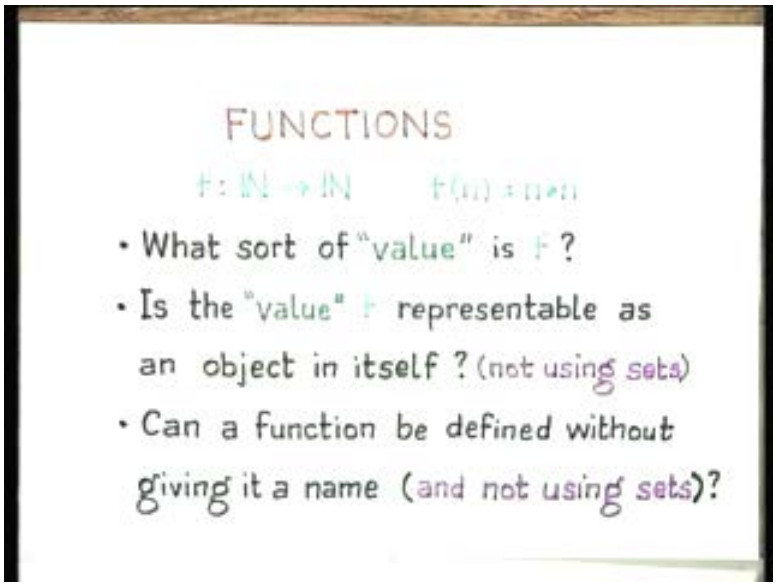
Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi
Lecture no 26
Lecture Title: The λ Calculus

Welcome to lecture 26. We will start the λ calculus proper. Let me just briefly recapitulate what we did in the last lecture.

(Refer Slide Time: 00:42)

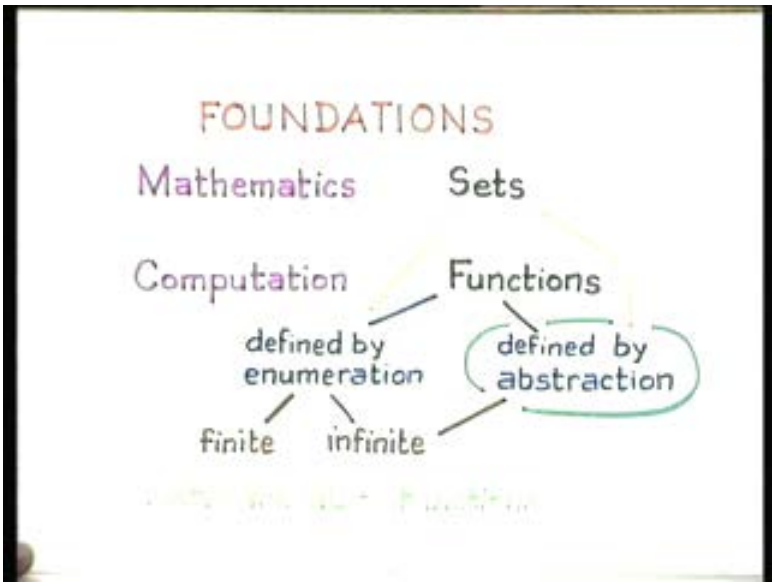


The whole idea of the λ calculus was to look upon functions as first class objects; to be able to say what kind of a value a function is, to look upon function spaces themselves as sets of points, to give unnamed functions a characterization and which essentially meant that you have to treat functions and also more importantly to give a fundamental theory of computation. I pointed out the parallel with sets that is when it comes to mathematics you consider sets to be a fundamental object. They reformulated the whole of mathematics so that everything depended on just set theory and set theory itself was axiomatized in first order logic.

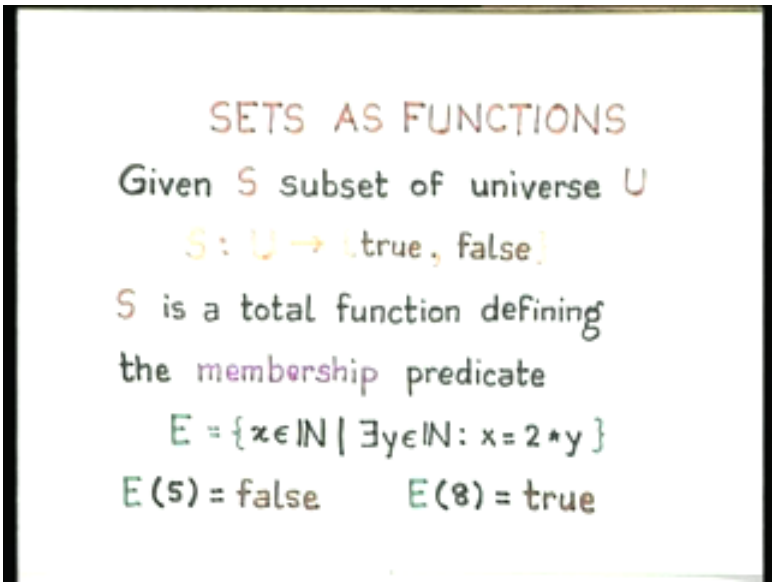
Similarly, Church attempted using functions that is the concept of a function as the fundamental object of a computation. You could also define sets as functions through the membership predicate and as I said there is a lot about the membership predicate which can be made to look like function application if you try hard enough. The main important point about function application is that you are really doing syntactic substitutions and it is totally symbolic. Even a machine could possibly do it. The λ abstraction so to speak or the λ calculus emphasizes the difference between function definitions where functions

are treated as objects and application of functions; the result of applying a function to a value object which might be either concrete or symbolic.

(Refer Slide Time: 01:12)



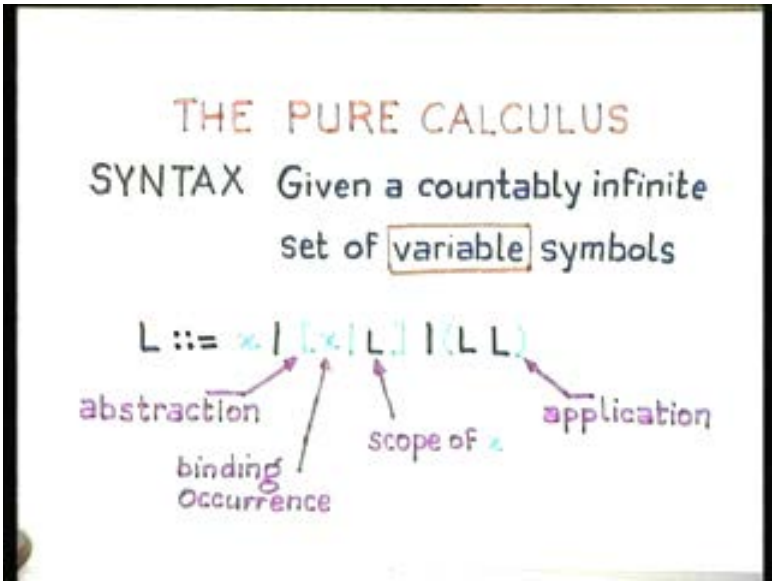
(Refer Slide Time: 01:56)



We will start with the syntax of the pure calculus. The λ calculus is important because firstly it can be regarded as the mother of all programming languages especially the functional programming languages. It was the first programming language with a cleanly defined syntax and semantics and it was never implemented till very recently. I will first look at the pure calculus. In the case of any logical theory you assume accountably

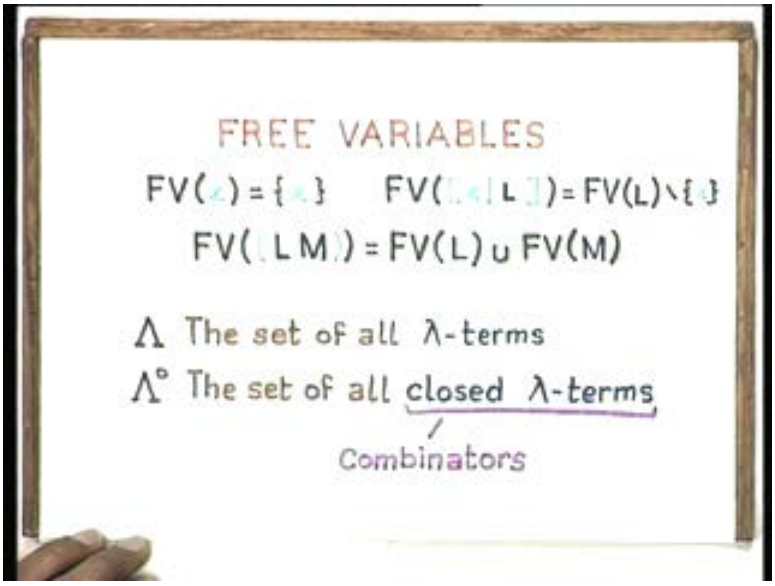
infinite set of variable symbols given to you and you have nothing else. Everything else has to come from the λ calculus itself.

(Refer Slide Time: 04:05)



You do not have any predefined sets of objects unless you are applying the λ calculus to some existing domain. The grammar is actually very simple. Any variable symbol is a λ term and I have the concept of λ abstraction which contains a binding occurrence of a variable and a λ term. I am trying to maintain the analogy with sets as far as possible and then for this binding occurrence there is a certain scope defined by the body of the λ term. If you have two λ terms the application of one λ term to the other enclosed in parenthesis is also a λ term. It is a simple syntax with essentially just two constructs and as usual we could define the notions of free variables and bound variables. For the λ term x, x is the only free variable for the λ term which I will call λXL. You will find Church's original notations in most text books.

(Refer Slide Time: 05:17)



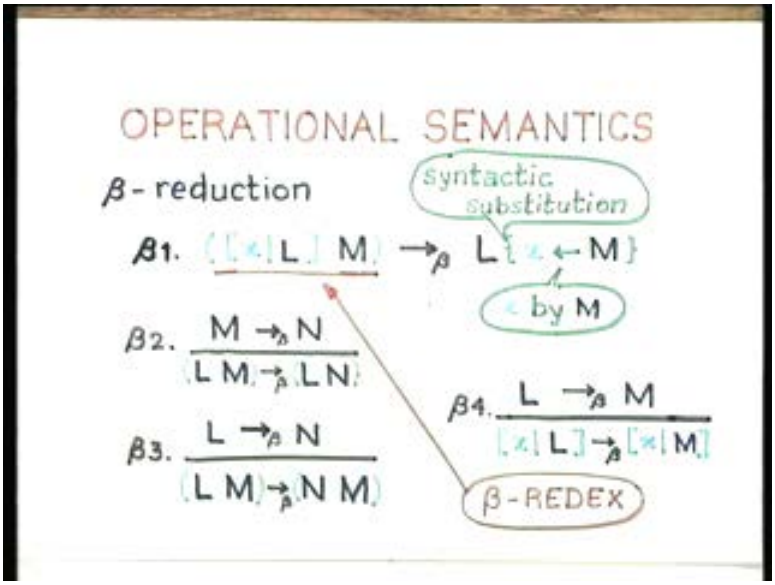
I like this analogy with sets so, I will use this notation and call it λXL and this has all variables that occur free in L except S as free variable. In λ application the free variables are each of the constituents where loosely speaking this λ term L is like an operator applied on an operand however they are both λ terms. Since you are starting essentially from nothing except variables you really cannot distinguish between the function and the value. It is very general in the sense that functions might be applied to functions; some of them might be meaningless, some of them might be meaningful but if you allow this facility then you will also get higher order functions.

I am using blue though it is against my convention to use it for a programming language for a very simple reason that most beginners find the λcalculus very abstract and that is why I thought we should use blue for that. The set capital λ is the set of all λ terms and the set of all closed λ terms. That means those λterms which have no free variables is the set of combinators. So, we have defined the syntax. Now let us define the semantics. The semantics really is through a notion called β reduction.

Let us suppose you have a λ term. It is λ abstraction applied to another λ term. Then the result of the application is that whatever is in green is a substitution. The whole point is that whatever is in green is not part of the λ calculus language. But if you remember I have to somehow express the result of this application in some form in terms of L and M and what I claim is that the result of this application is: you take the pattern of L and take all free occurrences of X; remove those free occurrences of X from that pattern and substitute the copies of the pattern M. This is a syntactic substitution which literally does a pattern matching and replacement. This operation in green braces is metasyntactic. It is at a level of abstraction above the syntax of the λ calculus. You take this pattern L; look at all free occurrences of X in it and replace all those single string x patterns by entire

copies of the pattern M. This is called a β reduction. Such a pattern which consists of λ abstraction applied to another λ term is called a β -Redex.

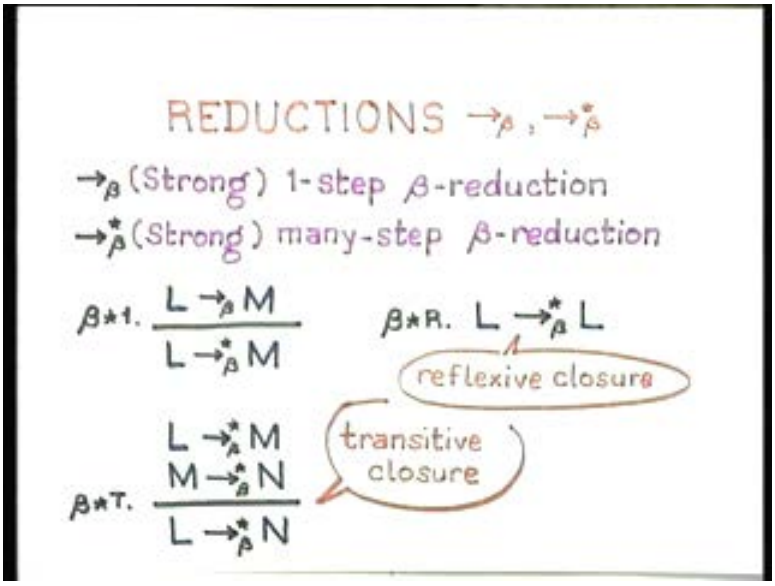
(Refer Slide Time: 09:57)



They have some fairly complicated names for all these things but let us just look at β redexes. After that you have to structurally close all these in contexts. If M goes on a β reduction to N then L applied to M goes on a β reduction to L applied to N. Since I said that what looks like the operand to an operator even though that itself might be an expression is capable of being reduced. If that is capable of being reduced in one step to another expression N then the result of the application reduces to another application. Similarly, it is possible that the so called operator itself is capable of being reduced without actually applying it to an operand. If L itself can go in one β step to M, then the result of applying L to M moves in a single β step to N that is applied to it.

Lastly, of course if you have λ abstraction, the body of the λ abstraction could be quite complex and might be capable of being reduced. For example, it might contain another abstraction applied to some application. A λ term could itself be an application. If L could reduce in one step to M then the abstraction reduces further. The rules β 2 to β 4 actually just close the notion of reduction to take care of all syntactic contexts. The main rule is really β 1 which gives a one step reduction.

(Refer Slide Time: 12:36)



Now as in the case of our other operational semantics we could actually define many step β reductions. Church actually formalized all these also as rules. On any step β reduction which we have mostly taken for granted as a reflexive transitive closure of this, is really this and these 3 rules just give you the reflexive transitive closure.

For example; if L is capable of being reduced in one step to M then L is capable of being reduced in 0 or more steps to M. The reflexive closure just says that L can be reduced in 0 or more steps to L itself and the transitivity just says that if L can get reduced to M in 0 or more steps and M can get reduced to N in 0 or more steps then L can get reduced to N in 0 or more steps.

I will talk about weak reductions but loosely speaking a weak one step β reduction is something that does not have $\beta 4$. If you just consider the rules $\beta 1$ $\beta 2$ to $\beta 3$ then you are implicitly saying that you cannot do any reductions inside λ abstraction. Unless the abstraction gets applied to something and therefore X moves out, an inside application cannot be reduced.

We could also look at an equality generated by a β reduction. For example; you could take the symmetric transitive relation generated by the many step β reduction of which many of course include 0. What is the meaning generated? If L goes in 0 or more steps to M then $L =_\beta M$. Notice that this equality is not the same as syntactic identity. It is something that will contain syntactic identity because L can go in 0 or more steps to L itself and so $L =_\beta L$. It is weaker than syntactic identity and if $L =_\beta M$ then $M =_\beta L$ and if $L =_\beta M$ and $M =_\beta N$ then $L =_\beta N$. This is the equality relation generated by the many step β reduction and this equality is actually very much like the normal equality that we might think of in our algebraic computations.

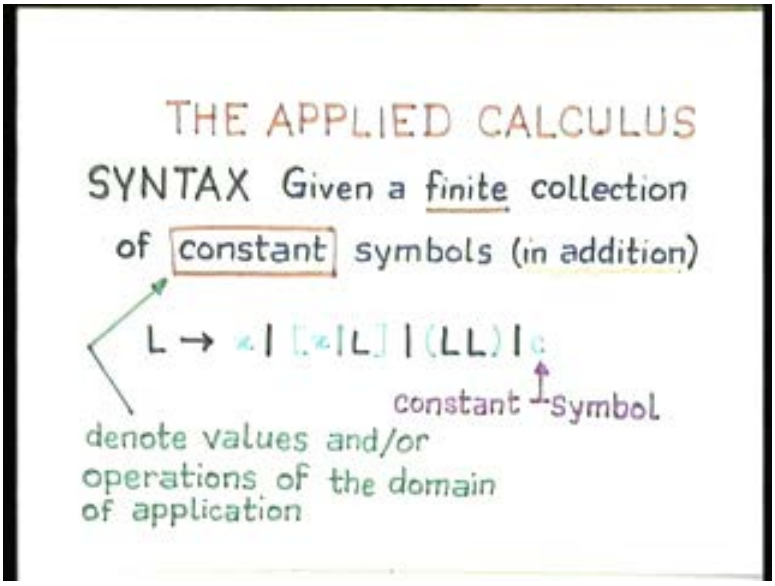
They are all motivated by similar notions of computation and considerations which come from algebra. If you take the simplification of a complicated algebraic expression you go through a process of reduction using some rules or some theorems that you already have and in each case the step that you get after a reduction is equal to the step that you had before. That is how you simplify and get a single value or a simple expression.

Symbolic computations also use the same thing. It does not matter whether they are values or symbols you still go through the same forms of reduction. All these concepts really come from your school algebra. Standard question after having studied $(a + b)^2 = a^2 + 2ab + b^2$ is: what is 99^2 , $(a-b)^2 = a^2-2ab+b^2$ and the question is: what is $(101)^2$; what is $(99)^2$ and then you go through a process of step by step reduction and that reduction has an equality that is already generated by the reflexive transitive closure of the notion of reduction.

Reduction is important and in simplification it is one directional. You write 99 as 100-1 which is actually an expansion. It is not really a reduction. You are actually expected to expand before you reduce in a different way. After all what is to prevent you from just multiplying 99 and 99 in a normal fashion in getting the square? But the whole point of that exercise was presumably to test whether you understood how to apply the formula $(a – b)^2$. So, you go through a process of expansion and then you go through reductions where you apply these formulas. The application of these formulas is very much like the β rules.

This is the pure calculus and the whole idea of having a pure calculus is that it should be applicable to any other discipline meaning it should be completely independent of and applicable to any other discipline which uses functions. Every functional programming language can actually be thought of as just an applied version of the λ calculus. Let us look at the applied λ calculus and see how the pure version is really distinct from the applied version.

(Refer Slide Time: 20:30)



As far as we are concerned from the stand point of the λ calculus itself, the applied calculus is really nothing more than the pure calculus with a new production which consists of a collection of finite constant symbols. Note the fact that you are doing all this from nothing meaning you do not have the distinction between values and functions, functions of functions and functions etc. These constant symbols could be either values of the underlying domain of the application or they could be operators on that underlying domain of the application.

If they are operators in the underlying domain of application then you also have your own reduction rules for that particular domain. This is how you would apply the λ calculus in general to any other domain which consists of values, functions or what ever it maybe. In general to any other kind of algebraic domain you can somehow convert domain into some form of an algebra which is not very difficult to do. These constant symbols are not just the values in the underlying domain. They include also the operators and the functions that you have predefined in that algebra which have their own forms of reductions.

Very often you have equations which do the reduction such as the distributive property on natural numbers. If you have something like A * B + A * C then there is a reduction step which is A * B + C or the other way could also be a reduction step. The algebraic equations give you two kinds of reductions where reduction is only a name. Sometimes the reduction can be an expansion but then it is also natural. You must have realized that it would have been necessary to expand something before reducing it. Reduction is a general term to denote some goal oriented activity towards some simplifying form which cannot be simplified any further. In the process the actual strings may actually expand. If they get you to your goal then it is a reduction. What are meanings? Let us look at an application.

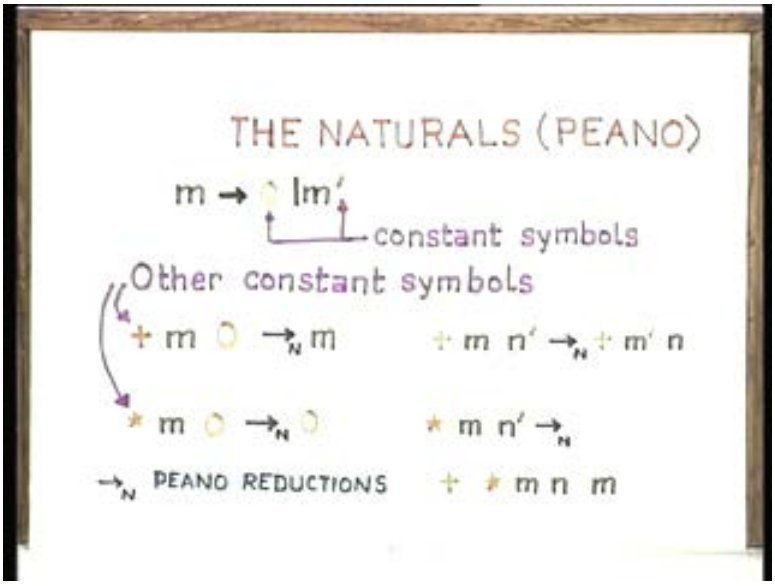
What is the notion of a meaning now? In any expression language the meaning of an expression is the value that it somehow reduces to. You have some complicated arithmetic expression then you would say that the value it reduces to is the meaning.

Now let us look at an application. I will tell you about the meanings of the λ terms also but before that let us look at an application and the simple application that I have in mind is the Peano arithmetic. If you look at the naturals there are an infinite number of symbols and we do not want to deal with infinite symbols. We will simplify the naturals into a grammar. Let us use only two symbols. Whatever is in light brown or ocar is the symbol of the natural. The natural numbers being completely down to earth are brown in color. So, 0 is a natural number and if m is a natural number then m prime which actually denotes the successor of m is also a natural number. This is what Peano said at the turn of the century and we have no reason to doubt him.

The two constant symbols here are 0 and a prime. You can look upon it as either an operator like the successor operator or more specifically in the case of a language you can look upon it as a constructor which allows you to construct arbitrary elements of a language. The natural numbers here are just a language. An arbitrary natural number is going to be 0 followed by a number of prime symbols. I am using a post fix notation which is quite usual and this whole process is tedious. You might define two more constant symbols. These two constant symbols might be addition and multiplication. The moment you define these two constant symbols (and I am using a prefix notation here) you have a 1 step n reduction rule on the Peano arithmetic. Remember all this is completely different from the pure λ calculus. It has got no relationship at all. It has got a relationship to a reality of counting but that is about it.

You define these constant symbols by means of reduction rules. I require two reduction rules for let us say addition and one just says that the sum of m and 0 is m and the other says that the sum of m and the successor of n is the sum of the successor of m and n. It is actually a reduction rule. It looks like it neither expands nor reduces but actually it is a reduction rule.

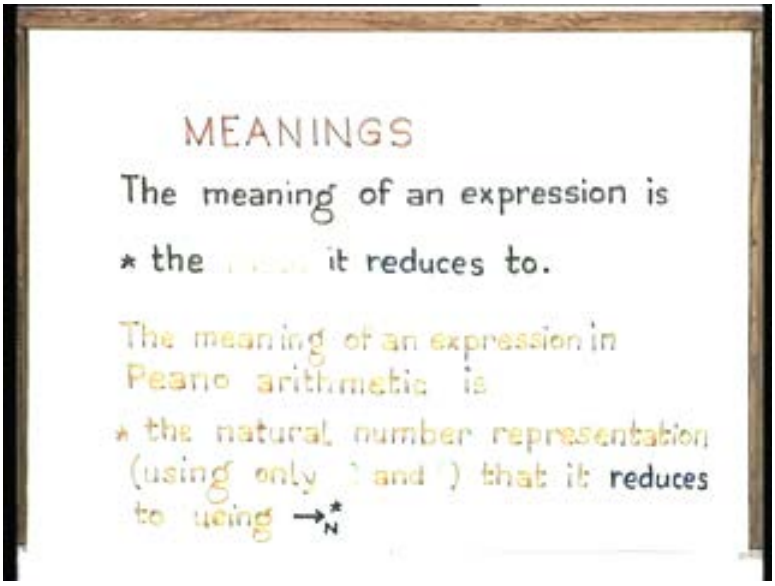
(Refer Slide Time: 27:40)



The reason is that eventually you will get all the primes from here to here and you will get a 0 there and then you will get a value. Similarly, the product of m and 0 is 0 and the product of m and successor of n' is just the sum of the product of m and n and m. It looks like an expansive rule but it is actually a reduction rule.

Just as we have one step beta reductions we have one step Peano reductions. Remember you are starting from the void where there is nothing and you have defined the naturals out of sheer genius. Now the question is when you have nothing what is the meaning of an expression in Peano arithmetic? From nothing you have defined the language of Peano arithmetic and now it is a language so it needs to have a meaning. The question is what is the meaning of Peano arithmetic assuming that there is nothing else in the universe? The only alternative is that the meaning of Peano arithmetic has to be found within peano arithmetic itself. This is in fact what we do in our school. The meaning of an expression in Peano arithmetic is just the representation of a number using only the two constructors in the language.

(Refer Slide Time: 29:00)



You do this one step Peano reduction. Peano reduction is not a standard name; I have just invented it. But the whole point is that you do those reductions defined till you have a string which is a string of the original language of peano arithmetic. This means that it can only consist of the constant symbols 0 and prime according to the rules of the grammar. When you have nothing and you are forced to give a meaning you declare that the meaning of a complicated expression in Peano arithmetic which has possibly + and * also in it, is just what it reduces to eventually, till no further reductions are possible. The only reductions rules you have are + and *.

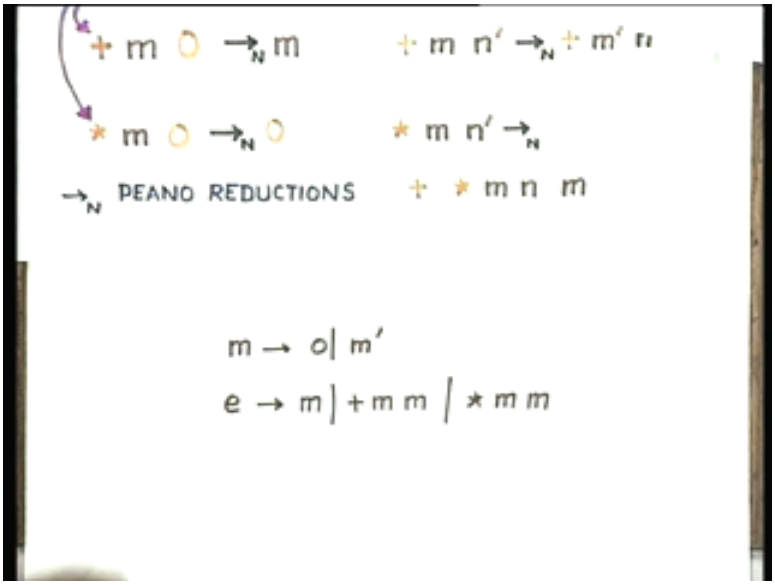
When no further reductions are possible it means that you should have an element of this language which means 0 followed by some primes. That is the notion of the meaning. Similarly in the λ calculus again we are starting from a void. The meaning of a pure λ term is a λ term that contains no more β redexes. Assume that you have got some λ terms. If you take a β term which is not itself a β redex and does not contain any λ redexes in it then what you declare is that its meaning is itself and there is no point going further for a meaning.

So, the meaning of a pure λ term is just a λ term obtained after sufficient number of λ reductions such that it contains no more β redexes anywhere. If it is a string that contains absolutely no more β redexes it means that it contains no more applications. This is a β redex and if it contains no more sub terms then you just claim that you have reached the absolute end. There is no more β reduction possible and that is the meaning of the original λ term. During the process of reduction of course you have also generated the equality relation. So, the final λ term that you get is β equal to the original λ term that you started out with.

Similarly, take an applied λ term. Let us assume that we are taking an applied λ calculus in which the constant symbols are the symbols of Peano arithmetic. That means we have taken the language of λ and we have taken the language of Peano arithmetic. In the syntax of the applied λ calculus you can replace those constants essentially by all the possible expressions of the Peano arithmetic. You can append the two languages so that they can intermingle. Let us take these notions of reductions quite seriously. Just to get you familiar with the notion of reductions in Peano arithmetic let us do a simple example using the rules that we have. Here is a simple example. I am giving an interpretation of this expression on the right hand side.

Essentially assume that you have to calculate $2 * 2 + 1$. Remember that the moment you add those two new constants you are also extending the language by expressions involving those two constants. You are essentially saying that the new language after having added these constants is of the form $m \rightarrow o \mid m'$.

(Refer Slide Time: 35:40)

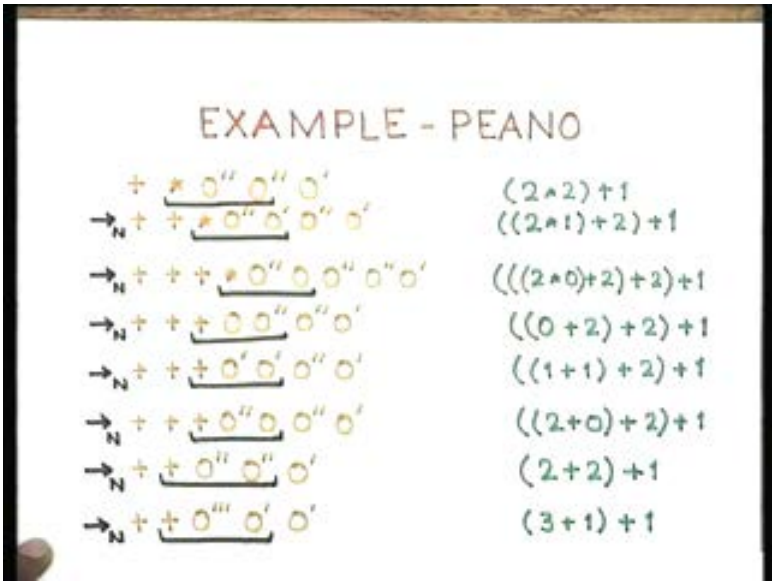


Firstly, you have m. Secondly, after having added these two constants you have an expression language $e \rightarrow m \mid +mm \mid *mm$ which actually allows any member of this original language and all expressions of this form. Now you add this expression language also to the pure lambda calculus and then you get an applied λ calculus. In my original syntax of the applied λ calculus I actually did not stick to this format. I just gave an extra production for 'c'. You could replace that c by e. The reason for using that c is rather pedantic but actually that was correct. But let us replace that production by e in the applied λ calculus syntax where e is an expression of the application.

Let us look at a pure Peano arithmetic example. If you have something like this then by the reduction rule for multiplication whatever is underlined in black is the redex in Peano arithmetic. It is capable of being reduced and we are following a prefix notation.

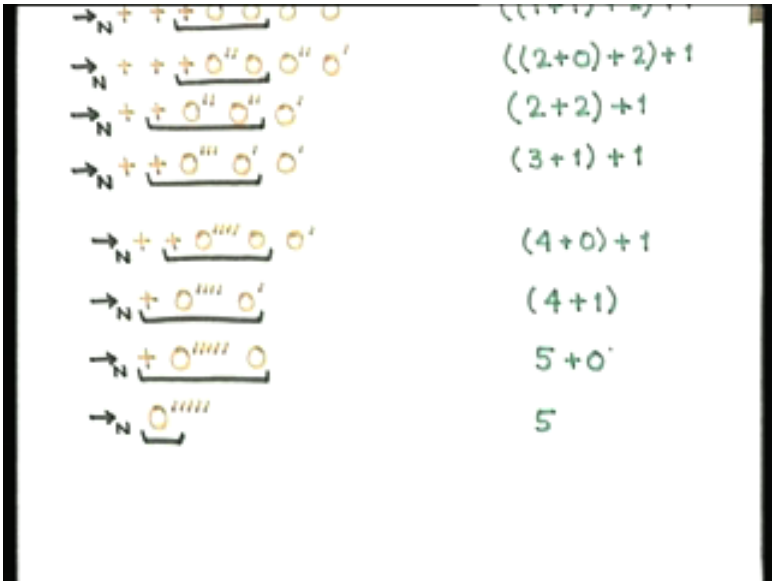
I do not want to go into specifying the orders of computation because it is implicit but we will assume that it is a normal prefix form and so the computation will go in from the innermost operator first to the outermost operator. This is a redex in Peano arithmetic and so this gives me this push. Remember the *mn' = + *m n m again. So, $*mn' \rightarrow +*mnm$.
This is how this reduction goes which is equivalent to actually saying that
$2 \times 2 = 2 \times 1 + 2$.

(Refer Slide Time: 39:45)



You can follow the right hand side to see the meaning. Again applying the same rule you get that $2 \times 1 = 2 \times 0 + 2$ and 2×0 by the basis rule and it is going to give me 0. This reduces to this 0 and then this is β redex. I use the addition rule and the addition rule says that despite whatever m might be if n is a successor of something if this is a successor of n then you take the addition of the successor of m with n. This Peano reduction gives me the successor of 0 and essentially the predecessor of this and a further peano reduction gives me this and then of course this is the base case of the addition operation so it gives me just this + m 0 which reduces to m. This new + here is a peano reduction which gives me this and then going on in that fashion I get this. For example; this β reduction yields this which is equivalent to essentially 4 + 0 + 1 and then this reduction actually is this itself and so now I tackle this reduction which is this which is again the basis and it yields this.

(Refer Slide Time: 41:10)



After a desperate number of computations we have actually decided that $2 \times 2 + 1 = 5$. But the whole point is that they are all symbolic reductions. Let us look at an example of a reduction in the pure λ calculus. Remember that having defined the language of Peano arithmetic and the expressions in the language we have just applied the reduction rules in a pure syntactic substitution. It is pure symbol substitution according to the rules of reduction and that is really what Church considered the basis of all computation; function application and reduction.

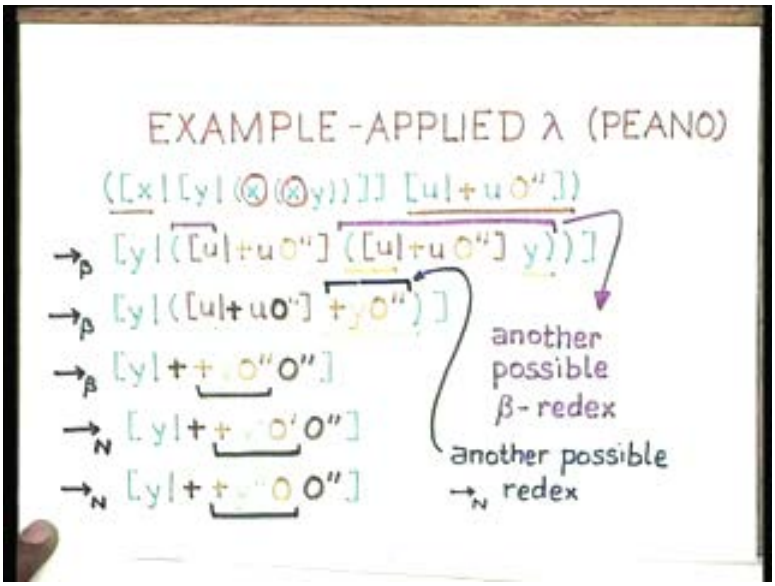
Let us look at this pure λ term. Now you will understand why I use the blue color for the syntax of the λ calculus. You could interpret Peano arithmetic very easily but we are not going to be able to interpret this very easily. I have for your convenience marked out the relevant portion. This term is λ x λ y λ z. This term here is x applied to z which is enclosed in parenthesis and whatever this object might be is applied to this object which is y applied to z itself and since it is an application I have enclosed it in parenthesis and then I close all the brackets. It is a nested scoping where the body itself is an application of essentially unspecified symbols. It is an application applied to another application. The body itself is an application applied to another application and now this whole thing is λ abstraction. The whole thing is applied to this λ term which is just something very simple for convenience. It is a λ u λ v u itself.

How do you do a β reduction? You scan this string from left to right till you encounter consecutive occurrences of open parenthesis and left square bracket. Then you know you have a β reduction if possible. Go to the matching square bracket and look for the operand after that and the closing parenthesis. Now your β reduction says that once you have identified what is the bound variable here and the term that is going to act as the operand you take this entire body and replace all free occurrences of this x in this body. If you look at this body in isolation then x is actually not bound any more; it is free.

Replace all free occurrences of x by this term. There is a single free occurrence of x here and that x is going to be throughout replaced by this. What you see in green is the effect of the substitution. I have replaced this x by this entire operand. Then this application of x to z looked rather abstruse. This x is not necessarily λ abstraction. You cannot expect to get anywhere with an application unless there is λ abstraction as an operator.

But now this substitution has created a new β redex by itself. Scanning from left to right in fact you get this β redex. I have marked in red so what has happened now is that this application is actually going to be over this λ abstraction. All free occurrences of u are going to be systematically replaced by z and this pair of parenthesis goes away; then this abstraction also goes away and that yields this.

(Refer Slide Time: 49:14)



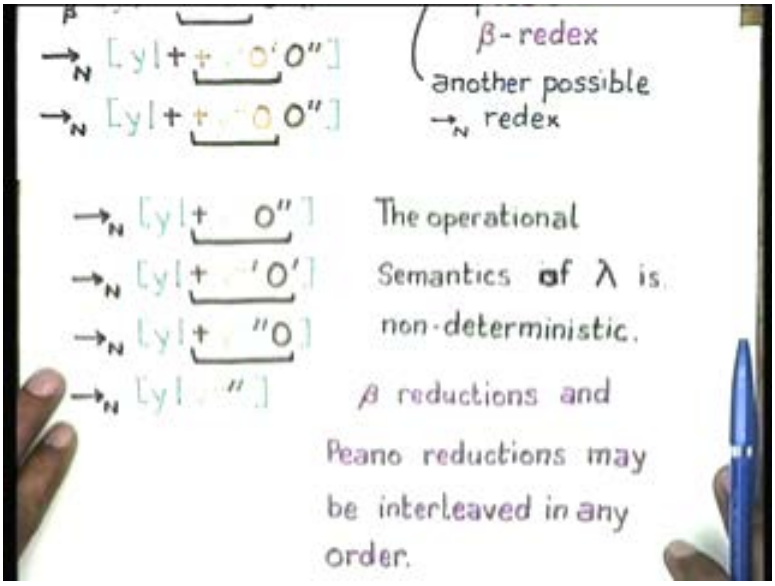
Now this was even more abstruse because you are performing an application here and now you automatically have that this term y applied to z is actually an operand of this λ abstraction and therefore is capable of being β redex and therefore this body is capable of being reduced.

You take this β redex; replace all free occurrences of v within the scope of v by y applied to z. There are no free occurrences of v here and so the result of this application is just that I get back z. This again is purely symbolic.

Suppose I mixed λ calculus with Peano arithmetic then what do we get? I have chosen to apply the β rules first so I have this redex and I follow the usual practice and there are two free occurrences of x here so I replace them by this entire term and I get this rather colourful object. I have chosen this orange redex here and I replace all free occurrences of y of u within this by y and I get this.

Now of course instead of trying to find a β redex I could have even done a Peano reduction here but I had decided not to do it. I could have equally well chosen this reduction and could have tried to reduce it at some point. When I perform this β redex all free occurrences of u in this body are going to replaced by + y 0''. I get this and then now I have a lambda abstraction in which there are no more β redexes but there are Peano redexes.

(Refer Slide Time: 49:42)



I continue to do the Peano reductions as I have illustrated before and I finally get this mixed λ Peano term, a symbolic term. Remember that y was not specified anywhere. I just say that the result of this is λ y, the 4th successor of y. But I could have chosen these alternate reductions any time I wanted; I could have intermingled β reductions with Peano reductions and you would have got the same result.