

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

**Lecture - 42**  
**Type Checking (Contd.)**

(Refer Slide Time: 00:17)

**Name Equivalence**

- Two types are name equivalent if they have same name or label

```
typedef int Value
typedef int Total
...
Value var1, var2
Total var3, var4
```

- Variables var1, var2 are name equivalent, so are var3 and var4
- Variables var1 and var4 are not name equivalent, as their type names are different

15

So, name equivalency as I was telling that in this case we have got two different types whose equivalency we want to establish and then for example, this type def int value and type def int total. So, here we are defining two different types value and total, though they are both of them are of type integer. But they are new types now. So, they are given two new names; value is a name and total is a name.

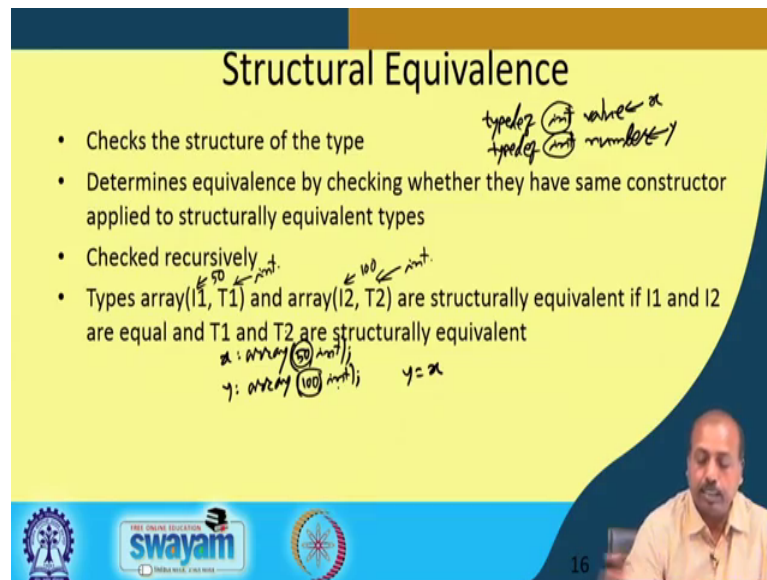
Now, we have got 2 variables var 1 and var 2 of type value and var 3 and var 4 they are of type total. Now, variables var 1 and var 2, so their name equivalent and similarly var 3 and var 4 they are also name equivalent. But this variable var 1 and var 4, they are not name equivalent. Because they are their names of different types because they are var 1 is of type value and var 4 is of type total. Though ultimately both of them are boiling down to integers, but name equivalency will not check that ok. So, it will be just looking into the immediate type and based on that it will be giving us the type of the variable. So, that is the name equivalency.

(Refer Slide Time: 01:32)

### Structural Equivalence

- Checks the structure of the type
- Determines equivalence by checking whether they have same constructor applied to structurally equivalent types
- Checked recursively
- Types  $\text{array}(I1, T1)$  and  $\text{array}(I2, T2)$  are structurally equivalent if  $I1$  and  $I2$  are equal and  $T1$  and  $T2$  are structurally equivalent

*Handwritten notes on slide:*  
type def (int) value ← x  
type def (int) number ← y  
x: array(50, int);  
y: array(100, int);  
y = x



Now, another case is the structural equivalency; it will check the structure of the type. So, it will determine equivalency by checking whether they have some same constructed applied to structurally equivalent types. So, if we have got say types integer; if we have got say types say integer; if we have got say type say one type is say previous example as I was taking that type def integer value and type def integer number. Now if I have got 2 variables x and y, where x is of type value and y is of type number; then, for structural equivalency it will they will go down. So, it will come to this integer levels and then they will be declared as equivalent.

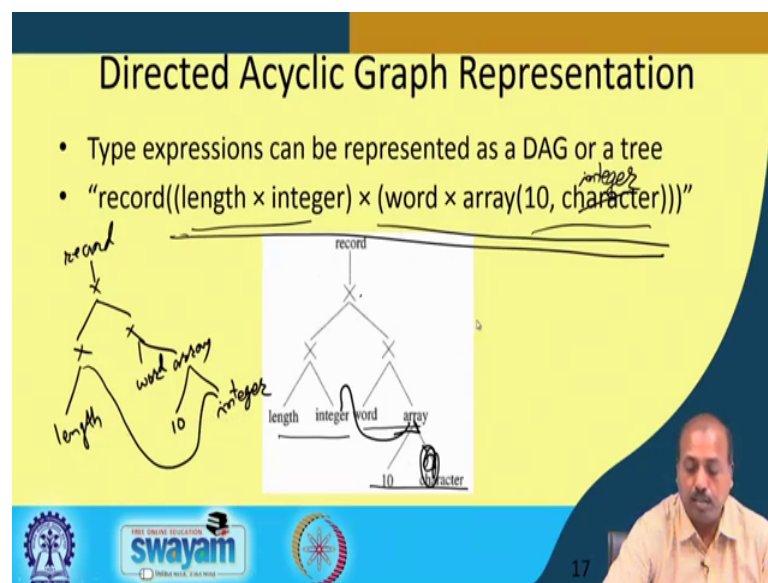
So, this check is done recursively like if it is 1 step further so till you reach a basic types. So, the check will continue. So, it will bring it down to the basic level and that basic level it will check whether they are going to be of same type or not. So, they will be checked recursively. So, if I have got types  $I1$   $T1$  an array of whose type is side size is  $I1$  and type of individual elements  $T1$  and you have got another array whose size is  $I2$  and type is  $T2$ , they will said to be structurally equivalent if  $I1$  and  $I2$  they are equal and  $T1$  and  $T2$  they are structurally equivalent.

So, this is like if I have got say 1 integer array. So, this  $I1$  equal to say 50 and  $I2$  equal to say 100; though both  $T1$  and  $T2$  are integers. So, this is also integer; this is also integer. So, in that case; so in that case, so  $I1$  and  $I2$  they are not of same type. So, naturally I will not have this the overall same size  $I1$  and  $I2$  are of this not same size.

So, if I have got some x which is array 50 int and we have got another array y which is also a variable y which is of type array of 100 integers. Then if you try to write say y equal to x, then this will give rise to a type error because so because they are basic values.

So, this is 100 and this is 50 though the basic types are integer, but it will give rise to an error. So, we can, so, this is the structural equivalences because this I 1 and I 2 these values are not matching. So, they are not structurally equivalent.

(Refer Slide Time: 04:41)



So, sometimes we use a directed acyclic graph for representing the types. So, type expression it can be represented as a directed acyclic graph or a tree, so, both of them are possible. So, for example, if we have got a record consisting of the field's length and word, where length is an integer and word is an array of 10 characters. Then this record is the overall type. So, this is a cross product of; so, at the lowest level, we have got this left side this length and integers, so, this is length cross integer.

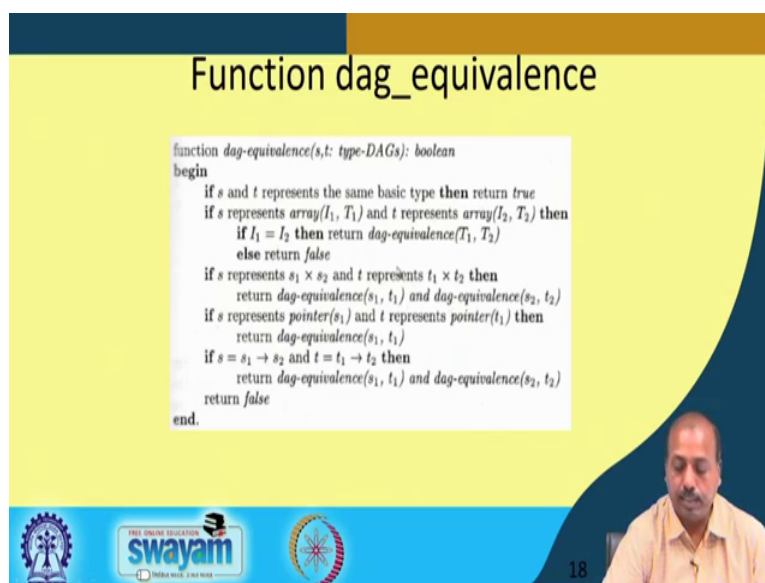
So, that is there and we have got this word cross array and that array is again this 10 characters, so, this 10 character is here. So, that is a product and then we have got the cross product of this word and array. So, that gives rise to this part and then, we have to go at this point. So, we have got the cross product of the whole thing, so, that is the record. So, in this way we can have some tree type of representation for this particular

declaration. So, there can be directed acyclic graph also like if at the lowest level it may so happen that this one instead of being a character, so this may be an integer.

If that is an integer, then this link instead of going here so, it can point to this ok. So, the second link, the second link of the array so instead of going there. So, it may point to something like that. So, this way I can have a directed acyclic graph. So, that is not a tree now because this node into. So, the overall structure is like this record. So, this is length and this one is array, this is word array.

This is 10 and this is integer and then, the other one of this also here. So, this is a directed acyclic graph representation, if this character is replaced by integer. So, we can have all this representations of this type and then, you can check for the type equivalency.

(Refer Slide Time: 07:06)



### Function dag\_equivalence

```

function dag-equivalence(s,t: type-DAGs): boolean
begin
  if s and t represents the same basic type then return true
  if s represents array(I1, T1) and t represents array(I2, T2) then
    if I1 = I2 then return dag-equivalence(T1, T2)
    else return false
  if s represents s1 × s2 and t represents t1 × t2 then
    return dag-equivalence(s1, t1) and dag-equivalence(s2, t2)
  if s represents pointer(s1) and t represents pointer(t1) then
    return dag-equivalence(s1, t1)
  if s = s1 → s2 and t = t1 → t2 then
    return dag-equivalence(s1, t1) and dag-equivalence(s2, t2)
  return false
end.

```

18

So, this is a function that checks for this equivalency of this directed acyclic graphs like if s and t represents the same basic type then return type is true; so, this is the first rule. So, if s and t they are of basic types if both are integer or real or character or Boolean like that. So, that in that case it will return that s and t are equivalent, if s is an array I 1 T 1 and t is another array I 2 T 2. So, in that case we have to if I 1 and I 2, these two values are same; then you need to check whether this at T 1 and T 2. They are of same equivalent type or not. So, accordingly it is calling this function dag-equivalency recursively with the parameters T 1 and T 2 otherwise it will return false.

So, if  $I_1$   $I_2$  are not same in that case it will say that it is they are not equivalent; otherwise it will depend on the types of this  $T_1$  and  $T_2$ . Now, if  $s$  is some record type of structure  $s_1$  cross  $s_2$  and  $t$  is another record  $t_1$  cross  $t_2$ . Then, we have to go for the equivalency check of  $s_1$  and  $t_1$  and further  $s_2$  and  $t_2$ . So, the return value will depend on the equivalency of  $s_1$ ,  $t_1$  and equivalency of  $s_2$ ,  $t_2$ . So, it checks the dag equivalency of  $s_1$ ,  $t_1$  and dag equivalency of  $s_2$ ,  $t_2$ . So, after this equivalency check; so if both of them written true, then only it will return true; otherwise it will written false.

Now, if  $s$  is a pointer  $s$  is a pointer  $s_1$  and  $t$  is a pointer  $t_1$ , so, they are 2 pointers. So, then it will be returning the dag-equivalency of  $s_1$  and  $t_1$  because if there is  $s_1$  and  $t_1$ , they are equivalent; then, this  $s$  is a  $s$  and  $t$ . So, they are also pointers so similar types. So, as a result they are declared to be equivalent. Now, if  $s$  and  $t$  they happens to be function call. So, if that is  $s$  goes from  $s_1$  to  $s_2$  its domain is  $s_1$  and range is  $s_2$  and  $t$  grows from  $t$  goes from  $t_1$  to  $t_2$  that is to had  $t_1$  is the domain and  $t_2$  is the range. Then, it will be checking the dag equivalency of  $s_1$ ,  $t_1$  to see that their domains are same and it will check the dag equivalency of  $s_2$ ,  $t_2$  to check that their ranges are same.

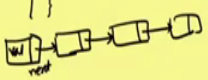
So, if they are; if they are correct then it is fine and so, otherwise if  $s$  and  $t$  do not satisfy they do not come into any of these 5 rules that we have in that case it is returning false. So, that way it is telling that this the  $s$  and  $t$  are not equivalent. So, this function dag equivalency, so it can be put a made a part of the compiler type checking procedure and then, it can check this equivalency rules and say see whether two expressions are or two expression or two function or any basic object that you have in the language, so, they are of same type or not. So, this dag equivalency check can be used.

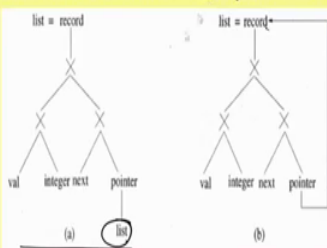
(Refer Slide Time: 10:25).

### Cycles in Type Representation

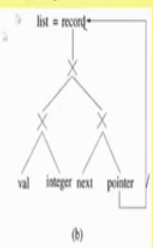
- Some languages allow types to be defined in a cyclical fashion

```
struct list
{
    int val;
    struct list *next;
}
```





(a)



(b)

- (a) Acyclic representation (b) Cyclic representation

19

So, there may be some cycles at some types of representation. So, this is particularly true when you are having say pointers. So, they that can that is a cyclical representation like in many languages like this is an example from C language, where we have got a list ok. So, in this list so as the name as the structures suggest. So, you can understand that this is actually talking about something like this.

So, this individual blocks, they have got a value and a pointer to the next field. So, this field is called next, it is pointing to the next element. So, this is again pointing to the next element, so, that way it is there. Now, how do we represent the type? So, in C language the syntax for that is something like this. So, we define the value and this struct list star next, so, this is star till struct list star. So, this part constitutes the type of the variable next ok.

So, next is a variable of type struct list star that is its a pointer to structure list. So, this is a pointer to that. So, when your when you draw the corresponding DAG, the Directed Acyclic Graph a directed graph representation, so, it no more remains a dag. So, it is no more a no more an acyclic representation. So, if you are; if you are trying to do it is an in an acyclic fashion, then it will be like this. So, this is we have got this list this record and ultimately this pointer is a pointed to list. However, this is not always very suitable because they as if you are trying to check for this type equivalency and all, then add this point there may be difficulty, because this is where is it pointing to.

So, you can have a cyclical representation here that this pointer is pointed to the list only. So, that way many times it helps in the type checking. So, both the representations are used. So, it is up to the compiler designer to check to see like which option should be used and depending on that can choose a proper parts so that this operation becomes simple ok, this check rules become simple.

(Refer Slide Time: 12:43)

### Cycles in Type Representation

- Most programming languages, including C, uses acyclic one
- Type names are to be declared before using it, excepting pointers
- Name of the structure is also part of the type
- Equivalence test stops when a structure is reached
- At this point, type expressions are equivalent if they point to the same structure name, nonequivalent otherwise

*Handwritten notes on the slide:*

- value var1
- typedef int value
- var1, var2
- Diagram showing two triangles pointing to each other, representing a cycle.

*Logos at the bottom:* IIT Bombay, Swayam, and a circular logo.

So, most programming languages, including C, they use the acyclic representation ok. So, type names are to be declared before using it excepting the pointers. So, excepting pointer so everywhere else the type names are to be predefined; like you cannot have something like this that you define a variable of type value. So, value variable 1 and sometime later you declare the you have the type definition for value. So, type def int value. So, it is coming sometime later, so, that cannot be the case. So, it has to be the other way.

So, this one should come first and then only the variable declarations can come. So, type names are to be declared before using it excepting the pointers; so, pointers you can do. So, it may so happen that this is you can say that this is a value star var 1. So, var 1 is a variable of type pointer to type value ok.

So, this since the pointer ultimately; so, pointer is a memory address. So, it is possible to allocate space for var 1 and all, so, this is allowed in the language. So, at the compiler designer does not know what is the type of var 1 actual type of var 1? So, if it is looking



for name equivalency, it is fine. But if it is looking for a structural equivalency of var 1 with something else, so it will not be able to do that because, it does not know the definition of value at this point; at this point of time. So, that way it can be problematic. However, so they are accepting pointers. So, everywhere else so there they are to be declared before using it name of the structure is also part of the type.

So, name of the structure like this value, so, that is a part of the type. Equivalency test stops when a structure is reached. So, that way it will be. So, this so, whenever your checking for this equivalency of say var 1 and some var 2, then while doing that if you draw the corresponding DAG's. So, whenever at the leaf level you have got this structure. So, it cannot be beyond that, so, it will stop at that point.

So, at this point they are equivalent if the point to the same structure name and non equivalent otherwise. So, if this is pointing to say least and say this is pointing to this is pointing to say list 1 and this is pointing to least 2. So, in that case they are taken to be structurally non equivalent. Though it may so happen that list 1 and least 2, if you define it further. So, they are boiling down to send set of fields in them. So, but this we have to stop at some point of time. So, this type check, so, it stops at that point. So, it does not go beyond the point, where it has seen some structure ok; so, it will stop at that point.

(Refer Slide Time: 15:42)

## Type Conversion

- Refers to local modification of type for a variable or subexpression
- For example, it may be necessary to add an integer quantity to a real variable, however, the language may require both the operands to be of same type
- Modifying integer variable to real will require more space
- Solution: to treat integer operand as really operand locally and perform the operation
- May be done explicitly or implicitly
- Implicit conversion → type coercion




```
int x;
float y;
...
y = ((float)x)/14.0
```

```
int x;
float y;
...
y = x/14.0
```

*Handwritten notes:*

$int\ x \rightarrow real\ x$   
 $real\ y$

$z = \frac{x+y}{14}$   
 $\downarrow$        $\downarrow$   
 $int$        $real$   
 $4\ bytes$        $8\ bytes$

Sometimes we need to do type conversion like from one type we need to convert to another type. So, it refers to the local modification of type of a variable or sub



expression. So, this is commonly known as this type coercion sort of thing. So, it may be necessary to add an integer quantity to a real variable. However, the language may required both the operands to be of same type. As I was telling previously that I am a required that  $x + y$ , so,  $z = x + y$  has to be done and this  $x$  is of type integer and this  $y$  is of type real. So, whenever the this language says that whenever you have got this plus operator. So, they are the operands should be of same type;  $x$  and  $y$  they should be of same type.

But it may not happen because at some points of time they will. So, it may be; it may be that if they are of different types and this may not be taken as an error. So, whenever the though the language says that they should be of same type, a sub expression should be of same type; but it may be that it will not be a completely an error. So, it can give a warning the compiler designer can produce an warning that this operands are of two different types for this particular operator. However, the major difficulty that we have is that these integer and real their sizes are not same; very often this integer maybe of 4 bytes and this real maybe of 6 bytes.

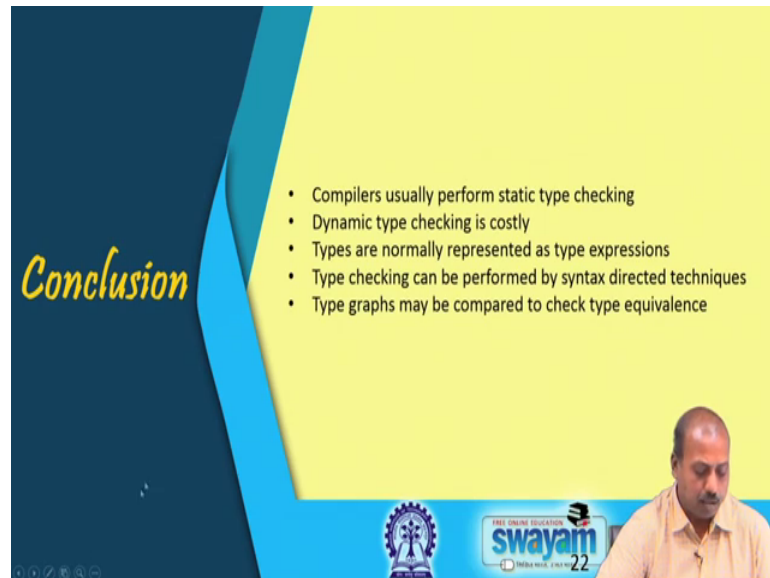
So, whenever you are doing this addition. So, you need to convert this 4 bytes to 6 bytes, then only you can really do this addition; otherwise it will be a problem. So, modify an integer variable to real it will require more space, so, to treat integer operand as real really as a real operand locally and perform the operation. So, one option is that in the whenever you find like this.

So, you have defined like integer  $x$  real  $y$  in your program. So, what the compiler may do is that it will covert this type of this integer to real; so, type of these  $x$  to real. So, that way it will always take  $x$  as a real variable, but if you do that, then the difficulties that for the entire program. So, this  $x$  variables representation will be 6 bytes. So, wherever this  $x$  will be referred it will require 6 bytes.

So, that way the space required by the program will be large. So, what the suggestion is that since at this point only we have got this addition with real. So, for this particular statement only we modify the type of  $x$ . So, this is called this is the local changing of this type and this may be done explicitly or implicitly. Either so if it is done implicitly, then it is called this is known as type coercion. So, this is an explicit definition. So, this is so, this  $y = x + y$ . So, since this  $x$  is a integer variable.

So, we convert it locally to a float by doing a typecasting and then this is the; this is the type coercion. So, this is basically this; so, this x you can also write like x by 14.0. So, in that case implicitly it will convert this x to float and then assign it to y; whereas, this one, so this is explicit and here the programmer has explicitly mentioned that this has to be done in this fashion.

(Refer Slide Time: 19:20)



*Conclusion*

- Compilers usually perform static type checking
- Dynamic type checking is costly
- Types are normally represented as type expressions
- Type checking can be performed by syntax directed techniques
- Type graphs may be compared to check type equivalence

swayam 22

So, compilers can perform this static type checking and dynamic type checking is definitely costly. So, types are normally represented as type expressions and type checking can be performed by syntax directed techniques and type graphs may be compared to check for type equivalency. Next we will be doing a few exercises on this type expression.

(Refer Slide Time: 20:09)

```
E → E1 + E2
{
  check E1.type and E2.type to be one of char, int, float, double;
  If other types E.type ← type-error
  If E1.type = float and E2.type = float
    E.type = float
  else if E1.type = double and E2.type = double
    E.type = double
  else if E1.type = int and E2.type = int
    E.type = int
  else E.type = char
}
```

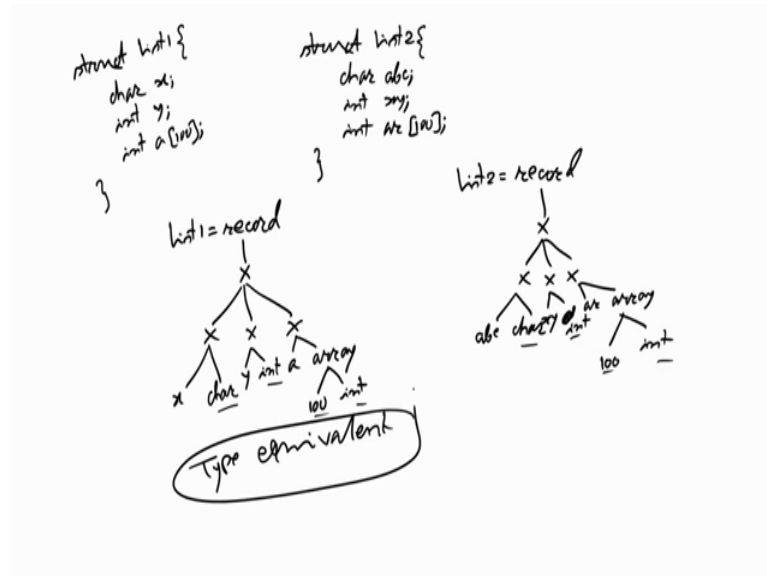
So, the first one that we will look into is for the type coercion. So, we have previously seen that type coercion this type check of this expression likes say  $E$  producing  $E_1$  plus  $E_2$ . So, previously we said that if these type of  $E_1$  and type of  $E_2$  are same then it is fine otherwise there is a problem. So, we can modify that check and we can do it like this.

So, first we can check  $E_1$  dot type and  $E_2$  dot type check  $E_1$  dot type and  $E_2$  type to be one of the basic types like character integer; then float doubles. So, if we are talking in terms of C language. So, these are the basic types that we have. So, it has to be one of those types. So, if it happens to be like that, then if other types if it is not one of those basic types, then we can say that  $E$  dot type is type error.

So, this is a type error, otherwise for if  $E_1$  dot type is. So, if  $E_1$  dot type is float and  $E_2$  dot type is also float if both of them are float, then we can say that this  $e$  dot type is equal to float. Similarly, if both of them are double else, if  $E_1$  dot type equal to double and  $E_2$  dot type equal to double, then you can say that  $E$  dot type equal to double. Else if  $E_1$  dot type equal to integer and  $E_2$  dot type is also equal to integer. So, in that case  $E$  dot type is equal to integer, else  $E$  dot type equal to character.

So, this may be the detailed set of rules to see that it follows its falls into one of these operator types and it is doing the operation properly ok. So, you can do this check and for other rules also we can do this thing.

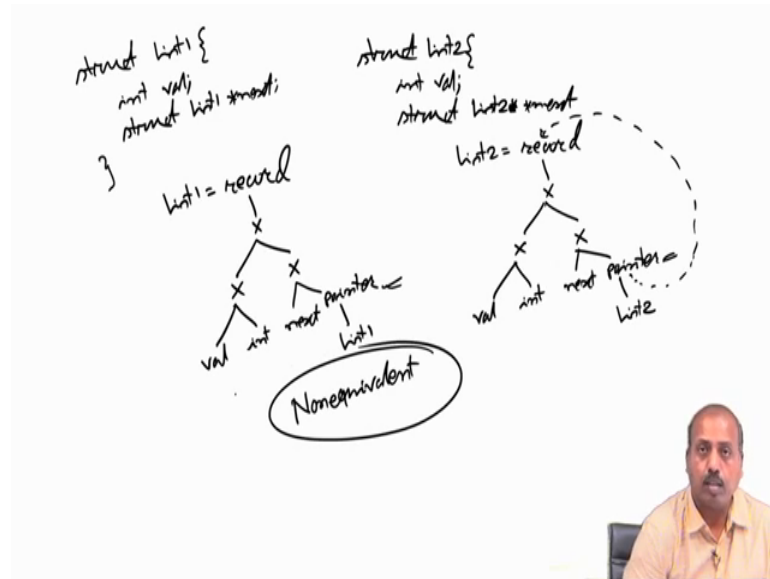
(Refer Slide Time: 23:42)



Next we will be looking into; next we will be looking into another example of say one least type. So, this is again a C declaration structure list 1 that has got fields like character x integer y integer a 100 and there is another structured list 2 and there we have got character a b c, then integer x y, then integer array 100. Now, whether they are type equivalent or not if we want to check, then we will be; we will be construct the corresponding DAG's. So, this is a record it has got 3 parts in it. This one is having x character, this is y integer and this is having a array and then we have got 100 and integer.

Now this one, so list 2 is a record it also has got 3 fields in it. So, one is sorry a b c character, then this x y integer and this is an array a r 100 integer. So, after constructing this 2 such DAG's we can see that they are going to be equivalent because at the lowest level you have got this character integer array 100 integers. So, here also you have got character integer array 100 integer, so, they are type equivalent. So, these are type equivalent. So, if you take some other example. So, it may be that they are not type equivalent.

(Refer Slide Time: 26:29)



So, let us take another example, where this first list is again a structure list 1 and there we have got integer value and this structure list 1 star next and we have got this structure list 2 integer value and structure list 2 star next. So, here also if we draw the corresponding trees, so, DAG's. So, this is list 1 equal to record and then we have got this 2 parts in it; one is integer value and here I have got this next which is a pointer to list 1 and here this list 2 is equal to record, then this is 2 part in it 1 is int value integer and the other part is next it is a pointer, but pointer to list 2.

Now, these are non equivalent because when it comes to this point. So, this up to this much it is fine this, but this is a pointer to list 1 and this is a pointer to list 2 as a result. So, these 2 are going to be non equivalent even if we. So, this is an acyclic presentation. So, even if we have got a cyclic representation, where instead of doing it like this. So, we give it back to this point, then also it does not help because here also it is telling that is a pointer to list 2. So, that as the equivalency check we have said that it stops, so, when it reaches the basic structure.

So, it will stop at this point and here it will find that this is a list 2, this is list 1; there records are difference. So, they are non equivalent, so, whatever way we represent it. So, these 2 types are non equivalent types.

Thank you.