

Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi
Lecture no 7
Lecture Title: Syntactic Classes

Welcome to lecture 7. Let me just briefly recapitulate what I said last time. Firstly, we are interested in defining the semantics of programming languages in somewhat the same way as we have rigorously defined syntax. While defining semantics of a programming language we should keep in mind that a programming language is capable of an existence that is independent of any machine very much like natural languages that do not have any implementation but exist as objects of our thought or our conception.

Similarly, there is no reason why programming languages cannot be thought of as objects of our conception or of our thought independent of any particular implementation. So, while defining the semantics of a programming language one point which must be kept in mind is that it is a mathematical object in itself interesting enough to study for its own sake.

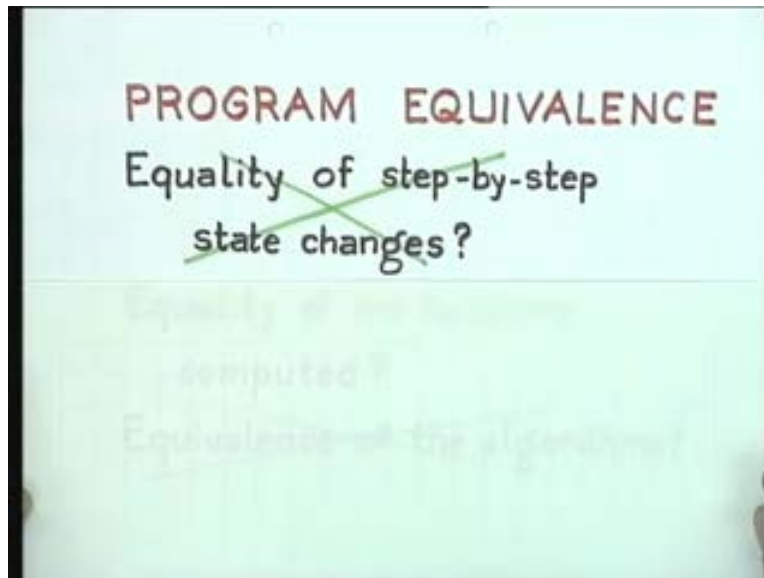
Secondly, what reinforces this view is also that if indeed programming languages have to be implemented then we would like to implement them or specify their meanings in a fashion that is independent of any particular machine or architecture. The fact that you have to make it independent of any particular machine or architecture itself gives it a standing that is abstracted away from actual implementations and it gives it an independent existence.

As far as programming languages are concerned, in recent times we have what are known as specification languages which are very much like programming languages except that they are not implement-able. They are an excellent means of conveying ideas either about programs or implementations but they really exist completely independently and do not have any implementation. In fact some simple specification languages that have been used in the past are languages like logic itself. They could be used as specification languages; they have a particular syntax, they have a certain grammar but they do not necessarily have a complete implementation except for certain subsets of them which could be implemented.

Specification languages for example, would include some implement-able concepts and also others which are meant to convey abstract ideas in a precise and unambiguous form without necessarily having an implementation. We will look at the basic syntactic classes of programming languages what from a semantic viewpoint constitute the basic syntactic classes that we should study in programming languages.

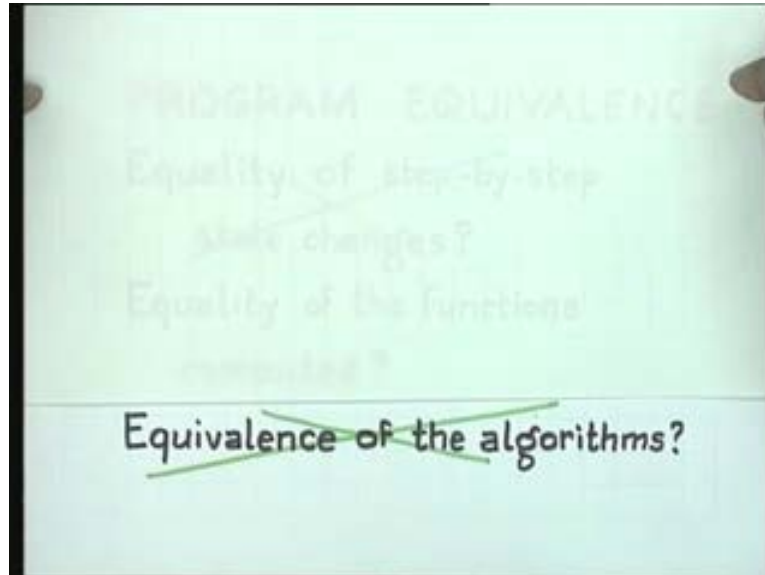
Previously, the syntactic classes we studied were the non terminals, the formation of terms etc. They had a purely syntactic existence which was really devoid of any meaning that we might associate with them but now we would like to classify the various kinds of constructs into some broad syntactic categories which are influenced by what we expect the meaning to be. So, basic to this view point is the notion of program equivalence which we have already seen.

[Refer Slide Time: 05:10]

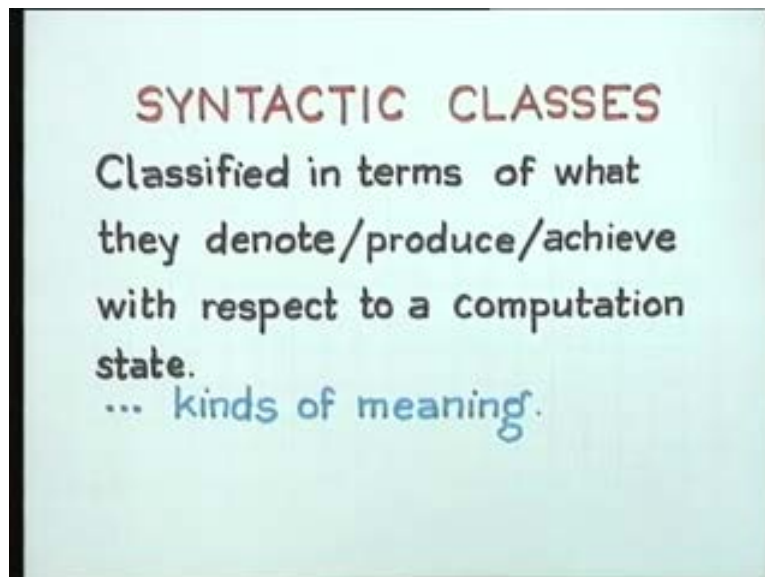


Firstly, as I said, a view of equivalence, which says that all the step by step state changes should be the same in order to deem two programs to be equivalent, is really too fine a criterion which we do not want. Secondly, saying that two programs are equivalent if the algorithms they represent are the same or equivalent is also too coarse a way of looking at programs because we would like to be able to reason about programs in terms of the underlying syntactic structure and every algorithm has an infinite number of implementations. What we would like to know is slightly finer than just the equivalence of algorithms or just the equivalence of functions. If we can do something finer than that we would be able to abstract away from that and get our final results namely the equivalence of functions or equivalence of algorithms.

[Refer Slide Time: 05:35]



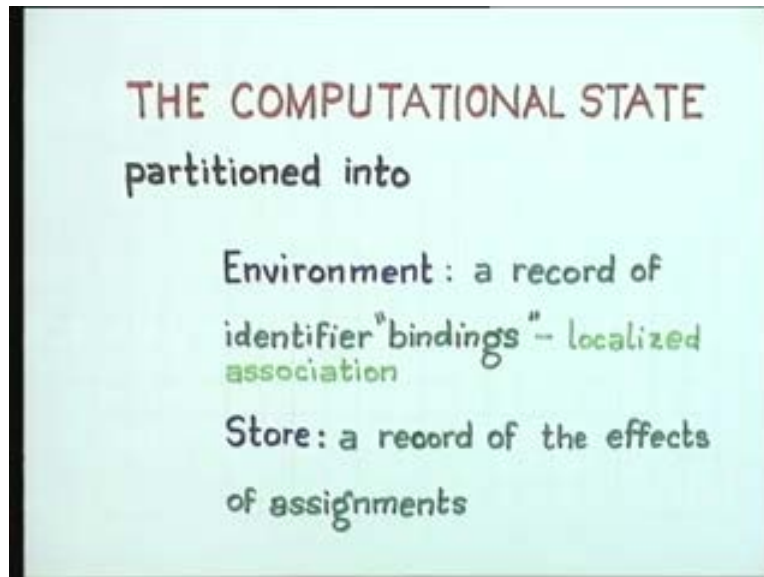
[Refer Slide Time: 07:00]



In order to use the syntax of a programming language in order to derive its meaning, we will define certain syntactic categories from the view point of semantics and from the view point of what kinds of functions we expect them to mean. This is the basic question. If you look at an arbitrary programming language what kinds of syntactic classes do you find which seem to be completely different in meaning. We will look at each programming language construct as a notation. The question is what does it denote? What does it produce or achieve with respect to some notion of a computation state?

We would like to see what the various kinds of meanings we might associate with programming language constructs are which are very different. The kinds of meaning are what we are interested in. First we should get to know about the kinds of meaning and then we can actually worry about the meanings themselves. In general for most programming languages we could look at the computational state. I have never actually specified the notion of a computational state. Let me first partition the computational state into two kinds of objects.

[Refer Slide Time: 08:25]



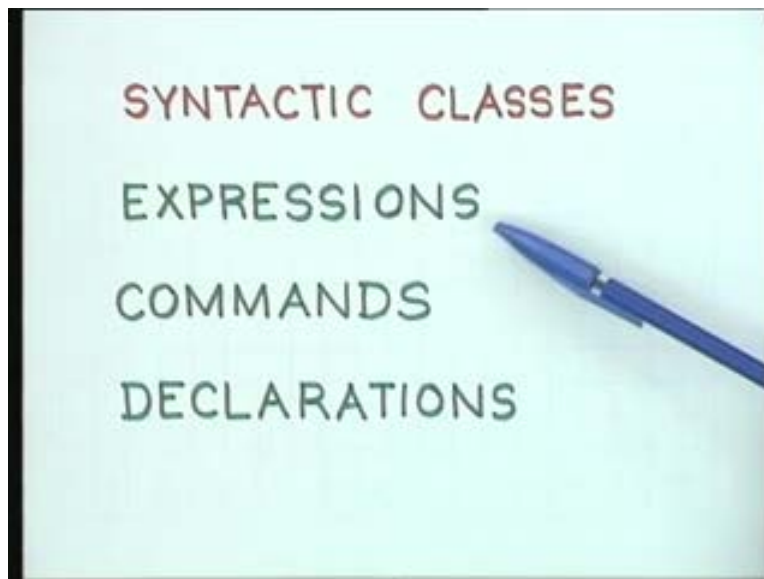
An environment is loosely speaking a record of identifier bindings and ‘identifier bindings’ means that an environment clearly specifies what names are being used for what kinds of objects in a localized fashion and a binding is some localized association. You might have named-value bindings or you might have named-location bindings or may be memory-location bindings. We will just look upon an environment as essentially specifying the different names that are being used in this program and the bindings associated with these names. The various kinds of identifiers in a programming language could be several; they could be constant identifiers, they could be variable identifiers, they could be typed identifiers, they could be procedure identifiers, function identifiers and the same identifiers could be used in different scopes with different bindings. That is what makes the binding a localized association.

You could use the same identifier let us say, ‘I’ as a global variable of a program and as a name of a function in some deep inner scope. So, there is only one name, ‘I’ but it has different bindings at different stages of the program. It has different localized associations while you are within some deep inner scope. Let us say that the procedure called ‘I’, which is available in that scope while you are in the outermost level of the program, would denote the variable that was declared globally. The notion of what ‘I’ denotes is important and that is a binding. So, names themselves are associated with bindings and till you know what names are being used for what purposes, you really

cannot talk about the meaning of a program and this is actually an age old practice even in Mathematics. Take any problem; given that 10 apples cost Rs.20 you have to find out how much 15 apples cost. So, first you let 'x' be the cost of 15 apples and there you are specifying a name 'x' with a binding that it is the price of 15 apples. You might use another name. Let 'a' be the cost of one apple and there again you are specifying a name-value binding. So, an environment is just a collection of names along with their bindings in the environment.

The next component of a computational state is the store. Loosely speaking, in most imperative languages at least the store is just a map of the memory. It is just the record of various state changes that have taken place. It may not be a complete history of the state changes but might be the result of all the latest updates. This store really gives you various kinds of location-value bindings. We will look at that in greater detail later but essentially we can look upon a computational state as consisting of two parts, an environment and a store and this is the basic notion of a computational state. Certain programming languages may not have the notion of the store at all and that is what distinguishes functional programming languages from imperative programming languages. There is no concept of a store in functional programming languages. However, we would like to talk about all these in a uniform fashion.

[Refer Slide Time: 14:30]

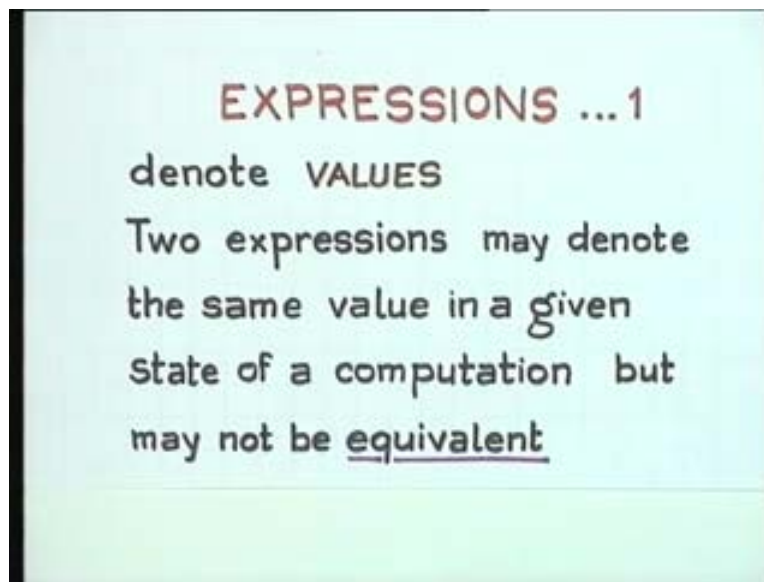


So, it is necessary for us to take the full generality of the computational state into account whenever we talk about the subject of programming languages. If you look at any programming language, let us say imperative programming languages; since they are the hardest to specify and they are also the most complex we might classify the various syntactical constructs in the programming languages into three broad classes. One is the class of expressions that the language allows that is a class of commands and a class of declarations. These three syntactic classes do not overlap but you can use one syntactic class to define another. If you look at what expressions denote, what commands denote

and what declarations denote, you will see that there is really nothing else to programming languages. Expressions, commands and declarations are the three aspects that are important when defining semantics.

Let us look at expressions. As far as we are concerned the meaning of an expression is just some value. There are constraints such as; the expression should have a value of the type mentioned as the type of the expression, the type of a complicated expression should be derivable from the types of the individual components of that expression etc but essentially any expression denotes a value. The whole idea of an expression is that it has to be evaluated to eventually obtain a value. The notion of this value is different from the notion of a memory location. For pragmatic reasons you might actually use a memory location in order to store the value but that memory location is not part of the language specification of the language. That is a pragmatic aspect which has to do with the architecture or the machine or a particular implementation of the language.

[Refer Slide Time: 16:05]

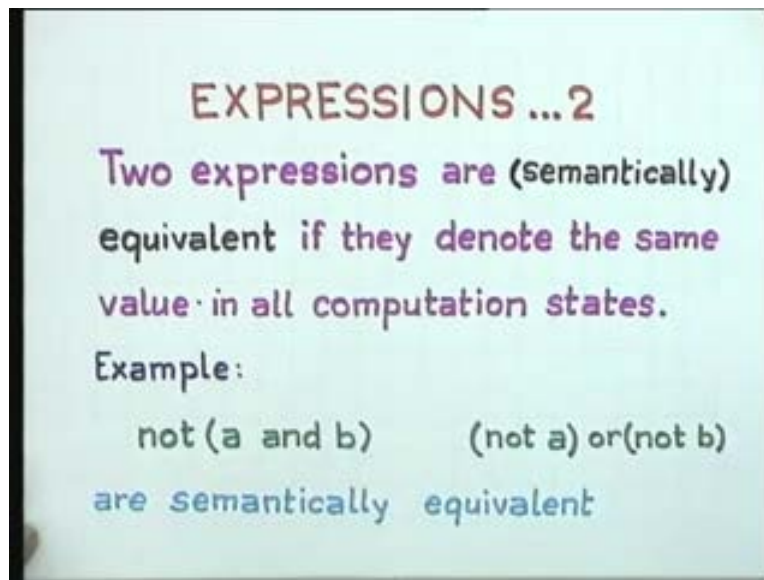


There is absolutely no reason why we should confuse values from values stored in memory locations. The two are logically distinct as we will see. In any particular environment and store you might have two expressions which denote the same value in that particular environment and store, but they may not be equivalent. We are eventually also interested in what constitutes program equivalence and since expressions form an integral part of any program we are supposed to abstract out program equivalence from the equivalence of the individual components of the program. When we talk of expressions, the first question that arises is what constitutes expression- equivalence. When can you say that two expressions are equivalent?

In a particular computational state two distinct expressions which have distinct abstract or concrete syntax trees might actually give you the same value but that does not mean that

the two expressions mean the same. The two expressions have a meaning which goes far beyond any particular computational state so if they yield the same value in a given computational state that means, in a given environment with a given store, then at that particular point in the program one of the expressions could be replaced by the other but that does not necessarily mean that the two expressions mean the same. You would say two expressions mean the same when under all states of computation the two expressions yield identical values. Regardless of what might be the computation state, if the two expressions yield identical values then we could say that the two expressions are semantically equivalent.

[Refer Slide Time: 19:19]



What we mean by semantical equivalence really constitutes equality in a broad mathematical area in which that expression resides. A simple example is that of these two Boolean expressions;

‘not (a and b)’ ‘(not a) or (not b)’.

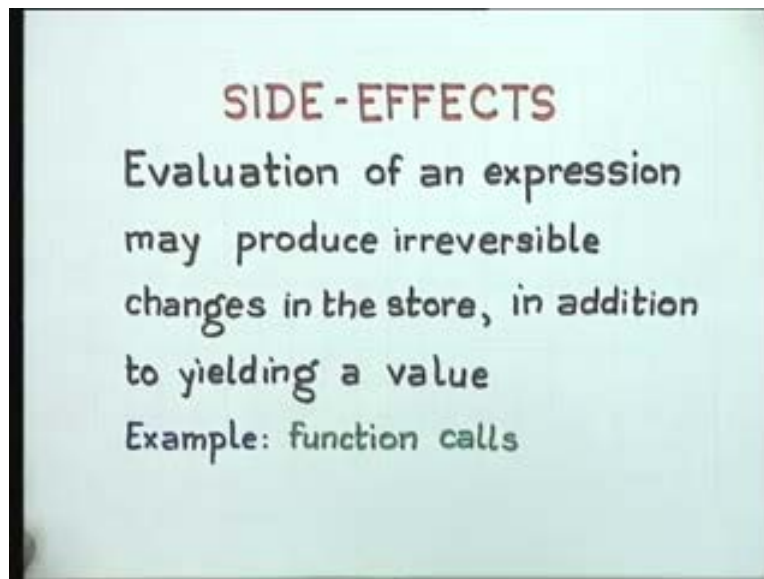
There is the standard De Morgan’s law in Boolean algebra which tells you that these two expressions ‘not (a and b)’ ‘(not a) or (not b)’ have the same value. So, they are logically equivalent which means it does not matter in what context these two expressions appear but in the larger framework of Boolean algebra these two expressions are the same. They are syntactically distinct entities but they are semantically equivalent from the laws of Boolean algebra. When we are talking about programs or expressions occurring within programs we are really talking about syntactic entities and defining the meanings of syntactic entities in terms of other semantic objects.

Your semantic objects could reside in any area of mathematics. A form of semantic equivalence really constitutes equality in mathematics. In most of mathematics we had not really worried about syntax. We are worried about their semantical properties or what might be called model-theoretic properties. A mathematical discipline focuses on a

certain class of objects and we are looking at truths with regard to that class of objects. Two statements in that area of mathematics are equivalent if they denote really the same object in that class of objects that the mathematical discipline is concerned with.

In the case of Boolean expressions we have two distinct syntactical entities which denote this same object in all possible contexts and therefore they are semantically equivalent. So, that is what we would consider expression equivalence. We should ensure that a programming language follows at least these basic principles. You should be able to derive equivalence, not necessarily through a theorem proved but at least by hand and they should not in anyway distort our conventional notions of equality in mathematics. However, you will see that they actually will distort our conventional notions. A lot of programming languages because of their particular implementations do distort conventional mathematics. Many expressions especially in imperative languages produce what are known as side effects.

[Refer Slide Time: 26:12]

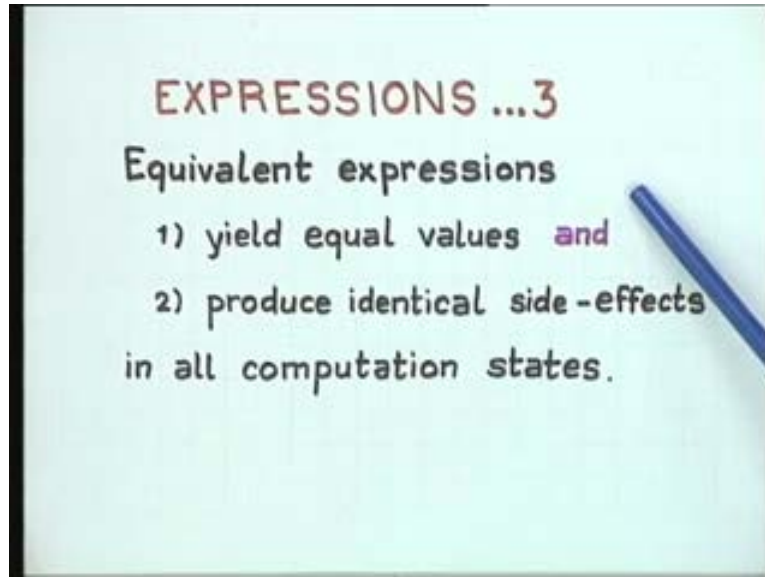


The expression denotes a value but, along the way, it alters the store. A simple example of this is a function in Pascal. A function in Pascal might be called from some outer scope and if it is a function call, it should reside within an expression. You could have some complicated expression involving let us say 'sin x'. If this function is declared by the user it is quite likely that the user inside that function body has changed some global variable.

Even though at the place of the call of the function it looks like an expression which just yields a value and nothing else. It makes no other changes because there is some assignment to some global variable inside the function body. The store that is the location-value bindings of the whole program are changed irreversibly and that is a side effect produced by a function call. So, two distinct expressions may yield the same value under all conditions but they may produce different side effects on the global store.

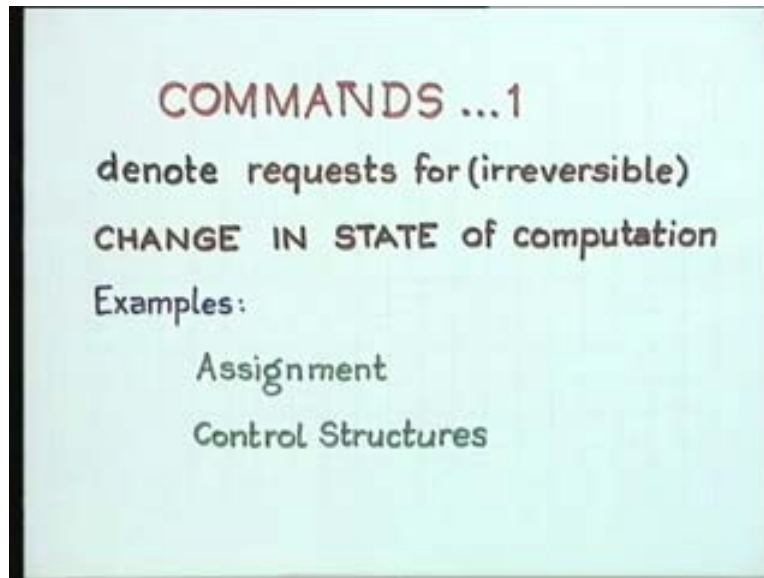
What it means is that we will have to change our ideas about what constitutes the equivalence of two expressions. I will talk about an irreversible change and then it is possible to understand why this particular change is irreversible. Side effects can be produced by expressions especially in most imperative languages; a large part of 'C' relies entirely on side effects. Then we have to refine the meaning of what constitutes the equivalence of two expressions. We can say that two expressions are equivalent if they yield equal values and also produce identical side effects in all computation states.

[Refer Slide Time: 26:47]



The side effect produced by an expression is not a problem in mathematics but in computer science. It is also one of the reasons why debugging programs is itself a hard task in most programming languages because the side effects are very carefully hidden under different names and different environment bindings. They are aliased and all these factors complicate matters for analyzing or debugging the program. That is what constitutes a large class of expressions in most programming languages.

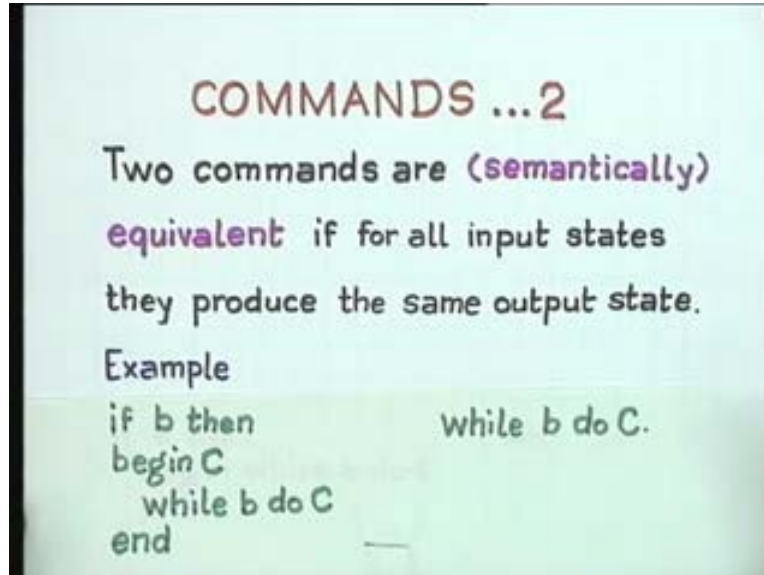
[Refer Slide Time: 30:21]



The next syntactic categories are what are known as commands. We would look upon a command as a request for a change of store and it is irreversible again. It is irreversible in the sense that it is not guaranteed that the command can be undone. Two subsequent commands may neutralize the effects of each other but it is not guaranteed that you can always do that. We would still look upon those two commands which neutralize the effects of each other. Start with a certain store sigma ' Σ ', one command produces a new store Sigma, ' Σ ' 1 and the other command produces back Sigma, ' Σ '. We will just regard this as a sequence of two irreversible changes and that it is a mere accident that the last state Σ 2 is the same as the first state Σ .

We will see what exactly is reversible to understand what irreversible means here. We will look upon a command that is essentially going to change the state of a computation in some 'permanent' fashion and permanent is within quotes because it is permanent till the next command changes it further. On the other hand if you take expressions without side effects, they really do not produce any change at all in the store of any computation. They only yield a value but they do not change a store. Most of the commands you are familiar with such as assignment commands, various kinds of control structures in most of the imperative programming languages clearly involve state changes. You could have a null command which does nothing but that is just a degenerate case of a command that can change a state.

[Refer Slide Time: 30:56]



The next question is; when are two commands equivalent? We will say that two commands are semantically equivalent if for all input states they produce the same output state. A command is essentially a state to state transformation. It takes an input state as an argument and yields an output state at the end of its execution.

For example; take the command;

if b then

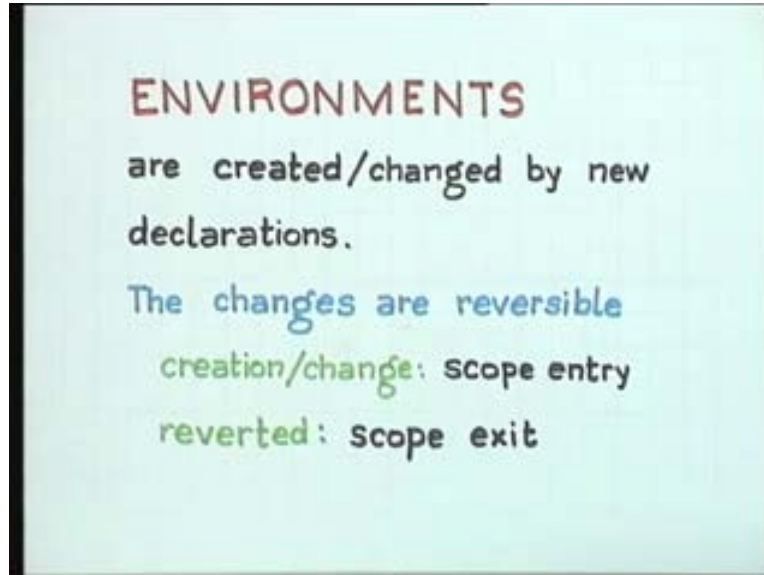
begin C while b do c.

while b do C

end

It consists of an 'if then' construct with a 'while' loop inside it which is semantically equivalent to this construct, while b do c which is just the 'while' loop itself. The construct on if b then...end could have been obtained by an unfolding of this 'while' loop in some standard form. Notice that 'b' in the example is a Boolean expression and this Boolean expression could actually produce side effects. If there is a function call inside this Boolean expression it could produce side effects but this is equivalent in the sense that even if b does produce side effects, the effect of one is exactly the same as the effect of the other importantly under all computational states. Being able to locally produce the same kind of state changes is not the same as being able to assert that under all computational states they produce the same effects.

[Refer Slide Time: 34:00]



The last categories are what are known as environments and environments are created and changed by declarations. Declarations constitute the last syntactic category and they change environments. They produce environment to environment transformations. Sometimes they also produce changes in the store, so in that sense environments could be changed also in an irreversible fashion. But when we normally talk of environments that is the changing of environments by new declarations, the changes are very often reversible in the sense that you create a change in the environment or you create a new environment when you enter a new scope and when you exit that scope you often revert back to the old environment.

The change in the environment is reversible in that sense. However, the change in the environment itself could have been produced by some commands or some expressions with side effects which could have altered the store. But the change in the store that a declaration produces is irreversible.

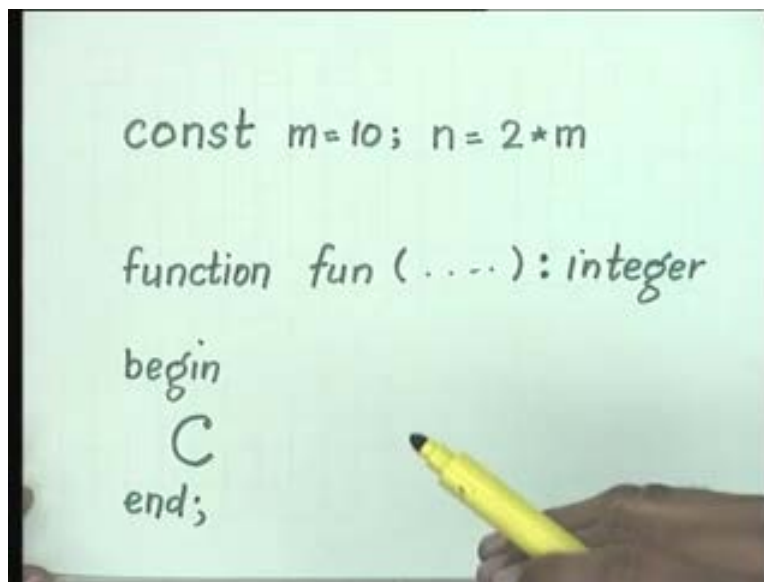
So the environment is reversible that is, the creation of a new environment is a reversible process and usually depends upon the normal scope rules of a programming language. There are different kinds of scope rules you could have but essentially the notion of a scope means that you could create a new environment temporarily and you could destroy that environment and revert to the old environment after some time. In that sense the change in environment produced by definitions or declarations is reversible.

However, as I said, you could have expressions involving commands, commands involving expressions, expressions involving declarations, declarations involving expressions and commands etc. So, declarations could also produce irreversible changes because they have underlying commands in them or they have underlying expressions in them which produce side effects. They could produce irreversible changes but if you look at declarations in a sort of a pure form as merely environment-creating objects, if you

look at declarations merely as a syntactic representation for a temporary change in environment, then what declarations would produce are reversible changes and not irreversible changes. In their pure form we would look upon expressions without side effects as just syntactic representations which do not change the environment or the store but which just yield a value.

Commands in their pure or impure form always change the store. In their pure form they do not create new environments. Declarations in their pure form just create new environments and do not change the store. But since all programming languages are recursively defined using these three kinds of classes, each of them could produce effects which are really impure in the sense that expressions could produce irreversible changes, declarations could produce irreversible changes and store and commands could produce new temporary changes in the environment. That typically happens when you have a procedure call. A procedure call is itself a command but a procedure call often means the creation of a new environment with new localized bindings, associations of new names or new bindings for names already used and temporarily created for the purpose of that procedure call.

[Refer Slide Time: 41:13]

A photograph of a whiteboard with handwritten code in black marker. The code is as follows:
`const m=10; n=2*m`

`function fun (...): integer`

`begin`
`C`
`end;`
A hand holding a yellow marker is visible at the bottom right of the whiteboard.

When I said that expressions, commands and declarations do not overlap, I meant that conceptually they do not overlap. Take the syntactic definition of a programming language say Pascal.

Const m=10;

The effect of this is to create a new environment in which there is a binding for the name m and that binding is the value, 10.

There might be an old `m` somewhere else but the effect of this declaration `Const m=10;` is to create a new environment. I could also have something like, `'n=2*m'`. Here is a declaration of two names `m` and `n` with fresh value bindings and the only way I can produce the value binding is by using an expression.

If you were to look at some function declaration in Pascal, what you might have is something of this form, `'function fun (...): integer'`.

This function declaration also has a function body;

```
'begin
    C
end;'
```

The function body is a command. The only way I can define new functions in Pascal is by using the language of expressions and commands to their fullest extent. Here is a case of a function which will use both the expression language and the command language in order to produce essentially what constitutes a value. Supposing this command in our example is a pure function in the sense that it produces no side effects on the global store then you would have these value (...) parameters. If you had reference parameters they are likely to produce side effects so let us just say that they are value parameters. You would have some local variables and the entire command `'C'` would just involve parameters and local variables and finally there would be some assignment to the name `'fun'` which is the value. But I cannot define new functions unless I use the command language and the expression language.

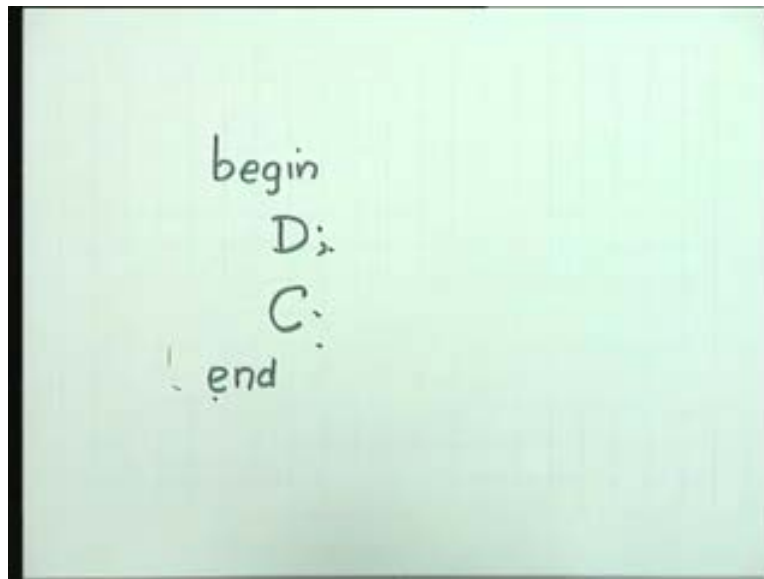
You can have declarations which use expressions and commands, you can have expressions like an expression involving `'fun'` which uses commands and uses a declaration and you could have commands which just use declarations. For example; many languages, from the days of Algol 60, allow you to define a local command, an unnamed function locally or an unnamed procedure locally with just local declarations

This is not allowed in Pascal though. For example; Algol 60 typically allows us to have a `'begin'`, a set of declarations `D; C end` and some command locally as part of some larger program. There is no name but there are some local declarations and this `'begin D; C end'` is a command which requires a new environment to be created by these declarations and this command inside it would in turn involve expressions. Commands might involve expressions and declarations, declarations might involve expressions and commands and expressions might involve declarations and commands.

One of the principles in programming language-design is that you can actually add syntactic sugar to one syntactic class to produce another syntactic class. You can add purely cosmetic reserved words or keywords which transform a declaration into a command, a command into an expression, an expression into a declaration, an expression into a command etc.

They are all mutually interchangeable but the meaning that is intended should be clear. If you are adding something to an expression to make it a declaration then it is clear that your idea is to create a temporary change in the environment.

[Refer Slide Time: 43:20]



```
begin
  D;
  C;
end
```

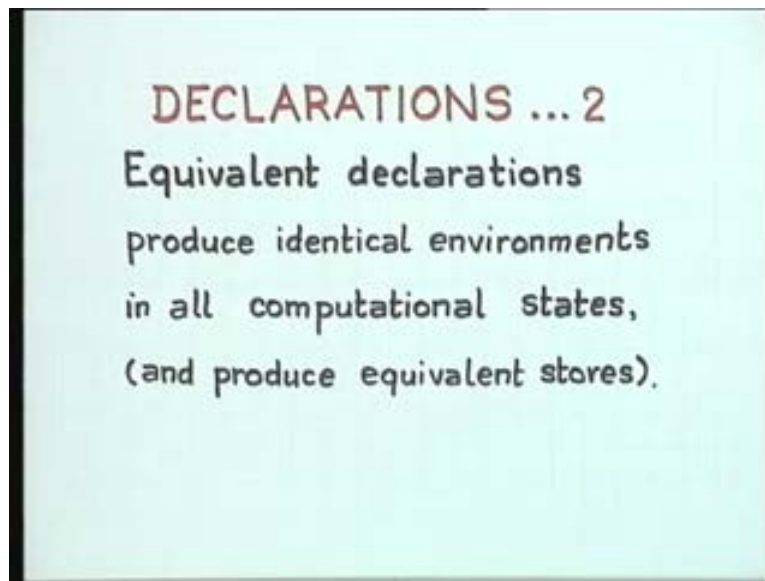
What constitutes an expression in this declaration 'const m = 10; n = 2 * m' is not the same as what constitutes an expression inside 'begin C end;' in the sense that the intension of this declaration, 'const m=10; n=2*m' is to produce a new environment. The intension of a command is to change state. For convenience, if your command becomes too complicated, to simplify it you might require the introduction of more new names which are just local to that command as in the case of unnamed blocks in Algol 60. But the intension of 'begin D; C end' is very clearly to modify the store and to produce an irreversible change in the store.

[Refer Slide Time: 45:40]



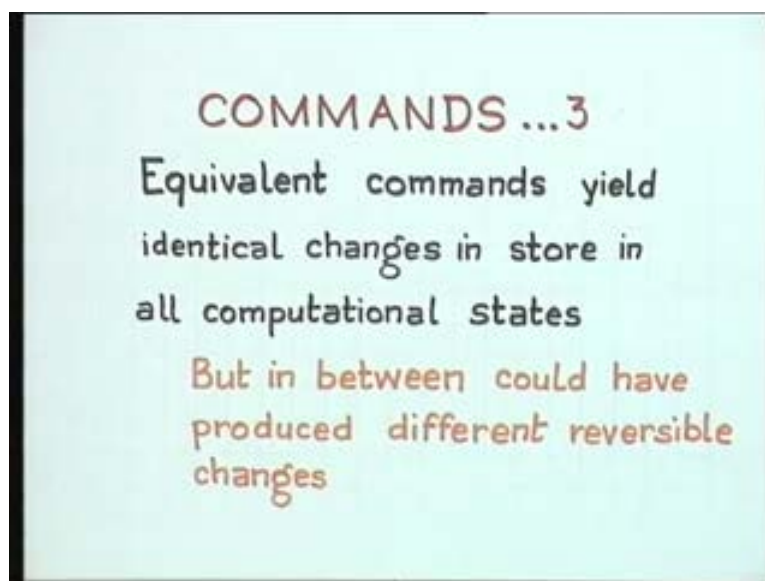
DECLARATIONS ... 1
denote new ENVIRONMENTS
(may also change store)

[Refer Slide Time: 45:55]



Declarations really denote new environments. They may also change the store. We would say that two declarations are equivalent provided they produce identical environments in all computational states and they produce equivalent stores. A declaration could produce irreversible changes as in the case of a function declaration or a procedure declaration.

[Refer Slide Time: 46:20]



When we look at commands we have not earlier specified, we would say equivalent commands yield identical changes in store in all computational states. However, two different commands could be equivalent even though the temporary changes in environment they produce are different. Inside the command you could have some declarations and the declarations in the two commands might produce different new environments but declarations are essentially a creation of essential changes in the environment. At the end of the two commands you should have the same environment and store in both cases.

You start with a computational state and the resulting output state of the computation should be identical in the two regardless of how different the intermediate changes produced in the computational state are. So, what distinguishes what are known as functional or applicative languages and what are known as imperative languages are that imperative languages contain all three syntactic classes whereas functional languages contain only the language of only two syntactic categories, expressions and declarations. The concept of a store and the concept of side effects is absent in functional languages.