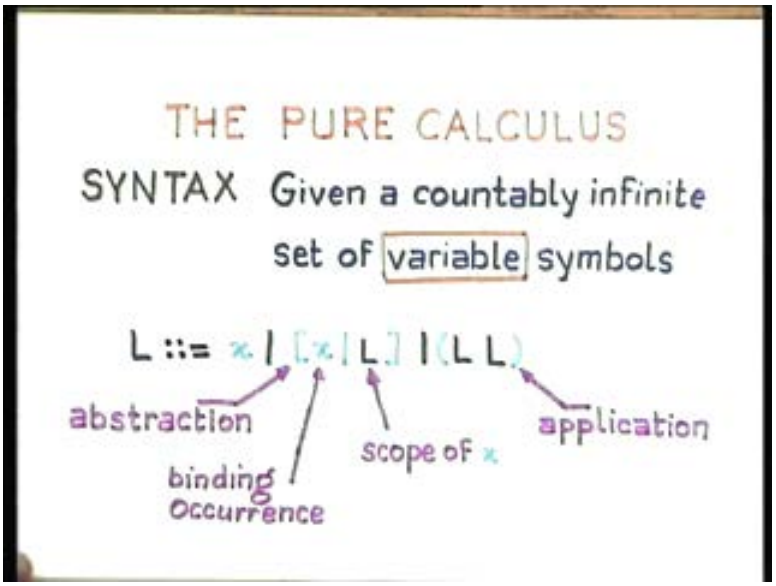
Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi

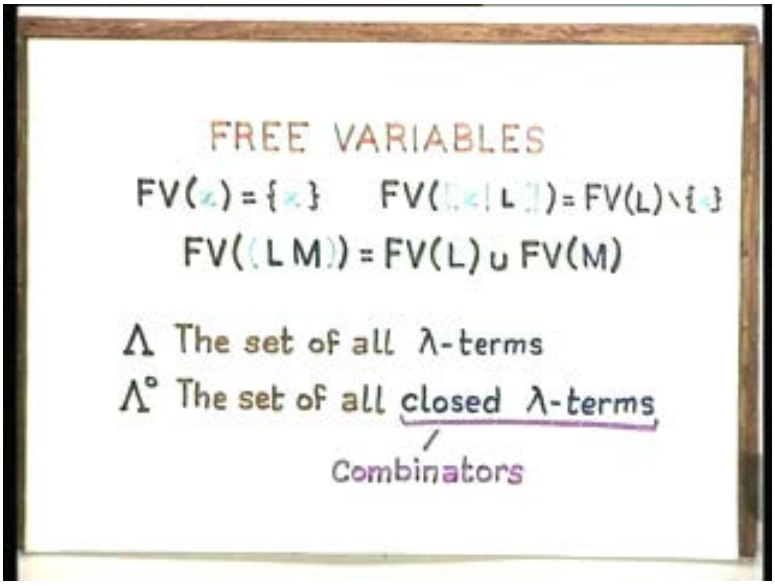
Lecture no 27
Lecture Title: λ α β : The Trinity

We will continue with what we did last time. I will just recap the pure λ calculus. Here is the syntax for the pure λ calculus and in the case of the applied λ calculus you just add a constant. This is the standard definition of free variables, the notion of a combinator or a closed λ term, which is according to whatever we know about local declarations, all equally applicable. Here is the operational semantics for β reductions.

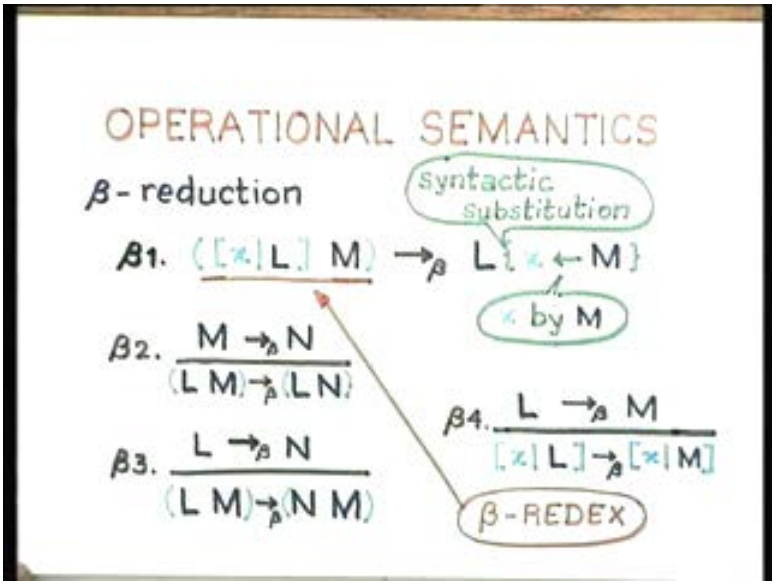
(Refer Slide Time: 00:44)



(Refer Slide Time: 01:03)

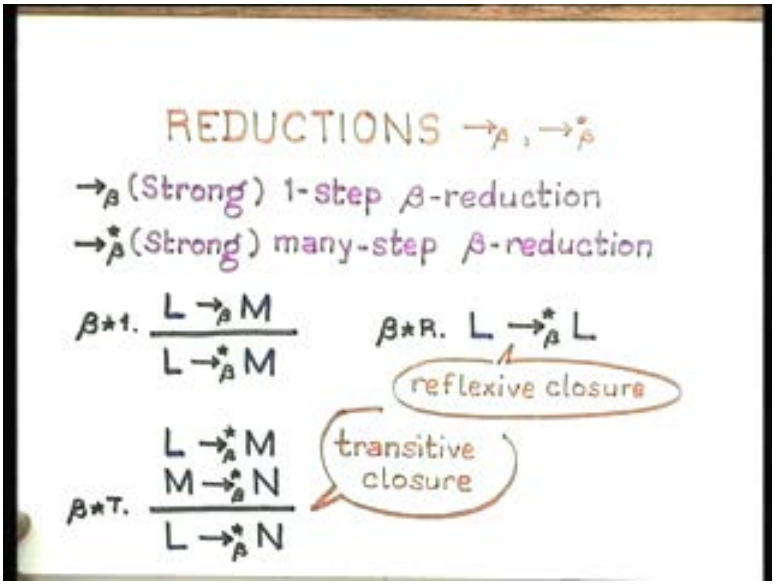


(Refer Slide Time: 01:44)



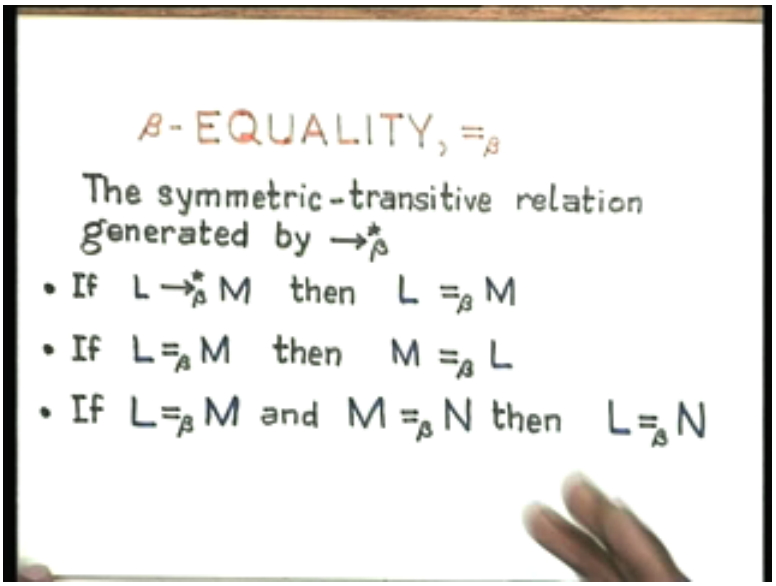
What is important to realize is this syntactic substitution. We have these usual rules. This is the main rule. Note that this notion is not part of the λ calculus. It is the syntactic act of replacing one pattern by another. It is what might be called a meta-syntactic operation. Then of course we close these rules in various contexts including the abstraction. If you do not have the abstraction then it becomes a weak β reduction.

(Refer Slide Time: 02:15)

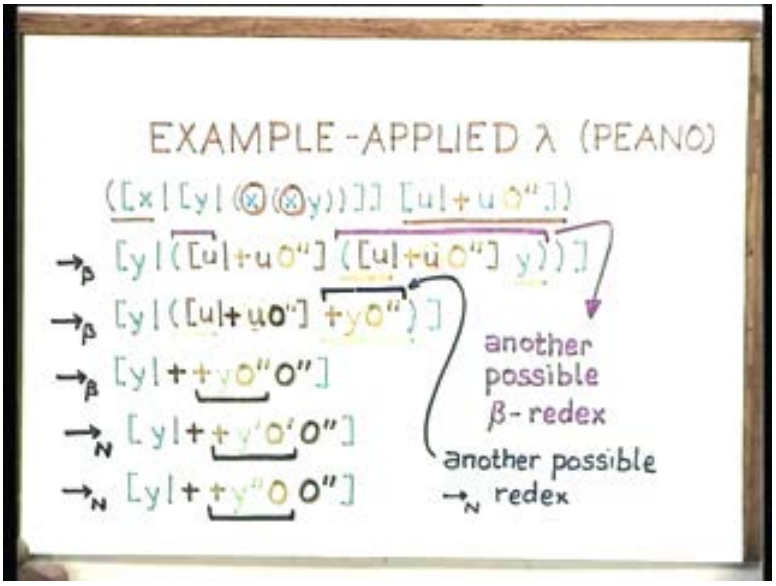


Such a term which is an application of an abstraction to a λ term is called a β redex and you can generalize these two. You can take transitive closures of β redexes or β reductions and there by many step β reductions that are either strong or weak. You can also take the equality that is generated by a β reduction by many step β reductions and you will get an equivalence reduction. Since the β reduction rules are closed under context this equality is actually a congruence relation. Then here is an example of an applied λ reduction using Peano arithmetic. As I said we could equally well have chosen other possible β redexes. There could have been several β redexes. We could have mixed β reductions with Peano reductions in any order.

(Refer Slide Time: 02:26)



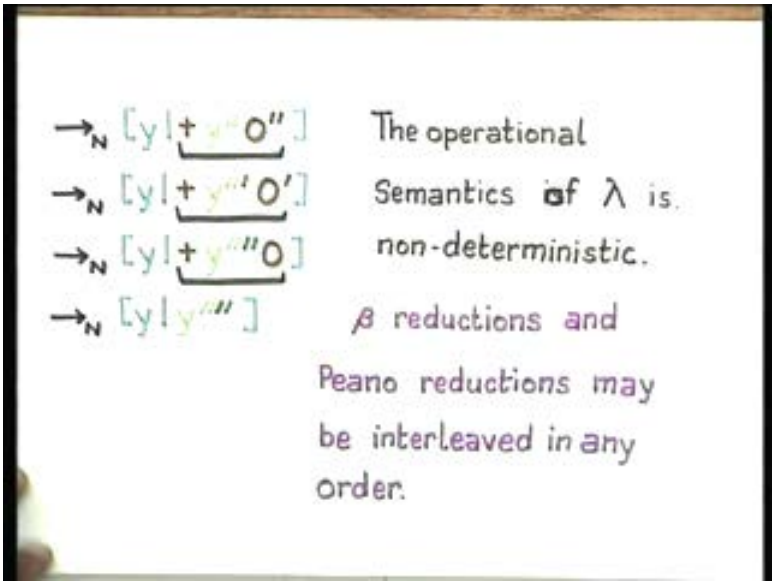
(Refer Slide Time: 02:50)



But even if you were to take the pure λ calculus what it means is that firstly the β calculus by its very semantics inherently possesses non deterministic execution behavior. There are other possible execution sequences for the same λ term especially when you have two different β redexes. There is no hard and fast rule, if you had an executor, which way it would take. Unless you determinise the executor and take up scheduling policy it is essentially non deterministic and when you couple it in an applied λ calculus with non determinism from the applied domain it could become even more non deterministic.

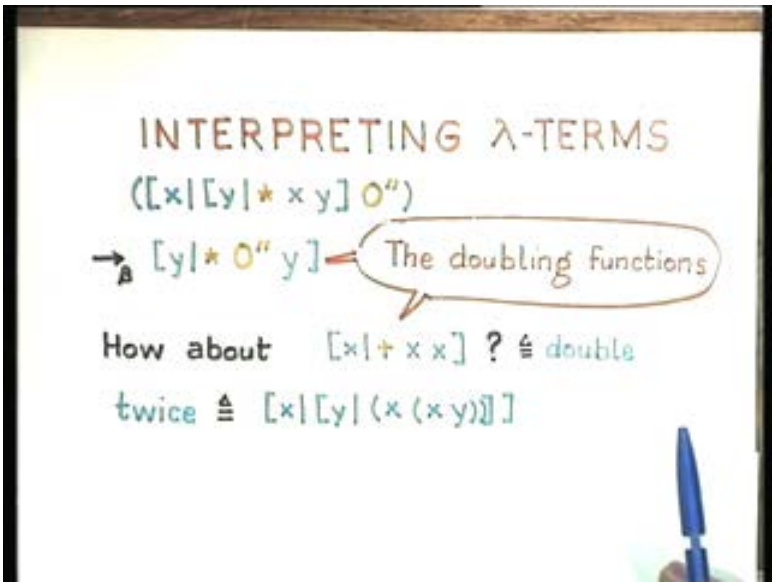
If you look at reductions in the Peano arithmetic they are purely deterministic. As far as the basic computations are concerned it is not yet clear what exactly this λ calculus is about. Let us look at how we can interpret λ terms which is essentially the original agenda which church had set out to himself. Let us look at this λ term. It is a mixed λ term with Peano terms in it. This says λ x λ y * x y. This * x y is clearly a Peano term which denotes multiplication.

(Refer Slide Time: 03:58)



Remember that in a normal multiplication which is also what is there in Peano arithmetic the multiplication is a binary operation which takes two arguments. Here this λ abstraction makes it a function of a single argument where the arguments are taken in sequence. The effect of essentially applying this λ term to the natural number 2 is to create a new function which is the doubling function. Given any natural number, it would double it. That means this x will be substituted by this 0 double prime and you get the doubling function. Of course you could define the doubling function independently itself in some such fashion but the point is that this is what is known as a curried form of the multiplication operation.

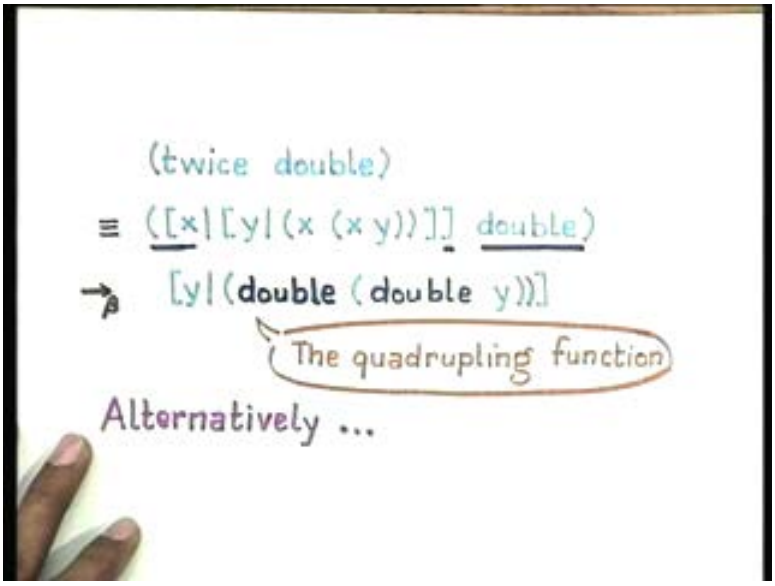
(Refer Slide Time: 07:22)



We will get into currying in some detail a little later but the effect is that the λ calculus treats every function as a unary function and the way to program binary functions or enary functions is to allow for a sequence of abstractions so that the arguments can be taken one at a time. This is essentially a binary function which requires two arguments before you can get a value but those two arguments are taken in order one at a time. So, you first apply to the first argument and get another function which when applied to another argument will give you a value. This function is obviously a unary function and so is this. There is no problem about that.

If you look back at this previous example we could actually give a name to this function. I am taking this λ term and I am calling it twice. It is λx λy x applied to x applied to y. What does this twice actually mean? We have nothing except just variable symbols and we do not know whether those variables symbols denote values or functions or higher order functions etc. Take this twice and apply it to a doubling function or one of the two doubling functions. I have used the word double itself to denote this function. When you apply twice to double you follow the normal β reduction rules and double is applied to double y and this is the quadrupling function which makes perfect sense.

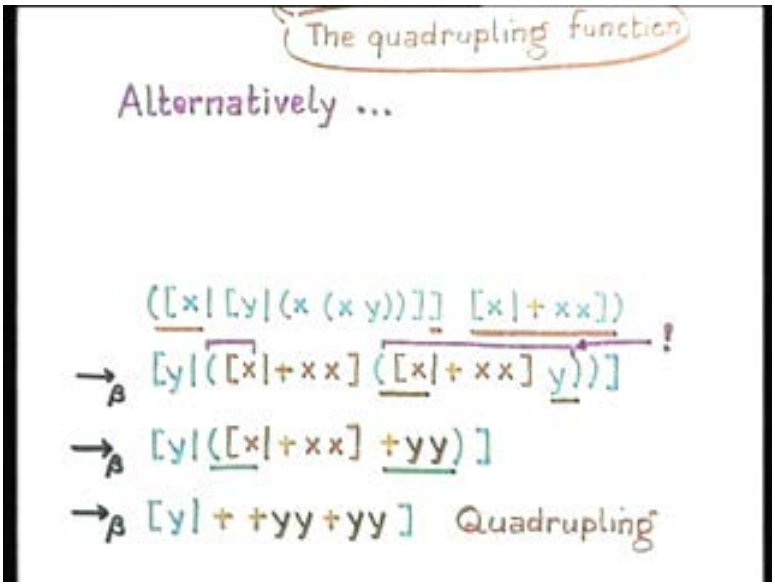
(Refer Slide Time: 09:00)



This is of course a simplistic way of looking at things. You could actually take the expansions out. Remember that one of the purposes of the λ calculus is to treat unnamed functions. It is to look at functions themselves as objects worthy of treatment. If you were to actually do this application you would actually get a non deterministic choice between two possible β redexes. This is a twice applied to the double and the effect of that is to replace all these blue Xs by this term. That is what comes out into red and of course this Y is here. Note that this is an application that is an open parenthesis followed by an open square bracket. So, I have the possibility of replacing each of these Xs by this entire application or I am taking an easier way out to avoid confusion. I am first doing this application. I am leaving this term as it is and I get this simplified

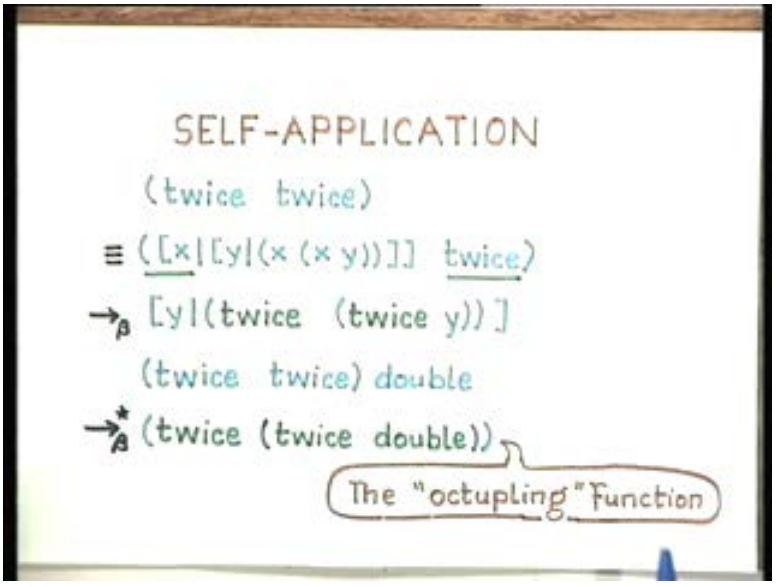
term and then when I perform this β reduction I get this. This is a quadrupling function which is perfectly understandable and actually if you look at this definition of double it really does not matter what you put here. If you did not put double you could have put something else for example, you could have put the squaring function.

(Refer Slide Time: 09:22)



Twice of square for example should give you just a square of the square and the square is very easily defined as a unary function and I could just get the square of the square. We could actually put any function in place of double or square and apply twice to it. In particular what I could think of doing and which Church did unlike most mathematicians is to apply twice to twice.

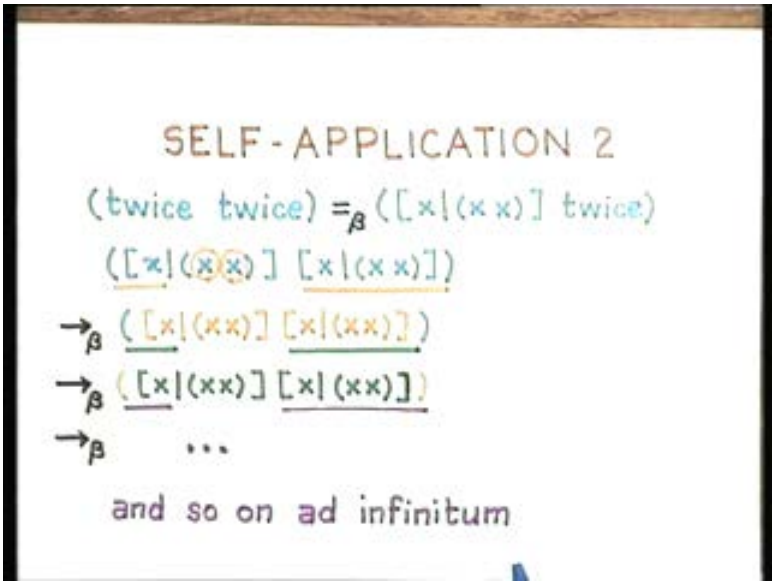
(Refer Slide Time: 12:15)



This is normally not found in mathematics but if you are talking of a purely symbolic method of computation and functions we should allow for such a possibility and see what you get. When you apply twice to twice you get twice of twice applied to twice applied to y. This is just perfectly reasonable. In fact if you were to apply it to double again you would get twice of double and twice of that which is the octupling function. This is my name. This is the function which goes 8 times and it is a perfectly reasonable function.

Then I could look at this twice and I could actually look upon this application of twice to twice as actually λ abstraction applied to twice. This is the particular λ abstraction for which if you do a β reduction, each of these X's will be replaced by twice and of course the abstraction x will go away and you will get twice applied to twice. This means that there is a way of getting this twice applied to twice by another λ application. Now this becomes an object of interest in itself. I could apply this function to itself. I apply it to itself and each of these X's is going to be replaced by this term and I get this, which is exactly identical to the previous term except for the colours. When you do this application again you get exactly the same term.

(Refer Slide Time: 14:13)

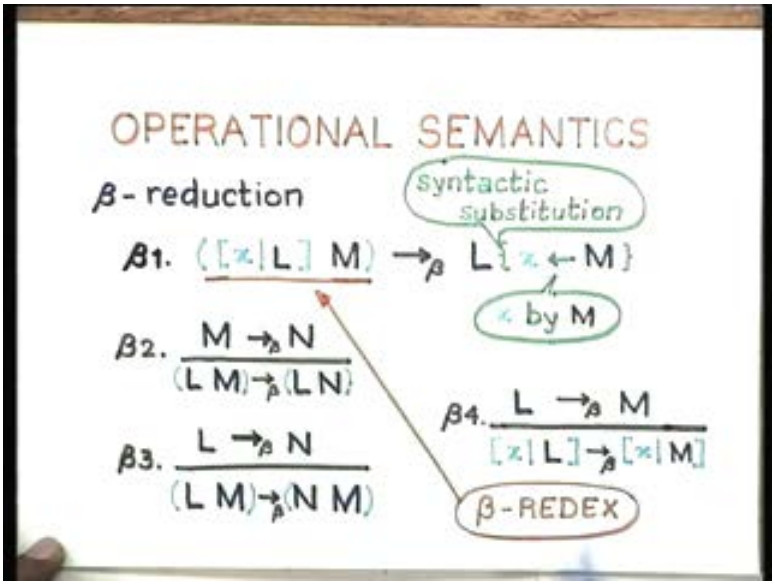


This was an example of a non terminating computation. Before going into what Church said we need to know what an algorithm is. An algorithm is an abstract object which is totally meaningless as far as I am concerned. If you want to give a logical definition of an algorithm it is a λ term which has a terminating computation. The only concrete representation of an algorithm is of a λ term which has a terminating computation. You should make a distinction between this non-terminating computation and a term of this form itself. Take this λ term which is an algorithm in the sense that the computation is already terminated. There are no more β reductions possible but this λ term is not an algorithm because the β reductions are possible and it is an infinite computation.

If you look at it in more modern terms what church has constructed is a full fledged programming language. He claims that the notion of an algorithm though 2000 years old is not an accurate notion in any sense and it is only a notion that could be understood by intuition. The only concrete objects are not algorithms but programs and algorithms are simply programs that are guaranteed to terminate and there are programs that are not guaranteed to terminate which can have infinite computations. That essentially settles the issue of the notion of computation and the notion of algorithm. Gödel did not quite believe what church said but the point is that Church's work was subsequently complemented by Turing's work and the two formalisms were proved equivalent which you will learn in your theory of computation.

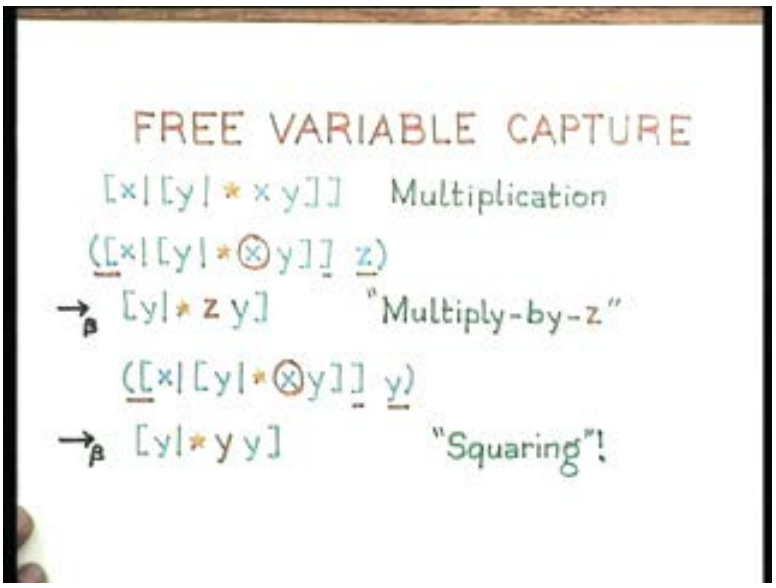
Let us get back to the β reduction which is most important. You have this syntactic substitution which is a replacement of a variable by another λ term. You could play with λ calculus. For example; I could take this λ term which really denotes multiplication operation in the applied λ calculus or mixed λ Peano calculus. Essentially I can look upon this λ term as a box that I acquired from somewhere. Typically you acquire a box like this from the US. You have this voltage problem and then you go to various shops and you get another box and you apply one box to the other and hope that it works.

(Refer Slide Time: 17:34)



Here I have another box called z and I apply them in the hope of getting something that works and actually it does. I get a new box which is the function multiplied by z whatever z might be. Since z is a product that is not very reliable after sometime it conks out. Having detected that it conks out I go to the local market and ask them to supply me a z but they say that we do not exactly have the z but we have something else which will work equally well and that is a y.

(Refer Slide Time: 20:20)



Now I plug them both together and I get a typical faulty product. I want it multiplied by something and I get squaring. That is the problem when you play around with incompatible toys. Church of course put it up more loftily and he said that essentially although this y that you acquired from the market looks identical to the y that is inside this box it is really different from the y that is inside this box and these two y's are actually different and they should not be treated as the same and that is where the problem is. Essentially this y is a free variable which presumably has a value in some context may be 110 volts but the moment you perform this β reduction this y gets captured by this locally declared y. So, a free variable gets captured even though it is not intended to be captured. This means that this notion of substitution in the operational semantics has to be very carefully worded up. You cannot just do substitutions like this. Take another possibility.

Assume I have actually a λ term of this form $[x|.....]\{x \leftarrow M\}$ where x should be replaced by M. This is not an application. It is a syntactic substitution and this x is not the same again as this x. This x is very highly local to this box and this x was supposed to be replaced with some free x that was outside this box. What if this whole thing was part of some larger λ term in which there was some reduction? Supposing I had this λ term; $([x|.....[x|.....]....]M$ then a beta reduction means that this x has to replaced throughout this term but if there are any local x's there they are different from this x. In the process of doing this replacement I might actually encounter this problem that I have to deal with a syntactic substitution of this form; $[x|....]\{x \leftarrow M\}$ and since this x in the right is different from the x in the left I should not do any replacement at all.

This is again another instance of a free variable capture except that this x is actually bound but it has a scope and this x is a whole in the scope of this x. To use our standard programming language terminology this scope is actually a whole in the scope of this x and as far as the body of this huge λ term is concerned the x is free and as far as just this body is concerned this x is free and it will get captured.

The first principle of doing successful computations on the λ calculus is not to allow free variable capture which means we have to define this notion of syntactic substitutions very carefully. Whenever we talk of substitutions it is clear from our β reduction rules that we are talking about only free variable substitutions and no other substitutions. Bound variables are not to be substituted; they get substituted in a two step kind of process. Bound variables are like parameters to a procedure where you apply a value and apply the function and you take the body of the λ term which has this bound variable x and then you do the replacement but then the x in the λ term is free. So, all substitutions that we will talk about are only free variable substitutions. All substitutions are going to be only free variable substitutions. You cannot replace bound variables because bound variables are logically distinct elements. Whatever you require to do in the sense of bound variable substitution is something that can be obtained through the normal process of λ application by first freeing the variable and then doing a free variable substitution.

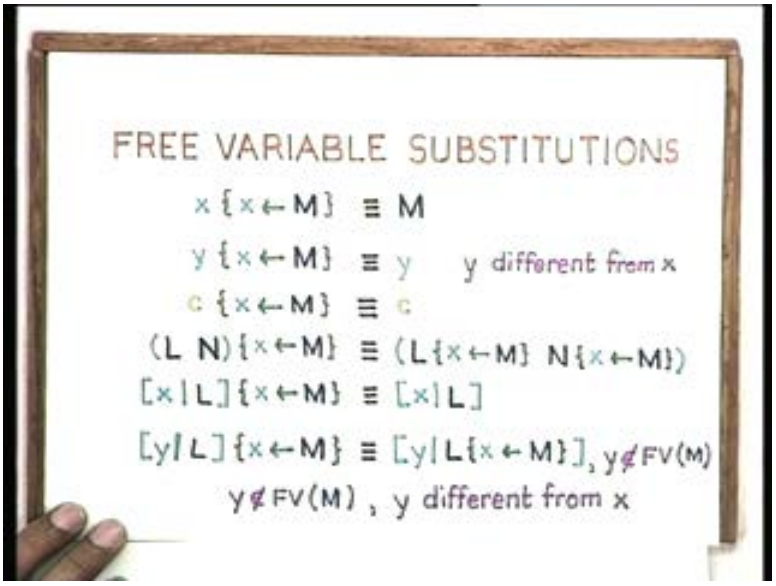
Let us define free variable substitutions in as accurate a manner as possible. This free variable substitution has had a very rocky history. Most books let me tell you are probably wrong. They have made a mistake somewhere or the other in the definition of these substitutions. So, do not believe most of these books but play around with it to see whether there is a bug in my definition. If I have a $[y \mid L]\{x \leftarrow M\} \equiv [y \mid L\{x \leftarrow M\}]$ term $x\{x \leftarrow M\} \equiv M$ then the effect of this syntactic replacement is just to give me M which is perfectly obvious. If I have the λ term $y\{x \leftarrow M\} \equiv y$ where y is different from x the effect of doing this syntactic replacement essentially means replacing all free occurrences of x in this term by M and there are no free occurrences of X in this term and therefore the result is just Y.

What if it is a constant in the case of an applied λ calculus? A constant is different from any variable you have by definition and the effect of this substitution is just to leave the constant unchanged. If it is an application of L to N then the effect of this substitution is to replace all free occurrences of X in this application by M and that works out to replacing all free occurrences of X in L by M and applying the resulting λ term to the term obtained by replacing all free occurrences of X by M in N. Now that is also obvious. Note that these are all structural definitions by some case analysis. Here you get an important set of case analysis. If I have $[x \mid L]\{x \leftarrow M\} \equiv [x \mid L]$ then this syntactic substitution clearly refers to an X which is different from this X and therefore there is no free X in this scope. There cannot be any free X in the scope L. The effect of this substitution is just to the leave the λ abstraction unchanged.

If on the other hand Y were different from L then the effect of this substitution is to replace all free occurrences of x in L by M. Moreover an added constraint here is that M itself should not contain any free variable that is it should not contain Y free. If M contains Y free then Y in M is different from this locally bound Y $[y \mid L]\{x \leftarrow M\} \equiv [y \mid L\{x \leftarrow M\}]$.

So, you can do this locally free variable substitution only if Y does not occur free in M.
If Y occurs ‘bound’ in M there is again no problem because that binding will be carried through in this substitution. Y has to be different from X and Y should not be free in M.

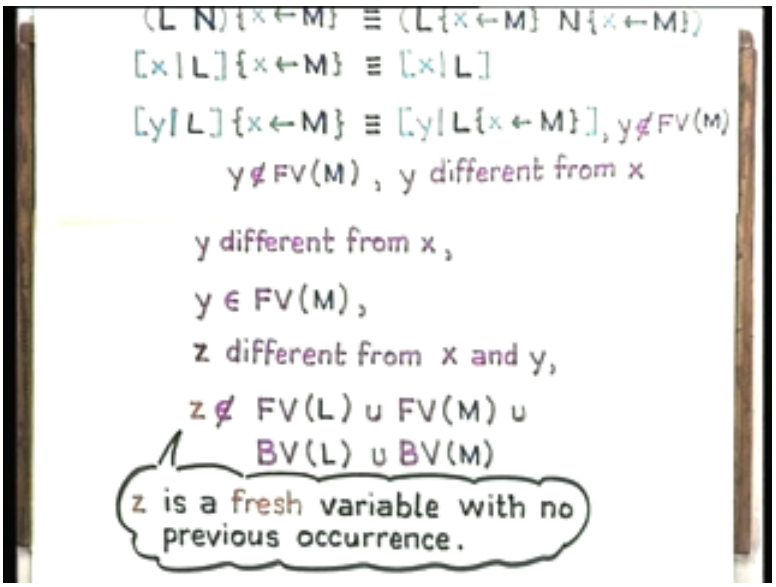
(Refer Slide Time: 30:40)



Supposing Y were occurring free in M what would you do? Take this λY L and let me lower this so that I can still retain this. This is a case very similar to the previous one except that Y does occur free in M but Y is different from X. Then I pick out a new fresh variable Z and then I systematically replace. Since Y is a bound variable here it probably does occur free in this term L.

I systematically replace all free occurrences of Y by Z. Note that if there are bound occurrences of Y then they do not have to be replaced. The same variable could occur both free and bound within a λ term and obviously they are two different objects.

(Refer Slide Time: 32:50)



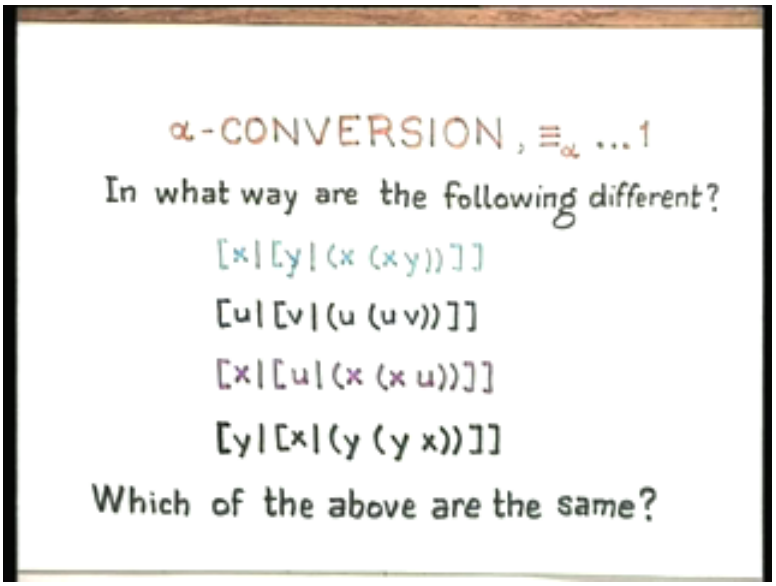
So, all free occurrences of Y in L are replaced by Z and I get a new term which is let us say L′ by this and now I do not have any problem. I replace all free occurrences of X in this new term by M and I can do this provided Z does not occur free in M. That is why I am saying take Z completely fresh. Freshness is not a virtue that many of these books have emphasized but it is the safest possibility. When in doubt pick out a fresh variable and there is no dearth of variables because we have covered our cracks by initially assuming a count-ably infinite set of variables available to us.

Any particular λ term is a finite syntactic object and so has only a finite number of variables; it cannot have an infinite set of variables. Therefore you are always guaranteed that there exists a fresh variable that does not occur anywhere. So, at least take a variable that ensures that it does not occur anywhere in the terms in which you are performing a substitution. Then first systematically replace this variable Y and this is the case when Y belongs to the free variables of M; systematically replace all free occurrences of Y by Z and then in the new term systematically replace all free occurrences of X by M. Note that X itself could occur either free or bound in M. These free variable substitutions are perfectly general if you follow the case analysis.

There is a problem and this is the mistake that is there in many books. The problem is that if Z is fresh there is no problem; if Z is not fresh but occurs bound somewhere deep inside the term and if Z occurs bound and you have Z occurring bound in M also and if I have some application like this deep inside then that will not affect but if this Z occurs free in M then there is going to be a problem. But the problem with these syntactic substitutions is that over 60 years it is very difficult to distinguish between a right definition and a wrong one and I am playing safe. Just take a new variable; do not allow for any possible confusion, just pick out a new variable that has never been used and do that replacement first. I am just playing safe. If you analyze some of the definitions you might find that some of them are really correct but you have to do a large amount of case analysis before you can prove them correct. Instead I prefer to take the safe way out because of such things. For example, in Ravi Shetty's book this bug is there; he just assumes that Z is not free in M. But in the act of replacement there might be confusions between two bound Zs. This confusion between two bound Zs should not occur. That is something that I suspect he has not taken care of. So, I would prefer always as a good pragmatic rule to pick out a fresh variable and do the replacement. In fact what I would advocate is that the moment you see two λ terms which have common bound variables I would suggest that you just replace them all by fresh variables and that is something all students know off-hand.

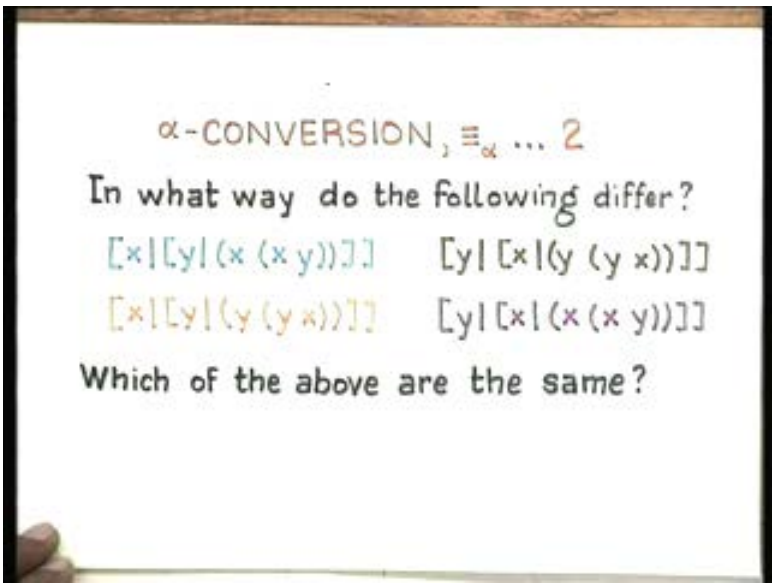
Look at these terms. In what way are the following different? I have taken the definition of twice and I have just done various kinds of replacements. Here is λX λY, X applied to XY which was the original definition of twice. I have systematically replaced all occurrences of X and not just free, meaning I have not performed the syntactic operation of substitution. I have just considered this as a string of symbols and I have just systematically replaced all occurrences of X whether free or bound by u and all occurrences of Y whether free or bound by v. In this case I have taken this and just replaced all occurrences of Y whether free or bound by u. In this case I have done a simultaneous replacement. With one hand I have replaced all occurrences of X by Y and with the other hand simultaneously, so that there is no confusion, I have replaced all occurrences of Y by X.

(Refer Slide Time: 39:55)



In what way are they all different? Actually they are not different and they are all the same. They are all obtained by what might be called the renaming of bound variables. One possibility is that you have this syntactic identity. These 4 terms are not syntactically identical but they are sufficiently identical in the sense that only the names of local variables are different otherwise the structure of the patterns is the same.

(Refer Slide Time: 40:44)



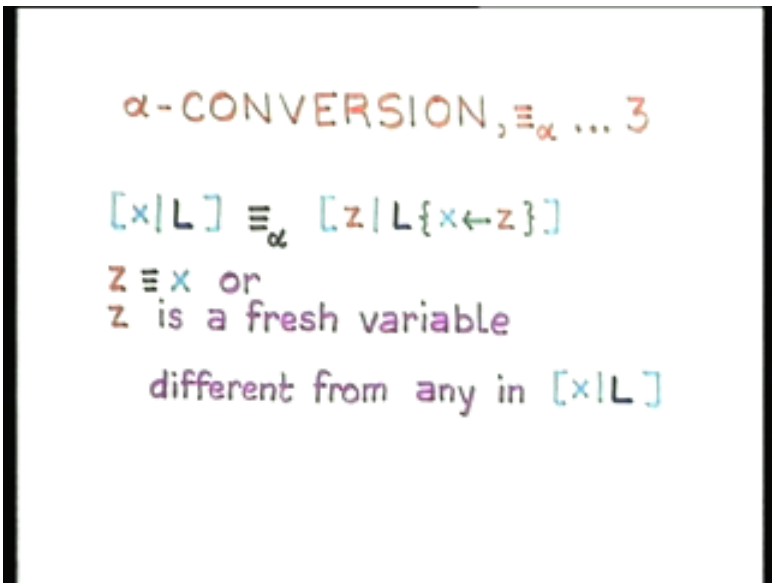
We would consider them almost identical and that is called α conversion. All these terms are really α convertible. My problem now is how I would express this α conversion in terms of our notion of syntactic substitutions which we have rigorously defined. For example, how do I get

this fourth term from the first term? It is actually very simple. In 2 steps I get the second term from the first term by two syntactic replacements using α conversion and then in another two steps I get the fourth term from the second term.

We have to define α conversion rigorously too. The α conversion just says that given a term of the form λ X L it is α equivalent meaning it is almost syntactically equivalent to a term in which there is a bound variable set and all free occurrences of X in L are replaced by Z where of course we should ensure that if 2 terms are syntactically identical then they are also α equivalent which is trivial. This Z is either the same as X or again I am playing safe; Z is a fresh variable different from anything in this term and this is the safest game to play and not get mislead by most of these books which only concentrate on free variables. It is possible to construct very often perverse and convoluted examples in which some of the definitions in most of these books will be proved wrong. It is also very hard to construct examples to show that they are wrong. But what is safe is always to pick a brand new variable preferably shining and do the replacement.
These are the basics of the λ calculus which I hope you will spend some time toying around with.

What does a calculus mean? I mean why is Pascal not a calculus as this language of α terms is? Why is it a calculus? What is a calculus? What is a calculus about propositional logic? Why is it called propositional calculus? Why is it called predicate calculus? But in the propositional calculus and the predicate calculus there are no computations unless you impose computations somehow. A calculus is a system with a formal language that is defined by a context-free grammar whose axioms and rules of inference are such that there exists an algorithm to decide whether a given application of an axiom or a rule of inference is correct.

(Refer Slide Time: 43:34)



A calculus is one in which regardless of meaning attributed to a formal language there exists purely syntactic and symbol pushing rules which allow you to perform inferences. That is in fact the property of the axiom systems of first order logic, for propositional logic, for Boolean

algebra and the λ calculus. The rules have to be purely syntactic and involve only symbol processing and by symbol processing usually I mean just pattern matching and replacement. That is what makes it a calculus.