

**Principles of Programming Languages**  
**Prof: S. Arun Kumar**  
**Department of Computer Science and Engineering**  
**Indian institute of Technology**  
**Delhi**  
**Lecture no 1**  
**Lecture Title: Introduction**

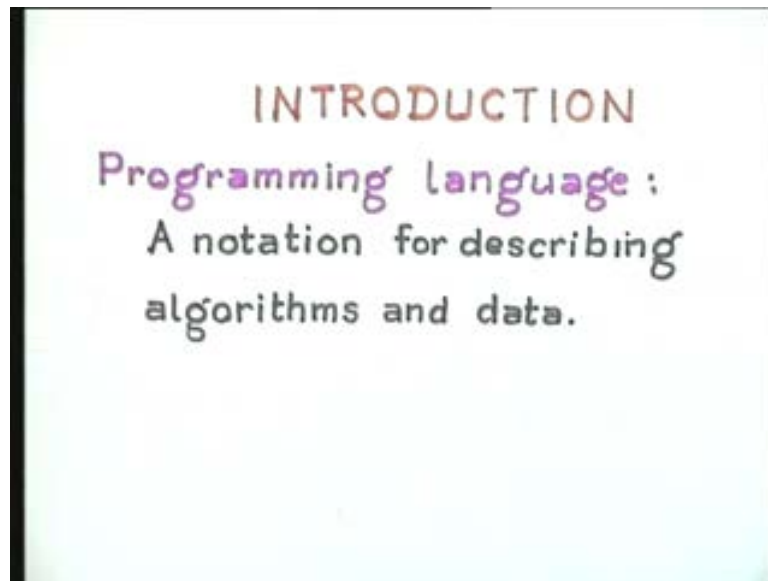
Welcome to programming languages. We will just do a few elementary concepts and broad classifications of programming languages without going into too much detail. This is the first lecture so let us just look at the notion of a program or a programming language. You are all familiar with the notion of a machine or a computer and it is what one would call a bare machine. It just has a piece of hardware which is usually in binary. It can be thought of as a whole lot of switches connected with complicated circuitry. The memory, the arithmetic unit etc. consists of switches activated one way or the other and it is going to be a big problem operating those millions and millions of switches. What you have in a bare machine is a language consisting of binary strings.

Binary string in what is known as the Von Neumann architecture is called the stored program concept. Both data and instructions have the same format and everything is a binary string depending upon how you look at it; it is either a data item or an instruction to execute a command. The Von Neumann concept means that programming such a machine basically helps you to interpret certain sequences of bits either as data or as instructions to manipulate some registers or store into the memory, perform some arithmetic or logical operation etc.

In general even that language what we might call the machine language can be called a programming language. Let us take a very general view. What is a programming language? A programming language is just some notation for describing algorithms and data. In general we could consider a programming language to give you a means of representing algorithms and data structures and when you have a representation of algorithms and data structures presumably you are able to perform your manipulations.

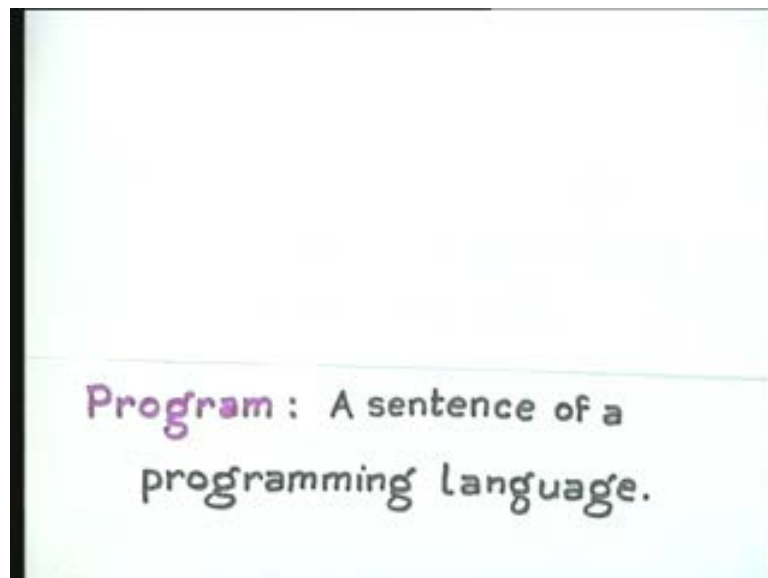
The first point about a bare machine is that if you are going to use the machine language itself then there is really no fundamental difference between the algorithm and the data, which means a sequence of instructions could just as well be regarded as a sequence of data items provided they have some circuitry that could also be executed as an algorithm.

[Refer Slide Time: 05:01]



In principle, you could execute even a sequence of data items as instructions by interpreting it suitably. The first distinction we would like to draw is between what constitutes the data item and what constitutes an instruction. Let us take a much more high level view. We are no longer in the fifties when the early machines came in and you had to program in machine language or assembly language. So, we will just look upon a programming language as a notation for describing algorithms and data.

[Refer Slide Time: 05:05]

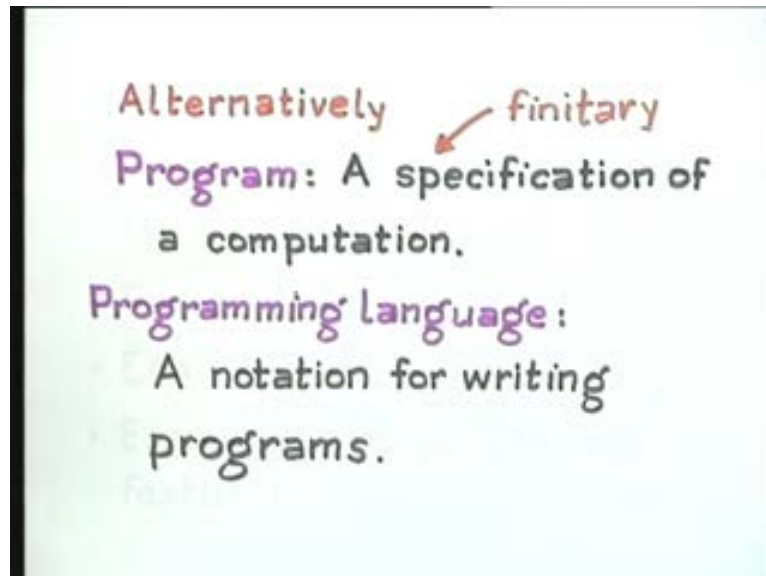


We could look at a program as just a sentence in this language. It is a language like any other language. It has certain rules and certain well formed sentences. A program is just some sentence of a programming language. A program is not necessarily an algorithm simply because you might have a well formed sentence which is not very meaningful.

For example; the program could be a non terminating program in which case it is no longer an algorithm. It is therefore important to realize that an algorithm is a very abstract object that does not have any concrete form.

Only what is put down as a program is concrete. The only concrete object that you can have is a program. The notion of an algorithm itself is an abstract entity which requires a concrete representation in the form of a program and if a program is a sentence of the programming language then what you require really is a programming language.

[Refer Slide Time: 06:50]



Another alternative way of looking at the notion of a program is to think of a program as a specification of a computation. This means we have some notion of what constitutes a primitive step of the computation and the program gives you a finite representation of possibly an infinite sequence of steps in a computation process.

The emphasis in all these cases is in the nature of a finitary specification. A program should be a finite object by itself. A programming language itself is not a finite object because there are an infinite number of programs that are possible but each program itself is a finite object because it is just a sentence of the programming language.

Then we might think of a programming language, if you look upon a computation and the steps in a computation as the most basic feature, just as some notation for writing programs. In all these cases, we should emphasize the fact that this notation is important because our notation is to give you a finitary specification of possibly infinite objects.

We might emphasize that this is actually a finitary specification and these programs themselves as concrete objects are finitary but their effects could be infinitary. The moment you are trying to represent any infinitary object in a finite manner you require it to be machine understandable and you also require certain rules. Let us look at this process of essentially giving a finitary representation to what you might consider infinitary objects.

What kinds of infinitary objects are we normally concerned with? In the most general case an algorithm is what you want to represent in a program. An algorithm in the most general case is a function from some domain to some co domain. A function need not necessarily be finitary because the domain could be infinite and the co domain could be infinite. We might think of an algorithm in general as computing either a function or a method for computing some mathematical function or relation. These functions and relations could be infinitary. We are looking at infinitary objects as functions. Basically mathematical functions relations can also be considered functions. All relations could be considered functions. In general we will concentrate on trying to get finitary representations of infinitary objects and these infinitary objects are really functions.

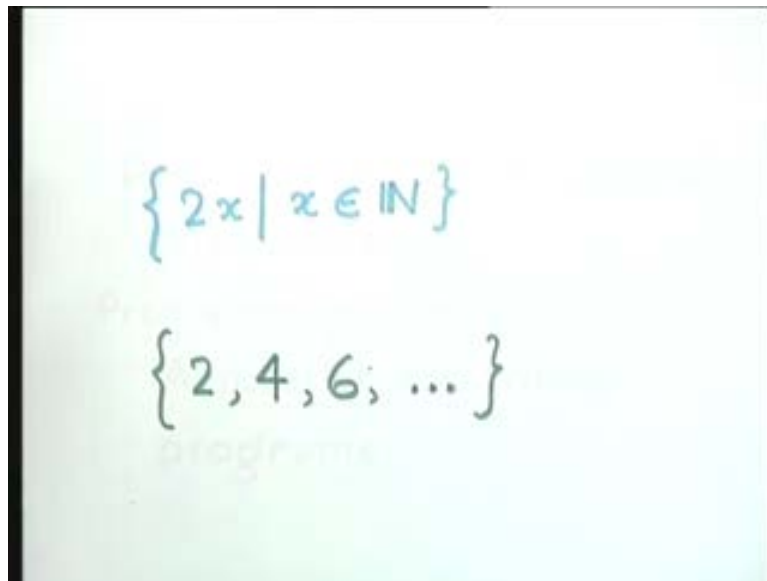
You can think of the whole study of programming or computation as trying to compute or trying to give finite specifications of computation steps of abstract mathematical functions. However, if you look at mathematics itself it has a fairly rigorous notation. You could think of mathematics itself as a sort of programming language except that it has one important drawback. The drawback is that it does not specify the primitive computations that are possible within the mathematical language.

Normally when you are talking about an algorithm to compute some function, you have implicitly defined a set of primitive functions or primitive computation steps in terms of which you are going to express this algorithm. One obvious case in which a lot of mathematics does not fit into the general framework of a programming language is the representation of infinite sets. If you look at them, the standard point in school is to say that you can either represent a set in a roster form or in set builder form.

Roster form just means enumerating a list of elements and a set builder form essentially means giving a predicate which the elements of the set should satisfy. The main difference between the roster form and the set builder form, also called a definition by abstraction comes up for infinite sets. Supposing in the case of infinite sets you want to specify the set of even numbers so you open braces, you write 0 or if you do not include 0 then you write 2, 4, 6... That is where the inadequacy of mathematical notation comes because you are not interested really in any underlying computation process.

As far as mathematics is concerned a large part of it is just that the existence is more important than a computational method. Whereas the set builder notation or the definition by abstraction gives you a finitary specification so that you can represent the set of even numbers through a notation which consists of braces that consists of a bound variable, and a predicate in terms of the bound variable.

[Refer Slide Time: 14:38]



The image shows two handwritten mathematical expressions on a light blue background. The first expression is  $\{2x \mid x \in \mathbb{N}\}$  and the second expression is  $\{2, 4, 6, \dots\}$ .

A typical definition of even numbers would look something like this. Take  $2x$  where  $x$  belongs to the natural numbers. If you look at  $x$ ,  $x$  is like a locally declared variable. In fact this is a sort of declaration of  $x$  and this  $2x$  is a property that the element of this set should satisfy.

Here is a case of our finitary specification as opposed to this infinitary specification. In fact this is a finitary specification in more ways than one. Firstly, this represents a logical predicate expressed in first order logic in a finite sentence of the first order logic. You might consider this as a succinct finitary specification of essentially an infinitary object, the even numbers. Whereas this is really open to many. This is really ambiguous in the sense that it is not at all clear from this enumeration what should be the next one. You are implicitly using human intelligence and human understanding or human ability to perform induction to claim that the next number would be eight but we cannot at all be sure that the next number should be eight. There might be other patterns.

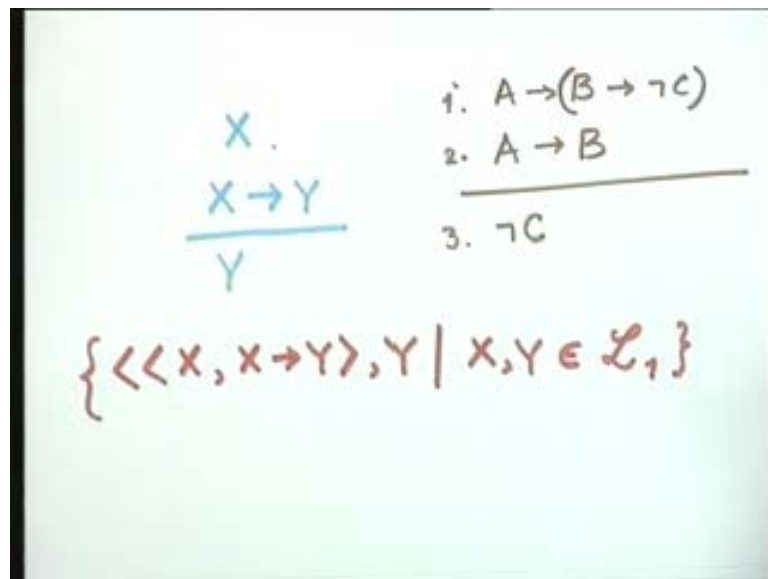
It might satisfy other predicates whereas this is what one might call an accurate succinct finitary representation using just the language of first order logic built up on a single binary predicate on sets, which is the binary predicate this belongs to.

A lot of what we are going to do is also going to be related to the language of logic in some ways. You will see the analogies between programming languages and logic as we go along. The main motivations of logic are really of a slightly more abstract nature but programming languages derive mainly from logic in the sense that a language like first order logic does not allow you the freedom to write these dots and there is no such thing. You have a method of construction of predicates which is always finitary.

You have rules of inferential logic which are always finitary or they might be infinitary like if you have axiom schemas, or rules like modes ponens etc. They are finitary representations again of infinitary objects. Further in a logical language with axioms and rules of inference it is implicitly understood that those axioms and rules of inference are such that there exists an

algorithm which when given any instance of the hypothesis of these rules, should be able to tell you whether the conclusion of the rule is a valid inference.

[Refer Slide Time: 19:25]



Let us take a simple logical rule like modes ponens. You have a predicate  $X$ , you have a predicate  $X \rightarrow Y$  and you have  $Y$ . This rule actually specifies a three tuple pair of this form where  $X$  and  $Y$  belong to (let us say) the language of first order logic, which we may call  $L_1$  as opposed to proposition logic which we may write  $L_{NAUGHT}$ . You take two sentences of first order logic and if they have this pattern then call one sentence  $X$  and the other sentence has the pattern  $X$  conditional  $Y$  then you are able to infer  $Y$  and you cannot have all rules of inferencing logic. They are finitary.

There are also finitary specifications and something that is absolutely essential is that it is decidable by an algorithm whether a certain step in the proof of a logical statement was derived by an application of a rule of inference on some preceding steps.

If you claim that you have some predicates of the form  $A \rightarrow B \rightarrow_{NAUGHT} C$  and then you derive from these premises if you were to claim that by the use of modus ponens you can infer  $NAUGHT C$  then there has to be an algorithm which when given these two as inputs will be able to tell you whether this is an instance of an application in these two definitions. In this case the algorithm should actually tell you that it is not an application in this rule of inference.

It should be able to give you both yes and no answers in finite time. Most programming languages that we will study will have a lot of their motivations actually derived from logic. A large part of logic was actually concerned with the notion of how much of mathematics is actually doable by a machine and what kinds of theorems in mathematics can be actually proved by algorithms by a machine whose basic primitive operations are that they are able to do pattern matching and substitution. This is an instance of doing pattern matching and substitution.

An inference rule is really an infinite object. A relation of this kind is a finite representation. A proof is a finite object. A theorem itself is a sentence of logical language and is a finite object representing possibly an infinite number of instances.

The finitary nature of all these will actually influence the nature of our logic. For example; you cannot give axioms and rules of inference which are infinitary in a logical language. Everything that is infinitary should have a finite representation. There are of course infinitary objects which will have no finite representations. They are clearly not going to be part of our computational process. For example; generating an infinite sequence of random numbers, not pseudo random numbers but pure random numbers is not a computational process period. We are interested in those kinds of infinitary objects which somehow have finitary representations. It can be infinite sets represented as predicates like unary, binary, and ternary but some finitary sets with a finitary representation. We are interested in infinitary computational processes which have finitary representations. We are interested in programming languages which allow for finitary representation of inherently infinitary objects.

Let us go ahead. This much of philosophy is perhaps sufficient for the moment but it is important to realize that right from nineteen hundred when the mathematician David Hilbert posed this problem to the congress in mathematics the main emphasis of logicians has been to try; to define the notion of an algorithm, to define the notion of the computational process, to be able to exactly define what is possible by a computational process and what is not possible by a computational process.

Everything that is possible by a computational process should have a finite representation and anything that is infinitary is not part of the computational process with some restrictions. If we just come down from logic a bit then we can look at a logical language itself as a mathematical object for example; there exists only a finite number of rules for generating an infinite number of sentences of that language.

Let us take a language like first order logic. We have only a finite set of formation rules, which allow you to generate an infinite number of logical sentences. A finitary nature of the rules also gives you an algorithm to check whether a given string of symbols is a syntactically valid sentence of the logical language.

An important element of that logical language is that the generation process should be finitary. There should be only a finite set of rules and there should be an algorithm which can clearly tell you whether a given sentence is a well formed sentence of the language. If you look at propositional logic, it does not allow you to specify infinitary objects that we require for applying propositional logic to some area of mathematics like number theory.

It does not allow you to specify infinite sets or certain properties of infinite sets easily. Very often, an extension of propositional logic to first order logic, which allows you to do this in a finitary way is the use of quantifiers. So, you can for example specify the whole of set theory in first order logic, the axioms of set theory and the predicates that are valid for all possible sets. By set theory I mean axiomatic set theory in the sense that we do not assume numbers or any predefined set of objects.

The only notion is the notion of a set. You generate all sets, numbers and everything from the notion of an empty set and a single binary predicate called belong stood. They have these

formation rules and so we are interested essentially in capturing infinitary processes within finitary languages. You can see a progression of ideas. Firstly, there is pure mathematics which is platonic in nature in the sense that the notion of a computation itself is not an important element of the formal discipline of mathematics. Then you have logic which actually gives you a loose notion of what is possible by a machine and what is not possible and allows you to specify infinitary objects in some finitary ways. Lastly, we have programming languages which specify with a great deal of accuracy exactly the primitive computational processes that you are allowed to use. A programming language also has to satisfy all the constraints of a logical language and in addition it should be consistent with what might be called the primitive computational processes.

For example; one primitive computational process that you must all have studied in school is that of ruler and compass constructions. There are only two primitive computational steps. We are able to draw lines with the ruler, mark off segments. We are able to use a compass to draw certain angles or to draw arbitrary angles. One impossible computation in this case is an algorithm using only this primitive concept to trisect an arbitrary angle.

For example; you are not allowed to use protractors etc. and you are not allowed to measure the angle. You can only prove that an angle is of a certain measure if you draw a line perpendicular to another line; with a construction proof it shows that it is perpendicular and then you bisect that. You can then claim that the bisected angle is let us say, forty five degrees. But given just an arbitrary angle to be raised from a point to be able to trisect it with just these primitive tools is an impossible task.

You might think of the algorithms of ruler and compass constructions as the only two computational processes you have in a programming language.

It is not machine readable. It is meant to be human readable so you write it in a loose fashion but essentially you use only those computations which are possible within the domain of Euclidean geometry which means you are not allowed to measure out angles yourself. You are only allowed to prove that a certain angle has a certain measure. You are not allowed to measure out lengths in terms of centimeters or meters. You are only allowed to measure out an arbitrary unit and take multiples of that arbitrary unit. You could bisect that arbitrary unit. You could trisect that arbitrary unit of length measure. You can therefore claim that it is actually one third of the unit you took but you cannot claim that you have constructed one by  $\pi$  of a unit of length, unless you can prove that just by this process you are going to get something that is one by  $\pi$  of a unit. Our programming language has ingrained in it a normal computational process which we associate with a digital computer.

It is not the last word because you could have other computational process such as the ruler and compass constructions. You could have analog computers etc. We are interested primarily in the computational processes associated with digital computers. We could look at even the machine language as a programming language but we are not really interested in machine language because it is a very simple sort of a language. It is very difficult to get any program right but the language itself is a very simple language and probably that is why it makes it so difficult to program and what we are interested in primarily are what are known as high level languages where the primitives of the computation or what you might say is the machine that is made available. Once we have implemented a language on a machine you could think of that as a machine of that language.

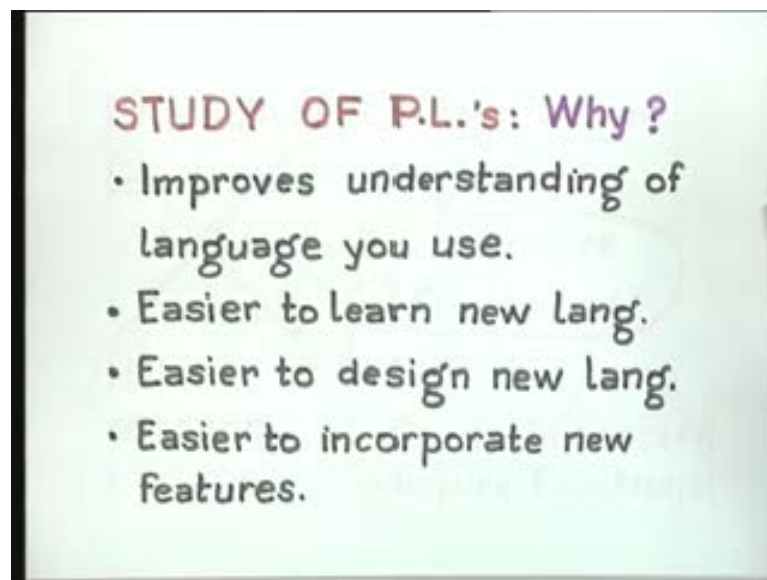


Supposing when we are doing PASCAL programming we are not really worried about the underlying machine language, the underlying architecture or about anything for that matter. As far as we are concerned what we have is a PASCAL machine. There is a level of abstraction at which PASCAL is the hardware machine language or it is just some software language. As far as we are concerned it is a PASCAL machine.

It is important to realize that we can actually take some bare machine and cover it up with layers and layers of software and think of just one abstract machine which gives us certain capabilities. If you look at the bare machine it gives you only the capabilities to manipulate switches to write programs in binary. If you look at a PASCAL machine it gives you no extra computational power but it gives you the ability to look upon the whole unit as a single machine allowing construction of complicated structured programs.

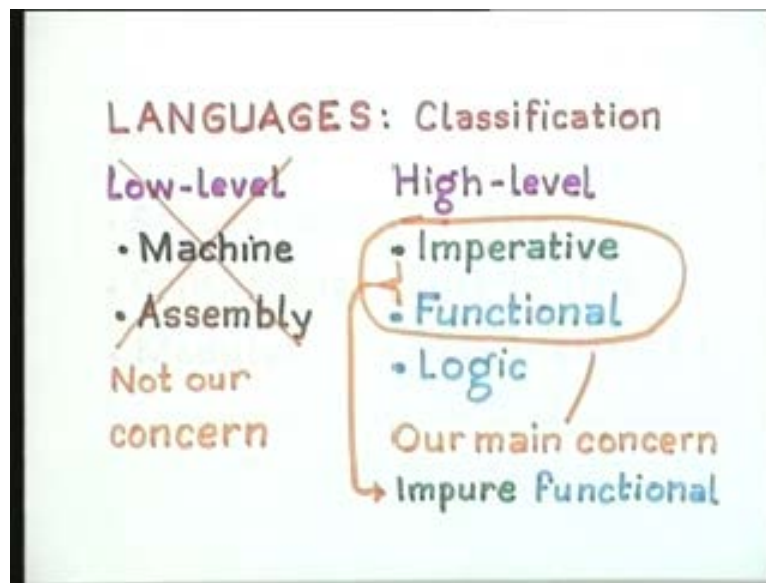
It allows various kinds of abstraction mechanisms, procedures, functions and it allows you to express differently from what the bare machine would have given you. Let us look at why we should study programming languages because all the time we are looking at the construction of some virtual machine and facilities that the machine gives us which we are not really interested in. We are interested in various kinds of features that are there. In the case of a bare machine you are interested in its architecture. If you have a PASCAL machine its architecture is really the features of PASCAL. If you have a LISP machine its architecture is really the features of LISP.

[Refer Slide Time: 39:20]



Our study of programming languages is mainly to understand why certain features have been included in the programming language. You want to understand for example how best those features could be used. If you want to understand how that language is implemented presumably you would be able to learn new languages easily. You could design a new language which is more important and perhaps you would also be able to understand the underlying implementations. May be you would be able to incorporate new features in a programming language.

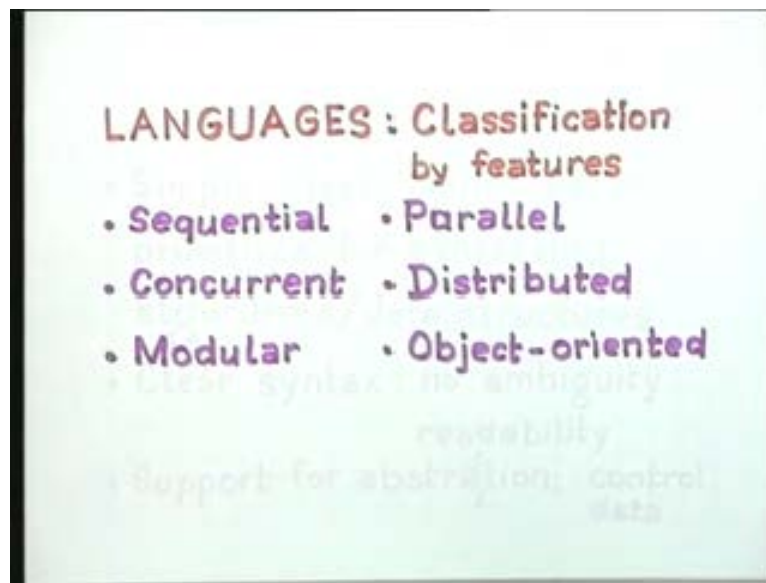
[Refer Slide Time: 40:19]



Let us just look at languages. We sort of classify what kinds of languages there are. Firstly, we have these low level languages, some machine and assembly languages which are not of our interest. You will learn about them in some course on architecture or organization but we are interested in primarily these high level languages of which we can think of three broad classifications: One is a class of imperative languages towards which most of the last forty years since the first digital computers has gone in their design.

Then there are functional or applicative languages. It is then possible to use logic itself as a programming language. You can actually mix up all of them and you can have impure functional languages. An imperative language means that it uses the notion of the command. It uses the notion of a state to change a state. So the commands change states. That is what an imperative language would do. A functional language is one which allows you to program something that is as close to mathematics as possible. We will get into these notions a bit more in detail later. A broad classification of languages is just in terms of high level languages imperative, functional, logic etc.

[Refer Slide Time: 42:27]

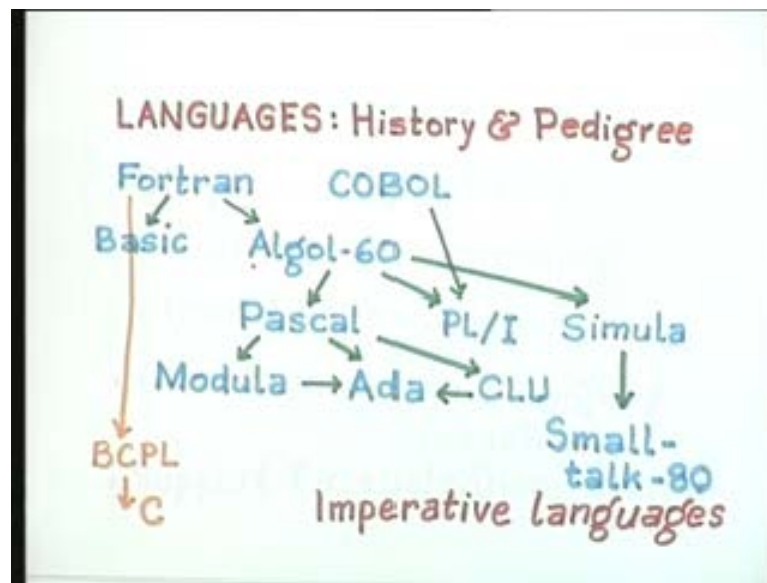


You could also classify languages by features and by features in the sense of what the most glaring feature in the language is. A large part of our languages are really what might be called sequential languages. Most of the languages that you have programmed in are purely sequential languages. Then you have parallel languages which are very often languages meant for certain specialized architectures. For instance you have a single instruction multiple data and you execute instructions in parallel and there are implicit methods to implement them in a parallel fashion. You have what are known as distributed languages. In the case of a parallel language you assume that there are so many processors which will execute the same instructions in a synchronous lock step fashion.

Most of the vector processors actually have sequential languages vectorized or made parallel like in the case FORTRAN 90, VECTOR PROCESSING FORTRAN etc. Distributed languages are those in which you actually assume that the different units of a program are going to lie geographically distributed across a network and they have to somehow co-operate to achieve some common task. Then in both parallel and distributed languages the notion of a process of a computational process into which a program is split is inherently or intimately related to the computational power to the number of units of computation. That number of units is of the CPUs that you have.

The notion of a process and a processor are really the same. You are writing one process per processor in both these cases. In the case of concurrent languages you are basically taking the notion of the process to be a loose entity completely different from the existing process which does not necessarily have to be mapped on to the existing processes. The notion of the process gets de-linked from the notion of the processor. You have other kinds of languages whose primary feature is that of modules of separate compilation and more recently you have what might be called object-oriented languages. These add extra features on top of the existing languages usually but there is something fundamental about the new feature that they introduce.

[Refer Slide Time: 46:48]



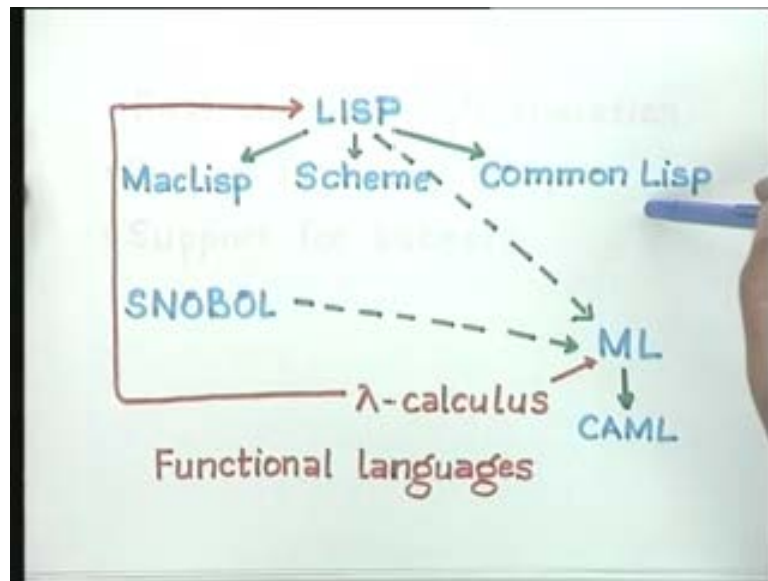
Let us quickly go through some of these languages. If you were to take the history of programming languages, you would find that there is a certain chronological dependence.

The first high level languages so to speak were FORTRAN which is mainly meant for scientific computation and then COBOL which is meant for business.

It was more verbose. It actually used full English sentences to represent computations. It made the first division distinction between data and program and was meant to use a large amount of data and do very low processing. They were IO bound programs whereas FORTRAN was meant for minimal IO and maximum computation. These languages gave rise to one important class of languages called the ALGOL like languages which came from the 'ALGOL 60' Report.

FORTTRAN also had its offshoots in basic. Then there were these ALGOL like languages. Among the ALGOL like languages you have PASCAL, PL1 and Simula etc. PL1 was an attempt at a unified language for both scientific and business commercial processing and from PASCAL you get extra features like Modula and Ada. From Simula you have got these object oriented languages starting from Smalltalk-80. All these and somewhere in a parallel stream you have BCPL and C. Actually BCPL was a transformation of a language called B which itself was a transformation of a language called A. The programming language 'C' was derived from BCPL by modification. Then when Smalltalk-80 came up object- oriented ness became a big buzz word. You had C++. That is briefly the pedigree of languages that we have covered. We also have functional languages.

[Refer Slide Time: 48:53]



Let us let us look at functional languages. Apart from these imperative languages you had basically the first functional language which was LISP from which we derived various versions MacLisp, Scheme Common Lisp. MacLisp and Common Lisp are really impure versions of LISP. When we understand functionality we will come to what we mean by impure versions but many of you have probably studied Scheme.

Scheme is a cleaned up version of LISP and is meant for LISP processing. There was also a language designed in the 60s called the SNOBOL which was meant for string processing. It allowed efficient pattern matching constructs to be programmed and these have actually yielded along with the emphasis on tied checking to a language called ML. It came up in the 80s and all these languages like LISP and ML were inspired by what is known as Lambda ( $\lambda$ ) calculus which we will study. It is the basis of all functional languages.