Principles of Programming Languages

Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi

Lecture no 13
Lecture Title: Commands

Welcome to lecture 13. Let us quickly look at commands in a simplified setting. We have previously looked at expressions and declarations which are two important classes of syntactical categories with different meanings. I will introduce commands separately with expressions and then later we will integrate them with declarations.

For the purposes of this lecture we will just assume a 'while' language which is like the most skeletally complete structured language for programming. We will assume that we have a language with no declarations but we can have lots of identifiers and all those identifiers are variables. For the present while we are still discussing the various major syntactic categories, we will assume that our variables are only integer variables. There are no declarations but we will assume that we can freely update variables. Since there are no declarations we really cannot distinguish between a constant and a variable.

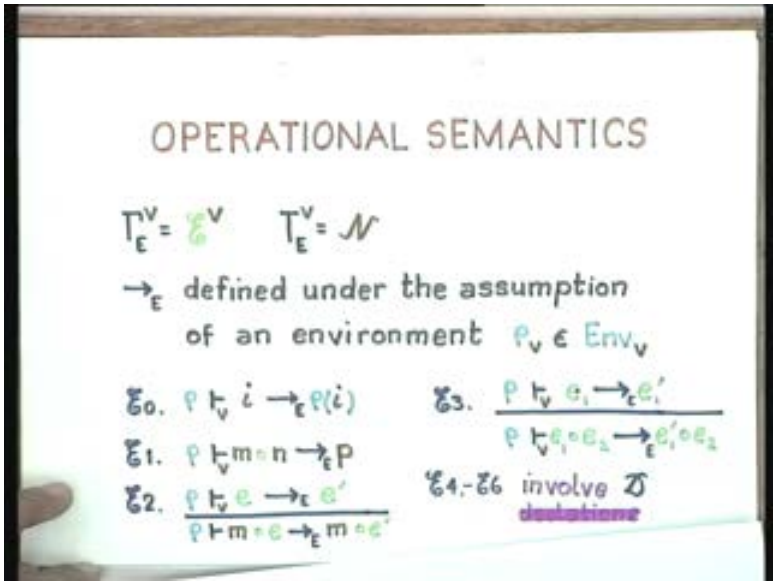
We will assume that there are no constants. Later, when we integrate declarations into this imperative language then we will also look at environments. For the present we will just assume that we have got this simple imperative language that consists of a language of expressions which have literals, integer literals, integer variables and the standard operations say the binary operations on integers.

A program in this language is just a command. I have written it in a bottom up fashion. A complete program is just a command and since we have conditions, here we require a sub language of Boolean expressions also. I will assume a simplified sub language of Boolean expressions and since there are no declarations I do not want to complicate matters by allowing Boolean variables. I will just assume that there is a Boolean constant

in the language called ‘true’ and the only kind of Boolean expression which consists of integer expressions is an equality checking expression. Otherwise you can have negotiation or disjunction. I just chose ‘OR’ because we have to choose one binary operator and the rest is simple; the one unary operator, one binary operator and one constant besides these equalities of expressions. Our main grammatical category is that of a command whose basis is an assignment statement.

There is a sequential composition of commands, as in a language like PASCAL. Then there are these conditional commands and looping commands. In the last few lectures we looked at an expression language with declarations so, it was very much like a functional language. Here we are looking mainly at an imperative language without declarations because in an imperative language the main syntactic categories are commands. If we have to give semantics of the simplified language we should in some sense be faithful to the previous operational semantics that we have already given for expressions.

(Refer Slide Time: 05:22)



Now since we are considering an imperative language with an assignment statement there is an updation of variables and we have to come up with a suitable concept which allows for updation of variables. However, we should not unnecessarily load our semantics with

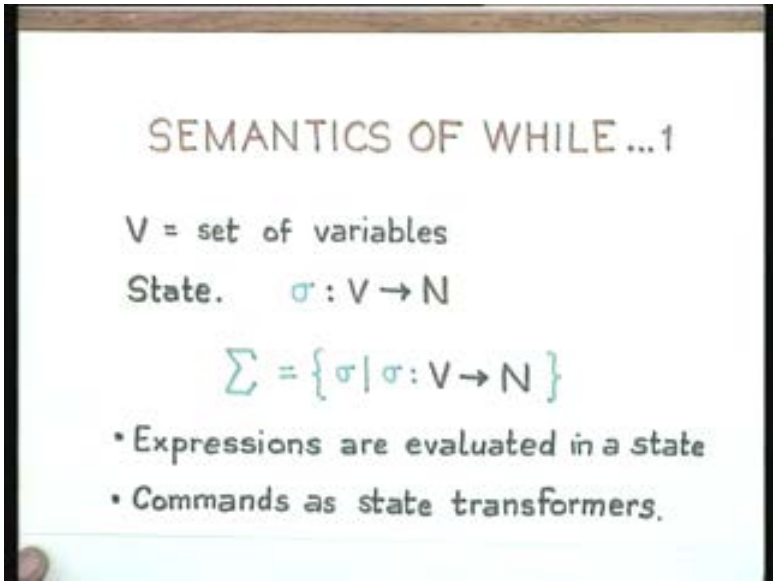
pragmatic features. We should not unnecessarily bring in data structures or algorithms but just give minimal concise rules, which will specify the language, in machine-independent and architecture-independent fashion.

We will do the concept of a state. In fact the most common way of describing imperative languages is that they are state based languages. There is a concept of state which is some abstract entity and for our purposes and for the purposes of most discussions on the pros and cons of imperative versus functional languages, a state is really nothing more than a variable-value binding. In that sense it looks no different from what we said about declarations in the functional language. They are also just identifier-value bindings.

The reason a state is different from an environment is that the variable-value binding really represents bindings that can change with time in the semantics. So, the semantics of the ‘while’ language is important because if you want to talk about a simplified imperative programming language, everybody would pick up a ‘while’ language and give it essentially just the state based semantics or define its meaning in a state based manner.

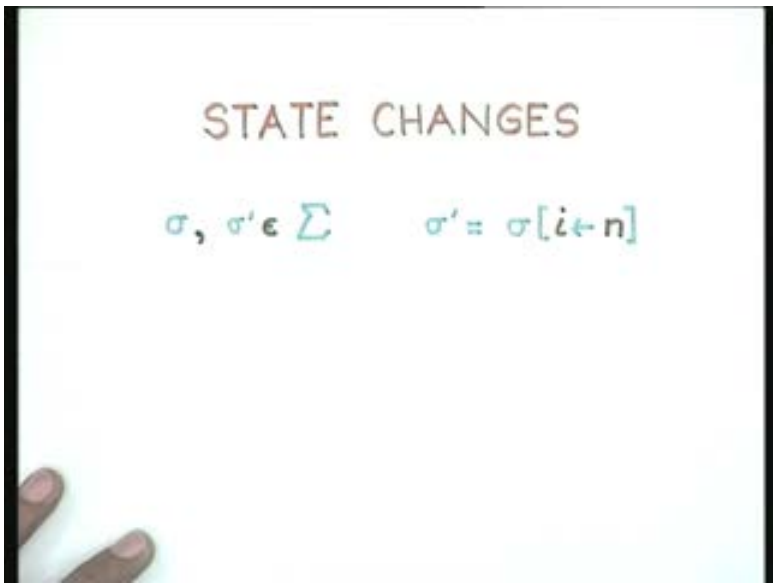
In all programming courses also the while language is considered the basic building block through which programming may start and you talk about states and changing a state. So, it is instructive to look at it. At the same time by looking at it in a simplified fashion we might also understand how to express the semantics of Boolean expressions and in general what happens in the case of commands. We will combine all of them in a single framework and then we will go on to actually looking at programming languages in which the notion of state is not sufficient and you require to augment it with a different concept which is closely related to it and closely related to environments but is actually separate.

(Refer Slide Time: 09:20)



We will just assume that expressions are evaluated in a state which is a very simplistic assumption and commands are state transformers. That is really what happens in the case of an imperative language. Each command can modify the state in some fashion and those state transformations are irreversible changes in the sense that to undo the change requires at least as much effort as was required to make the change. We will look at just state as an abstract concept which has got nothing to do with value memory and we will look at state changes. Given two states Σ and Σ prime belonging to the set of states we assume that there is a predefined collection of variables v always.

(Refer Slide Time: 10:10)

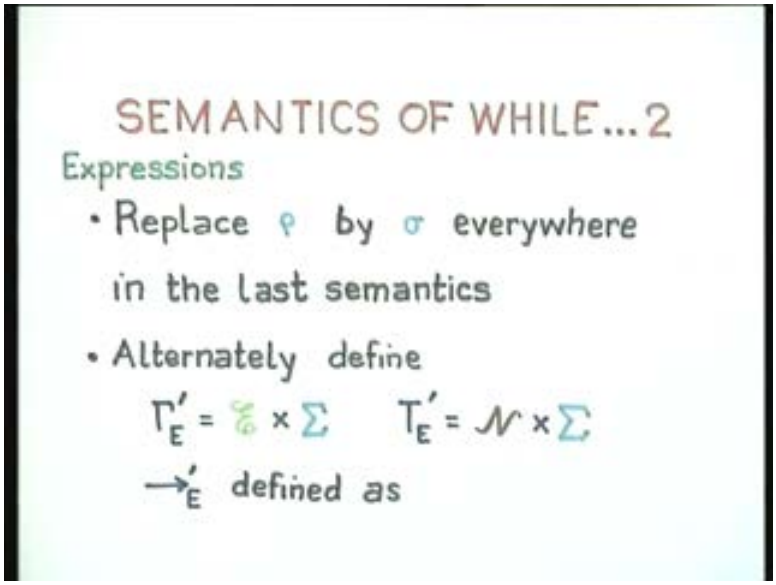


Since there are no declarations we have to assume an unbounded collection of variables available to us and we will just assume that they all have some value associated with them and very often that value could be undefined. You could augment your value domain with some element called the undefined and initially, an uninitialized variable could be regarded as having the value undefined. They have representations in actual hardware meaning there is a null value that can be used, which is not a data value, belonging to any type.

I will consider the change from sigma to sigma prime and represented as a one point change in the sense that sigma and sigma prime are everywhere identical except at the variable 'i'. At the variable 'i' whatever may be the image of i in the state sigma, in the state sigma prime its value is n, which is what $\sigma' = \sigma [i \leftarrow n]$ indicates. We are saying that σ' is everywhere identical to σ except at i where σ' has the value n and σ may have some value but it may not be the same as n. For all variables other than i, σ and σ' yield the same value and for i Σ' yields the value n. This is very important because the assignment statement involves a one point state change and so it is absolutely basic.

We will go through the semantics of the language in a rapid fire fashion so that it acts as a revision of whatever we have previously done and introduces a few new concepts at the same time. The language of expressions is not very different from whatever we have previously defined. It is just that now we are looking at expressions evaluated in a state. You can think of it as replacing row by sigma but to be more rigorous let us look upon the evaluation. We will look upon the set of configurations for expressions. Everywhere you carry a state with you in the execution of a program and the set of configurations of expressions is just the collection of all ordered pairs of expressions and states and the set of terminal configurations is just the collection of numerals along with the state and so, we have to define the transition function.

(Refer Slide Time: 14:00)

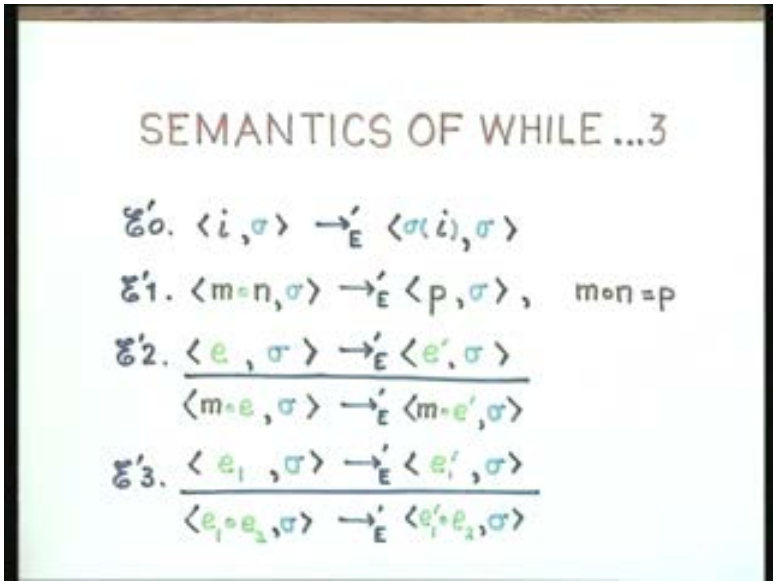


I have put a prime everywhere to distinguish it from the last semantics but you will see that it does not look too different from the last semantics. Let us quickly go through the expression language and you will see that it is more or less like the last one except that since I am considering expressions to be ordered pairs. The meaning of an expression, ‘i’ in the state sigma is the value assigned by sigma to ‘i’ in the state sigma. It is just the value of the expression ‘i’ in the state sigma and you carry the state along with you every time.

The meaning of a binary operation on two literals assuming that the binary operation is already available in the underlying virtual machine, is just whatever is the result provided by that operation in the underlying virtual machine. It is the base case of left to right evaluation for expressions. The rule just says that given a constant m binary operation some expression e this goes to m binary operation e' provided in the state sigma the expression e can move to e'. Note that we are considering a very simplified language in which there are no state changes that occur in the evaluation of expressions.

Going further given a complicated expression of the form e1 binary operation e2 in a state sigma our left to right evaluation strategy says that in the state sigma if e1 moves to e1' then the complicated expression moves to another complicated expression in the same state. So the e' Not to e' 3 are really a syntactic translation of the rules we gave e0 to e3 for left to right evaluation expressions under some environment.

(Refer Slide Time: 16:35)



Since it is an imperative language with conditional statements and looping statements it also means that we have separate categories of Boolean expressions. Before we can

actually give the semantics of the commands we have to define the semantics of Boolean expressions.

Let us quickly look at the semantics of Boolean expressions. I will assume that the set of Boolean expressions that are possible includes all Boolean expressions that we already have and two predefined constants t and f denoting true and false. There is no reason why we should assume that t and f have a representation similar to that of the constants.

We could be type respecting in whatever we do and we might keep Boolean constants separate from the integer constants and this is not too outlandish because there are architectures which are tagged. In many tagged architectures there is a tag associated with every memory location which a type could be specified and storage is allocated only based on the type. This is not too obscure or outlandish and as we did in the case of the declaration language though I have not explicitly specified it here we will also include these constants t and f as part of the Boolean expression language. In addition you can think of the Boolean expressions as having an extra two productions and the language that we will be considering for semantic purposes.

You can form Boolean expressions with constants. In the process of evaluating a Boolean expression you will also include constants. This ‘true’ is a language specific syntactic entity. This t is a constant available in the underlying virtual machine. The two are not exactly the same. The ‘true’ is in the syntax of the language and you will always be using it. You will never in a program actually use t or f. It is some constant provided in the underlying virtual machine. The semantics of this true will just turnover to be the t in our single step.

If you want you could for example have gotten rid of true and put this t and f here just as much as having it separate. But very often you have to distinguish between an element of a language from an element of its meaning. If you look upon Boolean expressions as a language then there is a separate construct called true and a separate construct called false.

In this truth table semantics what you actually give are not the constructs of the language but of a semantic domain which consists of two different values in which the true and the false have corresponding representations.

The distinction between syntax and semantics has to be kept clear. You can define the language of let us say, propositional logic or Boolean expressions without the syntactic constants true and false. I may not have it at all. I could just have variables no true and no false. If you want a representation of true or false I will take it to be of the form ‘a’ or not a. There is absolutely no need to introduce true into this syntactic language. I could keep true and false purely in the semantic language and have a language of Boolean expressions which is totally expressive. It is not as though the constants true and false cannot be expressive. I could just have the various Boolean constants other than true and false.

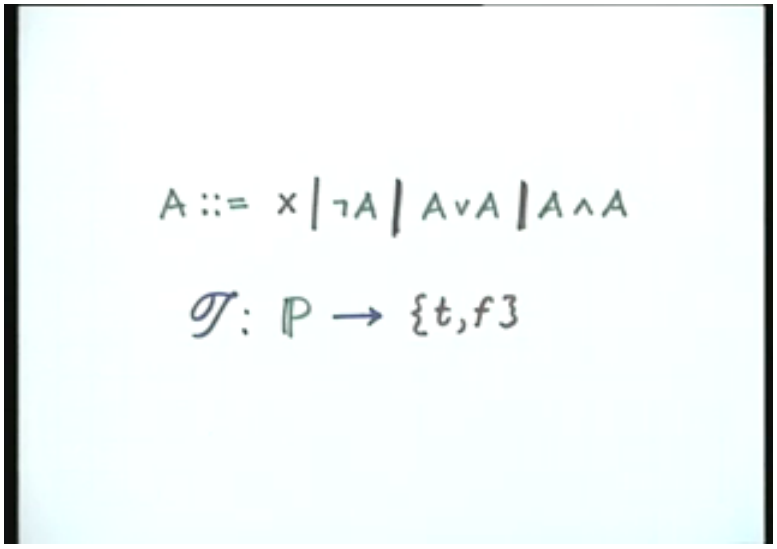
For example, I could just define the language of Boolean expressions or let us assume that you have truth values. I could just define it to be a language which consists of a set of Boolean variables and just these operations [¬A] [A v A] [A & A]. There is nothing which tells me that I should have true and false in the language at all. I can define a complete propositional set. This is a complete set of adequate connectives of propositional logic or of Boolean algebra. I do not need to have the constants true or false. Here x is just an identifier or a propositional variable or a propositional constant or Boolean variable or a Boolean constant other than true or false. Now this is a completely adequate set of connectors for Boolean algebra.

In specifying the semantics of this through truth tables I might specify that my semantic function is a truth value function which maps the set of all propositions. So, let me call this the language of propositions. It just maps this to the collection of truth values, t and f. It is important to realize that whatever is in green is the language of propositional logic and whatever I specify in the truth table is really a function of a function called t which maps given a truth value assignment to each of these identifiers. It gives me a truth value of compound expressions and there is no true anywhere in the language. This is not made

explicit in any of your previous courses but essentially there is a semantic domain called t and f which is completely different from the language of Boolean expressions or from the language of propositional logic.

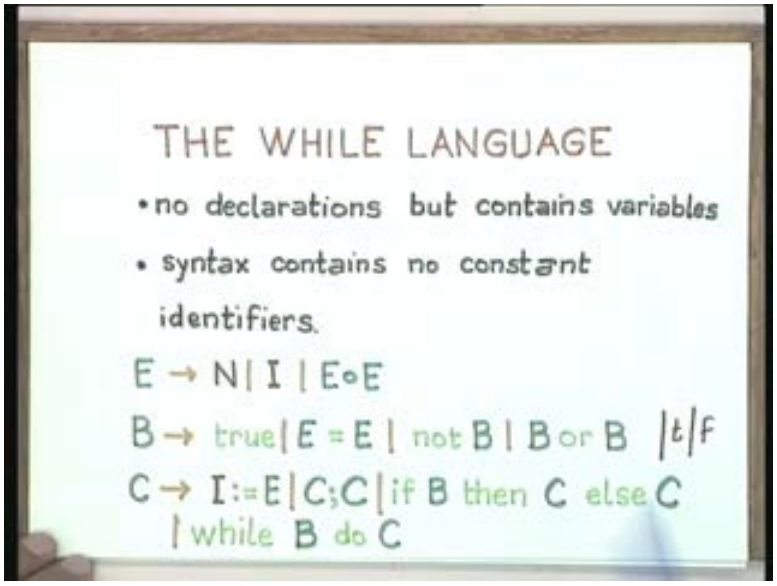
I have just introduced true in the language. I just wanted to introduce a Boolean constant and so I introduced it. It is enough for me to have these three connectors and I can express all possible propositions that I want. This t and f belong to the underlying semantic domain. They need not belong to the syntax of the language at all.

(Refer Slide Time: 27:30)



It is a different matter that very often you introduce it because you want to make a specification for example; a tautology is an identity for the AND operation or a contradiction is an identity for the OR operation. So, what you do is that you explicitly introduce two separate constants such as true and false to make it distinguishable from t and f into your language so that you can specify those equations in a convenient fashion. But there is absolutely no reason why you should do that. I could specify a tautology of the form A or ¬A. I could specify a contradiction of the form A and ¬ A. I have just introduced a constant into the languages. The constant may not be in the language at all. This is part of the language and what are there in the underlying semantic domain are these two truth values.

(Refer Slide Time: 28:20)

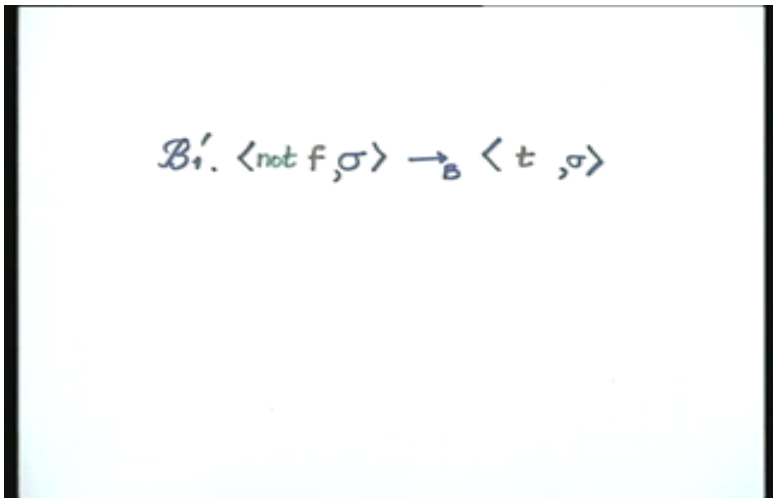


They are not necessarily the same and your elementary knowledge of architecture or hardware will tell you that there is an explicit true in PASCAL but what is its actual representation in the hardware? It is not true at all; it is something else depending on the underlying representation. Let us look at the semantics of Boolean expressions. We have this set B along with these two constants t and f and we have a set of configurations which I will call Γ b which is just this entire set. This B includes this t and f too. Unless it includes t and f you cannot have a set of terminal configurations. You might as well introduce this as equal to B' and call this B' instead.

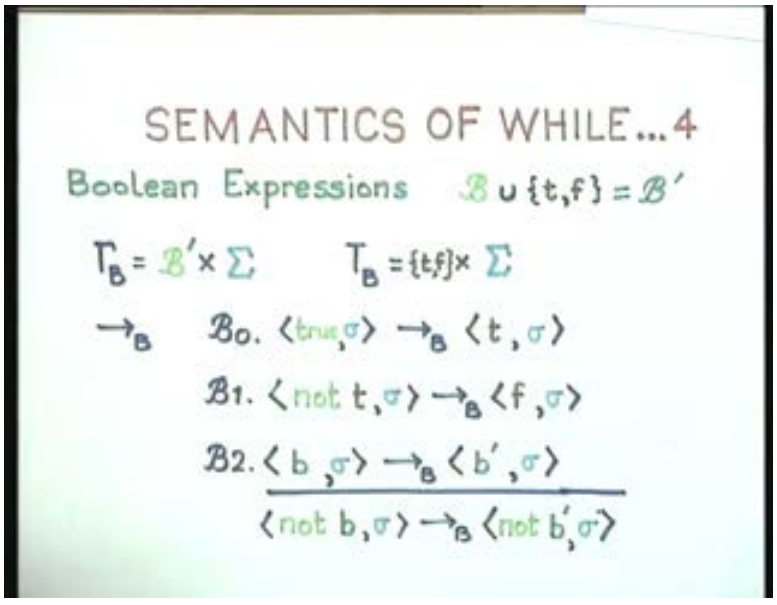
You have the set of final configurations and my first rule for the semantics of Boolean expressions is just that the constant true evaluated in any state sigma gives me the truth value of the underlying machine or interpretation. ¬ of this constant t is f. This ¬ is again an operator of the language and it has got nothing to do with anything that might be there in the hardware. This is the basis of the NOT operation and this is the induction step. If you have a complicated Boolean expression b which is being negated with a NOT then you can evaluate it not only after you have evaluated B and brought it down to a truth value.

If b is some complicated Boolean expression it might have to be evaluated in several steps. Remember that there are syntactic transformations and symbol manipulations. This b might go to some b' and therefore this NOT b will go to NOT b' and this rule will be applied several times till finally this b has been reduced to a truth value.

(Refer Slide Time: 33:07)



(Refer Slide Time: 33:26)

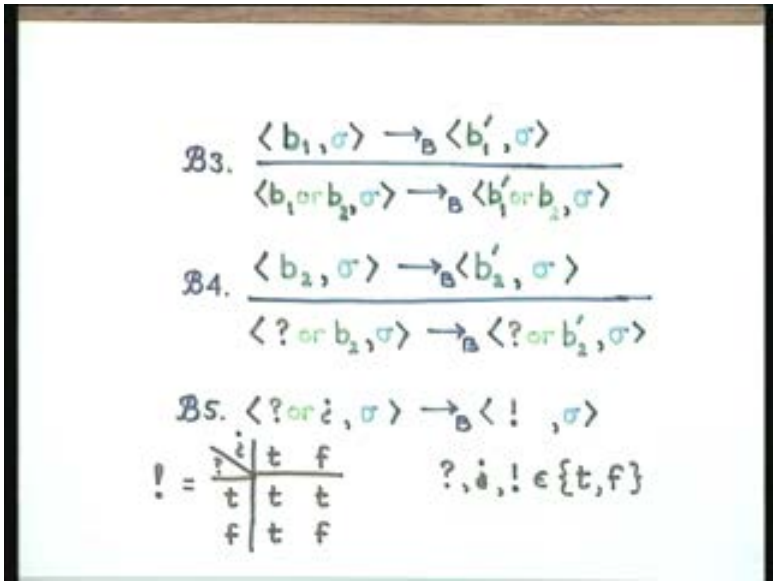


There should be one more rule which says ‘NOT f σ is t σ 2’. Let me introduce that. There is a b1’ which I will introduce here.This is NOT f = t. That gives me a complete set of evaluations of Boolean expressions.

We have to look at the OR operation. Here is the semantics of the OR operation. Again as in the case of the expression language we will look at left to right evaluations. If you have got a complicated expression of the form b1 or b2 then you first evaluate b1 till you reduce it to a truth value, then you evaluate b2 till you again reduce it to a truth value and once you have got two pure truth values you can actually get another truth value and b4 should actually be replicated separately for t and separately for f. Similarly b5 will actually have 4 copies depending upon the truth table. I use an inverted question mark and exclamation as being elements of this semantic domain. Arbitrary elements and the connections between them are given by the truth table.

Actually there are four different versions of b5 depending upon the truth values that you are interested in and similarly several versions of b4, one for each truth value. But I do not want to stress it too much because these are elementary materials. Rather then use new variables I have used symbols as variables over the semantic domain.

(Refer Slide Time: 35:15)



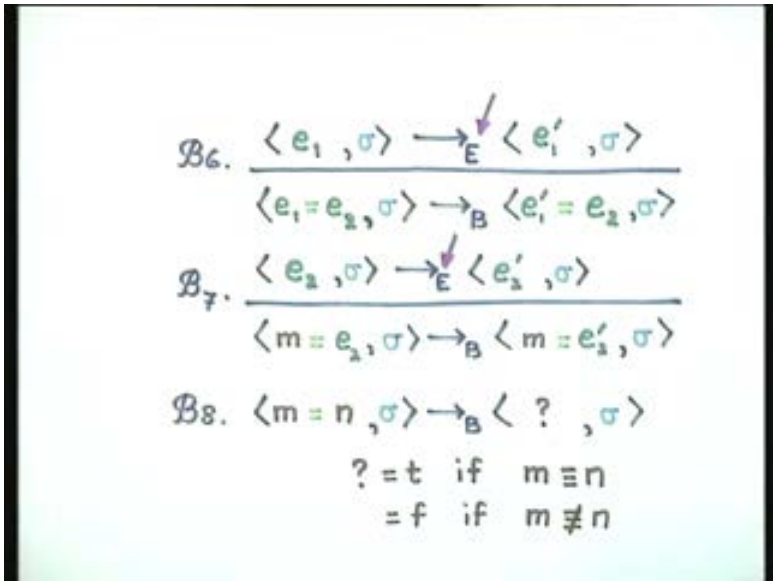
So this is as far as Boolean expressions and combinations of Boolean expressions are concerned except that the only way we can form Boolean expressions are through equality of expressions. We have to look at expression equality which means the evaluation of equalities is again specified in some such fashion. It has to be related to the transition system for expressions and B6 and B7 just as they did for the expression language for binary operators in an expression language. They do the same for the equality where equality is really regarded as a binary Boolean operation between expressions.

B6, B7 and B8 actually specify a left to right evaluation of expressions to yield an appropriate truth value where it must be understood that the resulting truth value says that these two constants should be identical as patterns. That is the only way a machine with no intelligence can really recognize equality. Given that it has no other information, what it can recognize as equality are just two identical patterns. If the two patterns that you get from evaluating e1 and e2 are identical, though I have given them different names here, then you would say that this yields a true otherwise it yields a false.

In our grammar we actually specified the Boolean expressions as a separate grammatical category which used the expression language whereas we are treating Boolean expressions and the equality of Boolean expressions at essentially the same level as the expression language. We are specifying a left to right evaluation of an equality expression which while we assume under a uniprocessor implementation makes pragmatic sense is really belabouring a point. It would be simpler actually to replace all these rules by a single rule of this form. This says that if e1 σ goes in 0 or more moves to some constant m and e2 is σ goes in 0 or more steps to some constant n, then e1 = e2 goes to some truth value depending upon essentially whether the two constants m and n are the same or different.

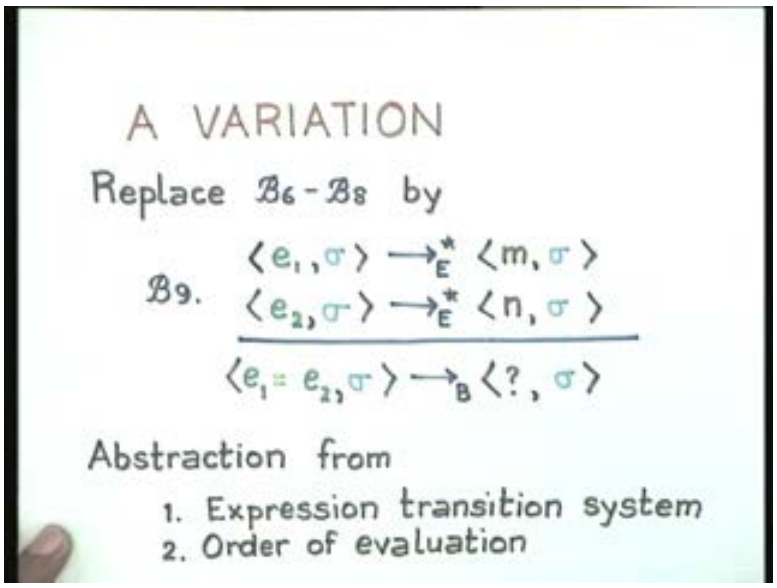
This way you provide an abstraction from the expression transition system and also by giving a rule, you have clearly abstracted away from the order of evaluation of expressions. The rule just says that in order to conclude you require two premises and it really does not matter in what order these two premises are true. You need to somehow prove that even σ -> m σ and e2 σ -> n σ; it does not matter whether you evaluate in parallel, non-deterministically, whether you partially evaluate one and then partially evaluate the other. You evaluate them in some order and if you get two constants then the equality of these two expressions moves to some truth value in a single step. Whereas an application of b6 to b8 says that the equality of expressions really has to move in several steps based on the depth of the Boolean expression e1 = e2 and it makes no difference in the abstraction level between the Boolean expressions and the expressions and even for a simple language it is actually tedious to go around doing this.

(Refer Slide Time: 40:20)



If you are going to do things in the recursive descent fashion it makes sense to do an abstraction at an appropriate level and leave that abstraction as a base case of a lower transition system. B9 is actually what you might call an abstraction from the underlying transition system. It also does not specify any order of evaluation and it greatly simplifies the number of rules that you require which is important. If such a simple language requires so many rules then you can imagine what a real programming language would actually look like. We will use the abstraction to define the semantics of commands. We will assume the existence of the underlying Boolean expression transition system and the integer expression transition system and look upon commands as some high level entity.

(Refer Slide Time: 41:55)



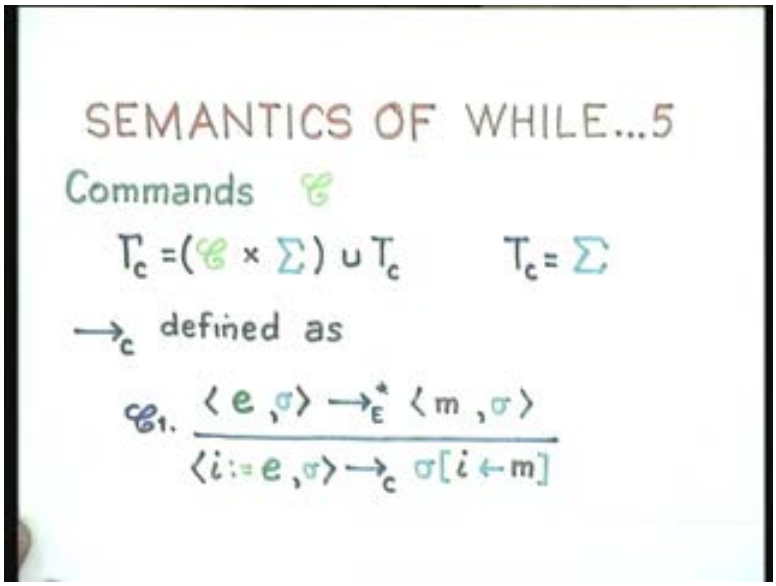
The whole point is that as far as the Boolean expression evaluation is concerned, how many ever steps the sequential expression evaluation might take, all of that is regarded as one single step in the Boolean expression evaluation. Similarly, in the case of the command language we will look upon as many transitions of the Boolean expression evaluation and the integer expression evaluation steps as constituting a single atomic step of this level.

The set of configurations is just the set of commands. I will use script C to denote the set of all possible commands and small c to denote the syntax tree of an arbitrary command. Here we have a slight difference in the sense that once we have executed a program there is really nothing more to execute so there is no command at the end of the execution. Once the program execution is terminated there is no command left and what is left is only a state. Your terminal configurations are just the collection of possible states. At the end of executing a program all you have is a single state; you can look at all the variable-value bindings and there is nothing else left.

In the intermediate stages you have some commands left. You can see that our command language has two distinct types of entities for the intermediate configurations and final

configuration. The assignment statement can go in a single step to an updated state provided the expression on the right hand side of the assignment can go in 0 or more steps of the expression language to a constant m.

(Refer Slide Time: 44:37)



In this setting, which is the meaning of the assignment statement you will see that it has some far reaching consequences. The assignment statement constitutes the basis of the command language and rule c1 actually is the base system. Now we have to consider compound statements. Just like we have a sequential composition of declarations we could have a sequential composition of commands.

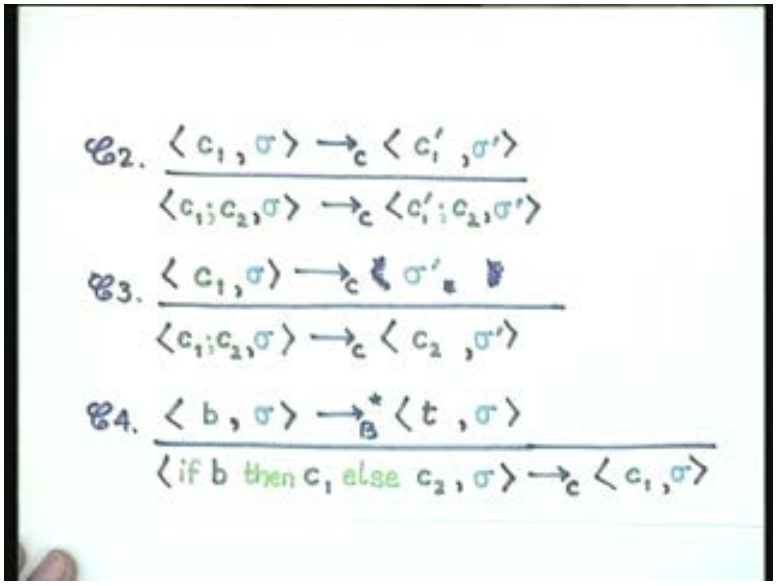
Let us look at c3 first. If c1 can directly go in a single step, c1 in a state sigma directly yields you a new state sigma prime with nothing in it and ‘c1; c2’ in the state sigma yields an intermediate configuration of the form c2 σ’. Here is where the first state changes actually take place. You can look upon this as a basis for the sequential composition. If you have a whole lot of sequential compositions this c3 tells you essentially that if c1 is an atomic command (that means it is an assignment command; in this language the only atomic command in the command language is the assignment statement) which just gives you a state change, then this sequential composition gives

you an intermediate change and c2 tells you the induction step for the induction on the number of semicolons that are there in a command.

If c1 is an atomic command which just yields a state change then the sequential composition gives you this. So, this is the basis of the induction over the number of semicolons. So, if c1 is not an atomic command but is instead itself some complex command then there is no reason to suspect that in a single step it will give you just a state. It will actually give you some other command. Remember that it is all symbol manipulation. It should give you some other command with possibly a modified state sigma prime. Then this entire program c1; c2 moves in a single step to this configuration with the modified state. This is as far as sequential composition is concerned.

The ‘c2’ is also applicable when c1 is something other than a sequential composition.
For example, c1 could be a conditional statement or a looping construct. In the case of a conditional statement we will assume that the Boolean can be evaluated in several steps possibly of the Boolean expression transition system to a single truth value and if that truth value is T then the effect of this conditional is just to get rid of c2 and everything else and only retain c1 as part of the configuration. Since our Boolean expression transition system did not allow for changes in state, the state remains the same. So, this c4 just specifies a transfer of control on evaluating a Boolean expression. Nothing else happens to the programming state.

(Refer Slide Time: 48:58)

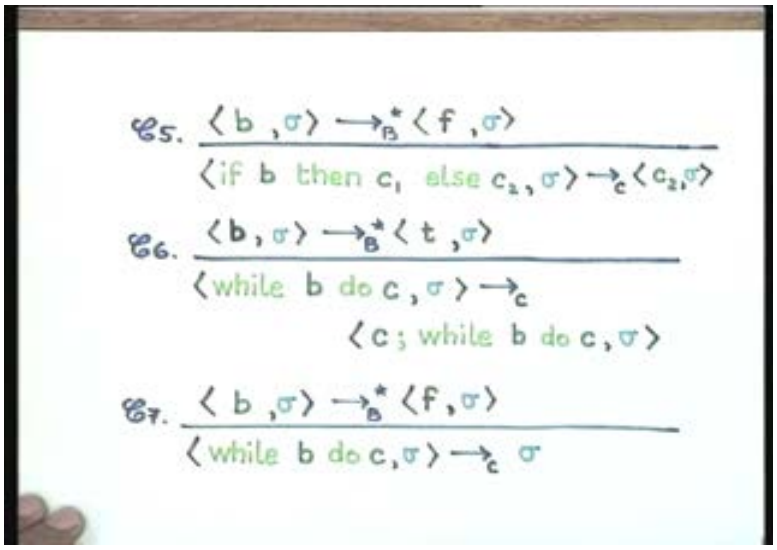


It is just a transfer of control that occurs and the control is such that c2 is discarded after consuming b and getting a truth value t and this is the program that you have to execute. What happens if the Boolean condition evaluates in 0 or more steps? This 0 or more steps is important because for example, in our language this would not be 0 or more steps; it would be 1 or more steps but if t and f are actually part of your language then there is no evaluation involved and so it could be 0 or more steps. Since my t and f are not part of the language this will actually turn out to be 1 or more steps. If the condition b evaluates to false then you just discard the branch c1 and you have c2 left. There are no state changes because the evaluation of the Boolean condition does not change the state and this also just signifies the transfer of control to c2.

The ‘while’ statement is very simple; if the condition b evaluates to true then you manipulate this ‘while b do c’. There is no state change. How is the control changed? You execute the body c and then you execute the ‘while’ statement again. So, having evaluated b and having got a truth value t, the program that you have now left to execute is more complex then the program you started out with because you have to execute the body c and execute ‘while b do c’ again.

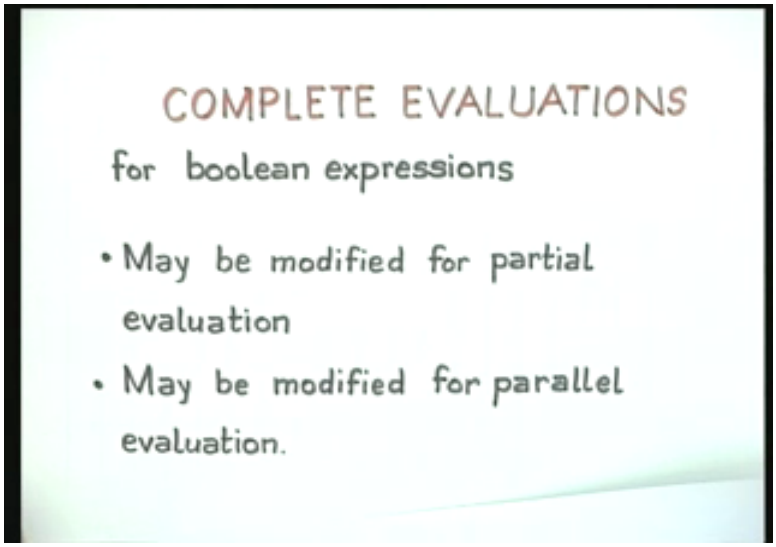
In the case of a looping construct this is how you specify in advance what the transfer of control is going to be. You first transfer control to c and then you append this entire construct again to the end. You sequentially compose this entire construct with the body c and if b is false then this entire construct works out to a no operation and you get just a state sigma. So, this denotes the termination condition for the while loop.

(Refer Slide Time: 51:53)



For Boolean expressions we have specified complete evaluations. You could actually modify this transition system for partial evaluations (by partial evaluations I mean short circuit evaluations). You could even do parallel evaluation of Boolean expressions and you could modify all these transition systems also for parallel evaluation.

(Refer Slide Time: 52:33)



I will stop here. This provides a brief overview of both how to specify commands and Boolean expressions. We will get into the more complex issue of stores in the next lecture and why we require stores and some modeling of memory.