

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

Lecture - 58
Intermediate Code Generation (Contd.)

So, another very important statement that we have in many programming languages is the Case Statement or Switch Statement.

(Refer Slide Time: 00:21)

Case Statements

```
switch(E) {  
  case c1: ...  
  ...  
  case cn: ...  
  default: ...  
}
```

Implementation alternatives:

- Linear search for matching option
- Binary search for matching case
- A jump table
- Linear or binary search may be cheaper if number of cases small, for larger number of cases, jump table may be cheaper
- If case values are not clustered closely together, jump table may be too costly for space

Handwritten notes:

- switch (---)
- case 0: ...
- case 1: ...
- default: ...
- Diagram showing a box labeled 'E' with a branch 'E + 1 = 4' leading to 'case 1'.

So, in typical case statement we will look something like this. We have got a for example, in the C language we have got this switch statement switch, some Boolean some expression and then we have got different cases, so case 0, case 1; so like that there can be different cases and there is a default case also.

Now, as we know that this case statement can be modified into some if-then-else a branch a bunch of if-then-else statement. Like first we have the code for evaluating. So, if we if we take this as the format first you have the code for evaluating E and then you have this type of code; so if the final value is available in E dot plus, so if E dot plus equal to c 1. So, if whatever be that E dot plus equal to c 1, then go to some code else again another if etcetera. So, this way you can generate a set of nested if-then-else statements and you can generate the corresponding case statement code.

So, so you can have different implementation alternatives. Like you can have a linear search for the matching option; like for after you have you evaluated E you can have a you can have a search and you can say that if this particular number is matching, if the particular case value is matching with the value of the expression they will go to that option, so that is the linear search type of matching.

You can have a binary search for matching case because if we assume that this cases, so they are all going in ascending order then I can do a binary search after evaluating this into some temporary t. So, you can search for t into this alternate values alternative values c_1, c_2 up to c_n . So, try at the middle and then that is a binary search. So, if it matches with t that is fine, if it is less than t, if t is less than this the middle value then you search in this portion of the area otherwise you search in the other part of this array. So, that was the binary search principles. So, here also you can have a similar such principle.

So, so after that you have some sort of a jump table. So, what is a jump table? So, jump table structure is like this; so I have got a table, ok.

(Refer Slide Time: 03:00)

Case Statements

```

switch(E){
  case c1: S1
  ...    S2
  case cn: S3
  default: ...
}

```

Handwritten notes:
 $0 \rightarrow \text{max} \downarrow 10,000$
 $10,000$
 max

Implementation alternatives:

- Linear search for matching option
- Binary search for matching case
- A jump table
- Linear or binary search may be cheaper if number of cases small, for larger number of cases, jump table may be cheaper
- If case values are not clustered closely together, jump table may be too costly for space

Handwritten notes:
 $\text{cost } 1, 5, 100, 250, 10,000$

Handwritten notes: values table index

c_1	x_1	s_1
c_2	x_2	s_2
c_3	x_3	s_3
\vdots	\vdots	\vdots
c_n	x_n	s_n

So, in this table there are two columns. So, one column is the value of this switch value of E and this is the three-address code index, three-address index. Now, in the code part I have got. So, here if I say that this block is s 1 this block is say s 2 like that. So, in the code file I have got this codes likes. So, here I have got the code for s 1, here I have got

the code for s 2, s 3 like that. Now, if this start address is say x 1, so this start address is x 2, these start of set is x 3 like that. So, you can have a table where c 1 is x 1, c 2 is x 2, c 3 is x 3. So, you can have this type of table.

Now, so, this we call a jump table because, so by consulting this table you can figure out where to jump, ok. Now, we have to search through this table to get the jump target that is x 1, x 2, x 3 etcetera. Now that is what I can search I can search using a linear search method I can try match with all these values one by one or we can do a binary search as I was telling. So, you can (Refer Time: 04:19) back the middle of the table, see whether the value is equal or less or greater and accordingly you can do that search. And as you know that linear search this has got a complexity of order n if there are n alternative cases, whereas the binary search we will have a complexity of order log of n for n number of cases.

Now, this jump table structure is good and this linear or binary search is cheaper if number of cases is small, ok. However, for this for this larger by larger number of cases, so jump table may be cheaper because; so if this, so other implementation that you can do is that you can just instead of doing the jump table instead of keeping all these values here directly, so you can use another type of jump table where it is like this. I do not keep the value of this c 1, c2. So, they are implicit the values of c1, c 2 are implicit. So, suppose this value of this expression it can go in the range from say 0 to say some maximum max.

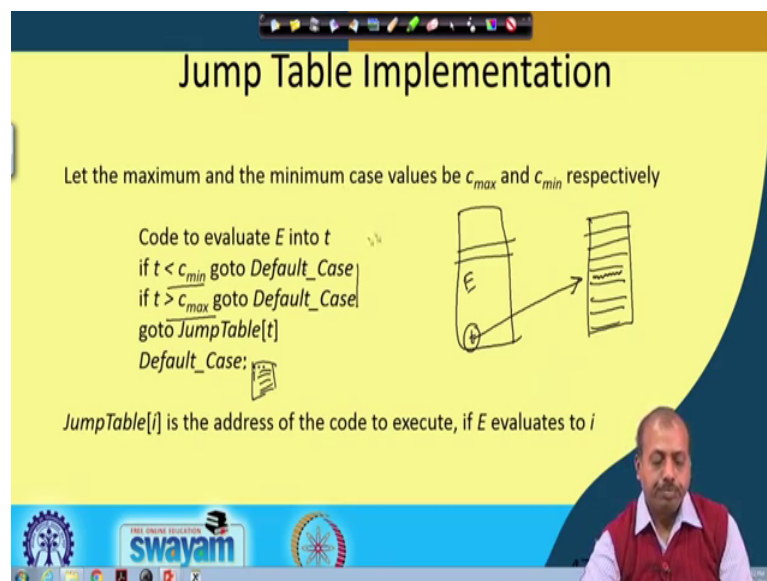
So, I have got a table where this indices they run from 0 to max, fine. And then at so assuming that there is a case for case 0 and that is x 1, so I write x 1 here. So, for 1, if the value is 1 suppose the jump target is x 2, so I write x 2 there. So, for 2 it may be x 3, for 3 it may not be there, case 3 may be absent, case 4 maybe something say x 10. So, like that. So, there are, so now what happens is that you do not need to do a linear search or binary search at all.

So, once you have evaluated the expression E, so you can use it directly to index into this table. And say this particular E t s; so, suppose this E evaluates to this particular index, so you take the jump target from here and go to that particular address. So, that way we can have a very efficient jump table implementation rather than searching through the searching through the index values.

Now, this is good if the number of cases are number of cases are large, so that way the jump table may be cheaper. However, it may so happen that numbers are more, but this the that the values are all the cases and not present, in the sense that this max may be a very large number. So, this may be say 10,000, ok, but the actual cases that you have are having say case 1, 5, 100, and say 250, and 10,000. So, these are the only few cases. But if you are thinking about this type of implementation then you have to keep a table with 10,000 indices whereas, if you are doing a if-then-else, so this type of realization where you will be comparing with each and every location and then come to a decision, so that way the table maybe compact.

So, this is a tradeoff between this normal implementation with linear search and a pure jump table based implementation. So, depending upon the choice of the compiler designer, so they can go for either of these two options. So, depend on, so it depends on the type of programs that are being written in the system, ok.

(Refer Slide Time: 07:46)



Jump Table Implementation

Let the maximum and the minimum case values be c_{max} and c_{min} respectively

```

Code to evaluate E into t
if  $t < c_{min}$  goto Default_Case
if  $t > c_{max}$  goto Default_Case
goto JumpTable[t]
Default_Case:

```

$JumpTable[i]$ is the address of the code to execute, if E evaluates to i

The diagram shows a variable 'E' in a box with a dot, and an arrow pointing to a vertical array representing the jump table.

So, this is the jump table implementation procedure. So, if maximum and minimum case values c_{max} and c_{min} respectively. So, first you have a code to evaluate the expression E into some t. So, first we have the code in this part evaluate t and ultimately this local variable t has got this E dot place. So, t corresponds to E dot place; so we have this thing.

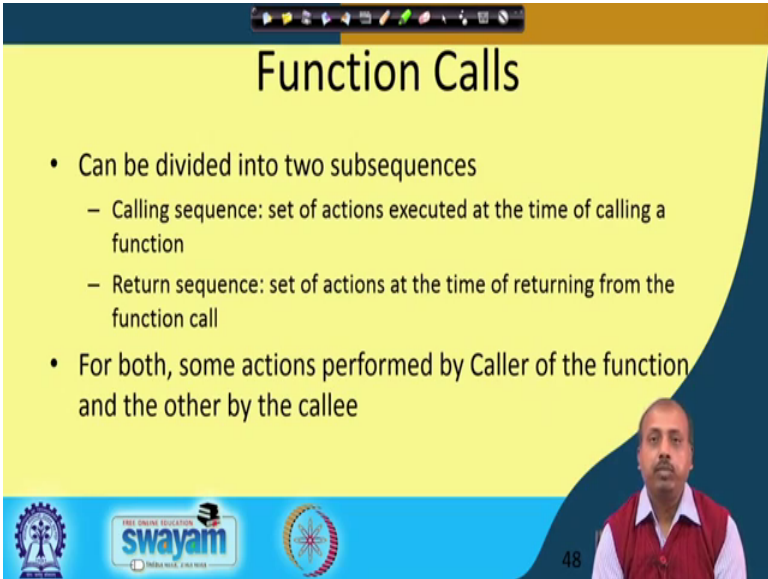
Now, there a default cases. So, if this t happens to be less than c_{min} t happens to be more than c_{max} ; that means, that that they will not match with any of the entries in the

jump table, so they will go to the default case. Otherwise it will as I as I was telling, so it will be consulting one particular entry in the jump table. So, if this is the corresponding jump table.

So, it will be consulting this particular jump table, and this t value will be used to index into this table and suppose it fix up, so this particular entry and goes there. So, that way I left this jump table t go to jump table t , otherwise this default cases (Refer Time: 08:52) this level will be there, and this go to statements are also there that will take the default statement, default part execution like this table.

So, jump table i . So, it will contain the address of the code to execute if E evaluates to i , so that is the jump table implementation. So, we can, so these are the alternatives that are done in realizing this case statements in programming languages for the three-addressed code translation.

(Refer Slide Time: 09:22)



The slide is titled "Function Calls" in a large, bold, black font. Below the title, there are three bullet points: "Can be divided into two subsequences" (with sub-bullets for "Calling sequence" and "Return sequence"), and "For both, some actions performed by Caller of the function and the other by the callee". The slide has a yellow background with a blue wavy border on the right. At the bottom, there are logos for "swayam" and "MOE, GOVT OF INDIA", along with a small video inset of a man in a red vest and a slide number "48".

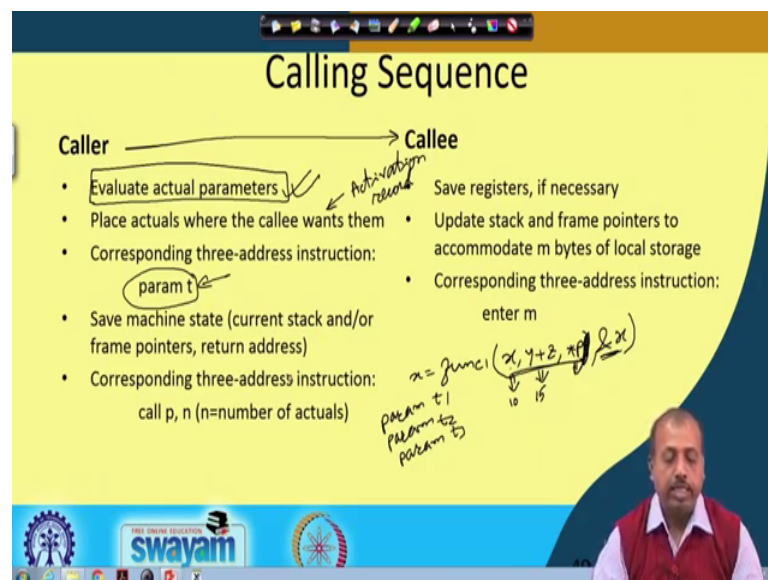
- Can be divided into two subsequences
 - Calling sequence: set of actions executed at the time of calling a function
 - Return sequence: set of actions at the time of returning from the function call
- For both, some actions performed by Caller of the function and the other by the callee

Next, we will be looking into the function calls. So, function calls they can be divided into two subsequences, so calling sequence and the return sequence. So, calling sequence is a set of actions that are to be executed at the time of calling a function and return sequence is the set of actions to be executed at the time of returning from the function call.

Though it appears to be pretty simple that you just note down the return address keep it in the stack and go back, go to the function and while coming back just take the return address from the stack, but that is not all. As we have seen that this runtime environment management, so it tells that you have to do many things for supporting recursion and all, so that way you need to do many many actions.

There is, there will be a set of actions at the time of returning from the function also at the time of going to the function also. And these actions are also distributed, in the sense that some part of the action it should be taken by the caller function and the other by the callee function. So, both of them have got definite role to play for generating this function calls, implementing the function calls.

(Refer Slide Time: 10:35)



So, what are the responsibilities? For the calling sequence, so caller so for the calling sequence the responsibilities are like this, the caller it is going to call the callee routine, ok. So, in the process before going to the callee routine, so it will be first evaluate the actual parameters, then place actuals where the callee wants them, then corresponding three-address instruction is the param t, parameter t. Then save machine status state that is current stack under frame pointer return address etcetera and the corresponding three-address instruction is called p n, where n is the number of actuals.

So, what do you mean by this? So, maybe I have I have given a call to a procedure or a function say x is equal to save function 1 and there I am parsing the parameters like x y

plus z star p like that. So, these are suppose these are the 3 parameters that I have passed. Now, this (Refer Time: 11:41) this x y plus z and star p they need to be evaluated first, ok. So, they are evaluated. So, this x maybe current value is 10, y plus z current value maybe say after evaluating this expression, so this may be 15, so that way I should have some code for evaluating this expressions, ok. And then the star p it has to be this a pointer axis, so it has to go to the corresponding location and get it. Sometimes we need to change send the address of a address of a particular variable. So, is send it like am percent x. So, then the this has to be evaluated.

So, address of a x has to be found out, so that evaluation code has to be there. So, this evaluate actual parameters though it is very simply written so, but it involves some proper amount of code and in fact, one parameter parsing the parameter that I am parsing that itself maybe another function call. So that way it becomes more complex, fine.

So, we place the, so after we have evaluated them then we have to place the actuals where the callee wants them. So, where will the callee want them? So, normally for this runtime environment management we have seen that the activation records are there, so the activation record will be created and all the all the parameters. So, they will be put into the activation record.

From the three-address code point of view, so we will call it as parameter t. So, if there are 3 parameters. So, there will be 3 parameter statement param say t 1, param t 2 and param t 3. So, but the idea is that this param t 1, t 2, t 3 when they are executed. So, they will actually be copying this values of this parameters on to the activation record slots. And after that it will save the machine state, so current state and of a frame pointer return address. So, whatever is required by the operating system, whatever is required to return from one procedure call and also it will be saving all those information into the into some proper place. So, it may be stacked, it maybe some other memory location etcetera, but it will do that.

And then, the accordingly the corresponding three-address code level, so just to model this phenomena at three-address code level we will have the statement call p n. So, it is calling the procedure p with n number of actuals n number of parameters, so that is the thing that will be done at the caller end.

At the callee end, so it will save registers if necessary. So, if the caller has not saved registers and it is necessary that this whatever register this caller had, so that should not be disturbed. So, it will first save the CPU registers into some (Refer Time: 14:33), some memory locations then update stack and frame pointers to accommodate n bytes or local storage, because at this point thus the callee routine knows like how many bytes of local storage it has all the variables that a defined in this.

So, it was not possible at the caller point because caller will not know like how many parameter, how many local variables the callee routine will have. But once we are in the callee routine, so we know the when the callee routine is parsed. So, by that time we know what is the number of parameters that are (Refer Time: 15:08) the number of local variables that are going to be there in the procedure in the function.

So, accordingly it will accommodate m m number of bytes of local storage and corresponding three-address code that will be using is enter m. So, a three-address code level keep it simple because this is this is a very machine dependent feature, so we do not want to go into much detail of it. So, I just keep a note that enough as a actions have to be taken, so that they are created in this in this fashion. So, this all the local variables are created there. So, that is about the calling sequence.

(Refer Slide Time: 15:49)

Callee	Caller
<ul style="list-style-type: none"> Place return value, if any, where the caller wants it Adjust stack/frame pointers Jump to return address Corresponding three-address instruction: return x or return 	<ul style="list-style-type: none"> Save the value returned by the callee Corresponding three-address instruction: retrieve x

Handwritten notes: $\text{jump}(-)$ and $x = \dots$

Now, for the return sequence, so it is just the other way. So, the callee routines, so it will place the return value if any where the caller wants it. So, normally as we know that in

the activation record, so there is a slot for this return value. So, depending upon that setting; so it will be putting this return value on to that particular slot. It will adjust the stack and frame pointers. So, maybe the pointer are to be reduced or increase whatever it is.

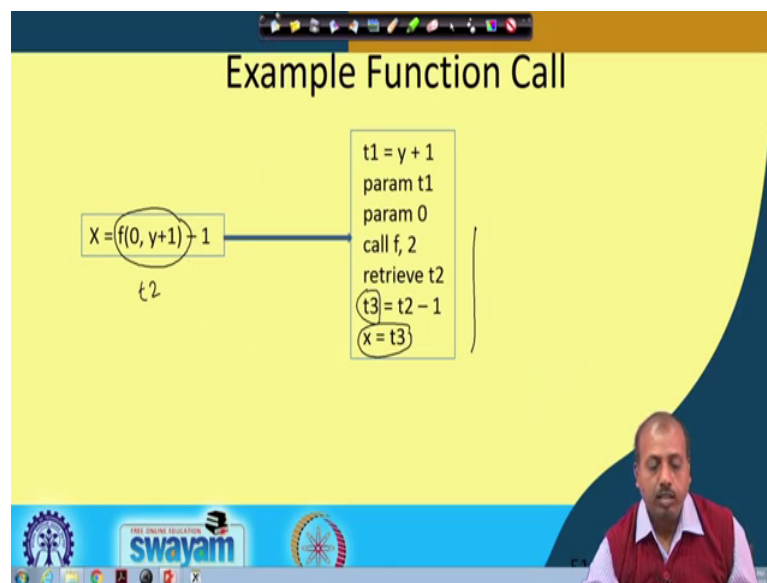
So, to it has to it maybe it has to get rid of that frame and all. So, all those things are done by adjusting the stack and frame pointer. Then it will jump to the return address. So, they say it will take the return address from the activation record and it will jump to the return address. So, the callee, that that way the callee will come back from the from it to the caller routine. And the corresponding three-address instruction is return x or return. So, this return x it will be returning the return value x and the simple returns. So, this is a void type of return. So, there is nothing there is no value that is return; so it is a void procedure call.

On the other hand if you look into the caller. So, it has got the responsibility to save the value returned by the callee. So, callee has return some value into some variable into some into some slot in the activation record, so that has to be saved into some register into some variable. For example, it may be like say x equal to sum function f 1 etcetera.

So, at the end, so this function once activation record. So, it will have this return value somewhere, so that value has to be copied into x, ok. So, for that matter, so it has to generate three-address code where some temporary will be equal to this return value and then this x will be equal to that temporary. So, something like that has to be the done. So, the corresponding three-address instruction is retrieve x.

Since, you do not know from which how this the return value will come exactly in the target machine target machine language. So, we just we just keep it this retrieve statement. So, the caller it will be retrieving the return value from the activation record. Now, how will it retrieve; so those things are not known at this point because that is very much machine specific.

(Refer Slide Time: 18:17)



So, next we look into an example how this function call will take place. So, this x equal to f 0 y plus 1, so these are the two parameters passed and minus 1. So, this first it will be evaluating the parameters. So, you see that the first statement was evaluate the actual parameters. So, first that evaluation is being made q 1 equal to y plus 1 param t 1. Then for the next one next parameter is 0, so that is also evaluated, so that its param 0.

Then it is call f and the number of actuals like how many parameters are passed etcetera, so how many local variables are there. So, all those are taken care, so call f 2. Then after calling after returning; so here I do not have the code for f in this particular piece of statement. So, I assume that code will be coming somewhere else. But after returning from that code, so the caller routine, so it has to retrieve the value from the return value from the activation record, so that is retrieve t 2 it is expected to do that.

So, it will be retrieving the value from the activation record and then t 3 equal to t 2 minus 1, so it will be retrieving it will be doing the computations after t 2 has come. So, this f 0, f 0 y plus 1, so this is available in t 2, so this t 2 minus 1, so that is made equal to t 3 and this x is finally, equal to t 3. So, this is the three-address code generated for the parameter part, for the function call sort of thing.

(Refer Slide Time: 20:00)

Storage Allocation for Functions

- Creates problem as the first instruction in a function is:
enter n /* n = space for locals, temporaries */
- Value of n not known until the whole function has been processed.
- There can be two possible solutions
 - Generating final code in a list
 - Using pair of goto statements

Handwritten notes on the slide:

```
int f1(a, b)
{
  int x, y;
  x = y + a * b;
  y = y + 15;
  x = x / (y + 19);
  ...
}
```

Handwritten notes in a circle:

```
t1 = a * b
t2 = y + t1
x = t2
```

The slide also features a video inset of a man in a red vest speaking, and a Swamyam logo at the bottom.

So, that makes it and the caller side. Now, for the callee sides there we have to do the storage allocation part, ok. So, because this is the local variables are to be created and all. So, but here is a problem because as the first instruction in a function is enter n, so enter n so it will create space for the locals temporaries etcetera and value of the value of n is not known until the whole function has been processed. So, how much local space has to be allocated, so that is not known at this point.

So, the typical situation that I am I can think about is maybe this is it this is a function the integer f 1 it has got some parameters, and into this I have got the local variables like say integer x and y and I write some expression involving x y etcetera, x equal to y plus some parameters passed a b c x plus a into b then y equal to y plus 15. Then again x equal to say x divided by say y plus 19. So, like that there maybe number of statements.

Now, one thing that is clear that for this x and y you we need space; so for then we can calculate the space. But the difficulty comes because for this expression x equal to y plus a in to b. So, this is not a single. So, the for this, so this whole competition cannot be done at ones, because three-address code we have assume that there are only two operands per instruction, for competition.

So, it has to be done like t 1 equal to a star b, then t 2 equal to y plus t 1 and then x equal to t 2, so that way this extra two temporaries t 1 and t 2 have got created. Similarly, for this statement on this statement also, so other temporaries will get created. Now, when

this temporary is are getting created, so this is not known at this point, ok. So, until unless I have seen this entire function, I do not know how many temporaries will be coming. So, it is not known the value of n is not known until the whole function has been processed.

So, how to handle this situation? So, there can be two possibilities. So, either we can generate final code in a list. So, we can say that instead of writing quadruples on to the file the code quadruples on to the file we just keep them in a list.

(Refer Slide Time: 22:42)

The slide is titled "Storage Allocation for Functions". It contains the following bullet points:

- Creates problem as the first instruction in a function is:
enter n /* n = space for locals, temporaries */
- Value of n not known until the whole function has been processed.
- There can be two possible solutions
 - Generating final code in a list
 - Using pair of goto statements

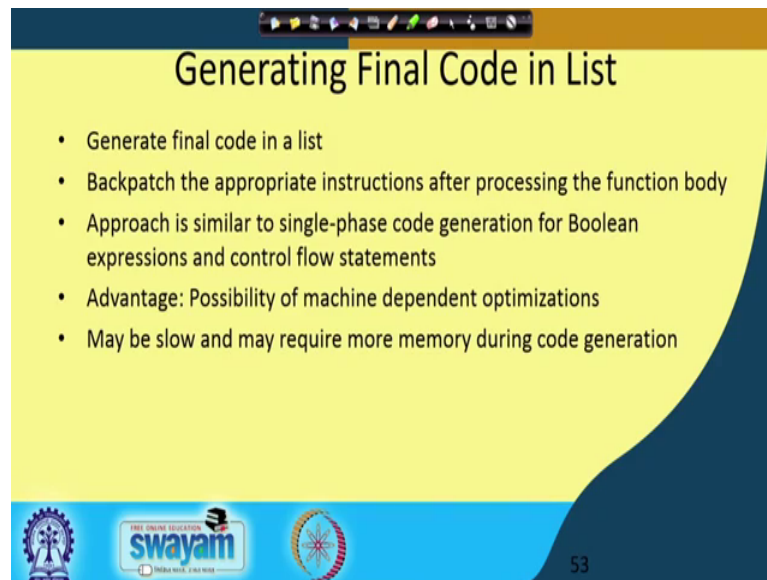
There is a diagram on the right side of the slide showing a sequence of boxes connected by arrows, representing a linked list. The first box is labeled "enter n". A handwritten arrow points from the text "Value of n not known until the whole function has been processed." to the diagram.

The slide also features a logo for "swayam" at the bottom left and a small video feed of a man in a red vest at the bottom right.

So, so this we have got the first statement as enter statement and this it will have another part which is the n part which is not known. And then I point to the next statement block, so next statement block that are generated. So, they are kept in a link list. So, at the end of this entire generation I will know the what is the value of n, so how many bytes of storage is necessary, then I will back patch, so this location with that particular value, ok. And then so then after that this whole code will be written on to the code file the intermediate code file in this order, so that is one possibilities.

So, generating the final code in a list, so that I can do all this corrections and then I can just dump on to a file. Other possibilities to use a pair of go to statements. So, use a pair of go to statements, so you can solve this problem. So, you will see how this can be done.

(Refer Slide Time: 23:43)



Generating Final Code in List

- Generate final code in a list
- Backpatch the appropriate instructions after processing the function body
- Approach is similar to single-phase code generation for Boolean expressions and control flow statements
- Advantage: Possibility of machine dependent optimizations
- May be slow and may require more memory during code generation

53

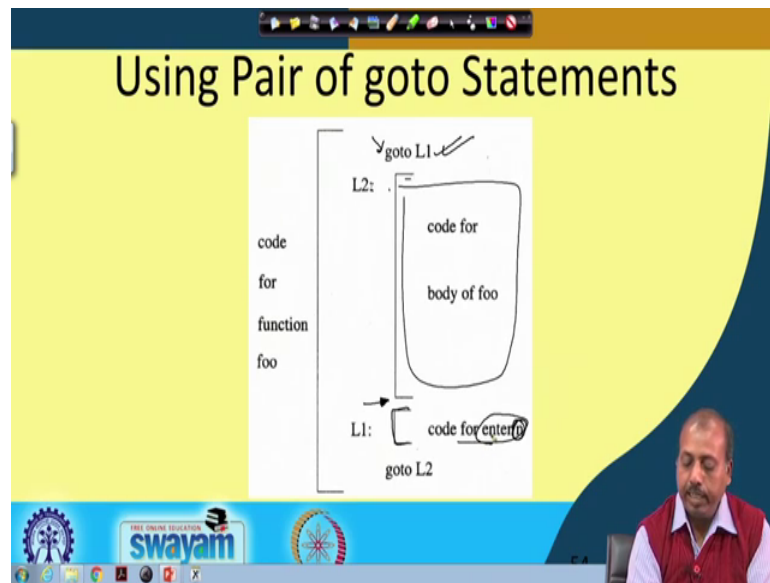
Logos at the bottom: A gear icon, the 'swayam' logo with 'FREE ONLINE EDUCATION' and 'INDIA WISE, FUTURE WISE' text, and a circular emblem.

So, this is the part which is generating final code in a list. So, generate final code in a list and back patch the appropriate instructions after processing the function body. This approach is similar to single phase code generation for Boolean expressions and control flow statements. So, in single phase generation also we have seen that we have producing the codes and after that we are doing some back patching.

So, here also I am telling the same thing you generate the code in a list and after that you do a back patching of them back patching of the code, so that is one possibility. So, it is possibility of machine independent optimization, so that is one advantage. So, we can do many optimization, so which are dependent on the machine. So, you can modify the statements accordingly, but it may be slow and may require more memory during code generation.

So, this link list type of organization as you know that as per as accesses is concerned. So, it is going to be slow because it has to travel through this dynamic memory and all, so it takes time. And the second thing is that the memory requirement is also more because the (Refer Time: 24:49) because the amount of space necessary for holding the pointer, so they will come into the picture. So, this final code generation using this approach is difficult.

(Refer Slide Time: 25:00)



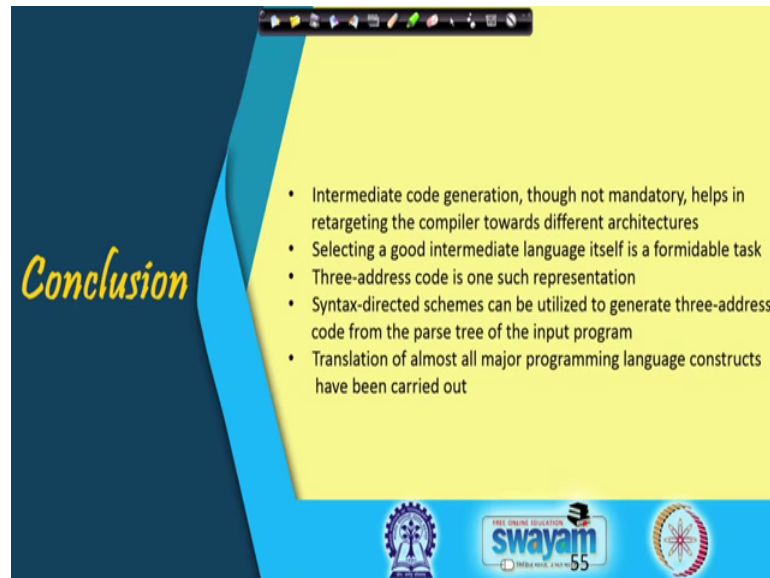
So, at the using a pair of go to. So, the code, so this is an this is an example suppose we have got a function called foo and in this function we have got. So, at the beginning of the function we generate a go to L 1. So, in the here I have put the code for enter n, so that will be put into this L 1 part. So, while generating the code. So, we just do it like this. So, we generate a level, so at this point we generate a new level and we put it, we generate a go to statement at this point. So, this is the.

Now, we generate another level L 2 and we call it like this. So, they are the code for the body of foo is presented. So, here I have got the body of the procedure. Then at L 1, so once, we are once we are at this point once the entire procedure and entire function has been generated the code has been generated we know how many temporary were used; so what is the space requirement and all. So, I can have this code for enter n. So, this n value is now known.

So, I can have the enter n thing. And after that we generate a go to for L 2, so it comes in. So, how the whole function works? As soon as it is called, so it will be starting with this go to statement it will come here. So, it will it will have the code for enter n, so that there it will be allocating enough space for the temporaries and all. And then it will come to this statement go to L 2. As a result, it will come here and start executing the body of the function.

So, this way using a pair of go to statements we can solve this local param in, local variable issue.

(Refer Slide Time: 26:48)



So, next will be coming to the conclusion; so we have seen a number of intermediate code generation mechanism. So, it is though intermediate code generation is not mandatory, it helps in retargeting the compiler towards different architectures. Now, selecting a good intermediate language itself is a formidable task because they have to be equally distance from the source language and the target language.

Three-address code is one such representation that we have seen it is very versatile and it is powerful that way. And syntax directed schemes are best used for generating the three-address code from the parse tree of the input program. And translation of almost all major programming language constructs we have seen and we can do this translation using them.

So, after this intermediate code has been generated. So, you can go to the code optimization phase, and the target code generation phase followed by code optimization. Now, the target code generation is nothing, but a template substitution. So, for intermediary code templates, so you can think about a best way to implement the target code for that and accordingly you can generate code for that. And also, if that the optimization part, so you can do some optimization at the intermediary code level itself like you can find that the same sub expression is being computed again and again.

So, they can be completed only once, put into some temporaries used there. Or maybe some competition you can find out that it is not necessary because it is may be carried out in a loop, but it is not dependent on the loop index. So, they can be taken out of the loop. So, this type of optimizations can be done at the intermediary code itself.

However, more important type of optimizations like say by putting the variables into registers and all, so that will be difficult because that will require the exact machine architecture and that can be done only at the target machine level, you cannot do it an intermediary code because we do not know how many registers we will have. So, this way, so the later part of the (Refer Time: 28:57) advanced concepts on this compiler. So, they will be discussing about this optimization issues, this target code generation issues and all. So, we will end our theory class with this. So, we will be doing some more exercises in the next classes.