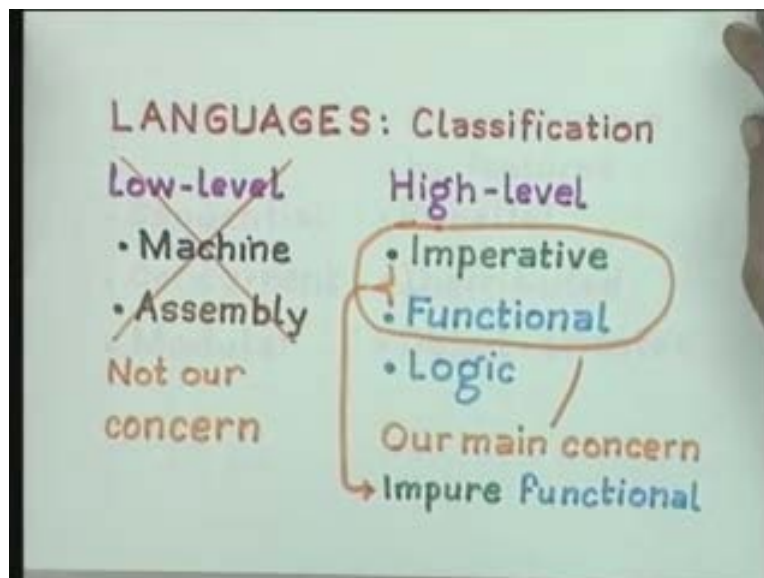


Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian institute of Technology
Delhi
Lecture no 2
Lecture Title: Syntax

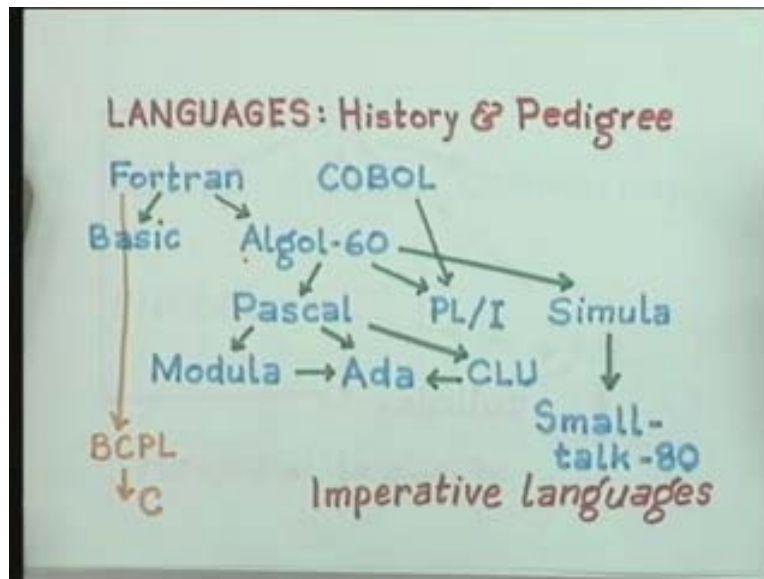
Welcome to the lecture 2. Let us just go briefly through what we did in the last lecture. We are mainly concerned with what might be called high level programming languages of which there are three kinds. Imperative and functional programming languages are the most important. Logic also can be used as a language but our main concern is with the imperative and functional languages. They are also similar in certain respects.

Imperative languages might be called state-based languages where state updation is the main action. However, functional languages are really value based languages much closer to our mathematical languages. The notion of variables in functional languages is very much like the notion of variables in mathematics whereas the notion of variables in the imperative languages is more like quantities in physics which can change over time such as acceleration and velocity. State based languages means that there is a state change along a time axis. Unlike physics we are not talking of continuous time. We are talking of discrete time in the case of functional languages. The notion of the variable is really a variable in mathematics which means that a variable is just a name for a value and it cannot change in time during the execution of a program.

[Refer Slide Time: 02:24]



[Refer Slide Time: 02:27]

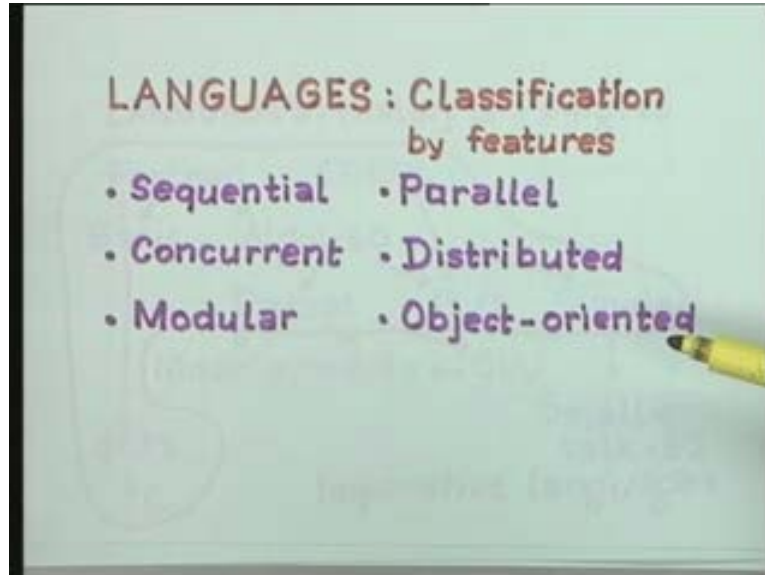


If we look at the history of languages, most of the work on high level languages is really concentrated on the imperative languages. There are hundreds and hundreds of programming languages. So firstly, it is impossible to study all of them. If you look at history, you will find that a large portion of the time during the fifties and sixties was concentrated on what might be called the basic features. This means that these represent the basic control structures in the basic data structures in order to obtain clean readable programs efficiently. Later once they were fixed in the 70s and 80s, most of the exploration of programming languages was in terms of what might be called features.

If you were to take Modula, which is just an extension of Pascal, its most important feature is that of a module. If you were to take Ada, it combines the module features of Modula and adds more features like concurrency as an important feature like in exception-handling, generics or polymorphism. Then you have CLU, which is a module based language very much like Modula but with a different syntax. But they are all extensions of Pascal in the sense that the basic control structure remains the same and then you extend the language.

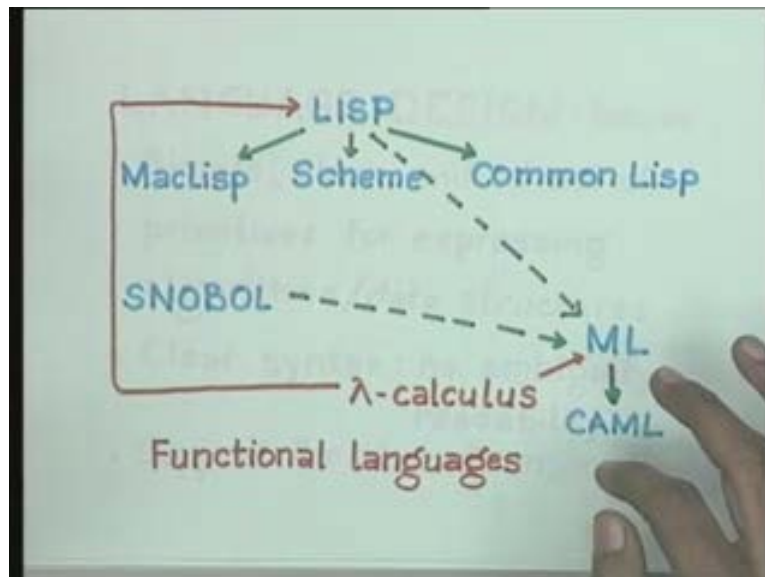
The basic control structures in these languages could be different too. They are not very similar except that these arrow marks denote the dependency in terms of similarity of even the basic control structures. Even though the Smalltalk-80 control structures or syntax are different from that of Simula, the basic extensional feature comes from a new feature of Simula which was extended and that was the notion of class or objects.

[Refer Slide Time: 05:00]



Most of the languages in the early 60s were sequential, whether concurrent or modular distributed or parallel etc.

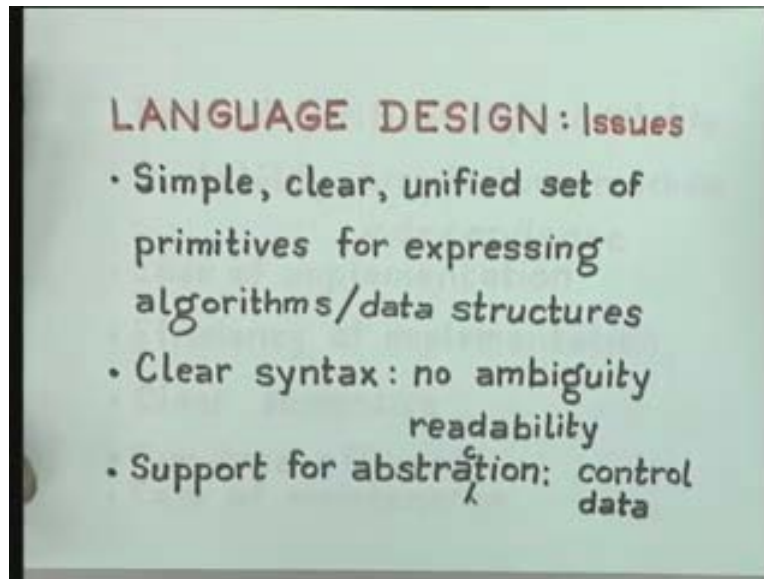
[Refer Slide Time: 5:50]



Nowadays by and large the exploration of most of the languages in the early 60s is mostly in terms of new features. I have listed the various kinds of features and in the current state of art, large amount of work is into trying to make them more efficient, more readable, more comprehensible etc. The basic functional languages also have this feature that the early functional languages were all an exploration of. They were very basic data structures and control structures. Languages like ML and CAML actually signify the addition of new features.

In fact the syntax of ML expressions etc. is much closer to that of Pascal than that of LISP. But philosophically, it is a more LISP like language because it is an applicative or functional language. It is not a state based language and it goes far beyond LISP in the sense that there are new features like the introduction of modules, introduction of exceptional handling, the introduction of very powerful data abstraction mechanisms and a type checking. LISP has no type checking at all.

[Refer Slide Time: 07:20]



Let us first study the basic features of languages and look at the issues in language design. Consider the kinds of issues you would really have to study if you were to design a language. One very major issue that the implementation of the language Pascal has taught us is that it is a good idea for a language to have a simple, clear and a small set of unified primitives for expressing all your algorithms and data structures.

One of the most convenient characteristics about Pascal or in general about what are known as Algol 60 like languages is that most algorithms are written in some crude variations of the dialect of Pascal or Algol system. So, the Pascal is a small language easily learnable which is why most people are taught Pascal initially. One issue could be that you should start with a small language with a small set of primitive operations over which you can build.

The next issue would be that you should have an absolutely clear syntax. There should be no ambiguity and you should be able to get highly readable programs.

It is very important to have readability meaning that we have to be able to read the source code of the program like a book and a very good reason for that is that no piece of software that you write is ever permanently fixed. Most pieces of software have bugs in them. Bugs might be detected years and years after their software has been commissioned. Somebody else should be

able to read the source code and modify it. In order to be able to read the source code, one should be able to understand the algorithms of the source code contained.

In fact consideration of efficiency comes much later. Readability is the most important aspect because it includes maintainability of the software. Normally, those who have written this software are not likely to be present to maintain the software. Software in general has to be maintained by somebody else, which means that the source code must be readable and that comes first. The second point is that over the years the software actually has more and more users who feel their need for its extension which is why you have to be able to extend the software by adding new features or by adding more conveniences for specific reasons.

Part of the maintainability of a piece of software is not just the detection and correction of bugs but also the extensibility of the software as years go by and more and more needs are felt. It is necessary for somebody who is usually not the original programmer or the original team that wrote the software to be able to extend it. For all that, readability is most important and secondly the language should provide what is known as a support for abstraction.

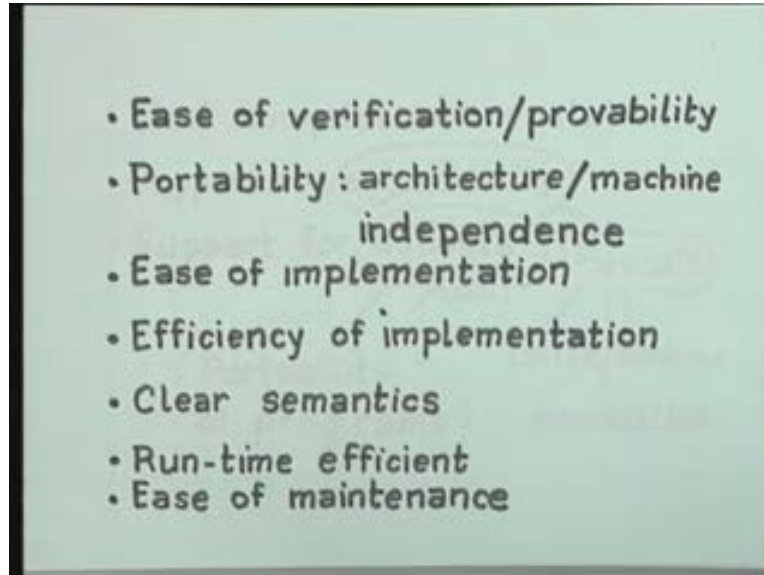
You are already aware of the basic abstractions. Control abstractions are procedures, functions in Pascal, loop statements etc. There are also data abstractions. The most primitive kinds of data abstraction that Pascal provides are the record structures and the arrays which almost all languages including the early languages have.

Arrays are one data abstraction. You think of a whole sequence of elements as a single logical unit like records and variant records. Over time it was felt that you should have data abstraction which allows you to not only take sets of data together as a single unit but also to operate on them as a single unit.

There are combinations of operations and data abstractions on a module basis or further abstractions for which you might require languages like ML and Ada that provide the notion of generality or polymorphism where you can use types themselves as variables and change types and instantiate the same kinds of algorithms.

For example; in stacks, it does not matter whether you are talking of a stack of integers, a stack of characters, a stack of reals or a stack of some complicated data element. It may be a stack of some record or stacks of arrays.

[Refer Slide Time: 14:25]

- 
- Ease of verification/provability
 - Portability : architecture/machine independence
 - Ease of implementation
 - Efficiency of implementation
 - Clear semantics
 - Run-time efficient
 - Ease of maintenance

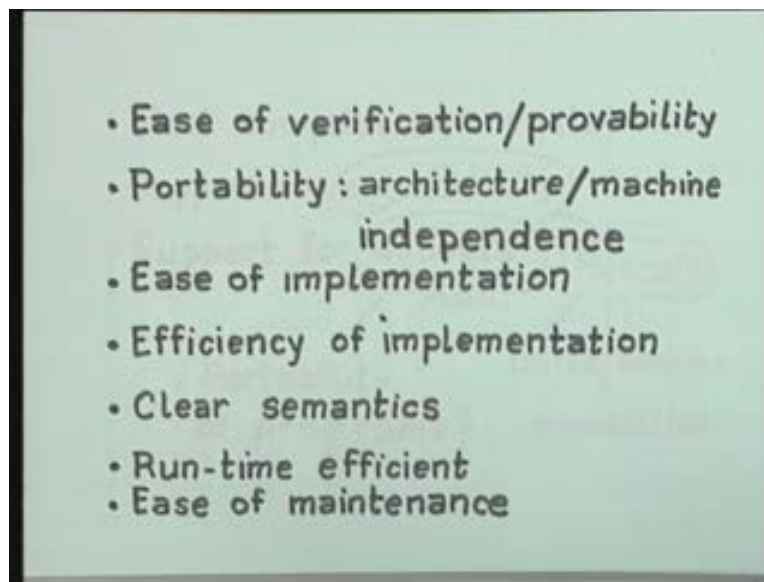
The basic operation on stacks is like pop push, checking for emptiness etc. They should all be available in one form and it should not be necessary for us to repeat the code depending on the type of the element. An abstraction should be one where the basic element of the stack itself is a variable which can be instantiated to a different type, each time the same piece of code is carefully written, verified, or thoroughly tested and the type instantiated. We get different kinds of stacks with the same kinds of operations which is why the support for abstraction is an important modern language design issue. The other kinds of issues that have come up are that you should be able to verify your programs. So, the language should also provide support for verification, provability of programs (not necessarily machine based provability but possibly hand based provability) or a mixture or a user interactive provability of programs. Portability was felt even in the 60s where a lot of time and effort was expended in the case of FORTRAN and COBOL compilers.

Nowadays portability means that the language design should be oriented towards an end user. It is not architecture or machine-specific simply because a certain machine has a certain kind of assembly instruction. It does not mean that you make that available in the language. Other machines may not have that. You should be able to provide as far as possible an abstract form which is not related to the machine that uses only the basic instruction set available in all machines. If you ensure that your language is really architecture or machine-independent in the sense that your main concern is one of convenience of the user or the abstractions required for the user, then they are not specific to particular machine instruction sets of a particular architecture. You might have register based architectures or stack based architectures. If you can design the language in such a fashion that it is not architecture-specific or machine-specific then you can move the entire language to another machine with a different architecture and with the minimum amount of effort.

There are certain machine specific details which still have to be changed when I move an entire language implementation from one machine or architecture to another. But the whole idea of the

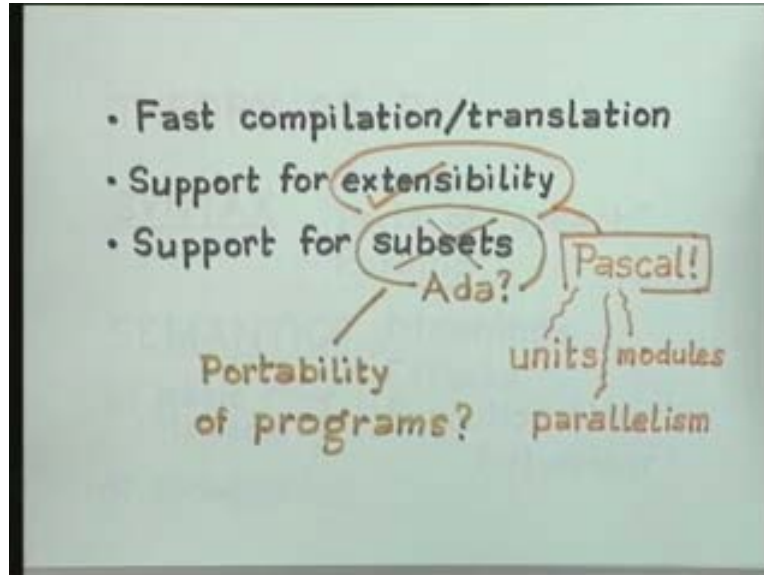
language design or the design of its implementation should be that the amount of changes to be made should be minimal. Then there are considerations that include the ability to easily implement it. You cannot compromise everything for the sake of portability. Ease of implementation and the availability of ready algorithms for implementing the language should be considerations. Ease of implementation was perhaps the most important reason for this and the fact that it used very low level primitives is perhaps the most important reason for the success of the C language. Efficiency is another important reason. C programs run very fast and they are very efficient.

[Refer Slide Time: 18:45]



If you have a language and you want it to be generally acceptable then it should have a clear definition of what its constructs do. The reason being that if it has to be widely acceptable there might be many different implementers trying to implement it on various machines and only if there is a common clear syntax, common clear semantics or clear specifications of the effects of each language construct, you can expect to get wide applicability. We will get to the notion of semantics later. Of course it has to be run time efficient. By run time efficient I mean that the programs have to run efficiently. This is the efficiency of the implementation which implies very often compile time efficiency. This means how fast you can compile programs written in the language. Whereas this runtime efficiency implies that it should have an excellent runtime support and the program should run as fast as the programs written in the language.

[Refer Slide Time: 19:53]

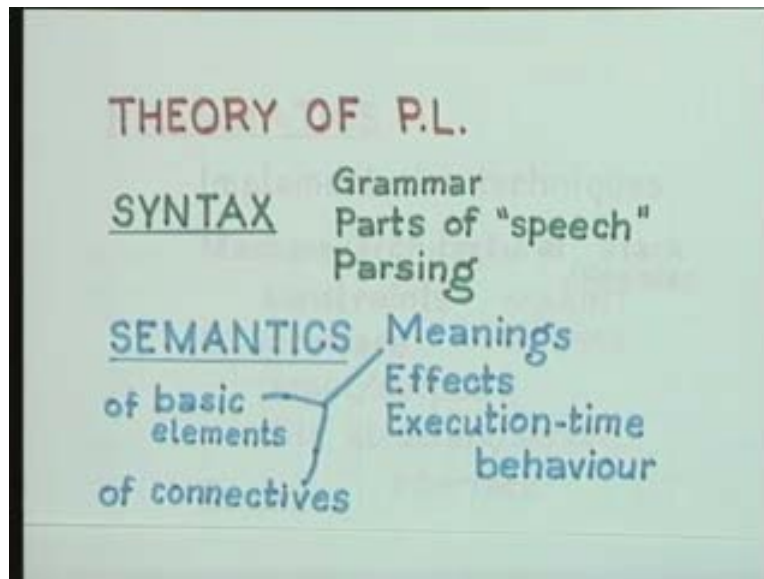


Ease of maintenance is about the maintenance of programs as well as the maintenance of the language by an implementation of the language. There might be bugs in the language implementation. Fast compilation translation and support for extensibility are in fact the most important reasons why Pascal is used as the basic language from which you add new features to get newer languages. This is one controversial feature, which is the support for subsets. Most books on programming languages specified this feature. It is that every language should support subsets of the language in the sense that it should be necessary for you to use only a smaller set of operations or features of the language than is actually necessary. It should be possible to divide up the language so that there is a small kernel and large sorts of extensions. All programs written for the small kernel run on all machines regardless of the subset supported. However, the language of Ada for example in the 80s has clearly specified that there should be no support for subsets and their reason is that it would then affect the portability of programs written in the language.

Support for subsets is a controversial aspect. It is not at all clear whether it is desirable especially in the most important language for embedded systems. They are real time systems that control sensors, various kinds of hardware, ballistic missiles etc. It could affect their portability because you would have a feature which is used in a specific implementation and then you move the program to another implementation which does not support that feature and your programs do not run.

The portability of actually programs written in the language gets affected if you allow arbitrary subsets. If you look at all these features, there are hundred and hundreds of programming languages and it is impossible to study all of them and moreover it is not necessary to do so. Therefore we would like to divide up the study of programming languages into a few small parts.

[Refer Slide Time: 22:50]



What does a theory of programming languages contain? A general theory of programming languages is based on three aspects. What might be the most important aspect or the most basic aspect is known as the syntax of the language. A programming language is really like a very highly simplified natural language; it has a certain syntax which implies that it has a grammar. Certain elements of a sentence can occur only in certain ways. You cannot arbitrarily form sentences of the language. As I said, a program written in that language is the sentence of the language like in a natural language. You take a sentence of a natural language and you find it has various parts. For example; in almost all natural languages any full sentence should have a predicate. A syntactic category called predicate in general may not be words; a predicate could be a clause or a phrase and so it might have a subject and an object.

Let us take a sentence in natural language. Supposing I say 'run'; that is a complete sentence because it has a predicate. No complete sentence in the natural language is without a predicate. So 'run' is a complete sentence because it has a predicate. Optionally, it might have a subject and may be also an object clause. Subject clauses have to be of a grammatical form called subject phrases or may be noun phrases which means they might have nouns qualified by adjectives or an article etc.

You can divide up any grammatically correct sentence in a natural language into firstly various clauses and each clause into various phrases and all these phrases have certain grammatical properties in a very much similar manner in a greatly simplified form that might be called the parts of speech of a program. Every programming language has a certain grammar, which specifies various parts of speech. It has a certain vocabulary and it is possible to take a sentence of this programming language and parse it. We will get into the meaning of these points more clearly later where artificial languages have similarities with natural languages; (a lot of the theory of the syntax was actually inspired by natural languages) where the construction of artificial languages did not have a lot of problems in the natural language.

The next aspect is 'semantics', very often called the semantics of a programming language, which is really specified by its reference manual.

In your study of Pascal programming one of the important references would be the ISO standard Pascal reference manual by Janson Edward. That reference manual really specifies for each syntactic entity of the language the effect to be expected by executing that syntactic entity which is the meaning associated for each language construct. So, we have to look at the notion of meanings. Unlike natural language the notion of meaning can be specified mathematically in a machine-independent fashion.

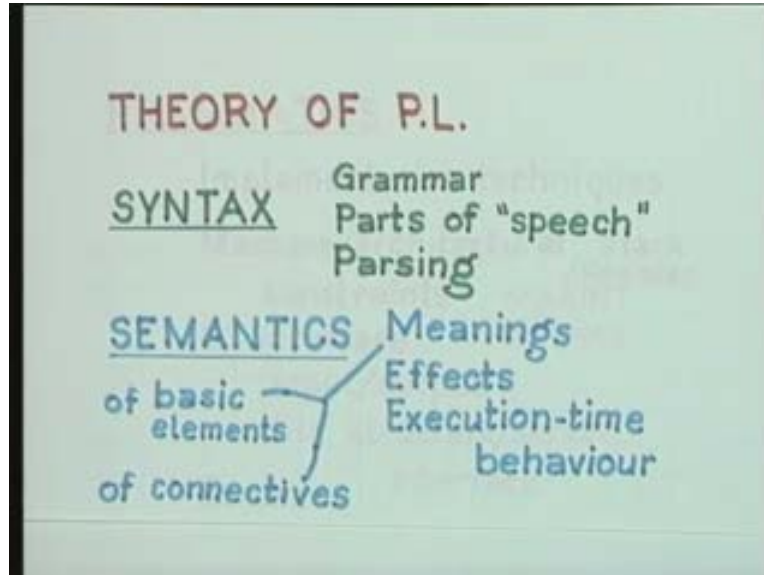
When we are talking about the semantics of this programming language, we are talking about a pure meaning associated with the language regardless of any machine on which the language is implemented. In general we are talking about meanings in an abstract setting in the sense that you assume for practical purposes that you have no restrictions on the word lengths, no restrictions on memory and you have no restrictions on computational power except that you can only do a finite number of operations at any instance.

The programming language itself can be thought of as a mathematical entity quite independent from all its implementations and in general when you think of it that way then you are thinking of some kind of an ideal machine which has no restrictions except one that at any instant of time only a finite number of operations can be performed.

You cannot do an infinite number of operations at any instant of time. If you consider such an ideal machine then you are really looking at the programming language as a mathematical entity in some ideal environment and the meanings of the constructs of the language in that ideal environment, form, what might be called the actual reference manual.

If you were to go through the reference manual of Pascal, most of the reference manual is independent of any machine. There are certain specific machine paragraphs or what are known as implementation dependent features. But if you look at the language itself, it is capable of looking at the language as an entity devoid of any machine.

[Refer Slide Time: 30:13]



In general the semantics would follow the syntax. The syntax has a certain structure. The syntax has got some basic elements and some compound forming operations. They are connectives which allow you to form compound sentences from simpler ones.

The meanings in general should be such that you give the meanings of the basic elements and then you give the meanings of the connectives in terms of the basic elements.

A programming language is really a finitary object which allows you the construction of the infinite set of programs.

It is not in general possible to predict the behavior of the language of a program written in the language unless you have this basic discipline where you express the effects of connectives in terms of the effects of the basic elements of the language.

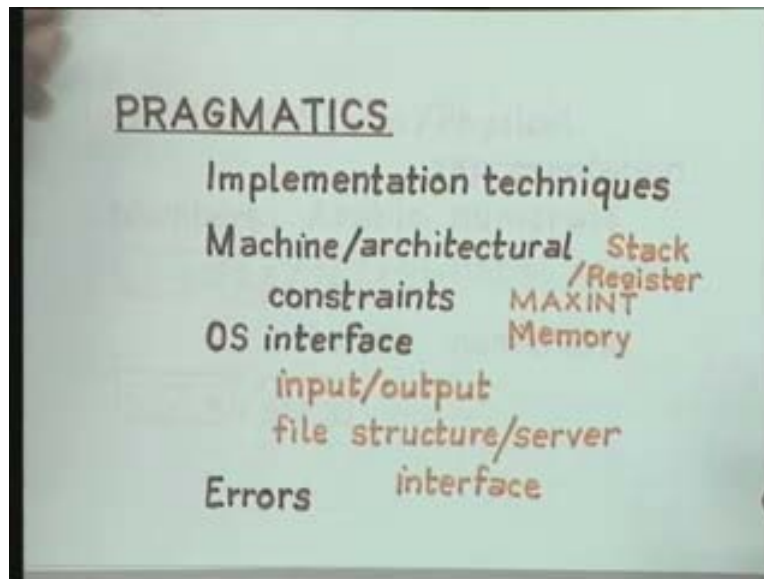
It is not in general possible to predict the behavior of the language of a program written in the language unless you follow this discipline.

The basic feature of any kind of semantics is that it should allow for the derivation of the meanings of an infinite number of programs which means the only way to derive the meaning of the program should be from the meanings of its finitary elements. You have certain basic elements from which complex elements are formed. So, the meaning of the complex element should be derivable in terms of the meanings of the basic elements and the meanings associated with the connectives that formed a complex element from the basic elements. It means that semantics is going to be intimately related to syntax.

The syntax is really a finitary way of specifying an infinite set of allowable objects. The infinite set of allowable objects are the programs of the language or the sentences of the language and the syntax gives you a finitary mechanism for specifying all possible allowable programs very much like a set theory notation, which allows you to give a finitary specification for an infinite set. What is really analyzable for an arbitrary program is its structure in terms of its syntax.

The meaning of the program has to be derived from its finitary specification to be expressed in terms of its finitary specification. Lastly, do not worry about machine constraints or about architecture because the programming language has to be portable. Do not worry about word lengths or about limits and also do not worry about memory constraints.

[Refer Slide Time: 33:58]



Assume infinite amount of memory available so that you get into what are known as pragmatic considerations. Most of these implementation dependant features are really pragmatics, for example; the Pascal reference manual does not tell you how to associate a disc file with a file variable inside the program. That is one example of an implementation dependant feature which will vary depending upon the operating system interface you have. Then there are various simple implementations.

The pragmatics firstly involves really all the algorithms that are going to be used for the implementation of the language and all kinds of machine and architectural constraints for example the MAXINT. The maximum integer allowable in a Pascal program is a typically implementation dependant feature because it really depends upon word length or byte length or on whether it uses two bytes for representing integers etc. The value of the MAXINT can vary from machine to machine. The amount of memory that is available for a program can vary whether it is a stack based machine or a register based machine. All those are implementation dependent features.

Normally, the way to make the language portable even in our implementation is to separate the basic algorithms of implementation from the architectural specific nature of the implementation. When we read through a small compiler you will see this happening. So, there are certain basic algorithms which are really machine-independent and then there are some machine dependant features and the actual code.

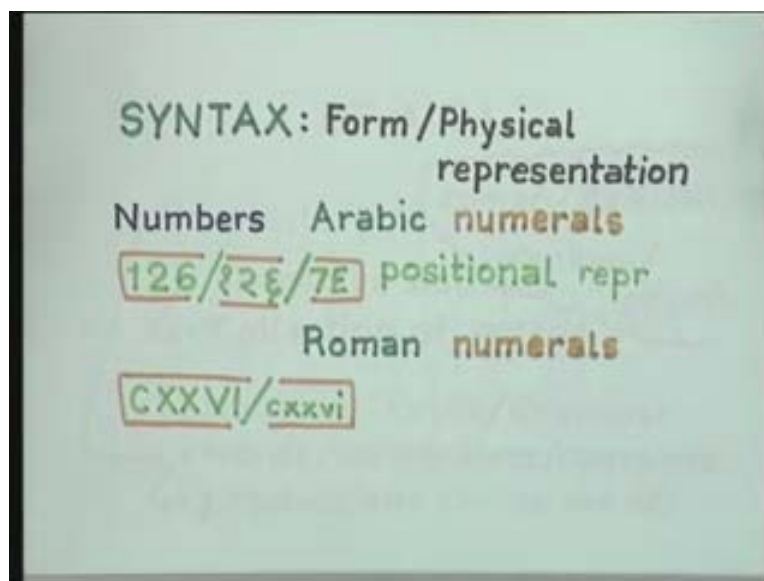
Then you have the OS interface, the nature of the input and output that is file-based, terminal based, sensor based, signal based, etc. The OS interface also includes the file server, how the language has to interact with the file server and how the language has to interact in general with the directory service of the machine etc.

Lastly, what is to be done about errors introduced by users in their programs? Errors could be of a syntactic nature or of runtime nature and we need to know what to do about them. There is a blanket policy where you could just abort but you know that does not help anybody in particular. What is the nature of error reporting?

Is there some error correction that can be done? Is there some way of recovering from errors so that as soon as the first error comes, you just throw out the program? - No, if you can you should at least be able to point out all the errors in the program or all possible errors so that the user gets to know all the errors at one time. It reduces the amount of compilation effort and similarly you cannot do that at run time but at least there should be some decent error reporting mechanism.

But error is a very dicey subject and dicey policy. Different languages have taken different attitudes and different implementations take different attitudes towards it. So, it is a very pragmatic feature. I will just briefly go through the notion of syntax. There are three issues and it is better to study all three issues separately. The semantic and pragmatic issues will closely depend upon the syntax and it is preferable that they depend upon the syntax. Let us look at what constitutes syntax.

[Refer Slide Time: 39:20]



Syntax has to do with a form or a physical representation of possibly an abstract object. If you look at numbers, they really are not very physical. At least the 20th century attitude towards numbers is that they are a conception of the mind and are not physical. They do have a physical representation in the form of numerals. What you write out and think of as a number is actually a

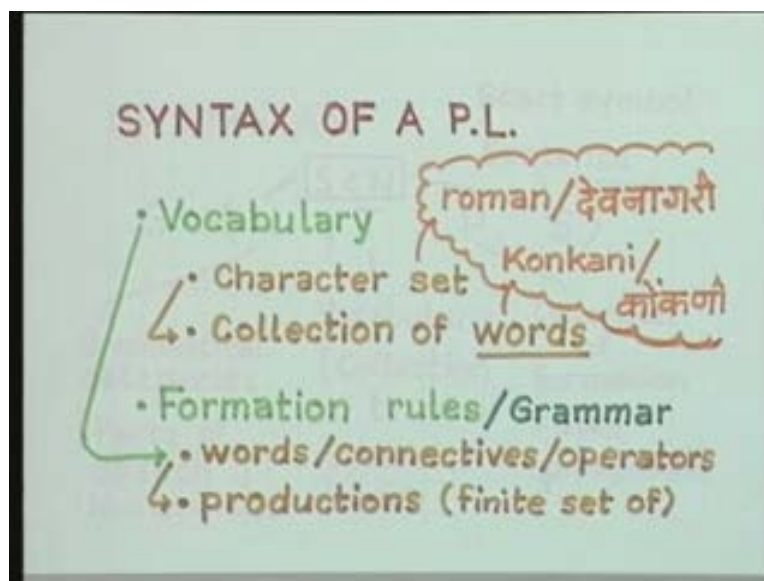
numeral so while you have various ways of representing the same number let us say the number 126 you can have what is known as the positional representation which is what we normally use.

In the basic form all these three representations of the number 126 are really the same. The roman representation differs from the DEVNAGRI representation in the sense that the characters used are different. What might be called the basic alphabet for the positional representation of numbers is different in the two.

It is just that the character set is different but that is incidental. It is a positional representation but in basic forms all these are unified by the fact that they are all positional representations. You go in units: 16s...64s... Then you have the roman numerals which is non positional. Firstly, it has a different alphabet. You could represent the same number 126 with different character sets but there is something unifying about this. There is some fundamental difference about this representation.

If you disregard the change in character set what makes these two classes different is the grammar of the numeral of the language used for representing them. In both these cases the grammar is exactly identical, the character sets are different and by and large the grammar of all these three is the same except for the character sets but the grammar is different in the sense the form of representation is fundamentally different. How you represent compound forms from simpler forms is different between the roman and Arabic case. What that grammar is, is what we have to study. Let us look at this in a more general setting of the programming language.

[Refer Slide Time: 43:08]



We might think of every programming language as having a vocabulary. A vocabulary is a complete dictionary of words of the language. A word of a language is formed from a fixed character set and you identify certain strings of characters as allowable words as part of the vocabulary of the language. A complete dictionary of the language is what states that vocabulary.

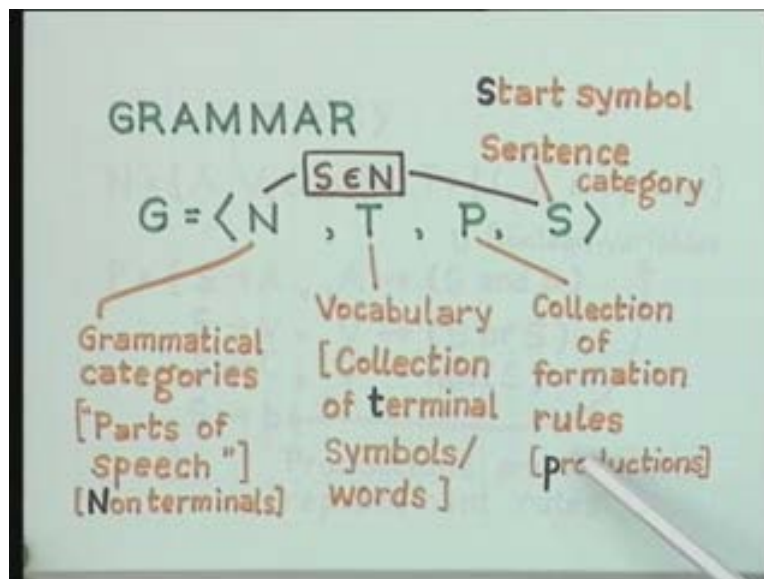
If you take languages, for example natural languages like Konkani or Sindhi they have different character sets but the same collection of words.

Sindhi for example is written by different people. Some people write in Devanagari, some people write it in the Urdu script or in the Arabic script. But the collection of the words is the same, so, a person who knows Devanagari can communicate with the person who does not know Devanagari but knows the Urdu script by speech because the words are the same although you cannot communicate by letters.

There is a certain fixed collection of words whose actual form might depend on the character set. But given the words there are formation rules. Given a vocabulary there are ways of combining words of the language to form sentences of the language and there is a finite set of formation rules called productions which allow you to generate all possible sentences in the language.

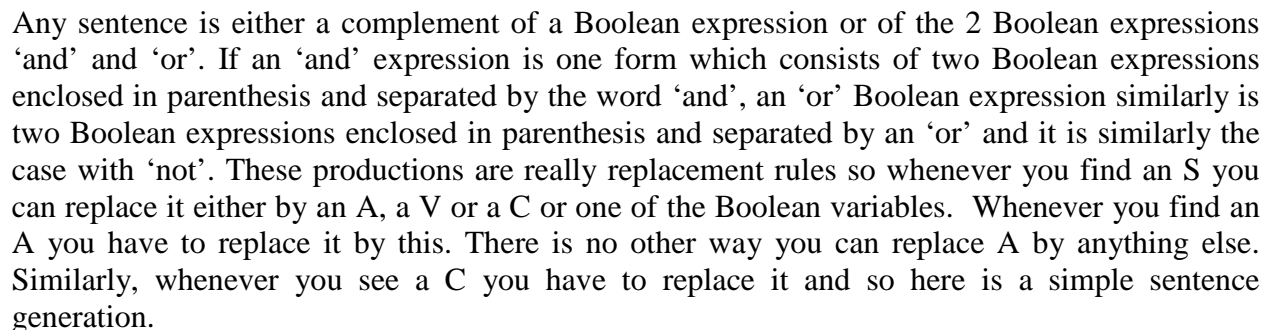
Let us quickly go through one example. The character set really depends upon the kind of codes you use. Nowadays most of the time we use ASCII codes but then we have 8 bit ASCII's and PC's and 7 bit ASCII's on the main frame. The character sets are different. There are all these kinds of differences but let us disregard them and look at what constitutes grammar.

[Refer Slide Time: 46:00]

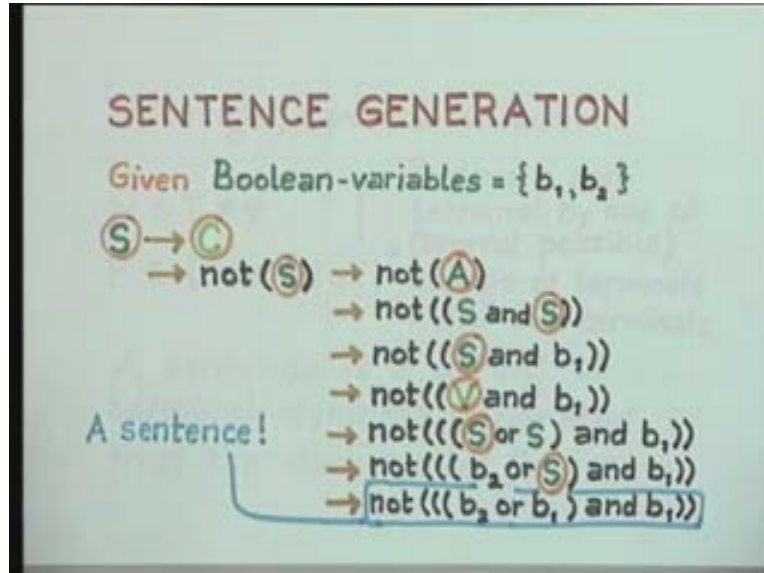


I would say grammar is really a four tuple of objects where there is a set N which is called the set of non terminals and this set of non terminals really specifies various kinds of grammatical categories of the language like the parts of speech, noun phrase, verb phrase, adjectival phrase, noun clause subject clauses, subject phrases, object clauses, predicates etc. This set N consists of the basic grammatical categories and all these sets are finite. Then the terminal T is the set of what are known as terminal symbols or terminal words and T is the complete vocabulary of the language. P is the collection of formation rules or what are known as productions and S is what is called as the start symbol but S really represents a grammatical category in N called as sentence.

[Refer Slide Time: 47:56]



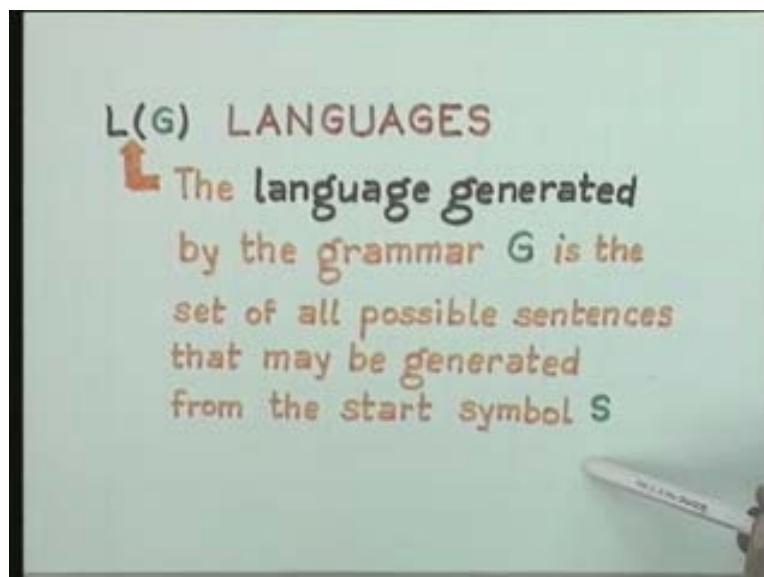
[Refer Slide Time: 50:17]



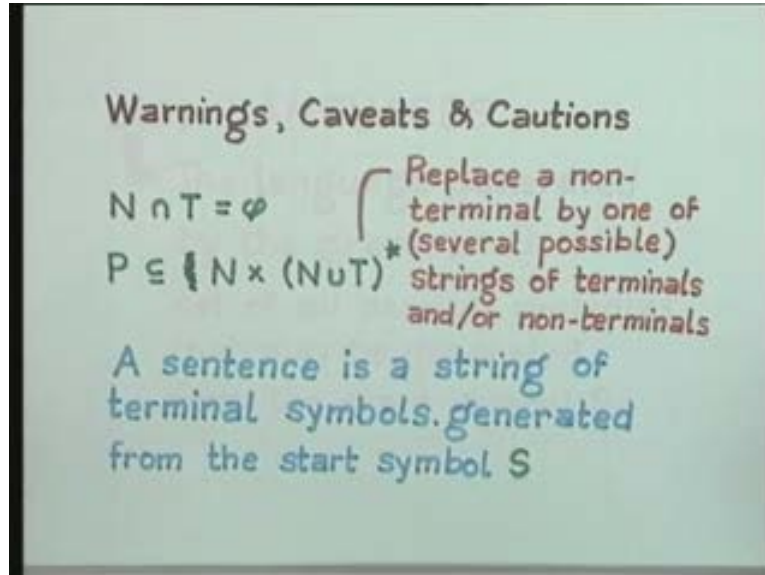
You start from S and one of the possibilities is that you can replace S by C. Whatever I am replacing, I have circled in orange. C has to be replaced and now I am replacing S leaving everything else intact. I have chosen here to replace S by A and once I have got an A there, the only possibility is to replace it by this form S. I have chosen to replace this S rather than this S first and I have chosen to replace it by a Boolean variable.

Let us assume that there are only two variables b_1 and b_2 and I proceed. Lastly, I get a complete sentence of the language which consists of only the terminal symbols which is generated by the grammar.

[Refer Slide Time: 51:30]



[Refer Slide Time: 51:40]



We talk of a language as being generated from a grammar as a set of all possible sentences that may be generated from the start symbol S . Important warnings are that the set of non terminals and the set of terminal symbols are disjoint and a production is really a replacement. It replaces a non terminal by a string consisting of terminals or non terminals and a sentence is just a string of terminal symbols generated from the start symbol.