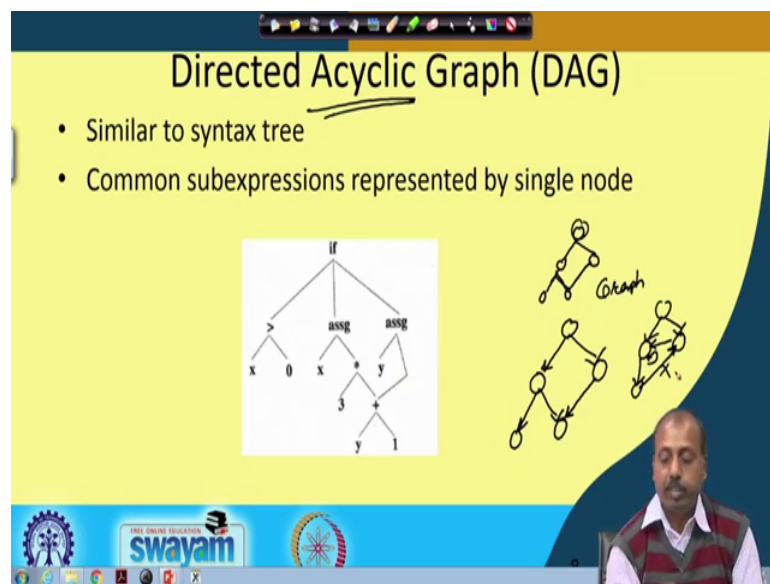


Compiler Design
Prof. Santanu Chattopadhyay
Department of E & EC Engineering
Indian Institute of Technology, Kharagpur

Lecture – 51
Intermediate Code Generation (Contd.)

The next high level representation that we will be looking into is known as directed acyclic graph.

(Refer Slide Time: 00:21)

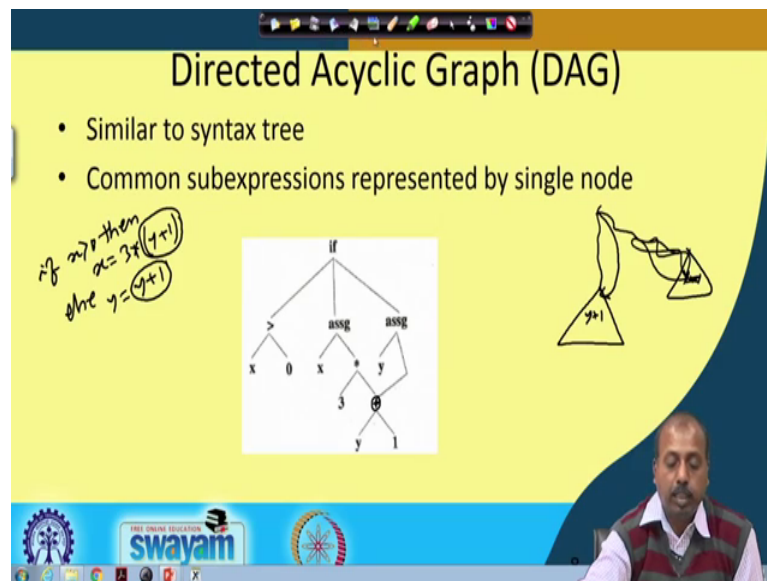


So, the previous representation that we had that was a tree. So, syntax tree and now this is a graph and you know that as a data structure the difference between tree and graph is that, in case of tree every node has got a unique parent node. And one so, it so that parent is only one every node has got at most one parent the root node does not have any parent, but otherwise like if this is a tree. So, the tree is always like this.

So, every node will have one parent accepting the root node which does not have any parent, but it is never the situation that one node has got two parents. So, that if that thing occurs then this is a graph and we call it a directed acyclic graph where this ages are directed so, from root node. So, it has got children so left and right children. Similarly from here so this is there. So, and then at some time so, if it is a graph then this type of ages will also be coming and it is not cyclic.

So, this is an acyclic graph. So, we do not have a situation where say like this. So, you do not have a situation like this so, where it creates a cyclic so, this is not there. So, this type of representation directed acyclic graph representation so, that is also used in many such intermediate language. So, it is similar to syntax tree in which common sub expressions they are represented by a single node.

(Refer Slide Time: 02:05)



So, like in the previous example that we had looked into so, it was like this that if x greater than 0, then x equal to 3 into y plus 1 else y equal to y plus 1. Now in this case so, this y plus 1 part so this expression part so this is becoming common ok. So, if you look into the parts tree so, in different portions of the parts tree I was x I was evaluating this y plus 1. So, one in the then part and another in the else part, so, in both the parts so we were evaluating that. So, in case of a directed acyclic graph; so, what is done is this common express common sub expression. So, they will be represented by a single node.

So, we do not do I do not keep a separate point a like this. So, this also points to this one only. So, that way it is helping like say here y plus 1. So, this node computes y plus 1 so, it was used in this expression also that 3 into y plus 1 and it is used in this statement also like y equal to y plus 1. So, in both assignment and both the assignment statements, so, this y plus 1 expression was common and that is taken out.

So, that gives rise to a graph representation and we will be we can go for a directed acyclic graph representation for that. So, that is one type of representation.

(Refer Slide Time: 03:33)

P-code

- Used for stack based virtual machines
- Operands are always found on the top of the stack
- May need to push operands to the stack first
- Syntax tree to P-code:

Handwritten notes:

Computer architecture

- Accumulator based
ADD B $\Rightarrow A \leftarrow A+B$
- Stack based

Syntax tree diagram:

```
graph TD
    Root((*)) --- E1[E1]
    Root --- E2[E2]
    E1 --- E1L[E1L]
    E1 --- E1R[E1R]
    E2 --- E2L[E2L]
    E2 --- E2R[E2R]
```

Code to evaluate E1 and E2:

Code to evaluate E1
Code to evaluate E2
→ Mult

OR

Code to evaluate E1
Code to evaluate E2
r0 = pop
r1 = pop
r2 = r0 * r1
push r2

Stack diagram:

pop
pop
mult
push

Then there is another high level representation which is known as P code. So, this is used for stack based virtual machines. So, what is a stack based virtual machines? So, virtual machine means the machine does not exist in actual ok. So, this is a conceptual machine you can say may be there are some machines which are built around this concept, but it is, but the basic idea is that this is a stack based machine.

So, like if you look into this computer architecture, then there are different types of architectures that we can think about or we can come across one is known as accumulator based architecture. So, in accumulator based architecture what happens is that all the arithmetic logic operations that you are doing one of the operand and the result.

So, that is always the accumulator like we have got instructions like add B which in mean that the accumulator A, we will get A plus B ok. So, this accumulator becomes a very important register and that way if all the operations arithmetic logic operations the accumulated is a part of it.

On the other hand this stack based machines, so, it will assume that we do not have any such registers or things like that. Rather there is a stack and whenever any operation is needed to be done. So, it will assume that the operands are always available in the stack. So, any operation to be done, so, it will be taking out 2 top most entries do the operation

and then it will be pushing the result back onto the stack. So, it is like this that suppose I have got say there this one say x equal to y into z .

Then what it will do it will have this it will be push putting this y and z into the stack and then when this operation has to be done so, it will take out this z and y from the stack. So, you can say that it is as if it will be doing 2 pop operations to get this y and z , then after that it will do a multiplication operation or in some sense in some cases you can say that the multiplication operator automatically it will pop out 2 topmost entries from the stack and do the multiplication on them. And then it will put the result back on to the stack. So, it may be like this that is suppose I am doing this operation.

So, multiplying two expressions E_1 and E_2 sub expressions E_1 and E_2 . So, somehow we have already have got the code for evaluating E_1 code for evaluating E_2 and then I say multiply. So, when it is there so that will mean that the operands after finishing this code to evaluate E_1 the final result is already put into the stack. So, the final result of E_1 is already available in the stack, then we have got the code to evaluate E_2 . So, after executing this is the final result will be available in the stack.

Then it comes to this MULT for a MULT instruction so, it will take out these two values to the multiplication and put the result back onto the stack or if you are looking into a more detailed version ok. So, that then it will look like this there is a code to evaluate E_1 code to evaluate E_2 , then r_{naught} equal to pop. So, that is why they are the top most entries popped out and it is kept in r_{naught} then r_1 equal to pop. So, that is also taken out of the stack. And then in r_2 we do this multiplication r_0 s into r_1 and then we do push r_2 . So, result is pushed into the stack so, this is called P code type implementation.

So, this is a high level representation so, where it is close to the stack based representation, but the difficulty that we have is definitely if the target is a accumulator based architecture, then converting it into target code becomes difficult so, that is there, but in still this is one of the some this is a technique for doing this code generation ok.

(Refer Slide Time: 07:53)

Low-level Representation – Three Address Code

- Sequence of instructions of the form " $\underline{x} = \underline{y} \text{ op } \underline{z}$ "
- Only one operator permitted in the right hand side
- Due to its simplicity, offers better flexibility in terms of target code generation and code optimization

Diagram illustrating the transformation of a high-level expression $x = y * z + w * a$ into three-address code:

$$\begin{aligned} t_1 &= y * z \\ t_2 &= w * a \\ x &= t_1 + t_2 \end{aligned}$$

So, next will be looking into the low level representations which are very common in this intermediate code; intermediary code generation which is known as three address code. In three address code the name came the name three address came from the fact that almost all the instructions.

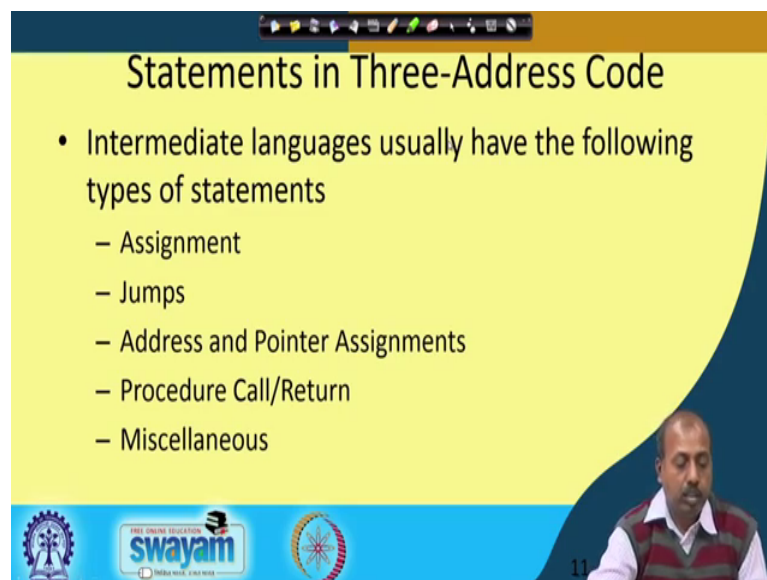
So, they will have three addresses three address components in it ok, it will have three address components in it like most of the instructions are of the form x equal to y of z . So, you need to tell now what is the operand y and what is so, you need to tell the this operands x y and z . And since they are identifiers or variables or numbers so, I will have three addresses for x y and z . So, that is why the name is for three address code and only one operated is permitted on the right side.

So, you cannot have an instruction like x equal to y star z plus P . So, that is not possible because on the right hand side we can have only one operator. So, if you have to write like this then first of all we have to write like x equal to say t_1 equal to y in to z and then write x equal to t_1 plus P . So, like here so you see that we have got an example y into z plus w into a , first t_1 equal to y into z t_2 equal to w into a , then x equal to t_1 plus t_2 . So, it is a very refined form of the complex expressions that you may have in the source language program, but at the same time which is not losing the meaning of the source language program ok.

So, everything is maintained but it is more elaborate and this individual instructions that we are having or the individual statements that we are having here so they are very simple. Due to its simplicity it offers better flexibility in terms of target code generation and code optimization. So, you see that most of the underlying processors they will support this three address format ok. So, in some machines what may happen is that you have got only two operand instructions some machines, you can have three operand instructions, but whatever it is so, it is very easy to convert these statements into that form.

So, that is why this is one of the most common formats in which this three address code is generated by the compilers. So, we will be also discussing with this three address code based in policy.

(Refer Slide Time: 10:29)



The slide is titled "Statements in Three-Address Code" in a large, bold, black font. Below the title, there is a bulleted list of statement types. The list is as follows:

- Intermediate languages usually have the following types of statements
 - Assignment
 - Jumps
 - Address and Pointer Assignments
 - Procedure Call/Return
 - Miscellaneous

In the bottom right corner of the slide, there is a small video inset showing a man with a beard and glasses, wearing a red and white striped shirt, speaking. The slide also features a blue header bar with navigation icons, a blue footer bar with logos (including Swayam and a circular emblem), and a small number "11" in the bottom right corner.

Now, statements in three address code. So, they are having this intermediate language that uses three address code, usually have these types of statements assignment jump address and pointer assignments procedure call return. And there is another category called miscellaneous. Now you see that almost all the major programming language constructs so, they are taken into consideration here.

So, we have got the assignment statement jump for go to. So, address pointer arithmetic is there and there is a the if then statement is also there. So, we will see that will come under the miscellaneous category.

(Refer Slide Time: 11:15)

The slide is titled "Assignment Statement" in a large, bold, black font. Below the title, there are two bullet points. The first bullet point is "Three types of assignment statements", followed by three sub-points: " $x = y \text{ op } z$, op being a binary operator", " $x = \text{op } y$, op being a unary operator", and " $x = y$ ". The second bullet point is "For all operators in the source language, there should be a counterpart in the intermediate language". To the right of the text, there are handwritten notes in black ink: " x, y ", " $x = \text{not } y$ ", " $x = y \text{ and } z$ ", and " $x = \ominus y$ ". At the bottom of the slide, there is a video inset showing a man with a beard and glasses, wearing a striped shirt, speaking. The background of the slide is yellow with a blue border at the top and bottom. The bottom border contains logos for "swayam" and other educational institutions.

- Three types of assignment statements
 - $x = y \text{ op } z$, op being a binary operator
 - $x = \text{op } y$, op being a unary operator
 - $x = y$
- For all operators in the source language, there should be a counterpart in the intermediate language

Assignment statement which forms the hard core part of this three address instructions. So, there are three types of assignment statements; one is like x equal to y of z . So, operator being a binary operator sometimes, we can have an unary operator say like there can be an operator like if x and y are Boolean variables.

Then I can have x equal to not of y ok. So, in that case so this not is an unitary operator so it takes only operand. Whereas, if I say x equal to y and z then this and is a binary operator like this or so, in arithmetic domain, so, we have got this unary minus x equal to minus of y , where this minus is a unary minus. Whereas, if I write like x equal to y minus z then these minus is a binary minus though so, they are meanings are totally different.

So, though both of them appear to be minus symbol, but they are different. So, in if you looking into any real compiler design task, then you will find that it is given to the lexica and analyzer to differentiate between these two situation in some one case it returns unary minus as a token in the second case it returns minus as a token.

So, that way sometimes it is a bit confusing and the third type of assignment is x equal to y . So, there is no operation in the value of y is assign to x , for all operators in the source language they are should be a counter part in the intermediate language. So, that is mast because otherwise you will not be able to generate all you will not be able to have easy code generation into intermediate code. So, you have to think about converting complex

source language operators into simpler operators of this three address code language three address code. And that is that often brings laws of optimization.

So, that way it is not advisable that we should have a three address code three, this intermediate language such that the less number of operator less operator said they are then the source language operators.

(Refer Slide Time: 13:45)

The slide is titled "Jump Statement" and contains the following bullet points:

- Both conditional and unconditional jumps are required
 - goto L, L being a label
 - if x relop y goto L

Handwritten notes on the slide include:

- goto L1
- goto L2
- if x > y then goto L1
- a = b + c
- e = b - g
- else a = b - c
- e = b + g
- L1: a = b + c
- L2: e = b - g
- L2: i

A video inset in the bottom right corner shows a man speaking. The Swayam logo is visible in the bottom left corner of the slide.

So, once that is said so, will be looking into this operators some more statements of this three address code one is known as the jump statement. So, we have got this goto L where L is a level so, you can have so this L is again a hypothetical one because, this is for this is for just an intermediate representation ok.

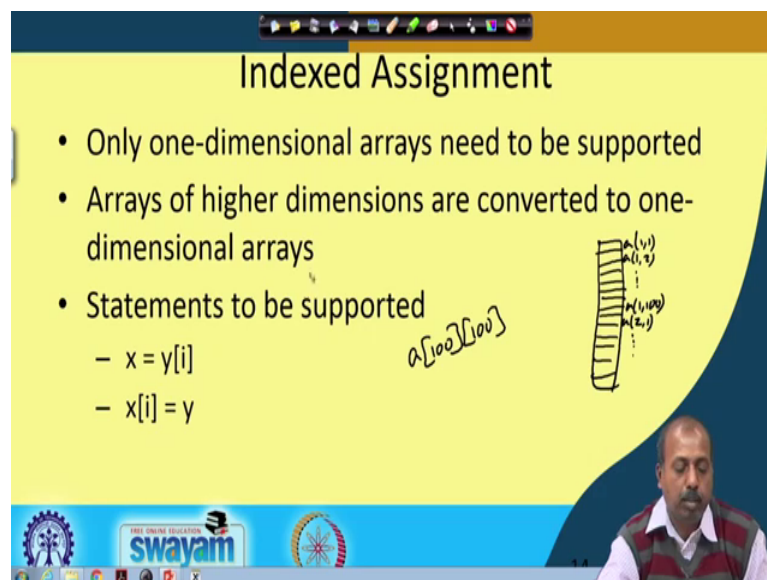
So, in the so I can just use any symbol here as L like in a program sorry so, any program so you can have at this point say go to L so, you go to say L 1 and this L 1 may be a level here some statement may have L 1 before that so, that will mean that this level is L 1. So, when it is assumed that when this statement will be executed so, it will be the control will be coming to L 1. Similarly there can be a go to L 2 and L 2 may be somewhere here. So, this way so this goto L is a generic go to statement of to the level L. And we have got this conditional go to, so, if x relational operator y go to L so ok.

So, this is the so, if then else type of statement so, if x relop y go to L. So, if this statement is there so, I do not need any separate if then else statement for example, if I

tell you like this that if x greater than y , then a equal to b plus c , then say e equal to f minus g else a equal to b minus c , e equal to f plus g say something like this. Then I can do it like this that I can I do it so, I can write like this. So, if x greater then y go to L say L 1. And then here I write like a equal to b plus c e equal to f minus g , then a go to L 2 and this is my L 1.

So, where I write like a equal to b minus c e equal to f plus g and this is my L 2. So, you see that in this language I do not need the then and else part of the if statement. So, if then else type of constructs they can be converted into this a if condition goto some level type of statements. So, that is done very often. So, we will see that this jump statements can be useful for doing this.

(Refer Slide Time: 16:39)



The slide is titled "Indexed Assignment" and contains the following content:

- Only one-dimensional arrays need to be supported
- Arrays of higher dimensions are converted to one-dimensional arrays
- Statements to be supported
 - $x = y[i]$
 - $x[i] = y$

Handwritten notes on the slide include $a[100][100]$ and a diagram of a 2D array with elements $(1,1)$, $(1,2)$, $(2,1)$, and $(2,2)$.

So, it also supports some array type of structure because array is now so, common in most in many of the almost all the programming languages. So, if you do not support array as a basic operation in the three address code, then it becomes very difficult to generate the corresponding code in the target machine. So, or so it becomes very difficult for converting the source language array indices to the intermediate code, so, only one dimensional array is supported.

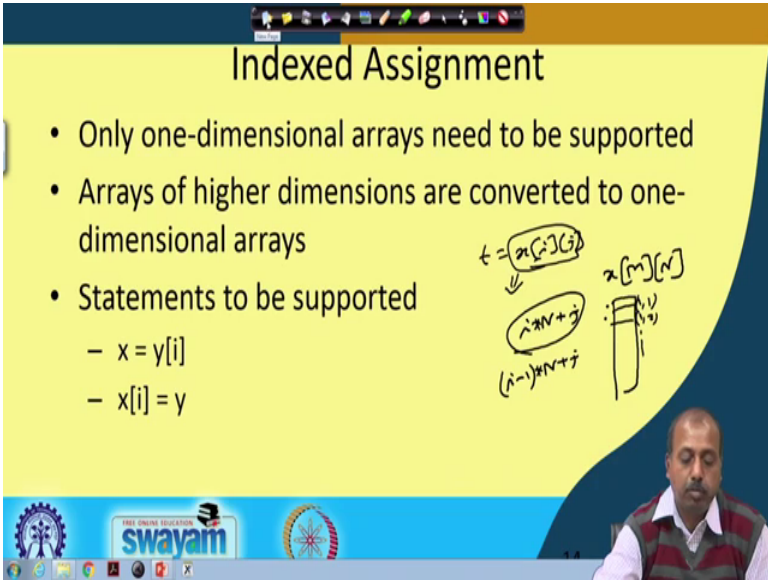
So, in your source language they are may not be any restriction on the number of dimensions, but ultimately what is happening is that see your memory on to which you will be storing the will be storing this array so, that is one dimensional in nature. For

example, if I have got a 2 d array a 100 by 100, then also I will be storing them in this fashion say a 1 a 1 2 up to say a 100, then a 2 1 so, I will be storing in them in this fashion ok.

So, I do not need to really have a 2 d representation, because while I am talking about the target code. So, target code it will be accessing memory only in a one dimensional fashion. So, even if my target machine supports this array access. So, there is no point having multidimensional array in the target code, so, that the target code will not support multidimensional array.

So, this conversion from this multidimensional array access to single dimensional array access, it is better done at the intermediate code generation phase itself. So, that at the target code generation stage so, we do not be bothered about this error dimensions and also. So, there we can concentrate more on other optimizations. So, arrays of higher dimension so, they are converted to one two one dimensional arrays so, that makes it the so, this conversion code has to be generated by the compiler.


(Refer Slide Time: 18:53)



Indexed Assignment

- Only one-dimensional arrays need to be supported
- Arrays of higher dimensions are converted to one-dimensional arrays
- Statements to be supported
 - $x = y[i]$
 - $x[i] = y$

$t = x[i][j]$
 \downarrow
 $i \cdot N + j$
 $(i-1) \cdot N + j$

$x[N][N]$


So, it will do that conversion and so, I like I can have definitely have a statement in my source language program like say t equal to say x i j. And then in the three address code I should have the appropriate code so, that this x i j so this is converted into some one dimensional array.

And you know that based on the size of these array the dimensions of this array. So, you can very easily compute the corresponding index at which you have to refer to so, that can be that is done in fact, it is if the psi, if the array is of dimension say capital M by capital, then each x axis is it is having say n such entries.

So, this i into N plus j . So, if you go to that so this if you are storing this x ray, in this fashion $1 \times 1 \times 1$, then $x \times 1 \times 2$ in this fashion, then you can do it in this way. So, that way it will be so, if this is this is not i this is i minus 1 i minus 1 into N plus j .

So, for example if it is 1×1 then i this will be this will come to this first lot. So, if it is 1×2 that will come to the second slot like that. So, that way it will be doing the conversion. So, any data structure book on this array so, that will discuss on that. So, this arrays of higher dimension they can be converted to one dimensional array and the compiler has to embed this code into the three address code that is generated ok.

So, this we will see that it is exactly what is done in the code generation phase ok. So, this will be done so, this statements to be supported are only one dimensional arrays x equal to y or $x[i]$ equal to y . So, these are the two star type of statement that we need to support.

(Refer Slide Time: 20:49)

Address and Pointer Assignments

- Statements required are of following types
 - $x = \&y$, address of y assigned to x
 - $x = *y$, content of location pointed to by y is assigned to x
 - $x = y$, simple pointer assignment, where x and y are pointer variables

$x = *p \Rightarrow x = x + 1$
 $x = *p$

Now, address and pointer assignment. So, this is another very important issue. So, there is may be a question like whether we should support this pointers or not this address

arithmetic or not, but this is required because many programming languages they support this address pointers and all.

So, if it is not supported then it will become very difficult to convert those statements like suppose for example, in c language so you have got this integer pointer `int star P` and at some point we have got a statement like `a equal to start P`. Now, how are you going to handle the how are you going to write it in terms of some lower level language program, in which this pointers are not supported it pointers are not supported, it is really it is almost impossible to write it in the proper with proper flavor.

So, we can always do something so, that it will give you some other representation which will not be truly a pointer, but something close to there, but you will never get a true pointer there. And the underlying machine so, most of the underlying processors the process, they support this type of pointers and all so, they have they support indirect access. So, at the architectural level pointers are beings supported.

So, the machine code that we are; that we are going to use they are going to support pointers. So, if the intermediate could does not support pointers, then of course, that is of no use. So, that is why in intermediate code also we need to support pointers. So, statements required for pointers are like this that `x equal to ampersand y`. So, address of y is assigned to x, then `x equal to start y` the content of location pointed to by y is assigned to x and `x equal to y` so, simpler pointer assignment so, x and y both are pointer variables so, you write like `x equal to y`. So, it does not tell anything about this implementing pointers and all.

So, it may be in some if you are high language supports that. If it supports that pointer arithmetic in terms of say making `x equal to x plus 1` and things like that. So, if this sort of things are supported then you have to have those features also incorporated into the three address code.

But for our understanding we assume that our pointers are simple so, we do not have any pointer arithmetic supported only pointer assignment can be there. So, in that case this representation is good enough ok. So, if you are in c type of language you have this pointer arithmetic which is mostly used for accessing arrays using pointers, but that may not be necessary in some other programming languages.

So, that way we so. For our discussion, so, we will not take this pointer arithmetic into consideration; we will take them simply as pointer assignment.

(Refer Slide Time: 24:03)

Procedure Call/Return

- A call to the procedure $P(x_1, x_2, \dots, x_n)$ is converted as
 - param x_1
 - param x_2
 - ...
 - param x_n
- A procedure is implemented using the following statements
 - enter f, Setup and initialization
 - leave f, Cleanup actions (if any)
 - return
 - return(x)
 - retrieve x, Save returned value in x

Handwritten notes and diagrams:

- A stack frame diagram showing parameters x_1, x_2, \dots, x_n .
- A call sequence diagram: $P(x_1, x_2, \dots, x_n)$ calls $P(\dots)$, which calls $P(\dots)$, and so on.
- A diagram showing the return flow: $P(x_1, x_2, \dots, x_n)$ returns to $P(\dots)$, which returns to $P(\dots)$, and so on.

Procedure call or return so, this is of course, important. So, a call to procedure P so, there are two part in it. So, we have to pass all these parameters so this way the this is the call for handling for doing this parameter passing. So, this $x_1 x_2$ up to x_n so, we call say that they are all parameters ok, so, this has to be done.

Now how these parameters will be implemented in the target language, so, that is a question. So, you do not answer that at this level, but we keep a note that this $x_1 x_2 x_n$ so, they are parameter. So, if we have already seen that in runtime environment management so, we need to create the activation record.

So, once this parameter statements are found so, this compiler we will know that the when it is generating the target code that there will be activation record created and these are the parameters going to be used in the activation record. So, when the activation record we will get created in that this $x_1 x_2 x_m$ so, they will be given space. So, you will know how big and activation record you need to make, once you know this parameter. So, three address code is not for execution. So, we just keep a we it is just for keeping a note that what will be required at the target level.

So, that is what is kept here so, this is x_1 to x_n . So, those parameters are kept and to proceed to a procedure is implemented using the statements like enter f level f return and return x retrieve x etcetera. So, enter f is so if I have got a procedure call. So, if this is the main routine and somewhere here I have given a call to procedure P with parameters like $x_1 x_2 x_3$.

Then what it will do. So, it will put this param statements param x_1 param x_2 param x_3 like that. And then it will tell enter P. So, enter P so this will be; this will be of so, when if this is enter P statement comes so, by this time the activation record has already been made so, this activation record will be pushed into the stack and then it will be branching to the subroutine P.

Similarly, this leave f so, this is for the cleanup actions. So, that will be called by this callee routine. So, there so, that will be done by the callee routine, so, it will be cleaning up the this activation record area and all. Similarly this return x retrieve x so, this so this return means there is no return value. So, it will just return from the procedure. So, this return values are not there, but in return x there is a written value. So, as a result if you have return like something like say equal a equal to P so, $P x_1 x_2 x_3$.

Then after returning so, this whole procedure should have some return value x that should that will be assigned to a. So, this return value is needed so, that is obtained by return x statement. So, the return value of P is taken into return x. So, if we if this is the procedure P. So, at the end of it you can have a return x statement, that will do return. And this retrieve x so, this is will also save return value in x so, that is also there.

So, with these are actually synonymous so many so you can use any of these statements. So, thus we see that the procedure call return. So, basic procedure call return is also implemented in a three address code. So, that we can also implement this procedure calls sub subprogram calls and returns. So, this three address code that we is not very simpler very low level at the same time it is not very high level also. So, it is exactly we can say it is exactly at the middle of that two end the source language and the target language.

So, it is almost at the middle of the thing so, that it is equidistant from both the sides. So, that makes it interesting because if you want to target two different target processors. So, you can put half of the effort you can say in some sense half of the effort of total

compiler writing for generating the generating a new compiler for the second target machine.