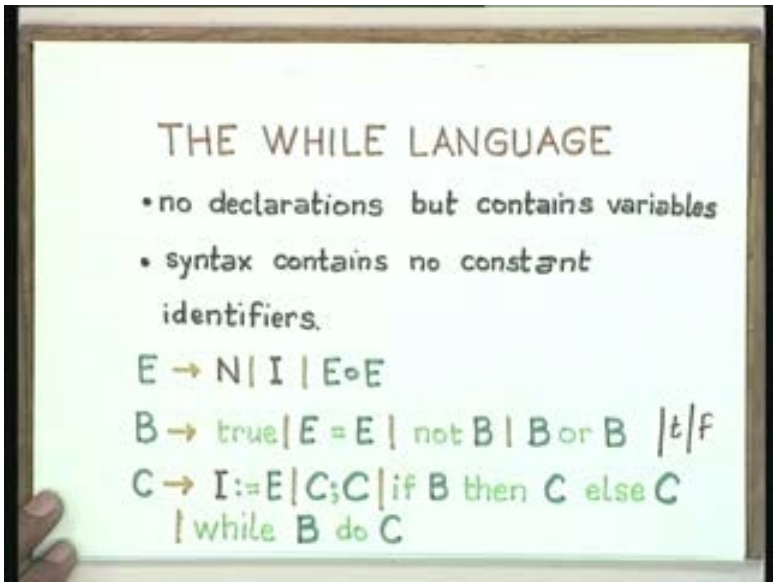
Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi
Lecture no 14
Lecture Title: Stores

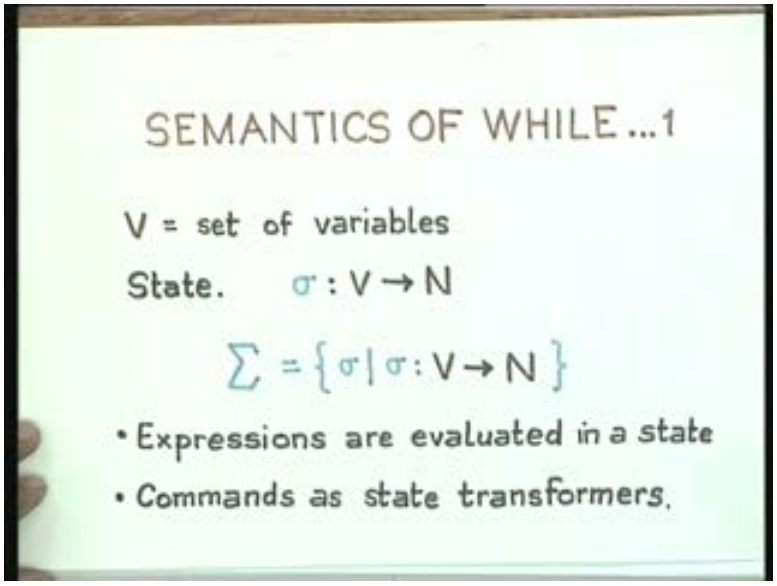
Welcome to lecture 14. We will quickly run through the semantics of the while programming language and then go on to what complicates matters. This is the syntax of the while programming language.

(Refer Slide Time: 00:50)



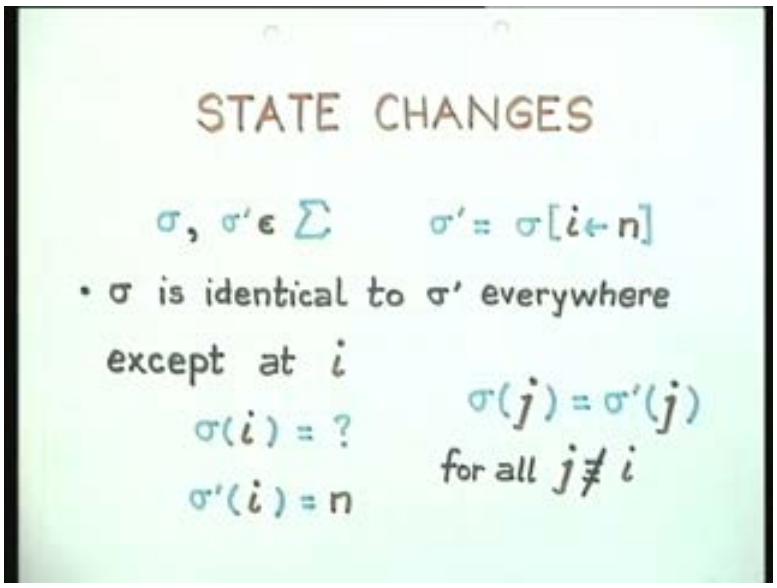
Most importantly, it consists of imperative commands which change some notion of state and I defined a state in a simplified form as a variable-value mapping. Sigma is the set of all states. We will assume an infinite set of variables available to us. Expressions are evaluated in a state and commands as state transformers.

(Refer Slide Time: 01:20)

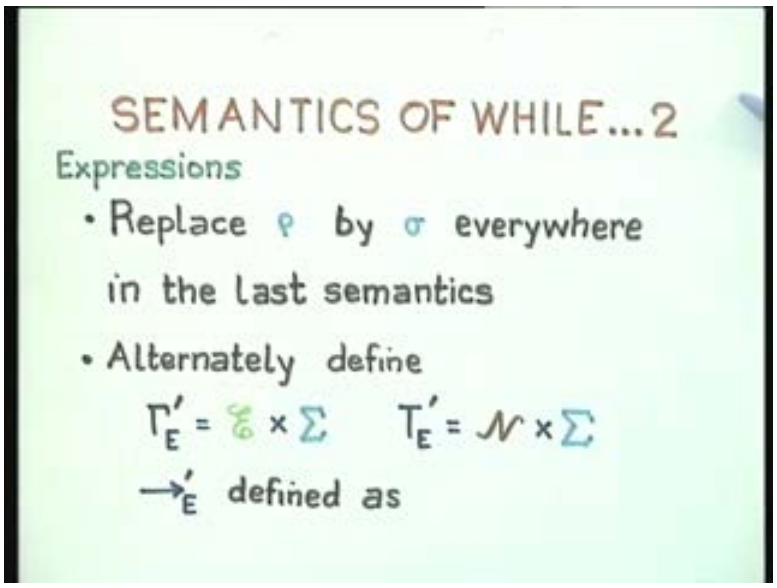


We will look at state changes. Given two states I defined a state updation which really corresponds to having a single assignment statement. This updation could be generalized also to multiple assignments.

(Refer Slide Time: 02:04)

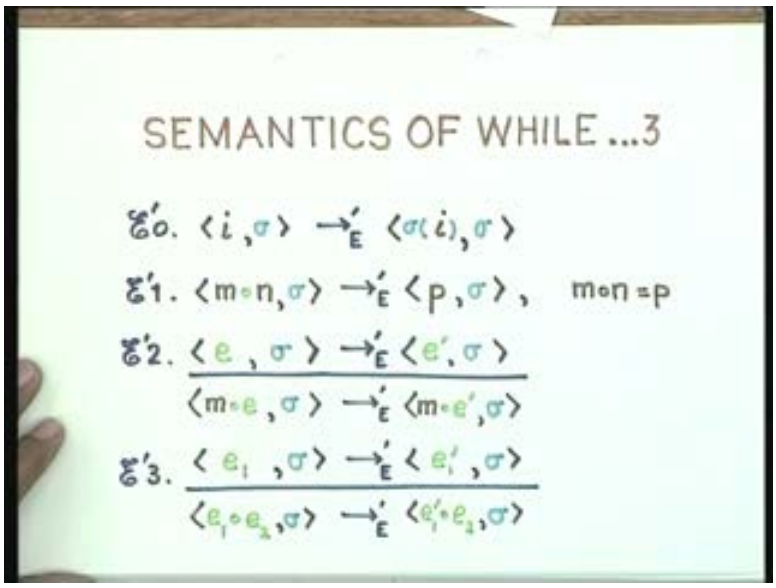


(Refer Slide Time: 02:20)



In the expression language what I said was that at this state at least there seems to be no real difference between environment and state. In the expression language at least there is no significant difference. So, you have the same kind of rules starting with the evaluation of identifiers in the state and the normal rules for expression evaluation which are strictly left to right sequential evaluations. Since our while language also contains Boolean expressions we required semantics of Boolean expressions. In a general programming language Boolean expressions would be treated on par with the normal expressions.

(Refer Slide Time: 02:46)



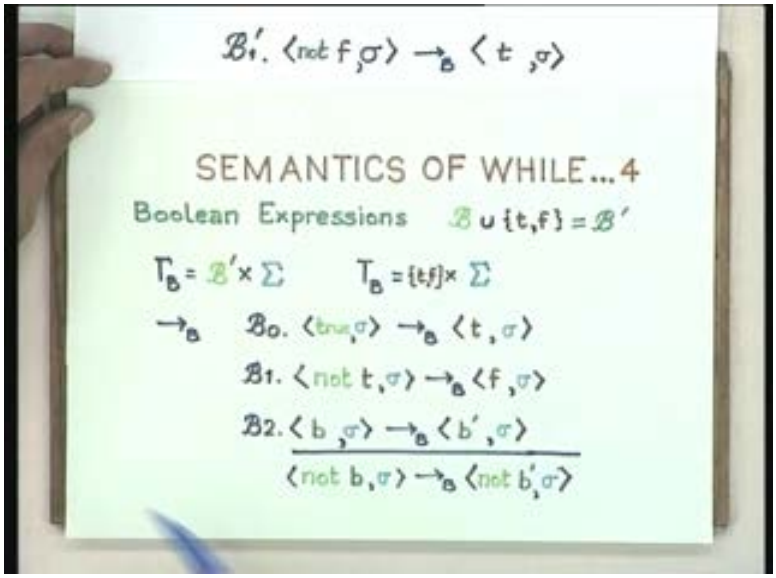
There is only a type difference in the sense that you would allow Boolean variables and assignments to them and whatever you can do with integer expressions you can similarly do with Boolean expressions. However, since this particular language is a sort of greatly simplified language and we did not allow Boolean variables, the Boolean expressions were used only as a via media in order to construct conditional commands.

This has this following semantics but even when the Boolean expressions are an integral part of the expression language the semantics would not change too much except that one would have to look at the type distinctions between the various kinds of variables and one would have to do something about types.

Here again we defined the evaluation of Boolean expressions in a left to right manner. This unary operator which we did not encounter in the expression language is of particular interest. The Boolean expression language actually assumes the existence of a Boolean algebra and semantics as defined by a Boolean algebra for Boolean expressions.

For example; the ground values t and f are not identical and unless you specify clearly that they are not identical you could have a one element Boolean algebra and all the properties of Boolean algebra could be satisfied.

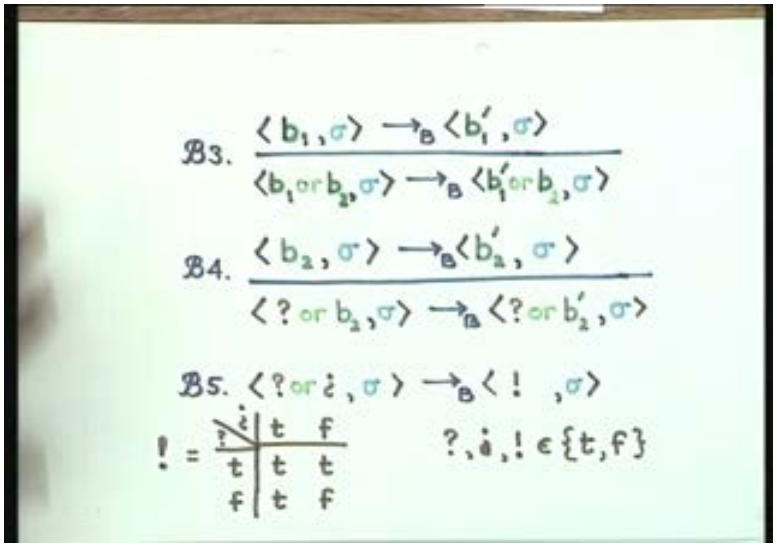
(Refer Slide Time: 05:17)



The fact they are not identical and they are actually compliments of each other is given by these two axioms B1 and B1'. Then we have these usual Boolean operations which essentially show a complete left to right evaluation. If you have a general Boolean expression of the form b1 or b2 then you would repeatedly apply b3 which means you would be evaluating the left operand till it reaches a truth value and then you would start evaluating the right operand. It shows a left to right evaluation and it always satisfies the standard truth table for a Boolean operation.

So, everything is subject to the existence of such a model of meaning in the Boolean expressions and since our Boolean expressions acted as some sort of intermediate stage between the expression language and the command language, we have to relate the Boolean expressions also to the derivations of the expression language of which we had only a simple operation; the equality relation on expressions.

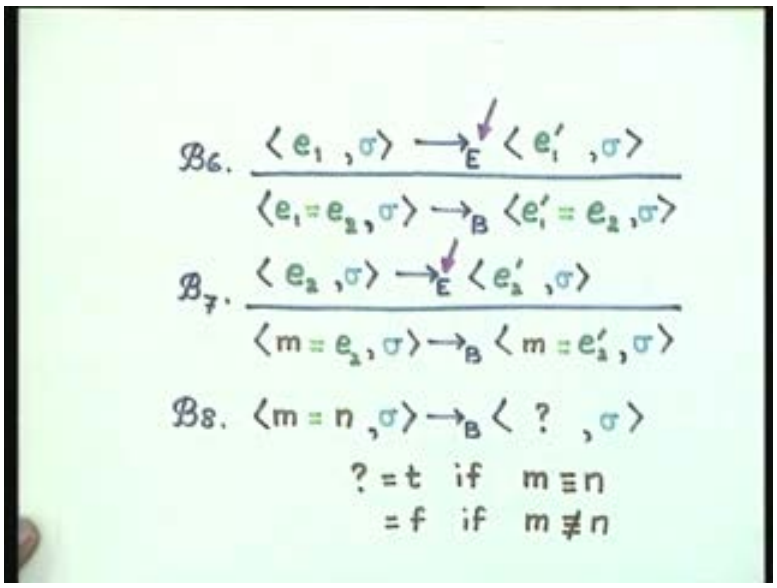
(Refer Slide Time: 06:04)



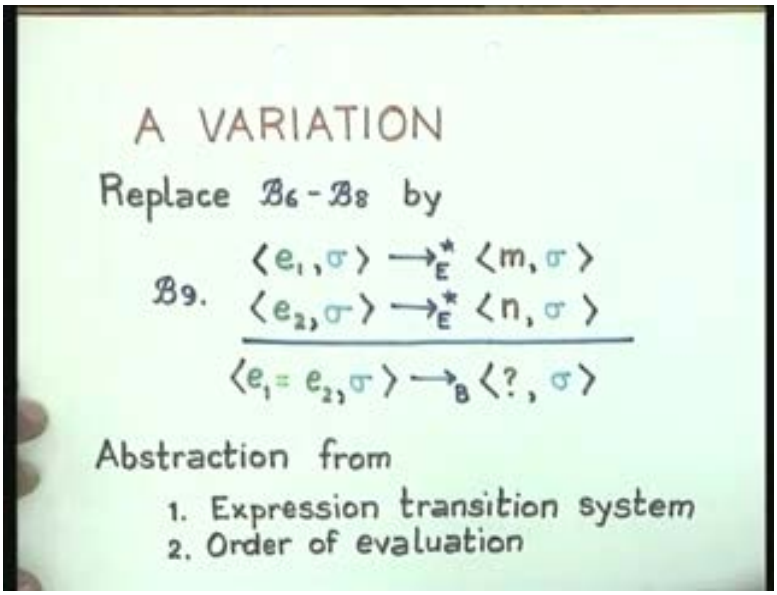
The rules B6 and B7 also give you a left to right evaluation of the equality.

You would apply B6 several times which means you would evaluate the left operand of the equality till it converges to some integer value, then you would evaluate the right operand till that also converges to some value and you would check the equality of these two based on the fact that if m and n (since they are there in the underlying machine as patterns) are identical as patterns then it would be true, otherwise it would be false.

(Refer Slide Time: 07:30)



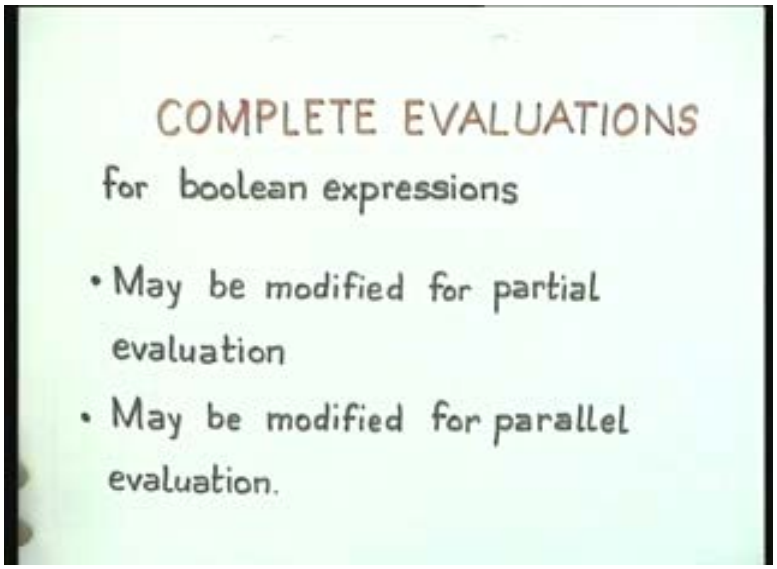
(Refer Slide Time: 08:25)



We do not need to go through this whole process; we could for example abstract away from the intermediate stages and simply give a fairly non deterministic rule which says that if e1 can evaluate in a finite number of steps (this star indicates a finite number of steps 0 or more to some integer value) and if e2 can be evaluated in a finite number of steps to n then this can be concluded depending upon these patterns m and n as given by

this. This rule does a further abstraction from the order of evaluation in the transition system and you could modify these Boolean expression evaluations. For example; you could have a strict right to left order of evaluation or you could have a partial evaluation where you could give rules for partial evaluation especially the 'OR' operation, for example or any other binary operation.

(Refer Slide Time: 09:00)



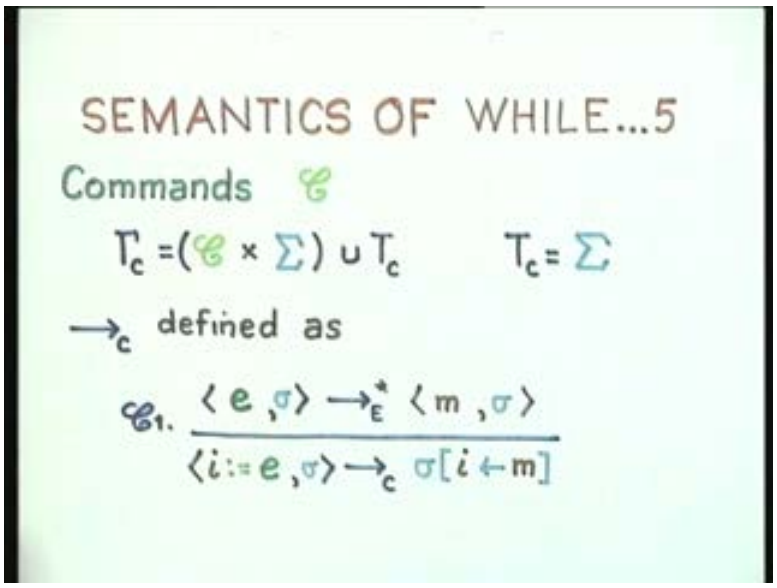
In particular if the left operand evaluated to the truth value t then in a single transition the entire Boolean operation could go to t so that you do not have to evaluate the right operand of the 'OR'. You could also do parallel evaluation or parallel partial evaluation.
So, there are many variations depending on how you want to specify the language. Then we came to the most important aspect and that is commands and I defined this set of configurations.

As I said commands change states and the first rule of the command itself tells you this one important difference between states and environments and that is that in an environment you can have a re-declaration. Whatever modifications you make are re-declarations and redefinitions of given identifiers. However, we had said that the little environments the declarations created are only intermediate stages in a larger computation.

A declaration does not stand by itself. An expression stands by itself especially in a functional language. The expression is evaluated in an environment and in the process of its evaluation little environments are created and destroyed. So, the evaluation of the expression from start to finish goes through some intermediate stages of the creation of new environments in which there might be redefinition of identifiers but the whole point is that those changes in the environment are reversible in the sense that whatever gets declared also goes out at the end.

In the case of states we are talking about capturing a general notion of an irreversible change inside the value of the identifier. Even if symbolically it did not look too different, conceptually at least there is an important difference which we have to capture somehow and one of the reasons why we will introduce stores is to capture this conceptual difference between environments and states. The assignment statement is an example of a change that is irreversible.

(Refer Slide Time: 12:05)

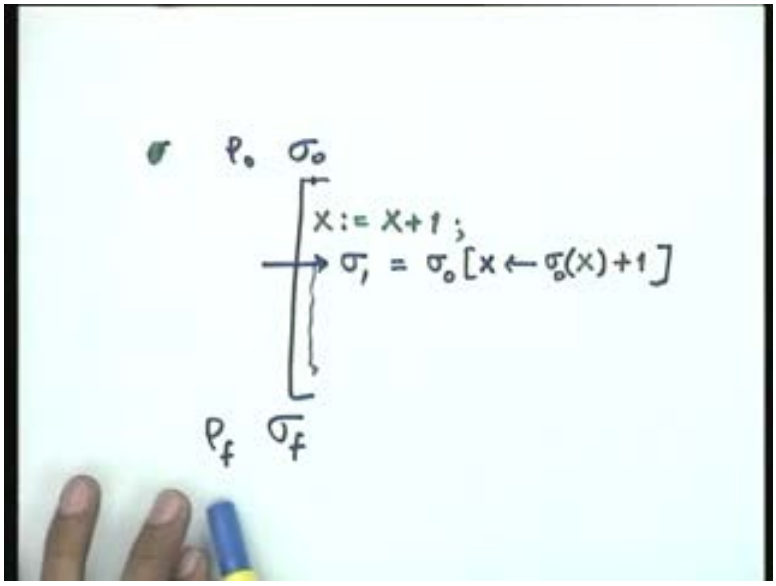


Suppose you have some command which is going to be evaluated in a state starting from some initial state sigma. Let us have some large command with some assignment here

and something else. When this is evaluated in a state sigma starting from a state σ NAUGHT at the end of this you get a new state σ 1 which is different from σ NAUGHT in the sense that x now has the value which is one more than the x given by σ NAUGHT and this state change is irreversible in the sense that it is starting from this state that further state changes take place as you execute the command and you finally end up with some final state say, σ f at the end of this command. In the case of an environment you could have had inside declarations which created little environments but at the end of it you still had the original environment with the same name-value bindings that you started of with.

In that sense all the changes that occurred inside in the evaluation of an expression were all reversible whereas in the case of state changes and in general in an imperative language we are looking at some concept which allows for irreversible changes to be made. The σ f need not be the same as σ NAUGHT even for the value x unless an equal amount of work was done to get x back to its original value in σ NAUGHT. Even if your command had been such that it restored all values back to the original and σ f was the same as σ NAUGHT, those changes were irreversible in the sense that the amount of effort required to undo the initial changes that you made, was as much as the amount of effort you required to make the changes in the first place. Whereas in the case of environment there is something automatic about the way the new environments that are created get destroyed at the end.

(Refer Slide Time: 15:40)

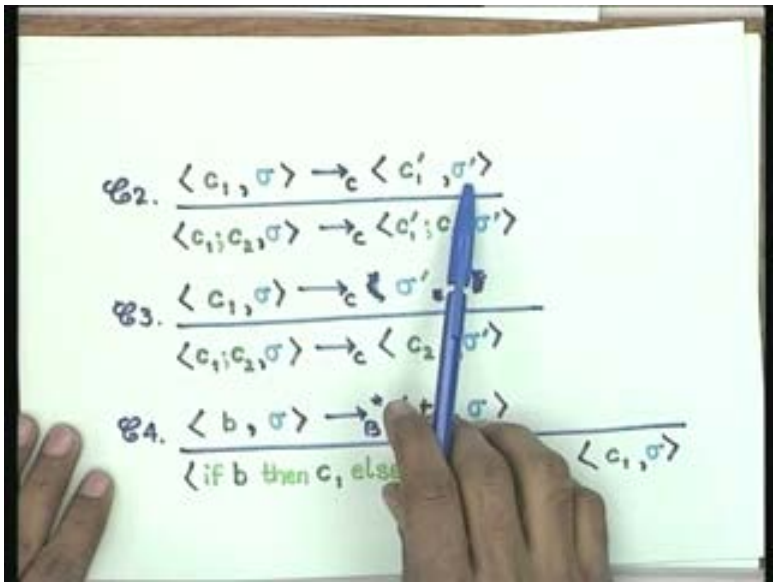


So, there is something conceptually different between states and environments which we should capture. A notion as abstract a state does not really capture this difference totally. But it does capture to this extent in the sense that you can see from the semantics that state changes occur and a final state need not be the same as the initial state. We will come to the notion of a store but let me first go through this while language. I showed a sequential evaluation of the sequential composition operation semicolon and so you evaluate the left operand of the semicolon till it reduces to having made just a state change and there is nothing more left of the command.

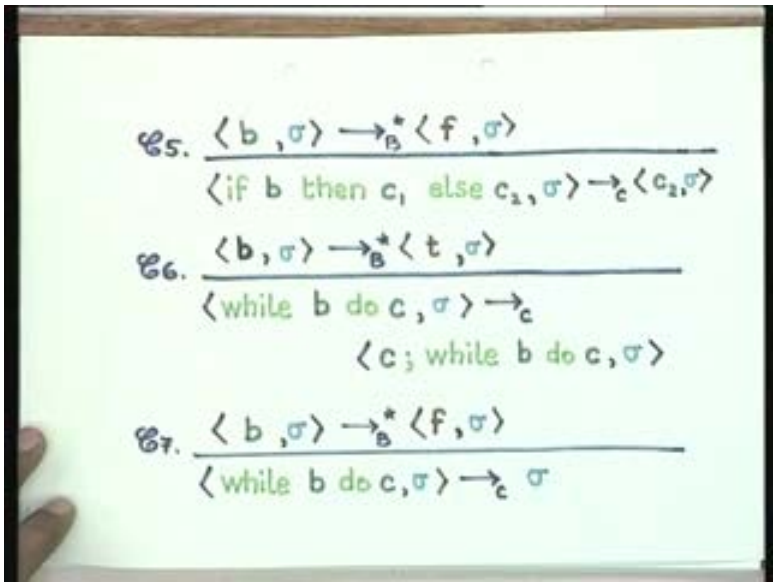
Commands get consumed and reflected as state changes and then you start evaluating the right operand. You could evaluate the right operand and in the case of the conditional we evaluate the Boolean expression and we transfer control. The transfer of control is like getting rid of one arm of the conditional depending upon the truth value of the Boolean and lastly, we had this while loop and of course there is something about the ‘while’ loop which is not quite correct, meaning it does not quite meet with the philosophy we started out with.

After the execution of the ‘while’ loop the state does change but c6 says that this moves in one step to a program which has this form and the sequential composition rules tell you that there could be state changes. For example; you have some c1; c2 which is really what the ‘c; while b do c’ is like. This is c1; this is your c2 and c1 could create state changes and could finally terminate and then you would evaluate c2 and this new state σ’.

(Refer Slide Time: 18:35)



(Refer Slide Time: 20:05)

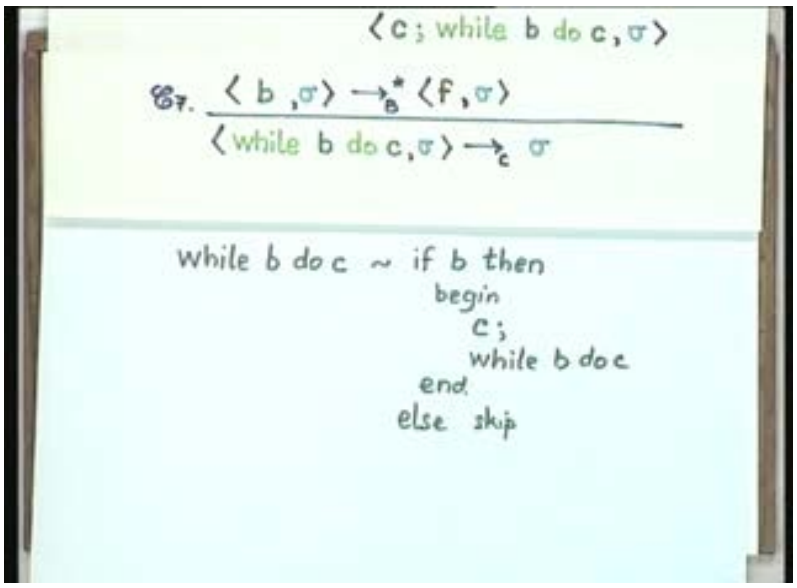


It is not as though there are no state changes but there is something that is seriously wrong with this ‘while’ loop. It conforms to our intuition about how while loops are executed but it does not conform to some other aspect of our semantics and notion of semantics. What is the most important feature about semantics, syntax directed translation and recursive descent parsing? What is common to all these definitions? This is what you get after having programmed enough in a while language but there is something wrong with it. The final expression is even more complicated then the initial expression we started out with which means that this definition is not inductive. Several applications of this might only give you more and more complicated results. What is wrong with you recursive definition?

Let us take a function on numbers the f (n) = 0 if n = 0 otherwise it is f (n+1) – 1.
What is wrong with this definition? What is wrong with it mathematically and computationally? Mathematically, it is just an equation which has to be solved for f and there is a solution too. The solution to this equation is f (n) = n as a mathematical equation. What is wrong with it is not mathematical. What is wrong with it is computational in the sense that it is not inductive. This semantics of the ‘while’ suffers from the same problem.

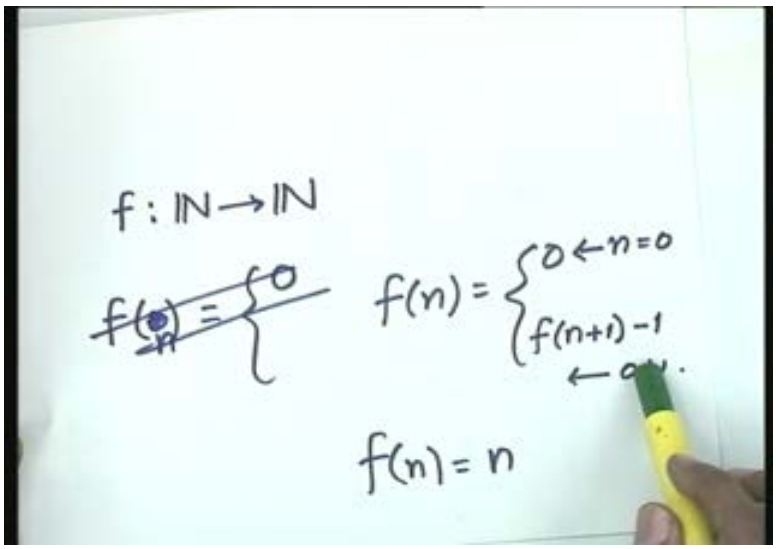
For example; if you take c6 and c7 then it means that in any state ‘while b do c is equivalent to if b then (I will use begin and end because I require to bracket) begin c; while b do c end’. So, c6 and c7 are mathematically quite meaningful in the sense that all that they say is that this is equivalent to this. If you had a ‘no’ operation in your language you could also write this as else skip. It just gives you an equation which is perfectly meaningful and which is why implementations actually work this way.

(Refer Slide Time: 24:30)



It is perfectly meaningful but however it is not inductive precisely because the resulting expression is much more complex structurally than the original expression. But the fact that while loops are executed and they give you meaningful results means that there must be a way of giving a semantics which is inductive. There is a very simple solution to make it inductive just like there is a very simple solution to make this computationally more meaningful. You could either just give this non recursive definition or you could even give a recursive definition which is computationally more meaningful. You may think that this should be the semantics of the 'while' loop except that it is not inductive. It has led to an important semantical notion which should come out as part of your operational semantics or any kind of semantics is program equivalence.

(Refer Slide Time: 25:27)



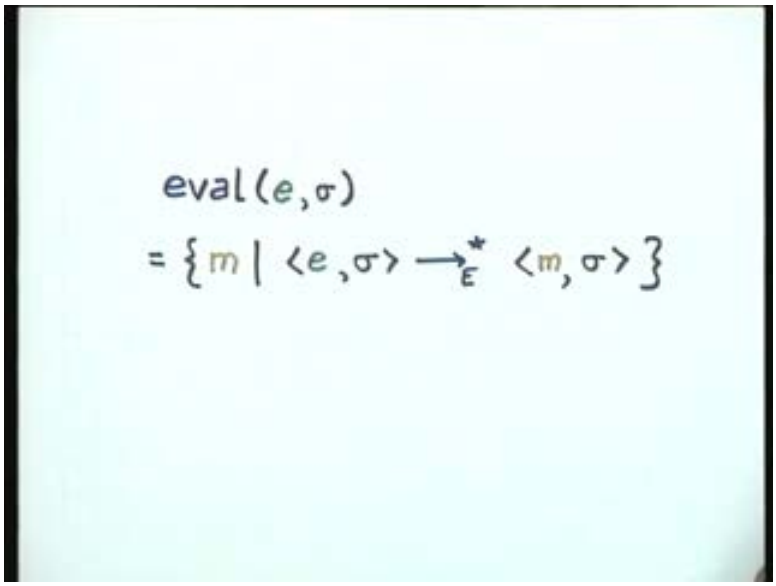
Let us look at program equivalence in the context of this language and let us start with expressions. Two expressions are equivalent if and only if for every state sigma in which they are both evaluated they yield the same values. I can define a function which denotes evaluation, eval (e, σ). Unless you have explicitly proved it, there is no reason to believe that evaluating an expression gives you a single answer. Until you have proved it you cannot be sure. We will define 'eval' of e sigma as being equal to the set of all values m that eventually lead to a terminal configuration.

In the expression language, terminal configurations are of this kind. Unless and until you have proved it you do not know whether expression evaluation actually terminates. You have to prove if all expression evaluations terminate. Supposing you have not proved it then you cannot assume that expression evaluations do terminate in which case this set would be empty.

There might be some evaluations of this expression which might terminate and some evaluations which may not terminate. In which case let us just for simplicity assume that the evaluation of this expression consists of just the set of values that are reached on termination which is not quite correct. We could also introduce a new fictitious element

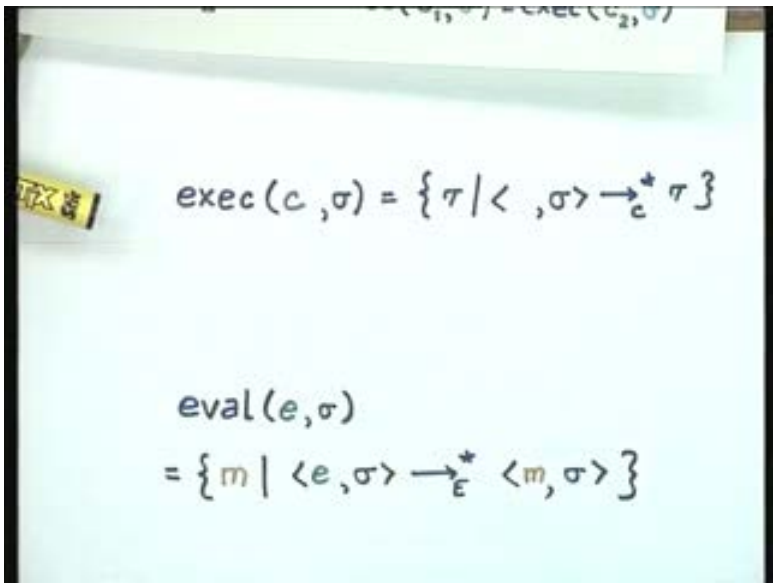
called some undefined which represents non termination and we could define the set as the set of all possible evaluations which yields proper values and an undefined value. Whatever may be your definition of this ‘eval’ we of course have to consider only a finite number of steps in the evaluation.

(Refer Slide Time: 29:57)



We would say that two expressions are semantically equivalent provided in all possible states the evaluations of these two expressions yield identical sets of values. Similarly, in the pervious model of a functional programming language with environments, we could think of it as evaluating these expressions in all possible environments. We would say that two expressions are equivalent only if in all possible environments in which they are evaluated they yield the same set of values and the same holds for Boolean expressions.

(Refer Slide Time: 32:15)



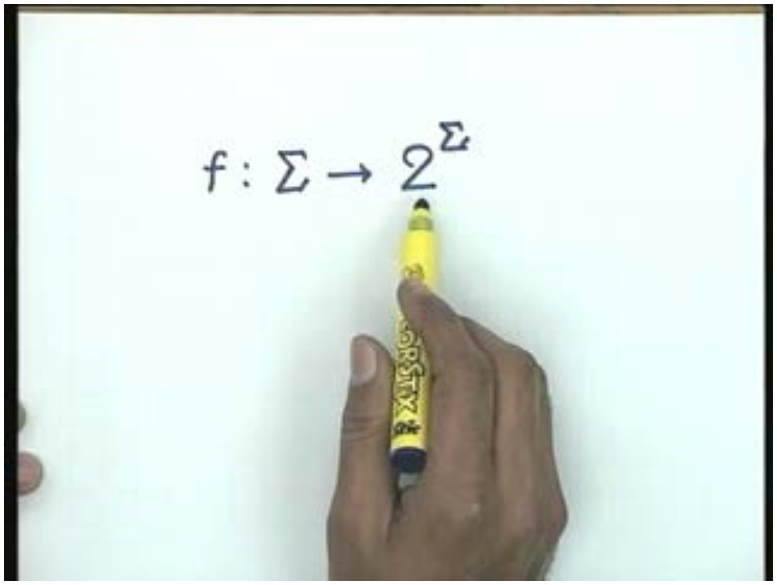
When are two programs equivalent? I will define the notion of an execution in which I will say that in our simplified programming language a command is also a program so there is no distinction really. I can define this function called executions and I can define it as a set of all final states. So, I can define this concept of an execution of a command or a program in our case and I would also call this a behavior. The behavior of an expression is just the set of all possible values it can yield which is what our definition of 'eval' does and the behavior of a command is just the set of all possible final states that it can yield given an initial state. When are two programs equivalent? Two programs are equivalent only when for every initial state in which you start executing these two programs c1 and c2, they yield the same final states at the same set of final states. In this case you do not know unless you have proved that a program will actually for a given input state yield only one output state.

There are non deterministic languages which do not necessarily satisfy the state to state functionality. Our semantics actually gives us a way of defining program equivalence and what does program equivalence finally work out to? What does Boolean expression equivalence finally work out to? It works out to just all the laws of Boolean algebra.

Program equivalence just works out to equality as functions. Think of functions of this form.

This is the power set of states. Note that we have not yet proved that a program would give only a unique final state, it could give several possible final states so typically if you want to regard a program as a function, then you have to regard it as a function from states to the power set of states. So, two programs are equivalent if they represent the same function. It should be possible to use the semantics to reduce it to two functions.
Given two programs c1 and c2 they represent corresponding functions f1 and f2 and if c1 and c2 are equivalent, it should be possible somehow to prove that f1 is the same function as f2.

(Refer Slide Time: 34:05)



One initial promise that I made of being able to define program equivalence and behaviors is essentially fulfilled by these definitions and you can generalize this. As we go along in each case you can define behavior in such a fashion that it represents some functions. It does not make much sense to really talk about the behavior of a declaration but we could define similar notions for declarations.

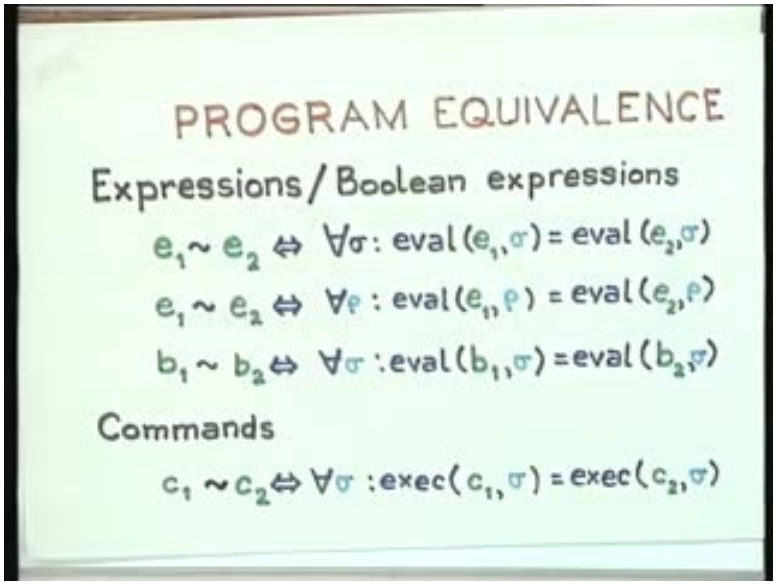
We can talk about two declarations being equivalent except that there is one problem associated with that. In the case of a declaration a new environment is built and that new environment is some name-value bindings.

Suppose I decide that in analogy to whatever we have done, the behavior of a declaration is just the set of new environments it creates when it reaches a terminal configuration, then I go further and say that two declarations are semantically equivalent if and only if they create the same new environment. Now declarations might occur within expressions. One reason why we had to talk about the behavior of expressions is because our programs cannot be semantically equivalent till we have actually defined an adequate notion of equivalence of expression. Since our commands depend upon expressions you have to define a notion of semantic equivalence in the case of expressions. What can happen in the case of program equivalence or in the case of expression equivalence?
What is inadequate about this notion of declaration equivalence? Just assume an expression language with declarations like the ones we have already done. Two expressions are equivalent if and only if under all environments they yield the same values.

Let us look at this. I define two declarations to be equivalent if and only if they yield the same environment. Suppose I define declaration equipments as two declarations d1 and d2. They are equivalent if and only if for all initial environments row (let me call this a new function called elaboration; we usually talk about elaborating a declaration in an environment) you could say that two declarations are equivalent, where of course this function elaboration elab (d, p) is defined as the set of all little environments row prime such that in the environment row if d is elaborated, then in a finite number of steps it gives you a new environment row prime. Of course I am forced to use this set because we have not actually yet proved it but it can be proved that an elaboration at least with the syntax and the semantics that you have given will give you a unique new environment row prime. So, with this definition of elaboration I can define this but while this might be correct it is somehow not adequate. Environments are temporary but that does not affect expression equivalence.

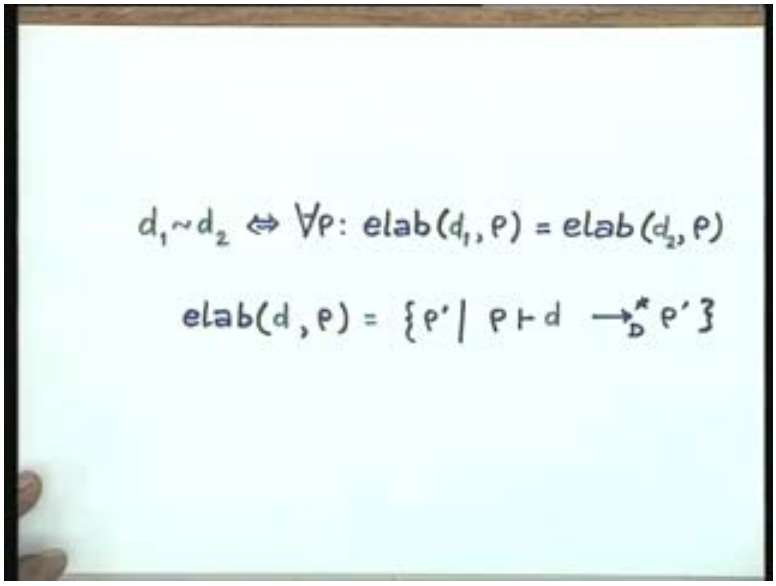
What does expression equivalence finally boil down to?

(Refer Slide Time: 42:20)



If you are talking about expression equality in say number theory then it means that two expressions will be equivalent only if their equivalents can somehow be justified by number theory eventually because when you evaluate you get these numbers or at least you get expressions on numbers on pure ground terms and you should be able to prove that those two numbers are the same and for that you will just be using pure mathematics and no programming.

(Refer Slide Time: 43:03)



We are considering their equality in the same environment. This universal qualifier says that you elaborate them both in the same environment and for all possible such same environments if those elaborations give you identical new environments then they are equal but this definition is not incorrect in the sense that you cannot give a counter example to it but it is weak in the sense that it does not really get us very far. Two expressions can be equivalent even if the declarations inside them are not equivalent. In that sense it is weak. Two expressions could work out to be equal but the declarations inside them could be completely different; they might create completely different environments and that happens in most cases.

For example; even if the declarations look identical they could still differ up to name changes. I could uniformly replace names in one declaration to give me another declaration and by this definition the two declarations will not be equivalent. This is equivalent to saying that when I have to submit my assignment I take a program and uniformly replace all the names; I draw up a table of names and I give corresponding different names and all meticulations are different but the two programs are still equivalent and the reason the two programs are still equivalent is that if I have made uniform name changes it is really because those names do not matter. They are part of the

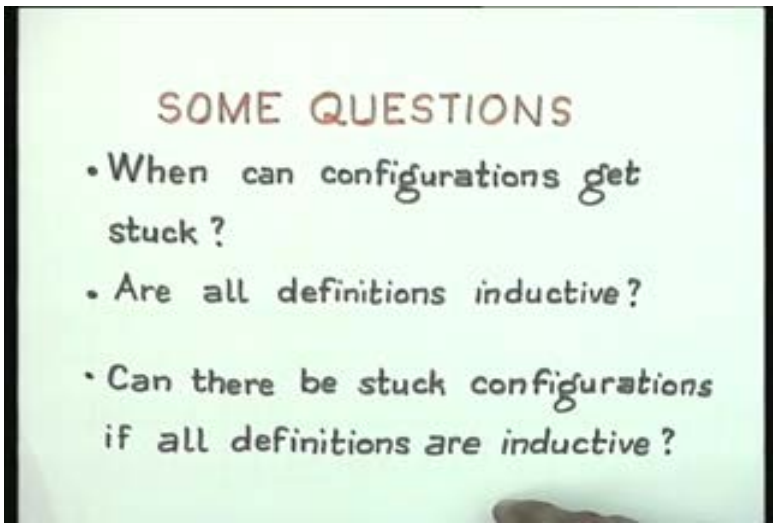
reversible changes which are very intermediate. Declaration is meant to perform various kinds of abstractions. Somebody might use less number of variables and more complicated expressions. Somebody might be able to deal with only very simple expressions and so uses more declarations. So, declarations could be different in various ways and the expressions could still be equivalent.

The equality of expressions does not necessarily depend upon the equivalence of declarations and these name changes are always done. That is one of the reasons why I should have a more sophisticated definition of equivalence to really bother about it though it can be done. There are other important questions with regard to whatever we have said about our semantics. We have to worry about when configurations get stuck and what is a stuck configuration?

A stuck configuration is one from which there is no movement. It is not a terminal configuration; it is a non terminal configuration from which there is no movement. Then you would say a configuration is stuck. Then of course there is this problem of the inductiveness of whether all our definitions are inductive and the while loop definitions are certainly not inductive. If your definitions are not all inductive then you do not know whether you have taken care of all the possible syntactical constructions. If your definitions follow the syntax at least you are sure that you have taken care of all possible syntax constructions. At the level of a context grammar at least you are sure that you have taken care of all possible cases.

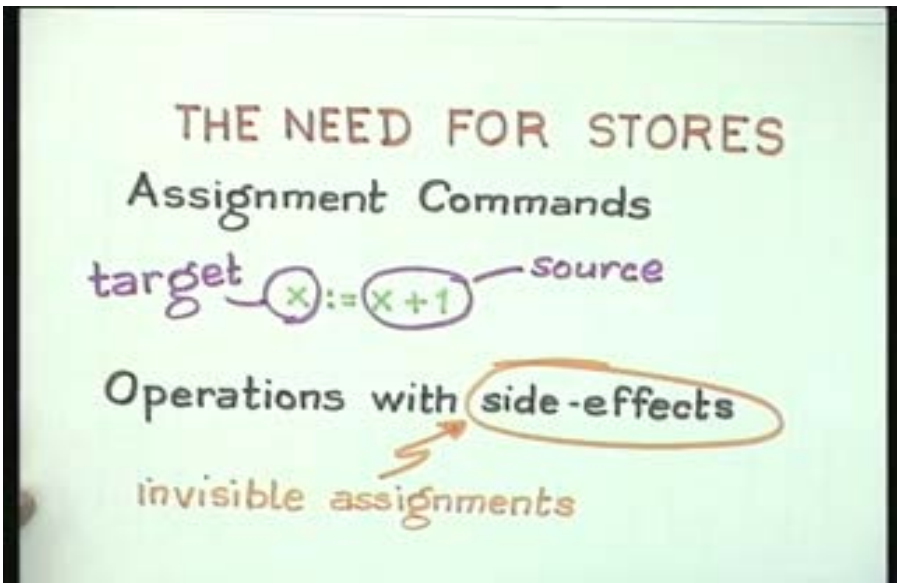
If your definitions are not inductive then there might be certain syntactical constructions which you have not taken care of and you have to explicitly prove that you have taken care of all possible syntactical constructions. If your definitions are not inductive you cannot even use induction in order to prove it. We will have to prove it by some other complicated means. Now can there be stuck configurations even if all definitions are inductive?

(Refer Slide Time: 48:30)



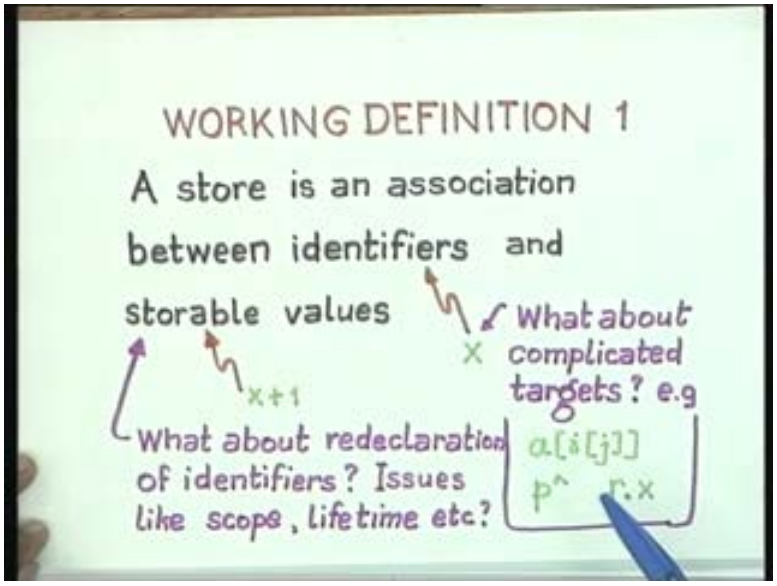
I will let you think about that. In the meanwhile let us talk about stores. Now it is clear that the environment and the state are really two different concepts and they try to capture two different concepts and ‘state’ is a very abstract concept and we have tried to model the assignment commands where we will consider the right hand side of the assignment to be a source and the left hand side to be a target but our assignment commands are very simple. For example, they did not allow side effects and when you talk about a side effect it is equivalent to some kind of an invisible assignment taking place somewhere without your knowledge. It is at least invisible outside some scope. Secondly, if you take our working definition as a store being on association of identifiers and values what happens when you have these complicated kinds of targets of assignments?

(Refer Slide Time: 49:16)



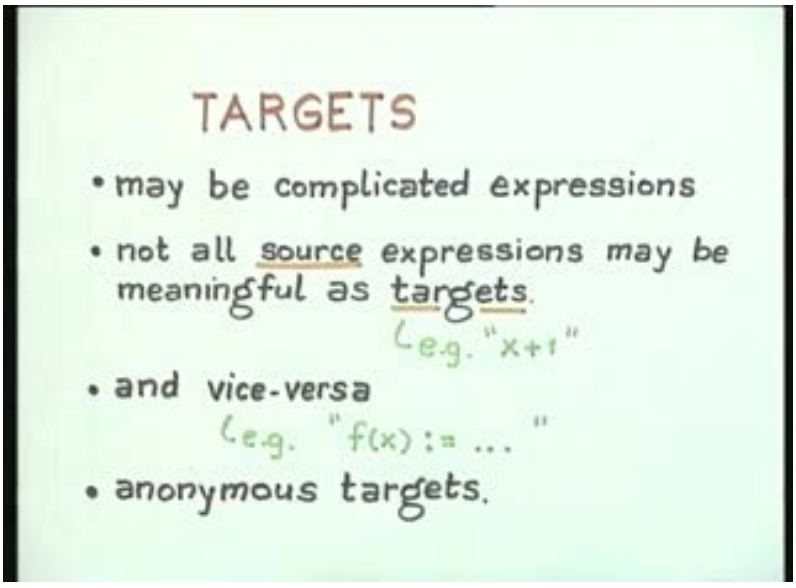
I can have for example an array which indexes into another array and I can update one array element in some complicated fashion through an indexing mechanism. I can have pointers. So, my targets are not necessarily just simple variables. They are not just identifiers. They are actually complicated expressions because the identifier p has some meaning and this is like an operation on p. The identifier ‘a’ has some meaning, ‘i’ has some meaning, ‘j’ has some meaning and this is some complicated ternary operation on a, i and j. Our targets are not necessarily just identifiers. They are actually expressions. Then there are further complications. What about when identifiers get re-declared? Can you just talk about a store as such an association?

(Refer Slide Time: 50:50)



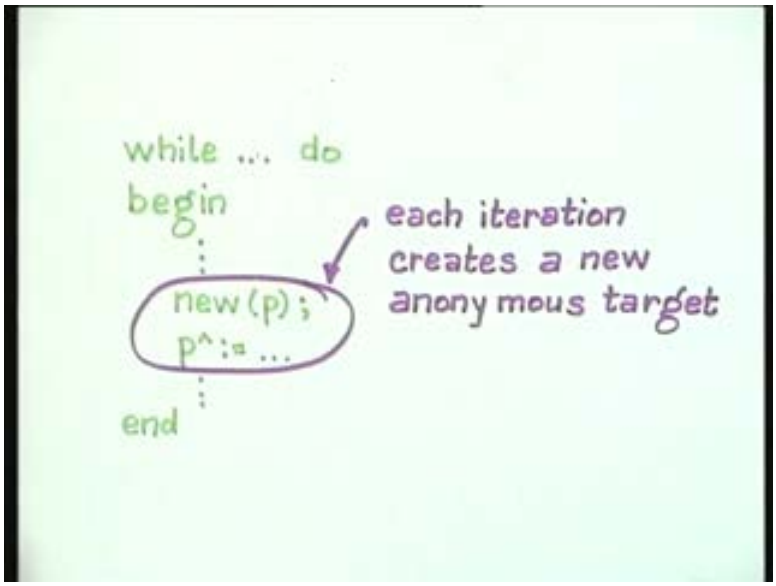
Is a re-declared identifier different from the identifier that was previously there? What about scope issues? Also not all target expressions might be meaningful. So, there is a separate subset of the expression language which targets can be and not all source expressions can be meaningful.

(Refer Slide Time: 51:09)



There is a separate language of source expressions; sublanguage. There are two subsets of target expressions; valid target expressions, valid source expressions and they are not disjoint. Then you also have anonymous targets.

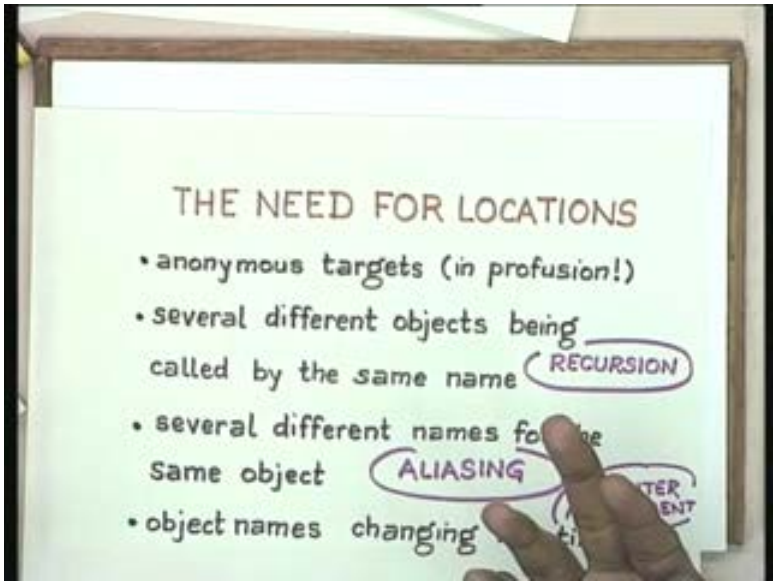
(Refer Slide Time: 51:52)



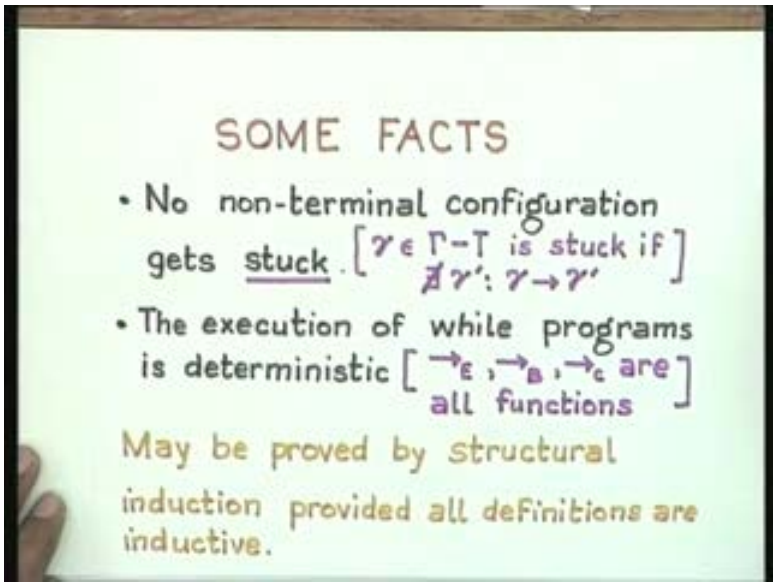
For example; if you had such a loop every time you create a new ‘p’ you are creating a new anonymous target. Actually you are creating a target which has some meaning. If you look upon this as a name it has the meaning there but then the next time you come around the loop the previous invocation is anonymous. It has no name. How are you going to look at the previous invocation?

So your environments themselves are not sufficient to worry about it and our abstract notion of a state is also not sufficient to look at complicated programming language behaviors. Further, when you have recursion you have the same identifier, the same name for several different logically distinct identifiers which have got nothing do with scope.
Then you have several different names for the same object when you have something like a call by value or when you have a call by reference, when you have the ‘war’ parameter declaration in languages. I can do a pointer assignment so that several different identifiers actually refer to the same object.

(Refer Slide Time: 53:25)



(Refer Slide Time: 53:49)



With recursion I can have several different objects having the same name. This means that we have to refine our notion of state to something that is more tenable. We will come out with the notion of a store. Looking upon states as just identifier-value bindings is simply not sufficient to account for most behaviors in real programming languages. We will define the notion of stores and then integrate that with the notion of environments.