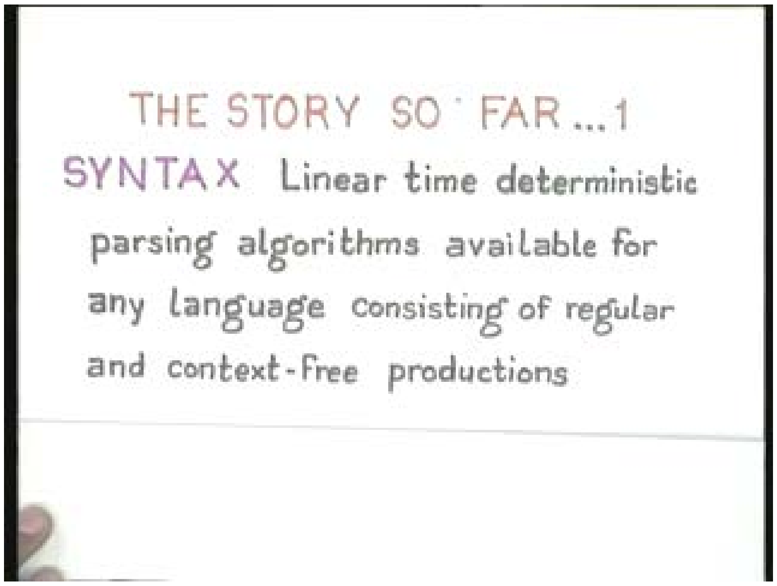
Principles of Programming Languages

Dr. S. Arun Kumar
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi

Lecture - 40
The Future

Welcome to lecture 40 and let us look at the future or rather let us look at all that you have not learnt so far. So we have broadly divided the field of programming languages into three categories; syntax, semantics and pragmatics. So let us look at each of them. The story of syntax which actually started with Chompsky finally ends with having found the murderer. So you can get linear time deterministic parsing algorithms for any language which consists of regular or context free and or context free productions. And boiling down to basics what that means is that as long as you do not have anything more complex than paranthesis matching you are safe with and you can get linear time deterministic algorithms, that's really what it all boils down to. Despite all the complex normal forms that might have been invented and so far it does not really matter. You have looked at parsing algorithms for example the PL0 compiler and so on and so forth.

(Refer Slide Time 1:06)

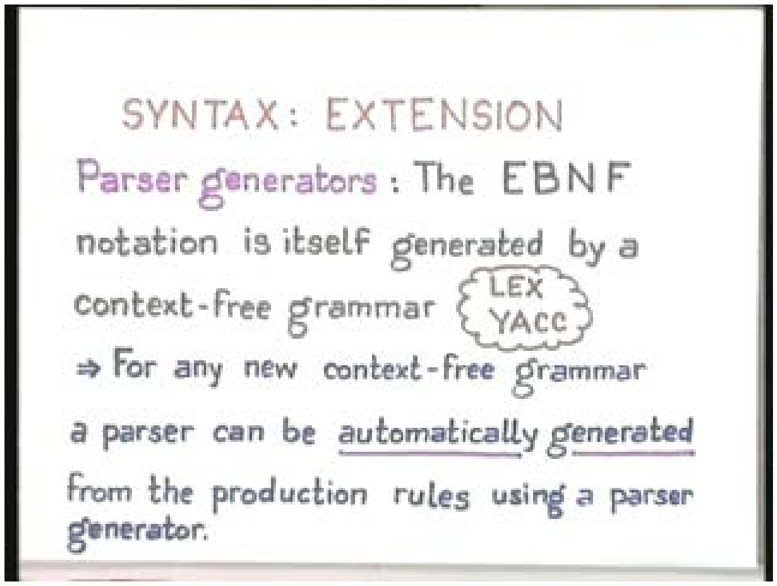


THE STORY SO FAR ...1

SYNTAX Linear time deterministic parsing algorithms available for any language consisting of regular and context-free productions

The next extension which is of course already available in most unix systems is that of a parser generator and that just uses the fact, firstly given any notation my first aim would be to see if can just convert it into grammatical rules and the extended BNF notation is one that is directly generated by a context free grammar. So the LEX and YACC programs which are available on any unix system are essentially are parsers for the extended BNF notation. So you take any grammar and given its production rules using braces and square brackets and so on and so forth it is possible to generate a parser automatically from the grammar rules.

(Refer Slide Time: 2:50)



SYNTAX: EXTENSION

Parser generators: The EBNF notation is itself generated by a context-free grammar {LEX YACC}

⇒ For any new context-free grammar a parser can be automatically generated from the production rules using a parser generator.

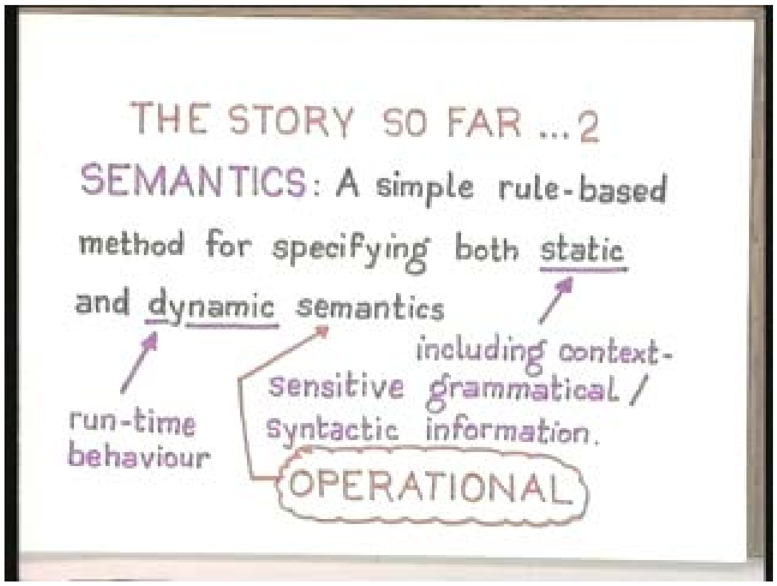
Of course the extended BNF notation is just one possibility, the other possibility is to use Pascal like syntax diagrams. There is a one to one correspondence between the extended BNF notation and Pascal like syntax diagrams. Therefore what it means to construct a parser generator is to write a parser for the extended BNF notation itself regarded as a language which generates syntax diagrams as graphs and that is automatically possible. So that's mainly what YACC does.

This (Refer Slide Time: 3:41) specification of LEX for lexical analysis of token generation if you like can also be done automatically and that requires no more power than that of a context free grammar. So you can use the same notation for both token generation and for parser generation. Therefore what it means is that you just give the production rules as inputs in an extended BNF notation and the parser generator like the LEX will produce tokens for the individual syntactic elements and YACC will take that and produce a syntax tree generator. Then you just have to introduce code generation and so on and so forth. There are methods of doing automatic code generation which are not very perfect but these two are fine and are actually used.

So the story of syntax is more or less finished. It is well understood that for context sensitive grammars or grammars which are more powerful in a certain sense which are more powerful not in the sense that they can generate a large number of sentences but in the sense that they can generate restricted class of sentences, after all grammars are a means of control which allows you fine levels of restriction. So, for anything like a context sensitive grammar or a type zero grammar it is more or less understood within the community that probably you wont get such good algorithms as you have for context free grammars. There was an attempt in Algol-68 to what are known as fan one Gordon grammars but it did not proceed very far.

So the next thing is semantics. What we have looked at in terms of semantics is rather than give algorithms after all for every algorithm that you give I can give a thousand variations of the same algorithm, rather than give algorithms give a simple collection of rules which are somehow syntactically motivated which provide the minimal framework on which an algorithm should be based. So you can use this for both static and dynamic semantics. And if your static semantics is structurally inductive then what it also means is that all context sensitive grammatical and syntactic information of which types is one can also be specified by a static semantics.

(Refer Slide Time: 6:27)



THE STORY SO FAR ...2

SEMANTICS: A simple rule-based method for specifying both static and dynamic semantics

run-time behaviour

including context-sensitive grammatical / syntactic information.

OPERATIONAL

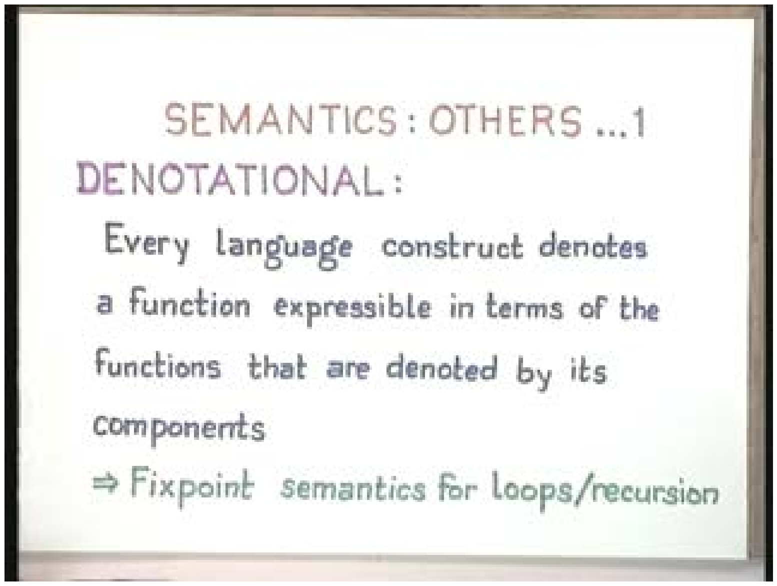
There have been methods for specifying semantics within the domain of a context free grammar itself and an important contribution in that respect is due to Don Knuth called attribute grammars where he took context free grammars as a framework and with each production you associate a semantical rule very much like something we do but he encoded it in the form of code generation rules to generate codes and idea was that now that you have parser generators you should automatically also do code generation by using those attributes. So a lot of what Knuth's work on attribute grammars is actually used in his software for text formatting called 'tech' which is really like a massive compiler, it generates code in a device independent fashion and he has used a whole lot including his own parsing algorithms, most of the table driven parsers the best known parsing algorithm with one step look ahead is due to Knuth for context free grammars, it is a bottom up parser he has used all these.

Text formatting programs does not really have anything to do with an excellent application of programming languages, compiler concepts etc. So 'tech' is one example, 'scribe' is another, 'eqn' on the unix systems is the third they are all methods of coding notation into context free grammars parsing them and then generating code which will give you the formatted output. So there is a higher level form of using these languages, grammars, semantical rules, attributes which you can use for applications just outside the domain of the programming languages. So these are very general methods, transition systems are very general; you can use it to describe anything. The notion of grammars and syntax directed translation or syntax directed semantics is also a very general notion which is going to be important whenever you are trying to automate any piece of software.

So whether it be automating symbolic computations in mathematics, automating proofs or doing just plane text formatting or doing hypertext translations or trying to map graphic images onto something then one good mechanism which people are followed with fruitful results is to somehow transform the whole problem into a grammatical problem and then into a semantical problem and use the principles of compilers construction to actually solve the problem in some satisfactory fashion.

Of course what in order to find unit you might have to introduce heuristics and so on what it means is that whatever we have done is not very restrictive, it is something that has a wider applicability and has been used by several people actually to do for example the image processing, text formatting etc. in fact the design of all user interfaces for all kinds of software means first encoding the interface into a language writing a translator or an interpreter for the language and executing it.

(Refer Slide Time: 12:01)


SEMANTICS : OTHERS ...1

DENOTATIONAL :

Every language construct denotes a function expressible in terms of the functions that are denoted by its components

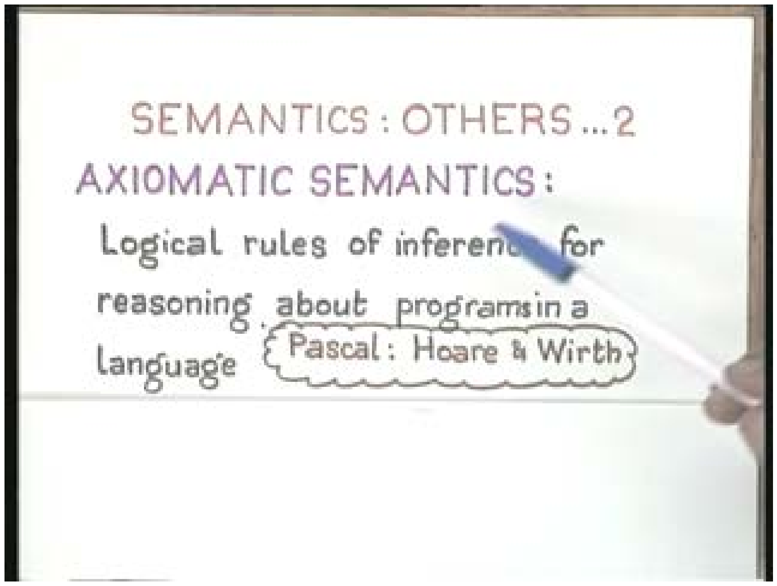
⇒ Fixpoint semantics for loops/recursion

Therefore in terms of application it goes quite far and the restricting ourselves to semantics itself what we have specified is what might be called operational semantics. Essentially the fact that we didn’t have to describe algorithms we just gave the minimal amount of information in terms of rules and then you can construct your algorithms based on that, it makes it an operational semantics because it gives execution time behavior in a step by step fashion. We had one step transitions we had many step transitions so it is really operational. What makes it operational is that you are actually considering a step by step transformation of some notion of a configuration. so you are looking upon the program as a transducer, you are looking upon each construct of the program as a little transducer and a program itself as a complex transducer made up of little transducers which provides transformations on the input so it is operational in that sense.

The other story is that you can require semantics as being denotation. That means you can look upon every program itself as a function from some domain to another purely as a mathematical function which means you are not looking at its step by step transitions but you are looking at just one feature what is the input to the output relationship of this program bypassing all the intermediate information that might be available. And here again we would like to do it in a syntax directed fashion so what you want to look upon is each language construct is denoting a function and a program which consists of language constructs is somehow connected together.

We would like to look upon them as functions which some how are connected together to give you one large function. So we would like to express the meaning of a program as a function in terms of the functional meanings of its components again in a structurally inductive fashion. in particular what this means is that we have to be able to account for the semantics of loops and recursion in a perfectly syntax directed fashion purely as functions, essentially as functions which compute a fixed point, functions which yield a fixed point. This is a functional semantics or mathematical semantics that is denotational.

(Refer Slide Time: 14:50)



SEMANTICS : OTHERS ...2

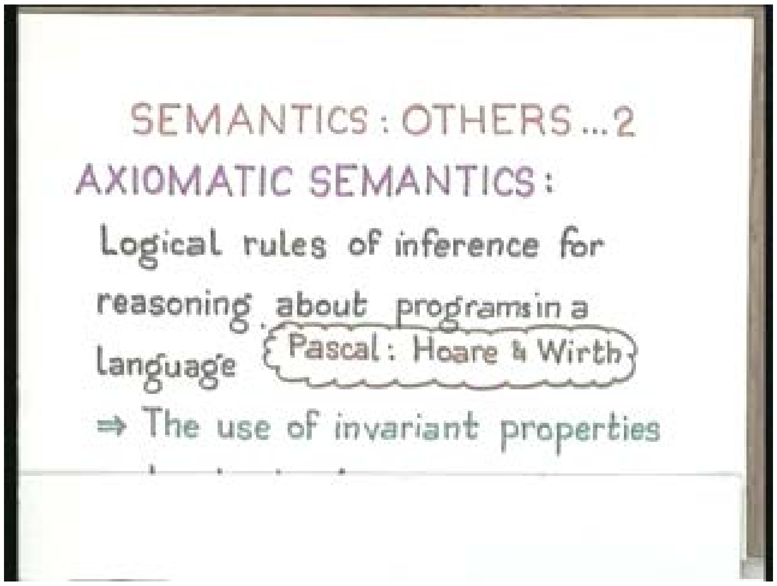
AXIOMATIC SEMANTICS :

Logical rules of inference for reasoning about programs in a language {Pascal : Hoare & Wirth}

Denotational comes from the fact that you are talking of a language construct as being a syntactic object which actually denotes some abstract object just like a numeral denotes a number in the same way you want a language construct as just a syntactic representation of an abstract function in our mind. The other thing which is of some importance is what is known as axiomatic semantics and there are several flavors of axiomatic semantics. But principally what you are looking at in axiomatic semantics is that you want logical rules of inference for reasoning about programs in a language. So here again you want a syntax directed logical rules of inference.

So when you are talking about reasoning about programs then you are talking about reasoning about the behavior of programs and you require a language in which to express your reasoning about the program. One possibility of course is first order logic and in fact a large part of Pascal was actually axiomatized by Hoare and Wirth in 1975 or so and their logical rules also influenced back the design of the language in order to make it clean. In fact the problematic constructs in the language are those that they did not axiomatize like variant records, types. It is clear that at that time they didn't have a much of a clue as to how to take care of those and those indeed are the problematic constructs. The other possibility is to use what is known as equational logic.

(Refer Slide Time: 16:10)



SEMANTICS : OTHERS ...2

AXIOMATIC SEMANTICS :

Logical rules of inference for reasoning about programs in a language {Pascal : Hoare & Wirth}

⇒ The use of invariant properties

The fundamental tool here is the use of invariant properties to develop, prove, verify, correctness against a specification where this specification is also in the logical language which you are going to use to reason about programs. So whether it is recursion or loops or anything what you want to develop is our rules of inference for reasoning about the correctness.

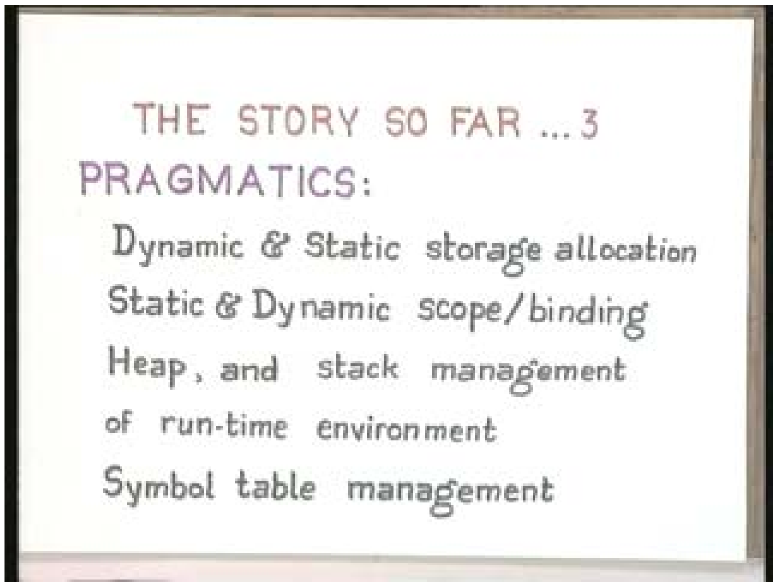
At this point we are not interested necessarily in specific functions, we are interested may be in broad properties that the program should satisfy. So express the broad properties as predicates in some language it turns out that first order logic is not a sufficiently powerful mechanism for example you will have to have first order logic augmented with mathematical induction in order to do reasoning. But the moment you introduce induction then you automatically get into the domain of a higher order logic because mathematical induction is not a first order logic specified predicate so there are problems about expressivity of the properties that you are interested in.

Many of the properties that you are interested in may not be first order and they might require higher order predicates. There is an extra complexity by introducing another language even though it is a logical language. The other possibility is to actually do an axiomatic semantics within a single language framework. Have a specification language which is a superset of your implementation language. Have the notion of semantic equivalence as defined from an operational or a denotational view point and do the reasoning as equations within the same language that is another possibility that is being explored. But the use of invariant properties for reasoning essentially about imperative programs that means reasoning about control which can change state is perhaps the most important reason why you have used an axiomatic semantics method.

And of course the moment you have two or three different kinds of semantics there is another problem of mismatch of the individual semantics. then you have the extra added obligation that you have to prove that the three semantics are mutually consistent and you have the extra constraint that the presence of all kinds of strange properties that you might have in your operational semantics the other semantics actually give you all the properties you are looking for otherwise you may be never able to prove a program is correct. There might be certain properties which are so intrinsically operational.

It is not just consistency that you require between various semantical formalisms you also require a completeness that every property that is expressible in operational semantics can be captured in the denotational framework or in the operational framework if I am to be able to prove all the property is correct, if I have to prove about all properties of the program that I am interested in then they have to be somehow expressive they have to be mutually expressive and that is what is known as a full abstraction problem. So you have to prove not only that the semantics are consistent but there is not so much information hiding in the operational semantics that you are not even able to prove certain properties in your axiomatic or your denotational framework.

(Refer Slide Time: 21:13)



THE STORY SO FAR ... 3

PRAGMATICS:

Dynamic & Static storage allocation

Static & Dynamic scope/binding

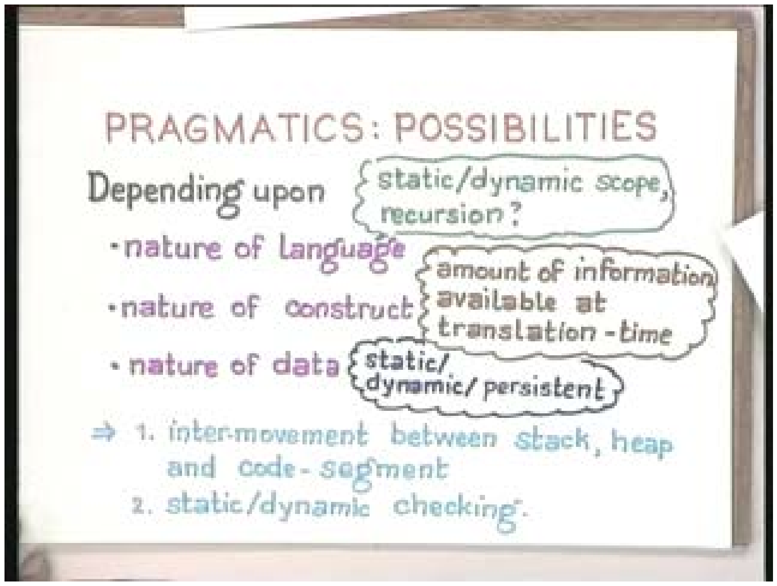
Heap, and stack management

of run-time environment

Symbol table management

So there is a vast body of knowledge on semantics and then when you look at pragmatics what we have essentially seen is where various dynamic and static storage allocation mechanisms are you have dynamic and static scope and binding mechanisms which we know how to deal with, you have essentially heap and stack management and management of the run time environment which essentially consists of the heaps, the stack and the code segment may be also data attached to the code segment then we also know about symbol table management at translation time and it so happens that this is in fact all you require as a basis for implementation, it is a matter of deciding now. at least for the language construct that we have not studied so far and for the language constructs that have evolved over the last twenty years it seems largely a matter of decision making for a given data item given these properties of the language whether you should store the data item in the stack in the heap or with the code segment.

(Refer Slide Time: 22:00)



PRAGMATICS: POSSIBILITIES

Depending upon {static/dynamic scope, recursion?

- nature of language
- nature of construct {amount of information available at translation-time
- nature of data {static/dynamic/persistent

⇒ 1. inter-movement between stack, heap and code-segment
2. static/dynamic checking.

Given the nature of any construct how much information is available at compile time. Therefore what other information is lacking which should therefore be checked at run time, what things can be checked at compile time and therefore they do not need to be checked at run time. These are the basic implementation issues which we have looked at and which actually govern whatever new language constructs that may probably come up. So essentially you have to look at the nature of the language whether it is a static language or a dynamic language look at whether there is recursion in it in some form or in more than one form for example the while loop is a form of recursion but implementationally it does not matter the while loop can be regarded as being different from recursion because it does not mean creating new activation records. But semantically the while loop can be regarded as being another form of recursion, in fact it is a form of tail recursion.

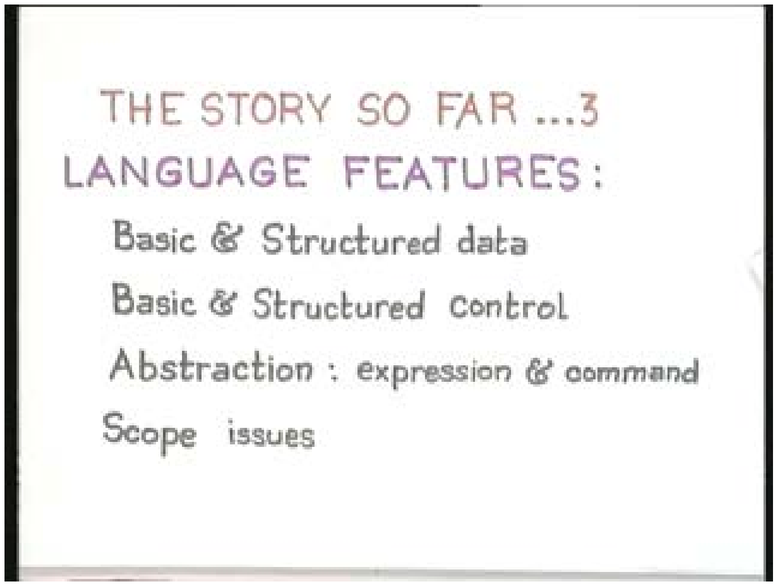
Thus, by recursion in a pragmatic sense we actually mean recursion syntactically determinable recursion. And essentially from a given construct and from the given language what is the kind of information that you can obtain at compile time or translation time. Based on that you can also decide is it most suitable to have an interpreter for that language than a compiler. But the point is these days for any language you will have to have both an interpreter and a compiler.

Essentially when you go into debugging mode of a compiled language like Pascal you are essentially interpreting the language. But you are interpreting the language after all the information that can be extracted at translation time has actually been extracted. If you take a language like ML or Lisp which is usually interpreted eventually if you are going to productionize it you cannot afford to run it in an interpreter mode. If there is some large piece of software which has to be run repeatedly it cannot be run interpreted because what it means is this manual intervention and there is manual intervention where it is not necessary. Therefore, what you would like to do is compile the program after having developed it so you use the interpreter mode for developing the program correctly and testing it out and after that you compile it into an executable or an object code and run that object code.

So essentially a part of our programming environment for any language is that there is that fine mix at development time you want an interpretive mode to be readily available and at production time you just want an object code you want a compiled version of the program to be executed. So based on the nature of the language what it therefore means is that what parts of the language can be readily interpreted.

Therefore essentially what parts of the language give you a confirmation at compile time what do they withhold from you at compile time and therefore what has to be obtained at run time. The symbol table for example should be present at run time, it is an important question a Pascal symbol table is never present at run time and its an important question, the Pascal symbol table is never present at run time but in a dynamic language like this you would probably have to maintain the symbol table at run time. So if it is going to be a language which does not allow for a static type checking mechanism but requires only dynamic type checking then you will have to maintain all that information at run time.

(Refer Slide Time: 29:26)



THE STORY SO FAR ...3

LANGUAGE FEATURES:

Basic & Structured data

Basic & Structured control

Abstraction : expression & command

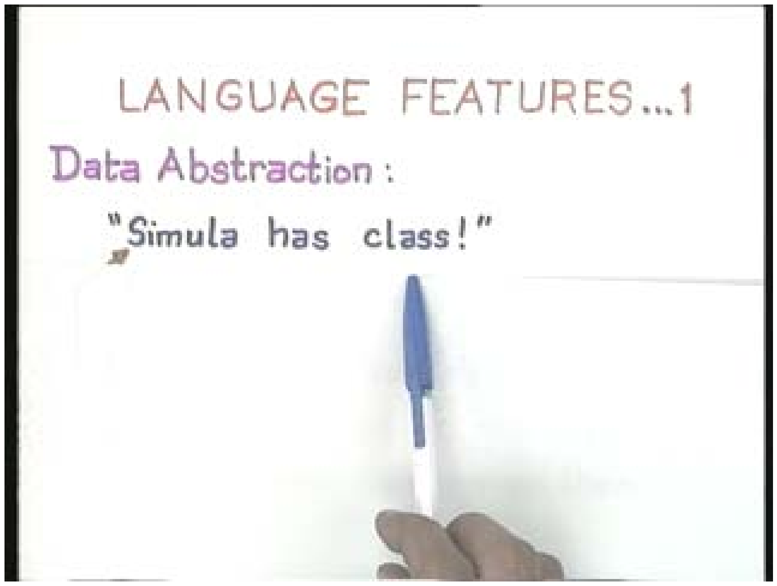
Scope issues

Then as far as the nature of data is concerned essentially the basic design decisions are going to be is it statically determinable data? Can I determine types, sizes, bounds at compile time? Is it dynamically created data or is it data that is persistent.

Depending on these classifications I essentially decide whether to store it on the stack, the heap or with the code segment. And in fact the pragmatic possibilities are not so high except when you move from our essential Von Neumann architecture to a different architecture. The other possible architectures that you might want to move into or that of parallel architectures where you have a little Von Neumann machine with its own local memory and connected through networks of connections or a completely non Von Neumann architecture may be a data flow architecture in which case you actually create new pieces of automation dynamically may be. But those are the other possibilities. But essentially within the framework of a single or multiple CPU sharing some memory essentially these are the only things that you can do. It is a matter of deciding between these possibilities.

Lastly we have to look at language features and that's where most of the development has been in the last fifteen years. So if you look at language we have looked at basic language constructs, basic data and data structures, basic notions of control in imperative and functional languages and we have looked at essential abstractions in expression and command language and we have looked at scope issues. In fact scope is a sort of overriding under current throughout the discussion ever since definitions and declarations came in. Ever since the issue of naming comes scope becomes an important issue.

(Refer Slide Time: 31:06)



LANGUAGE FEATURES…1

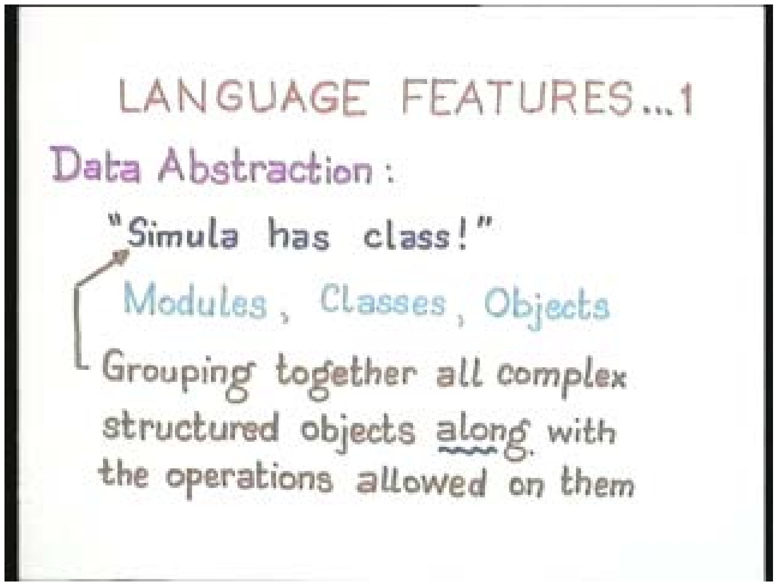
Data Abstraction :

"Simula has class!"

If you look at scope it is actually fairly clued in the sense that it either provides you direct visibility and complete freedom to deal with a name or it completely hides the name and allows you no access to the name. And a name of course represents some object either a data object or a control object. So when you look at it from several view points just like you have control abstraction the other possibility is to have data abstraction which means you group together declarations regarded as a single unit as an abstract of definitions and there are good reasons to deal with this. Actually this is a contribution which originates the language Simula 67 which is a descendent of Algol-60 but Simula distinguishes itself for two important features. One is that it is the originator of the class concept and a standard byline in any [di….31:10] implementation which had Simula 67 was a poster which said Simula had class; this was there in the 70s.

The whole idea is that you group together structured data and all the operations that are defined on that structured data in one single logical unit and that was the Simula class except that it did not provide too much difference in visibility, they use a standard scope rules. But now when you encapsulate it with a name you get the module facility of Modula and you get the classes of C++ or Smalltalk which actually elaborated on the class concept of Simula and provided the necessary abstraction in pragmatically on what all that they did in Smalltalk was that they provided the necessary abstraction by allowing you not a direct access but an indirect access through pointers with permission encoded in the access through pointers.

(Refer Slide Time: 32:20)



LANGUAGE FEATURES...1

Data Abstraction :

"Simula has class!"

Modules, Classes, Objects

Grouping together all complex structured objects along with the operations allowed on them

So pragmatically what it meant was that the whole philosophy was important in the sense that when we talk about an integer we are not talking only about the piece of data which is an integer we are also talking about all the allowable operations on integers. For example, you cannot exhort two integers, what I mean is you cannot logically exhort two integers and so along with integers comes the operations that are associated with integers addition, subtraction, multiplication, division excluding division by 0 and so on and so forth. There is absolutely no reason why we can't lift the basic notion of a data type from basic scalar data types to higher data types to structured data to data structures and when you bring in the abstraction what it means is that you regard a data structure primarily as an instance of an abstract data type.

So an abstract data type is just some structured data with the operations associated with that structured data grouped together as a single unit and pragmatically if you look at the classes of C++ all that they do is that the struct construct of C or the record structure of Pascal has just been elevated to deal with classes. So since there is a fundamental unity between data and control there is absolutely no reason why I cannot generalize a record structure in Pascal where a field of the record is a function and the record field specification gives me exactly the kind of visibility that I am looking for. The moment I specify the record name I get access into the fields of the record so similarly the moment I specify the abstract data type name I get access to the functions inside that abstract data type but before that I do not have any access to it.

So now, for any abstract data type for which at least pragmatically you can regard as a generalization of Pascal records where there is a unity between data and functions so a record field could be a function when you look upon it that way then if you insist that every data structure data type also has among the operations that are associated with the data type are also creation and destruction operations then what it brings about is a fine interface by which there is no way of creating an instance of that data type unless you use a creation function inside that abstract data type. There is no way of destroying an instance of that data unless you use the destroying function inside that abstract data type. There is no way of manipulating several instances of the same data type unless you use the functions inside that abstract data type which allow you manipulation. So now what happens is that the interface that I have is a name of the abstract data type. What it also means is that I cannot do undisciplined or indiscriminate changing of structure or manipulation of data without the permission of that abstract data type. Once I have done that what it also means is that I can clearly separate out the specification or the interface of that abstract data type.

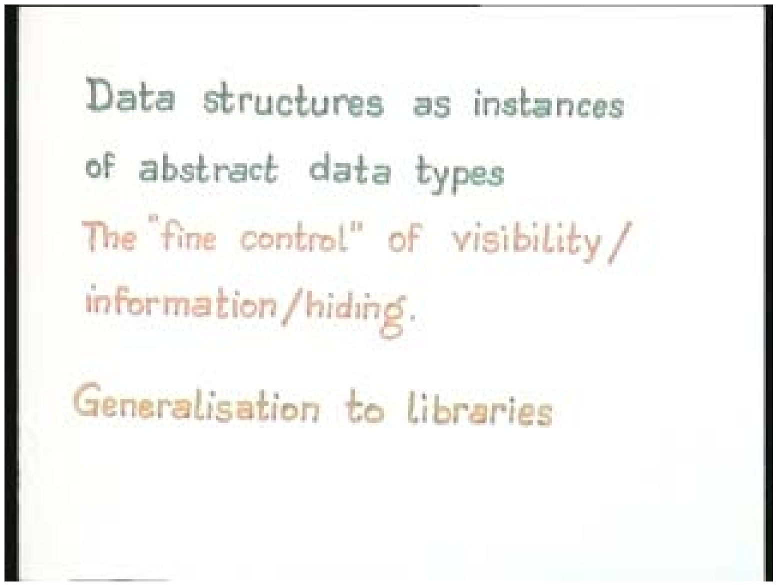
What is the interface of a procedure?
It is the name and the parameters.
What is the interface of an abstract data type?
It is the set of fields inside which means the names of the data that can be created the names of the functions that you can use. So I can separate out that interface from the implementation which means now I can change the implementation. Since the creation, destruction and manipulation of all objects created by a data type are all resident within it I can separate out the interface from the body of the abstract and I can change representations and therefore algorithms in the body of the abstract without affecting the interface, that's really what C++ classes are about. I cannot go into an instance of a class without essentially taking permission of the class.

The representation of an object in the class is not directly available to me. The only way I can manipulate that instance of a class is by using the operations defined inside that class. So you can use different representations, you can change implementations, a new fancy algorithm with a new fancy representation has come for some complex data structuring mechanism b trees or grid files whatever. What it means is that I throw out my old implementation and write a completely new implementation with new representations, new functions, new algorithms for defining the operations on it but I keep the interface intact so that all programs which use that old data type will still run with the new representation. As long as the interface does not change there is absolutely no reason why old programs should not run. These are all largely methodological issues so new features are all guided by new methodologies so the module is of Modula 2 the classes and objects of Smalltalk and C++ the signatures of ML.

If you look at the signature structure in ML and the implementation structure you have two separate units such that the signature forms the interface to any ML program which uses that data type which creates objects in the data type and manipulates them and even destroys them. Then there is a separate implementation unit which is hidden which is not available. So what it means is that I can separately compile programs with an abstract data type which means I do not have either the representation information or the algorithms available to me for that abstract data type but I can still use that abstract data type in my program and compile my program.

I can compile the specification the signature file separately I can compile the implementation separately provided the compiled version of the signature is available for the implementation in order to do type checking in order to check out that the same operations are available. I can even have lots of hidden operations which are not accessible from outside just like I can have local variables in a procedure which are not accessible from outside. Only the operations that are in the interface specified in the signature or in the module specification are actually available for manipulation and they use the representation information. So pragmatically speaking it is no big deal but when you look at it from the point of view of developing large libraries in a representation independent fashion and providing a certain fine control of visibility and information hiding then it is actually an important step forward.

(Refer Slide Time: 42:05)



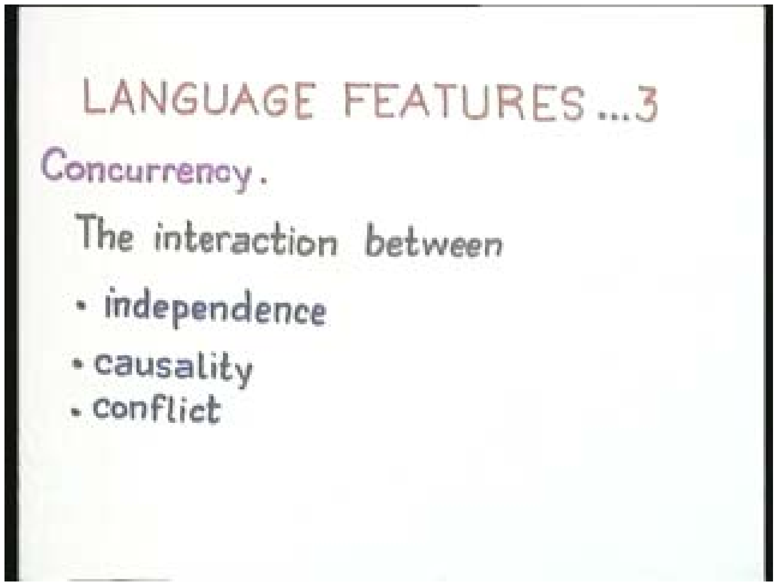
Data structures as instances of abstract data types

The "fine control" of visibility/information/hiding.

Generalisation to libraries

So this directly generalizes to libraries therefore the field of data structures goes out of the window and what you have is the field of data abstraction. The last and probably one of the most vigorous areas of research currently is concurrency. Here again the first possible language representation of parallelism probably came through the co routine concept in Simula where they actually wanted to simulate the fact that there is a CPU in a time sharing system with multiprocessing capability where a job is executed for sometime thrown into suspension and another job is executed for sometime and so on and so forth.

(Refer Slide Time: 42:35)



LANGUAGE FEATURES ...3

Concurrency.

The interaction between

- independence
- causality
- conflict

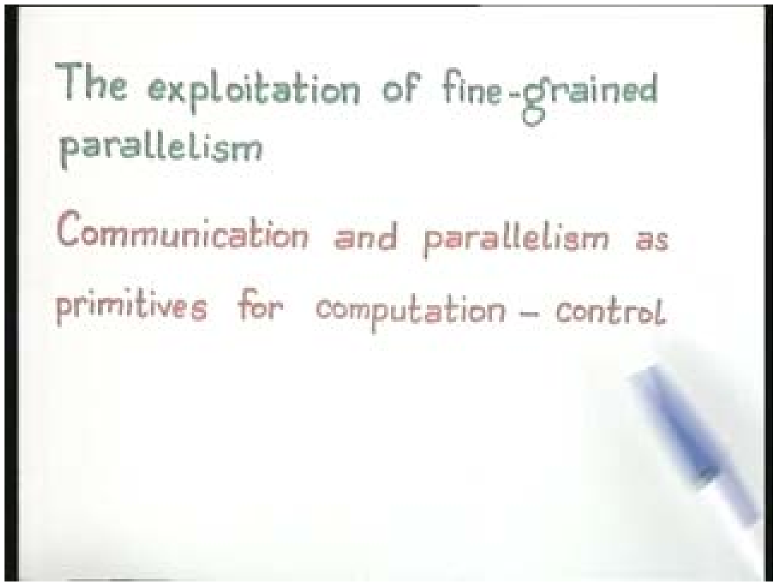
They wanted to use it as a simulation facility to study let us say operating system concepts. They also brought down the operating system concept through the co routine method into a language into the language to study this method and that gave a new method of control which is different from procedural abstraction in the sense that now a procedure from a main program is an asymmetric relationship you call the procedure and return at the end of the procedure to the main program. two co routines have a symmetric relationship, you execute part of one co routine with a resume command you move into the other co routine starting from where you left off or if it was a first call then you start from the beginning till you resume back so you pass control mutually between different co routines and that essentially simulates the behavior of jobs on a single processor system with time sharing.

You can generalize it of course to multi processor systems with time sharing, memory sharing or whatever you can generalize it further to distribute its systems with shared memory, distributed systems with local memory and no sharing or mixtures of these and what you get as a general logical notion is concurrent systems. And when you boil all this down to its basics when you look at a concurrent system it could be multiprocessing, it could be time sharing, it could be distributed, it could be memory sharing, it could be not memory sharing whatever when you look at the fundamental problems of concurrency then essentially it reduces to three important things; independence, causality and conflict and how do these three concepts interact with each other.

You can model the nature of distributed computations or time shared computations or mixtures of these by creating a new language construct which looks at these three problems and their mutual interactions and what it gives you therefore is once you have decided on independence, causality and conflict is that independence and conflict together actually give you another form of non-determinism. You can import non-determinism also. In fact it is from the elementary study of concurrency the co routine concept in Simula was a purely deterministic construct but when you analyze the large scale behavior of an operating system with respect to various jobs without knowing anything about the scheduler then you are forced to introduce into your simulation language a method of non-determinism which is not just probability based.

You want to be able to claim that those jobs execute correctly or fairly in spite of whatever may be the scheduling mechanism, you would like to prove your programs correct regardless of whether lightening or thunder strikes them and then what you have is that as an undercurrent you have non-determinism. In fact that's how non-determinism came as a construct into programming languages, the study of operating systems in bringing down operating system structures to languages or providing language support to operating system design for multiprocessor or time sharing operating systems. Then what you can do is once you have concurrency as a very general notion regardless of the underlying architecture you can actually exploit fine grain parallelism by making clear what exactly are dependent events and what exactly are independent events, what exactly are conflicting events which is a form of non-determinism and you can actually look at localized computations you can look upon the notion of a process or you can look upon parallelism in the abstract as a pure programming language construct completely devoid of its reality. When you want to get back to reality of course what you do is you map the parallelism into a multiprocessor architecture by looking at dependence, causality and conflict relationships.

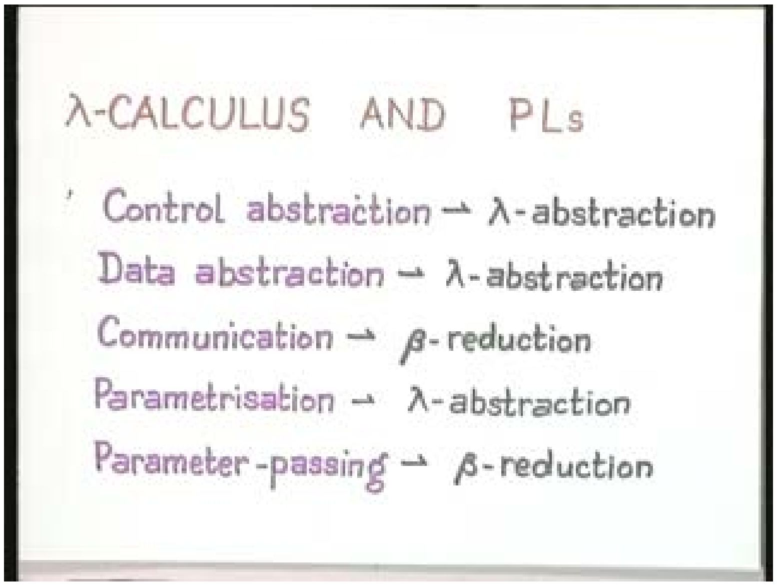
(Refer Slide Time 48:27)


The exploitation of fine-grained parallelism

Communication and parallelism as primitives for computation – control

So it is a very important and vigorous subject of study and what happens in this is that you can boil it down even further to its basics and regard communication and parallelism as the main primitives for computation control and express all possible computations in terms of communication and parallelism. And of course let us not forget one important thing there is a fundamental unity between data and control which means that control which means that under such a model firstly I can express all data as through processes I can express all processes also as data if I wanted to do it but essentially I can express all data and control as processes which somehow communicate and interact. I can even look upon the assignment statement as a form of communication between a process which is one memory cell another process which is that expression and the act of reading and writing are communications between two very small processes. And once you have this fundamental unity of course you shouldn't forget the lambda calculus.

(Refer Slide Time 50:07)



λ-CALCULUS AND PLs

Control abstraction → λ-abstraction

Data abstraction → λ-abstraction

Communication → β-reduction

Parametrisation → λ-abstraction

Parameter-passing → β-reduction

If you look at communication then it is just a form of beta reduction. The act of reading or writing the act of assignment is a form of beta reduction. The control abstraction is a lambda abstraction, data abstraction is a lambda abstraction, communication is a beta reduction, parallelism is a lambda application, parameterization is a lambda abstraction, parameter passing or instantiation is a form of beta reduction. And finally everything boils down to that.

Can you actually look at all these kinds of behaviors as forms of beta reduction? What are the abstraction mechanisms that you can impose on top of concurrency? What are the type checking mechanisms you can put in? How can you do how can you do higher types over communications? Can you define higher order processes just like you did higher order functions? What is lambda abstraction over higher order processes mean? What does parameterization do? How do you map process to processor?

How do you map the real life situation which is a geographic distribution of some sites to an existing abstraction and the importance of that abstraction is that if you were to change the architecture of your distributed system your abstractions still stands and you can do a fresh mapping of process to processor without changing your original arch. It is a new method of programming which looks at fine grained parallelism, fine grained independence, looking at essential conflict relations, essential causal relations and based on that you do a process to processor mapping and that's what the future holds in the light of the lambda calculus. So the most important thing to do is to study zen and the art of the lambda calculus.