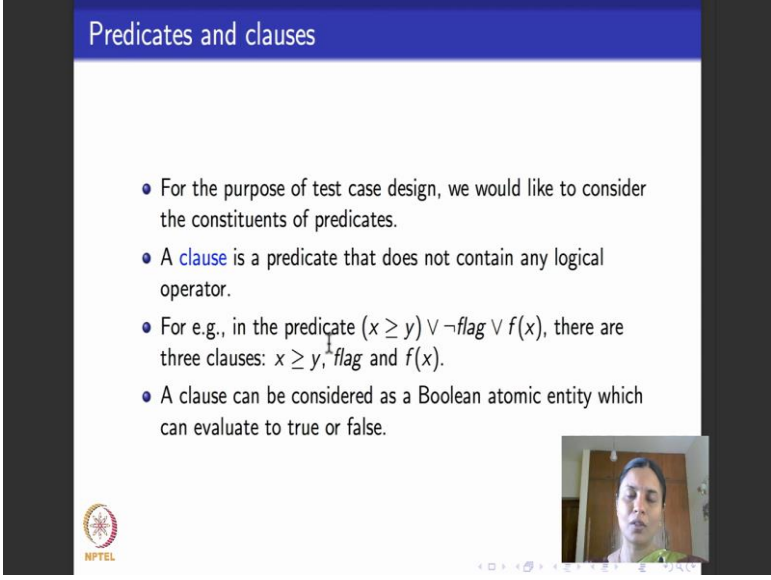
The slide is titled "Variables, functions and predicates" in a blue header. It contains a list of four bullet points. The first bullet point states that variables of type integers, reals, etc., and functions which return numbers as values can be compared using relational operators $<$, $>$, \leq , \geq , $=$, and \neq , and that they too constitute predicates. The second bullet point gives an example: if x and y are variables of type integer, $flag$ is a Boolean variable, and $f(x)$ is a function which returns a Boolean value, then $x + y \leq 5$, $flag$, and $f(x)$ are predicates. The third bullet point states that predicates can be combined using one or more logical operators to form additional predicates. The fourth bullet point gives an example: $(x \geq y) \vee \neg flag \vee f(x)$ is a predicate. In the bottom left corner, there is a small NPTEL logo. In the bottom right corner, there is a small video inset showing a woman speaking.

- Variables of type integers, reals etc. and functions which return numbers as values can be compared using relational operators $<$, $>$, \leq , \geq , $=$ and \neq and they too constitute predicates.
- For e.g., if x and y are variables of type integer, $flag$ is a Boolean variable and $f(x)$ is a function which returns a Boolean value, then, $x + y \leq 5$, $flag$ and $f(x)$ are predicates.
- Predicates can be combined using one or more *logical operators* to form additional predicates.
- For e.g., $(x \geq y) \vee \neg flag \vee f(x)$ is a predicate.

Now, you might have seen that in predicates that we use in programs, you would have used all these operators, so called relational operators. So, variables of types integers, real numbers and so on and functions which return numbers as values can be compared using the normal relational operators and numbers; less than, greater than, lesser than or equal to, greater than or equal to, equal to, not equal to and so on.

Like for example, if I have x and y as variables of type integer, $flag$ is a Boolean variable and f of x is a function which returns the Boolean value, then here are some examples of predicates. I can ask whether x plus y is less than or equal to 5; this entity x plus y will give me a number; 5 is another number; the whole predicate x plus y less than equal to 5 will return true or false. $flag$ itself is a Boolean variable; so it is true or false; f of x is a function which returns a Boolean value which is again true or false. So, all these are predicates. And, like we saw in propositional logic, each of these predicates can be combined using one or more logical operators. Like for example, if I had these predicates then here is a predicate that combines together. So, x is greater than equal to y or not a $flag$ or f of x . And the whole thing will evaluate to true or false because each of these entities will evaluate to true or false and I can use the semantics of the or operator to evaluate the meaning of the entire predicate.

(Refer Slide Time: 28:53)



Predicates and clauses

- For the purpose of test case design, we would like to consider the constituents of predicates.
- A **clause** is a predicate that does not contain any logical operator.
- For e.g., in the predicate $(x \geq y) \vee \neg flag \vee f(x)$, there are three clauses: $x \geq y$, $flag$ and $f(x)$.
- A clause can be considered as a Boolean atomic entity which can evaluate to true or false.

NPTEL

Small video inset showing a person speaking.

For the purposes of this course, to be able to define coverage criteria, I would need one more terminology. What we say is that each individual entity that comes without a Boolean operator in a predicate, we will call it a clause.

So, clause is a predicate that does not contain any logical operator. So, if I have a predicate that looks like this: x greater than equal to y or not $flag$ or f of x , then it has three clauses; one is x greater than equal to y , the other one is $flag$. Alternatively, you could say not $flag$ is also a clause, not a problem or you could say f of x . For our purposes, we say it does not contain any logical operator. So, we remove the not and just say a $flag$ is a clause. So a clause can be thought of as Boolean atomic predicate which always evaluates to true or false.

(Refer Slide Time: 29:45)

Satisfiability problem for predicate logic

- We say that a given predicate p is satisfiable if there is an assignment of values to its constituent clauses (in turn, variables and functions) that make p true.
- Satisfiability problem for predicate logic is undecidable.
- There are *SAT solvers* and *SMT solvers* that can check if a given predicate is satisfiable or not for many different kinds of predicates.

NPTEL

So, in the next lecture what we will do is, we will see how to define coverage criteria based on predicates and clauses. So, before I finish today's lecture, like we saw satisfiability problem for propositional logic, we also consider satisfiability problem for predicate logic for the same reason, because these predicates are going to come as labels of decision statements in the program and decisions in the program are taken based on whether these predicates are true or false.

Typically a program is evaluated given a set of values. A generalization of that problem is to ask given an arbitrary predicate is there at least one assignment of true, false values or is there at least one assignment of values to the variables of the corresponding types that make the predicate true or false. This is what is called as satisfiability problem for predicate logic. Unlike propositional logic, propositional logic we just assign true false values, build our way up using truth tables. You cannot do that for predicate logic because variables could be of different types and if it is a variable like integer then there are potentially an infinite number of values for which you have to check.

So, satisfiability problem for predicate logic is known to be undecidable. There are no algorithms in the general case that solve the satisfiability problem for predicate logic, but there what are called SAT solvers or SMT solvers that help you to do it. Please remember that when we use it in testing, even though we consider satisfiability problem, we are not looking for really one satisfying assignment. As and when program executes,

we check if the given assignment of values to the variables makes the predicate true. So, that is not really the satisfiability problem, but it is useful to know that the satisfiability problem for propositional logic is tough, it is NP-complete and the satisfiability problem for predicate logic in general is undecidable.

So in the next lecture, I will introduce you to coverage criteria based on predicates and clauses.

Thank you.