Principles of Programming Languages

Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi

Lecture no 25

Lecture Title: The λ Notation

Welcome to lecture 25. We will talk about the λ notation. Long ago in the year 1900 the great German mathematician David Hilbert actually in the world congress of mathematicians enunciated a program for the next century. He said that the outstanding problems that are going to govern mathematics in the 20th century are going to delve on what exactly the notion of a computation is. The main question Hilbert wanted answered was how much of mathematics is actually mechanically computable by a machine? What he wanted was not the actual presence of machines but he wanted a logical concept of what constitutes mechanical computation.

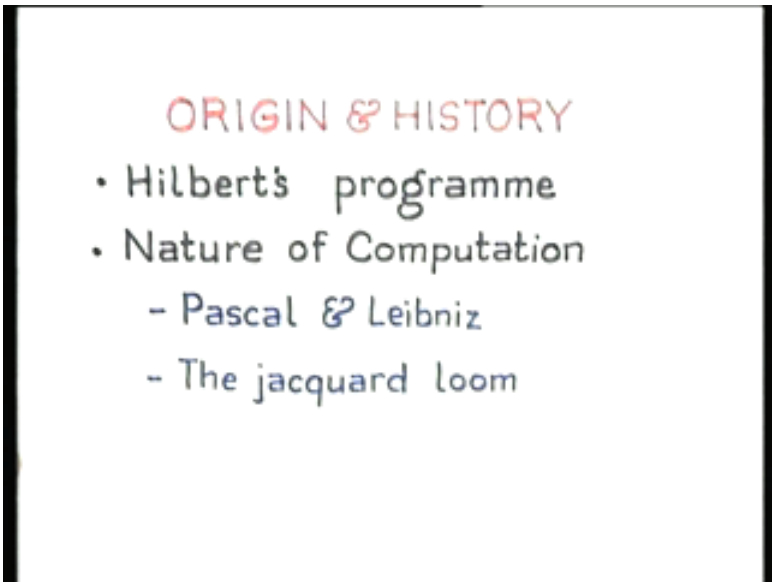
This means that you are really looking at some very fundamental nature of a computation. If you look at the scene at that time firstly the fact that some things are mechanically computable was already known because for example Pascal and Leibniz had created mechanical calculators. Pascal's father was some kind of an accountant and he had to do lots and lots of arithmetic which is why Pascal decided that all this was getting out of hand so, the need of the hour was to use gears etc and create a calculating machine. In fact Pascal was quite successful. What he created was a calculating machine with gears and wheels using gear reduction which was very close to the registrics machines that used to be available till a few years recently. He had the 'carry and borrow' for the arithmetic integer operations and Leibniz had something similar. Since Leibniz was originally interested in the concept of what is probably now called linguistic philosophy he was interested in the notion of symbols and symbol pushing. The closeness with which differentiation is associated with division led Leibniz to come out with this notation for derivatives which we use commonly now.

There was a large amount of stuff that predated the current linguistic philosophy from Leibniz and Descartes onwards. Also there were lots of other concepts which people were aware of. One is the jacquard loom which consists of what might be considered a primitive stored program concept in the sense that there is a card with holes in it and you use that card essentially to weave patterns on a cloth.

In fact the first jacquard loom probably came up in the 18thcentury and had very complicated patterns for example if you were to see the Baluchari saris which were produced in Bengal in the turn of this century and the last century they actually are the first instances of the use of an automatic loom with a stored program concept to define weavings in such a fashion as to come out with thematic designs on cloths. They are not just repetitive designs. There were whole themes like the Kurukshetra war or the Bhagavadh Geetha. One whole sari would just have a weaving which denotes these

themes. Those cards were probably the first programs if you were to look at it fairly objectively for a certain model of computation which is other than mathematics.

(Refer Slide Time: 5:38)

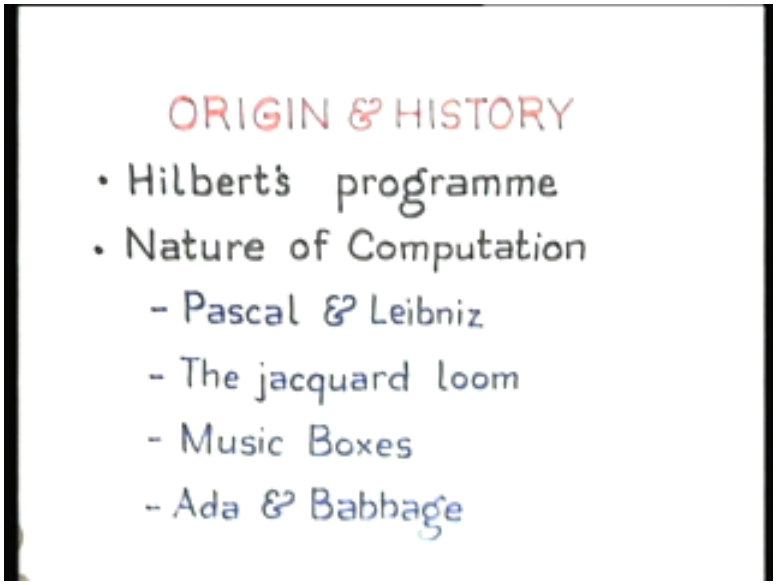


Also well known were the music boxes. The whole idea is that if the music is sufficiently good then the thief who comes to steal your jewellery will just stand perplexed there and be caught by the police. These mechanical music boxes were very common for example in Switzerland and they use a clock mechanism and a whole song could be programmed with just wheels and gears. It is a purely mechanical contrivance with just weaving gears under wheels and a certain amount of ratchet pinion mechanisms etc.

There are some gigantic music boxes which were produced in those days because a countess would probably have such a large number of jewellery that she would require a really mammoth music box.

(Refer Slide Time: 05:53)

(Refer Slide Time: 7:14)



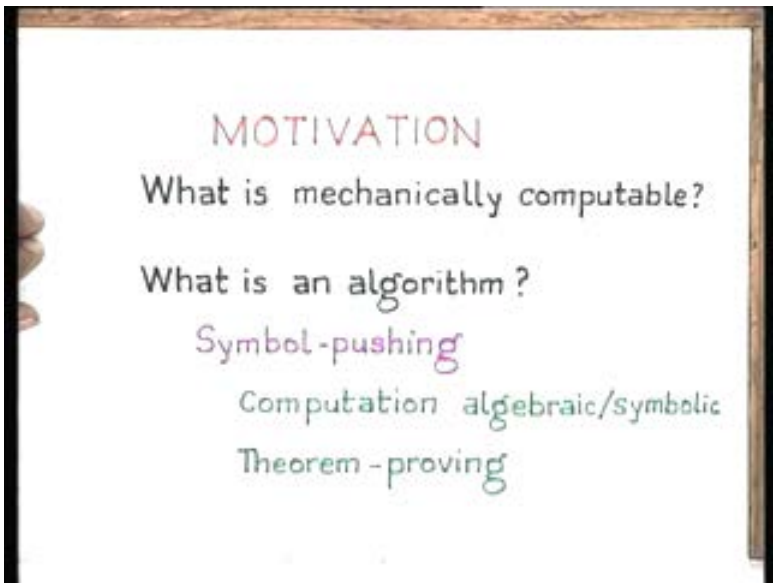
The first mechanical machine was created by Babbage on paper and the first programs were written by Countess Ada Lovelace a niece of Byron. The lady Ada Lovelace actually wrote programs for Babbage's analytical engine in a notation which could be read through cards very much like in a jacquard loom and could be translated. The fact is that Babbage's machine was never produced. It is still there as blue prints. He never had the money and nobody would finance him but Ada probably gave the first proper notation for writing programs for computations on Babbage's machine.

With all this as a background the notion of an algorithm was sort of informally known for the last 2000 years starting from Euclid's algorithm for gcd the co farathas thenis and in the work of Valharrisme and if you look at the 1900s Russell and Whiteheads' book on Principia Mathematica had just been published at Russell's own expense. It was the hay day of linguistic philosophy and the whole philosophical movement and logical movement at that time was to somehow define everything rigorously through the syntax of a language.

Hilbert raised an important problem in his program when he spoke about mechanical computation. We know that Euclid's algorithm is an algorithm. We know various things are algorithms but the question is: what exactly is an algorithm? Why is not a calculation of convergence up to a limit an algorithm? Or is that an algorithm at all? How can we think of certain things as algorithms and what are not algorithms? So, fundamental to Hilbert's program was the question of having to actually define the notion of an algorithm. It had to be accurately defined and only after you have defined the notion of an algorithm or given a sufficiently precise characterization of what exactly constitutes an algorithm can you really talk about mechanical computation; a model of computation that is based on algorithms and only then can you talk about what part of mathematics is actually mechanically computable.

For example can theorems be proved mechanically? Can we talk about Hilbert spaces? Is Cantor's diagnolization a mechanical procedure? What about the proofs of geometry? What about the constructive proofs of geometry? What about the proofs by contradiction? Are they all mechanically computable? Could a machine have come out with these proofs? These are the sort of questions that were raised by Hilbert's program and there were several attempts in different ways. One was to take the actual view that really Hilbert is talking about the construction of a machine. Is there a machine which you can construct? The second is: are Hilbert's questions merely of a logical nature? So, they have to be answered purely mathematically. These different aspects were actually taken up by different people. The questions finally boil down to loosely speaking; what is mechanically computable and what is an algorithm?

(Refer Slide Time: 11:58)



A large amount of consensus because of the prevailing linguistic philosophy atmosphere like the Vienna school and Russell's Principia Mathematica made it seem more or less clear to most logicians and mathematicians that really what should go into the basics of an algorithm or a computation is symbol manipulation and somehow we have to think of computations in terms of symbol manipulation and then the whole idea was that any other thing which looks mechanical could probably be converted into a notation of symbols and then we could think of it again as symbol manipulation.

This symbol pushing was quite an attractive idea because a large amount of algebraic computation was really symbolic manipulation and with logic being described in a purely symbol pushing way, it looked like a large amount of mathematics could actually be expressed completely in terms of symbol manipulations. In fact the whole idea of defining a logical language and a set of inference rules which were very closely related to symbol pushing was actually to algebraise geometry. One previous algebraisation of geometry was by Descartes in the sense that he actually transformed all geometrical principles into pure algebraic principles in coordinate geometry but Descartes' notion never actually exploited symbol manipulation to a very rigorous extent and the whole idea constitutes geometric reason? What makes Euclid's elements rigorous and what constitutes a rigorous proof? Why is not the parallel postulate provable from the other axioms of Euclid? The only way one could think of actually looking at what constitutes correct geometric reasoning is if somehow one could make theorem proving in geometry also somewhat mechanical then everybody could be assured that if you enunciated the principles of a proof in purely mechanical terms then most people may not be able to prove any theorems but at least most people would be able to check whether a proof is correct.

If the symbol pushing followed a certain finite set of rules then you could be at least assured that a proof of a theorem is right. If it is going to be symbol pushing then is theorem proving also mechanically possible? What kinds of theorems are mechanically proved? If you look at the algebraic theorems a large amount of them are really mechanical symbol processing and could perhaps be done by a machine. If you can convert geometric reasoning also into some logical notation and through the use of pure symbol mechanisms with no notion of meaning and if you could define inference rules then perhaps even a large amount of theorem proving in mathematics could be done mechanically. Even if you do not attribute creativity to a machine at least the checking of a proof could be done by a machine. That is the reason when you learn logic you learn it as a purely symbol processing discipline.

Then in the rest of pure mathematics there was also the question of foundations of mathematics. There were lots of things which were not understandable. In cleaning up the foundations of mathematics they decided that what is fundamental to mathematics is the notion of sets and that all of mathematics was being rewritten so that it conformed to sets as a basic notion in mathematics. By sets we are talking about the membership, the empty set and equality of sets etc.

All of mathematics was actually rewritten including Euclidian geometry because there were lots of linguistic problems with a construction of proofs in many of the theorems enunciated by Euclid. While they were more or less rigorous there were lots of ambiguities and sometimes there were hidden assumptions which made one theorem actually dependant on a theorem that came later. So, the notion of circularity had to be removed and a large amount of mathematics was being rewritten in terms of sets and in terms of a collection of postulates or axioms. But in all this the question is again of what exactly constitutes a computation and is it really different from the rest of mathematics? Can you use sets themselves as a basis of computation describing computations?

It is not clear actually and there was a school of thought started by Sion Finkle which gained much appreciation in the last twenty years called combinatory logics. He used symbol processing as a basic mechanism to define a set of rules for what he called a computation. That led to the development of combinatory logic mostly through the efforts of Haskell Curry. Sion Finkle himself does not gain any reputation for what he did but Curry took it up and for the next sixty years he and a lot of his followers actually developed it into a fairly perfect programming language. There is this programming language called Haskell that is based on Curry's combinatory logic which is the interpreter for which was written less than seven years ago.

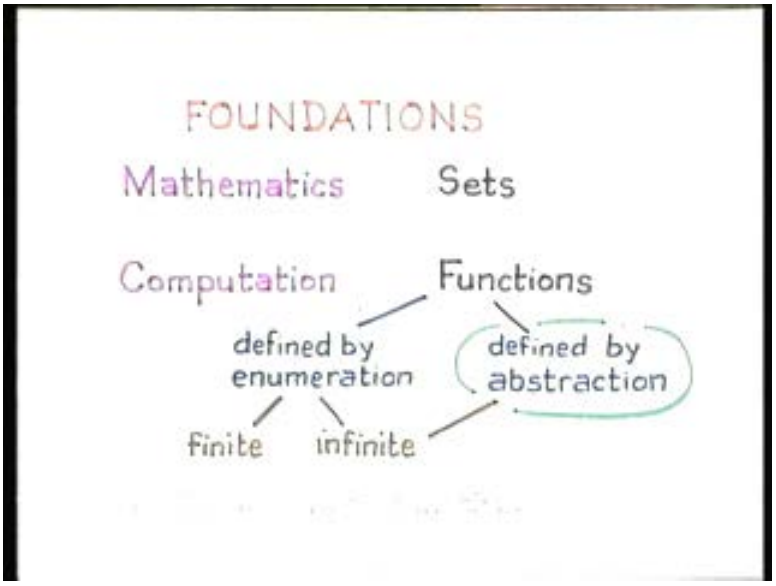
But probably a more important influence was that if you are looking at computations then you should consider the notion of a function as basic to computations. This was a view enunciated by Alonzo Church who was at Princeton University at that time. His basic idea was that sets might be basic to mathematics but what actually is basic to computation is the notion of a function and his whole idea was to answer Hilbert's main question; what exactly is mechanically computable by using functions as the main notion? But there is a problem.

If you look at set theory in mathematics, since everything else is to be defined in terms of sets you cannot posit the existence of individuals. The only foundation that is there in mathematics is that of a set itself. What is there is just the empty set. Everything else; numbers, ordinals, cardinals and functions should be defined in terms of the underlying set theory and the idea was that set theory itself should be defined as an exercise in first order logic definition.

The axioms and the postulates of set theory should be clearly enunciated so that you get a clear and consistent view of sets. You also somehow allow infinite sets through and the infinity axiom. Through the axioms you have to capture the fact that sets cannot contain membership relation that is infinitely descending but at the same time there is no basic element except a set itself and there should be no assumption of individuals. Since sets are supposed to be a conception of the mind and do not bare any relation to reality and all our mathematics has to be expressed in terms of sets the only way you could define sets is through a collection of axioms very much like the only way you could define points, lines and planes in Euclidian geometry and that is through a collection of axioms which characterize their properties of points, lines and planes. You cannot use a definition because if you use a definition then you are assuming the existence of something else which is more basic for the concept. The only way to define sets is to use characterizing axioms that will exactly characterize a set's meaning which will not allow the construction of sets that somehow lead to inconsistencies. There are lots of them like Russell's paradox, Burali Forti paradox etc.

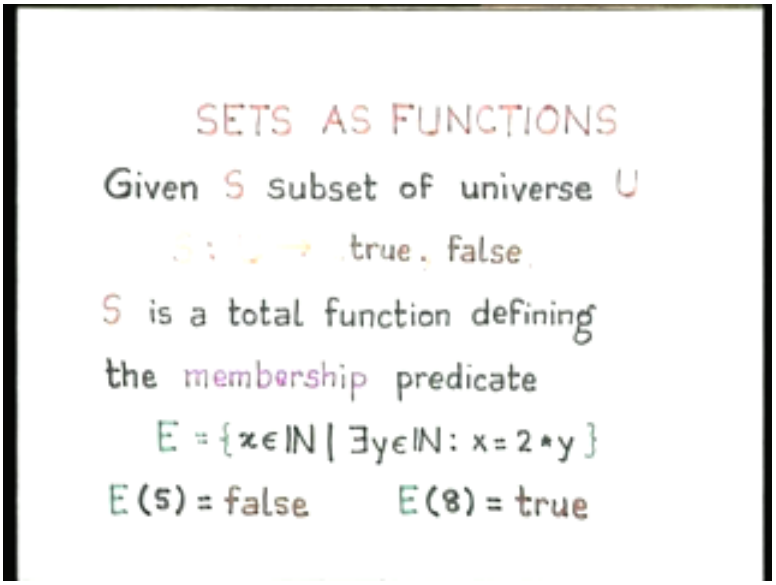
If you take a similar view to computation and if you regard functions as the most basic element of computation then all that you can think of is to take a first order logic like view. This means that you have an infinite collection of variables, variable symbols and you have to define the notion of function somehow by just manipulating these symbols. These variable symbols are all indistinguishable in the sense that there is no notion of types. For example, types have to be built up later somehow. Each symbol is as good as any other and they do not have any special status except for some since any first order logical language has some basic constant symbols. So, the only distinctions you can have are constant symbols and variable symbols. Actually Church found that the analogy with sets was quite proper. You have two ways of defining sets. One is you can define them by enumeration or you can define them by abstraction.

(Refer Slide Time: 24:18)



Defining by enumeration is the roster form as they call it in school and defining by abstraction is a set builder form as they call it in school. When you define by abstraction you are usually using a logical predicate to define a set and it has to be a first order logic predicate. Similarly, there are two ways of looking at functions. You can actually define them by enumeration or you can define them by abstraction. Actually Church also showed that sets are also really functions. If you look at them carefully for any subset of some given universe you may define a set as really the membership predicate.

(Refer Slide Time: 25:30)



Any set which is contained in a universe U is really a predicate and therefore has a logical status of a predicate and I can think of this set as a function to which I can apply a value.

$$\mathrm{E} = \{x \in IN | \exists y \in IN : x = 2 * y\}$$

$$E(5) = false$$

$$E(8) = true$$

What do you do when you actually do a function application? Let us look at this set definition work carefully in view of whatever we have done in programming languages. There is a binding occurrence of x and there is an applied occurrence of x. There is a binding occurrence of y here which is actually bound by this quantifier and an applied occurrence of this y. If you look at this predicate, x is free in this predicate and y is bound.

What does function application mean? Function application means that you take a set binding occurrence of this x and not a binding created by quantifiers.

$$E(5) = \{x \in IN | \exists y e IN : x = 2 * y\}(5)$$

So, you are essentially taking all free occurrences of x here and syntactically replacing them by 5 and that is function application. This function application is actually very much ML like. It is an ML like functional application and we might think of it as a pure syntactic substitution. We can think of this expression or this predicate as being obtained from this predicate by an application of this function on 5. What does function-application mean? Function application just means a syntactic replacement in the body of the function of all free occurrences of this bound variable by the argument.

In fact Church felt that our normal mathematical notions of functions and their function application was exactly that. Except that mathematicians over the last 2000 years were a little careless about notation and so, the whole idea was to somehow clean up that notation and make two points very clear. One was that mathematicians usually confused function definition with function application in their notation. It was never very clear except when they explicitly told you verbally what a function definition is and what actually function-application is because the only mathematician's view to define functions was by their application to some argument.

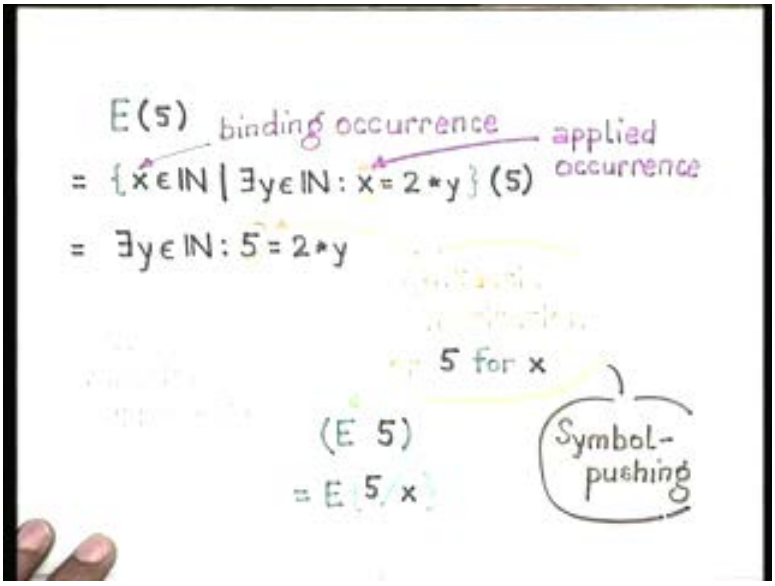
Let us say you take a polynomial like this. It was verbally stated that a, b and c are constants where 'a' is not zero. The whole idea was that this whole thing is a function but I cannot express it in anyway except by actually defining the effect of that function on an arbitrary argument x. All function definitions were function definitions by applications. Very often mathematicians would use something like this to denote that there is this concept of a function which I do not know how to define. But let us say I have defined its effect on an argument x like this. I am now defining a value which is the value obtained when this function is applied to this value x NOT.

There was no distinction except by verbally specifying what is arbitrary and what is particular. They never really distinguish between function definitions and function applications but actually conceptually they are two different things and that conceptual difference has to come out in their notation. The conceptual difference between the definition of a function and the value obtained by applying that function on to an argument are two different aspects and the difference in the two concepts should come out.

How can you define a function otherwise? How did you define the set as a function? You bound the variable and likewise you just bind this variable and you have to give application rules. All that Church wanted to say was that there is a conceptual distinction between definition of a function and the effect of the function on an argument and that conceptual difference was more or less obscured in all mathematical texts. That conceptual difference was only made clear verbally by specifying that let this x be some arbitrary thing or actually specifying; consider the function f of x equals this, now at the value x NOT consider f (x NOT) but this f (x NOT) is not a function; f (x NOT) is a value. That is what they meant but it was not clear from their very notation itself.

If you had removed all the plain text from their notations you would be quite at a loss to know the difference between f (x) and f (x NOT). In your language of mathematics you should be able to make this conceptual distinction between a function and its application keeping in mind the fact that the only way often to define a function is by means of its application to some argument.

(Refer Slide Time: 33:50)

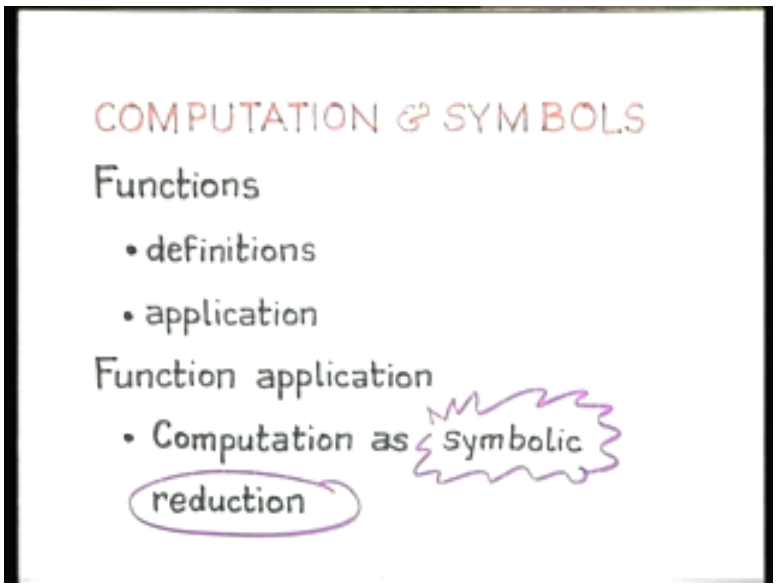


If you look back at this function application Church also said that function application by this kind of syntactic substitution is actually a reduction of the function and this sequence of steps actually constitutes a computation and the computation is purely symbolic in the sense that you are using only syntactic substitutions.

The point about this emphasis on functions is that just as sets form the basic foundation of mathematics you could not use for example numbers to define sets except of course in a metasyntactic way. You might define a list of sets; s 1, s 2…s n but that s 1, s 2…s n are not numbers. They are not numbers as a value domain of interest. They are just symbols for you to play around with to distinguish between 1 s and a different s.

Since functions are basic to most of mathematics and to any other fields of endeavor the whole point was to concentrate on the notion of function and clean up the notation in such a way that your new notation is independent of everything else and can be applied in all areas. If you are looking at a general notion of computation and you have defined the notion of a function quite precisely and independent of all other concepts where functions are used, then you could import this new cleaned up version and use it in every discipline where functions are required. Then you could define notions of computation regardless of your underlying models. You could define notions of computation in particular models derived from this basic notion of function and function application and this basic notion of a computation as a series of syntactic reductions. The whole idea was to really make this distinction clear.

(Refer Slide Time: 36:18)



Look at any of the developments of logic. There is a pure logic in which there is absolutely no mention of an underlying model or a particular discipline. The entire logic is itself defined in such a way with axioms and symbols and a language that you could apply it to any other discipline. You have pure first order logic which just assumes that there is some domain of discourse and that values are drawn from the domain of discourse. Nothing of the properties of the domain of discourse interferes with your logical reasoning. It just assumes the existence of a domain of discourse; it just assumes the existence of a few function symbols of given arity etc. You develop your theory of first order logic completely independent of any particular mathematical discipline.

Once you have developed your theory of first order logic you have certain logical theorems, logical axioms and logical postulates which are true across any mathematical discipline. You can apply this logic to the theory of numbers and then you get the first order theory of numbers. In addition to the logical axioms and postulates, you also have the axioms and postulates of number theory let us say, Peano's axioms. For example Peano's axioms do not come as part of logic but when you apply logic to number theory you might use Peano's axioms and in addition to the logical axioms you will also have Peano's axioms and then you will try to derive theorems about number theory.

That logical language and its theory form a sort of underlying foundation and when you apply it to some other theory you have extra axioms and postulates of that theory. For example; you could just as well give the first order theory of plain geometry. Then you have this logical foundation in terms of first order logic and then on top of it you have may be the theory of real numbers and then on top of it you have the theory of Euclidian geometry maybe. In your Euclidian geometry, if you look at the proofs carefully, you will find that you are using theorems, postulates and axioms which are sometimes logical in nature where all your proofs by contradiction are all of that kind and you are using theorems axioms and postulates of real numbers. That means Euclidian geometry assumes already that you have real numbers otherwise how do you have Pythagorean theorems and the converse of the Pythagorean theorems? You are using real numbers as a field in that theory and on top of it you also have the normal axioms which define lines, planes and points.

You are using all of these theorems, postulates and axioms in proofs and they are so neatly stratified that the lack of circularity in your proofs in geometry is guaranteed provided you only ensure that you are not using a result which you are going to prove later or result in geometry which you are going to prove later. But you can use any results from real numbers and any results from logic that you want. By neatly stratifying your theories that is by resting one theory on top of another as a foundation, you can actually have a better control over what might constitute inconsistency in the theory.

Similarly, you want the definition of computation also to rest to form a foundation on which all other theories which use functions can rest and therefore you can define computations on all those theories. We require some basic notion of function and function-application which is independent of any particular application or any particular discipline in which we are talking about computations. Since you do not have anything else you really have to look upon this computation as a purely symbolic reduction.

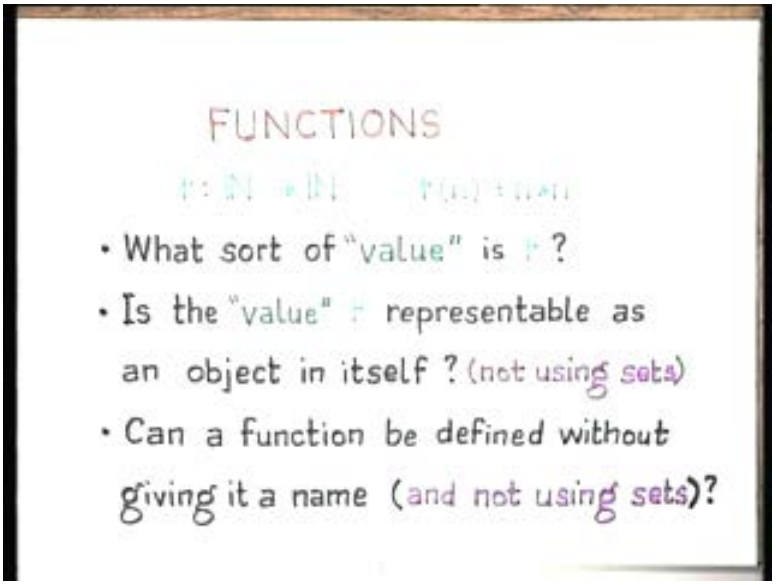
Let us come to more down to earth aspects. Let us say you have a function from natural numbers to natural numbers of this form. If I did not give a name to this function and if there was not already a name for this function then I am defining this function by means of its application on a natural number and the question that arises is: what sort of an object is f itself? This f is a name for an object which represents a function. What sort of value is f because I can think of the space of all functions from natural numbers to natural numbers itself as a set on which I might define more functions; I might define the set of

all functions which transform functions from natural numbers to natural numbers to functions and from natural numbers to natural numbers again.

This function space itself can be regarded as a set of points and they can be regarded as a set of values. What sort of a value is this function in the set of all functions from natural numbers to natural numbers? Can you represent this value f itself as an object in itself? A sufficient distinction is then made and it is clear what an object of n is and what an object of $n \rightarrow n$ is.

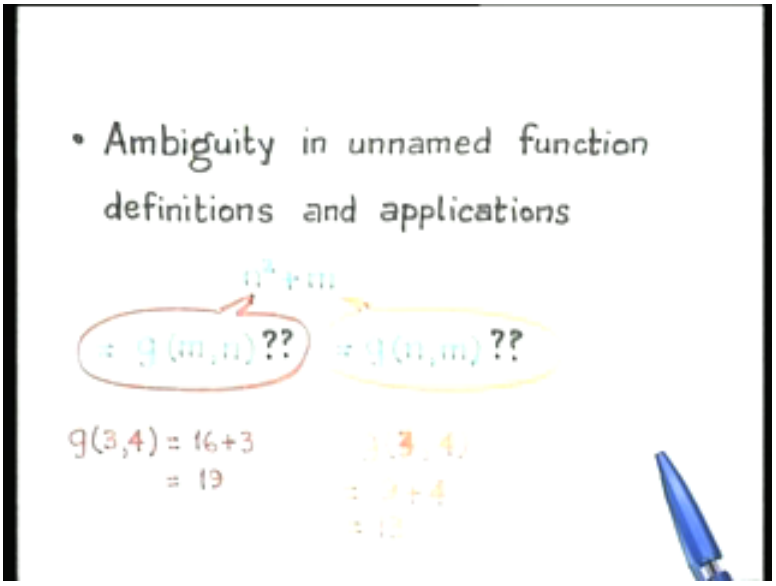
If you have decided that computation functions are basic to computation then you cannot use sets. What is done once you define set theory in mathematics is to define functions as relations of a particular kind which are also sets.

(Refer Slide Time: 43:27)



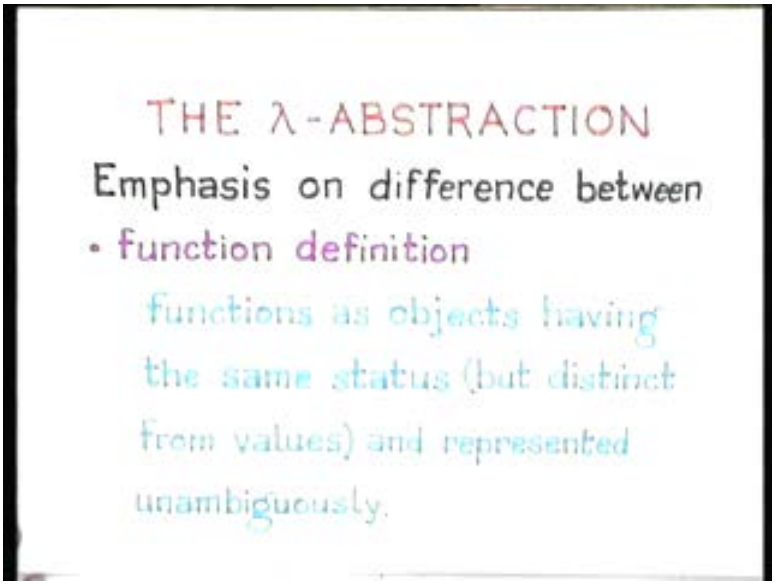
You are not allowed to do that because you are considering functions themselves to be absolutely basic. They form the foundation of all notions of computation. Essentially, can a function be defined without giving it a name? Of course you are not allowed to use sets An associated problem is that you should get rid of ambiguity somehow. If I give an unnamed function definition like this then it is clearly a function of two arguments n and m but is it gmn or gnm? In other words is it g applied to mn or nm? Depending on how you look at it you might actually get different results.

(Refer Slide Time: 44:18)



You have to be somehow consistent with normal function notation. You change it a bit to avoid ambiguity but not too much, meaning, the basic notions should remain the same. You should also be able to get rid of ambiguity but essentially once you have cracked the problem of how to define unnamed functions and treat them as linguistic objects in themselves and define function application in a purely symbolic fashion then you have the notion of a computation. Church said that whatever you can do with just those two notions is all that is computable. Whatever is represent-able linguistically with just these two notions is the concept of an algorithm. Anything that is not represent-able by this is not an algorithm and nobody believed him. The whole point was that what Church said turns out to be consistent with other theories on computation. The whole point was what he defined was what he called a λ-abstraction.

(Refer Slide Time: 45:40)

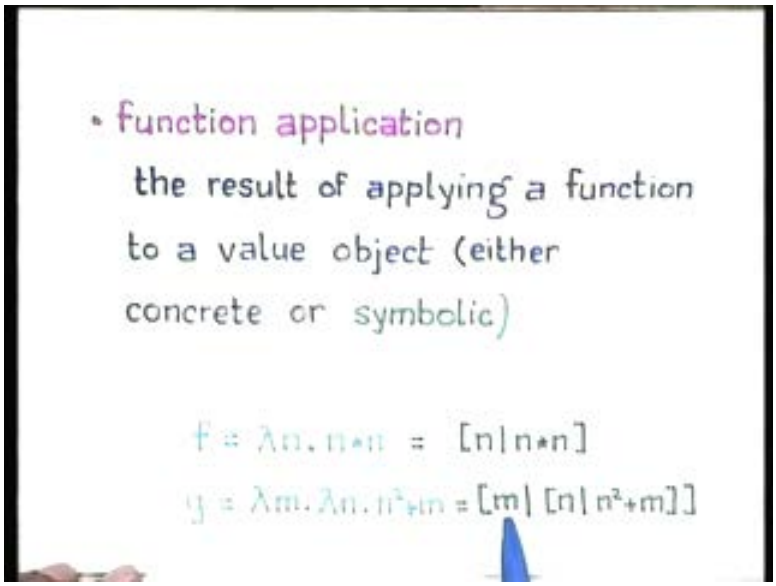


I am not absolutely certain about why he used λ but it was fashionable to use Greek symbols those days. We have to differentiate between function definitions and functions as objects having the same status but distinct from each other and represented unambiguously. So, functions and values should have the same status. Secondly, you should be able to define function application in a domain independent fashion and in as symbolic a manner as possible. The whole function has to be linguistic and symbolic or syntactic.

What Church actually used as his λ notation which I am not going to use is this. I do not need to give names to these functions. This string just represents the squaring function on natural numbers and I do not have to give it a name. This clearly tells you, unambiguously, which is the first argument of the function which calculates $n^2 + m$ and which is the second argument of the function. These arguments are taken one at a time. He resolved this problem by using a local declaration. In modern programming language terms that is what it means. You have a local declaration and a function has a local declaration or a parameter declaration and a function body. Whatever is a parameter of the function is abstracted away that is made local through such a declaration.

What distinguishes n*n from λn*n? In n*n the n is free. In λ n * n the n is bound. There exists a binding of n which that λn does. It is like a local declaration very much like whatever we have done previously. Of course once you have a local declaration you also have the notion of a scope. This is the scope of this local declaration. Similarly for λm this whole thing is the scope and for λn this whole thing is the scope. So, you have nested scopes. You have cleared the ambiguity in function applications by using nested scopes. But I will use this notation which is a parallel to the set notation. This is like a normal set notation for procedures for example in Pascal. You have procedure heading, a function heading and local declarations ‘begin end’.

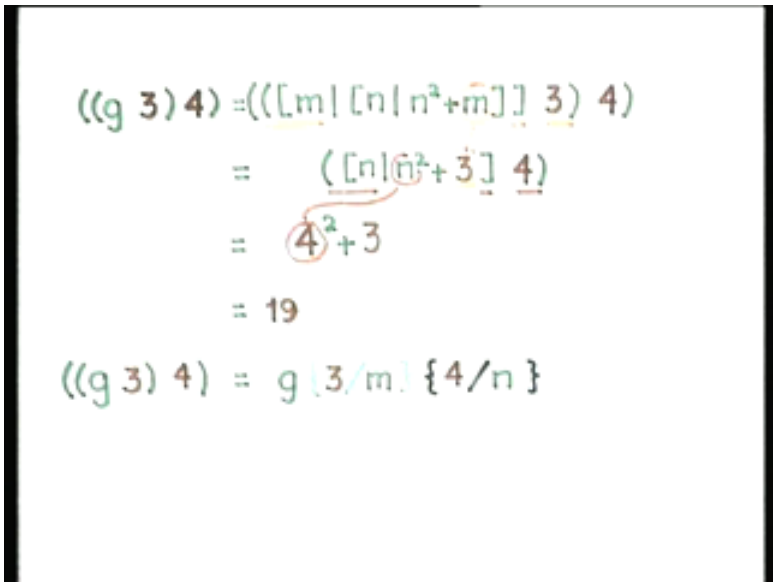
(Refer Slide Time: 49:50)



You have this reserved word here; you have another reserved word and you have a clear delineation of the scope. You first have a local declaration for m and then you have a function defined inside and you have nested scopes. Then what does function-application do? You take a function definition and apply it. An application is denoted by a pair of parenthesis around the function and its argument. This g 3 4 is really the application. You apply 3 to the function in the outermost square brackets; you enclose the whole thing in parenthesis and then you apply 4 and you enclose the whole thing in parenthesis. Syntactically it is very simple. You have a function application if you have a left parenthesis and a left square bracket occurring one after the other that is occurring immediately in sequence.

When you have this occurring in sequence you have to match those corresponding brackets. This application clearly says that you have to replace all free occurrences of m in the expression by 3 and you get that here. When you replace all free occurrences of m by 3 the application gets rid of these parentheses and the brackets. It is again clear that it is another application in which there is a left parenthesis and a square bracket with the matching square bracket and the right parenthesis and you have the argument there. This application clearly says that you should replace all free occurrences of n in the scope by 4. Then the rules of your discipline such as numbers or addition or multiplication somehow give you the next step. But that is not part of the λ notation. The λ notation just gives you the two steps of the computation as a form of syntactic reduction.

(Refer Slide Time: 53:15)



This application is actually g with all free occurrences of m replaced by 3 syntactically. That means anyone without knowing mathematics should be able to look at these patterns and do it. Rather a machine has to be able to do it and once you have done this the next is a syntactic replacement of all free occurrences of n by 4.