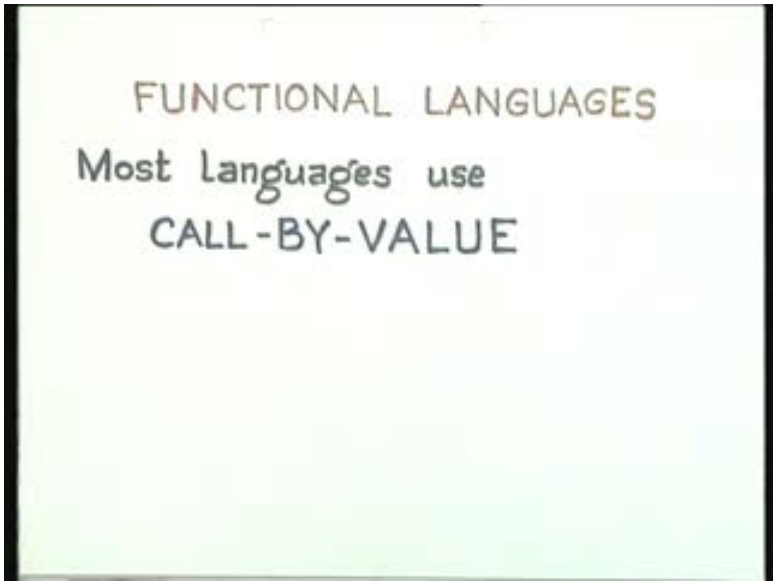
Principles of Programming Languages
Dr. S. Arun Kumar
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi
Lecture - 39
Parameters

Welcome to lecture 39. So today we will discuss parameter passing. Since most of the issues are pragmatic though it is really and it is also tremendously complicated to give semantics, what we will do is we will discuss most of the basic parameter passing mechanisms from a purely pragmatic view point. And it is possible to give semantics but then that is the semantics of an after thought and it is not a predefined semantics. So as a result especially for imperative languages it becomes very messy and rather complicated and actually does not provide too much intrusion at least in my opinion. But what it can do of course is to accurately pin down how it should be implemented, what are the considerations and so on. But however, we will look at it from a purely pragmatic point of view. It is most elegantly discussed only in a functional setting or in a lambda calculus setting with the toy language.

(Refer Slide Time: 1:36)



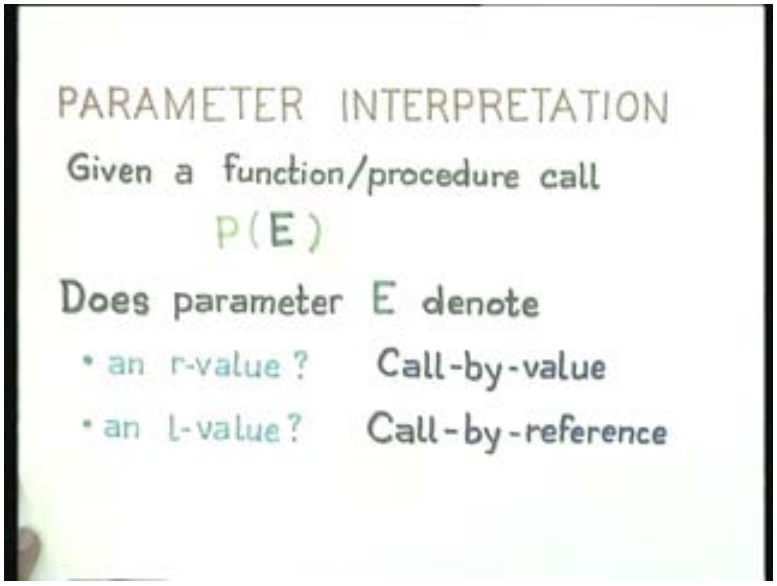
So as I said most functional languages use call-by-value which is that, given an operator and an operand you completely evaluate the operand before you actually perform the application. In the particular case of the typed applied lambda calculi which is what most functional languages are call-by-value and call-by-name give the same results but call-by-value is likely to give you a faster convergence to the normal form. In the untyped case call-by-name is always guaranteed to give you a normal form if it exists but call-by-value might go off into an infinite computation. So, that is one important reason why you should introduce typing then you are guaranteed that you will always get normal forms.

Secondly from a purely pragmatic view point you can use call-by-value which is an easy implementation compared to a call-by-name implementation.

Anyway coming to procedural languages or imperative languages the whole question of parameter passing is closely related to what might be called an attitudinal problem. so essentially given a function or a procedure call the question is, and this E (Refer Slide Time: 3:15) in general could be an expression which is shown in dark green but very often it is just a single name of a variable or of an array component or record field. So very often it is just a name of some object. And in imperative languages if you remember objects could have even simple variables actually have different interpretations for example depending on whether they occur on the left hand side of an assignment or on the right hand side of an assignment.

Assignment in some form of the other is the most important construct in an imperative language and a name has a different meaning depending upon whether it occurs in the left hand side or at the right hand side of an assignment. So when it occurs in the left hand side you are essentially referring to its location or its address and a variable is really like a container which can contain a value and if it occurs in the right hand side you are really looking at not the location but the value contained in a particular location. So I made this distinction between l-values and r-values long ago and essentially that confusion (Refer Slide Time: 4:46) resides here.

(Refer Slide Time: 5:13)



So the attitude or the interpretation that you can give in an expression if it is a simple arithmetic expression or some such thing there is really nothing you can do. Actually there is still confusion there too as we will see. So the attitude that you can take is if the parameter E denotes an r-value then what you mean is that E is really a value and therefore the value is what has to be passed as the actual parameter of this procedure call.

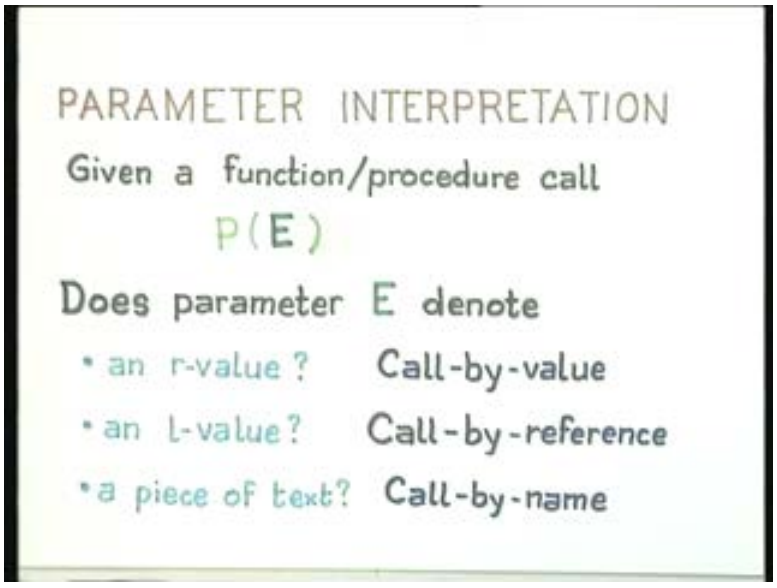
And if you regard this expression as denoting a value then essentially you will be using a call-by-value mechanism.

On the other hand you could regard this expression as an l-value which means that it is it should be an expression for which there is a possibility of that confusion between an r-value and an l-value. If it is just some arithmetic expression of the form 2 plus 3 then clearly it has no l-value where it could be a simple variable or an array component or a record field. In such cases what happens is that then that expression is open to two different interpretations either as the value contained in a location or the location itself.

Then here what you are saying is should I pass the location itself as the parameter?
After all if that expression could be interpreted as an l-value then it is just the name of a certain object so am I passing that object itself or am I passing the value contained inside that object?
So if I am passing that object itself then it is really an l-value I am looking at and what you get if you take that attitude is what is known as the call-by-reference mechanism. And there is actually a fair attitude. If it is some expression then it certainly has an r-value but it may or may not have an l-value but then can I say that when I am calling this procedure I am not passing either the l-value or the r-value but I am passing the expression itself as a parameter.

In large parts of mathematics actually this confusion never arose and that is because whether you pass the expression itself or you pas the value of the expression as a parameter you are always guaranteed the same results in mathematics and that really has to do with the fact that whether you do a call-by-value reduction of a lambda term or a leftmost outermost reduction of a lambda term in either case the results if they are not an infinite computation will yield the same normal form and that is the call-by-name implementation.

(Refer Slide Time: 8:41)



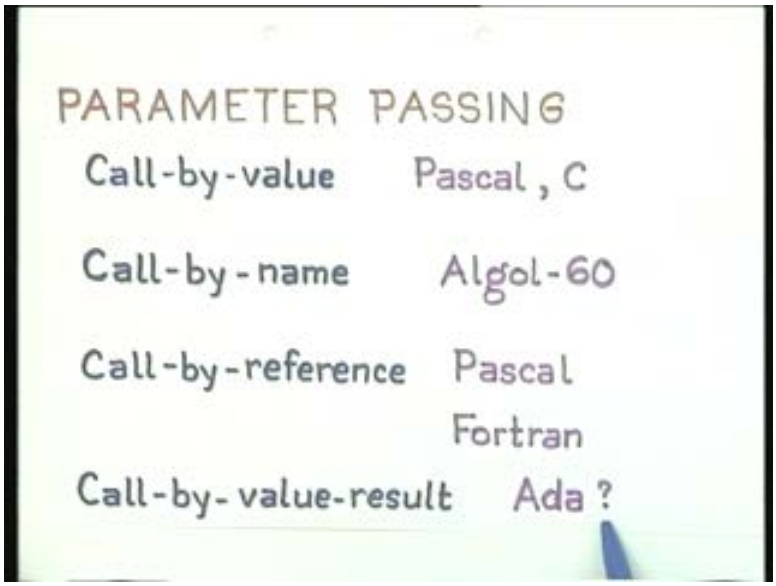
I can look upon the parameter itself as just a plane piece of text which is what I am interested in passing. I am not interested in passing its value, I am not interested in passing its location but I am really passing just a plane piece of text and this is really what happens in a beta reduction in the lambda calculus. And because the lambda calculus is such that especially the type lambda calculus regardless of whether you pass it as a value or you pass it as a piece of text you get the same normal forms.

Therefore in Mathematics there is no question of an l-value. Therefore the only confusion that could have been is when you do a function application whether you are passing a value or an expression regarded as a piece of text. But in the typed lambda calculi since it is guaranteed that you will always get normal forms and you will get the same normal form always it did not matter whether it is a call-by-value or a call-by-name implementation and there was no confusion.

However, in any programming language in which locations are also closely interwoven into the language what happens is that this call-by-value and call-by-name can produce different results. So the call-by-name or whatever is in the lambda calculus would be the leftmost outermost beta reduction rule. That means you scan a lambda term the moment you find a lambda application take the operator and the operand do not reduce the operand and starting from the left the outermost application has to be done first. Then in that case the operand itself might be capable of being reduced but you do not reduce it. And when you do the beta reduction what you are doing is substituting the entire text of the operand in place of a free variable in the body of a lambda abstraction that is a call-by-name method of passing.

These three are the most important for these reasons actually though a call-by-name is really not being used too much but you cannot really justify a lot of things especially in functional languages without actually proving for example that they will give you the same semantics as call-by-name and so on.

(Refer Slide Time: 12:41)



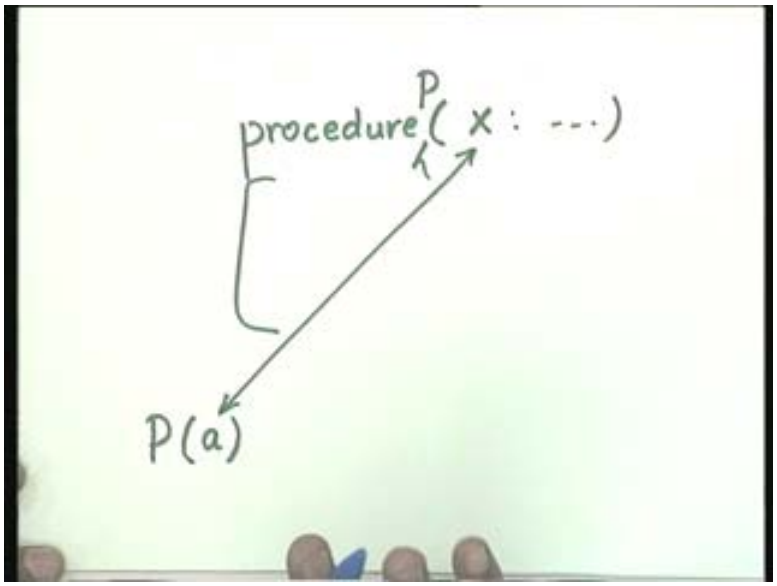
So the call-by-value is something that is implemented in Pascal, C and most imperative languages, and the only imperative language in which call-by-name was implemented was Algol-60 and certain derivatives of Algol-60 like the Stanford artificial intelligence language, Sail I think they use call-by-name. But it is a very dicey kind of thing to implement so it is quite unpopular. Call-by-reference is of course is there in Pascal and Fortran and then there is also something called call-by-value result and I say Aida question mark because it is not clear what Aida wants you to implement so it is not absolutely clear and we will see the reason for that.

If you remember the activation records, stack and so on there was a place for parameters, just keep that in mind. So, in a call-by-value implementation what you do is you evaluate the actual parameter to obtain a value. So the actual parameter could be an expression including an arithmetic expression containing just constants or whatever you like but you actually do all the possible reductions that are possible till you get a value. And having done that there is a formal parameter.

Let us look at the general structure of procedures; you will have a procedure text which has some formal parameters let us call it x and just assume there is one parameter and the procedure has a name P and you might have a call to this procedure. This procedure could also be a Pascal function. What I mean is there is no fundamental difference between functions and procedures in Pascal or in Fortran because all it means is having an extra local variable to store the result otherwise they are both absolutely the same.

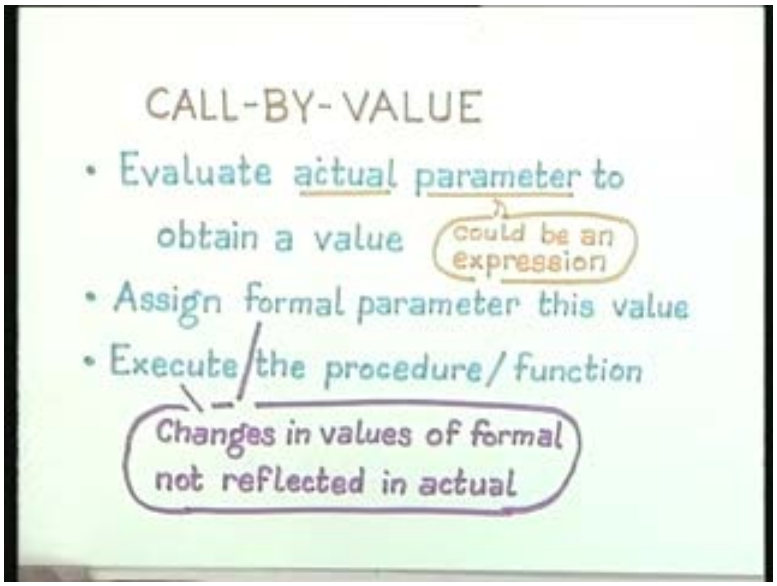
So we will just look at them as procedures and there is an actual parameter. And of course you have to draw the formal to actual correspondence. And the whole question of interpretation now reduces to in this call should x denote the r-value of a, should x denote the l-value of ‘a’ or should x be the name for this piece of text, just a name for this piece of text.

(Refer Slide Time: 15:07)



Those are the three mechanisms that we are really interested in looking at. Then what do you do in a call-by-value implementation is you evaluate the actual parameter down to an r-value and assign the formal parameter this value. That means you are performing an implicit assignment operation where the formal parameter is initialized to this value and then you execute the procedure or function with this initialization of the formal parameters.

(Refer Slide Time: 15:47)



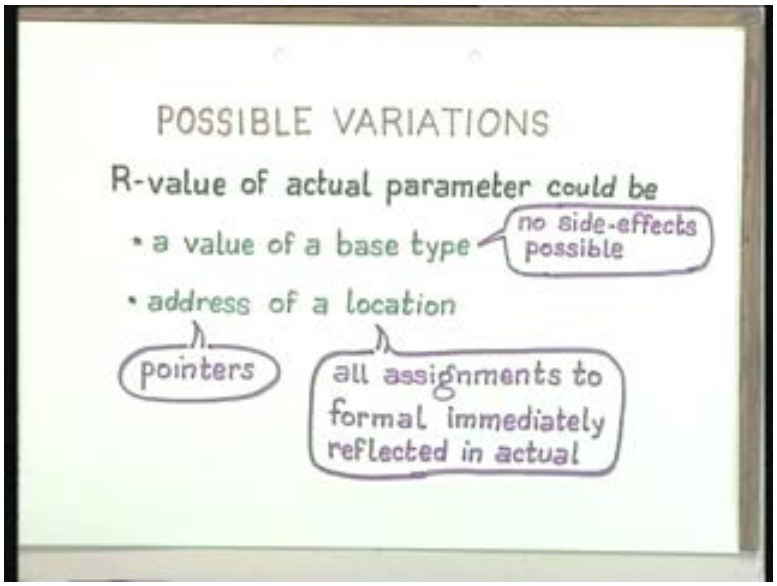
So now, because using this particular way of going about with things supposing the actual parameter were an expression that could be an l-value either the actual parameter is a simple variable, an array component or a record field then what happens is this process does not change the values of (Refer Slide Time: 16:33). Therefore within the procedure or function if you change the value of the formal parameter that is not reflected in the value of the actual parameter.

So in this call-by-value implementation if you take a broad pragmatic view is really for doing things like, I have got a value 'a' and I want to find sine a or cos a, elementary mathematical functions where you are not really interested in producing side effects. Of course what happens with most of our programs is that within the function there are side effects due to various reasons. Very often the reasons are [lie..17:10] in Pascal or the appropriate language. For example Pascal has a restriction that a Pascal function can return only a scalar data type value.

Supposing I have a function which actually has to return two different values what do I do?
I might do some silly thing like making one of the values global returning one value and producing a side effect within the function, that is one possible way of doing it. So we do such things but whatever are the changes in the values of the formal parameter in the execution of the procedure will not be reflected back unless you actually use or you modify that actual parameter also in the procedure or function. So, changes in the formal will not be reflected back in the actual in general. But there are specific instances where it could.

What kinds of variations you can have?
For example, your actual parameter could be a value of a base type which means that there will be no side effects possible, here whenever I talk of no side effects possible I mean side effects through modification of the formal parameter and not side effects created by explicit assignments to global variables within the function or the procedure. However, the r-value of the actual parameter could be a location. After all the addresses are also r-values so it could be a location in which case what happens is that you are essentially dealing with pointers and then all assignments to formal parameters are immediately reflected in the actual.

(Refer Slide Time: 19:20)



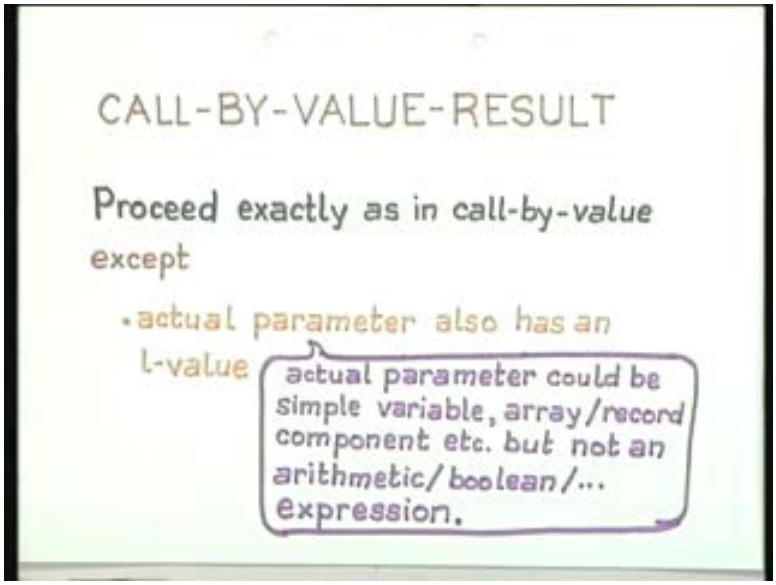
And in fact if I were to write various Lisp based programs in Pascal, for example I want to take all these elementary scheme functions and programmed in Pascal then what would I do?
I would define a function call const which takes a value and a pointer to a list and I would actually const it and do things and that pointer to the list could just be a call-by-value parameter. But since it is a pointer and I am going to go through this process of first creating a new location and then modifying the pointer and so on and so forth I will actually get side effects the end result that I get would actually be a new list with that new element const to it.

So under cases where the r-value is actually an address, so what it means is that there is a heap based data structure which you are modifying somewhere but you are not actually modifying the value of the location in the formal. But you are modifying some large structure that is pointed to by this r-value and once you have modified that you have essentially changed your global environment. So, on your return from the procedure you have got a modified global environment. So, for example, lopping off the head of a list, I could just pass the list as a pointer, lop off the head by de-allocating the head, putting the pointers in some particular way and then just get back, ostensibly there are no side effects because of a call-by-value implementation but the new list that I get is really the tail of the original list. And in fact this is what is used mostly in ‘C’ to produce side effects. The ‘C’ uses only a single parameter passing mechanism and that is call-by-value. And the way it uses call-by-value, even though for normal variables call-by-value produces no side effects. The way it produces call-by-value is that it uses value parameters through explicit referencing and dereferencing operations.

Therefore, you can actually pass the value which is of pointer to something, dereference that pointer change the value through an explicit call-by-value mechanism and get back. And you would have produced side effects though the actual parameter would not have

changed its value. It could still be a pointer to the same location but you have changed the contents of that location and so you have produced a side effect. That is how ‘C’ for example does not allow for any other parameter passing mechanism. It allows only for call-by-value but since it has primitives for referencing and dereferencing you can produce side effects through the call-by-value mechanism. There is another variation of the call-by-value parameter mechanism and that is called the call-by-value result and that is very simple. You just proceed exactly as in call-by-value; the only extra constraint is that your actual parameter should also have an l-value.

(Refer Slide Time: 23:20)



If your actual parameter is some arithmetic expression 2 plus 3 is what comes to the mind always then obviously call-by-value doesn’t satisfy this constraint. But if the actual parameter is something which also has an l-value then what I can do is proceed exactly as in call-by-value and just as there is a ‘copying in’ phase in a call-by-value mechanism you first find the r-value of the expression and then you actually do this assignment to the formal parameter so this is the ‘copying in’ phase. so just like you have a ‘copying in’ phase after executing the procedure you have a ‘copy out’ phase where you will actually copy back values from the formals to the actuals.

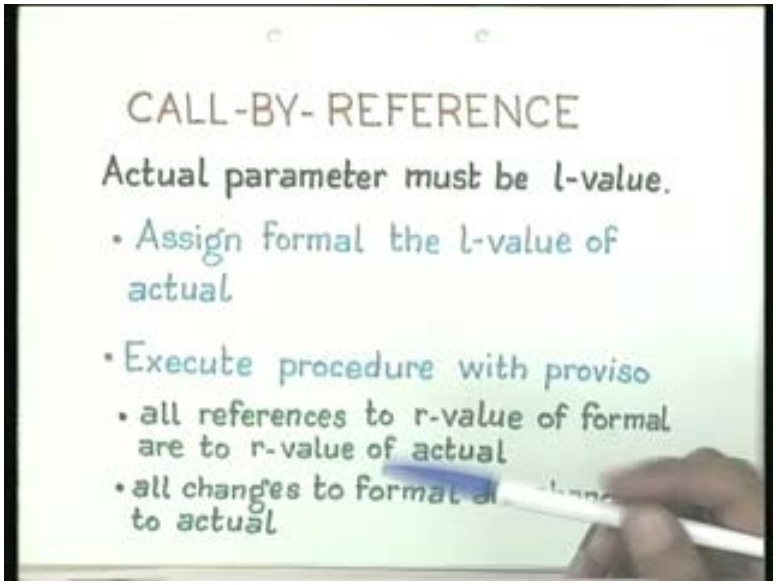
So a call-by-value result is just call-by-value plus this extra step firstly with this constraint that the actual parameter should also be a location and after executing it is because you have to copy back values from the formal to the actual and you cannot do that in a memory based system without the actual parameter actually also denoting a location also having an l-value and then this copying back is done. And essentially you have created a side effect but in a clean fashion in the sense that there is no great interference between the globals and the body of the procedure. The procedure still remains as an abstract but you are copying back the values at the end of the execution procedure. All intermediate values that the formals take within the procedure will not be reflected in the actual parameter.

The call-by-value result is not actually used in any programming language except that Aida has three kinds of parameter mechanisms. So, a typical Aida procedure could have what are known as ‘in’ parameters which are quite faithfully reproduced by the call-by-value mechanism. Then there are ‘in out’ parameters which means there is a ‘copying in’ phase of these parameters and at the end of the execution procedure there is a copying out phase also.

And then there are also ‘plane out’ parameters. So you can think of these ‘plane out’ parameters as a form of what might be called just call-by-result, there is no ‘copying in’ phase for these ‘out’ parameters, they stand essentially uninitialized when the procedure begins execution but at the end of the execution there is a ‘copy out’ of values so you do not have a ‘‘copying in’’ phase but you have a ‘‘copy out’’ phase so you can call that a call-by-result mechanism if you like.

But the whole point about the Aida language is that it does not specify how these things have to be implemented. It is just a generally accepted conjecture that this is what the designers meant so that is why there is a question mark there. Now what actually most languages from Fortran always provided was what is known as a call-by-reference mechanism which is the variable parameters in Pascal.

(Refer Slide Time: 27:48)



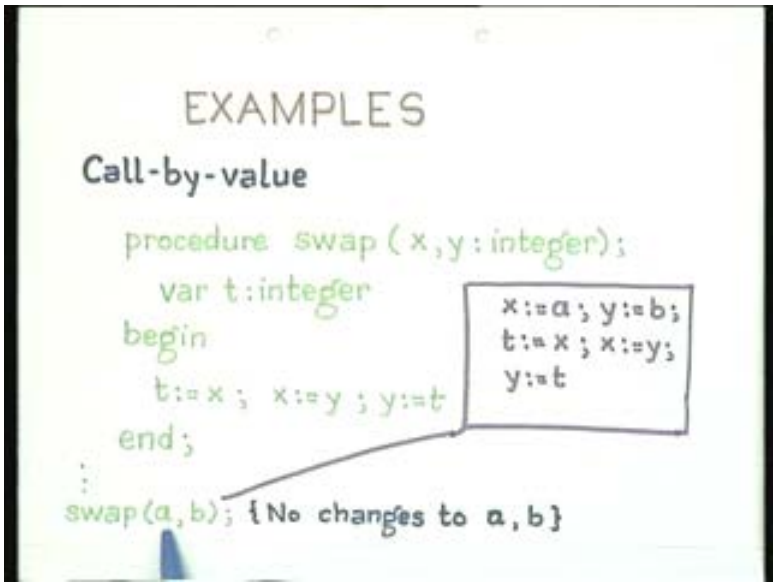
So what happens is you actually assign to the formal the l-value of the actual parameter and then you execute the procedure with this proviso so that whenever the r-value of the formal parameter is required you take the r-value of the corresponding actual parameter and whenever you want to change the value of the formal parameter you actually change the value of the corresponding actual parameter.

Thus, the formal parameter or whatever is the space reserved for the formal parameter in the activation record is just an address to the actual parameter. and within the procedure

whenever it is a formal parameter that is being assigned or being used whose value is being used actually the value is not taken from that formal parameter but you go down that pointer to the location where the actual parameter is and either use the value inside that or modify that. Therefore the call-by-reference mechanism is what was used in Fortran long time ago, it is used in Pascal through the var declaration, it is there in Algol-60, it is there in so many languages and it is there in 'C' by virtue of the fact that 'C' allows referencing and dereferencing. But it is not explicitly provided by the 'C' language. The only explicitly provided mechanism is the call-by-value mechanism.

So let us look at a few examples of these things. I will use a Pascal like syntax throughout. Let us consider just this trivial example taken from an elementary programming course on Pascal. So I have this procedure swap with two formal parameters x and y declared integer, I have an integer, an intermediate variable t also declared to be an integer and then I have this standard swap procedure.

(Refer Slide Time: 30:35)

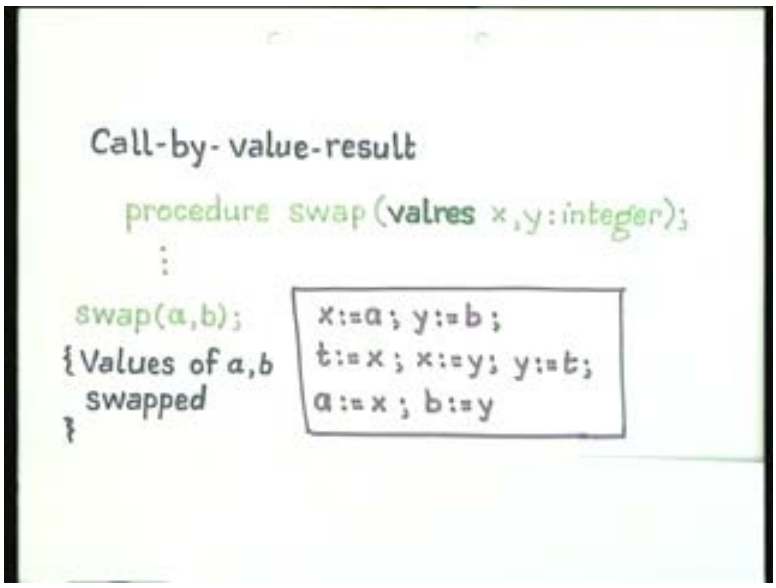


And now I call this procedure with two actual parameters 'a' and 'b'. By default if this is a Pascal program then this x and y are implemented as value parameters. So what it means is that first there is a 'copying in' phase. So the r-values of 'a' and 'b' are stored in the locations for x and y and then the values of x and y are actually swapped. And when you get out the values of 'a' and 'b' are still the same old values so there are no changes to the values of 'a' and 'b'. Therefore what you could do is you could implement it by a call-by-value result mechanism.

Supposing we extended the Pascal parameter passing mechanisms to value result then what it means is that I would assume that the body of the swapped procedure is the same except that I have made this change, I have declared x and y to be value result. Then what happens is that in the 'copying in' phase x and y are assigned the values of 'a' and 'b',

then the values of x and y are swapped in the procedure body and then there is a ‘copy out’ phase where the actual parameters are assigned the values of the formals. So that is what I mean by a clean interface.

(Refer Slide Time: 32:21)

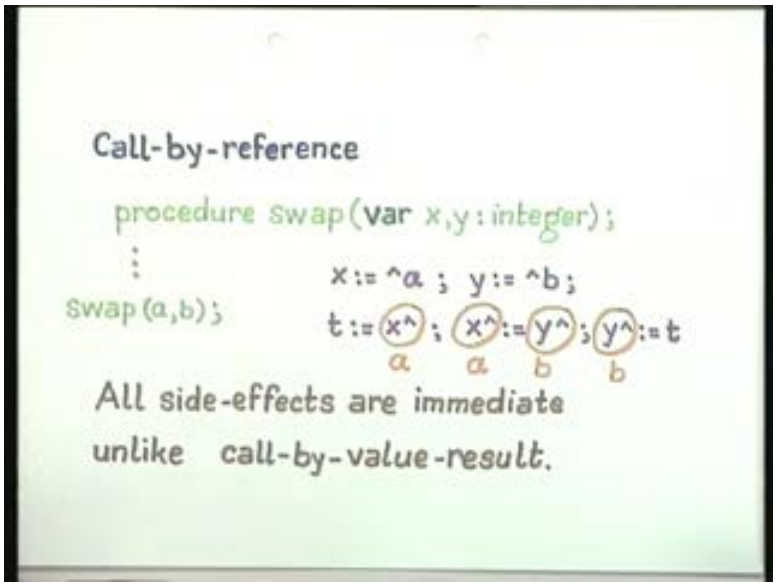


If you look at the procedure as a single entry, single exit, black box with the parameters forming the interface then it is a clean interface, that is the transmission of the parameters is through the ‘copying in’ portion and then there is a output through the ‘copy out’ phase. And while this procedure is executing for the rest of the program essentially if globals are not referenced, globals are not modified and so on then this procedure stands alone and can be used by others as a distinct complete functional unit in itself, as an abstraction in the true sense of the word.

On the other hand, since the value result mechanism is not available in Pascal what you do is you usually implement this swap by reference. So you actually define these two formal parameters to be var parameters or reference parameters and then the mechanism goes something as follows: So x is assigned the value of a pointer to ‘a’ which means the address of ‘a’, y is assigned the address of ‘b’ so this type information here only refers to the type of the actual parameter and not to the type of the formal var. All that you are saying is that x can be a pointer to an integer actual parameter and then after that what happens is that the body of the procedure executes actually in this fashion.

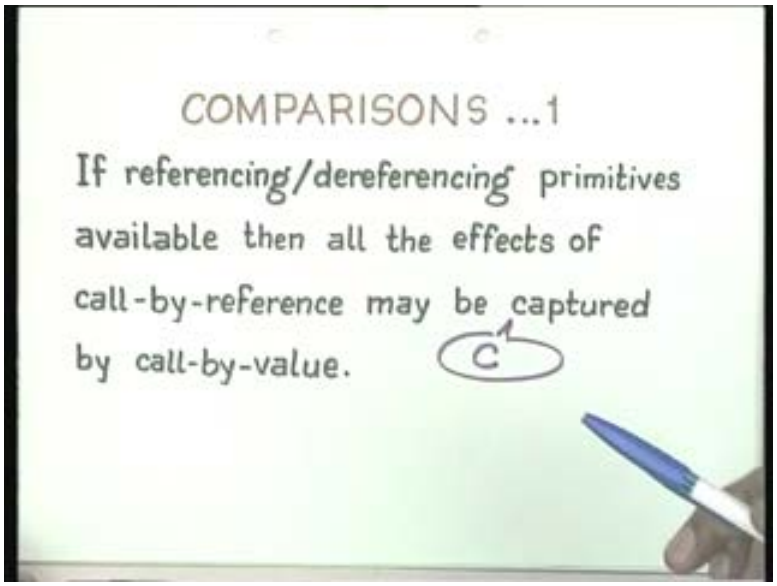
You dereference x and take that value and put it in t and the act of dereferencing x means going to the variable a taking its value and assigning it to t then the dereference x that means go to ‘a’ and to it assign the dereference value contained in y. Thus dereference y means go to the location ‘b’ and copy that value into the location pointer 2 by x and then of course copy the value of t into the location pointer 2 by y. So the side effects are immediate unlike the call-by-value result mechanism.

(Refer Slide Time: 35:15)



So essentially if you look at these three mechanisms then as long as you have referencing dereferencing primitives available what you could do is all effects of call-by-reference can be captured by call-by-value and which is what 'C' does and many people do in Pascal because of Pascal's peculiar constraints on what functions can return.

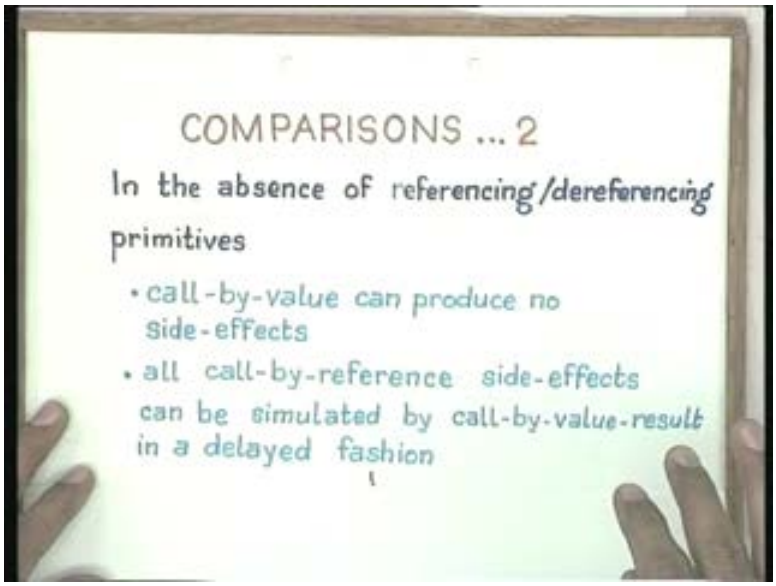
(Refer Slide Time: 35:50)



Either you define it as a procedure or define it as a function and return a pointer parameter to some structure then you will get the same result. Essentially all the effects of call-by-reference may be captured by call-by-value provided you have these referencing and dereferencing primitives already in the language otherwise you cannot. Now in the

absence of referencing and dereferencing primitives what we have is of course that the call-by-value can produce no side effects except explicit assignments to globals and so on. But all call-by-reference side effects can actually be captured by call-by-value result and the only difference is that these effects are not immediate in a call-by-value result mechanism. The effects are delayed, they are delayed till you actually are exiting the procedure.

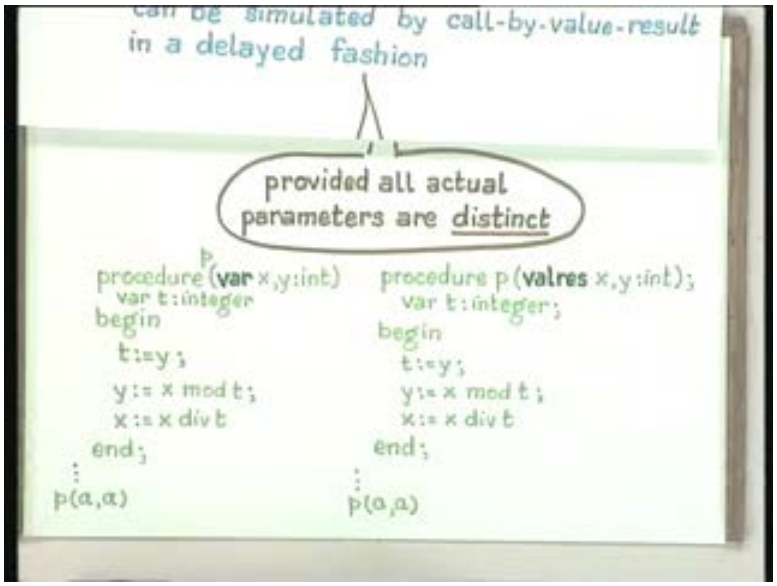
(Refer Slide Time: 36:57)



But however there is a catch to this which actually most books do not point out and the catch is here. And this you can capture all side effects of call-by-reference by call-by-value result only provided all the actual parameters in your procedure call are distinct. If you have an aliasing problem then there is no guarantee that the effects of call-by-reference will be captured by call-by-value result and vice versa. There is absolutely no guarantee that the effects of call-by-value result will be captured by call-by-reference.

Here is a simple example which actually does this. This procedure presumably takes values x and y which are integers and then it gives the quotient and remainder of the input values x and y and stores them also in x and y because assuming somehow that you do not require the original x and y and you want only the quotient and remainder then what this procedure does is it somehow does this. But then there is nothing which prevents me from calling this procedure with x and y being the same l-value. Then actually what will happen is this order of evaluation and the last statement which modified ‘a’ is what is going to take effect. In order to buffer the effects being immediate I have used this intermediate variable t and used it throughout otherwise I might have used y here.

(Refer Slide Time: 40:07)



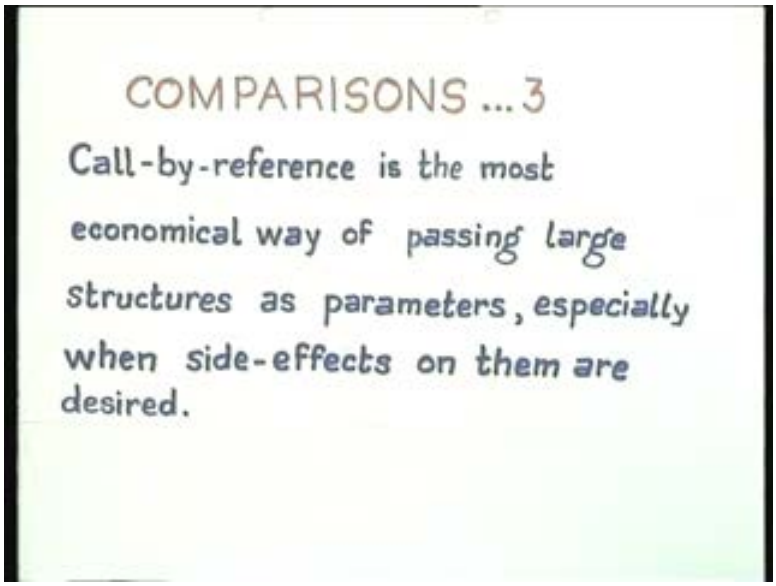
So essentially what it means is the result of this procedure call is going to be that a will have the quotient which means 1 if a is not 0 at the end of this procedure or it will not have that may be, it will have a is equal to x is 0, yeah it will have 1, I have luckily programmed it right.

Under the value result implementation what happens is that here at the end of this procedure x actually contains the quotient of the original ‘a’ and ‘a’ which is 1, y actually contains the value 0 but depending on in what order you do the ‘copy out’ phase it will be reflected in this actual parameter which means supposing you assume that all sensible compilers and run time systems do the ‘copy out’ phase in the order because parameters order are important always. So if they do it in this order then essentially what it means is that ‘a’ will have the value of y which means it will have 0. In this case you will have a is equal to 0 and in this case you will have a is equal to 1 which is an excellent reason for not doing this kind of programming.

Once you use the actual parameter inside the procedure then there is just confusion. This is just to point out (Refer Slide Time: 41:28) the difference between these two in effects which is not usually pointed out in most books on Programming Languages. Most books actually claim that the effects of call-by-reference can be captured by call-by-value result and it is not entirely true.

But call-by-value result is a very clean mechanism in the sense that it does not give you unnecessary headaches. However, the problem really is what happens when you have to pass really large structures, whereas call-by-reference actually can make things quite complicated especially with array referencing, array index computations, array index modifications inside the procedure and so on and so forth. It can make things quite complicated and reasoning about in a call-by-reference environment can be quite tricky. It means debugging also could be tricky.

(Refer Slide Time: 43:57)



However, call-by-reference is actually the most economic way of passing large structures, huge arrays, a bitmap of yourself for example to be transformed somehow into may be to make you look handsome or something. But if you are going to pass large structures then you would also like to be economical about your usage of the store so what you do not want is to create an output which is another large structure.

So very often you just want to create the side effects on that structure itself and how those side effects are available to you at the end of execution of the procedure. If you have really large structures like a million bits by a million bits passed as a parameter then the 'copy in' phase and the 'copy out' phase are going to be extremely expensive.

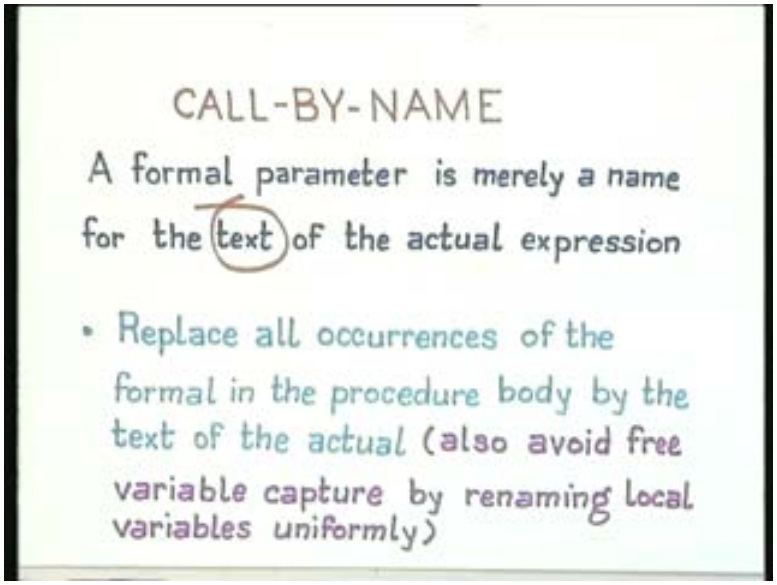
So even though you might get your results through a call-by-value you might get correct results through a call-by-value result mechanism. It is actually faster and more space efficient to use a call-by-reference mechanism, to produce the required side effects so that is what happens with call-by-reference. This is one of the reasons why call-by-reference is most popular because even for large structures all you require a single address, that is your parameter. So your parameter will just contain a single address to the start of the structure.

Whether it is a large list structures or large array structures or record structures it does not matter. One address is all that you require a single pointer. Then let us look at the call-by-name mechanism. So in a call-by-name mechanism essentially what you are saying is that a formal parameter is just a name for the piece of text which comprises the actual parameter. So what you do is you go back to the lambda calculus and try to model the behavior of the beta reduction there exactly in your programming language. So how would you do that?
That means before executing the procedure I replace all occurrences of the formal parameter in the procedure body by the text which is the actual parameter. Therefore

essentially the effect of a call-by-name is to produce a macro expansion. You produce a macro expansion except that just as in the case of a beta reduction you have to be careful there should be no free variable captured.

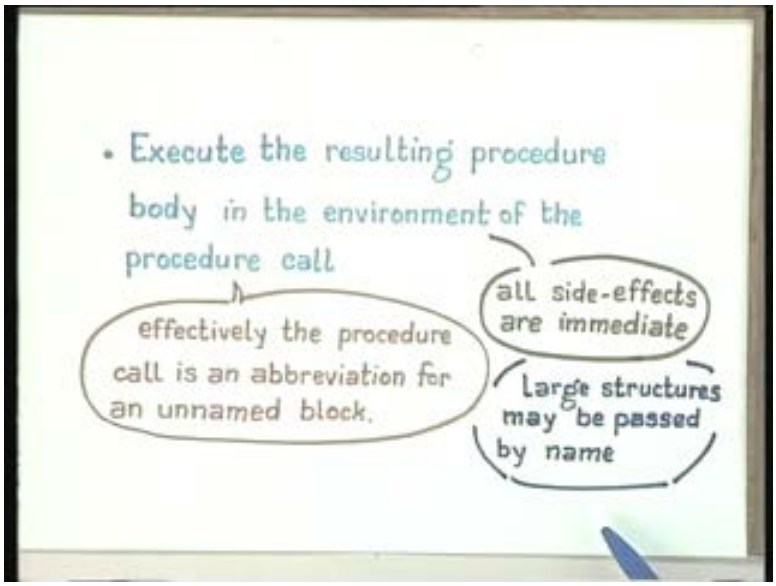
Of course in a compiled language this may not be such a serious problem. Since every variable would have an address determined by the base of its activation record and the relative address with respect to the base of its activation record this problem may not be so serious at run time for a compiled language. But looked at as a form of a beta reduction being modeled in an imperative programming language what it means is that you have to replace all occurrences of the formal in the procedure body by the text of the actual and to avoid free variable capture what you have to do is you have to rename all local variables in the procedure by an alpha conversion mechanism.

(Refer Slide Time: 47:05)



But in a compiled language that part of it could be automatically done. But what happens is that essentially when you replace all these formals by the text of the actual expression the resulting procedure should be executed as an unnamed block in the calling environment. That means the procedure call is just an abbreviation for taking the entire body of the procedure after replacing all the occurrences of the formal parameter by the text of the actual parameter. So the procedure call just has the meaning that it is an unnamed block, a procedure call is just an abbreviation a parameterized abbreviation for an unnamed block that has to be executed at that point with a substitution of the parameters by the text of the actual parameter and of course ensuring that there are no free variable captures. So this aspect is really a beta reduction in the lambda calculus and that is what is obtained by a call-by-name mechanism. And since all the execution is in the environment of the procedure call all side effects are immediate. And you can actually pass large structures because you will just be giving the large structure a name and passing it that is it.

(Refer Slide Time: 49:05)



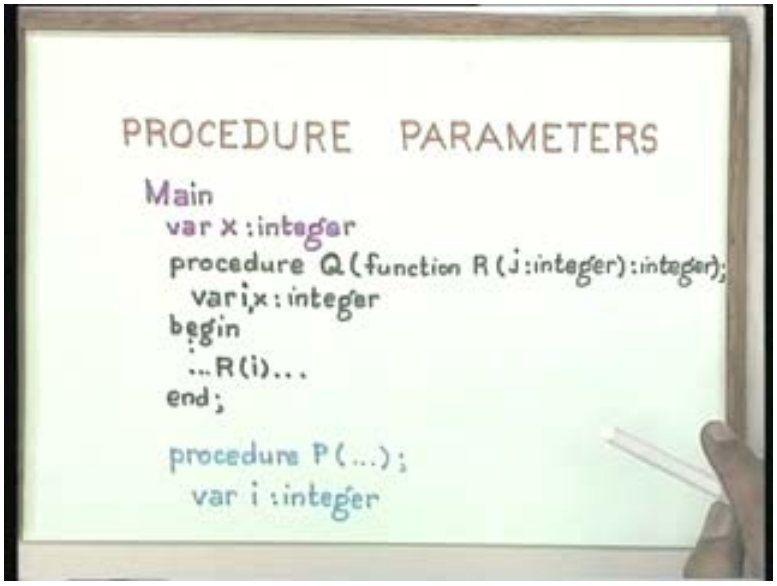
So depending on how your run time environment deals with large structures, for example arrays regardless of what language you are looking at arrays are still just a pointer. Therefore your side effects will be immediate and you do not have this extra overhead of 'copying in' 'copying out' and all that the effects on large structures are also captured immediately, array dereferencing or array indexing and so on. The only problem with this is that reasoning about the program from its text can be quite a problem. And of course there is no guarantee that call-by-name is equivalent to call-by-reference or any other mechanism for that matter all these mechanisms are different. Because what happens is that it is different from call-by-value because in call-by-value the parameter is evaluated once to a value and that value is used throughout all references of the formal within the procedure. In the call-by-name because of the textual replacement that actual parameter is evaluated every time that formal is referenced inside the procedure. So what it means is an expression like a plus b passes an actual parameter to the procedure so that a plus b can mean different things for different references of x in the procedure because you may have modified a between two successive references to the formal parameter.

So, in general, for imperative languages this is not the same as using a call-by-value mechanism. It means repeatedly evaluating that expression every time that formal is encountered. You execute the text corresponding to the expression and essentially reevaluate that expression. But you re-evaluate that expression in the calling environment further this call-by-name mechanism can produce different effects depending upon whether your language is a statically scoped language or a dynamically scoped language because the text of your parameter really consists of non-local references. Then depending upon where it is being called those non local references have different bindings.

If you have it in the environment of the call then in a statically scoped language the parameter is referred to the innermost textually enclosing block whereas in a dynamically

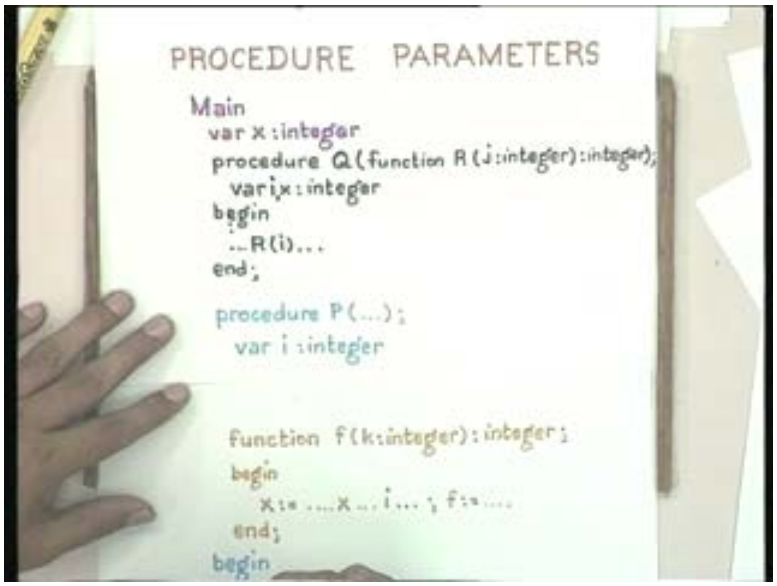
scoped language they refer to the binding in the innermost calling block. So even call-by-name depending upon whether you are talking about a statically scoped language or a dynamically scoped language can produce different effects. Lastly there is just one more thing that is many languages like Pascal have procedures as parameters. And here there is no question of whether it is call-by-name or call-by-value or anything of that sort.

(Refer Slide Time: 53:05)



So supposing you have this set of procedures this main body where of course the main body has this call to the procedure P which is blue the main body has two procedures Q and P, Q is black and P is blue, the blue procedure P has a function inside it, the black procedure Q is a procedure which calls a function R as a parameter and this R is also a just a formal name and then let us say there is a call to this R(i) here, (Ri) is used here and now what happens is if you look at this call there is a call to P and then within P there is a call to Q with the function F passed as a parameter. That means this formal parameter R now refers to this function deeply nested inside this P.

(Refer Slide Time: 54:23)



So the question is what is the environment of non-local references inside the function?
In a statically scoped language this pink x here in the main body, any reference to I here should be this blue I here, Q itself has an x and an I, this black I here refers to this black I here but when you call this function with this I here then you are actually using this blue I within Q so you have to somehow create the static environment suitable for F in order for F to be executed at this point and that is very simple.

What you do is at the point of this call for a function or a procedure parameter along with the parameters and so on you send the static chain pointer address for the current base so that all references to I and x within this function would refer to the non local references in keeping with the statically scoped structure of the language, this is separate. But this is a very weak parameter passing mechanism and actually creates more confusion than it solves but it is an excellent way of programming higher order functions like map at least first level higher order functions like map and whatever list based functions which have to be applied over the entire list.

Or, for example, for solving you can have a Newton-Raphson's method in which the actual method is called through a parameter. The actual function could be a trigonometric function or a hyperbolic function or could be a polynomial or mixtures of these things you call that as a parameter and that is the reason it was introduced in Pascal for doing such kinds of computations independent of actually what the value of the function is. So you can call Newton-Raphson's with this function F and may be find a fixed point solution. But what it means is along with the function name the type checking becomes problematic, the rules of Pascal are inadequate to adequately type check this mechanism because Pascal tries to save on your effort by saying that you do not need to declare all the variables all the parameters but then if you do not declare all the parameters how are you going to type check.

Pascal thinks it is being more flexible by not allowing parameters so that now you can actually call it by different places with a unary function or a binary function or a ternary function but then you can't type check those things and it violates the static type checking rules of Pascal that at compile time everything should be type checkable, it violates that basic requirement. So anyway it is not a very popular thing but it is a way of programming higher order functions in an imperative language.