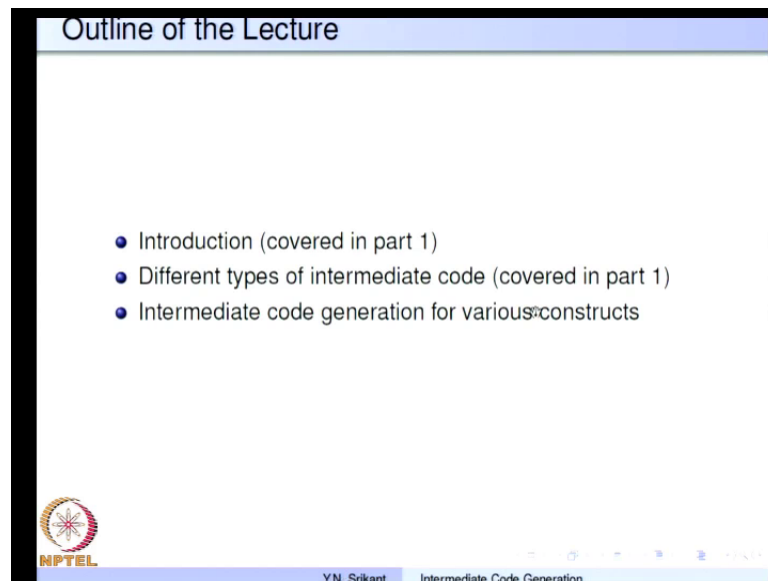


Principle of Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture - 20
Intermediate code generation Part-4
Run-time environments part-1

(Refer Slide Time: 00:19)



Outline of the Lecture

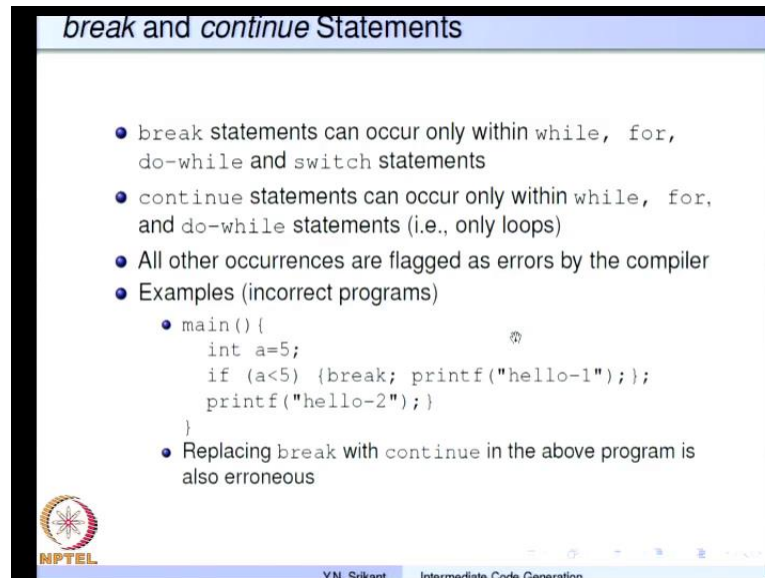
- Introduction (covered in part 1)
- Different types of intermediate code (covered in part 1)
- Intermediate code generation for various constructs

NPTEL

Y.N. Srikant Intermediate Code Generation

Welcome to part 4 of the lecture on intermediate code generation. Today, we will continue with the intermediate code generation strategy for various types of constructs.

(Refer Slide Time: 00:27)



break and *continue* Statements

- *break* statements can occur only within *while*, *for*, *do-while* and *switch* statements
- *continue* statements can occur only within *while*, *for*, and *do-while* statements (i.e., only loops)
- All other occurrences are flagged as errors by the compiler
- Examples (incorrect programs)
 - ```
main() {
 int a=5;
 if (a<5) {break; printf("hello-1");};
 printf("hello-2");
}
```
  - Replacing *break* with *continue* in the above program is also erroneous

MPTEL  
Y.N. Srikant Intermediate Code Generation

So, in the last lecture, we saw how to generate code for the loops, but we still have not covered the *break* and *continue* statements which can occur inside loops. So, for example, a *break* statement takes the control away from a loop or *switch* statement. And of course, it can occur only within the *while* for *do while* and *switch* statements. If it occurs in any other type of statements it would be flagged as an error by the compiler. A *continue* statement simply skips the rest of the iteration of the loop and goes on to the next iteration.

And since I mentioned iteration; obviously, a *continue* statements are valid only within *while* for and *do while* statements that is loops, and others are flagged as error by the compiler. So, let us look at a couple of incorrect and correct programs with *break* and *continue* to understand how to generate code for these statements, here is an incorrect program. Then `int a = 5` initialization if `a < 5` then `break` `printf` hello 1 and outside is `printf` hello 2, this would be flagged as an error by the compiler. And similarly replacing this `break` with a `continue` will also be flagged as an error by the compiler. It is not as if the `then` part is skipped and it goes in `prints` hello 2 it does not do that.

(Refer Slide Time: 02:12)

*break and continue Statements (correct programs)*


- The program below prints 6  

```
main(){int a,b=10; for(a=1;a<5;a++) b--;
printf("%d",b);}
```
- The program below prints 8  

```
main(){int a,b=10; for(a=1;a<5;a++)
{ if (a==3) break; b--;} printf("%d",b);}
```
- The program below prints 7  

```
main(){int a,b=10; for(a=1;a<5;a++)
{ if (a==3) continue; b--;} printf("%d",b);}
```
- This program also prints 8  

```
main(){int a,b=10; for(a=1;a<5;a++)
{ while (1) break;
if (a==3) break; b--;} printf("%d",b);}
```



YN, Srikant Intermediate Code Generation

Now, a couple of programs which are correct so here is a main program its 2 variables a and b, b initialized to 10 and a is uninitialized. There is a loop for starting from a equal to 1 goes on till a less than 5; that means, it executes for a equal to 1 2 3 4 a is incremented by 1 every time the loop body is completed. And the body of the loop is simply a decrement operation on b after the loop is over the value of b is printed out. So, as you can see this program will print 6 the reason being we start with a equal to 1. Then B is decremented to 9 a equal to 2 it becomes 8 a equal to 3 it becomes 7 and a equal to 4 it becomes 6 and with a equal to 5 the loop terminates. So, the value is 6 and it is printed out.

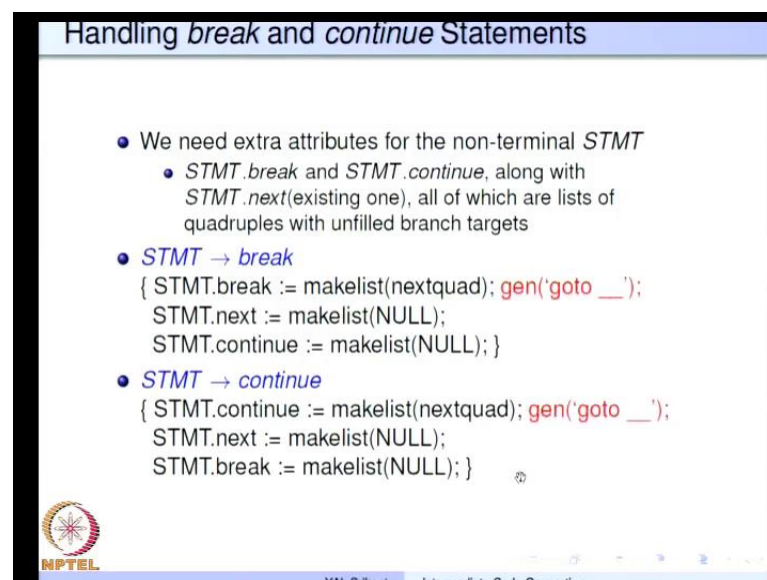
So, now let us introduce breaks and continuous into this loop. So, the same loop, but the body says if a equal to 3 break. So, in other words for the value of a equal to 3 this loop is terminated and then it does not you know iterate anymore. So, the loop would have executed for a equal to 1 and b would have become 9. It will execute for a equal to 2 and B becomes 8 and as soon as a becomes 3 the break comes into picture, The loop is broken the control comes here and prints out the value of B which is 8. So, this is how the break statement works and you must also keep in mind that the break always breaks to the breaks control and brings it to the next nested loop.

So, if there are 3 loops nested and from the innermost loop it will break to the next outer loop. And if the break is in the next outer loop then it would break to the outermost loop

and similarly, in the outermost loop a break statement will carry it outside the loop nest itself. Similarly, let us introduce a continuous a statement instead of the break. So, here the program will prints 7 the reason being with a equal to 1. The B value of B becomes 9 with a equal to 2 it becomes 8 and when a equal to 3 there is a continue. So which implies, that the control skips the rest of the iteration, and goes to the next iteration by incrementing a. So, the value of a value of b instead of getting decremented you know 4 times it will get decremented only 3 times. And thereby the value becomes 7 and the break statement can also be used to get out of infinite loops.

So, for example, the same program as here, there is a while loop with 1 which is always true. So, if there is no break, this is an infinite loop, but because of the break the while loop control goes out the control goes out of the while loop and it comes to the then else if then statement. So, the rest of the, you know iteration is carried out as before. So, it again prints the value of 8. So, every time the control comes to the beginning the while loop tries to execute and because of the break it comes out goes to the, if statement continues and so on and so forth. So, and this break of course, works as before and for a equal to 3 it gets out of the loop itself. So, this is how the, you know continue and break statements work in C.

(Refer Slide Time: 06:20)



The slide is titled "Handling *break* and *continue* Statements". It contains a bulleted list of points and code snippets. The first bullet point states that extra attributes are needed for non-terminal STMT, specifically *STMT.break*, *STMT.continue*, and *STMT.next* (existing one), all of which are lists of quadruples with unfilled branch targets. The second bullet point shows the code for *STMT → break*, which sets *STMT.break* to a list of quadruples, *STMT.next* to NULL, and *STMT.continue* to NULL. The third bullet point shows the code for *STMT → continue*, which sets *STMT.continue* to a list of quadruples, *STMT.next* to NULL, and *STMT.break* to NULL. The slide also features the NPTEL logo in the bottom left corner and the text "Y.N. Srikant Intermediate Code Generation" in the bottom right corner.

- We need extra attributes for the non-terminal *STMT*
  - *STMT.break* and *STMT.continue*, along with *STMT.next* (existing one), all of which are lists of quadruples with unfilled branch targets
- *STMT → break*

```
{ STMT.break := makelist(nextquad); gen('goto __');
 STMT.next := makelist(NULL);
 STMT.continue := makelist(NULL); }
```
- *STMT → continue*

```
{ STMT.continue := makelist(nextquad); gen('goto __');
 STMT.next := makelist(NULL);
 STMT.break := makelist(NULL); }
```

So, let us see how to generate code for such statements. So, because of the special nature and the control flow, you know they are because of the disruption of the control flow.

We need to maintain extra attributes for the non terminal statement one of them is statement dot break again these are synthesized attributes. And the other is statement dot continue, the attributes statement dot next which is already existing will of course, be present here as well.

So, all these 3 actually are lists of quadruples with unfilled branch targets for the production statement going to break. We simply generate a goto statement and put it on the break list next becomes null. Because control cannot proceed to the next statement after break it goes out of the loop and similarly, there is no continue this is a break statement. So, this is also null for the continue again we generate a goto statement and put in on the continue list next and break become null. So, even though these 2 you know statements generate the same code goto the patching that is required for the target of this goto will be different for the break and different for the continue.

(Refer Slide Time: 07:59)


**SATG for *While-do* Statement with *break* and *continue***

- $WHILEEXP \rightarrow \textit{while } M E$ 

```
{ WHILEEXP.falselist := makelist(nextquad);
 gen('if E.result ≤ 0 goto __');
 WHILEEXP.begin := M.quad; }
```
- $STMT \rightarrow \textit{WHILEEXP do } STMT_1$ 

```
{ gen('goto WHILEEXP.begin');
 backpatch(STMT1.next, WHILEEXP.begin);
 backpatch(STMT1.continue, WHILEEXP.begin);
 STMT.continue := makelist(NULL);
 STMT.break := makelist(NULL);
 STMT.next := merge(WHILEEXP.falselist, STMT1.break); }
```
- $M \rightarrow \epsilon$ 

```
{ M.quad := nextquad; }
```



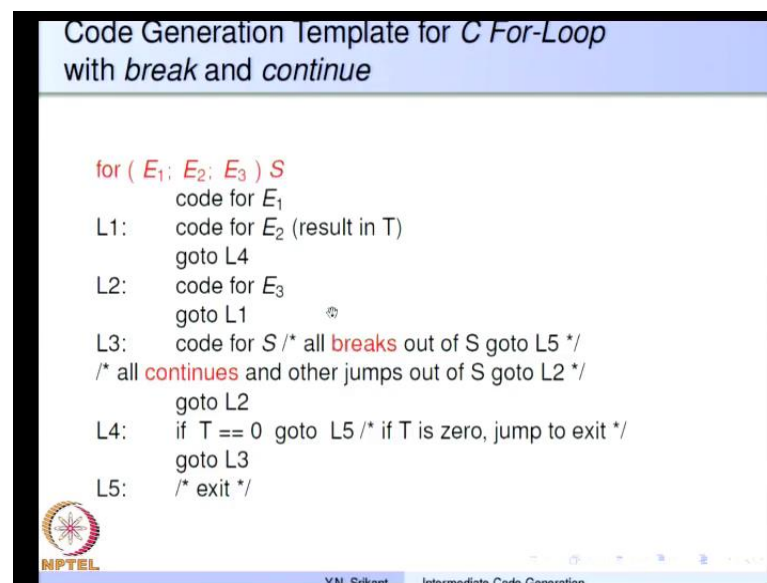
Y.N. Srikant    Intermediate Code Generation

So, let us understand how that happens in the case of a while loop. So, here is the while X going to while M E and the statement goes to whileexp do statement 1. So, this part is same as before not much difference whileexp dot falselist is nextquad. And the next quadruple generated is the test on the result of E and whileexp dot begin is M dot quad. So, we remember the beginning of the expression E and the target here will be unfilled it will be filled later. So, once we have parsed the entire while statement we will be at this point. So, now, we execute the code going to whileexp dot begin. So, which brings you

to which brings the control to the beginning of the while loop. The statement backpatch statement 1 dot next with whileexp dot begin is as before all the jumps out of statement 1 or brought to the beginning of the statement we do the same with continue.

So, if there is a continue statement within the list of statements in statement 1. We patch them to the beginning of the while loop and a statement dot continue becomes a null statement dot break also becomes null. And the statement dot next list now contains along with whileexp dot falselist it will also contain statement 1 dot break. So, all the brakes out of statement 1 really bring the control out of the while loop. So, and whileexp dot falselist of course, brings the control out of the while loop. So, therefore, these 2 are together merged and put on to statement dot next M is a marker as usual. So, this is the code that is generated for you know break statement and continuous statement in the case of while loops.

(Refer Slide Time: 10:06)



The image shows a slide titled "Code Generation Template for C For-Loop with break and continue". The slide contains the following code template:

```
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 breaks out of S goto L5 */
 /* all continues and other 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 */
```


The slide also features the NPTEL logo in the bottom left corner and the text "Y.N. Srikant Intermediate Code Generation" in the bottom right corner.

So, let us see how the code gets generated for for loop in C. So, there a code template is almost the same, but the patching that is done for the breaks and continues is slightly different. So, remember we generate a goto statement for the break and also a goto statement for the continue all the breaks out of S are patched to the exit. So, they are put actually on the statement dot next and all the continues and other jumps out of S really goto the beginning of E 3 which is to be executed after the code for S is executed.

(Refer Slide Time: 10:48)

### Code Generation for C For-Loop with *break* and *continue*

- $STMT \rightarrow \text{for} ( E_1; M E_2; N E_3 ) P STMT_1$   
{ gen('goto N.quad+1'); Q1 := nextquad;  
gen('if E2.result == 0 goto \_\_'); gen('goto P.quad+1');  
backpatch(makelist(N.quad), Q1);  
backpatch(makelist(P.quad), M.quad);  
backpatch(STMT<sub>1</sub>.continue, N.quad+1);  
backpatch(STMT<sub>1</sub>.next, N.quad+1);  
STMT.next := merge(STMT<sub>1</sub>.break, makelist(Q1));  
STMT.break := makelist(NULL);  
STMT.continue := makelist(NULL); }
- $M \rightarrow \epsilon \{ M.quad := nextquad; \}$
- $N \rightarrow \epsilon \{ N.quad := nextquad; \text{gen('goto __');} \}$
- $P \rightarrow \epsilon \{ P.quad := nextquad; \text{gen('goto __');} \}$



Y.N. Srikant    Intermediate Code Generation

So, here is the production for statement the production you know with markers is the same the actions for generating code which we saw in the last lecture remain the same, what is special for continue and break have been marked in violet. So, statement 1 dot continue is patched to N dot quad plus 1. So, this would be this goto is in dot quad and N dot quad plus 1 will be the beginning of E 3. So, that is correct you can see that here, continue will be patched to the beginning of code for E 3, then we put S statement on dot break and the Q 1 you know Q 1 is nothing but the test and jump. So, these 2 are merged and put on statement dot next, because there the exit points of the, for loop the exit can be because we have reached a 0 here. Or it could be because of the breaks within statement 1 and since the breaks and continues cannot go beyond the loop you know they actually cannot be transmitted to other loops. So, we make them null as before. So, this is how we you know generate code for break and continue.


(Refer Slide Time: 12:20)

### LATG for *If-Then-Else* Statement

Assumption: No short-circuit evaluation for E

```
If (E) S1 else S2
 code for E (result in T)
 if T ≤ 0 goto L1 /* if T is false, jump to else part */
 code for S1 /* all exits from within S1 also jump to L2 */
 goto L2 /* jump to exit */
L1: code for S2 /* all exits from within S2 also jump to L2 */
L2: /* exit */
```

```
S → if E { N := nextquad; gen('if E.result <= 0 goto __'); }
 S1 else { M := nextquad; gen('goto __');
 backpatch(N, nextquad); }
 S2 { S.next := merge(makelist(M), S1.next, S2.next); }
```



Y.N. Srikant    Intermediate Code Generation

So, now let us look at the code generation using L attributed grammars for some of the constructs. So far we have concentrated on the S attributed grammars, because they are very relevant to yacc. But you know the learning would be incomplete if we do not see how to generate code using L attributed grammars, because they introduce a few extra facilities into the code generation schema. Let us consider the expressions without short circuit evaluation. So, the first one is the, if expression if expression S 1 else S 2 the code template is the same as before. Obviously, there can be no change in the code template it is only the translation scheme which changes.

So, the code for E then the test on the result of E; the then part and then jump to the exit; the else part and the exit itself. So, this is our schema which we used for S A T G as well here, because we are permitted to introduce actions in between the production symbol S you know on the right hand side symbols on in the production on the right hand side. We can now introduce these variables N and M instead of the marker nonterminals N and M in the S A T G. How are we justified in using these variables and you k now introducing them here and using them elsewhere in the production? We are justified because a single function or procedure is created for each nonterminal.

So, for this entire production, the code for this entire production will lie within a single function or procedure. So, of these variables N and M are declared as local variables within the procedure for S they will be accessible anywhere within the procedure. And



therefore, we are justified in using them anywhere in the production as well so N equal to nextquad. So, we remember the quadruple which is generated now which is the test quadruple for E. And if the result is less than equal to 0 we have to jump out of the then jump out and goto the else part otherwise we fall through and execute S 1.

So, the code for when we call the procedure for E the code for E will get generated then this action is executed call to the procedure S will generate the code for this S 1. Then we have just before S 2, we can we generate another goto which is nothing but this goto. Then we backpatch N with nextquad N is this quadruple the test quadruple if the result is really false then we need to go to S 2 which is the else part. That is why this backpatch is correct this code is similarly, to the S C T g, we merged the quadruple M the S 1 dot next and S 2 dot next.

(Refer Slide Time: 15:45)

### LATG for *While-do* Statement


Assumption: No short-circuit evaluation for E

```

while (E) do S
L1: code for E (result in T)
 if T ≤ 0 goto L2 /* if T is false, jump to exit */
 code for S /* all exits from within S also jump to L1 */
 goto L1 /* loop back */
L2: /* exit */

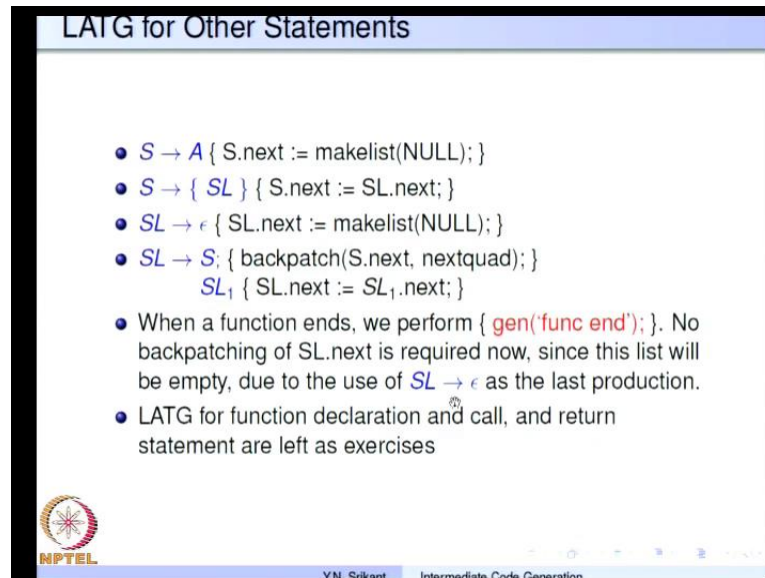
S → while { M := nextquad; }
 E { N := nextquad; gen('if E.result ≤ 0 goto __'); }
 do S1 { backpatch(S1.next, M); gen('goto M');
 S.next := makelist(N); }

```


Y.N. Srikant    Intermediate Code Generation

Now, we see how to generate code for the while loop which is very simple. So, we have code for E then the test then code for S and go back to the loop. So, again we remember the beginning of E with M equal to nextquad. And after we E we generate the test and then we generate the code for S 1 do the backpatch as S 1 dot next comma M. So, all the jumps out of S 1 really goto the beginning of the expression finally, go to M at the end of S 1. We will also generate jump to the beginning of E and S dot next will be just this quadruple. So, this is how we generate code for the while loop using the L A T G.

(Refer Slide Time: 16:46)



The slide, titled "LATG for Other Statements", contains the following content:

- $S \rightarrow A \{ S.next := makelist(NULL); \}$
- $S \rightarrow \{ SL \} \{ S.next := SL.next; \}$
- $SL \rightarrow \epsilon \{ SL.next := makelist(NULL); \}$
- $SL \rightarrow S; \{ backpatch(S.next, nextquad); \}$   
 $SL_1 \{ SL.next := SL_1.next; \}$
- When a function ends, we perform  $\{ gen('func end'); \}$ . No backpatching of  $SL.next$  is required now, since this list will be empty, due to the use of  $SL \rightarrow \epsilon$  as the last production.
- LATG for function declaration and call, and return statement are left as exercises

The slide also features the NPTEL logo in the bottom left corner and the text "Y.N. Srikant Intermediate Code Generation" in the bottom right corner.

Now, what about the rest of the statements you have still not seen the most important assignment and expression etcetera which we will see very soon. These are the other statements rather other productions and the code generation is not different from the, you know S A T G itself the code the compiler code appears here. So, a has no jumps. So, is  $S$  dot next is null here, we transfer whatever is in  $S L$  to  $S$  here again there is nothing for  $S L$  dot next. And we generate many statement, you must also observe that we use right recursion wherever we need to generate lists of statements instead of using left recursion, because using left recursion makes the grammar non L 1 1.

So, all the jumps out of  $S$  or patch to  $S L 1$  using the backpatch statement here. And  $S L$  dot next will be  $S L 1$  dot next as usual when the when a function ends we generate function end quadruple. But since we have used  $S L$  going to epsilon as the last production remember right recursion. So, this would become epsilon for the last statement which is a null statement really rather nothing not even null statement, null statement would be semicolon. So, at this point if we have used  $S L$  going to epsilon for  $S L 1$  we would have a null next list. So, there is no need for backpatching anything to this quadruple at all there is you know. It is a null list really and L A T G for function declaration call return. They are very similarly, to the S A T G and they are left as exercises.

(Refer Slide Time: 18:45)

**LATG for Expressions**

- $A \rightarrow L = E$   
{ if (L.offset == NULL) /\* simple id \*/  
  gen('L.place = E.result');  
  else gen('L.place[L.offset] = E.result'); }
- $E \rightarrow T$  { E'.left := T.result; }  
   $E' \rightarrow \epsilon$  { E'.result := E'.result; }
- $E' \rightarrow + T$  { temp := newtemp(T.type);  
  gen('temp = E'.left + T.result'); E'.left := temp; }  
   $E'_1 \rightarrow \epsilon$  { E'.result := E'\_1.result; }

Note: Checking for compatible types, etc., are all required here as well. These are left as exercises.

- $E' \rightarrow \epsilon$  { E'.result := E'.left; }
- Processing  $T \rightarrow F T'$ ,  $T' \rightarrow *F T' \mid \epsilon$ ,  $F \rightarrow ( E )$ , boolean and relational expressions are all similar to the above productions

MPTEL  
Y.N. Srikant    Intermediate Code Generation

So, now let us continue with expressions. So,  $A$  going to  $L$  equal to  $E$ , the code generation scheme is very similar, there is no difference really it is the same. And so we if it is a simple  $I d$ , we generate  $L$  dot place equal to  $E$  dot result. And if it is a you know an array expression we generate  $L$  dot place with  $L$  dot offset as  $E$  dot result. So, this is the assignment statement. So, now, let us expand the expressions. So, observe that the expression uses an unnatural form. Because the original expression we had used was left recursive  $E$  going to  $E$  plus  $T$  or  $T$  etcetera. Now, this is right recursive, because as I said your left recursive grammars cannot be used for  $L L$  parsing.

So, it becomes  $E$  going to  $T$   $E$  prime  $E$  prime going to plus  $T$   $E$  prime  $E$  prime going to epsilon etcetera. So, here we use an inherit attribute for  $E$  prime the reason is  $E$  is an expression and its code would have been generated. And  $T$  dot result is the address of the result of  $T$  then  $E$  prime is another expression. And we have still not you know operator between  $T$  and  $E$  prime that is for example, if we use  $E$  prime going to plus  $T$   $E$  prime the operate is plus this plus is actually between this  $T$  and this  $T$  right. So,  $E$  prime expands to plus  $T$   $E$  prime. So, we would really have  $T$  plus  $T$   $E$  prime.

So, we must generate code for  $T$   $E$  plus  $T$  somewhere else not in this case in not in this particular production. So, what we really do is we pass  $T$  dot result as an inherited attribute to  $E$  prime. So,  $E$  prime dot left that is the left operant goes into  $E$  prime as an inherited attribute. And  $E$  prime dot result would become  $E$  dot result eventually. So,

within the production E prime going to plus T E prime, we now generate temp equal to E prime dot left plus T dot result. So, you may remember E prime dot left is an inherited attribute and T dot result is synthesized attribute again for this E 1 prime. The result of this entire left side expression that its stored in temp will be passed as an inherited attribute and the same code generation scheme continues.


So, at the end again E prime dot result that is the left hand side synthesized attribute is E 1 prime dot result. So, I have not really shown checking for compatible types etcetera here that was shown in the S A T G and that holds here as well. So, the appropriate code for checking types you know and conversion of types etcetera has to be inserted here. And that is left as an exercise if E prime generates epsilon so; that means, there is nothing more to it.

So, for example, if this is E going to T E prime E prime going to plus T E prime. So, we would have generated T plus T E prime and if E prime goes to epsilon you would be left with T plus T. So, there is nothing to do we just say E prime dot result is E prime dot left, because we have already completed the generation of code for T plus T in this case processing the other productions with star. For example, T going to F T prime T prime going to star F T prime or epsilon F going to parenthesis E parenthesis Boolean and relation expression. They are all very similar to these productions. So, these are left as exercises

(Refer Slide Time: 23:10)

### LAI G for Expressions(contd.)

- $F \rightarrow L$  { if (L.offset == NULL) F.result := L.place;  
else { F.result := newtemp(L.type);  
gen('F.result = L.place[L.offset]'); }
- $F \rightarrow num$  { F.result := newtemp(num.type);  
gen('F.result = num.value'); }
- $L \rightarrow id$  { search(id.name, vn); INDEX.arrayptr := vn; }  
INDEX { L.place := vn; L.offset := INDEX.offset; }
- $INDEX \rightarrow \epsilon$  { INDEX.offset := NULL; }
- $INDEX \rightarrow [$  { ELIST.dim := 1;  
ELIST.arrayptr := INDEX.arrayptr; }  
ELIST ]  
{ temp := newtemp(int); INDEX.offset := temp;  
ele\_size := INDEX.arrayptr -> ele\_size;  
gen('temp = ELIST.result \* ele\_size'); }



YN\_Srikant    Intermediate Code Generation

So, now let us concentrate on the expression. So, the F which is the right side of an assignment can go to L I can generate an array expression. So, this part is very simple if it is offset is null; that means, it is a simple i d we transfer L dot place as F dot result. Otherwise if this is an array expression, we need to generate code, because as I already said both left hand side and right hand side of an arithmetic of an assignment statement cannot contain array expressions. So, we generate a new temporary and then generate the instruction f dot result equal to L dot place with L dot offset num.

Of course, we generate as you know temporary and place, the value num dot value into f dot result that is the new instruction which is generated now the array part. So, L going to i d followed by index. So, if index goes to epsilon then the, it is a simple i d and with index goes to epsilon the offset of the expression subscript part would be null. So, which is correct so what we really do is search for the name here get its entry in the symbol table. So, v N is the pointer to it, pass it as an inherited attribute to this index. So, because that will be required within the subscripts to you know get the dimension of the array etcetera and then the index is passed. So, L dot place becomes the array pointer and L dot offset is index dot offset which is generated during this index parsing of this index. So, we already saw index going to epsilon.

So, if index you know goes to right left bracket elist right bracket; that means, we have seen all the subscripts. So, far elist is yet to be expanded. Of course, now, elist contains the entire offset in terms of the number of elements of the array and we need to multiply that with the size of the array element. So, we generate an instruction temp equal to elist dot result star ele size where size is the size of the array element which is obtained using the array pointer passed to index as an inherited attribute. So, here elist gets 2 inherited attributes 1 is the initialized dim attribute. So, dim is set an number of the first dimension is will be scanned first. That is why elist dot dim is 1 array pointer is of course, of obtained from index dot array pointer and passed to elist as an inherited attribute. Here we generate a new temporary and then you know generate the quadruple for that particular elist dot result star ele size operation.

(Refer Slide Time: 26:35)

LATG for Expressions(contd.)

- $ELIST \rightarrow E$  { INDEXLIST.dim := ELIST.dim+1;  
INDEXLIST.arrayptr := ELIST.arrayptr;  
INDEXLIST.left := E.result; }
- $INDEXLIST \rightarrow \epsilon$  { ELIST.result := INDEXLIST.result; }
- $INDEXLIST \rightarrow \epsilon$  { INDEXLIST.result := INDEXLIST.left; }
- $INDEXLIST \rightarrow ,$  { **action 1** }  
 $ELIST$  { gen('temp = temp + ELIST.result');  
INDEXLIST.result := temp; }

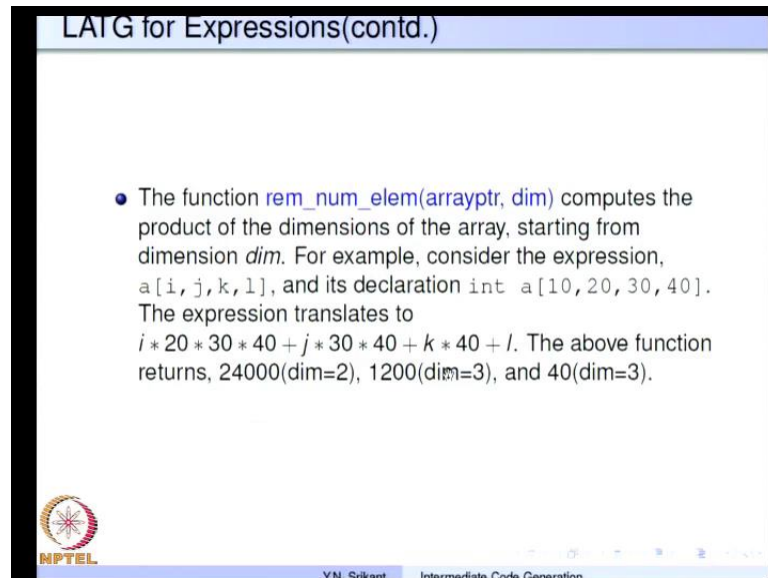
**action 1:**  
{ temp := newtemp(int);  
num\_elem := rem\_num\_elem(INDEXLIST.arrayptr,  
INDEXLIST.dim);  
gen('temp = INDEXLIST.left \* num\_elem');  
ELIST.arrayptr := INDEXLIST.arrayptr;  
ELIST.dim := INDEXLIST.dim; }

MPTEL  
Y.N. Srikant Intermediate Code Generation

So, here the expression list is being expanded. So, expression list will generate E followed by index list. So, if there is only 1 expression then index list goes to epsilon if there is more than one expression then it generates comma followed by elist and elist will again you know is guarantee to generate at least one more expression. So, this is the way we generate a number of subscripts. So, this is; obviously, you know one of the subscripts. So, what we really do is for index list we increment the dimension elist dot dim and pass it as an inherited attribute, because we have seen the assumed that this is the first dimension. So, you have seen it now whatever is going to be seen in index list will be the second dimension. So, it will be added 1 will be added to it and passed to this the array pointer is just copied straightaway and index list at left would be E dot results.

So, we get E dot result into index list as an inherited attribute and elist dot result would be index list dot result. So, that is the so here whatever is generated by index list will also take care of E. Because we have passed index list dot left you know equal we have this E dot result which is passed to index list left. As an inherited attribute if the index list is epsilon then you know index list result is index list dot left. So, nothing to do really whereas, if it expands the, we are guarantee to generate at least one more expression. So, here after the comma we have an actions I have written it here simply to make it clear not that it is required by the compiler that way. So, action 1 generates a temporary now there is a very important part here we must get the number of elements as the remaining number of elements in the subscript list.

(Refer Slide Time: 28:55)



The slide is titled "LATG for Expressions(contd.)" and contains a single bullet point. The bullet point describes the function `rem_num_elem(arrayptr, dim)`, which computes the product of the dimensions of the array starting from dimension `dim`. It provides an example with the expression `a[i, j, k, l]` and its declaration `int a[10, 20, 30, 40]`. It explains that the expression translates to  $i * 20 * 30 * 40 + j * 30 * 40 + k * 40 + l$ . The function returns 24000 for `dim=2`, 1200 for `dim=3`, and 40 for `dim=3`. The slide also features the NPTEL logo in the bottom left corner and the text "Y.N. Srikant Intermediate Code Generation" in the bottom right corner.

So, let us understand what that is. So, the function `rem num elem` is given the array pointer and the dimension at which it has to look at the data structure in the symbol table. So, it computes the product of the dimensions of the array starting from the dimension `dim`. For example, if you have a `i j k l` 4 dimensional array and this is the these are the subscripts simple subscripts. Of course, and the declaration of this array is a `10 20 30 40`, let us say `int` now, this expression will the subscripts will translate to `i` multiplied by the rest of the dimension 3 of them `20 30 40`. So, these are to be skipped then `J` star thirty forty. So, again that is within `j k star 40` within `k` and then add `l` to it.

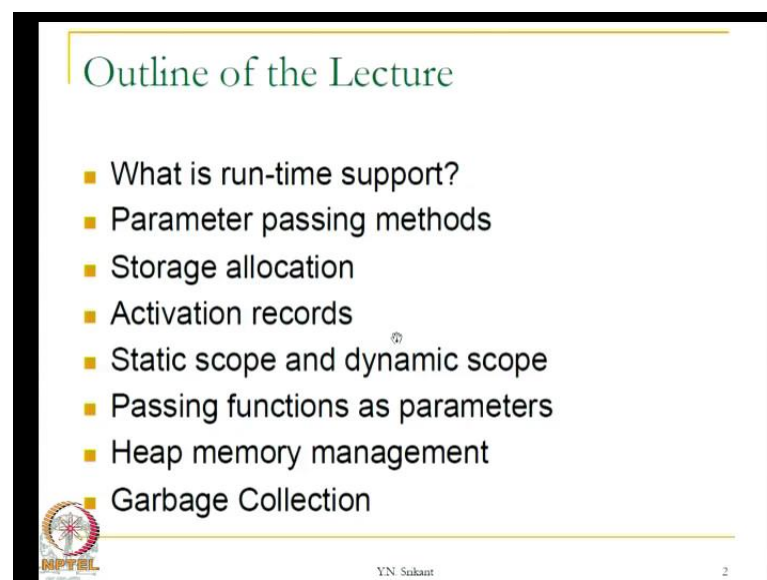
So, this is the effective subscript you know that is necessary for translating a `i j k l`. Of course, this entire subscript has to be multiplied by the `int` size and then that becomes the index and address of `a` becomes the base address of the array. So, we really this `rem num elem` when the number of dimensions you know the dimension count is 2. It returns 20 into 30 into 40 that is 24000 when `dim` is 3 it generates 1000 result it returns 1200 and when `dim` is 3 it returns 40. So, this is the way `rem num` works. So, what we really do now is get the number of elements starting from this dimension and that is `num elem` we multiply it with index list dot left which is obtained as an inherited attribute here right. So, that is this expression which is obtained.

So, each star `num elem` that is what we would be doing. So, we would be really doing this is `E star` this is `rem num elem`. So, that is what we are going to do successively then

we take  $j$  and do this multiplication and. So, on we passed the array pointer index list dot array pointer to elist dot array pointer and index list dot dim is passed to elist dot dim. So, these are the inherited attributes of elist. So, now, we we have completed elist so we generate an instruction to add elist dot result to this temporary. So, that is the same temporary which was generated here. So, we did index list dot left star num elem now we add elist dot result and that is passed back as index list dot result.

So, in essence if we had 2 dimensions for the array this would take care of the first dimension and multiplying it with the number of elements that is the second dimension is done here. Then you know the second dimension is taken care of by index lists which again goes to elist and the E. So, we generate the code elist dot temp equal to temp plus elist dot result. So, in you know we would really have done  $E$  star second number of elements in the second dimension plus  $E$  for the second dimension itself. So, that is the offset finally, we really have to pass it back and get the you know get into index where we generate the tem equal to elist dot result star ele size. So, this is how the code generation happens using L attributed grammars. So, that brings us to the end of intermediate code generation. So, now, we will begin with the next part in compiler design. So, welcome to the lecture on run time environments.

(Refer Slide Time: 33:44)



Outline of the Lecture

- What is run-time support?
- Parameter passing methods
- Storage allocation
- Activation records
- Static scope and dynamic scope
- Passing functions as parameters
- Heap memory management
- Garbage Collection

MPPEL  
Y.N. Srikant 2

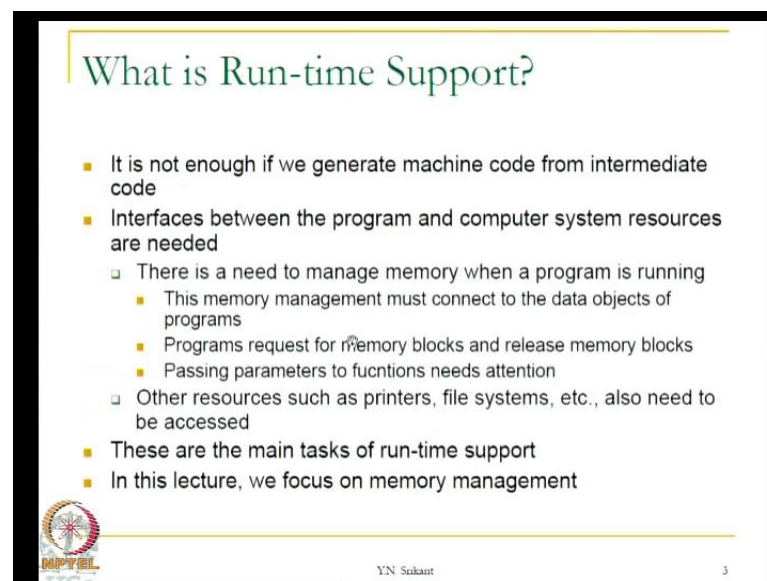
So, far we have seen you know the lexical analysis which takes care of processing characters. And then we saw parsing which takes care of processing the tokens and



producing the syntax tree etcetera. Then we saw semantic analysis which checks whether the semantics of the programming language are satisfied by the program. And then recently we also saw intermediate code generation for various constructs of the language. Now before we move on to the machine code generation, we must understand what exactly is required for a program to execute at the time of, you know when it is put into memory and the execution begins.


So, this type of run time arrangement is required and the code generator expects that these are available at run time these services are available at run time. So, we must understand what is run time support. And then we will understand the various parameter passing methods different types of storage allocation; the format of activation records; the difference between static scope and dynamic scope how to pass functions as parameters. Finally, we will see heap memory management and garbage collection. So, these are the various things that are done at run time.

(Refer Slide Time: 35:28)



**What is Run-time Support?**

- It is not enough if we generate machine code from intermediate code
- Interfaces between the program and computer system resources are needed
  - There is a need to manage memory when a program is running
    - This memory management must connect to the data objects of programs
    - Programs request for memory blocks and release memory blocks
    - Passing parameters to functions needs attention
  - Other resources such as printers, file systems, etc., also need to be accessed
- These are the main tasks of run-time support
- In this lecture, we focus on memory management

 YN. Sankant 3

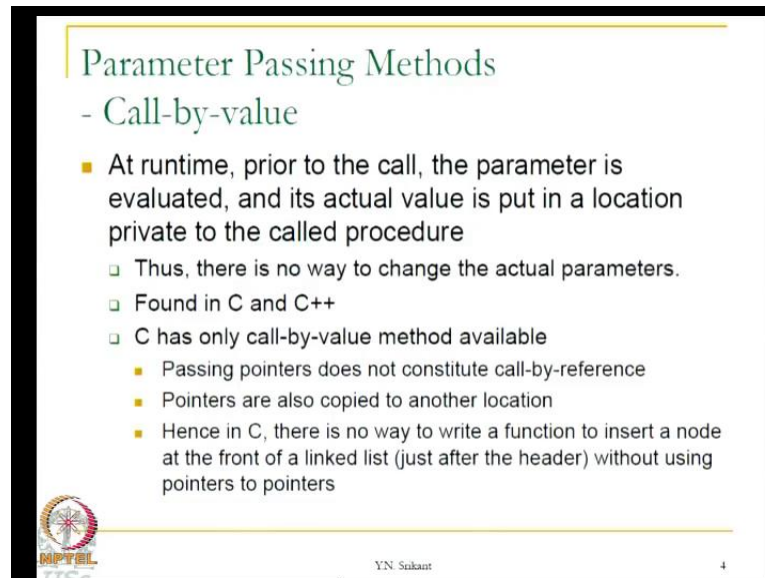
So, let us begin with the slide to understand what exactly is run time support. It is not enough if we simply generate machine from intermediate code that is fairly easy say generating simple code from intermediate code is very easy. But the program is supposed to run in a computer which actually has an operating system. So, the program that we generate, you know from the intermediate code cannot run in a computer system without getting the resources appropriately. So, in other words for example, there is a need to

manage memory when a program is running. So, when the program wants to run it must get an amount of memory necessary to place its instructions. And it also requires memory to place its data and then the control has to be transferred to the beginning of the program and it runs.

So, even within you know even the program starts running the programs actually may request for memory blocks. And they may release memory blocks this happens in dynamic memory allocation when we use pointers. And we want to build data structures such as linklists etcetera and the parameters must be passed to functions. So, the order in which the parameters are passed the way in which the result is passed back to the function. All these actually require protocols which are part of the run time system and run time support. So, and we also have other resources such as printers file systems etcetera which are required within our programs.


So, the operating system provides support for all this and therefore, we must understand how to use these systems print rather these operating system routines. So, all these are actually part of the run time support that is a method to manage memory method to manage other resources. And provide services to you know allocate and deallocate memory blocks how to pass functions parameters to functions, these are all the main tasks of the run time support. So, in this lecture, we will mainly focus on memory management. Because the rest of the services such as printers how to use printers file systems etcetera are all provided by the operating system itself.

(Refer Slide Time: 38:26)



**Parameter Passing Methods**  
- Call-by-value

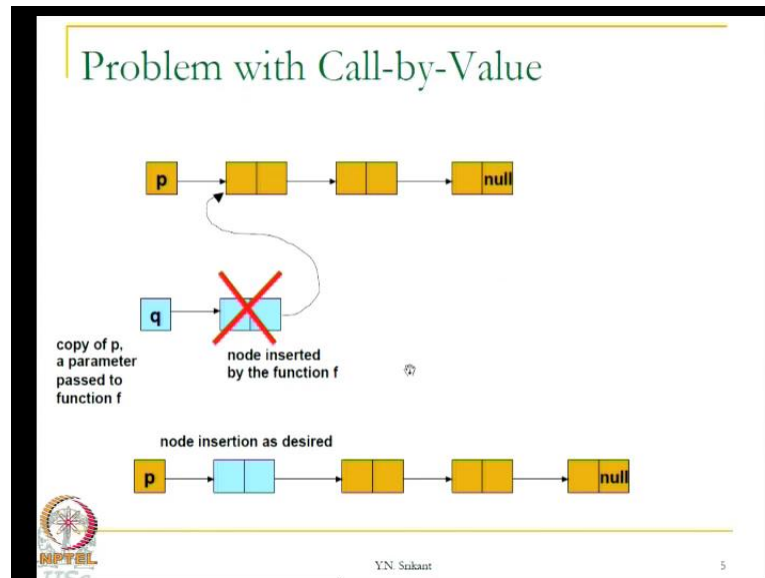
- At runtime, prior to the call, the parameter is evaluated, and its actual value is put in a location private to the called procedure
  - Thus, there is no way to change the actual parameters.
  - Found in C and C++
  - C has only call-by-value method available
    - Passing pointers does not constitute call-by-reference
    - Pointers are also copied to another location
    - Hence in C, there is no way to write a function to insert a node at the front of a linked list (just after the header) without using pointers to pointers

 YN Srikant 4

So, let us begin with parameter passing methods. So, the first parameter passing method and the most popular 1 is the call by value method of passing parameters. So, I will give you examples of how to use different parameter passing methods after we discuss all the 3 methods of passing parameters rather 4 methods of passing parameters. So, call by value everybody understands it this is available in C and C plus plus. The parameter which is passed as call by value parameter is evaluated prior to the call and its actual value is put in a location which is private to the called procedure.

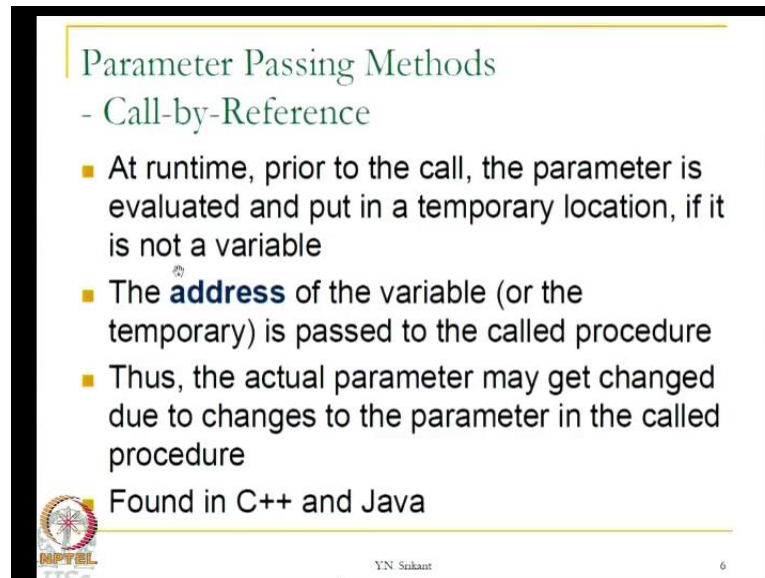
So, in other words there is no way to change the original parameter which is the actual parameter. The value is copied to a local location and then used in the function and C has only call by value method whereas, C plus plus has call by value call by reference as well. So, in the case of C passing parameters does not constitute call by reference. This is a, you know misunderstood concept in C it is not that call by reference you know is possible by passing parameters. So, sorry passing pointer, when we use pointers the pointer are also copied to another location. So, the effect of this is that there is no way to write a function to insert a node at the front of a linked list just after the header without using a double star that is pointers to pointers.

(Refer Slide Time: 40:24)



Let me explain what I mean with the help of this picture, here is a link lists right and we want to insert a new node here before p and the first node. So, in other words this is the picture we want the node must be inserted here just before just after P. And just before the first node if we write a simple insert node function in C pass the header of this link list p as a parameter to it immediately call by value kicks inn. It makes a copy of p within the function as q so; obviously, q would also be pointing to this particular node. The first node when we insert create a new node and insert it, it would be inserted after q and just before the first node of the original list and but this is not what we wanted. So, we wanted the modification to be like this all right. So, passing this pointer to the function which tries to insert a node here cannot be done, because C permits only call by value and copy of this pointer is made into a local location. Of course, this can be easily solved using pointers to pointers and that is something not necessary to be discussed at this point of time.


(Refer Slide Time: 42:01)



**Parameter Passing Methods**  
- Call-by-Reference

- At runtime, prior to the call, the parameter is evaluated and put in a temporary location, if it is not a variable
- The **address** of the variable (or the temporary) is passed to the called procedure
- Thus, the actual parameter may get changed due to changes to the parameter in the called procedure

Found in C++ and Java

 YN Srikant 6

So, what is call by reference? Call by reference is evaluate the parameter prior to the call and put it in a temporary location if the parameter itself is not a variable. So, for example, the parameter is an expression then it is evaluated and put in a temporary location if it is a variable then there is no need to put it into a temporary location. So, now, the adverse of the variable or the temporary is passed to the called procedure. So, why did I say put it in a temporary and pass the address of the temporary. That is because either is no way you can you know say a an expression corresponds to a single location.

So, we must evaluate the expression and then put it in put the value in a temporary location and pass the temporary location address of the temporary location. But if it is a single variable we do not do any of this we pass the address of the variable itself to the called function or procedure this way. So, the actual parameter may get changed due to the changes in the, you know changes in the function that are done to the parameter itself. So, the parameter can also change of course, if it was an expression and we had put it in the values was put in a temporary location it does not matter. Because that value of the temporary location will change, but the original expression cannot be; obviously, changed because expressions do not have any single location to deal with.

(Refer Slide Time: 43:53)

**Call-by-Value-Result**

- **Call-by-value-result** is a hybrid of Call-by-value and Call-by-reference
- Actual parameter is calculated by the calling procedure and is copied to a local location of the called procedure
- Actual parameter's value is not affected during execution of the called procedure
- At return, the value of the formal parameter is copied to the actual parameter, if the actual parameter is a variable
- Becomes different from call-by-reference method
  - when global variables are passed as parameters to the called procedure and
  - the same global variables are also updated in another procedure invoked by the called procedure

Found in Ada

YN Srivastava

So, this is possible in C plus plus and java of course, then we have the third method of passing parameters known as call by value result, this is really a hybrid of call by value and call by reference. What we really do, what is the call by value part? The actual parameter is evaluated before the procedure call is made and then its value is copied to a local location of the called procedure. So, this is the call by value part when we modify the, you know the parameter within the function or procedure. We are really making changes to this local location and therefore, the original or actual parameters value is not affected during the execution of the function or procedure.

Then how is it different from the call by value that is because at the end of the function or procedure. The final value of the formal parameter is actually copied to the actual parameter if the actual parameter is a variable other it does not make sense to make a copy anyway. So, that is the result part. So, to begin with we have call by value we put the value into a temporary and then use it within the function. But when the function is supposed to return, we copy the final value of that call by value result parameter into the actual parameter and then return. So, that is how the call by value result mechanism works.


But how can this be different from a call by reference mechanism. Suppose you have global variables in the program and they are passed as parameters to the called procedure. And of course, we change the global variables that is no problem, but the

same global variables are also updated in another procedure invoked by the called procedure. So, so if the a calls B, B gets this global variables as parameters, but then B calls c and c changes the global variables directly. So, then the affects are going to be very different we will see this later and this mechanism is found in the language Ada.

(Refer Slide Time: 46:37)

### Difference between Call-by-Value, Call-by-Reference, and Call-by-Value-Result

| <pre>int a; void Q()     { a = a+1; } void R(int x);     { x = x+10; Q(); } main()     { a = 1; R(a); print(a); }</pre> | <table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th style="padding: 2px;">call-by-value</th> <th style="padding: 2px;">call-by-reference</th> <th style="padding: 2px;">call-by-value-result</th> </tr> </thead> <tbody> <tr> <td style="text-align: center; padding: 2px;">2</td> <td style="text-align: center; padding: 2px;">12</td> <td style="text-align: center; padding: 2px;">11</td> </tr> </tbody> </table> <p style="text-align: center; margin-top: 5px;">Value of a printed</p> <div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 5px auto;"> <p><b>Note:</b> In Call-by-V-R, value of x is copied into a, when proc R returns. Hence a=11.</p> </div> | call-by-value        | call-by-reference | call-by-value-result | 2 | 12 | 11 |  |
|-------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------|-------------------|----------------------|---|----|----|--|
| call-by-value                                                                                                           | call-by-reference                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | call-by-value-result |                   |                      |   |    |    |  |
| 2                                                                                                                       | 12                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | 11                   |                   |                      |   |    |    |  |


Y.N. Srikant
8

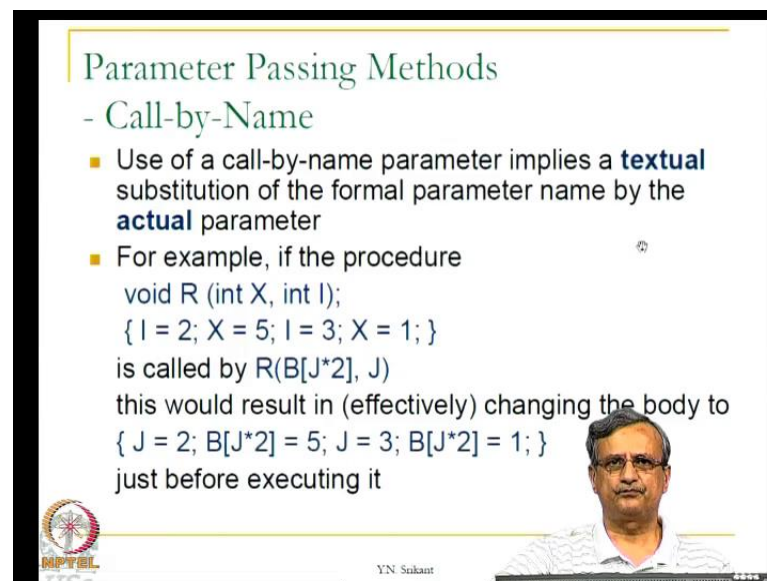
So, let us understand the difference between call by value call by reference and call by value result here is a simple program written in C lex syntax. But it is not really the C language because C language does not have call by reference and call by value result it is not even C plus plus, because C plus plus does not have call by value result. So, in the main program we have a global variable a, this is assigned 1 then we call the function R. So, R takes a parameter. So, this is the parameter a and that becomes x inside x is added 10 here you know and then we call the function Q which also increments a finally, we print a.

So, let us assume that call by value is the only mechanism available. So, we set a equal to 1 call R with as. So, this is a, but a copy of a is made in x because its call by value. So, the x part is the local copy which is which now becomes 11 then we call Q, Q modifies the global variable a. So, really speaking a was earlier 1 even though this became 11 this was the local copy. So, the original a is not modified a now becomes 2 the control comes back here and then it returns here and we print a. So, we get 2 for call by value call by

reference this a and this X are the same. So, the original a really, now becomes 11 and by a call to Q makes this 12.

So, the call by reference mechanism prints out value of a as 12 call by value result. So, a is actually copied into the local variable x. So, this is now the local variable X which becomes 11 then we call q. So, Q increments the global variable a to become 2 now we return to R and just before exiting the procedure R we copy the value of X back into the global variable a. So, X was 11 here now even though a was incremented here we have that is our copy will destroy the old value of a and put 11 into it and 11 is printed here. So, this is what I meant by the global variable a being modified independently by 2 routines 1 which takes it as a parameter and other which takes it as a global variable. So, the affect is neither call by reference nor call by value its very different.

(Refer Slide Time: 49:43)



**Parameter Passing Methods**

- Call-by-Name
  - Use of a call-by-name parameter implies a **textual** substitution of the formal parameter name by the **actual** parameter
  - For example, if the procedure  

```
void R (int X, int I);
{ I = 2; X = 5; I = 3; X = 1; }
```

is called by R(B[J\*2], J)  
this would result in (effectively) changing the body to  

```
{ J = 2; B[J*2] = 5; J = 3; B[J*2] = 1; }
```

just before executing it

NPTEL  
Y.N. Srivastava

Now, let us look at very complex parameter passing mechanism called call by name which is available in the programming language ALGOL. And also in functional programming languages it is not used in C C plus plus or Pascal simply, because its effects are very difficult to predict and understand how does it work. Use of call by name parameter implies a textual substitution of the formal parameter name by the actual parameter. So, let us understand what it means we have a procedure R which returns nothing there are 2 parameters X and I. So, within the body it is all very innocent I equal

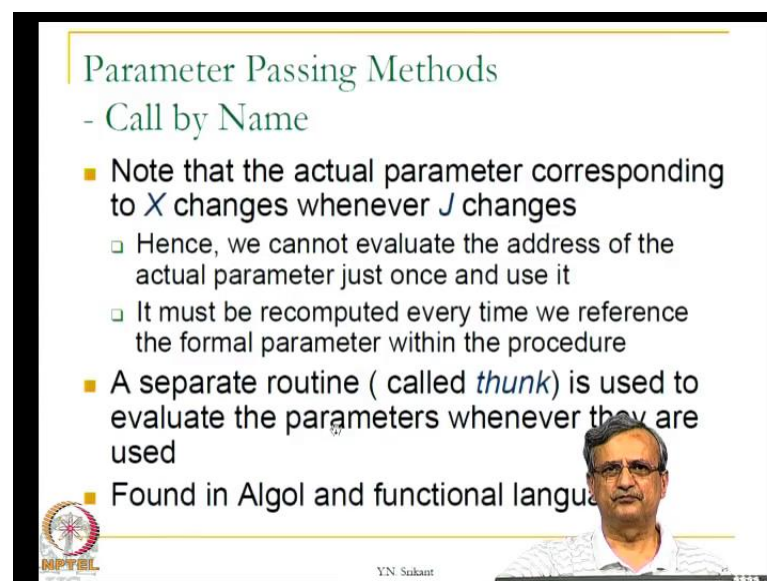


to 2 then we assign X equal to 5 then we have reassignment I equal to 3 and another reassignment X equal to 1.

So, let us say the procedure is called as B of J star 2 comma J. And let us assume call by name for all the parameters. As we said here the effect is as if the each one of these parameters is you know there is a textual substitution for this parameter in terms of the actual parameter which is passed. So, in this case the parameter I corresponds to the actual parameter J here. So, in effect I equal to 2 is really J equal to 2 next one is a X equal to 5 X corresponds to B of J star 2. So, the effect is as if we execute B of J star 2 equal to 5. So, now, J is 2. So, this is B of 4 being assigned the value 5 the next I equal to 3 it again means textual substitution of I J you know I by J.

So, we have J equal to 3 and X equal to 1 becomes B of J star 2 equal to 1. So, now, this is B of 6 equal to 1 really, because J has been changed. So, if you had just call by value of course, the effect is very simple to predict and if you have call by reference we would have evaluated the address of B of J star 2 and pass that. So, the element you know would have changed, but we would not really have here we have changed J B of 4. And here we have changed B of 6 this would not have happened in call by reference it would have happened, it will happen only in call by name.

(Refer Slide Time: 52:52)



**Parameter Passing Methods**  
- Call by Name

- Note that the actual parameter corresponding to *X* changes whenever *J* changes
  - Hence, we cannot evaluate the address of the actual parameter just once and use it
  - It must be recomputed every time we reference the formal parameter within the procedure
- A separate routine ( called *thunk*) is used to evaluate the parameters whenever they are used
- Found in Algol and functional languages

MPTEL  
Y.N. Srikant

So, the implication of this is that the actual parameter corresponding to X changes whenever J changes. Therefore, we cannot evaluate the address of the actual parameter

just once like in the case of call by reference. And then use it. It must be recomputed every time we reference the formal parameter within the procedure. So, in other words if you must evaluate the parameter as if we have done a textual substitution. So, that is what it really means. So, what we really do is we create a separate routine called thunk to evaluate the parameters whenever they are used. So, every time we refer to J we see what that parameter is that corresponds sorry the parameter I whenever we refer to I we see what that parameter is that would be J. So, this second parameter is now evaluated and then used here.

Similarly, whenever we have X equal to 5 we evaluate the parameter X with the current values. So, B of J star 2 would be evaluated that becomes a B 4 here whereas, in this case J has changed. So, this would become B of 6. So, every occurrence of the formal parameter invokes a small routine to evaluate the actual parameter and then use it here. So, such routines are called thunks they are actually nameless subroutines or nameless functions. So, these will be called whenever a parameter needs to be evaluated. So, this. So, we need to pass a thunk as a parameter in the whenever there is a call by name mechanism in place. So, for this parameter we create a thunk and pass that thunk as a parameter here for this parameter we create another thunk and pass that thunk as a parameter here. So, passing functions or procedures as parameters would be necessary to implement call by name mechanism.

(Refer Slide Time: 55:16)

### Example of Using the Four Parameter Passing Methods

1. void swap (int x, int y)
2. { int temp;
3. temp = x;
4. x = y;
5. y = temp;
6. } /\*swap\*/
7. ...
8. { i = 1;
9. a[i] = 10; /\* int a[5]; \*/
10. print(i,a[i]);
11. swap(i,a[i]);
12. print(i,a[1]); }

■ Results from the 4 parameter passing methods (print statements)

| call-by-value | call-by-reference | call-by-val-result | call-by-name |
|---------------|-------------------|--------------------|--------------|
| 1 10          | 1 10              | 1 10               | 1 10         |
| 1 10          | 10 1              | 10 1               | error!       |

Reason for the error in the Call-by-name Example  
The problem is in the swap routine

```
temp = i; /* => temp = 1 */
i = a[i]; /* => i = 10 since a[i] == 10 */
a[i] = temp; /* => a[10] = 1 => index out of bounds */
```

Y.N. Srikant

11

Let us understand this example to see how the 4 parameter passing methods are different. This is a very simple swap routine and then we want to study the effect of swapping on `a[i]` where `i` equal to 1 and `a[i]` equal to 10. So, with call by value swapping really does not happen. So, it prints out 1 and 10 with call by reference the swapping happens. So, 1 and 10 becomes then and 1 call by value result also the swapping will happen, because the values are copied back to the original locations.

So, 1 and 10 and 10 and 1, but in the case of call by name there is an error the first of course, we print 1 and 10. But then when we try to execute the body of swap `temp` equal to `a[i]` becomes `temp` equal to 1 `i` equal to `a[i]` makes it `i` equal to 10. Because `a` equal to 10 and now we try `a` equal to `temp`; that means, we are trying `a[i]` which is `a` of 10 `a` of 10 does not exist, because the array is only 5 locations long. So, this gives an error out of bounds and stops the program. So, this is the difference between the 4 parameter passing methods, we will stop the lecture here and continue in the next lecture.

Thank you.