

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

**Lecture - 52**  
**Intermediate Code Generation (Contd.)**

Till last class, we have seen a number of three address code instructions, which are there in three address coding strategy. And the last one that you were looking into the procedure call and return. So, all most all the programming languages they have got some sort of procedure facility. So, if we get rid of that in a intermediate language or this then, there will be problem in getting the translation process.

So, what these intermediate code designing technique assumes is that, there is a very primitive type of parameter passing strategy and very primitive type of function call, a procedure call that is supported. And similarly there is a strategy for returning from the procedure.

(Refer Slide Time: 01:01)

**Procedure Call/Return**

- A call to the procedure  $P(x_1, x_2, \dots, x_n)$  is converted as  
param x1  
param x2  
...  
param xn
- A procedure is implemented using the following statements  
→ enter f, Setup and initialization  
leave f, Cleanup actions (if any)  
return  
return x  
retrieve x, Save returned value in x

*Handwritten notes:*  
param x1, x2, ...  
 $x = f_1(a_1, a_2)$   
param a1  
param a2  
enter f1  
retrieve t1  
 $x = t1$   
f1  
local param  
return  
leave f1

The slide features a video inset of Prof. Santanu Chattopadhyay in the bottom right corner. The bottom of the slide has logos for IIT Kharagpur and Swayam.

So, to call a procedure with a P with the parameters  $x_1, x_2$ , upto  $x_n$  so, it will be converted into a block of parameter statement param x1, param x2, upto param x n. So, you see that we have not included the statements like say param and a number of parameter  $x_1, x_2$  etcetera. So, in that case the difficulty is that there is no limit on the

number of parameters that you are allowed to have in this statement. So, naturally you cannot code it in 3 address coding strategy so, to make it a very simple.

So, it has been converted into if there are  $n$  parameters, then it will be converted into  $n$  parameters statements, parameter  $x_1$  to parameter  $x_n$ . And then for processing for a procedure call so, it is done as enter  $f$ . So, enter  $f$  at this point several actions may be necessary depending upon the language for which we are generating the code and the target machine.

So, it for example, it may require that these local variables  $a$   $b$  are assigns space ok. So, that is all those translations will be done at the enter  $f$  point. So, at a high level we just note it that it is calling the procedure  $f$ , so it is enter  $f$ . And similarly at the time of returning certain actions are necessary to clean up the thing like clean up the space. So, that is leave  $f$ . So, it will be doing some cleanup actions like, say these the space that was allocated to local variables are they should be given back. Similarly, there may be some dynamic variables created in the body of the procedure.

So, after the procedure is over those dynamic variable they will not have any meaning. So, they have to be disposed of so etcetera, that way there can be a leave statement at the end. So, these encapsulates all possible actions that may be needed for the cleanup procedure.

Now, apart from that we have got the return statements to return from a procedure. So, in this in the first version of return so, there is no; there is no return value. So, it is basically the wide type of functions that you have in  $c$ . So, if it is if you want to come back from that procedure or functions, it may be a returned. So, if there is a returned value then you can use this return  $x$ .

So, that will be at the procedure you can have at the function end. So, you can converted into return  $x$  and retrieve  $x$ . So, this save return value in  $x$ . So, this will be return value will be saved in the variable  $x$ . So, that way it will be converted like. If, I have got some say functions say  $f_1$  being called with some parameter  $a_1$   $a_2$ . So, at the time of call I should have these three addressed statements param  $a_1$ , then param  $a_2$ , like that and then I should have enter  $f_1$  ok.

Then, at the body of when I am translating the f 1 part, I should have these local set up part. So, this is a local part so, this so, this enter f 1. So, if the local part may be there they may be created so, at the end. So, I can have these return or return x etcetera and then this leave f will be coming at the bottom.

So, leave f 1 will come at the bottom. So, that will be leaving thing and this return statements should come before that and this in the main program. So, I can have this retrieve thing, like it may be that I have got a call like x equal to f 1 something. So, retrieve the return value in to some temporary variable t 1. And then I can have x equal to t 1. So, this may be the final code that is generated the 3 address code that is generated. So, there may be some variant of this, but this is one possible strategy by which you can do this translation.

(Refer Slide Time: 05:07)

The slide is titled "Miscellaneous Statements" and contains two bullet points:

- More statements may be needed depending upon the source language
- One such statement is to define jump target as,

Below the text, there are handwritten annotations. On the left, it says "label L". To the right, there is a diagram showing a circle containing "L" with an arrow pointing to a box containing "81". Above the circle, there is a handwritten "goto L". To the left of the circle, there is a handwritten "x = f1()".

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

So, there may be some other statements like, they are come under the broad heading of miscellaneous statements, more statements may be needed depending upon the source language. So, the some source language it has got some special type of instructions, which are not generally present in other programming languages. Like say in C programming language, we have got this next statement break statement like that.

So, which may not be present in some other language, so, they come under this broad heading. So, how do you; how do you realize a continue statement that you have in say C language, how to converted into intermediary code? So, that may be a issue, that may be

an issue and that can be done by means of some go to statements, as you will see later and one such statement is to define jump target that is label L.

So, it is like this for jumping to a procedure we said that it is like goto L and then what is this L? So, somehow if this L is a procedure ok. So, maybe I have in the main in the source language program I have got a call for a procedure. So, I have got a x equal to f 1 something ok. Then, they are their the goto L is there. So, it is expected that L, I should start the procedure f 1. Here, I should have the procedure f 1.

So, to do this things, so, this label has to be generated, but 3 address code it is level is not put like in this fashion, what we do is that we put another 3 address statement called label L. And then in this label L is encountered means at this point defines the beginning of the code, wherever we say that its a jump to the label L. So, the execution should come to the this point. So, that is how this labels will be implemented or labels will be meaningfully three address code.

(Refer Slide Time: 07:13)

The slide is titled "Three-Address Instruction Implementation". It contains a bulleted list describing the quadruple representation of instructions:

- Quadruple representation – each instruction has at most four fields:
  - Operation – identifying the operation to be carried out
  - Upto two operands – a bit is used to indicate whether it is a constant or a pointer
  - Destination

Below the text, two instruction quadruples are shown as examples:

**Example 1:  $x = y + z$**

Op	PLUS
Src1	y
Src2	z
Dest	x

**Example 2:  $\text{if } t1 \geq t2 \text{ goto } L$**

Op	JMP_GE
Src1	t1
Src2	t2
Dest	instruction labelled L

The slide also features a Swayam logo and a small video feed of a man in a red vest in the bottom right corner.

Now, how do you represent this three addresses instruction? Three addresses instruction means that there are atmost three addresses. So, every instruction is a quadruple. So, this is the first representation that we have so, its a quadruple representation. So, if so, the for a each instruction has atmost four fields; one is the operation that identifies the operation to be carried out. Like, here is an example, like this is the so, here the operation is plus

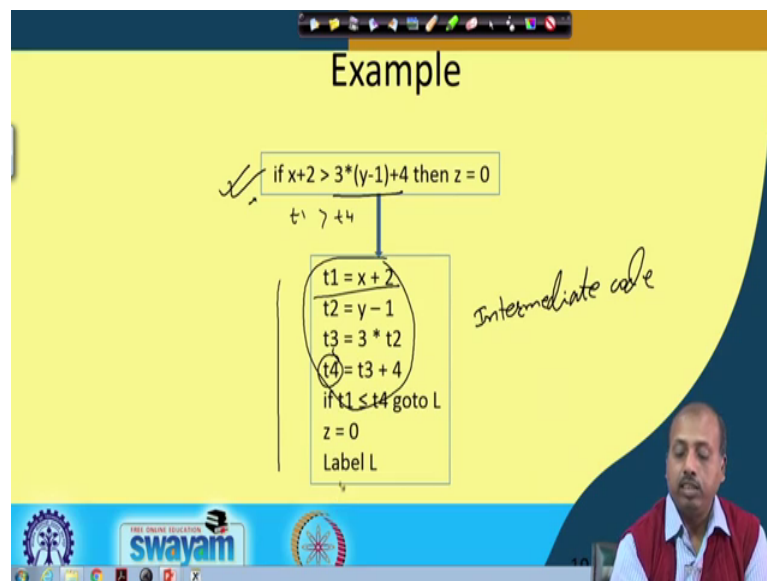
here the operation is jump on greater or equal so like that. Operation it will identify the operation to be carried out upto two operands.

So, so, this is like this source 1, source 2, these are 2 source operand and these destination is the destination. Now, it may so, happen that we directly mention the I would directly mention the operand here or we say that its a pointer.

So, there may be a bit here. So, that will identify whether this is a pointer or a normal variable, like say it is a constant or a pointer, like say  $x$  equals  $y$  plus  $z$ . So, this is not a, so, this is pointing to  $y$ . So,  $y$  is the symbol table for symbol table entry for that. So, whatever be the memory allocation allocated to  $y$  offset of that, so, that will come here. So, this bit will be 1 telling that its a pointer.

So, and the destination will also be there. So, this is a generic type of representation for three address code instructions. So, we have got an operation field to upcode field for two operand fields and one destination field.

(Refer Slide Time: 08:55)



Now, so, this is a typical example like say if  $x$  plus 2 greater than 3 into  $y$  minus 1 plus 4 then  $z$  equal to 0. So, we are not talking about how are we going to do this translation? So, that will unfold as we proceed through this content of this particular chapter, but after everything has been done the intermediary code, it will look something like this. So, this

is your intermediate code, this is the intermediate code generated corresponding to the statement that we have at the beginning.

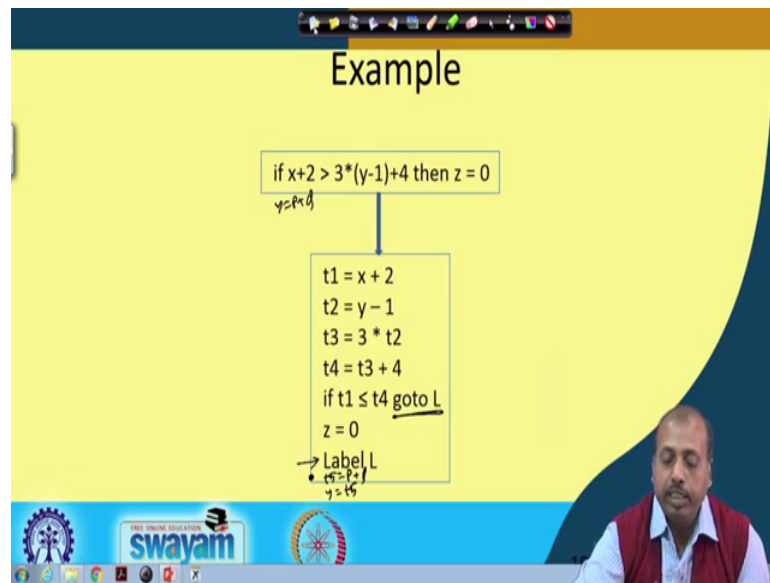
So, here is a single line source single source language statement corresponding to that. So, many three address code instructions have been generated. So, let us try to check it. So, first this  $x + t_1$  equals to  $x + 2$ . So,  $t_1$ ,  $t_2$ ,  $t_3$ ,  $t_4$  so, they are all temporary variables. And you will see that in three address code there will be large number of temporaries created. Because, in three address code the individual operations they have got only two operands in them.

So, will be doing it like that we will have we will generate two address sub expressions or sub parts sub instructions to store the result of those two address operations. Like, in  $t_1$  we calculate  $x + 2$  in  $t_2$  we calculate  $y - 1$   $t_3$  we calculate  $3 + t_2$ .

So, ultimately  $t_1$  contains  $x + 2$  and  $t_3$  contains  $3 + y - 1$ , then  $t_4$  is  $t_3 + 4$ . So, this way we have generated the first. So, this part is evaluating, this part is evaluating, the expression that  $x + 2$  and so, this  $t_1$  is evaluating  $x + 2$  and this  $t_4$  has got the value of this right hand side expression. Now, so, if  $t_1$  is less or equal 4, so, the our condition is  $t_1$  is greater than  $t_4$ . So, in that case  $z$  will be made equal to 0. So, the condition is generated in a in just the reverse of the fraction. So, if  $t_1$  is less or equal  $t_4$  goto L. So, goto L so, this is a label and then made  $z$  made  $z$  equal to 0 and then the label L comes.

Since, we have got only a single statement in this code. So, after label L there is nothing.

(Refer Slide Time: 11:23)



So, if there was a statement here, if there was a statement like say y equal to P plus Q after doing this, then we will have this code for y equal to P plus Q. So, another t 5 equal to P plus Q, then y equal to t 5. So, that would have been done, but here it is so, so when it is says when it says goto L, label L. So, it will start it will come to this label L.

So; that means, my execution should start from this point, if t 1 is less or equal t 4. So, this way three address codes are will be generated. So, will see how using different translation rules or syntax directed translated mechanism, how can we generate such three address code.

(Refer Slide Time: 12:03)

### Three Address Code Generation - Assignment

**Grammar:**

$$S \rightarrow id := E$$
$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid id$$

**Attributes for non-terminal E:**

- E.place – name that will hold value of E
- E.code – sequence of three address statements corresponding to evaluation of E

**Attributes for non-terminal S:**


- S.code – sequence of three-address statements

**Attributes for terminal symbol id:**

- id.place – contains the name of the variable to be assigned

Grammar Rule	Semantic Actions
$S \rightarrow id := E$	$S.code := E.code \parallel gen(id.place := E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp();$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place := E_1.place '+' E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp();$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place := E_1.place '*' E_2.place)$
$E \rightarrow -E_1$	$E.place := newtemp();$ $E.code := E_1.code \parallel gen(E.place := '-' \text{unminus} E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow id$	$E.place := id.place;$ $E.code := ''$

• Function newtemp returns a unique new temporary variable.  
• Function gen accepts a string and produces it as a three-address quadruple.  
• '||' concatenates two three-address code segments



So, first we considered the very simple type of statement which is the assignment statement. So, we have to start with the grammar for the assignment statement. So, the grammar for the assignment statement is this one, S producing id assigned as E, then E producing E plus E or E star E or minus E that is unary minus within bracket E and id.

Now so, depending upon the assignment statement, that is given to us. So, it will be it will generated parse tree, whether the route of the parse tree will have this S and it will have like this id assigned as E and then E will further break down into a parse tree for the expression. Now, as for as reductions are when this shift reduce parses will work. So, it will be trying to reduce the handles like this. And when it is doing this handle deduction or handle pruning, then it will be consulting the corresponding action part.

So, this is a so, in this table we have written the semantic actions that will take place when the corresponding reductions are going to happen. So, suppose we have got this whole thing, this entire expression has been calculated. So, at the last this particular reduction is being made ok. So, at that point this codes should be generated, s, that this id will be assigned the value of E.

And how is it done? So, assume that there are two attributes for the nonterminal E; one is called the E dot place that holds the name that will hold the value of E and E dot code is the sequence of three address statements for corresponding to evaluation of E.



(Refer Slide Time: 14:11)

### Three Address Code Generation - Assignment

**Grammar:**

$$S \rightarrow id := E$$

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid id$$

**Attributes for non-terminal E:**

- E.place – name that will hold value of E
- E.code – sequence of three address statements corresponding to evaluation of E

**Attributes for non-terminal S:**

- S.code – sequence of three-address statements

**Attributes for terminal symbol id:**

- id.place – contains the name of the variable to be assigned

Grammar Rule	Semantic Actions
$S \rightarrow id := E$	$S.code := E.code \parallel gen(id.place := E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp();$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place := E_1.place + E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp();$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place := E_1.place * E_2.place)$
$E \rightarrow -E_1$	$E.place := newtemp();$ $E.code := E_1.code \parallel gen(E.place := 'unminus' E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code;$
$E \rightarrow id$	$E.place := id.place;$ $E.code := ''$

• Function *newtemp* returns a unique new temporary variable.  
 • Function *gen* accepts a string and produces it as a three-address quadruple.  
 • '||' concatenates two three-address code segments

So, if I tell you that E dot code has got so, this E, it has got 2 portion; one is the place and another is the point at code. So, this is the code that is required for evaluating E. And E dot place is the name of the place or name of the temporary that finally, holds the value of E. So, if the last assignment here last computation here is say t 5 equal to something and that holds the value of this E dot place then these value of the expression E, then this E dot place will be equal to t 5 ok. Now, given this thing so, if I just look up by one more step, this i d assigned as E. Then, what is the thing to be done for generating the code for S.

So, in the code for S, I should have the first the code for E code for evaluating E. So, this should come as it is. And after that I have to do the assignment that is this i d dot place i d dot place holds the place name of the i d, like actual statement may be a equal to b plus c. So, this i d, so, i d dot place it will hold the value a. So, then this a equal to so, this I will have this code followed by this extra code, that i d dot place assigned as E dot place. I will have this extra code i d dot place assigned as t 5.

So, this is the additional thing that I am going to have. So, this way we will try to write down the semantic actions that should take place, when the corresponding reduction is being made. So, this reduction is done at the end. So, the this is the parse tree for E, you see that in a shift reduced parsing policy. So, this reduction will be done at the end of this all other reductions at that time.

So, by the time you are going to read do the thing the reduction at this point. So, this place and code these two are already computed for E and then we generate the extra code that i d dot place assigned as E dot place. So, this way I have got this two attribute E dot place and E dot code, that holds the name of the symbol that will hold the value of E. And the sequence of three address code where three address statements that corresponding to the evaluation of E. Then, for the non-terminal S, we will assume that there is an attribute S dot code just like E dot code holds the codes for E, then this code holds the code for S. So, this is our S dot code; this is our S dot code.

So, sequence of three address statements. So, that will be required for this statement to be executed. Then, this attribute for the terminal symbol i d so, this is i d dot place contains the name of the variable to be assigned. As, I was telling that if it is a equals b plus c, then i d dot place is equal to a. And this is very simple to find out, because the lexicon analyzer will get the value will get s as symbol as a, it will return i d as a token, but you can always return as an attribute of this i d, the actual name of the symbol E.

So, parser will know the name of the symbol by means of that y y L y type of attribute that we have, then this from that you can while doing this reduction, we can take help of this and generate the complete tree for a complete code for S. So, this i d dot place is there. Now, let us look into a very simple a rule that is a E producing i d.

So, what is to be done? So, E dot place should be equal to i d dot place because E dot place it holds the name that will hold the value of E and that is and i d dot place holds the name of i d. So, naturally dot place has to be assigned to i d dot place. And nothing has to be done E dot code is null nothing has to be done at this point ok.

Similarly, if you look into say this rule say E producing within bracket E 1, then E dot place equal to E 1 dot place. So, because the here whatever is the so, variable that hold the, whatever may be the symbol that holds the value of E, that is E 1 dot place. So, that is that itself will hold the value of E ok. So, E dot place equal to E 1 dot place. And then these E dot code is also equal to E 1 dot code; so, that much is fine.

Now, let us look into this rule E producing unary minus E 1. At this point so, I have to have something like this. So, I will so, for generating the code of E first I have to include the code for E 1. So, I have to include the code for E 1 followed by I have to generate a

piece of code, where this I need a new E dot place. Because, E dot place whatever temporary has been used in the computation of E 1.

So, that cannot be used for the I have to hold the value of E the E dot place. So, in this statement so, you have made E dot place equal to new temp, where new temp is a function that returns the unique new temporary variable. So, this is very simple, like if you have a counter like how many pair of temporaries have generated so, far in this in a in a compilation process. So, that counter may be incremented by one and the after that the value may be attest to t to give a new next temporary variable.

So, this way this E dot place equal to newtemp. So, this newtemp function will return a new temporary variable. And that will be assigned to E dot place. So, as if that temp new temporary variable. So, it will hold the value of this expression E. And what is the code for E? Code for E is the code for E 1 so, this particular symbol. So, this concatenates two, three-address code segments.

So, this is a concatenation operator. So, if we think about the three-address codes statements as strings, then it is something like a string concatenation ok. Definitely so, because ultimately what you have is nothing, but a text file only three address code, that is generated is a text file. And so, you can always take the individual lines of the text file as string.

So, E dot place assigned as unary minus E 1 dot place. So, this particular code will be generated. So, this unary minus E 1 dot place. So, this unary minus thing will introduce a E 1 dot place was holding the value of E 1 is the name of variable that holds the value of E 1. So, that will be negated by this unary minus symbol and that will be assigned to E dot place.

Similarly, if you look into say this particular statement E producing E 1 or star E 2, then also in the code for E I should have the code to evaluate E 1, I should have the code to evaluate E 2 and then I should generate an additional piece of code. So, that will multiply E 1 dot place and E 2 dot place.

So, it is done in this fashion. So, E 1 dot code concatenated with E 2 dot code concatenated with this gen function. So, gen function it accepts a string and produces it as a three address quadruple. So, this is the whole string that is accepted that is taken as

input by the gen function. And then it is generating this piece of code that E dot place. So, E dot place is already obtained the it is the corresponding temporary that we have got here.

So, that temporary assigned as E 1 dot place so, a temporary variable which holds the E 1s final value and then multiplied by E 2 dot place. So, E 2 dot place holds the final value of E 2. So, that way they will be so, this particular piece of code will be concatenated with the code for E 1 code for E 2 and then this one. And then the similarly this E producing E 1 plus E 2. So, this is also similar as for as code generation is concerned.

So, E dot place equal to new temp and then E dot code equal to E 1 dot code concatenated with E 2 dot code concatenated with generate E dot place assigned as E 1 dot place plus E 2 dot place. And then off course I will need this particular statement E dot place equal to newtemp to hold the to get new temporary variable. So, this way for arithmetic assignments statement so, we can draw the corresponding we can draw the corresponding parse tree and then follow this particular semantic actions ok.

You can follow this these are always for the syntax directed mechanisms. So, by which we can generate the corresponding code.

(Refer Slide Time: 23:17)

**Example**

$x := (y+z)*(-w+v)$

Reduction No.	Action
1	$E.place = y$
2	$E.place = z$
3	$E.place = t_1$ $E.code = \{t_1 := y + z\}$
4	$E.place = t_1$ $E.code = \{t_1 := y + z\}$
5	$E.place = w$
6	$E.place = t_2$ $E.code = \{t_2 := \text{uminus } w\}$
7	$E.place = v$
8	$E.place = t_3$ $E.code = \{t_3 := \text{uminus } w, t_3 := t_2 + v\}$
9	$E.place = t_3$ $E.code = \{t_3 := \text{uminus } w, t_3 := t_2 + v\}$
10	$E.place = t_4$ $E.code = \{t_1 := y + z, t_2 := \text{uminus } w, t_3 := t_2 + v, t_4 := t_1 * t_3\}$
11	$S.code = \{t_1 := y + z, t_2 := \text{uminus } w, t_3 := t_2 + v, t_4 := t_1 * t_3, x := t_4\}$

So, here is an example like a using that previous grammar. So, we have got an example string that is  $x$  produces. So, this  $x$  assigned as  $y$  plus  $z$  into minus of  $w$  plus  $v$ . So, this is if you look into the grammar. So, the route is  $s$  producing  $i d$  assigned as  $E$ . So, this is the parse tree. So,  $s i d$  assigned as  $E$ , so,  $i d$  this  $i d$  is  $S$ . So, that will be were known by the lexic analyzer. So, lexic analyzer will identify that  $i d$  dot place has  $x$ . And, then this  $E$ , so, this will be so, this star will be generated so,  $E$  star  $E$ .

So, I am not going into ambiguity and all those things and I assume that my parser is clever enough. So, it has been given enough hint about this precedence of operators and everything. So, that it gives the multiplication higher priority than addition. And unary minus higher prediction higher priority, than any other operator that we have in the statement.

So, this produces like this that  $E$  producing,  $E$  star,  $E$  then this  $E$  produces within bracket  $E$ , and then this  $E$  produces  $E$  plus  $E$  then this  $E$  produces  $i d$ . This  $E$  produces  $i d$ , then this is broken down into within bracket  $E$ . And then this  $E$  is broken down into  $E$  plus  $E$  and then the first  $E$  gives give raise to this unary minus  $E$ . And then this  $E$  gives me  $i d$ , which is  $w$  and then this  $E$  gives me  $i d$  which is  $v$ .

Now, so, when this parse tree is being generated. So, you know that this  $L r$  parsing strategy. So, it will be doing reduction in the reverse order. So, you have to identify the you have to identify which reduction will be done at the beginning. So, the first reduction that will be done is this one ok. The first reduction that is done is this one so,  $E$  producing  $i d$ .

So, this will be reduced so, we number this reduction as 1 for the sake of our understanding. Then after that the next reduction that will be done is this one.  $E$  producing  $i d$ , then once these 2 reductions had been done, then this the number three reduction will be made  $E$  plus  $E$  and then after this reduction has been made. So, this will be done. So, we number this as 4 ok. And after that so, this will be 5. So, this  $E$  producing  $i d$  so, this will be reduced next so, this produces 5. So, you can what you can do is you can start looking from the bottom left right and then you try to see like how far can you proceed ok.

So, when I start with a bottom or left sides. So, I can see that I can do this thing, but I cannot do this. So, I after doing this I have to stop and go towards right a bit and then see

what is the next available reduction, so, that is the this one. And then once this two are done so, this reduction is now ready. After, once this is done this reduction is now ready. So, but after that I find that I cannot proceed further, then I have to come back and see what is the next operand at the next reduction at the lowest level that is possible. So, I find that this is the next reduction.

So, the so, now this is 5 then the after this is done. So, this is our number 6. So, this is 7 and once these are done so, this is 8, this is 9, this is 10 and this is 11. So, that is how so this so, we call it annotation of parts.

So, there is something more, but we will come to that later. Now, what semantic action takes place, when you look at different reduction steps. So, at reduction step number 1 that is this reduction. So, what we have to do is that we have to look for the corresponding rule. So, E producing i d it says E dot place equal to i d dot place and E dot code equal to null. So, this E dot place equal to i d dot place, so, i d dot place is y.

So, E dot place equal to i d dot place. So, this is done and the code is null. So, it is not written here, then the second reduction E producing i d. So, this E dot place equal to z. So, that is done and then there was i d dot place, then at reduction number 3, at reduction number 3 we have got E producing E plus E. So, let us go back and see what was the corresponding action.

So, it will first generate a new temporary variable, it will first generate a new temporary variable in E dot place and then it will generate the code E 1 dot code E 2 dot code and this particular line being generated separately. So, how is it happening you seen that this E dot place equal to t 1. So, here I have got a new temporary variable t 1. And then E dot code E 1 dot code and E 2 dot code so, they are null E 1 dot code E 2 dot code are null. So, nothing done and then this new temporary it says the E dot place that is t 1 equal to E 1 dot place plus E 2 dot place so, y plus z. So, this is the code that is generated.

(Refer Slide Time: 28:41)

**Example**

$x := (y+z)*(-w+v)$

Reduction No.	Action
1	$E.place = y$ ✓
2	$E.place = z$ ✓
3	$E.place = t_1$ ✓
4	$E.code = \{t_2 := y + z\}$
5	$E.place = w$ ✓
6	$E.place = t_2$ ✓
7	$E.code = \{t_2 := \text{uminus } w\}$
8	$E.place = t_3$
9	$E.code = \{t_2 := \text{uminus } w, t_3 := t_2 + v\}$
10	$E.place = t_4$
11	$E.code = \{t_1 := y + z, t_2 := \text{uminus } w, t_3 := t_2 + v, t_4 := t_1 * t_3\}$ $S.code = \{t_1 := y + z, t_2 := \text{uminus } w, t_3 := t_2 + v, t_4 := t_1 * t_3, x := t_4\}$

$t_1 = y + z$

So, the code that is generated at  $t_1$  equal to  $y$  plus  $z$  this line has been generated. After, that it will come to reduction number 4, so, within bracket  $E$ . And then within bracket  $E$  the action the semantic action is this  $E \cdot \text{place}$  equal to  $E_1 \cdot \text{place}$   $E \cdot \text{code}$  equal to  $E_1 \cdot \text{code}$ . So, that is what is done. So,  $E \cdot \text{place}$  equal to  $t_1$ ; so,  $E \cdot \text{place}$  equal to  $t_1$  and  $E \cdot \text{code}$  equal to this line only. Then at reduction number 5, so, I have got this  $E \cdot \text{place}$  equal to  $w$  that is no  $E \cdot \text{code}$ , then at reduction number 6.

So, it is unary minus so, this so, if you look into the action for this unary minus you see that it is so, it is written like E producing minus of E 1. So, that is a newtemp will be generated and E 1 dot code followed by this temp generation of this extra thing ok. So, that will be done.

So, that is done here. So, this  $t_2$  the new code that is generated, so, a new place has been generated at  $t_2$  and then  $t_2$  assigned as unary minus  $w$ . So, unary minus  $w$  so, that will make it this they make this minus  $w$  part. So, this way the code generation continues will go through this in the next class.