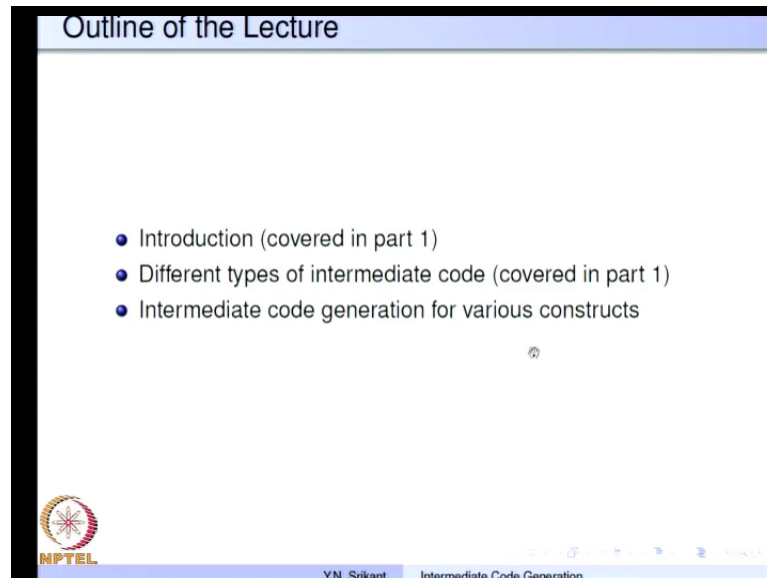


**Principles of Compiler Design**  
**Prof. Y. N. Srikant**  
**Department of Computer Science and Automation**  
**Indian Institute of Science, Bangalore**

**Lecture - 18**  
**Intermediate code generation Part-2**

(Refer Slide Time: 00:20)



Outline of the Lecture

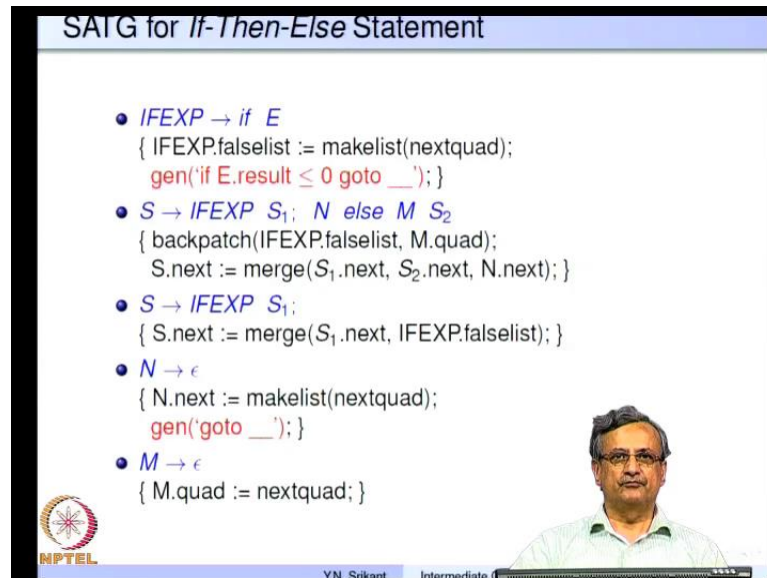
- Introduction (covered in part 1)
- Different types of intermediate code (covered in part 1)
- Intermediate code generation for various constructs

NPTEL

YN. Srikant Intermediate Code Generation

Welcome to part two of the lecture on intermediate code generation. So, in this part of the lecture, we will continue with intermediate code generation techniques for various types of constructs in programming languages.

(Refer Slide Time: 00:31)



**SATG for If-Then-Else Statement**

- $IFEXP \rightarrow if\ E$   
{ IFEXP.falselist := makelist(nextquad);  
  gen('if E.result  $\leq$  0 goto \_\_'); }
- $S \rightarrow IFEXP\ S_1; N\ else\ M\ S_2$   
{ backpatch(IFEXP.falselist, M.quad);  
  S.next := merge(S<sub>1</sub>.next, S<sub>2</sub>.next, N.next); }
- $S \rightarrow IFEXP\ S_1;$   
{ S.next := merge(S<sub>1</sub>.next, IFEXP.falselist); }
- $N \rightarrow \epsilon$   
{ N.next := makelist(nextquad);  
  gen('goto \_\_'); }
- $M \rightarrow \epsilon$   
{ M.quad := nextquad; }

NPTEL  
Y.N. Srikant Intermediate

So, to do a bit of recap last time we looked at this code generation technique for if then else statement. So, let us just go through it again. So, the production for the, if then else statement can be of the form ifexp S going to ifexp S 1 semicolon N else M S 2. And then for the if then statement it is of the form S going to ifexp S 1 semicolon ifexp expands to if expression. The reason for F G doing this, I already mentioned during semantic analysis phase. And also during the last lecture, we need to generate you know the test for this expression. So, since this is a S attributed grammar and is used with LR parsing.

We want to perform an action at the end of this particular handle just before the reduction by this handle happens, that is the reason for breaking up this particular these productions. Then we also have a couple of markers N and M here the markers are required to generate appropriate goto statements you know at these places. So, for example, after these then part of the, if then else statement that is S 1 is executed. We are actually required to jump out of the, if then else to the end of this whatever it this statement is to do that when a, the marker N is used. And when the reduction by to epsilon happens a goto statement with a an unknown label target is generated.

So, this particular goto statement is put on N dot next which is a list. So, at this point N dot next will contain the quadruple this you know this goto quadruple with its target unfilled M is required to remember the beginning of the code for S 2 this is required.


Because we want to jump at if the expression is FALSE we want to jump and execute S 2 we do not want to execute S 1 at all. So, M remembers the quadruple number of the beginning of S 2. So, that is M dot quad. So, at this point the attribute M dot quad contains the quadruple number of the beginning of the code for S 2. So, when we reduce by if E ifexp going to if E, we generate this comparison statement if E dot result less than equal to 0 goto and we do not know the a target at this point. So, we leave it as blank and this particular quadruple with its blank target is put on ifexp dot FALSE list here.

So, during this production the reduction by this production the ifexp dot FALSE list which comes of out as a production as a an attribute of ifexp is patched to M dot quad. Thereby if the result is actually FALSE in E dot less than equal to 0 is FALSE, we jump to the else part and not the then part if the result is TRUE. Then we just fall through and execute S 1 that is the idea and what is S dot next S dot next contains all the possible jumps out of S 1. So, if there is any jump with in S 1 then we need to go to the end of S 2 here after S 2 similarly this N dot quad must also now N dot next must also go to this point. And then any other jumps out of S 2 should also go into this point I already gave you an example of how this can happen during execution in the last lecture. So, we merge all these 3 lists and put it on S dot next. So, this is the way we generate code for if then else so.

(Refer Slide Time: 05:00)

### SAILG for Other Statements

- $S \rightarrow \{ ' L ' \}$   
 { S.next := L.next; }
- $S \rightarrow A$   
 { S.next := makelist(nil); }
- $S \rightarrow \text{return } E$   
 { gen('return E.result'); S.next := makelist(nil); }
- $L \rightarrow L_1 \{ ; \} M S$   
 { backpatch(L<sub>1</sub>.next, M.quad);  
 L.next := S.next; }
- $L \rightarrow S$   
 { L.next := S.next; }
- When the body of a procedure ends, we perform the following actions in addition to other actions:  
 { backpatch(S.next, nextquad); gen('func end'); }



Y.N. Srikant    Intermediate Code Generation

Let us move on and see how we can generate code for rest of the constructs here is a compound statement. So, there is begin that is the parenthesis you know left flower bracket then a list of statements  $l$  and end that is the right flower bracket. So, here we just have to transfer all the unfilled jumps from  $l$  dot next to  $S$  a is an assignment. And therefore, it does not generate any jumps. So,  $S$  dot next is any is a nil list the for the return  $E$  that is a return statement we generate the quadruple  $\text{return } E \text{ dot result}$  and  $S$  dot next will be nil. Because the control goes out of this to the calling procedure itself  $l$  going to  $l$   $l$  semicolon  $M$   $S$  is the production, which produces a list of statements so; obviously, we need the marker  $M$  to remember the beginning of  $s$ .


So, all the jumps out of  $l$   $l$  are patched to  $M$  dot quad, which is nothing but the beginning of  $S$ . And whatever jumps remain within  $S$  or either out of  $S$  will be put on  $l$  dot next. And the terminating production  $l$  to  $x$  simply copies  $S$  dot next into  $l$  dot next when the procedure body ends you know there will still be some more you know jumps which are left out. So, we perform backpatch  $S$  dot next comma nextquad and followed by generation of the statement function end function end typically is returning from the procedure. So, all the left out quadruples the, their targets will all be patch to func end there by there will be no unfilled targets at the end of the procedure itself.

(Refer Slide Time: 07:07)

### Translation Trace for *If-Then-Else* Statement

$A_i$  are all assignments, and  $E_i$  are all expressions  
 $\text{if } (E_1) \{ \text{if } (E_2) A_1; \text{ else } A_2; \} \text{ else } A_3; A_4;$   
 $S \Rightarrow \text{IFEXP } S_1; N_1 \text{ else } M_1 S_2$   
 $\Rightarrow^* \text{IFEXP}_1 \text{ IFEXP}_2 S_{21}; N_2 \text{ else } M_2 S_{22}; N_1 \text{ else } M_1 S_2$

- 1 Consider outer if-then-else  
Code generation for  $E_1$
- 2  $\text{gen}(\text{'if } E_1.\text{result} \leq 0 \text{ goto } \_\_\_')$   
on reduction by  $\text{IFEXP}_1 \rightarrow \text{if } E_1$   
Remember the above quad address in  $\text{IFEXP}_1.\text{falselist}$
- 3 Consider inner if-then-else  
Code generation for  $E_2$
- 4  $\text{gen}(\text{'if } E_2.\text{result} \leq 0 \text{ goto } \_\_\_')$   
on reduction by  $\text{IFEXP}_2 \rightarrow \text{if } E_2$   
Remember the above quad address in  $\text{IFEXP}_2.\text{falselist}$


VN. Subramanian Intermediate Code Generation

So, now let us look at a translation trace of the if then else statement, how exactly the various backpatch and gen statements are executed during a process of l r parsing. And

reduction our example is the, a example that we have already studied before if E 1. Then there is an nested if then else if E 2 a 1 else a 2 followed by the else part of the first if else a 3 and here it terminates followed by a 4 which is a different statement. So, this is this entire thing is 1 statement and a 4 is another statement and within this statement is another if then else statement. So, we have seen this already. So, let us see a I will show you only a partial derivation of this. So, S derives ifexp S 1 semicolon N 1 else M 1 S 2. So, this is the high level if then else.

So, ifexp derives if E 1 this S 1 is nothing but you know this entire if E 2 a 1 else a 2 then S 2 is nothing but a 3. So, if we expand S 1 we get ifexp 2 S 2 1 N 2 else M 2 S 2 2 where S 2 1 is nothing but a 1 and S 2 2 is nothing but a 2. So, this is the partial derivation of our, you know statement here. So, let us see how the code generation happens first of all consider the outer if then else. So, if E 1 would reduce to ifexp 1 before that the code generation for E 1 would have happened. Then you know on reduction by ifexp 1 going to if E 1 we generate the code if E 1 dot result less than or equal to 0 goto dash.

So, at this point the just after this we have generated code for the jump by testing the expression E 1 and this quadruple address is remembered in ifexp 1 dot FALSE list. So, so far we are here then the inner if then else code generation for E 2 will now happen. So, that is just before reduction to this particular handle rather the, if this particular non terminal. And another if then you know if E 2 dot result less than or equal to 0 goto dash is generated during the reduction by ifexp 2 going to if E 2. So, that is what and that is remembered on ifexp 2 dot FALSE list. So, we have so far finished this much.

(Refer Slide Time: 10:02)


Translation Trace for *If-Then-Else* Statement(contd.)

```
if (E1) { if (E2) A1; else A2; }else A3; A4;  
S ⇒* IFEXP1 IFEXP2 S21; N2 else M2 S22; N1 else M1 S2
```

Code generated so far:

```
Code for E1; if E1.result ≤ 0 goto __ (on IFEXP1.falselist);  
Code for E2; if E2.result ≤ 0 goto __ (on IFEXP2.falselist);
```

- 6 Code generation for S<sub>21</sub>
- 6 gen('goto \_\_'), on reduction by N<sub>2</sub> → ε (remember in N<sub>2</sub>.next)
- 7 L1: remember in M<sub>2</sub>.quad, on reduction by M<sub>2</sub> → ε
- 8 Code generation for S<sub>22</sub>
- 8 backpatch(IFEXP<sub>2</sub>.falselist, L1) (processing E<sub>2</sub> == false) on reduction by S<sub>1</sub> → IFEXP<sub>2</sub> S<sub>21</sub> N<sub>2</sub> else M<sub>2</sub> S<sub>22</sub>  
N<sub>2</sub>.next is not yet patched; put on S<sub>1</sub>.next



Y.N. Srikant Intermediate Code Generation

So, the code generated so far is just this code for E 1 and then a jump code for E 2 and another jump, but we still do not know the jump targets now we are here. So, code generation for S 2 1 happens and then we have N 2. So, N 2 by epsilon is the reduction that happens at this point. And we generate a goto at this point and we remember it on N 2 dot next then we have M 2 dot quad which is remembered. So, that is the label say l 1 on reduction by M 2 to epsilon and then we generate the code for S 2 2 and now this entire violate part is reduced to a statement.

So, on reduction by S 1 going to ifexp 2 S 2 1 N 2 else M 2 S 2 2 we backpatch ifexp 2 dot FALSE list to l 1 that is the M dot quad. So, this entire thing you know if the expression is FALSE we know where to go. So, we actually patch it to M 2 dot quad. So, that is this l 1. So, N 2 dot next is not yet patched, because we are suppose to jump out of this. So, that is if this is if S 2 1 is executed N 2 has generated a goto which will take you to really speaking a 4. So, we still do not know that there is a 4. So, we cannot do anything about it, it is just put on S 1 dot next which is the left hand side non terminal here.


(Refer Slide Time: 11:53)

Translation Trace for *If-Then-Else* Statement(contd.)

```
if (E1) { if (E2) A1; else A2; }else A3; A4;  
S ⇒ IFEXP S1; N1 else M1 S2  
S ⇒* IFEXP1 IFEXP2 S21; N2 else M2 S22; N1 else M1 S2
```

Code generated so far:  
Code for E<sub>1</sub>; if E<sub>1</sub>.result ≤ 0 goto \_\_\_ (on IFEXP<sub>1</sub>.falselist)  
Code for E<sub>2</sub>; if E<sub>2</sub>.result ≤ 0 goto L1  
Code for S<sub>21</sub>; goto \_\_\_ (on S<sub>1</sub>.next)  
L1: Code for S<sub>22</sub>

- 10 gen('goto \_\_\_'), on reduction by N<sub>1</sub> → ε (remember in N<sub>1</sub>.next)
- 11 L2: remember in M<sub>1</sub>.quad, on reduction by M<sub>1</sub> → ε
- 12 Code generation for S<sub>2</sub>
- 13 backpatch(IFEXP.falselist, L2) (processing E<sub>1</sub> == false)  
on reduction by S<sub>2</sub> ⇒ IFEXP S<sub>1</sub> N<sub>1</sub> else M<sub>1</sub> S<sub>2</sub>  
N<sub>1</sub>.next is merged with S<sub>1</sub>.next, and put on S.next



Y.N. Srikant Intermediate Code Generation

So far we have generated code for S<sub>22</sub> and did some back patching. So, because of that back patching this target has been filled the other 2 are still not filled now we have to take care of N<sub>1</sub>. So, we generate a goto on reduction by N<sub>1</sub> to epsilon and remember it in N<sub>1</sub>.next now we come to M<sub>1</sub> and let us say that level that quadruple level is L2. So, there is no level in practice, but it is just a number, but let us give it a level for understanding. So, this is the level which is remembered by the on reduction by M<sub>1</sub> to epsilon and that is remembered as M<sub>1</sub>.quad now we generate code for S<sub>2</sub> and finally, this entire thing you know is reduced to s.

So, on reduction by S going to ifexp S<sub>1</sub> N<sub>1</sub> else M<sub>1</sub> S<sub>2</sub> we can patch the FALSE part of ifexp<sub>1</sub> right. So, if this ifexp<sub>1</sub> is you know if this expression is FALSE then we actually have to jump to S<sub>2</sub>. So, that is remembered by M<sub>1</sub>.quad. So, that we can do at this point and N<sub>1</sub>.next is still unfilled it is merged with S<sub>1</sub>.next and both are put on S.next. So, at this point S.next contains jumps out of many points for example, it contains jumps due to N<sub>2</sub> right. And then it will contain jumps due to N<sub>1</sub> and we still have not seen a 4. So, we really cannot do anything right now.


(Refer Slide Time: 13:43)

Translation Trace for *If-Then-Else* Statement(contd.)

```
if (E1) { if (E2) A1; else A2; }else A3; A4;  
S ⇒* IFEXP1 IFEXP2 S21; N2 else M2 S22; N1 else M1 S2  
L ⇒* L1 ';' M3 S4 ⇒* S3 ';' M3 S4  
Code generated so far (for S3/L1 above):
```

Code for E<sub>1</sub>; if E<sub>1</sub>.result ≤ 0 goto L2  
Code for E<sub>2</sub>; if E<sub>2</sub>.result ≤ 0 goto L1  
Code for S<sub>21</sub>; goto \_\_ (on S<sub>3</sub>.next/L<sub>1</sub>.next)  
L1: Code for S<sub>22</sub>  
goto \_\_ (on S<sub>3</sub>.next/L<sub>1</sub>.next)  
L2: Code for S<sub>2</sub>

- 14 L3: remember in M<sub>3</sub>.quad, on reduction by M<sub>3</sub> → ε
- 15 Code generation for S<sub>4</sub>
- 16 backpatch(L<sub>1</sub>.next, L3), on reduction by L → L<sub>1</sub> ';' M<sub>3</sub> S<sub>4</sub>
- 17 L.next is empty

 Y.N. Srikant Intermediate Code Generation

So far we have generated up to this code, code for S 2. So, what; that means is the code for a 4? So, to do that let us look at the production which handles a list of statements. So, l going to l 1 semicolon M 3 S 4 so at this point this l 1 is our statement actually. So, that is the S 3 right this statement. So, S 3 semicolon M 3 S 4 so S 3 is our complete nested if then else and S 4 isa 4. So, M 3 is M 3 dot quad remembers the beginning of S 4. So, that is M 1 3 and then the code generation for S 4 happens now we reduce using this entire production l l going to l 1 M 3 S 4 and we backpatch all the jumps out of l 1. So, that is whatever comes this this goto and this goto both are actually patched to a 4 which is nothing but S 4. So, now l dot next is empty and we, because a a 4 does not generate any jumps and the process of code generation is over.




(Refer Slide Time: 15:04)

Translation Trace for *If-Then-Else* Statement(contd.)

```
if (E1) { if (E2) A1; else A2; }else A3; A4;  
S ⇒* IFEXP1 IFEXP2 S21; N2 else M2 S22; N1 else M1 S2  
L ⇒* L1 ; M3 S4 ⇒* S3 ; M3 S4
```

Final generated code

```
Code for E1; if E1.result ≤ 0 goto L2  
Code for E2; if E2.result ≤ 0 goto L1  
Code for S21; goto L3  
L1: Code for S22  
goto L3  
L2: Code for S2  
L3: Code for S4
```





YN. Srikant Intermediate Code Generation

So, here is the final code that is generated. So, this is the trace of actions that happen during the code generation process including backpatch and gen.

(Refer Slide Time: 15:20)

SAIG for *While-do* Statement

- $WHILEEXP \rightarrow \text{while } M E$   
{ WHILEEXP.falselist := makelist(nextquad);  
gen('if E.result ≤ 0 goto \_\_');  
WHILEEXP.begin := M.quad; }
- $S \rightarrow WHILEEXP \text{ do } S_1$   
{ gen('goto WHILEEXP.begin');  
backpatch(S<sub>1</sub>.next, WHILEEXP.begin);  
S.next := WHILEEXP.falselist; }
- $M \rightarrow \epsilon$  (repeated here for convenience)  
{ M.quad := nextquad; }



YN. Srikant Intermediate Code Generation

So, let us move on let us find out how to generate code for while loops. So, we need to break up the while statement into the production for the while statement into do parts for the same reason as the, if then else. So, we have WHILEEXP going to while E while M E. So, M is a marker again which denotes which remembers the beginning of this expression, why do we need this at the end of the loop? We actually have to jump to the

beginning of the code for E. So, that E is evaluated again and then that it is tested. So, at this point whileexp dot FALSE list is makelist nextquad nextquad is nothing but this particular quadruple. So, the test for E so the target is unfilled this is the exit for the while loop. So, we have no idea where we are we have to go this will known only when there is a list of statements. So, whileexp dot begin is nothing but M dotquad.

So, we remember that also as an attribute of whileexp here S going to whileexp do S 1. So, we generate we have actually finished this entire code generation for expression and S 1. So, we are here we need to back to the beginning of the while loop. So, gen goto whileexp dot begin will take care of that. So, remember here whileexp dot begin is nothing but the beginning of the expression code then all the jumps out of S 1 have to again take the control to the beginning of the while loop. So, backpatch S 1 dot next to whileexp dot begin finally, what are the jumps out of this statement S dot next equal to whileexp dot FALSE list that is the only jump out of the entire while loop.

(Refer Slide Time: 17:24)

**Code Template for *Function* Declaration and Call**

```

Assumption: No nesting of functions
result foo(parameter list){ variable declarations; Statement list; }
func begin foo
/* creates activation record for foo - */
/* - space for local variables and temporaries */
code for Statement list
func end /* releases activation record and return */

x = bar(p1,p2,p3);
code for evaluation of p1, p2, p3 (result in T1, T2, T3)
/* result is supposed to be returned in T4 */
param T1; param T2; param T3; refparam T4;
call bar, 4
/* creates appropriate access links, pushes return address */
/* and jumps to code for bar */
x = T4
  
```

So, let us now move on try to understand code generation for function declaration and function call. So, assume that there is no nesting of functions that is the norm in c as well. So, let us take a result a function called foo its type is result it has a parameter list it has variable declarations and it has a statement list. So, this is the body of the function foo. So, the code generated for the function foo would be func begin foo. So, this intermediate code is supposed to create the activation record reserve the space for local

variables and temporaries. So, then the code for the statement list is here remember there is no code that is necessary for parameter list and variables. It is just the space needed for them and the space you know offsets etcetera have already been computed then the function end. So, this releases the activation record and returns.

So, it is fairly simple you know as far as the function declaration is concern, let us look at a function call. So, here is an assignment  $x$  equal to and then a function call  $\text{bar}$  of  $p_1$  comma  $p_2$   $p_3$  3 parameters so; obviously, before the call is made we need to evaluate the 3 parameters. So, code for evaluation of  $p_1$   $p_2$  and  $p_3$  the results are in the 3 temporaries  $T_1$   $T_2$  and  $T_3$ . Then the result of the call let us say is supposed to be returned in a temporary call  $T_4$  finally, we need to actually you know assign that to  $x$ . So,  $x$  equal to  $T_4$  after the call returns will take care of this.

But there is a point that needs to be stressed here the temporary  $T_4$  that we are going to use to return. The result is now in the callers space that is the function in which this  $x$  equal to  $\text{bar}$   $p_1$   $p_2$   $p_3$  is enclosed it is not a local variable of the function  $\text{bar}$ . But it is a variable in the local space of the enclosing function for this assignment statement. So, now, the parameter code is  $\text{param } T_1$   $\text{param } T_2$   $\text{param } T_3$  these 3 will actually push the parameters onto the stack are the activation record. Then there is a  $\text{refparam } T_4$ , why did we really want to pass the address of  $T_4$  and not  $T_4$  itself we would like to accesses. You know this particular address of  $T_4$ , which is in this callers space within the function  $\text{bar}$ .

So, if we had simply said  $\text{param } T_4$  then you know we would have actually return the result within  $\text{bar}$  to a local variable. But here because it is a  $\text{refparam}$  the address of  $T_4$  is passed and whenever we assign the result within  $\text{bar}$  we would use indirect addressing. So, that the actual you know temporary  $T_4$  gets the value. So, including this result part  $T_4$  there are now 4 parameters 3 explicit and 1 implicit. So, we call  $\text{bar}$  with 4 parameters. So, this procedure call or function call is supposed to create access links push return address and jump to the code for  $\text{bar}$ . We will we are going to see the details of all this in one of the future lectures on runtime environments after the call has materialized  $T_4$  will contain the return result. So,  $x$  equal to  $T_4$  takes care of assigning the result to the variable  $x$ .

(Refer Slide Time: 21:55)

The slide is titled "SATG for Function Call". It contains the following text:

Assumption: No nesting of functions

- $FUNC\_CALL \rightarrow id \{action\ 1\} ( PARAMLIST ) \{action\ 2\}$   
 $\{action\ 1:\}$  {search\_func(id.name, found, fnptr);  
call\_name\_ptr := fnptr }  
 $\{action\ 2:\}$   
{ result\_var := newtemp(get\_result\_type(call\_name\_ptr));  
gen('refparam result\_var');  
/\* Machine code for return a places a in result\_var \*/  
gen('call call\_name\_ptr, PARAMLIST.pno+1'); }
- $PARAMLIST \rightarrow PLIST \{ PARAMLIST.pno := PLIST.pno \}$
- $PARAMLIST \rightarrow \epsilon \{ PARAMLIST.pno := 0 \}$
- $PLIST \rightarrow E \{ PLIST.pno := 1; gen('param E.result'); \}$
- $PLIST_1 \rightarrow PLIST_2 . E$   
{ #PLIST<sub>1</sub>.pno := PLIST<sub>2</sub>.pno + 1; gen('param E.result'); }

At the bottom left is the NPTEL logo. At the bottom center, it says "YN. Srikant Intermediate Code Generation".

So, now that is the code template. So, let us understand how to generate that type of code. So, again we are assuming that there is no nesting of functions. So, the function call is a very simple production function call going to *id* and followed by parameter list. So, let us embed 2 actions in between and expand them here. So, action 1 is the code for action 1 is search func *id* dot name comma found comma *fnptr*. So, we are going to check and get the pointer to the function in the symbol table. So, that returned in *fnptr*. So, call name *ptr* equal to *fnptr*. So, at this point we know that this is the function pointer this is required to process the parameter list and emit the code for parameters. So, here in action 2 the code for parameters has already been emitted. So, remember call name *ptr* is a global variable, because we have no inherited variables have been inherited attributes available.

So, now, we need to generate a temporary for the result what is the type of that temporary we do not know that. So, we use the function pointer into the symbol table function name pointer into the symbol table. Execute a function get result type which is a compiler function which goes into the symbol table and gets the type of that function and generate a temporary of that particular type. So, let that be result var. So, we generate refparam result var we already know why this is necessary and the machine code at the time of you know earning will actually place the *a* in return *a* in the variable result var. How is this possible? This is possible, because the position of the result in the parameter list is always known to the code generator and the run time system.

So, the code generator knows that this will be the last in the parameter list it knows the number of parameters and it also knows the position of the result. So, it is easy for the machine code generator to generate an instruction to place this return result into the variable result var using indirect addressing. Remember result var has been passed as a refparam now the call itself call name pointer is the pointer to the function the in the symbol table. And how many parameters PARAMLIST dot p no is the number of parameters plus 1 corresponds to the result which is an implicit parameter. So, this is the call for the function. Now, how do we compute the number of parameters that is fairly easy PARAMLIST going to PLIST. You know just transfers the plist dot p n o to PARAMLIST dot p n o we need this kind of a unique production, because we want to make sure that we permit 0 number of parameters as well.


So, PARAMLIST going to epsilon make sure that 0 number of parameters is permitted. So, PARAMLIST dot p n o is 0 and plist generates 1 or more parameters plist going to E or plist going to plist comma E if it is plist going to E, this is a first parameter. So, we set the number to 1 and the first param E result is generated for the rest of it as and when the expressions are parsed we generate plist 1 rather we generate gen param E dot result. And the number of parameters is counted and incremented. So, plist 2 dot p n o plus 1 is assign to plist 1 dot p N o. So, this is how plist accumulates, the number of parameters and passes it on to PARAMLIST and that is used in this production to generate the appropriate call within the number of parameters.

(Refer Slide Time: 26:42)

### SATG for *Function Declaration*

Assumption: No nesting of functions

- *FUNC\_DECL* → *FUNC\_HEAD* { *VAR\_DECL BODY* }  
 { backpatch(BODY.next, nextquad);  
 gen('func end'); }
- *FUNC\_HEAD* → *RESULT id ( DECL\_PLIST )*  
 { search\_func(id.name, found, namptr);  
 active\_func\_ptr := namptr;  
 gen('func begin active\_func\_ptr'); }

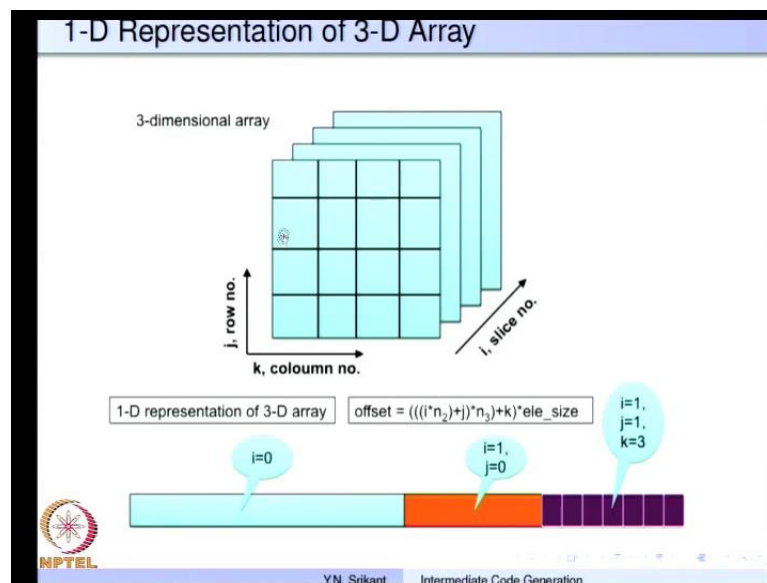


Y.N. Srikant    Intermediate Code Generation

So, what about declaration fairly easy function declaration is function head followed by var declaration body. So, at this point we generate gen then func end. So, because we have parsed this entire right hand side and what about body dot next. So, as I already showed you body dot next corresponds to all the jumps out of the loops etcetera within this body. And the last one last part of the body and that is patched to the function end.

So, that there are no more hanging goto's function head goes to a result i d followed by declaration plist. So, now, we search for i d dot name find it in the symbol table and its pointer is namptr and active func p t r would be. So, this is the function name. So, that would be a namptr we are now looking at the body of the function. So, the pointer is active func p t r now generate func begin active func p t r. So, the code generation for function declarations is quite straight forward.

(Refer Slide Time: 28:03)



Let us now understand how to generate code for various types of assignment statements. So, before that we must understand the representation for multidimensional arrays because we will use multidimensional arrays in assignments and other expressions. So, if there is a 3 dimension array we can look at it in 3 dimensions as slices. So, these are the various slices and each slice consist of squares right. So, each square is an element. So, this is the 2 d array and this is the 3 third dimension of the array. So, this is the i part; this is the j part and this is the k part. So, when we represent a 3 dimension array in memory;

obviously, memories are single dimensional. So, we must map the 3 d array to a 1 d representation how is that done for the 3 d array that we have. So, we these are the slices.

So,  $i = 0$   $i = 1$   $i = 2$  etcetera. So, this is  $i = 0$  and this entire thing is  $i = 0$  to 1 similarly we will have  $i = 1$  etcetera within 1 of the slices. So, say this is the second slice. So, we have  $i = 1$  we have  $j = 0$   $j = 1$   $j = 2$   $j = 3$  etcetera. So, this is  $j = 0$  and this is  $j = 1$ . So, let us say there are only 2 possible indices here. So, within this slice and the part corresponding to  $j = 1$ . So, so  $i = 1$  and  $j = 1$ . So, this is the third element 0 1 2 3. So,  $k = 3$ . So, whenever we say a of 1 1 3 we must map it to this element in the entire single dimensional array representation right. So, that offset is computed as  $i * n_2 + j$  is the size of the second dimension number of elements in the second dimension.

So, this is the slice number then  $n_2$  corresponds to this you know 2 d array size each one of the elements of the slice contains as many relevants in the as in this 2 d array. So, that is why  $i$  begins with a 0 here we are considering c like representation c like programming language. So,  $i$  starts with a 0 that is why it is  $i * n_2$ . So, if we have  $i = 0$  then this becomes 0; obviously, with  $i = 1$  we need to skip this  $i = 0$ . So, that is why when  $i = 1$  we need to skip  $n_2$  element. So, we have skipped here and come to this point skip this entire slice and come up to this point for  $i = 1$  then we add  $j$  to it. So, let us say we are looking at this element. So, we have we now want  $i = 1$   $j = 1$   $k = 3$ .


So, we add  $j$  to it so that is plus 1 and then multiply the whole thing by  $N_3$  that is because we need to skip. So, many elements you know within the  $j$ . So,  $j = 0$  here  $j = 1$  here. So, we need to skip this entire  $j = 0$  and come to this part of the slice rather the 2 d array. So, that is done by  $i * n_3$ . So,  $i * n_3$  says you know this is  $j$ . So, this is the row number. So, how many elements are there in each row that is  $n_3$  now plus  $k$  means the number of elements that we have skipped here. So, 0 1 2 and then  $k = 3$  so we come to this point by adding  $k$  all this has to be multiplied by the number of bytes that each element of the array. So, that is this element takes. So, that is why we multiplied by the element size. So that will bring us to the beginning of this element and then we can do what we want with it.

(Refer Slide Time: 32:55)

```
Code Template for Expressions and Assignments

int a[10][20][35], b;
b = exp1;
code for evaluation of exp1 (result in T1)
b = T1
/* Assuming the array access to be, a[i][j][k] */
/* base address = addr(a), offset = (((i*n2)+j)*n3+k)*ele_size */
a[exp2][exp3][exp4] = exp5;

10: code for exp2 (result in T2) || 141: T8 = T7+T6
70: code for exp3 (result in T3) || 142: T9 = T8*intsize
105: T4 = T2*20 || 143: T10 = addr(a)
106: T5 = T4+T3 || 144: code for exp5 (result in T11)
107: code for exp4 (result in T6) || 186: T10[T9] = T11
140: T7 = T5*35
```



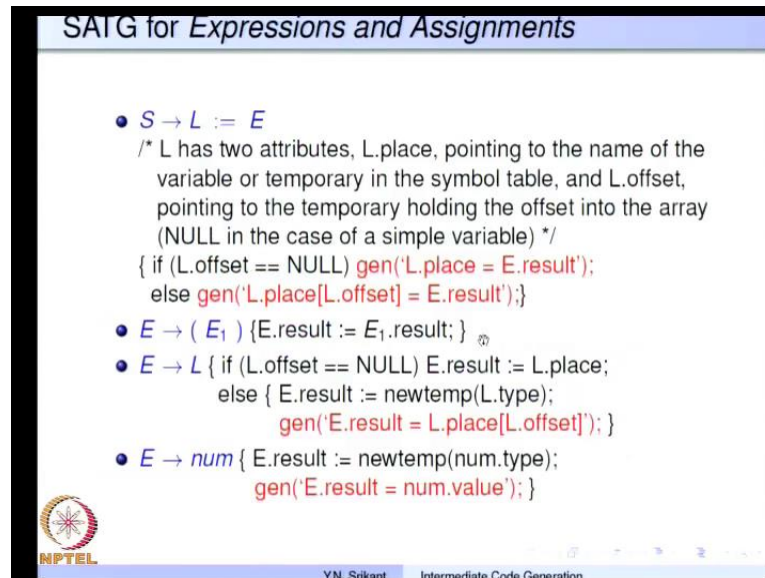
Y.N. Srikant Intermediate Code Generation

So, let us see the code generation template for this consider a declaration a 10 20 35. And then another simple variable b a is an array b is a simple variable for the simple assignment b equal to exp 1 where exp 1 is a arithmetic expression. The template is code for evaluation of exp 1 exp 1 result in T 1 and then the quad b equal to T 1 fairly simple. But suppose the assignment statement is complicated a of exp 2 exp 3 exp 4. So, this is the element of a, to which we want to assign exp5. So, I have a mentioned the offset part again here. So, that we can understand how the code is computed or rather code is generated.

So, exp 2 is evaluated first result in T 2 exp 3 is evaluated next result in T 3 we do T 2 star 20 which is nothing but the n 2 part second dimension 20. Then we add the j that is exp 3 now we generate code for exp 4 then you know this T 5 is multiplied by N 3 that is 35. We add the k part that is T 6 here exp 3 or rather exp 4 now the each element is an in T here. So, element size is in T size. So, T 8 into in T size is T 9 now T 10. We get address a, we generate code for exp 5 in T 11 and now finally, we can say T 10 T 9 is T 11. So, T 10 is the base address of the array T 9 is the offset that we have computed byte level and T 11 is the expression on the right hand side. So, this is the type of code that get's generated for arrays.



(Refer Slide Time: 35:16)



The slide is titled "SATG for Expressions and Assignments". It contains the following content:

- $S \rightarrow L := E$   
/\* L has two attributes, L.place, pointing to the name of the variable or temporary in the symbol table, and L.offset, pointing to the temporary holding the offset into the array (NULL in the case of a simple variable) \*/  
{ if (L.offset == NULL) gen('L.place = E.result');  
 else gen('L.place[L.offset] = E.result'); }
- $E \rightarrow ( E_1 )$  { E.result := E<sub>1</sub>.result; }
- $E \rightarrow L$  { if (L.offset == NULL) E.result := L.place;  
 else { E.result := newtemp(L.type);  
 gen('E.result = L.place[L.offset]'); }
- $E \rightarrow num$  { E.result := newtemp(num.type);  
 gen('E.result = num.value'); }

At the bottom left is the NPTEL logo. At the bottom right, it says "YN. Srikant Intermediate Code Generation".

So, let us see the S attributed grammar for assignments and expressions and understand how the code gets generated during LR parsing this is SATG again. So, we have the assignment statement S going to LR equal to E now L is a left hand side it could be an array element, it could be a simple variable. So, if it is an array element then we need the base address and the offset and if it is a simple variable and we just need the address way of that variable. So, L requires 2 attributes L.place pointing to the name of the variable or the temporary in the symbol table. So, this could be the array base address or array name or the simple variable name and L.offset is the offset in the case of an array. So, if it is a simple variable then it is null.

So, we check whether the variable is a simple or array. So, if L.offset equal to null then it is a simple variable. So, we simply generate L.place equal to E.result. So, this is the assignment which is generated. So, I already showed you that here you know. So, this is simple right this is simple variable. So, L will have L will have only the address of L and nothing else. So, if it is an array variable then we need to generate L.place[L.offset] equal to E.result. So, this is the indexed instruction that needs to be generated. So, this is what we generated here you know this particular instruction then parenthesis expression is as only transfer of the attribute and E going to L. So, this E is on the right side. So, remember that this can also be either a simple variable or an array expression. Again we need to distinguish between these two.

So, if l dot offset is null then E dot result is l dot place. So, simple straight forward if it is not null then in the right hand side is an array expression. So, the point is a single quadruple cannot have array expression on the left hand side and array expression on the right hand side high level language statements can. So, I can write a of b equal to c of d in the high level language statement, but in the quadruple level there can be an array access or indexed instruction only on the left side or on the right side 1 at a time. So, we know that it is an array and we know that this is the right hand side. So, we generate a temporary E dot result equal to new temp l dot type.

So that is the type of l and generate E dot result equal to l dot place bracket l dot offset. So, an indexed instruction is generated and it is a, you know and that will be in E dot result. So, the value will be in E dot result. So, this is how we take care of assignments from an array and here we take care assignments into an array what remains is computation of the mapping from 3 d array to or 4 d array to a single dimensional array. We will see that very soon if E is a number we just generate E dot result equal to num dot value. So, E dot result is a temporary of the type num dot type addition.

(Refer Slide Time: 39:13)

SAIG for *Expressions and Assignments* (contd.)

- $E \rightarrow E_1 + E_2$ 

```

{ result_type := compatible_type(E1.type, E2.type);
  E.result := newtemp(result_type);
  if (E1.type == result_type) operand_1 := E1.result;
  else if (E1.type == integer && result_type == real)
    { operand_1 := newtemp(real);
      gen('operand_1 = cvrt_float(E1.result); };
  if (E2.type == result_type) operand_2 := E2.result;
  else if (E2.type == integer && result_type == real)
    { operand_2 := newtemp(real);
      gen('operand_2 = cvrt_float(E2.result); };
  gen('E.result = operand_1 + operand_2');
}

```

NPTEL  
Y.N. Srikant Intermediate Code Generation

So, E going to E 1 plus E 2 not. So, trivial because here the types of E 1 and E 2 will drive the code generation process. So, if E 1 and E 2 are of the same type and then there is no problem you know we do not need any type conversion if E 1 and E 2 are of different type then we actually have to do a type conversion. So, for example, result type

is compatible type E 1 dot type comma E 2 dot type. So, if E 1 dot type and E 2 dot type is also real absolutely no problem then result type will be real. Similarly if both are integer this will be integer if 1 is integer other one is real then result type will always be real. So, this is how compatible type checks all this we have seen this during semantic analysis. So, we require this to generate the temporary for E dot result new temp result type now generating code for type conversion is done here.

So, if E 1 dot type is result type then there is no type conversion necessary operand 1 equal to E dot result. So, we just transfer the address and if E 1 dot type is integer and result dot type is real so; that means, we must generate code for type conversion. So, we generate a temporary operand 1 equal to new temp real and generate instruction operand 1 equal to convert float E 1 dot result. So, remember E 1 dot result is integer type. So, we are converting it into real type and assigning it into the temporary operand 1. So, this is the instruction can generated for the type conversion exactly the same thing happens for E 2 as well. So, if there is no type conversion operand 2 retains the address of E 2 dot result if there is a need for type conversion.


Then we generate a temporary and you know generate the instructions operand 2 equal to convert float E 2 dot result. So, now, at the end of this exercise operand 1 holds the address of the first operand, operand 2 holds the address of the second operand with or without type conversion. And now we are ready to generate the plus operation E dot result equal to operand 1 plus operand 2. So, this is the instruction for E 1 plus E 2 if there was no problem and all type were just integer. Then we the all these checking was not necessary type conversion was not necessary. We would have generated a simple instruction E dot result equal to E 1 dot result plus E 2 dot result. So, that is not possible anymore because of many types which are permitted in expressions.

(Refer Slide Time: 42:43)

SAIG for *Expressions and Assignments* (contd.)

- $E \rightarrow E_1 || E_2$   
{ E.result := newtemp(integer);  
  gen('E.result = E<sub>1</sub>.result || E<sub>2</sub>.result');
- $E \rightarrow E_1 < E_2$   
{ E.result := newtemp(integer);  
  gen('E.result = 1');  
  gen('if E<sub>1</sub>.result < E<sub>2</sub>.result goto nextquad+2');  
  gen('E.result = 0');  
}
- $L \rightarrow id$  { search\_var\_param(id.name, active\_func\_ptr,  
  level, found, vn); L.place := vn; L.offset := NULL; }

Note: search\_var\_param() searches for id.name in the variable list first, and if not found, in the parameter list next.



YN. Srikant Intermediate Code Generation

Exactly the same thing happens for the logical operation or E dot result we generate a new temporary. And we generate E dot result equal to E 1 dot result or E 2 dot result I am skipping some of the details like type conversion here, because even here character or integer both are permitted in c. So, some simple type conversion may be necessary even for this and it is on the same lines as the E 1 plus E 2. So, I will skip that and leave it for exercises E going to E 1 less than E 2. So, there is a relational operation here. So, E 1 less than E 2 will actually generate either a 1 or a 0 depending on the conclusion. So, if E 1 is definitely less than E 2 then the result is TRUE. So, we generate a 1 if E 1 is not less than E 2 then we generate a 0 and that value is put in E dot result.

So, E dot result get's an integer temporary the address is stored in E dot result first we initialize it to 1. So, generate instruction E dot result equal to 1 then we generate the comparison instruction if E 1 dot result less than E 2 dot result go to next quad plus 2. So, that is the result is TRUE. So, this is next quad the, this if E dot result less than E E 2 dot result is actually next quad. So, this is the number next quad in the quadruple array this is next quad plus 1 and after this that is after the expression E is next quad plus 2. So, that is where we need to jump and continue from that point onwards.

So, if this is TRUE than we do not reset the result to 0, we keep it as 1 and then go further if the result is FALSE then we make the result 0 and then continue with the operation. So, that is how it would be. So, if the left hand side l is just a name then search

for the name. So, the name could be either a variable or a parameter. So, we search the symbol table and get its pointer. So, l dot place becomes V N and l dot offset becomes null, because this is a simple variable.

(Refer Slide Time: 45:23)

**SAIG for Expressions and Assignments (contd.)**

- $ELIST \rightarrow id [ E$ 

```
{ search_var_param(id.name, active_func_ptr,
  level, found, vn); ELIST.dim := 1;
  ELIST.arrayptr := vn; ELIST.result := E.result; }
```
- $L \rightarrow ELIST ]$ 

```
{ L.place := ELIST.arrayptr;
  temp := newtemp(int); L.offset := temp;
  ele_size := ELIST.arrayptr -> ele_size;
  gen('temp = ELIST.result * ele_size'); }
```
- $ELIST \rightarrow ELIST_1 . E$ 

```
{ ELIST.dim := ELIST_1.dim + 1;
  ELIST.arrayptr := ELIST_1.arrayptr
  num_elem := get_dim(ELIST_1.arrayptr, ELIST_1.dim + 1);
  temp1 := newtemp(int); temp2 := newtemp(int);
  gen('temp1 = ELIST_1.result * num_elem');
  ELIST.result := temp2; gen('temp2 = temp1 + E.result'); }
```

NPTEL  
YN. Subant Intermediate Code Generation

Now, we come to the important part of expressions array expressions. So, again we have broken the productions into parts as in the case of semantic analysis. So, ELIST going to i d bracket E. So, this is the first expression search for the expression find it is pointer now the dimension number is 1, because this is the first expression the ELIST dot array pointer is V n. So, that is the pointer to the name of the array in this symbol table. So, that is the entry and ELIST dot result will be E dot result. So, that is the first subscript of the array expression the rest of the subscripts are generated by this production.

And finally, the expression is closed using this production. So, let us look at this first ELIST has generated all the expressions and. So, i d bracket E is also a ELIST. So, first expression is generated then the second third etcetera are all are generated here. Finally, the bracket closes the subscript expression list. So, once all the subscripts are generated l dot place is still ELIST dot array pointer. So, that is the base address part of l then temp equal to new temp int. So, we generate a new temporary and offset will be actually nothing but that new temporary. We are going to generate instructions to collect the offset into the temporary size of the element is obtain through the symbol table pointer.

So, ELIST dot array pointer point to element size. So, this will give us the element size from the symbol table now we generate the instruction temp equal to ELIST dot result. So, that is the accumulation of all the you know subscripts here star E l E size. So, we have done the nested evaluation finally, what remains is to multiply by the size of the element like int or float. So, that is done here after ELIST dot result accumulates the entire expression value so. So, now, at the end of this we l dot place and l dot offset will contain the appropriate pointers into the symbol table and the code for evaluation of the entire subscript list has been generated. So, here is the subscript list generated.

So, this is the first part of the subscript list and this is the next last one. So, if there are 3 subscripts; first one is generated here then you know ELIST 1 comma E. So, this is the second 1 and then the third 1 and so on and so forth using this we can generate as many as we really want. So, if we if we do this 3 2 times we generate ELIST 1 comma E and ELIST 1 comma E again. So, and finally, ELIST going to i d E expression generates the first one. So, this is the first one and then the second one and third one can be generate using these productions again and again number of dimensions ELIST dot dim is this plus 1. So, this is an x dimension. So, that we did array pointer is just transferred you know from here.


So, we found it we got that here right. So, we are going to transfer it to ELIST dot array pointer number of elements we look at the symbol table using the array pointer. And then the number of dimensions or number of elements in this dimension for this subscript will be ELIST 1 dot dim plus 1 right. So, this is the dimension number of this. So, we search the symbol table and get the number of elements corresponding to that particular dimension that is this E. Now, generate temporaries do partial evaluation temp 1 and temp 2 both are int type, because subscripts are always integer expressions temp 1 multiplies. So, ELIST 1 dot result star num elem.

So, the number of elements in this dimension is multiplied with ELIST 1 dot result. So, that is I star into plus j whole thing star N 3. So, that is the, you know a nested evaluation that we are doing here then we add temp 2 generate temp equal to temp 1 plus E dot result. So, this is this does I star into then plus j that is what we are using. So, if we repeat this using the same production again and again you can generate the instructions for the entire subscript list. So, now, we have completed code generation for expressions assuming that there is nothing called short circuit evaluation.

(Refer Slide Time: 50:59)

**Short Circuit Evaluation for Boolean Expressions**

- $(exp1 \ \&\& \ exp2)$ : value = if  $(\sim exp1)$  then FALSE else  $exp2$ 
  - This implies that  $exp2$  need not be evaluated if  $exp1$  is FALSE
- $(exp1 \ || \ exp2)$ : value = if  $(exp1)$  then TRUE else  $exp2$ 
  - This implies that  $exp2$  need not be evaluated if  $exp1$  is TRUE
- Since boolean expressions are used mostly in conditional and loop statements, it is possible to realize perform short circuit evaluation of expressions using control flow constructs
- In such a case, there are no explicit '||' and '&&' operators in the intermediate code (as earlier), but only jumps
- Much faster, since complete expression is not evaluated
- If unevaluated expressions have side effects, then program may have non-deterministic behaviour

 Y.N. Srikant Intermediate Code Generation

So, let us understand what exactly is short circuit evaluation for Boolean expressions. So, expression 1 and expression 2 suppose we want evaluate this, the straight forward message would be evaluate expression 1 completely. Then evaluate expression 2 completely and generate an instruction for the two. So, that is what we did here right. So, instead of or we can have and here. So, you would generate E 1 dot result and E 2 dot result, but there is another way of generating the result. So, its value is if expression 1 is FALSE then the entire expression is FALSE that is TRUE, because this is and operation. So, if this is FALSE then the entire expression is FALSE.

So, if the expression 1 is TRUE then you know we must evaluate expression 2 otherwise we would not know how exactly what exactly the value would be. So, the value would be  $exp2$  itself whether it is TRUE or FALSE whatever that is. So, this implies that  $exp2$  need not be evaluated if expression 1 is FALSE. So, in other words we are short circuiting expression 2 that is why it is called short circuit evaluation the same thing holds for or as well in. But it is exactly the converse in the case of and if this was FALSE the expression would have been FALSE in the case of or if this is TRUE  $exp1$  is TRUE the entire expression is TRUE.

So, if  $exp1$  is TRUE we can short circuit  $exp2$ , because it is not necessary to evaluate it anymore, but if expression 1 is FALSE then we must evaluate expression 2 as well. So, the value is if  $exp1$  then TRUE else  $exp2$ . So, that is the way short circuit evaluation is

done expression 2 is not evaluated if expression 1 is TRUE. So, Boolean expressions are commonly used in conditional loop statements. And if we use short circuit evaluation it is possible to realize this short circuit evaluation of expressions using control flow constructs I will show you how that can be done. So, in other words in such cases there are no explicit or and operators in the intermediate code as we saw earlier. But there are going to be only extra jump statements this is much faster, because complete expression is not evaluated at all.


And if unevaluated expressions have side effects for example, if expression 2 has a function which alters a global variable or prints out something into a file etcetera. Then it will never be done if expression 1 is TRUE, because we are going to short circuit expression 2. So, in such cases the program may have non deterministic behaviour, because the one once first time possibly expression is TRUE and this entire thing is skipped, but next time possibly expression 1 is FALSE. So, expression 2 will also be evaluated and at that time the printing will happen. So, the person who is running the program will see that printing happen sometime and printing does not happen some other time. So, this is nondeterministic behaviour.

(Refer Slide Time: 54:41)

**Control-Flow Realization of Boolean Expressions**

```
if ((a+b < c+d) || ((e==f) && (g > h-k))) A1; else A2; A3;
```

100: T1 = a+b  
101: T2 = c+d  
103: if T1 < T2 goto L1  
104: goto L2  
105:L2: if e==f goto L3  
106: goto L4  
107:L3: T3 = h-k  
108: if g > T3 goto L5  
109: goto L6  
110:L1:L5: code for A1  
111: goto L7  
112:L4:L6: code for A2  
113:L7: code for A3

 NPTEL  
Y.N. Srikant Intermediate Code Generation

Let me show you a a template or rather an example of how it can be done and then we will see how to generate code. So, the expression is if a plus b less than c plus d or this entire thing then a 1 else a 2 followed by the statement a 3 irrespective of what happens



here. So, we evaluate  $a + b$  and  $c + d$  if  $T_1$  is less than  $T_2$ . So, we are evaluating this now go to 1.1. So, if this is TRUE then the entire thing can be skipped. So, and where is 1.1.1.1 is code for a 1. So, this is short circuit we have short circuited these entire expression. So, that is taken care of by go to 1.1 otherwise we go to 1.2. 1.2 now evaluates the second expression second expression has and here.

So, if  $E = f$  go to 1.3. So, that then we must evaluate this expression if  $E = f$  is FALSE then you know we do not have to evaluate this entire expression at all. So, that means, we go to 1.4 which is code for a 2. So, that is the else part of it. So, inside this we again check  $g > h - k$  go to 1.5 or go to 1.6. So, that is the code for a 2 finally, code for a 3 will be executed in all the cases. So, this is how short circuit evaluation really happens in the case of if then else we will stop here. And we will continue with code generation scheme for such expressions in the next part of the lecture.

Thank you.