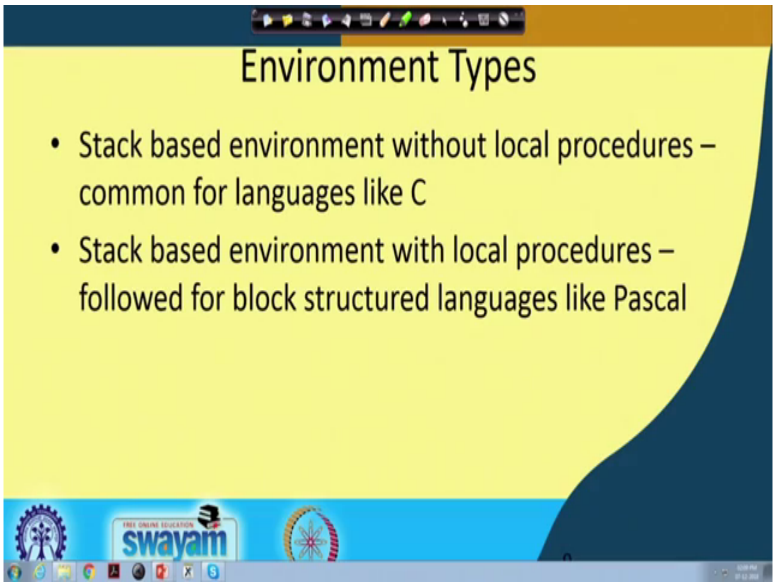
Compiler Design

Prof. Santanu Chattopadhyay
Department of E & EC Engineering
Indian Institute of Technology, Kharagpur

Lecture – 48
Runtime Environment (Contd.)

(Refer Slide Time: 00:18).



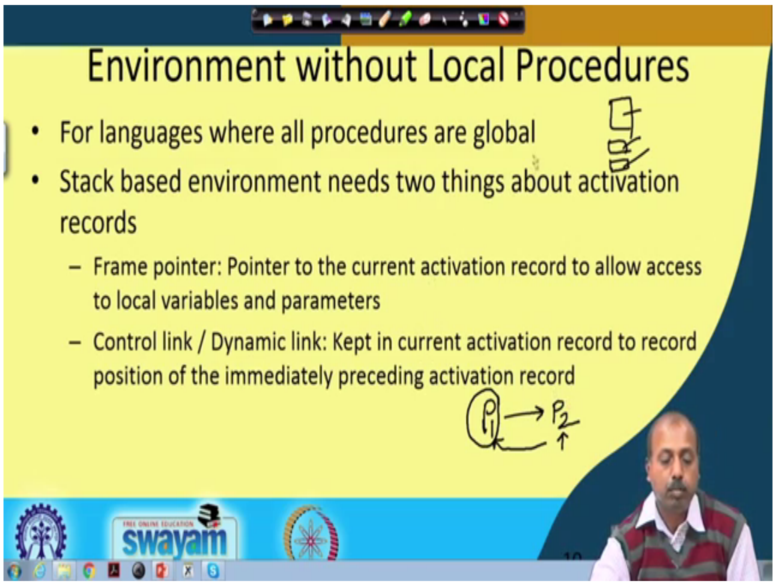
Environment Types

- Stack based environment without local procedures – common for languages like C
- Stack based environment with local procedures – followed for block structured languages like Pascal

So, in our last class we have been discussing on Runtime Environments. And we have seen that there can be two types of environments that are possible; one is without local procedure and another is with local procedure. So, without local procedure means; within a procedure we are not going to define any new procedures. However, and with local procedure mean within a procedure; so we can define your own procedures further.

So, that the procedure is local for so, inner procedure is local for the outer procedure only and they are not visible outside the main procedure; so we will come to that. So, this the first type of case, where we do not allow this local procedure. So, that is C type of languages they have this facility. On the other hand so, the impel languages like Pascal; so you can have local procedure so that within a procedure, we can define another procedure.

(Refer Slide Time: 01:13).



Environment without Local Procedures

- For languages where all procedures are global
- Stack based environment needs two things about activation records
  - Frame pointer: Pointer to the current activation record to allow access to local variables and parameters
  - Control link / Dynamic link: Kept in current activation record to record position of the immediately preceding activation record

$$\begin{tikzcd} P_1 \arrow[r] & P_2 \arrow[l, bend left] \end{tikzcd}$$

So, we will see both of them and then how to go for the their implementation in runtime, how are they going to be handle. So, with local procedures we sorry, without local procedures; so for languages where all procedures are global. So, that is the situation where we have got lo local non-local procedures. So, here if you look into the type of this languages.

So, if you are writing a program. So, we can have a main program and after that you can have a number of procedures and this procedures are visible to, all this procedures are visible for in the entire program; so that is the situation. So, we do not have difficulty as per as this definition of procedures are concerned. However, there maybe problems with that situation, that or the every all the procedures are visible at all or at all places.

So, that may be a concern and some sort of hiding that you have otherwise so that is not visible. So, if you have got some design style, where you have got a main operation to do and under that only certain operations are valid. So, that type of situation; so you may like to hide the other procedures from this the internal procedures from outside ao, that is not possible.

So, here all procedures are global; so any procedure defined in the language; so. for the entire program it is visible. So, in this type of situation, this stack based environment that we have that we create for this local procedures, they need two things for the activation record; one is known as frame pointer. So, in frame pointer so pointer to the current activation record is there; so that is a there is a that is called a frame pointer. So, this allows access to the local variables and parameters. So, this actually points to the current frame, that is the frame corresponding to the current procedure that has been called. And there is another link called control link or dynamic link. So, that will be that will keep the information about from which procedure this was invoke.

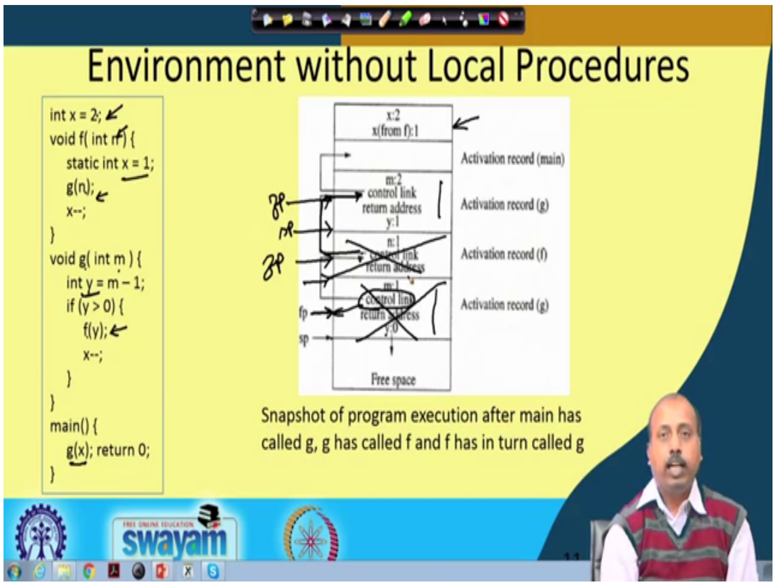
Accordingly, from which activation record so this activation record got created. So, this is basically the parents parent relationship like if there is a say procedure P 1 and that procedure P 1 is calling , if that procedure P 1 is calling say procedure P 2.

So, if P 1 is calling P 2 then, if this P 1 is calling P 2, there in the P 2s activation record. So, there will be a pointer, which will 0.2 this P 1s activation record; so that will happen. So that is that will be called control link or dynamic link. So, the idea is that when P 2 will be over so, we need to say that P 1s activation record is the current 1.

So, accordingly that can be active that can be made the current, current procedure. So, basically this frame pointer that we have so, when we come back from P 2 to P 1, this frame pointer can be made to 0.2 the activation record for P 1 and we keep the information about activation record of P 1 in the control link of P 2.

So, we will see some examples that we will make it more clear. So, as we, as we go into more and more, some example; so this will be more clear. So, let us see how this things are going to work.

(Refer Slide Time: 04:43).



Environment without Local Procedures

```
int x = 2;
void f( int n) {
   static int x = 1;
   g(n);
   x--;
}
void g( int m ) {
   int y = m – 1;
   if (y > 0) {
      f(y);
      x--;
   }
}
main() {
   g(x); return 0;
}
```

Snapshot of program execution after main has called g, g has called f and f has in turn called g

So, this one is an example like; here I have got a function say, this is the main, this is a function f and there is another function g and we have got a main routine. So, as there is so, this is f and g, you see they are global procedures or global functions, whatever you call it and that is available throughout the program.

We have got a global variable integer x 2 here and the within this functions f and g we have got this local variables. So, this is static i integer and this is a normal integer. So, this is this is a local integer. So, this is a local variable y is a local variable and x since, it is defined as a static integers so that is in that sense it becomes global.

So, it gets allocated to the static area or the global area. So, you see that in this area this x are the main program the global variable x and this static variable x so they have been allotted. Now, when this function is called so from the main routine; so this is the activation record for main and from main, we have given a call to g.

So, in the call for g we have got this variable this, this activation record got created, when this when this function g is called so, this part of the activation record is created and there; we have got this, this is the parameter m that is passed. So, it is getting the value gx and these x refers to this global variable x. So, m is m is m is getting the value 2. So, this parameter is passed, after that there is a control link and that control link points to the previous frame ok.

So, from where this g has been invoked. So, this g has been invoked from this main. So, as a result this control link points to this. So, then we have got there is this return address so that that will be stored and a local variable y has been created. So, y is allocated space in the activation record for g and y equal to m minus 1. So, y evaluates to 1 at this point of time.

So, after sometime when this, this point comes, it is calling the function f and as a result, there will be another activation record created for this call f and in that we have got this, this parameters got a parameter n

So, this n is given allocated space on to this activation record and this value of n is equal to the value of y and value of y was 1 so this gets the value 1. And the control link in the activation record for f, it points to the for the; it points to the frame corresponding to the function g from where the f is called like here the function f is called from function g.

So, this activation record this control link points to the, previous frame from where this particular call was invoked. And then at that at some point later when is the in the call f so, here again g is called so, f and g are they are to mutually recursive procedure.

So, at some point of time suppose this g has been called within f. So, as a result so, next activation record for g will be getting created and here I will have this, this m. So, there, here we have got this call to g and g has got parameter m. So, this m is coming and this m is a assign the value 1, because this g of, it is calling g of n. So, whatever value was passed here; so with that it is called; so, it is calling with n 1.

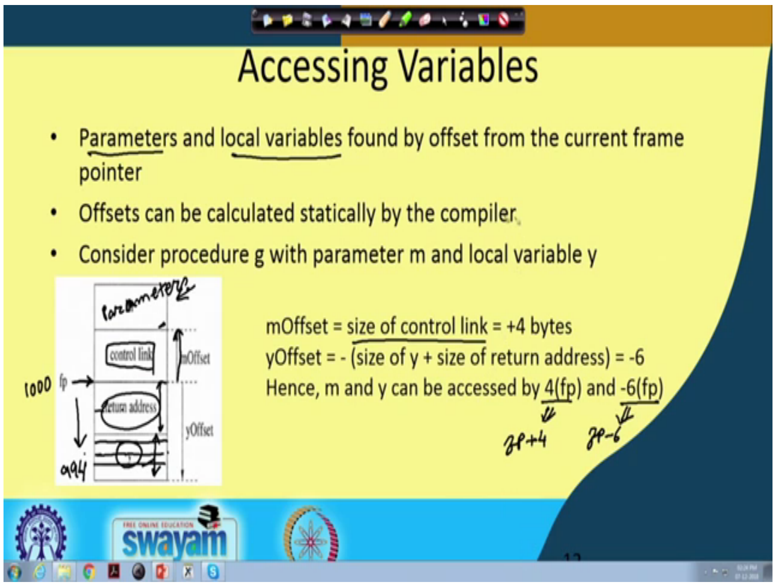
So, that way, this is this will be this g is called with the value of m being equal to 1 and then the control link points to the previous activation record of f and this return address and the another copy of this local variable is created y, that is made equal to 0, because that is equal to m minus 1, it evaluates to 0. And then you see that this control link points to the previous activation record and the current frame pointer points to this current frame, the currently active procedure and the stack pointer points to the up to which this is full, this stack is full activation record stack is full; so it points to this.

So, this is the snapshot of the program when main has called g, g has called f and f has called g recursively; so this is the situation. Now, after say this g gets over so, in the execution of g so, suppose y is such that. So, y is equal to 0. So, it will not call f any more. So, it will be over after sometime then this part of the activation record will get deleted and this frame pointer, it will get the value from this control link.

So, what this control link we are storing the frame pointer for the previous activation record. So, that value will be copied to the frame pointer as a result the frame pointer will be coming to this point. It will point to the activation record for f and so once this is d, deleted. So, stack pointer will automatically come to this point; after sometime when this one also gets over so, this will get deleted and now, the frame pointer will get the value of previous control link that we had so, this control link was pointing this.

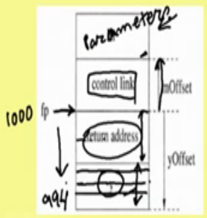
So, this frame pointer will be coming to this point and the stack pointer will come to this point. So, this way the so this one also get delayed in destroyed. So, this way this activation records they grow dynamically on to stack. So, as and when we are calling a new procedure the corresponding activation record is getting created and it is getting stored into the stack.

(Refer Slide Time: 10:54).



Accessing Variables

- Parameters and local variables found by offset from the current frame pointer
- Offsets can be calculated statically by the compiler
- Consider procedure g with parameter m and local variable y



mOffset = size of control link = +4 bytes

yOffset = - (size of y + size of return address) = -6

Hence, m and y can be accessed by 4(fp) and -6(fp)

fp+4 fp-6

Now, so, this is fine. So, this is fine as long as we do not have local procedures. Now, how do you access variable? So, this is a typical situation where we have got this frame ok. So, this is this is a this is a frame in the activation record and in that frame. So, there is a frame pointer. So, this frame pointer; so this actually points to the, a points to a position where before so, there is a portion of the activation record above the frame pointer and there is a portion of the activation record below the frame pointer.

So, above frame pointer, so, we have got the parameters that are passed ok. So, parameters and local variables so, these are the two things that the compiler we will need to access; need to get the addresses so that during execution, those locations can be accessed by the computer. So, this is parameter and local variables, they are found by offsets from the current frame pointer.

So, current frame pointer is here and above that there will be, there is a, there control link is told. So, this is again and a address so, certain number of bytes will be needed. So, above that we have got all the parameters. So, this portion is for the parameters; this portion is for the parameters. So, in this case we have got only one parameter m.

So, that is told there and then so, starting from the frame pointer. So, if you go by a negative offset so, so, go by a positive offset. So, you can find you can reach this particular point; so this offset for the parameters. On the other hand this local variables; so, they are stored after the frame pointer.

So, that is after this frame pointer, we have got this return address and after that we have got all the offsets all the local variables created. So, the way local variable y gets created here. So, we can understand that the offset of this frame pointer is, offset of this parameters is equal to the size of control link ok. So, that is the beginning of the parameter. So, in this particular case there is only one parameter.

So, the offset of m is equal to size of control link equal to plus 4 bytes and in this function g that we had previously considered there is only one local variable that is named y. So, offset of y is size of y plus size of return address, because we have got this field size of so the size of return address will come and after that we have got this space for y.

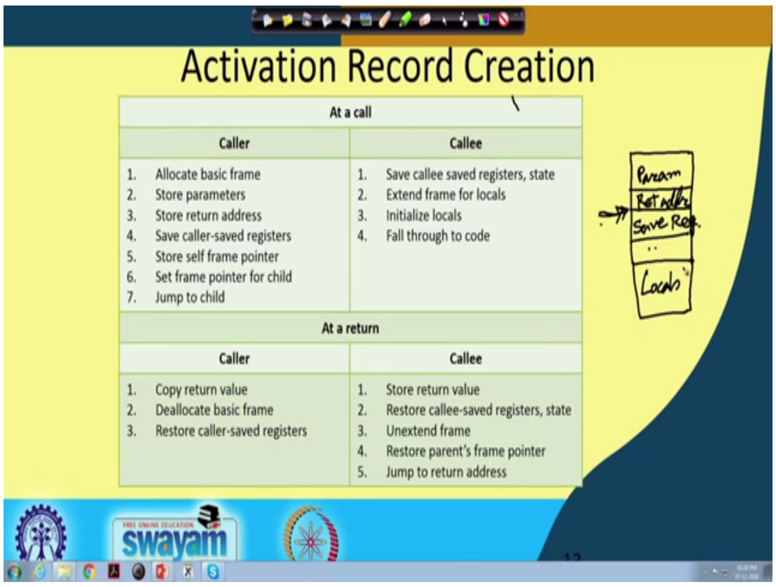
So, they so negative of that, so minus of size of y plus size of return address. So, if this frame pointer is say 1000, then this offset of y becomes 994 ok. So, 2 bytes are used for storing the return address and 2 bytes are used for the storing this y. So, you can say that at an offset of so, return address is 2 bytes and say integer is 4 bytes.

So, as a result total is 6. So, from 994 so this, so this is 4 bytes. So, if I take integer to be 4 bytes. So, this will be 2 plus 4 6. So, it will be by minus 6 offset from the frame pointer. So, this m and y so, they can be accessed by 4 fp and minus 6 fp. So, 4 fp means;

it is fp plus 4 and then this a minus 6 fp means; fp minus 6; so these are the two values that we have.

So, accordingly it can access the parameters and this local variables. So, this accessing of these parameters and local variables are with respect to the frame pointer. Next, so if I have got multiple number of parameters passed or multiple number of local variables. So, they are offsets will be calculated accordingly the compiler can find out the actual byte at which this is value will be stored it can do that.

(Refer Slide Time: 15:11).



Activation Record Creation

At a call	
Caller	**Callee**
1. Allocate basic frame 2. Store parameters 3. Store return address 4. Save caller-saved registers 5. Store self frame pointer 6. Set frame pointer for child 7. Jump to child	1. Save callee saved registers, state 2. Extend frame for locals 3. Initialize locals 4. Fall through to code
At a return	
Caller	**Callee**
1. Copy return value 2. Deallocate basic frame 3. Restore caller-saved registers	1. Store return value 2. Restore callee-saved registers, state 3. Unextend frame 4. Restore parent's frame pointer 5. Jump to return address

So, activation record creation. So, the so, there is some part of responsibility on, on behalf of the caller, the function which calls this new function and the procedure which calls the inner procedure and then there is and there is responsibility of the callee also.

So, this caller calls the callee. So, this activation record creation it is partial responsibilities is with the caller and partial responsibility with the callee; so let us see what are the things that are going to happen at a call at the caller end first. At the caller end; so it will be allocating the basic frame, because the first a frame has to be created. So, that frame gets created by the caller, store parameters so, parameter values will be stored there.

So, it you can understand that. So, it is like this. So, like activation record suppose this is the activation record that has been created by the caller ok. So, in that caller so, it will be first it will store the parameter; so in the first part, it will be storing the all the parameters. All parameters are stored, then it will store the return address; so this is the return address that is stored.

So, these are known by the caller. So, they are filled by the caller and then there may be some registers CPU registers, which are saved by the caller, because there maybe some registers, which are very useful for the caller and the caller does not want that the callee routine should override them ok.

So, that way it can save some registers. So, some save registers which are important for the caller. So, that can happen and after that it will store self frame pointer. So, it will store the frame pointer in the actually, this is that. So, it will the self so, in the frame pointer that I had previously so, that frame pointer will be stored here and then it will jump to the child routine. So, this set frame pointer of our child so, frame pointer so, for child will be pointing to this new frame pointer and then it will be jumping to the child routine.

At the callee end we have got the, their maybe some registers that callee wants to save. So, they will be saved and then it may, it like to save, some state of the some variables also so, that can also be saved then extend the frame for locals after that it will extend this frame so that it can create all the local variables there.

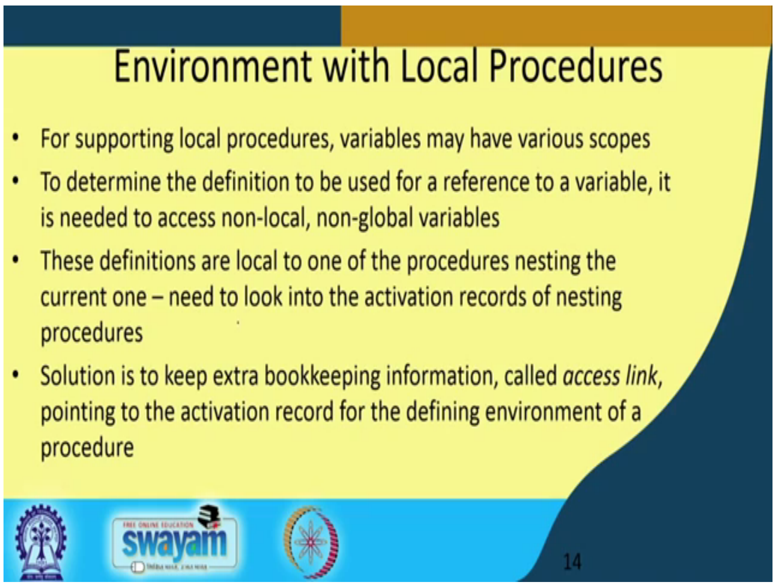
So, this is the all this is the space for the local variables and then it will fall through the code; so this is the responsibility on part of the callee. Similarly when you are returning from the procedure then there is are there are certain responsibilities for the caller and certain responsibilities for the callee. On the caller side, it will copy the return value deallocate. So, let us first talk about the callee, because while returning callees for portion comes first.

So, it will store the return address a return value in the slot in the activation record restore callee saved registers and state. So, whatever registers callee had saved so, they will be retrieved and they will be restored. Then unextend frame; so unextend frame means; this locals will get deleted, the stack pointer will be implemented; as a result all the locals will get deleted. Then restored parents frame pointer; so frame pointer of parent was stored. So, that is restored now and it will jump to the return address available in the activation record.

So, this way so, this is the portion of the job that is done by the callee. On the other hand, the caller it will copy return value. So, the return value may be copied into some array some variable or so, so, that will be done by the caller. So, then the frame will get deallocated and so, the caller saved registers whatever registers the caller had saved. So, it may like to restore those registers; so that way this will be done.

Now, one thing you should understand that these saving of registers and also. So, it is very much dependent on the system ok. So, it may in some cases; so it may be that this saving registers is necessary some cases it is not that important. So, based on that so, this compiler designer may take a decision that whether to save this registers into the frame or not.

(Refer Slide Time: 19:57).



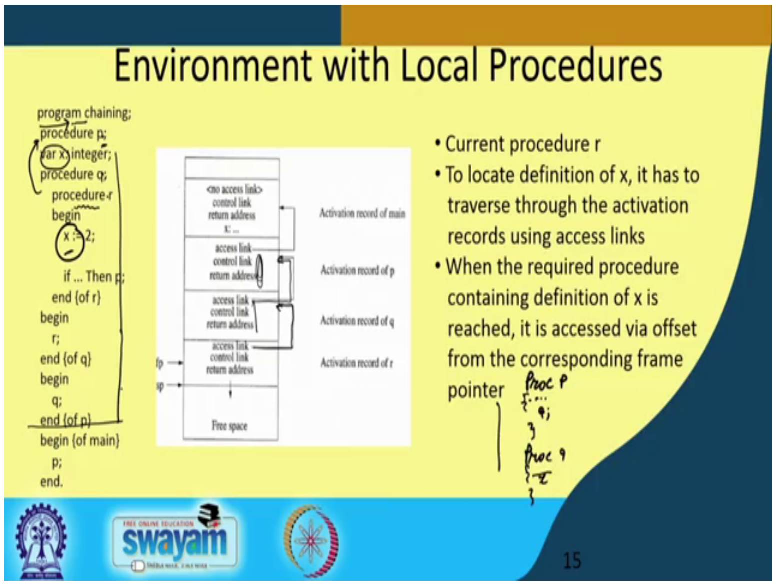
Environment with Local Procedures

- For supporting local procedures, variables may have various scopes
- To determine the definition to be used for a reference to a variable, it is needed to access non-local, non-global variables
- These definitions are local to one of the procedures nesting the current one – need to look into the activation records of nesting procedures
- Solution is to keep extra bookkeeping information, called *access link*, pointing to the activation record for the defining environment of a procedure

14

So, next we will be looking into this environment with local procedures like if there are local procedures in my description, in my program then how are they going to be handled?

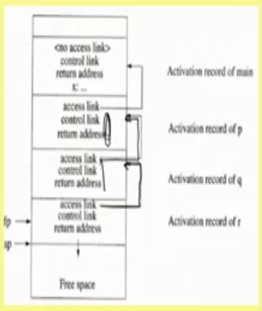
(Refer Slide Time: 20:13).



Environment with Local Procedures

```
program chaining;
procedure p;
var x: integer;
procedure q;
  procedure r
  begin
    x := 2;

      if ... Then p;
  end {of r}
begin
  r;
end {of q}
begin
  q;
end {of p}
begin {of main}
  p;
end.
```



- Current procedure r
- To locate definition of x, it has to traverse through the activation records using access links
- When the required procedure containing definition of x is reached, it is accessed via offset from the corresponding frame pointer

```
Proc p
{ ...
   q;
}
Proc q
{ r
}
```

15

So, the basic problem that we have is something like this like this is a program whose name is changing. So, here I got a procedure p, it has got a variable x of type integer and a procedure q. So, procedure q is defined inside procedure p. So, within procedure q another procedure r is defined. So, this begin end is the portion for r after that we have got. So, this begin is corresponding to the procedure q.

So, procedure of q up to this much; so this is actually the definition parts. So, the variables, local procedures, etcetera; so they can be defined here. So, of course, in this case we do not have any local variable in procedure q. So, the nothing comes there; however, if it is there then they can be defined in this portion, then it this begin is for the body of procedure q. So, procedure q starts at this point; so procedure q starts at this point and then it gives a call to procedure r and then it is the end of q then this is the begin of procedure p.

So, procedure p is starts at this point, it gives a call to procedure q and that is the end of p and you have got this main program, which is calling the procedure p. So, now this is the situation like. So, environment with local procedure so, this is the situation like so, current so, we have got this. So, this is we will need a concept of access link. So, let us explain that first, then will be coming back.

So, for supporting local procedures variables may have various scopes. So, like say suppose, I have got a variable reference here. So, this suppose I have got a variable reference within a procedure, that gives us like this, that say here, I have got a reference to some variable like here, I have got a reference to x.

Now, this x assigned as 2 now how are you going to handle this? Like when you are searching for this procedure for this variable x. So, you do not find the definition here. So, you do not find the procedure of the x definition here. So, you need to go up and look into the nested, next higher level procedure that is procedure r. Now, here after coming to this sorry, in procedure p.

So, after coming to procedure p, you find that the variable x is defined. It may so happen that this x is not defined even here; so x is defined globally. So, that way there is a hierarchy of say hierarchy of this frames into which you need to look for and then and that is defined by the scope of the language.

So, it here it is said that if some in this particular language. So, it is assumed that if something is defined here so, that will be visible to entire program, if something is defined within p. So, that will be defined only within p so, it is visible to this up to this much this x will be visible ok, because that is the begin of the end the p ends are this point. So, up to this much this x is visible; so how to take care of this situation?

So, for that matter I need to know in which sequence, I should search the variables. So, this is so, this x in this particular case, you see that x is not local to the procedure, at the same type x is not a global variable. So, this is a non local, non global type of situation.

So, it is defined in some higher level procedure, but it is not defined here ok. So, how to handle this situation so, that is actually told here; so this is for supporting a local procedure. So, variables may have various scopes, so that may be there and to determine the definition determine the definition to be used for a reference to a variable, it is needed to access non local non global variable.

So, that is that is important, that is at a typically a type of thing that we are coming across and these definitions are local to one of the procedures, nesting the current one and need to look into the activation records of nesting procedure. So, we need to go up by one or several levels for getting the corresponding definition.

So, now how to do this traversal? So, that is important. And for doing this we keep one extra bookkeeping information called access link. So, this access link comes important here. So, another so, we have got a frame pointer and control link previously, which was storing the frame pointers. Now, we will see that one access link will be in introduced so, which are going to keep the note of this next variable, next procedure, or next nesting procedure.

So, how is access link works? See you so, this is the situation. So, here at currently at procedure r, we are currently at procedure r so, how did you come to procedure r? So, this is the main this is the activation record for the main. So, there is no access link because it is at the highest level only then at procedure p, we have got one access link that points to the frame of main ok. So, this, because if I do not find any definition in procedure p from the nesting part. I know that if this definition has to be searched for a global variable.

So, that way it is pointing to the activation record of main. Now, when you are at procedure q so, from p we have given a call to q and when did when we called this q so this next frame got created. So, this frame got created and in this frame we have got so, in this frame we have got this thing, your, this frame pointer frame pointer is there, but is access link points to the this activation record for p. So, and at the next level from p your, from q we are giving a call to r.

So, if we are within r then this access link points to the frame corresponding to q. So, that if something is not found in r. So, it will be referring to procedure q, qs definition and if something is it is not found here, also then it will find it here. Now, you see here what has happened is that so this procedure p, within procedure p we have got procedure q and within q we have got procedure r, if that was not the situation then of course, this will not happen like this.

Like say, if the situation was something like this, so you have got this procedure p ok. So, we have got some definitions and it is giving a call to procedure q, but it is not defined inside this. In that case so, if this is procedure q, in that case that the access link of procedure q will not point to the frame for p, but it will point to the main routines, the access link for main routine, because q is not nested within p.

So, in this case what was happening is the q is nested within p. So, if in q we do not find some definition, we can look into the procedure p is variables ok, but if in this particular example. So, p and q they are two different procedure; there is no nesting of p and q.

So, even if p calls q so, so, q cannot use the variables that are local to p as its variable ok. So, what you what will be the happening is that if some in some variable is referred here, x is referred here so, it will not look into the activation record for p for getting and by getting the location for x, but it will look into the as into the global variable. So, this will be more clear as we see more and more examples.

Now, so this so this is the situation, when the current procedure is r and to locate the definition of x, it has to traverse through the activation records, using access links and when the required procedure containing definition of x is reached. It is accessed by a offset from the corresponding frame pointer.

So, in this case from for r, it will go to q and so, at this point so, in the, in procedure r, there is a reference to x. So, when trying to generate code, for that it will not find the reference x. So, with this access link tells me that you have to go to the activation record for q. So, it will come here, it will search for x here and again it will not find it, then it will go up by one more level and it will come to this point and then it will search into this local variables of this p and then it will get x. So, it has it will traverse by the access links to get the corresponding variable.