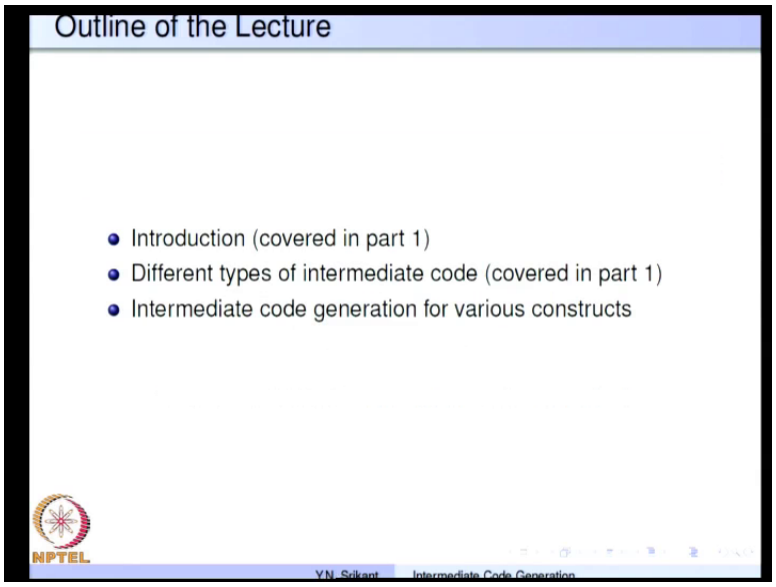
Principle of Complier Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture - 19
Intermediate code generation Part - 3

(Refer Slide Time: 00:21)



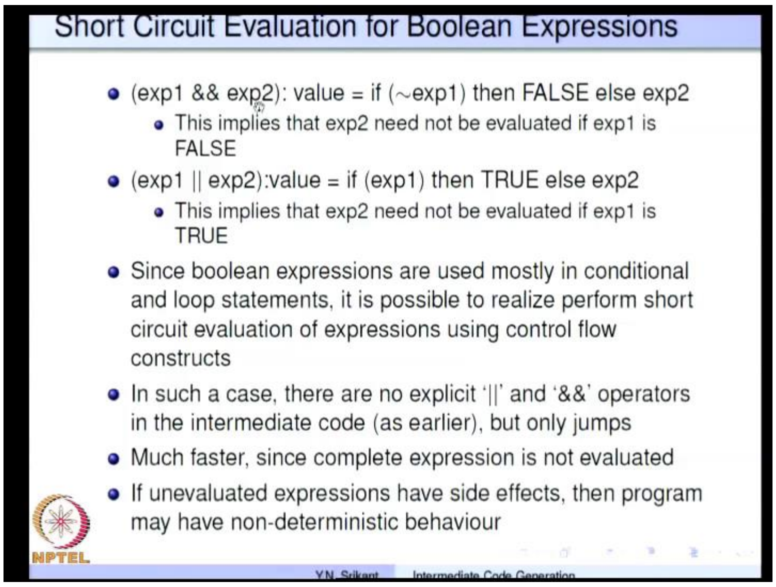
Outline of the Lecture

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

Y.N. Srikant Intermediate Code Generation

Welcome to the part three of the lectures on intermediate code generation. So, today we will continue to discuss the code generation strategies for various constructs.

(Refer Slide Time: 00:30)



Short Circuit Evaluation for Boolean Expressions

- (exp1 && exp2): value = if (~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

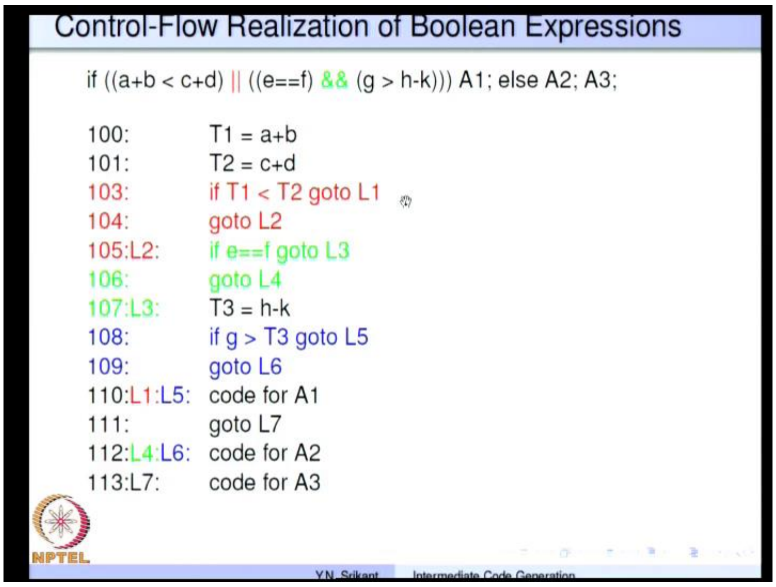
Y.N. Srikant Intermediate Code Generation

So, we briefly discussed short circuit evaluation of Boolean expressions in the last lecture. So, today, let us consider in detail and see what difficulties it can actually give raise to. So, if there is an expression Boolean expression, expression 1 and an expression 2 or expression 1 or expression 2. It is not necessary to evaluate the complete expressions in all cases to know the value of the expression. For example, this is an and operator, if expression 1 is true then we must actually evaluate expression 2 in order to know the complete value of the expression. However, if expression 1 is falls it is known that the entire value will be falls, it is not necessary to evaluate expression 2.

So, similarly in this case if expression 1 is true, it is not necessary to evaluate this complete expression at all it is necessary to evaluate expression 2 only if expression 1 is falls. So, therefore, in the case of short circuit evaluation, we try to evaluate only parts of the expression as necessary in for the complete value. You know the value of the complete expression; we do not really evaluate the entire expressions in all cases. And it is also possible to generator a control flow code for such off circuit evaluation as we will see very soon. And since such Boolean expressions are mostly used in conditional and loop statements, we can always take advantage of the control flow representation of Boolean expressions.

The advantage of this short circuit and control flow, you know expression representation is that it is much faster than the normal evaluation. But the disadvantage is that if by chance the expression has some side effects. For example, expression 2 has a function call or a print statement and this function call has alters some global variable in such a case if expression 2 is never evaluated. Because the value of the entire expression was inferred using expression 1 then non deterministic behavior can arise. So, in some cases exp 2 is not evaluated. So, the printing and other side effects should not take place in some cases expression 2 will be evaluated and in these cases the printing will happen. So, such non deterministic behavior if it is not desirable then short circuit evaluation should not be performed.

(Refer Slide Time: 03:45)



Control-Flow Realization of Boolean Expressions

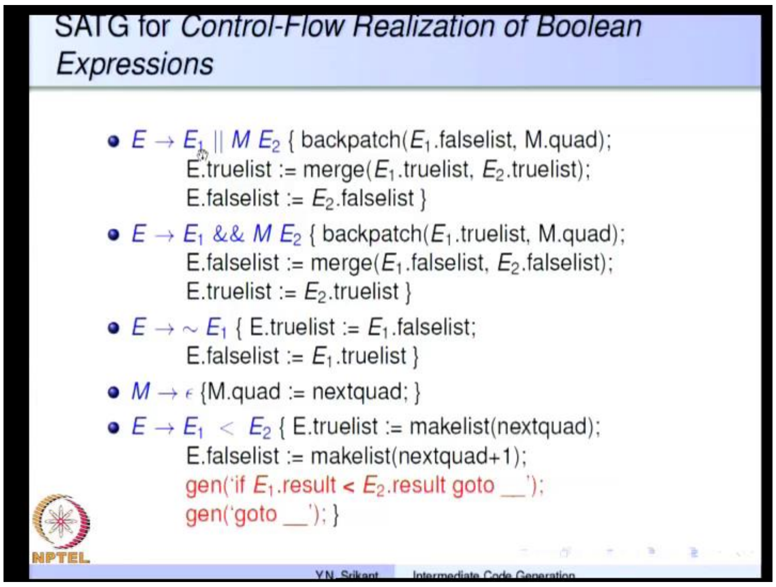
if ((a+b < c+d) || ((e==f) && (g > h-k))) A1; else A2; A3;

```
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
```

Y.N. Srikant
Intermediate Code Generation

Here is a detailed example. So, we an expression here and it is add with another expression. So, in this case if the first expression is true we do not have to evaluate the second expression at all. So, the code here shows that. So, we do T 1 equal to a plus b T 2 equal to c plus d and if T 1 is less than T 2 we go directly to L 1 which is the code for a 1 that is the then part of the expression. So, we skip the evaluation of this entire expression, but T 1 that is then T 2 is falls then we have to go and evaluate the second expression. So, that is L 2 part again here we have and expression. So, if this is falls. So, we do not have to evaluate this part and if this is true, we need to continue evaluation for this part as well. So, if E equal to f go to l 3 so; that means, we must evaluate h minus k otherwise we go to L4 which is the really the code for else part that is S 2 right here. So, this is how the short circuit evaluation and control for representation of Boolean expressions takes place.

(Refer Slide Time: 05:06)



SATG for *Control-Flow Realization of Boolean Expressions*

- $E \rightarrow E_1 \mid\mid M\ E_2$ { backpatch(E_1.falselist, M.quad);
  E.truelist := merge(E_1.truelist, E_2.truelist);
  E.falselist := E_2.falselist }
- $E \rightarrow E_1\ \&\&\ M\ E_2$ { backpatch(E_1.truelist, M.quad);
  E.falselist := merge(E_1.falselist, E_2.falselist);
  E.truelist := E_2.truelist }
- $E \rightarrow \sim E_1$ { E.truelist := E_1.falselist;
  E.falselist := E_1.truelist }
- $M \rightarrow \epsilon$ {M.quad := nextquad; }
- $E \rightarrow E_1 < E_2$ { E.truelist := makelist(nextquad);
  E.falselist := makelist(nextquad+1);
  gen('if E_1.result < E_2.result goto __');
  gen('goto __'); }

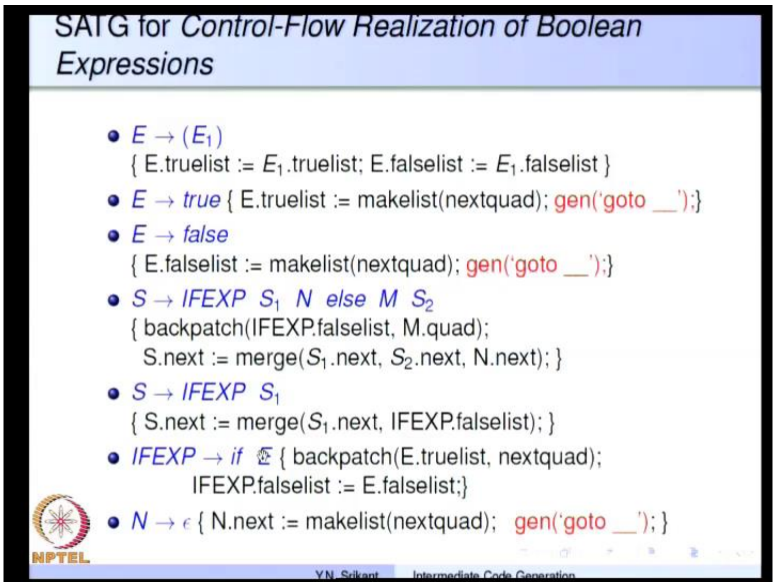
Y.N. Srikant Intermediate Code Generation

So, now let us understand how to generate such code so, E going to E 1 or E 2. So, as usual we place a marker M here m really goes to epsilon and only remembers the quadruple number of the next quadruple to be generated. So, in this case it remembers the beginning of the code for E 2. So, if E 1 is true then you know we do not have to evaluate E 2 at all. So, therefore, E dot truelist, which is very similar to the statement dot next in the case of statements. So, E dot truelist becomes a merger of E 1 dot truelist and E 2 dot truelist. So, there could be jumps out of this E 2 as well that is why we need to have truelist and merge them. This is the exit part for the expression, in case E 1 is true then if E 1 is false we must compulsorily evaluate E 2.

Therefore, it is logical to patch E 1 dot falselist to the beginning of M 2 which is m dot quad and E dot falselist in this case will become E 2 dot falselist. Because we come to E 2 only if E 1 is really false what about the, and operator very similar it is you know in some sense complimentary to the, or operator. So, if E 1 is false the entire expression would be false. So, we have to take an exit. So, E dot falselist is a merger of E 1 dot falselist and E 2 falselist and if E 1 is true we still continue and evaluate E 2. So, we backpatch E 1 dot truelist to the beginning of E 2 which is nothing but m dot quad and E dot 2 truelist would be; obviously, E 2 dot truelist.

Because we come to E 2 only if E 1 is true otherwise we do not even come to E 2 E going not E 1 we simple swap the list truelist becomes E 1 dot falselist and falselist becomes E 1 dot truelist. And we saw a m types are an already E going to E 1 less than E 2. So, we must; obviously, generate a test here. So, generate E 1 if E 1 dot results less than E 2 dot result goto and then followed by goto this is the false part and this is the true jump. So, that is why E 2 truelist takes on it just quadruple and E dot falselist takes the next quadruple. So, these 2 list contains these quadruples.

(Refer Slide Time: 08:12)



SATG for *Control-Flow Realization of Boolean Expressions*

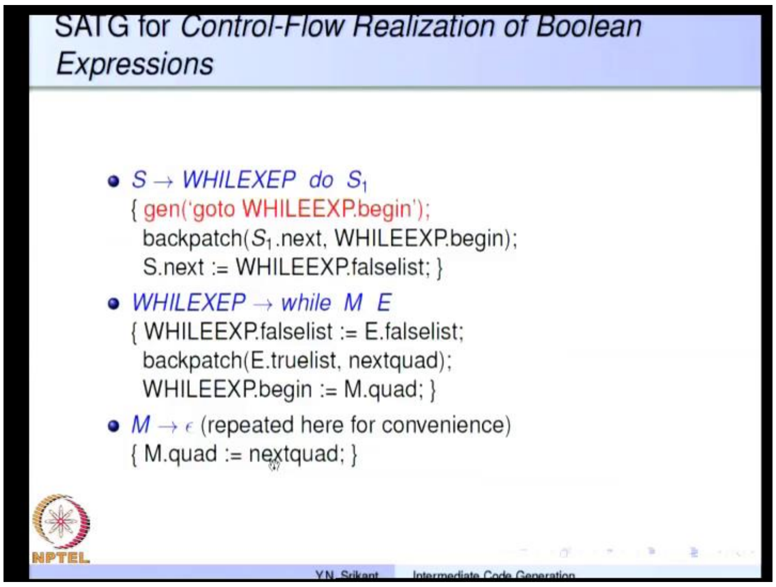
- $E \rightarrow (E_1)$
  { E.truelist := E_1.truelist; E.falselist := E_1.falselist }
- $E \rightarrow$ *true* { E.truelist := makelist(nextquad); gen('goto __');}
- $E \rightarrow$ *false*
  { E.falselist := makelist(nextquad); gen('goto __');}
- $S \rightarrow$ *IFEXP* S_1 *N else M* S_2
  { backpatch(IFEXP.falselist, M.quad);
  S.next := merge(S_1.next, S_2.next, N.next); }
- $S \rightarrow$ *IFEXP* S_1
  { S.next := merge(S_1.next, IFEXP.falselist); }
- *IFEXP* $\rightarrow$ *if E* { backpatch(E.truelist, nextquad);
  IFEXP.falselist := E.falselist;}
- $N \rightarrow \epsilon$ { N.next := makelist(nextquad); gen('goto __'); }

Y.N. Srikant Intermediate Code Generation

Here E going to parenthesize even parenthesis parenthesize expression. So, we just copy the list and nothing much happens and if E is the constant true just like numeric's. This is the Boolean constant true and this is a Boolean constant false then truelist will takes in itself this goto. And in the case of falselist E dot falselist will take this particular goto statement. So, this is easy to see you know use of this is very easy to see for example, you know if we have this if expression S 1 so, if E this E is let us assume the constant true right. So, in that case E dot truelist will be patched to nextquad which is nothing but the beginning of then statement and in this case E false E dot falselist will be nil.

So, nothing is patched here and the similar for E equal to false if this is false then you know we have to jump the else part and that automatically is done later. So, let us go through this S going to ifexp S 1 n else m S 2. So, ifexp is false ifexp dot false list is you know if expression is false then it is patched to we need tojump to m dot quad which is the beginning of the else part. So, false list is patched to this and the exits from S would be the exits from S 1 and the exits from S 2. And also the jump here n the jump from N to epsilon generated during the reduction of into epsilon this we already saw if S going to ifexp S 1. So, the jumps out of S would be merger of S 1 dot next and ifexp dot false list these are the only ways to jump out of if then statement.

(Refer Slide Time: 10:38)



SATG for *Control-Flow Realization of Boolean Expressions*

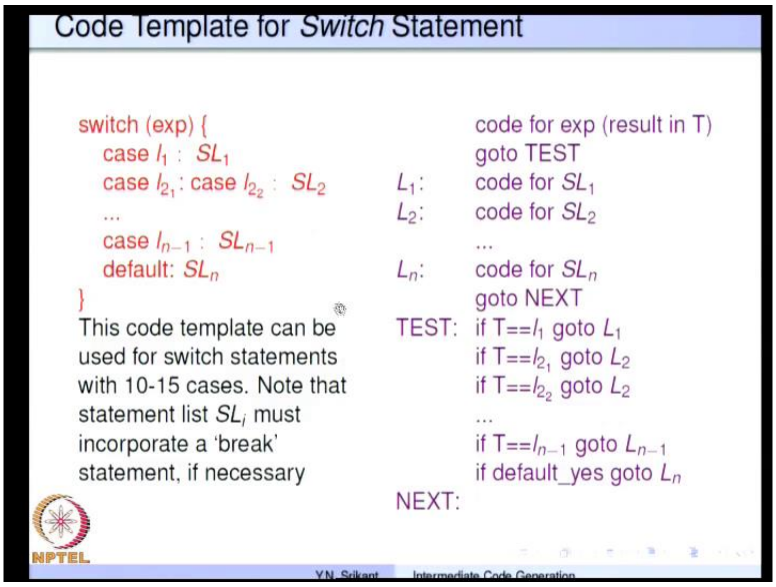
- $S \rightarrow WHILEXEP\ do\ S_1$
  { gen('goto WHILEEXP.begin');
  backpatch(S_1.next, WHILEEXP.begin);
  S.next := WHILEEXP.falselist; }
- $WHILEXEP \rightarrow while\ M\ E$
  { WHILEEXP.falselist := E.falselist;
  backpatch(E.truelist, nextquad);
  WHILEEXP.begin := M.quad; }
- $M \rightarrow \epsilon$ (repeated here for convenience)
  { M.quad := nextquad; }

Y.N. Srikant Intermediate Code Generation

So, the now while statement. So, here we must be you know we must observe in in this case we must observe that for the, or and operators we really have not generated any code. So, it is just that we direct the truelist and false list appropriately. So, the actual code gets generated in a comparison all right and these quadruples will be patched appropriately in these productions. So, the really speaking this is the comparison and goto are the only statement generated and that is why it is called a control flow representation or implantation of Boolean expressions. So, going to the while loop this is very similar to what we have seen already. So, whileexp do S 1 so, we need to jump we are here. So, we jump to the beginning of the while loop.

So, we generate a goto whileexp dot begin and all the jumps out of S 1 must go to whileexp dot begin. So, that is backpatch here the only exit out of S would be when this expression is false that is whileexp dot false list this is very simple. So, whileexp going to while M E. So, false list will become E dot false list that is that is the exit and truelist will be patch to nextquad which is nothing but the beginning of S 1. Because when the while when the expression is true we just fall through and execute the statement S 1 and whileexp dot begin would be the beginning of the expression M dot quad. So, M to epsilon of course, is the same as before.

(Refer Slide Time: 12:44)



Code Template for *Switch* Statement

```
switch (exp) {
   case $l_1$ : $SL_1$
   case $l_{2_1}$ : case $l_{2_2}$ : $SL_2$
   ...
   case $l_{n-1}$ : $SL_{n-1}$
   default: $SL_n$
}
```

This code template can be used for switch statements with 10-15 cases. Note that statement list SL_i must incorporate a 'break' statement, if necessary

```
          code for exp (result in T)
          goto TEST
$L_1$:    code for $SL_1$
$L_2$:    code for $SL_2$
          ...
$L_n$:    code for $SL_n$
          goto NEXT
TEST:     if T==$l_1$ goto $L_1$
          if T==$l_{2_1}$ goto $L_2$
          if T==$l_{2_2}$ goto $L_2$
          ...
          if T==$l_{n-1}$ goto $L_{n-1}$
          if default_yes goto $L_n$
NEXT:
```

Y.N. Srikant   Intermediate Code Generation

So, now let us proceed and look at the code generation mechanism for one of the most complicated programming language constructs the switch statement in c or a c like language. The syntax of the statement is switch and there is an expression. So, this expression must necessarily be producing integer types and then we have a list of values here l 1 l 2 1 l 2 etcetera. And we also have a list of statements S l 1 S l 2 S l 3 etcetera. So, the way it goes if this expression is evaluated at run time and l 1 l 2 etcetera are all these l 2 1 etcetera are all constants either integer or character. So, if this expression is equal to this value at run time then the list of statement S l 1 is executed if it is equal to l 2 1 or l 2 or any of these. Then S l 2 is executed etcetera and if none of them will match then the defaults takes over and S l n will be executed.

So, this code can be this switch statement can be complied in many ways. One simple way would be an, if then else like this you know an explicit if tests and soon and so forth. So, let us go through this and this is good for 10 to 15 cases so, code for expression. So, the result is in temporary t now we must why is this switch statement difficult for code generation the problem is the expression is evaluated. So, code for generation code generation for expression is very simple straight forward we really cannot generate a test for l 1. And then say let us go to S l 1 etcetera will have to generate the code for S l 1 S l

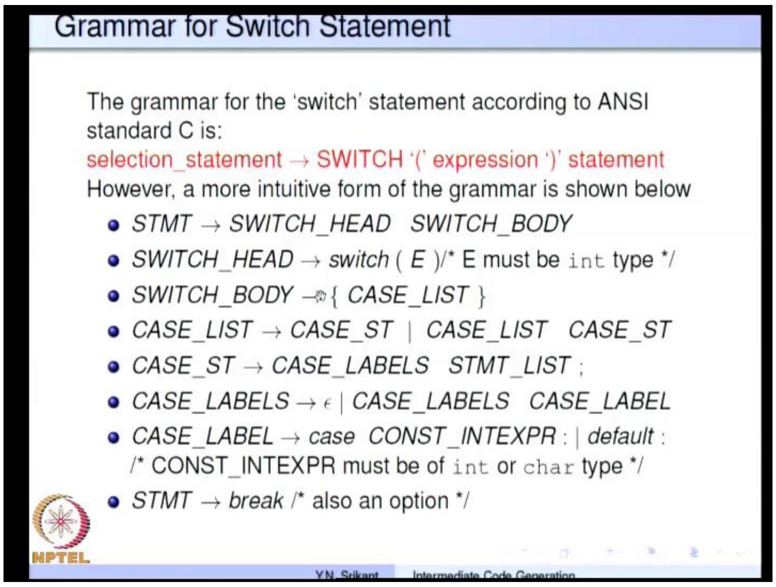
2 etcetera S l n minus 1 S l and then you know make a test you know. So, the reason for this is if all the test are the end after the statements are all generated code for statement is generated. The complier can possibly replace this entire set of if then statements by a more efficient implementation such as a table search. Or sometimes you even know may be a hash table and things of that kind.

So, after the code for expressions so, we have a goto so, we directly go here and following that it is the code for S l 1 S l 2 etcetera. But they control flow will bring us here if execute the test if t equal to l 1 then goto l 1. So, we execute the code for S l 1 if we wanted to execute only a S l 1. And then jump to exit we should have had a break statement within S l 1 as a part of it. So, if there is no break statement in S l 1 automatically after S l 1 it will goto S l 2 and execute this, this is known as false true execution and the c language does not really prohibit this. So, a break statement must be compulsorily inserted here if we really want to get out of the switch statement. After executing S l 1 the same is true for S l 2 etcetera up to S l n as well and after this we have the code for next h w jumps to the exit.

So, in the worst case, if we start executing the code for S l 1, because t equal to l 1 is true. And we have no break anywhere all these would be executed and then we goto the end of the switch. So, this how the code generation must happen for the switch statement. So, let us see how it can be done. So, remember this order first the code for expression and then a goto and then we have the code for S l 1 S l 2 etcetera. But in the mean while when we generate the code for S l 1 S l 2 etcetera we must remember the beginning address of these codes l 1 l 2 l n so that we can generate these, if then statements at the end of this sequence of codes.

So, if we do not know l 1 and if we have not remembered this then you know will not be in a position to generate this statement, statement at all out strategy would be generate the code for expression. Then as we as the parsers cans this l 1 l 2 1 l 2 etcetera its creates a list of these labels and along with it also keeps track of the beginning of the corresponding statements. So, for l 1 it would keep the beginning of S l 1 as a pair and in the case of l 2 1 it keep the beginning of S l 2 for l 2 again the beginning of S l 2 etcetera. And for the default it will remember separately the beginning of its block of statements S l n. So, this is our strategy for code generation.

(Refer Slide Time: 18:44)



Grammar for Switch Statement

The grammar for the 'switch' statement according to ANSI standard C is:

selection_statement → SWITCH '(' expression ')' statement

However, a more intuitive form of the grammar is shown below

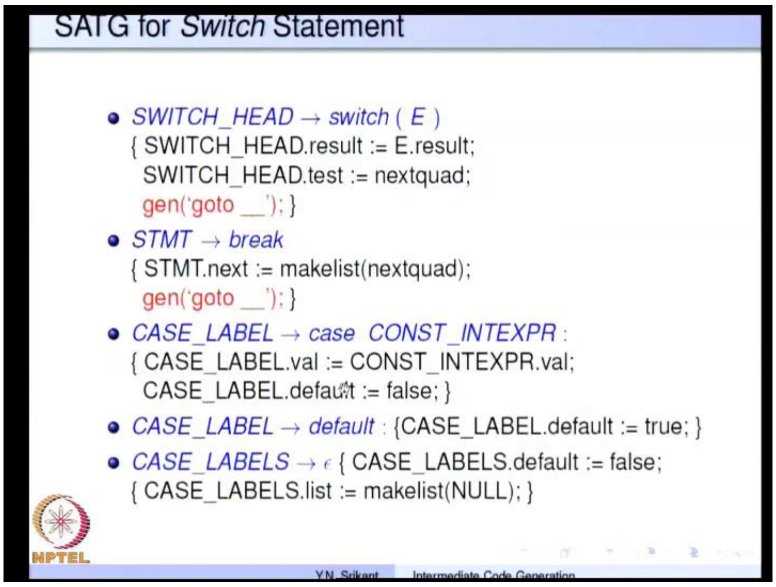
- *STMT* → *SWITCH_HEAD SWITCH_BODY*
- *SWITCH_HEAD* → *switch* (*E*)/* E must be `int` type */
- *SWITCH_BODY* → { *CASE_LIST* }
- *CASE_LIST* → *CASE_ST* | *CASE_LIST CASE_ST*
- *CASE_ST* → *CASE_LABELS STMT_LIST* ;
- *CASE_LABELS* → ϵ | *CASE_LABELS CASE_LABEL*
- *CASE_LABEL* → *case CONST_INTEXPR* : | *default* :
  /* CONST_INTEXPR must be of `int` or `char` type */
- *STMT* → *break* /* also an option */

Y.N. Srikant Intermediate Code Generation

So, let us look at the grammar for the switch statement. So, here according to the c reference manual switch statement is the one of the selection statements. So, selection statement can be switch followed by expression followed by statement. So, if this statement is not one of you know the case statements as we know then there is really no effect of the switch at all. So, there is much you can do unless you have cases and the labels. So, that is why here is a more intuitive form of the switch statement which inherently has to be taken care of any parser anyway.

So, let us look at it statement is the switch statement i mean is switch head followed by switch body. The switch head is the reserved word switch followed by the expression and E must be of the int type. The switch body has a, has a pair of flower brackets followed by that is you know case list. So, the case list has a list of case statements and each case statement has a list of case labels followed by a statement list the case labels can be empty or you know many case labels. So, each case label has the reserved word case followed by a constant integer expression. So, this must be either int or char type or it could be the reserved word default one of the statement can be a break to ensure that the correct flow of statements happens.

(Refer Slide Time: 20:37)



SATG for *Switch* Statement

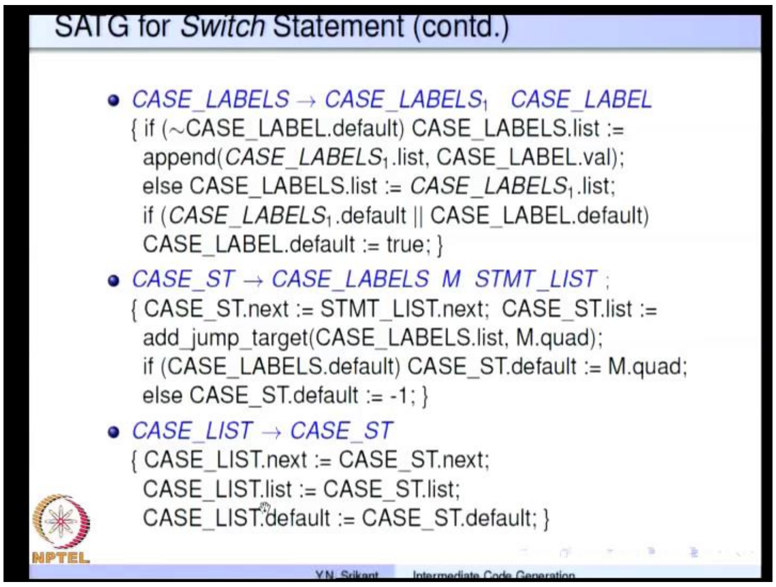
- *SWITCH_HEAD → switch (E)*
  { SWITCH_HEAD.result := E.result;
  SWITCH_HEAD.test := nextquad;
  gen('goto __'); }
- *STMT → break*
  { STMT.next := makelist(nextquad);
  gen('goto __'); }
- *CASE_LABEL → case CONST_INTEXPR :*
  { CASE_LABEL.val := CONST_INTEXPR.val;
  CASE_LABEL.default := false; }
- *CASE_LABEL → default :* {CASE_LABEL.default := true; }
- *CASE_LABELS → ϵ* { CASE_LABELS.default := false;
  { CASE_LABELS.list := makelist(NULL); }

Y.N. Srikant Intermediate Code Generation

So, let us look at the productions and see how to generate code. So, switch head goes to switch followed by E. So, we have an attribute called switch head dot result and another attribute called switch dot test. So, switch head dot results stores the address of the result of E. So, E dot result and switch head dot test stores the address of the next quad which is nothing but a goto statement. So far if you observe our code template we have generated the code for expression and we have generated a goto. But we still do not know the address test here we will know that only at the end of the sequence of statements S you know at the end of the switch statement really speaking. So, that is why we need to retain it on switch head dot test and will patch it at a later point. If the statement is a break statement then it is like the exit of the statement. So, the statement dot next would take this on and a next statement generated would be a goto. So, nothing else is done really.

So, we generate agoto here the reason why it is. So, simple to take care of a break statement within a switch is that we do nothing but getting out of the switch statement. So, it can be put on statement dot next we are going to see later that for loops the handling break is not a. So, trivial its requires much more machinery, what is a case label it has a case followed by an integer constant integer expression. So, we keep the value of constant integer expression which is available in const int expr dot val keep it in case label dot val and; obviously, this is not a default case. So, default is set as false if it is a default then we say case label dot default equal to true. So, now, we go on generating many case labels. So, if this goes to epsilon then it is simply set it at false and put a null list on case labels start list.

(Refer Slide Time: 23:06)



SATG for *Switch* Statement (contd.)

- *CASE_LABELS* → *CASE_LABELS*$_1$ *CASE_LABEL*
  { if (~CASE_LABEL.default) CASE_LABELS.list :=
  append(*CASE_LABELS*$_1$.list, CASE_LABEL.val);
  else CASE_LABELS.list := *CASE_LABELS*$_1$.list;
  if (*CASE_LABELS*$_1$.default || CASE_LABEL.default)
  CASE_LABEL.default := true; }
- *CASE_ST* → *CASE_LABELS M STMT_LIST* ;
  { CASE_ST.next := STMT_LIST.next; CASE_ST.list :=
  add_jump_target(CASE_LABELS.list, M.quad);
  if (CASE_LABELS.default) CASE_ST.default := M.quad;
  else CASE_ST.default := -1; }
- *CASE_LIST* → *CASE_ST*
  { CASE_LIST.next := CASE_ST.next;
  CASE_LIST.list := CASE_ST.list;
  CASE_LIST.default := CASE_ST.default; }

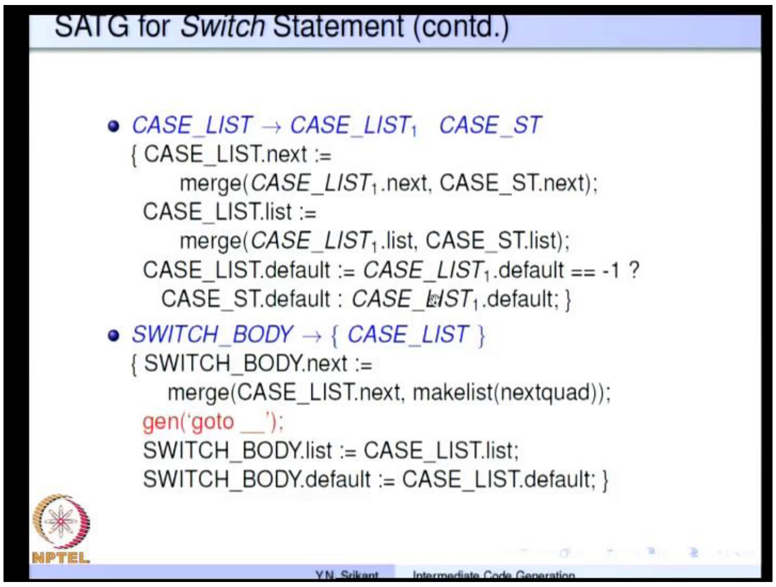
Y.N. Srikant Intermediate Code Generation

If we are generating many case labels well you know if case label dot default this one is not true. In other words this is not a default case then case labels dot list is got by appending case labels dot case labels 1 dot list followed by case label dot val. So, if it is default we have to maintain it separately that is why we are doing all this. So, of it is not a default then we add this case label value to this case labels dot list along with the other one case labels one dot list. So, this is the list of labels that we are maintaining we still have not really place the beginning of these statements you know paired them we have still not paired it with these values we will do that very soon.

So, of case labels dot default case labels 1 dot default or case label dot default is true then there is a default among these. So, case labels dot default is set to true case statement goes to case labels M statement list. So, we have taken care of all the case labels here now we get the statement list and its beginning is remembered in dot quad. So, case statement dot next is statement list dot next. So, that is the exit out of the statement list case statement dot list is you know obtain by executing add jump target to case labels dot list with M dot quad. So, what we are doing here is to take the address of the beginning of the statement list which is available in M dot quad attach it to all the values on this case labels dot list and make that the case S t dot ha you know list.

So, now we have a list of labels and list of associated statement address as well. So, at the end of the switch statement we can generate the, if then tests. So, if case labels this is a default statement then we set it as m dot if m dot quad, because for this again we need to jump to the beginning of statement is otherwise we set it as minus 1. So, case list generates a list of case statements. So, this is nothing but a transfer of all the attributes here there is nothing more it then that. So, this is so, now so far we have generated this expression we have generated this test goto and then we have generated all the code for S l 1 S l 2 S l n. We have we remembered the beginnings of these codes and we have also remembered the values l 1 l 2 1 l 2 2 l n minus 1 paired with it l 1 l 2 l n. So, far this is a work done.

(Refer Slide Time: 26:25)



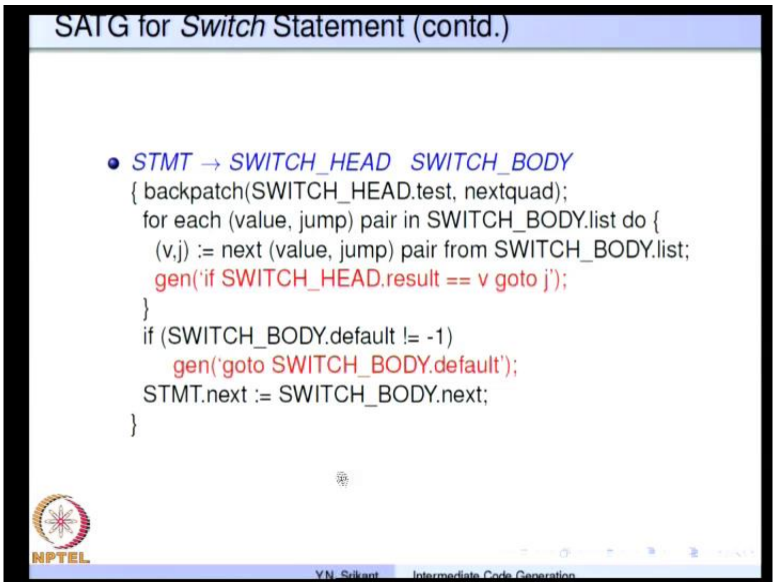
SATG for *Switch* Statement (contd.)

- $CASE\_LIST \rightarrow CASE\_LIST_1 \quad CASE\_ST$
```
{ CASE_LIST.next :=
      merge(CASE_LIST1.next, CASE_ST.next);
  CASE_LIST.list :=
      merge(CASE_LIST1.list, CASE_ST.list);
  CASE_LIST.default := CASE_LIST1.default == -1 ?
    CASE_ST.default : CASE_LIST1.default; }
```
- $SWITCH\_BODY \rightarrow \{ \ CASE\_LIST \ \}$
```
{ SWITCH_BODY.next :=
      merge(CASE_LIST.next, makelist(nextquad));
  gen('goto __');
  SWITCH_BODY.list := CASE_LIST.list;
  SWITCH_BODY.default := CASE_LIST.default; }
```

Y.N. Srikant   Intermediate Code Generation

So, now the again we have a transfer of attributes case list goes to case list 1 case list case statement. So, we merge the 2 next and put it on this merge the 2 list and put it on case list a default is set appropriately depending on whether this is the default or this is the default. So, now, the switch body is nothing but case list. So, we have generated code for all this. So, we must generate code for goto next and then these test this is what remains. So, switch body is completed. So, we generate a goto here switch body dot next would be merger of case list dot next. And then the nextquad which is nothing but the goto here and for the other 2 it is just a transfer of attributes.

(Refer Slide Time: 27:29)



SATG for *Switch* Statement (contd.)

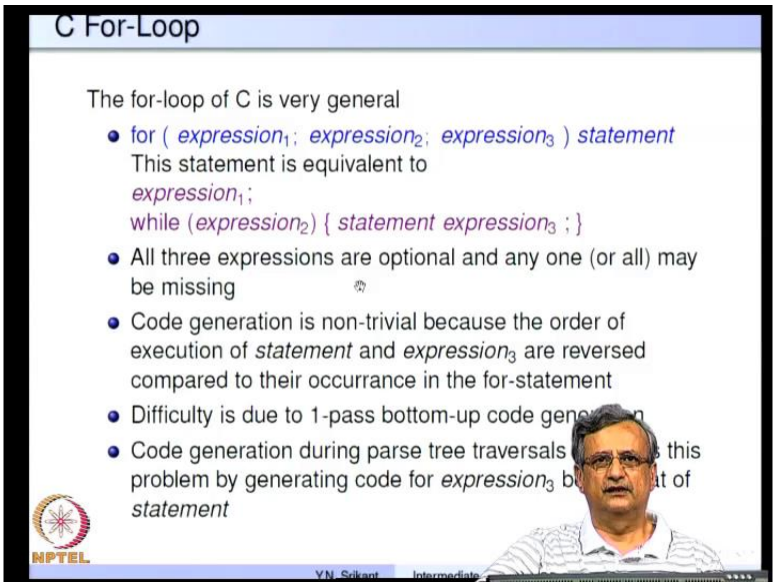
```
• STMT → SWITCH_HEAD  SWITCH_BODY
  { backpatch(SWITCH_HEAD.test, nextquad);
    for each (value, jump) pair in SWITCH_BODY.list do {
      (v,j) := next (value, jump) pair from SWITCH_BODY.list;
      gen('if SWITCH_HEAD.result == v goto j');
    }
    if (SWITCH_BODY.default != -1)
        gen('goto SWITCH_BODY.default');
    STMT.next := SWITCH_BODY.next;
  }
```

Y.N. Srikant   Intermediate Code Generation

What remains is the generation of the, if then statement, statements at the end of the switch. So, now, we come to statement going to switch head an switch body. So, we have all the list with a switch you know the expression etcetera is all here. So, switch head dot result contains the, you know address of the temporary which maintains the result and in switch body we have the entire list of values and pairs. So, backpatch switch head dot test to nextquad. So, this is the text goto we had generated at the of switch head. So, that must that goto nextquad which is nothing but the beginning of the series of tests now we iterate through the switch body dot list. So, for each value jump pair in switch body dot list we comma j equal to you know the next value comma jump pair from switch body dot list.

So, we take the next value pair value jump pair assign it to v comma j. Now, generate if switch dot result E equal to v goto j. So, for each value jump pair we generate this. So, automatically we would have generated something like this. So, all the stress would have been generated at the end if there is a there was a default you know within the list. Then we generate goto switch body dot default if there is not any we simply add switch statement dot next equal to you know. We do not generate any of this and simply says a statement dot next equal to switch body dot next nothing else written. So, the list of test and the last you know default are both taken care of in this set of in this particular production.

(Refer Slide Time: 29:36)



C For-Loop

The for-loop of C is very general

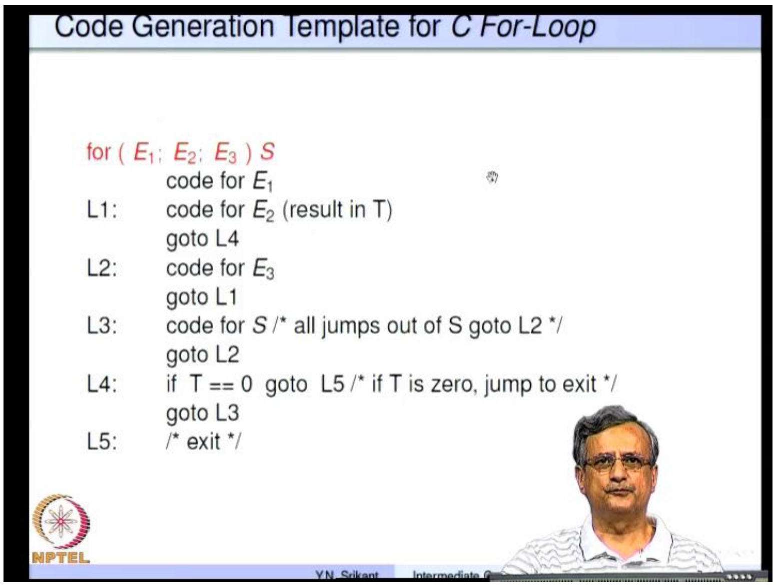
- for (*expression*$_1$; *expression*$_2$; *expression*$_3$) *statement*
  This statement is equivalent to
  expression$_1$;
  while (*expression*$_2$) { *statement expression*$_3$; }
- All three expressions are optional and any one (or all) may be missing
- Code generation is non-trivial because the order of execution of *statement* and *expression*$_3$ are reversed compared to their occurrance in the for-statement
- Difficulty is due to 1-pass bottom-up code gen
- Code generation during parse tree traversals this problem by generating code for *expression*$_3$ b t of *statement*

So, that completes these switch statement now, the next very important statement is the for loop in C the for loop in C is a very general for loop. So, here is a, for then we have 3 expressions expression 1 expression 2 and expression three followed by a statement. So, here the all any 1 are all of the 3 expressions can be missing the manual say. So, you know so, you could simply have for semicolon statement. So, in that case the statement keeps running forever otherwise the, this for statement is equivalent to these 2 together expression 1 is the initialization. So, it is executed first and then the expression 2 part is the test part.

So, there is a while loop expression 2 then we execute the statement followed by a expression 3 which corresponds to the increment in general. So, statement followed by expression 3. So, this while loop is repeatedly executed until expression 2 becomes 0. So, the code generation becomes a bit default why if you look at this equivalence we have expression 2 expression 1. No problem we can generate the code for that followed by that is the expression 2. We can generate the code for expression 2 also no problem at all, but then we have statement and then expression 3. So, whereas, in the source we have expression three followed by statement.

So, because of this the code generation becomes a little more difficult we need to introduce a extra jump statements and all this happens. Because we want to generate code in 1 pass during l r parsing. Suppose we did not have to do that we had the parse tree already and this intermediate code was being generated by a walk over the parse tree. Then you know we would have had the false statement node having expression 1 expression 2 expression 3 and expression 4 has children. So, we could have visited them in the order expression 1 expression 2 statement. And then expression 3 and generate a code appropriately, but in 1 pass bottom up parsing we really cannot do that.

(Refer Slide Time: 32:40)



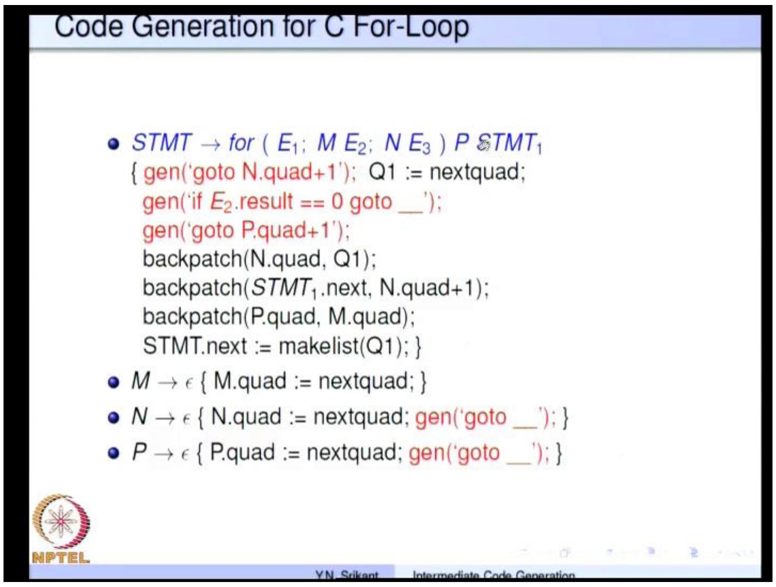
Code Generation Template for *C For-Loop*

```
for ( E1; E2; E3 ) S
        code for E1
L1:     code for E2 (result in T)
        goto L4
L2:     code for E3
        goto L1
L3:     code for S /* all jumps out of S goto L2 */
        goto L2
L4:     if  T == 0  goto  L5 /* if T is zero, jump to exit */
        goto L3
L5:     /* exit */
```

So, here is the code template for the c for loop for E 1 semicolon E 2 semicolon E three s. So, code for E 1 and then the code for E 2. Let us say the result is in the temporary T and then we need to have a gotogoto L 4 why we need to actually generate a test if T equal to 0. Then something that is where the goto takes you then we have the code for E three right so, but the when you actually and then we have the goto for L 1 code for S and then the goto L 2 then test goto L 3 and exit. So, let us look at the flow of control. So, this is executed first this is executed next.

Then we go to l 4 execute the test if t equal to 0 goto l 5 l 5 would be the exit part otherwise we goto l 3; that means, we execute the code for S then goto l 4 2. That means, we execute the code for E three and then goto L 1 which is the beginning of code for E 2. So, because of the order in which we need to generate code and the order that we need to generate code is exactly the order in which the source appears. We intersperse the code with goto statement in order to get the appropriate flow of control.

(Refer Slide Time: 34:28)



Code Generation for C For-Loop

- $STMT \rightarrow for\ (\ E_1;\ M\ E_2;\ N\ E_3\ )\ P\ STMT_1$

```
{ gen('goto N.quad+1'); Q1 := nextquad;
  gen('if E2.result == 0 goto __');
  gen('goto P.quad+1');
  backpatch(N.quad, Q1);
  backpatch(STMT1.next, N.quad+1);
  backpatch(P.quad, M.quad);
  STMT.next := makelist(Q1); }
```

- $M \rightarrow \epsilon$ { M.quad := nextquad; }
- $N \rightarrow \epsilon$ { N.quad := nextquad; gen('goto __'); }
- $P \rightarrow \epsilon$ { P.quad := nextquad; gen('goto __'); }

Y.N. Srikant Intermediate Code Generation

So, here is our production statement going to for E 1 semicolon m E 2 semicolon n E three followed P S T M T 1 we have a couple of markers M N and P. So, as usual m does not generate code it simple remembers the beginning of the expression E 2. Obviously, that is necessary, because we need to jump to the beginning of E 2 and evaluate it after the entire statement sequence is executed. So, M generates a goto and P also generates a goto let us see how what kind of code is generated. So, in bottom up parsing we would have parse this entire thing and generated a code for E 1 E 2 E 3then M N P and statement 1. And we have we will be here just before reduction we need to execute this action.

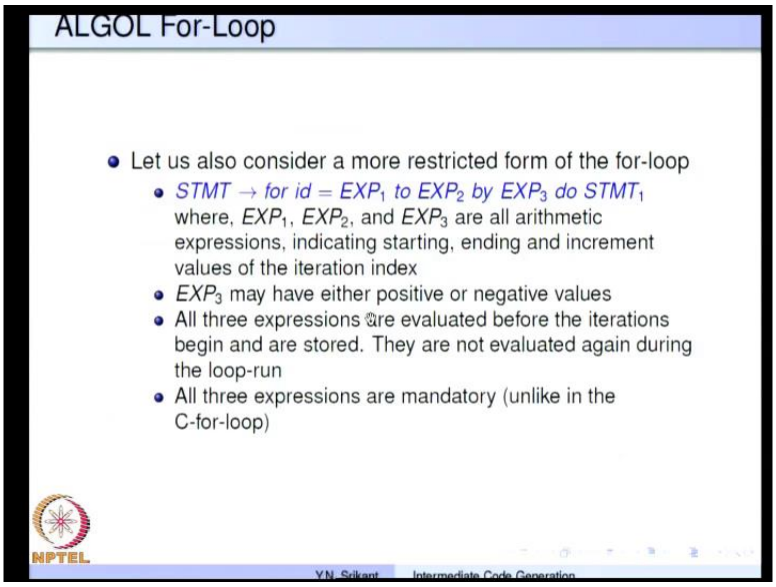
So, what does that action do? So, we have executed statement 1 there is a code goto n dot quad plus 1. So, n dot quad is the, a you know goto statement here. So, plus 1 would be the beginning of E 3. So, that is correct right. So, we have you know we goto. So, this after the statement 1 is completed we are here right code for S is over so, we goto the beginning of E 3. So, right goto l 2 that is the beginning of the E 3 that is we are going. So, n dot quad plus 1 is the beginning of the E 3 that is where we jump next q 1 remembers this quadruple.

So, what is this quadruple we have a test if E 2 dot result equal to 0 goto something. So, automatically we would have jump to this place and we will see where from where we jumped to just assume that Q 1 remembers this goto statement and that is also according what we wanted. So, here is this goto generated just now and here is the comparison that we have generated now we need to generate another goto. So, now, goto p dot quad plus 1. So, if E dot result is not equal to 0 then we goto dot quad plus 1 p dot quad is nothing but this goto statement and the beginning of the statement 1 S P dot quad plus 1.

So, if the result is greater than 0 or less than 0 whatever it is definitely not equal to 0 then we jump to the beginning of the statement that is what we wanted here right. So, goto l 3. So, beginning of code for S so far we have justified the generation of these three codes, but we are still not filled up this now we backpatch end of quad with Q 1. So, what is n dot quad? n dot quad is this goto statement and so, that would be filled with Q 1. So, we have a evaluated E 2 now this goto takes you to the beginning of the Q 1 is nothing but this quad. So, this, the goto in n takes you here. So, which is correct again?

So, we can see that after this E 2 right we have jump to l 4 which is nothing but the test for T then we backpatch statement 1 dot next with n dot quad plus 1. So, statement 1 dot next is all the jumps out are the all the jumps out of statement 1. So, that is patch to the beginning of E 3 N dot quad plus 1 is the beginning of E 3. So, we must execute E 3. So, this jump this backpatch is also correct then we have backpatch P dot quad and M dot quad out offset P dot quad is this goto which is generated here right and that is to the beginning of the E 2 again. So, at the end of E 3, we have a goto to l 1. So, code for E tow is executed all over again. So, then of course, statement dot next is nothing but the jump out of the, this when E dot result equal to 0. So, it is makelist Q 1. So, this is the justification for this code generation for the, for loop in C.

(Refer Slide Time: 39:24)



ALGOL For-Loop

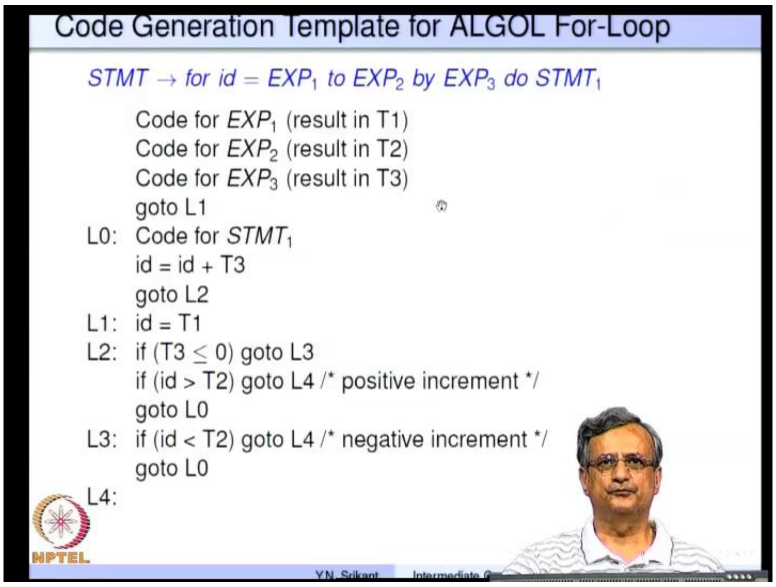
- Let us also consider a more restricted form of the for-loop
  - *STMT* → *for id* = EXP_1 *to* EXP_2 *by* EXP_3 *do* $STMT_1$ where, EXP_1, EXP_2, and EXP_3 are all arithmetic expressions, indicating starting, ending and increment values of the iteration index
  - EXP_3 may have either positive or negative values
  - All three expressions are evaluated before the iterations begin and are stored. They are not evaluated again during the loop-run
  - All three expressions are mandatory (unlike in the C-for-loop)

Y.N. Srikant Intermediate Code Generation

So, there are other varieties of for loop in other programming languages. So, let us take one variant which is very popular. So, this is the ALGOL for loop this is also available in part of it partially it s available in Pascal as well. So, let us also look at this note that the syntax is statement going to for i d equal to expression 1 to expression 2 by expression 3 2 statement 1. So, expression 1 2 and 3 are all arithmetic statements. So, what we really do is this is a, we assume that this is the beginning value i d; this is the name i d; this is the ending value of the name and this is the increment. So, we start with this value run the loop and every time we check whether we have reached expression 2 and we increment or decrement based on the value of exp 3 and then execute the statement 1 if necessary. So, exp 3 can have either positive or negative value.

So, if the value of exp 3 positive then you know the expression will be lower than expression 2 usually we rise from an expression 1 to expression 2 analytically increasing order and if expression 3 has a negative value then usually expression 1 is higher and expression 2 is lower. So, we go down from expression 1 to expression 2 in decreasing order. And then and during this course execute this statement another important thing is expression 1 2 and 3 are all evaluated only once. Whereas, in the case of C loop the expression tow and expression 3 where evaluated every time in every iteration of the loop. So, this does not happen in the ALGOL loop they are executed only once and values are stored. Finally, all the 3 expressions are mandatory it is not that you can skip any 1 or more of them.

(Refer Slide Time: 41:41)



Code Generation Template for ALGOL For-Loop

$STMT \rightarrow$ *for id* $=$ EXP_1 *to* EXP_2 *by* EXP_3 *do* $STMT_1$

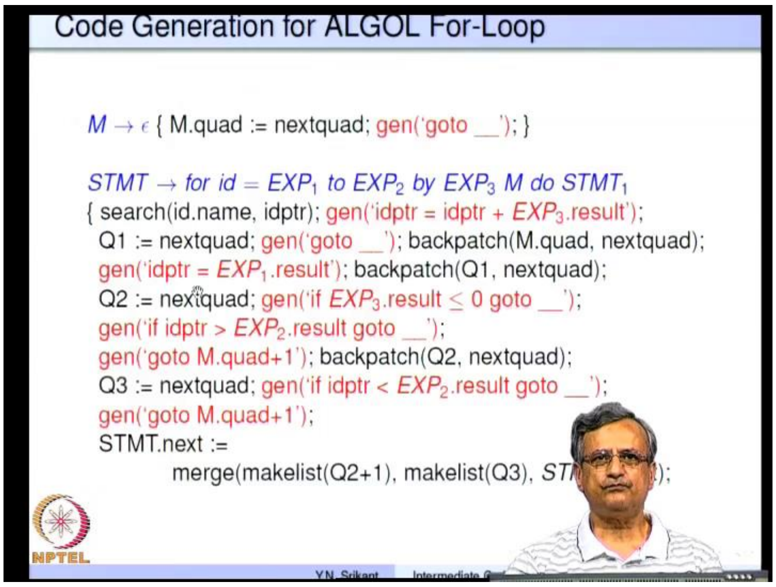
```
     Code for $EXP_1$ (result in T1)
     Code for $EXP_2$ (result in T2)
     Code for $EXP_3$ (result in T3)
     goto L1
L0:  Code for $STMT_1$
     id = id + T3
     goto L2
L1:  id = T1
L2:  if (T3 $\leq$ 0) goto L3
     if (id > T2) goto L4 /* positive increment */
     goto L0
L3:  if (id < T2) goto L4 /* negative increment */
     goto L0
L4:
```

Y.N. Srikant Intermediate C

So, what is the code template? So, you have for i d equal to expression 1 to expression 2 by expression three do statement 1. So, we have the code for the expression 1 result in T 1 code for expression 2 result in T 2 code for expression three result in T 3. So, that is in the same order no problem at all in that now we have the statement 1. So, before that we have a jump. So, and then we have the code for statement 1 then the increment i d equal to i d plus T 3 and then finally, goto L 2 we checks whether the value of T 3 is less than or equal to 0. So, or it is greater than 0. So, the reason is T 3 could be positive or negative and based on that value the termination condition will also be different. So, let us then there are 2 tests 1 for the positive increment and the other for the negative increment and finally, we go back to goto l 0 execute the statement and repeat.

So, let us look at the flow of control here first of all the control executes and evaluates all the 3 expressions. Then we goto l 1 this is an initialization for the name i d then we check whether the increment is negative. So, if so we goto l 3 if the increment is negative then which, check whether i d is less than T 2 that means, if it is less than the we need to jump of the loop we have finished the loop. Otherwise goto l 0 code for statement 1 is executed then the increment is added and we go back to the test again similarly if the increment was positive. We check whether i d greater than T 2 and jump out if it is greater otherwise we go back to l 0 execute the statement 1 etcetera.

(Refer Slide Time: 43:52)



Code Generation for ALGOL For-Loop

```
M → ε { M.quad := nextquad; gen('goto __'); }

STMT → for id = EXP1 to EXP2 by EXP3 M do STMT1
{ search(id.name, idptr); gen('idptr = idptr + EXP3.result');
  Q1 := nextquad; gen('goto __'); backpatch(M.quad, nextquad);
  gen('idptr = EXP1.result'); backpatch(Q1, nextquad);
  Q2 := nextquad; gen('if EXP3.result ≤ 0 goto __');
  gen('if idptr > EXP2.result goto __');
  gen('goto M.quad+1'); backpatch(Q2, nextquad);
  Q3 := nextquad; gen('if idptr < EXP2.result goto __');
  gen('goto M.quad+1');
  STMT.next :=
          merge(makelist(Q2+1), makelist(Q3), ST[illegible]);
```

Y.N. Srikant

So, how does the code get generated in syntax translation. So, we have statement then for i d equal to expression 1 to expression 2 expression 3. We do not need markers among these simple, because there is no need to jump to the beginning or end of exp any of these expressions we they are evaluates only once. But we definitely require a marker to remember the beginning of the statement and also to generate a goto at the end of expression 3. So, remember there is a goto at the end of the expression 3. So, M remembers the quadruples which is nothing but a goto here now the name i d searched for in the symbol table and we generate the increment. So, we are here, we have finished statement code generation for statement 1 also now we generate the increment statement and also a jump.

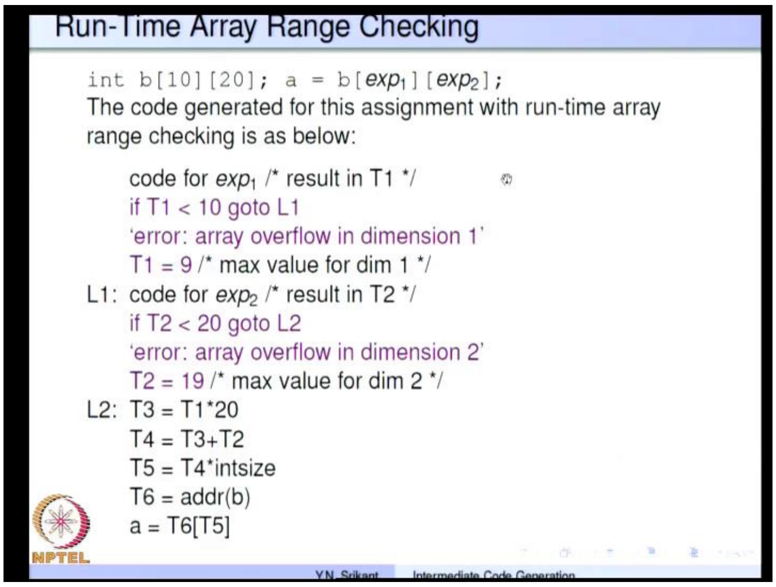
So, we generate the increment statement then we remember this Q 1 which is the next quad generate a jump here. Then since we had generated a goto here in the marker M and now we know where to patch that. So, for example, here this is the goto generated by the marker and that is supposed to goto the initialization statement i d equal to T 1. So, for we have generated all these codes we are yet to generate this. So, we generate i d p t r equal to expression 1 dot result and this is the quad to which M dot quad was backpatch. So, the goto has been now patch that goto has been patched properly next we patch backpatch Q 1 with next quad and remember next Q 2 as nextquad as well. So, what is the next quadruple? This is the test if expression 3 dot result less than equal to 0 goto.

So, for example, we have generated this. So, we are generating T 3 less than equal to 0 goto. So, that is what we are generating and we have patched Q 1 which is nothing but this goto to come here. So, remember after the increment statement the goto suppose to goto the test. So, we so, that is what we had done here. So, increment and then goto l 2, that is the test part. Now, if it is not less or equal to 0 then it means it is a positive increment. So, we check whether i d p t r greater than expression dot result goto and then this is yet to this is the exit part this will go on to the statement dot next. So, we can see that. So, Q 2 this is q 2 this is Q 2 plus 1. So, we have put Q 2 plus 1 also on statement dot next then after this if the exit is not going to take place. We generate a goto to the beginning of the M you know the statement which is nothing but M dot quad plus 1.

So, m dot quad is this. So, m dot quad plus 1 is statement 1. So, we generate that goto here and we backpatch Q 2 which is nothing but again this quadruple right with the nextquad. So, nextquad is again the check whether i d p t r has gone beyond its limit in the case of the negative increment. So, that is what we have done here you know. So, we backpatch that into this and finally, generate a goto to the beginning of the statement if we have not cross the limit. So, statement dot next contains q 2 plus 1 then q three which is this this also an exit this is 1 exit this is the other exit statement 1 dot next would be third possible exit. So, all these three exits are put on statement dot next. So, this is the code generation a strategy for the ALGOL for loop it is also possible to generate code for the ALGOL for loop in a slightly different way, but this requires a little extra work.

So, for example, generation of code for a expression 1 expression 2 and expression 3 straightforward. So, that will not even worry about now we try to do the initialization of i d straight way here before we check whether the T 3 is less than or equal to 0 or greater than equal to 0. Now, we check whether it is less than 0 or if row goto l 2 if it greater than T 2 then goto l 4 etcetera. So, the order in which we have generated this code is slightly different from the order in which we have generated code for the other one. So, in so, this would be the left as an exercise for students to do. So, the hint is we may have to break the production appropriately and insert more markers in order to generates this code properly.

(Refer Slide Time: 49:57)



Run-Time Array Range Checking

`int b[10][20]; a = b[`exp_1`][`exp_2`];`

The code generated for this assignment with run-time array range checking is as below:

```
    code for exp1 /* result in T1 */
    if T1 < 10 goto L1
    'error: array overflow in dimension 1'
    T1 = 9 /* max value for dim 1 */
L1: code for exp2 /* result in T2 */
    if T2 < 20 goto L2
    'error: array overflow in dimension 2'
    T2 = 19 /* max value for dim 2 */
L2: T3 = T1*20
    T4 = T3+T2
    T5 = T4*intsize
    T6 = addr(b)
    a = T6[T5]
```

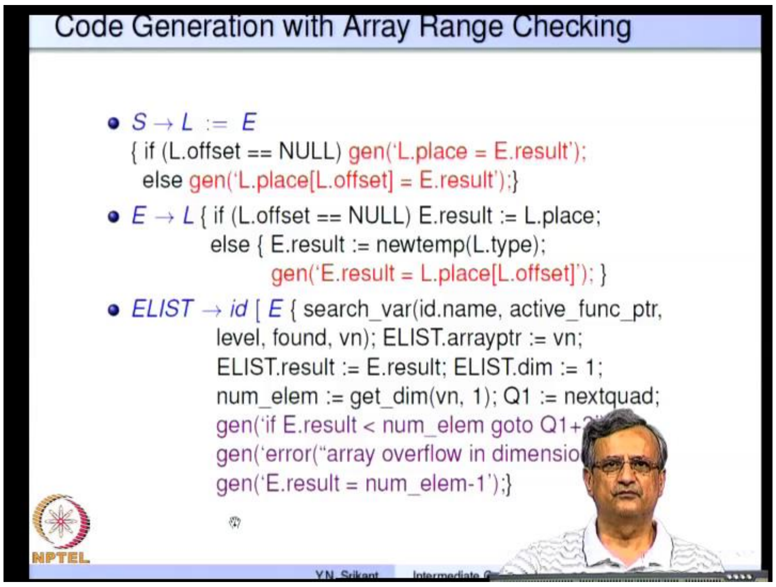
Y.N. Srikant Intermediate Code Generation

So, now it is also time to look at you know a important aspects of checking the range over flow in arrays. So, let us look at an example. So, here is int b 10 20 for 2 dimensional array. And here is an assignment a equal to b of exp 1 exp 2 what we want to do is make sure that expression 1 is always less than 10 remember c arrays run from 0 to 9 and this part runs from 0 to 19. So, we want to make sure that the exp 1 is less than 10 strictly and exp 2 is less than 20 strictly if it is not we want to issue errors right. So, the code generation generated code template would be something like this code for expression 1 result in T 1. No problem now before generating the code for expression 2 we check whether T 1 is less than 10 if so goto l 1.

So, this is normal course of action if T 1 is equal to 10 or greater than 10. An error message error array overflow in dimension 1 is issued T 1 is reset to the maximum value of dimension 1 which is 9 0 to 9 is the array index range. So, we set it to nine something very similar happens for expression 2 code for expression 2 result in T 2. If T 2 is less than 20 it is normal course of action otherwise we issue the error array overflow in dimension 2 set t maximum value of T 2 as 19 that is the maximum for the dimension 2 part and then we proceed. So, since we have set it to 9 and 9 there are no more errors, but the results that we obtain by the competition may be wrong. But at least the program will try to execute to completion even though it may not produce the right value.

So, T 3 is equal to T 1 another option possible here is to just exit the program, but if we had exited here we would not have caught this error that is the reason why we have set it to maximum and then try to continue. So, then the it is normal code T 3 equal to T 1 star 20 and then T 4 equal to T 3 plus T 2. And so, we multiplied by the number of elements in the second dimension. Then add the second dimension value subscript value that is T 2 then multiplied by the element size. So, T 4 into int size finally, a equal to T 6 of T 5. So, this is the code that is generated.

(Refer Slide Time: 53:15)



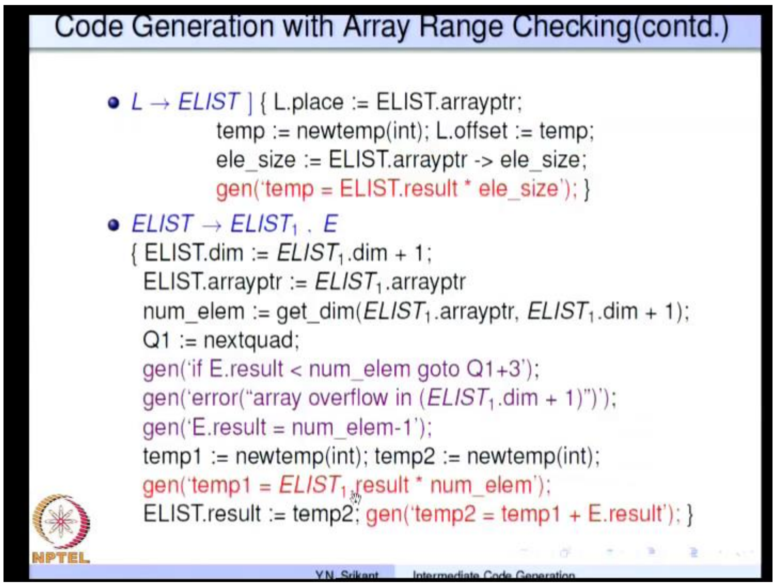
Code Generation with Array Range Checking

- $S \rightarrow L := E$
  { if (L.offset == NULL) gen('L.place = E.result');
  else gen('L.place[L.offset] = E.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]'); }
- $ELIST \rightarrow id$ [E { search_var(id.name, active_func_ptr,
  level, found, vn); ELIST.arrayptr := vn;
  ELIST.result := E.result; ELIST.dim := 1;
  num_elem := get_dim(vn, 1); Q1 := nextquad;
  gen('if E.result < num_elem goto Q1+
  gen('error("array overflow in dimensio
  gen('E.result = num_elem-1');}

So, here is the, you know syntax directed translation for it. So, these two are the same as before they have been reproduced here to just to set the context properly S going to l colon equal to E. So, this is the assignment. So, if l dot offset is null we generate simple i d. So, it is a simple i d. So, we generate l dot place equal to E dot result otherwise we set generate a an indexed assignment. So, the right hand could be an array again. So, again we check whether it is a an error or simple i d if it is an array then we generate an index assignment once more. So, this is necessary as I already mentioned an both left and right sides cannot be array assignments at the array expressions at the same time now, this part elist going to i d bracket E.

So, we search for the same i d dot name and get its pointer then that pointer is transfer to elist dot array pointer. The result of E is stored in elist dot result the dimension number is set to 1, because this is the first expression in the subscript list. We get the number of elements from the for the dimension 1 using this array pointer now, we generate the required quadruples if E dot result is less than num goto Q 1 plus 3. So, this is Q 1; this is Q 1 plus 1; this is q 1 plus 2 following that is the normal course of actions. So, that is Q 1 plus 3 otherwise generate the error and then set it to num element minus 1.

(Refer Slide Time: 55:05)



Code Generation with Array Range Checking(contd.)

```
• L → ELIST ] { L.place := ELIST.arrayptr;
          temp := newtemp(int); L.offset := temp;
          ele_size := ELIST.arrayptr -> ele_size;
          gen('temp = ELIST.result * ele_size'); }
• ELIST → ELIST1 , E
  { ELIST.dim := ELIST1.dim + 1;
    ELIST.arrayptr := ELIST1.arrayptr
    num_elem := get_dim(ELIST1.arrayptr, ELIST1.dim + 1);
    Q1 := nextquad;
    gen('if E.result < num_elem goto Q1+3');
    gen('error("array overflow in (ELIST1.dim + 1)")');
    gen('E.result = num_elem-1');
    temp1 := newtemp(int); temp2 := newtemp(int);
    gen('temp1 = ELIST1.result * num_elem');
    ELIST.result := temp2; gen('temp2 = temp1 + E.result'); }
```

Y.N. Srikant   Intermediate Code Generation

So, this is the maximum for the dimension 1, the same thing happens for other dimensions as well. So, here is the elist of subscripts that is being process so, elist going to elist 1 comma E. So, the same thing happens here we get the number of elements in the appropriate dimension elist 1 dot dim plus 1 generate the 3 codes check whether it is less than num element if. So, jump to the third expression otherwise generate an error message and set the maximum to num element minus 1 at the end of it we do the normal processing temp 1 is equal to elist 1 dot result star num element.

So, that is the number of elements in this dimension E right add the E dot result to it. So, now we are ready to goto the next level. So, this is how we process the expression. So, now, this is a ending production elist bracket. So, here we just multiplied by the element size and that is finally, the offset that we have generated for elist. So, the, we will stop the lecture here and continue with break. And continue statements in the next part of the lecture.

Thank you.