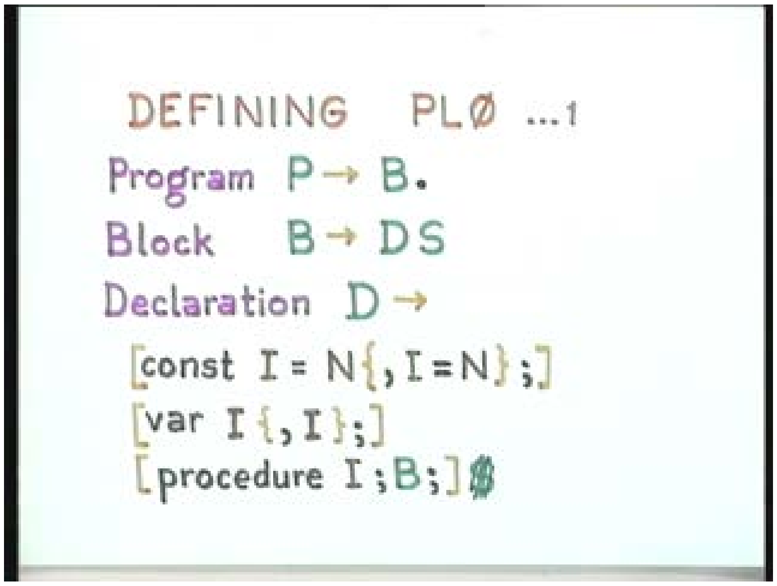
Principles of Programming Languages

Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi
Lecture no 6
Lecture Title: Semantics

Welcome to lecture 6. This lecture will complete in some sense the basic frame work of the rest of the course. Before we start on semantics let me just briefly go through the definition of the programming language, PL0.

[Refer Slide Time: 01:04]



DEFINING PLØ ...1

Program P → B.

Block B → DS

Declaration D →

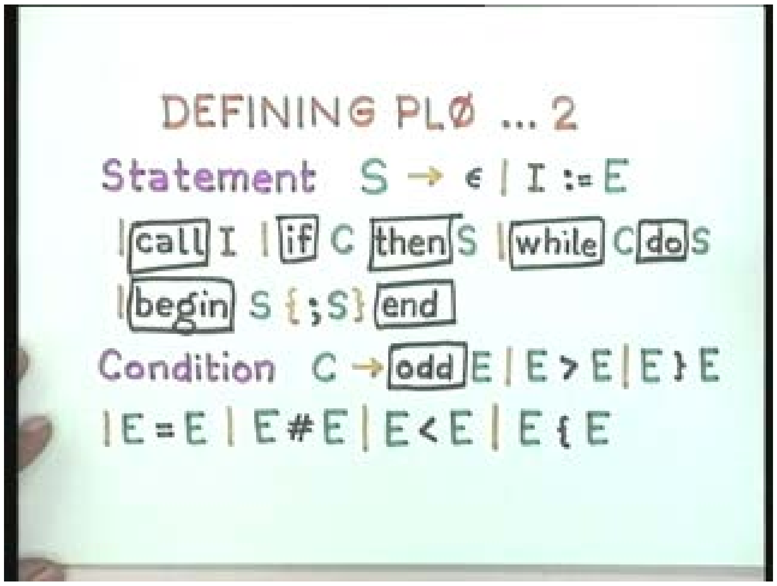
[const I = N{, I = N};]

[var I {, I};]

[procedure I ;B;]*

A program is defined as a block with some syntactic entity. Unlike a period a block consists of a sequence of possibly empty declarations and a sequence of possibly empty statements. It also consists of declarations of various kinds which specify values or types or functionalities as in the case of procedures and statements.

[Refer Slide Time: 01:50]



DEFINING PLØ ... 2

Statement $S \rightarrow \epsilon \mid I := E$

$\mid \boxed{\text{call}}\ I \mid \boxed{\text{if}}\ C\ \boxed{\text{then}}\ S \mid \boxed{\text{while}}\ C\ \boxed{\text{do}}\ S$

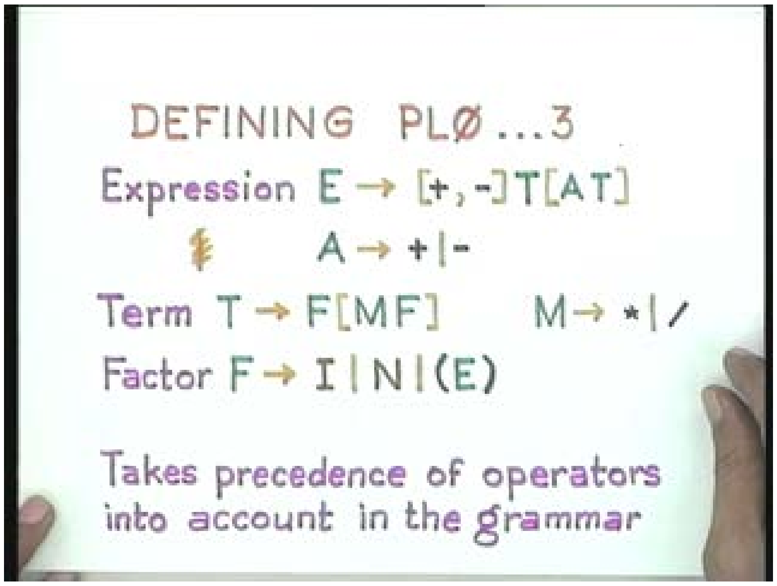
$\mid \boxed{\text{begin}}\ S\ \{;S\}\ \boxed{\text{end}}$

Condition $C \rightarrow \boxed{\text{odd}}\ E \mid E > E \mid E \geq E$

$\mid E = E \mid E \# E \mid E < E \mid E \leq E$

A statement is a particular syntactic or grammatical category which is specific to imperative languages and PL0 is an imperative language because it is based on state changes and we have this usual suite of statements for example, assignment, conditionals and loops. The conditionals and loops are usually dependant upon the truth or falsity of some condition and so we have various kinds of conditions. Most of them are relational arithmetic operators. The conditions and the arithmetic operations in turn imply evaluation of expressions.

[Refer Slide Time: 02:40]



DEFINING PLØ ...3

Expression $E \rightarrow [+,-]T[AT]$

$A \rightarrow + \mid -$

Term $T \rightarrow F[MF]$ $\quad M \rightarrow * \mid /$

Factor $F \rightarrow I \mid N \mid (E)$

Takes precedence of operators into account in the grammar

We have various kinds of expressions and operators on expressions. We have to take into account the precedence of operators so that we have a convenient way of writing unambiguous expressions without having to bother about parenthesizing them. A large part of context-free grammars has really got to do with parenthesizing correctly so that we can generate the right kinds of trees. The specificity of a context-free grammar also is such that you are interested in pragmatic issues. The issues are questions like; is there a successful compilation procedure given the kind of grammar you have defined for the language, is it compilation-procedure efficient, should the grammar be changed in some way if you change the compilation procedure and how should you change the grammar if the compilation procedure does not work very well and it is not able to parse for various reasons? The context-free grammars for specifying syntax are powerful enough so that you can consider alternate but equivalent grammars which might simplify the problem of parsing your programs. It can also be coarse enough to specify just the broad outlines of the language.

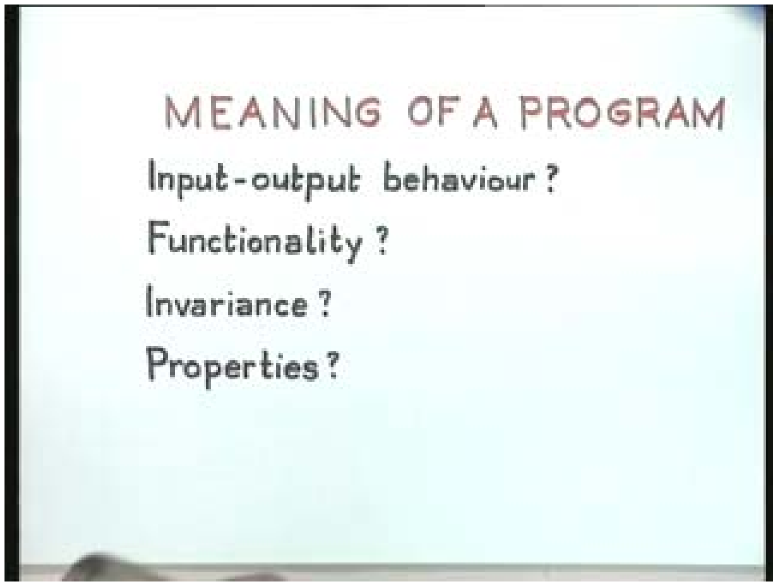
As syntax normally, one would have to specify the grammar in sufficient detail so that firstly you are not restricted to a particular parsing strategy. In the case of PL0 in fact that is one drawback. We have defined the grammar in such a way that it is actually a sub class of the set of all context-free languages. The language itself has been defined so that it falls well within the subclass and you can use various strategies for parsing or compiling these programs in PL0. But on the other hand the moment you include more powerful features and the moment your language becomes larger; your grammar also becomes very large and it might be necessary to go outside the sub class in order to add more facilities. By facilities we mean things that are convenient for the users which do not hassle them unnecessarily. You might require more general grammars so that the specificity can be taken care of at the implementation time.

You can specify programming language syntax to a great level of detail with context-free grammars and importantly from the point of view of a user of the programming language that level of detail is not needed. Most of the time a user is only interested in knowing the abstract syntax of the language. Implementer's view point of the language and user's view point differs in this very important aspect. You require a very coarse-level grammar without ambiguity probably for the purpose of the user but which is fine enough so that the implementer has no trouble in designing a parser for example, for this language.
Instead of restricting ourselves to a highly formalized notation like a context-free grammar, at the level of the user, we might think of the user as being interested only in the abstract syntax trees for each program.

He would like to know essentially what would be the order of evaluation if he wrote a program in a certain way. For example; if he omitted some parenthesis, some semicolons or some sequencing operators, what would be the order of compilation? Does it make any semantic difference whether he introduces those parentheses or not? There is also another aspect which is that the context-free grammar formalism is not powerful enough to enforce certain context-sensitive features.

The subject of semantics really deals with these problems. Let us for the moment not dwell too much on those context-sensitive aspects of the language. Let us look deeper and see what exactly it is that we mean by a meaning.

[Refer Slide Time: 09:25]



MEANING OF A PROGRAM

Input-output behaviour ?

Functionality ?

Invariance ?

Properties ?

There are various ways of looking at the notion of meaning. One is that you might just think of the meaning of a program as input-output behavior. Very often for interactive programs, sequences of inputs and outputs are what we are interested in. For many programs which can be divided up into neat procedures you are more interested at the procedural level to determine what function that procedure represents. Supposing we do not have too much of IO in the program, one view point we can take is that every program is a function or at least if it is an interactive program we can divide up the program between its interactions and look at those segments which are free from any interaction with the outside world or from the user's interaction and determine what function they represent. If you want to determine that, then you also need to know what the domain of the function is, what the range of the function is and by function I mean a mathematical function. There are various schools of thought about how programs should be designed. One very prominent view point is that essentially all programs are functions or mathematical functions which means they should have a clearly defined domain and a clearly defined range.

The program is just a notation to express a mathematical function. This view point also subsumes the view that an algorithm is also really nothing more than a function. An algorithm is an abstract object in the sense it has no concrete representation. Its only concrete representation is a program that implements the algorithm. So, if an algorithm is an abstract object then really what you are looking to do as a matter of correctness is to specify whether the program does not implement the algorithm, which is often not very important. You are interested in whether this program actually implements the function that the algorithm is meant to specify. There is a mathematical function for which you have an algorithm which means that you have spilt up the computation of that mathematical function into various computational steps in some computational model in your mind and in our case most of the time it is whatever computational model that is specified by Von Neumann machines.

There is a computational model in mind whose primitives are used in specifying how a certain function is to be calculated. It is a method of calculation of some large function, in which, it has to be expressed in terms of the primitives of your computational model. Only then you can actually evaluate for each argument of that function the value of the function at that point. There is a huge school or a methodological thought which actually looks upon functionality as the primary motivation for writing programs. Algorithms are simply intermediate steps which are essential because the jump from a function to some computational primitives is a very large job. You require some coarse-grained intermediate steps to start from the function. Refine each of the computational steps till you reach the primitives of the computational model and that should be program. But the point is that theoretically speaking every program can be looked upon just as a mathematical function. Very often that is also a convoluted way of looking at it though it is theoretically sound. There is another property which is that very often you are really looking for the invariance or the other properties that are either created by the program or the invariant properties that are maintained by the program.

So, you can look upon programs as relating various data objects which the programs manipulate but the way they manipulate them so as to maintain certain invariant properties or to create fresh properties out of the data is also a way of looking at it. You can think of a program as completely being defined by a set of properties. An algorithm then, which is somewhat more abstract since it is an intermediate step between a function and a program, can be thought of as maintaining some subset of the final set of properties that you should achieve. You are gradually adding to the properties that you require by refining your algorithm in various stages. That also means you are getting into a finer and finer level of detail when you write a program.

Theoretically, both the ways are tenable because firstly, every mathematical function is also a relation and therefore any relation can be thought of as a property or as a collection of properties. Any relation or a collection of properties can also be thought of as defining some classes of functions. Even from the point of view of the user of a programming language, the question is what exactly does a program represent? It is some concrete notation for something in your mind but is that a function, a collection of properties or a collection of invariances? That is a matter of view point. The main important point is that it really does not matter because all these view points can be considered equivalent. You can always represent any sets of invariant properties as functions, any sets of function properties as relations or predicates. The attitude that most people would take seems more logical if you were to look at a database program. To look upon a huge database program as a function from some domain to some range might be mathematically correct but in a database system you are trying to maintain certain consistency properties which is in some sense certainly one of the primary motivations in the maintenance of databases. It involves going through transactions and maintaining consistency. So, you probably maintain some invariant properties and some consistency properties.

You can also look upon programs and data in a different manner. For example; most of the normal programs can be thought of as filtering data through a program and the transformation of data as the program executes. But in the case of a database, it somehow seems that it is more like a program moving through a mass of data and going through some transformations. There are various view points you can take and some view points seem more tenable as in certain problem specific areas and some other view points seem more tenable or more convenient in some other problem specific areas.

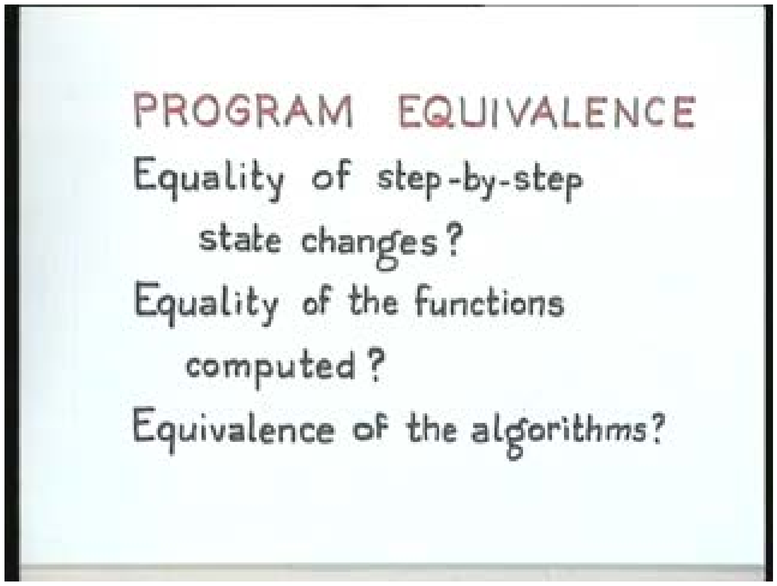
Let us take an implementer's view of a program. While the implementer's view of a program is that he should somehow specify state changes, for an implementer the computational model is already present; a user need not even worry about the computational model except that he must know what there is and what he wants. If you look at the ultimate customer of some software he is not really interested in the state changes the program goes through or in the state changes of a data item. He maybe more interested in it implementing a function or maintaining a relation or a property or creating a fresh property out of some massive data.

For example; a completely random sequence of items has to be sorted. It should create a property of a sorted list out of the massive data without changing the data. From the point of view of an implementer we might actually be interested in the step by step state changes. The next question is what constitutes program equivalence. One view point is that the step by step changes that take place should be the same. Given two programs the state changes that take place should be the same but that is not really right because, as I said, for any particular mathematical function there might be any number of different algorithms and those various algorithms might use various different kinds of data. There might be different data representations for example; if you take a simple problem like generating the first hundred primes, I can have different algorithms. One is that standard cyavophrathenis.

For example; there are fairly accurate estimates of prime generation, the n +1th prime is usually less than or equal to 2 $\times$n th prime. In order to generate the first hundred primes or first n primes you can take a large Boolean array and just knock out all the composites using the cyavophrathenis algorithms or you can follow an algorithm in which you start with a prime 2 and systematically generate the next prime till you reach your end primes. For each of these algorithms there are almost an infinite number of programs which implement those algorithms.

If you look at the problem as generating the first hundred primes and outputting them then there could be a variety of algorithms of probably different complex days, of different space requirement and for each of those algorithms there is an infinite number of programs which implements those algorithms. Step by step state changes is not an interesting way of determining when two programs are equivalent. Two programs will still be equivalent if the functions they represent are the same. Note that it is possible to write the same function in several different ways so the equivalence of functions is also important. We will take the simplistic view that two programs are equivalent if they compute the same functions.

[Refer Slide Time: 26:25]



PROGRAM EQUIVALENCE

Equality of step-by-step state changes?

Equality of the functions computed?

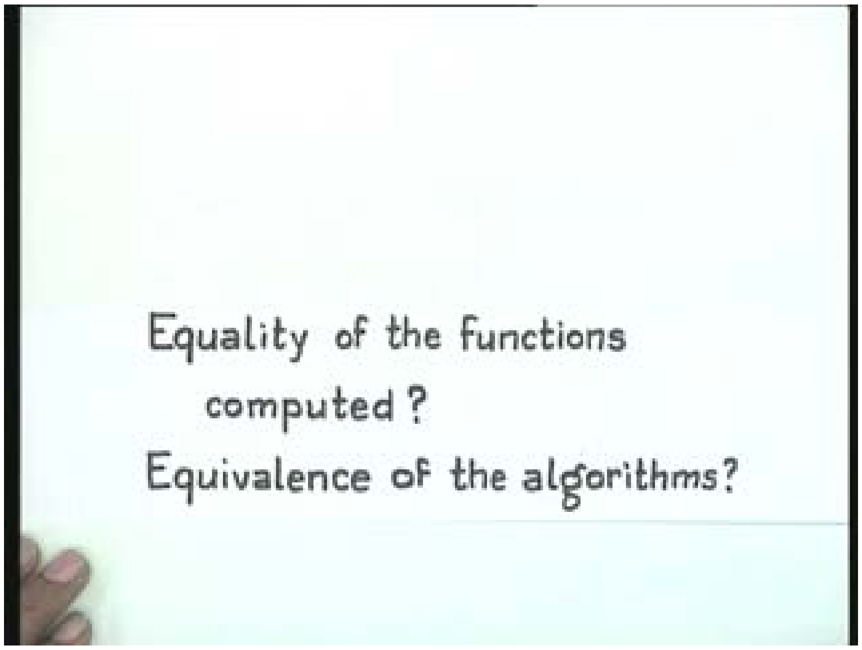
Equivalence of the algorithms?

In the case of interactive programs you really cannot think of an interactive program as just one large function but you can think of it as a sequence of input output behaviors. Between any two successive interactions there is a function and we might think of program equivalence in the case of interactive programs as really a form of black box equivalence in which given two programs regarded as black boxes for the same kinds of inputs you get the same kinds of outputs and then you would consider the two black boxes to be the same. They need not be identical because they could implement different algorithms or they could both implement the same algorithm in several different ways and so as far as program equivalence is concerned we can take the view that the input-output behavior should be the same or the function that is computed (if it is not an interactive program and if it has a single source of input and a single output) by the program should be the same. Going a level lower you have this notion of equivalence of algorithms.

You have algorithms which are not equivalent in the sense that one is either space-inefficient compared to the other or time-inefficient compared to the other etc, but implicit in all this is that the two algorithms actually implement the same function. You can compare two algorithms only if they both implement the same functions or the same input-output behavior. Otherwise it is completely meaningless to say that one algorithm is better than the other. It is meaningless to say that the binary search algorithm is better than a quick sort algorithm because they are really for different functions.

You can compare two sorting methods and compare two searching methods which ultimately have to give the same function but you cannot compare algorithms for different functions. When you look at this equality of functions computed, the equivalence of the algorithm is just a comparison of efficiency for the same function let us say of the same input-output behavior. When we are interested in program equivalence our main view would be that two programs are equivalent if they implement the same function. This is what you might call the correctness problem.

[Refer Slide Time: 30:57]

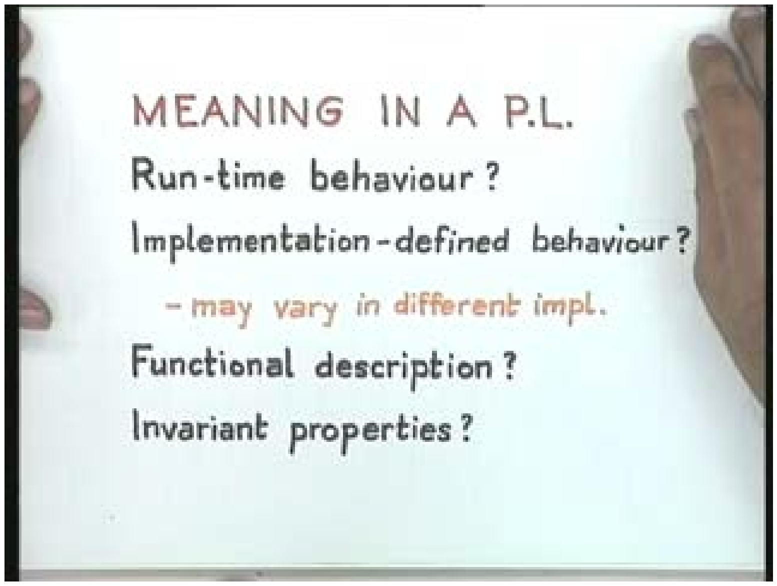


Equality of the functions
computed?
Equivalence of the algorithms?

The correctness problem is that, given a function is it possible to show that a particular program actually implements that function? The program equivalence problem is given two programs claiming to implement the same function do they in fact implement the same function? It is only after you have gone through the correctness problem that you can really get into the efficiency problem. The efficiency problem is which program is more efficient than the other and in what way? What applies to algorithms also applies to programs at a finer level of detail. They may be incomparable because some program might use a large amount of space but do it in a very short time; another program might use a large amount of time but use very little space which could be a constraint in certain situations. Even then they may not be comparable but at least the idea of comparing them can only come up if you have already proved that they both represent the same function.
We can take the implementer's view of being able to specify step by step state changes and abstract out the function from that in order to decide whether the correctness problem is being solved for a particular program. The problem is that all of these have to be mutually interrelated. Whether it is the user's view point or the implementer's view point, they all have to be equivalent in some way.

So, you have to be able to abstract out and prove properties about the program in order to show that the program is correct against a specification. You can compare two programs only after you have proved that they are equivalent and you can compare them for efficiency. Then regarding the meaning of a program, since a program is a concrete object generated by syntactic grammatical rules and there are an infinite number of programs generated by a finitary process of generation, the question of what constitutes the meaning of a program has to also be derived from some finitary object which is the construction of the program from the syntax rules. There are an infinite number of programs in any programming language and they represent various kinds of functions and the only way to be able to determine the meaning of a program is to be able to use the syntax as the basic framework from which the meaning of the program is to be derived, whether it is from the user's point of view or from the implementer's point of view.

[Refer Slide Time: 32:20]



MEANING IN A P.L.

Run-time behaviour ?

Implementation-defined behaviour ?

- may vary in different impl.

Functional description ?

Invariant properties ?

As far as the meaning of a programming language is concerned from the implementer's point of view, it is closely linked to how you are going to derive meanings of individual programs. A meaning for an individual program will have to be derived from the construction of the program. The meaning as far as an implementer of the language is concerned is the kind of state changes that are necessary in order to successfully implement each construct. After all only the syntactic rules are finitary in that programming language and an implementer has to cater to an infinite number of possible programs that might be written by users of the language which means that he has to be able specify the run time behavior of all his programs and the only way to do that is to use the generation process of the syntax.

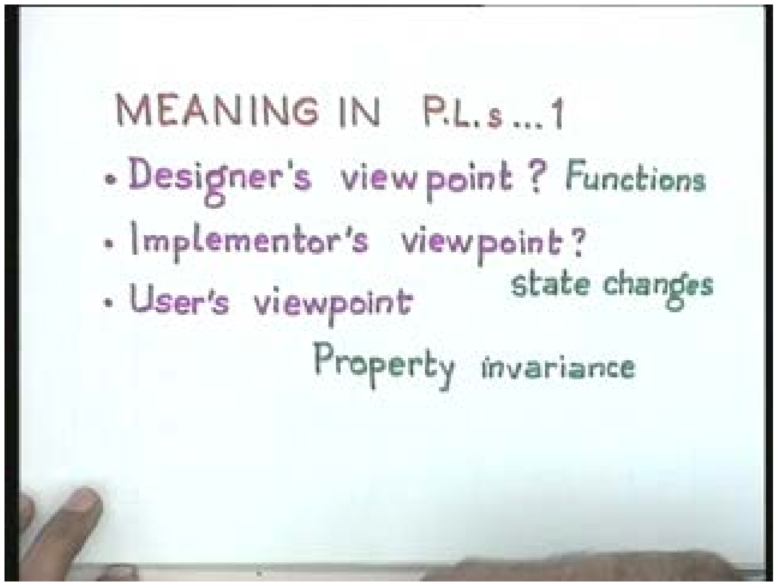
Using the syntax is absolutely essential. The syntax forms the basic framework for an implementer to specify the execution of an arbitrary syntactically valid program in the language. But as I said, the syntax from the point of view of an implementer is very detailed. It avoids ambiguity; it takes precedence of operators in account. Sometimes in order to suit the parsing algorithm, the grammar is changed in a certain way to make it convenient to implement etc. However, there is an abstract syntax for this language which is a common platform of understanding between the implementer and the user of the programming language. As far as the user is concerned he should be able to derive the functional description of a program again from the abstract syntax.

Firstly, the abstract syntax and the concrete syntax should be equivalent in the sense that if you look at abstract syntax as a collection of trees then the concrete syntax should generate exactly those trees. That is an implementer's problem. The abstract syntax is given a meaning, which should be captured by the implementer, on a phrase by phrase basis using the generation process of the abstract syntax, only these various productions are finitary. You should be able to give the meaning of each construct in the language, whether you are looking at it from the point of view of a user or from the point of view of an implementer, in terms of the syntax that defines that construct and you should be able to define how constructs are composed to form larger constructs. You should be able to define the meaning of these larger constructors also in some form of isolation and that should form the treaty between an implementer and a user, otherwise one of them is going to yield unexpected behavior.

We have already seen that syntax is a fairly complex entity and can be quite intimidating. A normal user is not going to be able to come to grips with the fine level of detail of the syntax which the implementer requires. So, the implementer should ensure that whatever syntax he uses, faithfully captures the same set of trees that the user's view of the abstract syntax captures and secondly, he should provide an execution behavior which matches exactly the functional behavior that the user has in mind for each construct. Both the user and the implementer should produce equivalent functional behaviors, only then can you avoid misunderstandings. That would also ensure that there is a meaning for the programming language completely independent of any other considerations.

The abstract syntax will form our basic framework for giving the meaning of a programming language. We will use the abstract syntax and we know how to convert abstract syntax into more concrete syntax into implementation convenient grammars. But the point is that there has to be an understanding between an implementer and a user as to what is expected of each construct in the programming language and both have to derive the final behavior of the program from the behaviors of each individual construct in the language.

[Refer Slide Time: 39:02]



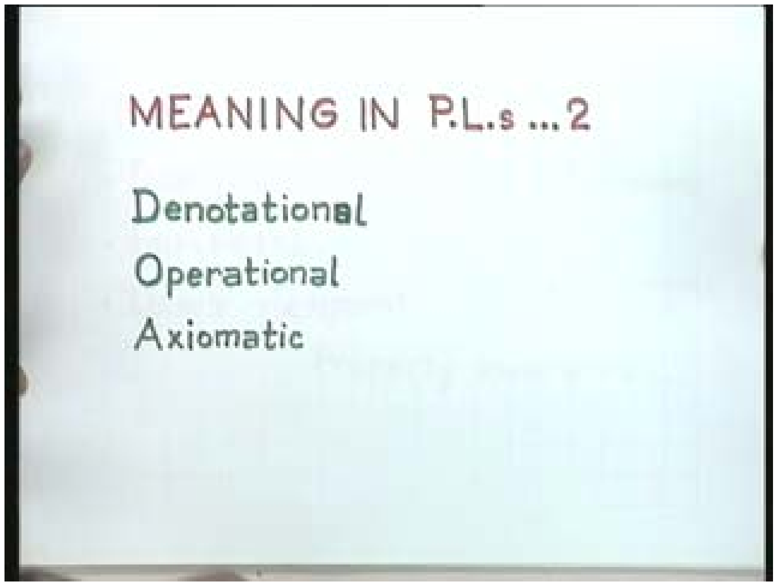
MEANING IN P.L.s ...1

- Designer's viewpoint ? Functions
- Implementor's viewpoint ? state changes
- User's viewpoint Property invariance

When we look at meanings in a programming language, the language designer's view point is that he is really providing a set of high level primitive functions. The idea of designing a high level language itself is that your original machine is too primitive to admit off convenient programming. Convenient programming means that I have some mathematical function and the machine operations are too small and too atomic for me to conveniently come down to that level and program every function at that level. So, as a language designer I provide a set of high level primitives which may be conveniently used by some user to define those functions he is interested in.

The designer's view point is that he is providing a set of high level functions for a user. The implementer's view is that these high level functions somehow have to get translated into the low level functions of the machine and so the step by step state changes actually define the implementation. The user's view point might be that he is not concerned with all these high level functions but just wants to know how he can maintain these properties. All these three view points have to gel together. We will not go into great detail about these three view points.

[Refer Slide Time: 41:05]



MEANING IN P.L.s ...2

Denotational
Operational
Axiomatic

These are the three view points from the subject of semantics of programming languages. One is called the denotational semantics. Its viewpoint is purely functional. It just says that this construct denotes this function. There is an operational view point which actually is the implementer's view point. It is the collection of high level translations that the implementer provides for the programming language and the last is the axiomatic which is what presumably a user should use and all these three view points should somehow be equivalent. The equivalence of all these three viewpoints is called the full abstraction problem.

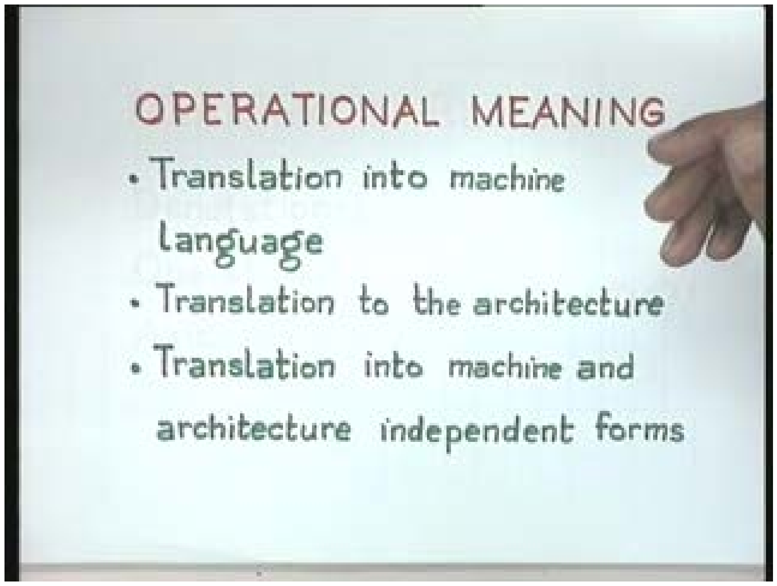
The main problem in the 60s and 70s that people noticed (which persists) was that most programming languages do not really conform to the reference manual of the programming language or that the reference manual for the programming language provided by the designer is not sufficiently detailed. Many implementers would read the reference manual and feel that it is not sufficiently detailed in describing the functionality of certain features of the language and very often the implementers had to take independent decisions. Many semantic issues actually moved into the area of pragmatics. The result was that a large number of programs had to be either patched up or rewritten when they were moved from one machine to another and because there was this misunderstanding between the implementer and designer.

Secondly, in the early languages there was no reference manual. There was complete chaos as far as implementations were concerned and many users doubted if there is some level of detail which is not present in the reference manual for which the implementer has to take the decision. The user's view could also be that some other decision was taken. Unless you try out particular implementations you do not know what decision has been taken. The problem of full abstraction actually persists. Most languages are defined in some loose ambiguous terms and implementers are forced to take certain decisions because the language manual is silent on the issue which it is most often. It is of importance to implementers and it is often ambiguous to a user what exactly the construct represents. He is not able to derive what behavior a combination of those constructs would yield because he is not able to infer the meaning. There could be one of several possible meanings actually implemented.

A classic case of this even now is the Ada programming language which is fairly recent in the sense that it went through a 10 year gestation period starting from about 1976 when the US department of defense felt that it had various installations all over the United States and abroad but the programs were not working. Programs could not be moved from one installation to another. There was a variety of languages many of which were created by committees within the department of defense itself and variety of implementations for each of those languages many of which were created by the sub committees of the implementation committees and there was complete chaos. Many of the departments of defense programs are old and they cannot easily rewrite them because the price of competent programmers is very high. What they decided to do was to embark on a unified language for all matters concerning defense programming except for the business programming.

The departments assumed that COBOL which is also created by a committee is good enough for a business kind of environment and for all scientific embedded systems, real time systems and control systems they would have one unified programming language.
They went through this process of design over ten years by actually deciding that perhaps they were incompetent themselves so they should sub contract it to somebody else and if you see the Ada programming language reference manual, which came out sometime in the 80s, it is extremely voluminous and extremely ambiguous on various issues and there were no implementations at that time. You could not even experiment but you had to take certain implementation decisions. It was silent on a variety of issues, far too detailed on a variety of other issues and they specified that there should be no subsets supported. So, the resulting document is a huge document which is very difficult to understand. The syntax is clear but the semantics is not.

[Refer Slide Time: 49:30]



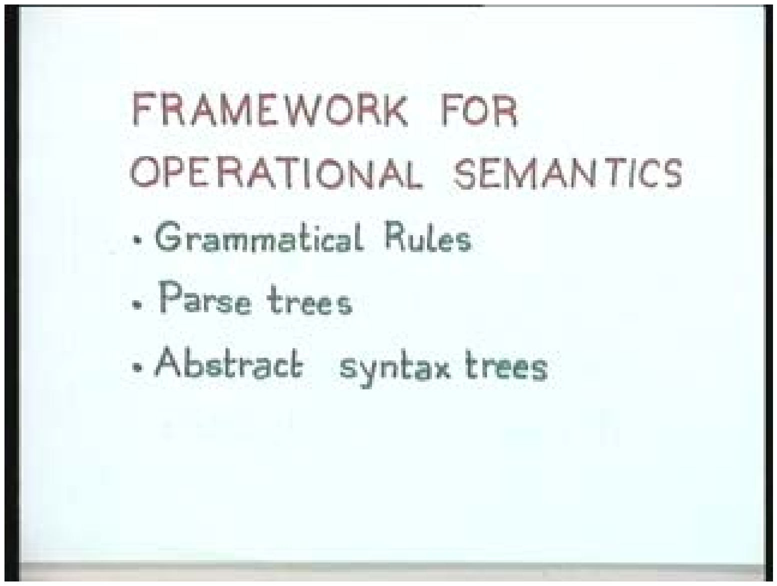
OPERATIONAL MEANING

- Translation into machine language
- Translation to the architecture
- Translation into machine and architecture independent forms

We will be primarily concerned with operational semantics and that is the implementer's view point from a broad informal understanding of what function the construct represents. For each construct we will give an operational meaning, however, we do not want to give any kind of meaning which is machine-specific or architecture-specific. The reason is that if you have to give an operational meaning again there are several levels of granularity at which you can give an operational meaning and we would not like to be restricted by either a machine or architecture.

We will just try to give meanings in machine and architecture independent forms as far as possible. It used to be a common practice sometime ago for a language reference manual to define an abstract machine into which they translated each construct in an effort to give a meaning to each construct of the programming language. However, it is being realized more and more that it does not suit a lot of people. It does not suit users. It does not suit designers. It is sometimes too specific. So, a nice method of operational specification has now emerged in the last 10 years which is machine- independent, architecture-independent and is still able to give a user step by step state changes and it also performs abstractions so that you get purely functional descriptions. You can mix and match your level of detail in this operational semantics. You can become extremely machine-specific or you can become almost denotational in whatever you specify.

[Refer Slide Time: 52:10]



FRAMEWORK FOR OPERATIONAL SEMANTICS

- Grammatical Rules
- Parse trees
- Abstract syntax trees

We will use this as a general method for specifying semantics. The meanings of constructs for our purposes basically are the runtime behavior of programs. Any kind of semantics of a large program has to be derivable from the semantics of the smaller constructs which make it up. This means an essential portion of the semantics hinges on being able to use the abstract syntax trees or the grammar rules in order to be able to specify the semantics. Since we understand now how we can go from an abstract grammar to more concrete grammars in some fairly simple fashion we would not like to be hindered by parsing strategies or scanning strategies unless it is absolutely essential.
We will just consider abstract syntax trees and specify meanings of abstract syntax trees. The compromises we make are that the abstract syntax trees form the common level of syntactical understanding between a designer and a user and an implementer. Our operational semantics while it is closer to the implementer's view can be abstracted out to provide a user's view and it is sufficiently abstract not to get into the details of machine architecture or machine language.

It is a reasonable compromise in a general course like to give an operational meaning which is architecture-independent, machine-independent and which depends only upon the abstract syntax trees because the abstract syntax trees provide a method of induction.
Induction on trees is a powerful method which you can use to derive meanings of complex constructs from simpler constructs. We will define for PL0 initially without declarations as semantics to familiarize you with the general method of operational specification.