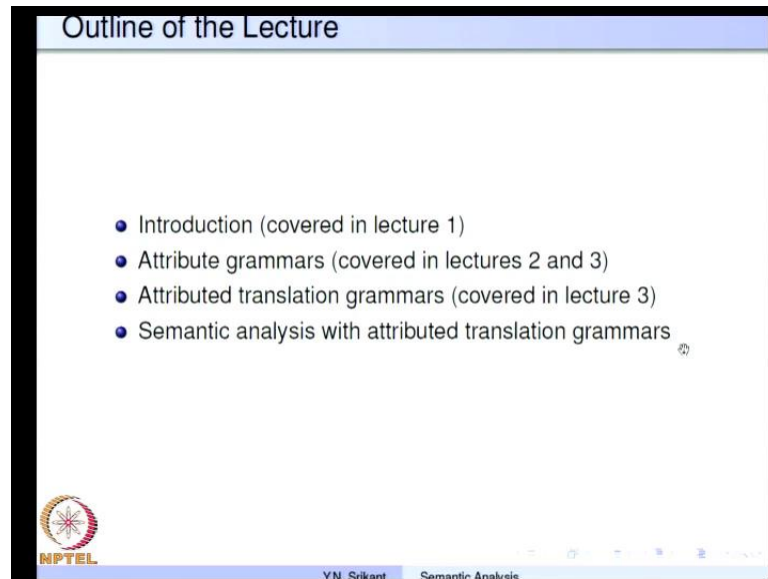


Principle of Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Lecture - 16
Semantic Analysis with Attribute Grammars Part - 5

(Refer Slide Time: 00:19)



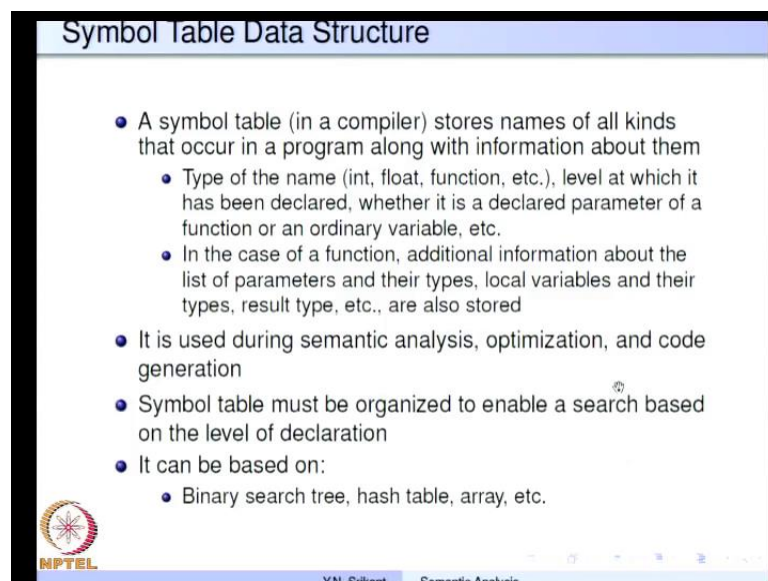
Outline of the Lecture

- Introduction (covered in lecture 1)
- Attribute grammars (covered in lectures 2 and 3)
- Attributed translation grammars (covered in lecture 3)
- Semantic analysis with attributed translation grammars

MPTEL Y.N. Srikant Semantic Analysis

Welcome to part five of the lecture on attribute grammars. We will continue with semantic analysis and attributed translation grammars today as well.

(Refer Slide Time: 00:28)



Symbol Table Data Structure

- A symbol table (in a compiler) stores names of all kinds that occur in a program along with information about them
 - Type of the name (int, float, function, etc.), level at which it has been declared, whether it is a declared parameter of a function or an ordinary variable, etc.
 - In the case of a function, additional information about the list of parameters and their types, local variables and their types, result type, etc., are also stored
- It is used during semantic analysis, optimization, and code generation
- Symbol table must be organized to enable a search based on the level of declaration
- It can be based on:
 - Binary search tree, hash table, array, etc.

MPTEL Y.N. Srikant Semantic Analysis

So, let us start looking at the symbol table structure, which is necessary to perform semantic analysis. So far I only added a few routines to search and insert into the symbol table, and showed you how to do some semantic analysis. But I did not give you a structure for the symbol table and data structure for the symbol table. So, let us do that now, a symbol table stores names of all kinds that occur in a program along with the information about them. So, for example, the type of the name you know whether it is an integer or a floating point number or function, etcetera. The level at which it has been declared. So, it is possible to have blocks in C and in Pascal and you know Algol we can even have procedures and functions embedded within other procedures and functions.

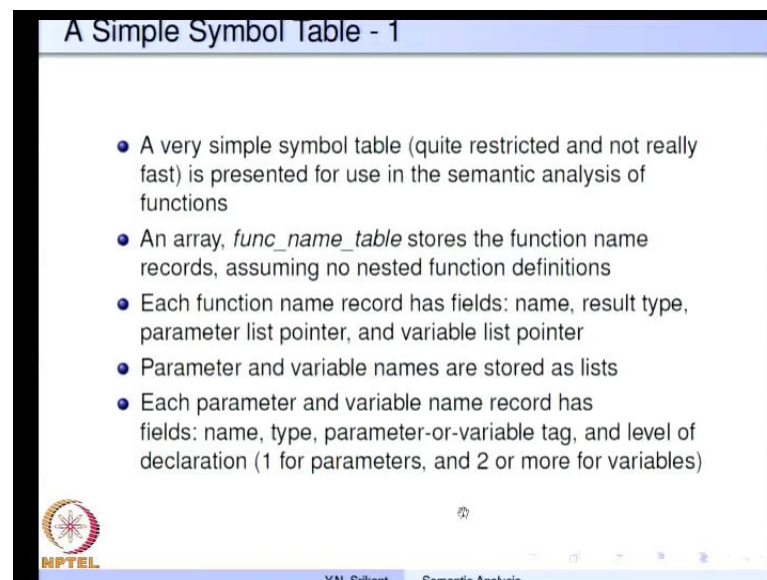
So, the level at which the name has been declared. So, which block you know the level of the block whether it is a declared parameter of a function or it is an ordinary variable all this information must along with the name itself. In the case of a function we also need to access the list of parameters from the function name along with their types of course. And we must also be able to say that these are the set of local variables corresponding to this particular function the result type of the function, etcetera, etcetera. All these must be stored along with the function name in the symbol table.

During semantic analysis of course, we use the symbol table we already know that it is necessary. But we actually perform you know intermediate code generation in which the variables that we use including the temporaries that are necessary will all be installed in the symbol table you know. So, at the end of the intermediate code generation phase we will still have pointers into the symbol table for all the names. Therefore, during the machine independent code optimization phase, it becomes essential to access the symbol table as well. And during machine code generation we definitely need to access the symbol table to find out the type of the name, how much storage is needed for a, what is the format in which must be stored. These are can be these all this can be determined only by looking at the type of that particular name how do we organize the symbol table.

So, it must be organized to enable a search based on the level of declaration this will become clear as we go along. And what are the data structures that can be used to organize a symbol table. Well if we organize it as a binary search tree then it is fairly straight forward to insert, delete, search this search tree. But then the amount of time needed to search or insert or delete a name is proportional to $\log n$ where n is the number of nodes in the tree that is assuming that the binary search tree is also a balanced binary

search tree. To get a better you know search time a hash table is quite easy to use. So, we can sue a hash table to store the symbol table, but there may be some issues regarding levels. And one must organize the hash table carefully if you want to you know search based on levels, a simple array that is always possible a link list is also possible.

(Refer Slide Time: 04:38)



A Simple Symbol Table - 1

- A very simple symbol table (quite restricted and not really fast) is presented for use in the semantic analysis of functions
- An array, *func_name_table* stores the function name records, assuming no nested function definitions
- Each function name record has fields: name, result type, parameter list pointer, and variable list pointer
- Parameter and variable names are stored as lists
- Each parameter and variable name record has fields: name, type, parameter-or-variable tag, and level of declaration (1 for parameters, and 2 or more for variables)

NPTEL YN. Srikant Semantic Analysis

So, let us look at a simple structure to store a symbol table. So, we will our data structure will be a very simple data structure. And it is going to be quite restricted and not really very fast the reason is the description of a complex symbol table. You known is actually an unrelated topic any complicated data structure may be suitable for a symbol table. And it does not add value to the semantic analysis or other compiler functions as far as the lecture goes.

So, we will use a simple symbol table and then I will leave it to the students to understand how to use more complicated symbol tables. We will have it as an array called as the function name table and this table stores the function name records. Assuming that there are no nested function definitions of there are nested function definitions. Then the table can be changed its possible to do that I will show you how to do it little later each function name record has fields, it has name it has result type parameter list pointer and variable list pointer.

(Refer Slide Time: 06:02)

The slide, titled "A Simple Symbol Table - 2", illustrates two data structures used in semantic analysis. The first is a table labeled "func_name_table" with five columns: "name", "result type", "parameter list pointer", "local variable list pointer", and "number of parameters". The second is a table labeled "Parameter/Variable name record" with four columns: "name", "type", "parameter or variable tag", and "level of declaration".

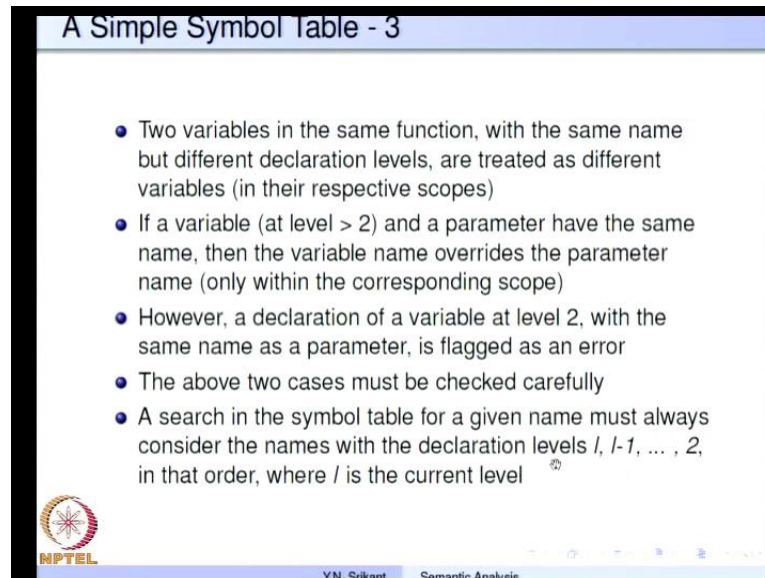
name	result type	parameter list pointer	local variable list pointer	number of parameters

name	type	parameter or variable tag	level of declaration

So, let me show you a picture. So, the function name table is a simple array or a table. So, with several fields name result type parameter list pointer you know local variable list pointer and then the number of parameters. So, each function name record you know has fields. So, and then the parameter and variable names are stored as lists. So, I showed you the pointer to these lists. So, why should we store these as lists? So, for example, we want to access all the parameters possibly together and similarly we may want to search only variable names and not get mixed up with parameters and so on.

So, it is for this reason the order in which the parameters are stored is also very important for us while checking the declaration and the call. So, each parameters and variable name record has many fields. So, it has name it has type it has parameter or variable tag and the level of declaration. So, the level is set as one for the parameters and two or more for the variables. So, this is how a parameter variable name record looks like there is name type a tag and then level of declaration.

(Refer Slide Time: 07:28)



A Simple Symbol Table - 3

- Two variables in the same function, with the same name but different declaration levels, are treated as different variables (in their respective scopes)
- If a variable (at level > 2) and a parameter have the same name, then the variable name overrides the parameter name (only within the corresponding scope)
- However, a declaration of a variable at level 2, with the same name as a parameter, is flagged as an error
- The above two cases must be checked carefully
- A search in the symbol table for a given name must always consider the names with the declaration levels $l, l-1, \dots, 2$, in that order, where l is the current level

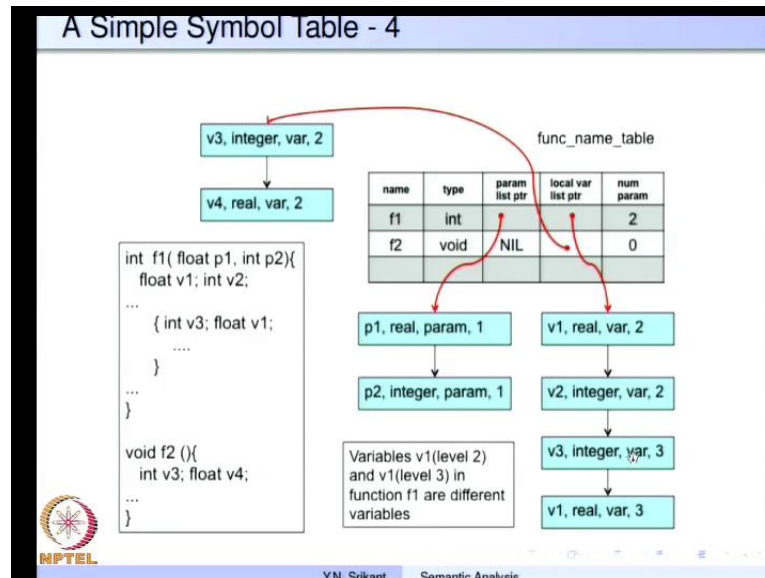
MPTEL

Y.N. Srikant Semantic Analysis

Suppose the, let me not describe the procedure for searching and inserting names etcetera. Suppose there are two variables in the same function, with the same name, but declaration levels. These are; obviously, treated as different variables in the respective scopes this is governed by the language specification itself and the implementation will also support it. If your variable at level greater than two and a parameter have the same name then the variable name overwrites the parameter name. So, only within the corresponding scope this is very important, but let me read the next point. And then let us see what the implications are; however, a declaration of a variable at level two with the same name as a parameter is flagged as an error.

So, that implication is at the level just after the parameter declaration if you have a name which is the same as that of a parameter then it is an error. But if you declare it an interior level a name which is the same as that of a parameter then the local name which is declared in the block overwrites the parameter name. So, these cases must be checked very carefully and we will see how to do that. And whenever we search a symbol table for a given name then you know suppose we start at level L . The name has been used at level L then we must search the name in the symbol table at level L and if it is not found then at level minus one etcetera, till level 2 in that order where L is the current level. So, if you find the name at level L itself and that is the declaration which is taken as relevant to us. Otherwise whenever we as soon as we hit a declaration which is the, which satisfies our requirement we take that as the relevant declaration.

(Refer Slide Time: 09:48)



So, let me show you how the symbol table looks like for a small sample program here there is a function f 1. And there is another function f 2, f 2 does not return any result whereas, f 1 returns an integer result f 1 has 2 parameters P 1 and P 2 the first one is of type floating point the other one is of type integer. So, whereas, f 2 it does not have any parameters either. So, let us look at the name function name record in the function name table. So, the name of the function is f 1 it is it returns a result of type int whereas, f 2 returns a result which is y void and then the name the function f 1 has a parameter list.

So, the parameter list points to a link list containing P 1 and P 2 in which the information about its type real whether it is a parameter or otherwise and the level are all stored. And then the second function f 2 does not have any parameters. So, this becomes a nil pointer and within the function f 1 we have float V 1 and int V 2 at the level just below the parameters. And then somewhere inside there is a nested block with int V 3 and float v one. So, corresponding to that there is a link list you know of variables which is pointed to by the local var list pointer. So, the variables are stored in the order of declaration.

So, first is V 1 its type is real it is a variable and its level is 2. The next one is V 2 it is an integer it is a variable and level 2 the third one would be V 3 integer variable. But the level is 3 similarly the fourth one again is V 1 real variable and level three. So, what we must observe here is this name V 1 is the same as this name v one. But since this has been declared in an interior block, this name actually holds within this block whereas,

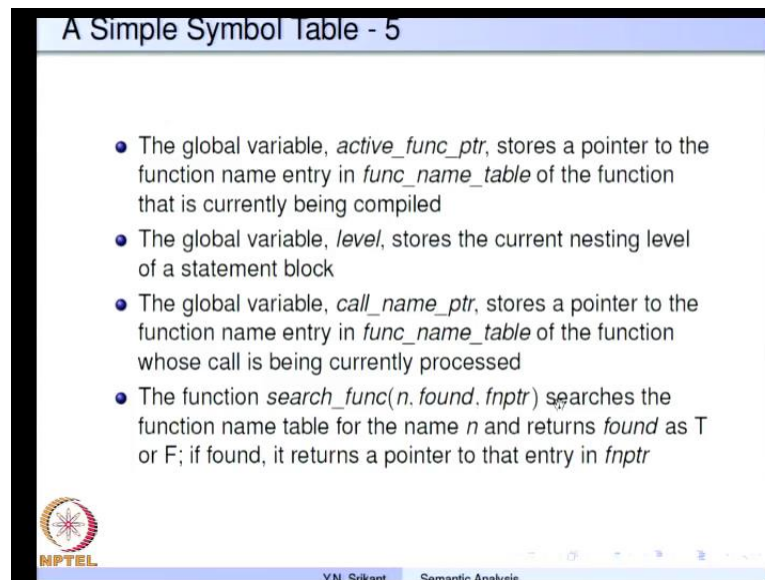
this name does not hold within this block. So, whenever you make an assignment or read from the variable V this is the declaration which is valid for us. So, this could have been integer as well. So, if you had made it int V 1 then int V 1 would have been relevant within this block.

So, and similarly suppose this name P 1 was again redeclared here you know. So, say int P 1 this would have been an error, because it is just below the parameter list. Whereas, here if we had another int P 1 it would have actually overwritten this declaration within this particular block it would not have been flagged as an error what about function f 2. So, the function f 2 also has 2 variables V 3 and V 4. So, they are linked in this list and its pointer is present in the local var list pointer here the number of parameters of f 1 is 2 and the number of parameters of f 2 is 0.

So, this is the structure of our simple symbol table. So, the variables V 1 at level 2 and V 1 at level 3 in the function f 1 are different variables they are not treated as the same. And when we search for a name we actually start from let us say we are within this block. So, we really start from this point then you know we search this list and then we need to search the rest of it as well if we do not find anything within this list then we need to search the others as well. So, this is how it is. So, what we really need to do is make sure that we get the right variable.

So, of course, it is always possible to have another you know maintain another link which is in the reverse directions. So, that the search becomes faster and. So, on, but that is you know a minor detail. So, if we have a reverse link as well then whenever we want to search something we could search here if you do not get anything. Then we could start search in the reverse direction, but we should be careful to make sure that the search covers the levels at you know in the appropriate order.

(Refer Slide Time: 14:47)



A Simple Symbol Table - 5

- The global variable, *active_func_ptr*, stores a pointer to the function name entry in *func_name_table* of the function that is currently being compiled
- The global variable, *level*, stores the current nesting level of a statement block
- The global variable, *call_name_ptr*, stores a pointer to the function name entry in *func_name_table* of the function whose call is being currently processed
- The function *search_func(n, found, fnptr)* searches the function name table for the name *n* and returns *found* as T or F; if found, it returns a pointer to that entry in *fnptr*

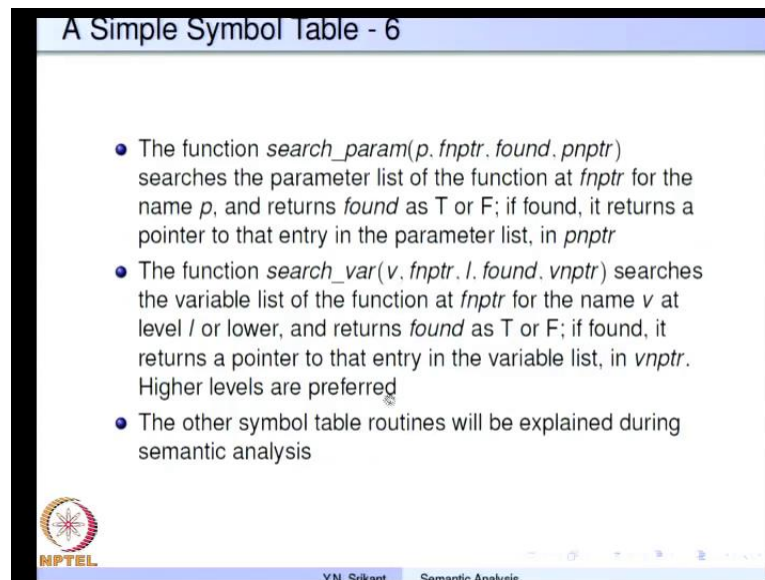
NPTEL

Y.N. Srikant Semantic Analysis

So, carrying this discussion further now, let us see what are the various global variables and function we require to operate the symbol table. So, we require a variable active func p t r its stores a pointer to the function name entry in the function name table of the function that is being currently compiled. So, this will tell us that you know we are now looking at a particular function and it is being processed. The global variable level stores the current nesting level of a statement block. The global variable call name pointers stores a pointer to the function name entry in the function table of the function that is being whose call is being currently processed.

So, there is a difference between the call name pointer and the active function pointer. Active function pointer is useful when we compile functions and call name pointer is useful when we process calls to a particular function. Then the search function function the it actually takes three parameters n found and f n p t r it is you know starts for its searches the symbol table for the name n its searches only the functions and then if it is found it returns found as true otherwise it returns found as false if the name is found then a the pointer to that particular name is returned in f n p t r. So, this is the use of such function routine.

(Refer Slide Time: 16:43)



A Simple Symbol Table - 6

- The function `search_param(p, fnptr, found, pnptr)` searches the parameter list of the function at `fnptr` for the name `p`, and returns `found` as T or F; if found, it returns a pointer to that entry in the parameter list, in `pnptr`
- The function `search_var(v, fnptr, l, found, vnptr)` searches the variable list of the function at `fnptr` for the name `v` at level `l` or lower, and returns `found` as T or F; if found, it returns a pointer to that entry in the variable list, in `vnptr`. Higher levels are preferred.
- The other symbol table routines will be explained during semantic analysis

MPTEL
Y.N. Srikant Semantic Analysis

Then the routine search PARAM searches the parameter list of the, you know function at `fnptr`. So, this, the function pointer whose parameter list has to be searched and if you find it this found is made as true otherwise it is made as false. And if it is if we find a parameter declaration then a pointer to that is a turned in `pnptr`. So, the function search var does the similar thing on the variable list. So, the name is `v` the function in which we want to search is `fnptr` the pointer to that the level is `L` found is a flag and `vnptr` is the pointer to it if the name is found.

So, this search must be started at level `L` and then it should go to lower levels. So, this is where you know providing a reverse link is useful in this list so, but I am only showing you the minimal structure that is needed to make the symbol table functional. But as I said efficiency is not the most important thing it is the understanding of how semantic analysis is performed and how the information is stored in the symbol table that is important to us at this point. So, the other symbol table routines will be explained as we go on.

(Refer Slide Time: 18:22)

The slide displays a grammar for semantic analysis of functions and calls, consisting of 12 numbered rules:

- 1 $FUNC_DECL \rightarrow FUNC_HEAD \{ VAR_DECL \ BODY \}$
- 2 $FUNC_HEAD \rightarrow RES_ID (DECL_PLIST)$
- 3 $RES_ID \rightarrow RESULT \ id$
- 4 $RESULT \rightarrow int \mid float \mid void$
- 5 $DECL_PLIST \rightarrow DECL_PL \mid \epsilon$
- 6 $DECL_PL \rightarrow DECL_PL \ . \ DECL_PARAM \mid DECL_PARAM$
- 7 $DECL_PARAM \rightarrow T \ id$
- 8 $VAR_DECL \rightarrow DLIST \mid \epsilon$
- 9 $DLIST \rightarrow D \mid DLIST \ ; \ D$
- 10 $D \rightarrow T \ L$
- 11 $T \rightarrow int \mid float$
- 12 $L \rightarrow id \mid L \ . \ id$

The slide also features the NPTEL logo in the bottom left corner and the text 'Y.N. Srikant Semantic Analysis' in the bottom right corner.


So, here is the grammar that we consider for as analyzing the functions and calls to functions. A function declaration consists of a function head then it has a variable declaration list and a body function head says there is a result and I d. So, this is a, you know we are breaking up this production to enable better semantic analysis. So, there is result is either integer float or void i d is the name of the function and then we have a parameter lists here. So, the declaration PLIST generates several declarations.

So, DECL P L of epsilon so, this is necessary to make sure that we can have 0 parameters or more than you know 0 parameters. So, if you have 0 parameters then we use the epsilon option otherwise we use the DECL P L option. So, DECL P L option so, DECL P L generates a list of params. So, declaration of a parameter is a type followed by name. So, the type can be either integer or float and variable declaration here again this we have seen already there is a list of declarations and each declaration is of the form D going to T l. So, L is a list of names for this declaration.

(Refer Slide Time: 20:02)

SATG for Sem. Analysis of Functions and Calls - 2

- 13 $BODY \rightarrow \{ VAR_DECL\ STMT_LIST \}$
- 14 $STMT_LIST \rightarrow STMT_LIST ; STMT \mid STMT$
- 15 $STMT \rightarrow BODY \mid FUNC_CALL \mid ASG \mid /*\ other\ */$
/ BODY may be regarded as a compound statement */*
/ Assignment statement is being singled out */*
/ to show how function calls can be handled */*
- 16 $ASG \rightarrow LHS := E$
- 17 $LHS \rightarrow id\ /*\ array\ expression\ for\ exercises\ */$
- 18 $E \rightarrow LHS \mid FUNC_CALL \mid /*\ other\ expressions\ */$
- 19 $FUNC_CALL \rightarrow id\ (PARAMLIST)$
- 20 $PARAMLIST \rightarrow PLIST \mid \epsilon$
- 21 $PLIST \rightarrow PLIST , E \mid E$

 Y.N. Srikant Semantic Analysis

Going further the body has variable declarations followed by statement lists. So, these are the variables which are local to the block and the block is began using the flower bracket. And ended with another flower bracket statement list in the block generates a list of statements that is quite easy to understand and each statement could be again a body itself. So, the, it starts a new body a block it could be a function call it could be an assignment and many others we have already considered the other types of statements. So, we will consider only these three to show our semantic analysis on function declarations and function calls.

So, the body actually can be regarded as a compound statement and assignment statement has been singled out. Because we want to show how functions can be functions calls can be handled an assignment has a form left hand side and then there is a, an assignment operator and E an expression. So, the left hand side we have already seen this can be either an identifier or it you know array expressions are all leftover exercises. So, we are not we have done that already to some extent.

So, extension of the symbol table for arrays etcetera is left for exercises then the expression on the right hand side. So, L H S can only be a name expression on the right hand side can be either L H S which generates the name. Or it could be a function call or many other types of expressions which we are not concerned with. So, what is a function call? It is a name followed by a parameter list. So, the parameter list can be empty or it

could be one or more parameters. So, we generate the parameters as expressions here using this production.

(Refer Slide Time: 22:11)

SATG for Sem. Analysis of Functions and Calls - 3

- ➊ $FUNC_DECL \rightarrow FUNC_HEAD \{ VAR_DECL \ BODY \}$
 $\{ delete_var_list(active_func_ptr, level);$
 $active_func_ptr := NULL; level := 0; \}$
- ➋ $FUNC_HEAD \rightarrow RES_ID (DECL_PLIST) \{ level := 2 \}$
- ➌ $RES_ID \rightarrow RESULT \ id$
 $\{ search_func(id.name, found, namptr);$
 $if (found) error('function already declared');$
 $else enter_func(id.name, RESULT.type, namptr);$
 $active_func_ptr := namptr; level := 1 \}$
- ➍ $RESULT \rightarrow int \{ action1 \} \mid float \{ action2 \}$
 $\mid void \{ action3 \}$
 $\{ action \ 1: \} \{ RESULT.type := integer \}$
 $\{ action \ 2: \} \{ RESULT.type := real \}$
 $\{ action \ 3: \} \{ RESULT.type := void \}$

NPTEL
YN, Siliguri Semantic Analysis

So, let us look at the semantic analysis for the function declarations first function declaration starts with function head var declaration and body. So, we now look at a Tg that is the, with synthesized attributes. So, there is no need to write the arrow. So, what is it that we require? So, when we process the function declaration by the time we try execute this action the entire function has been parsed and analyzed. So, at this point we really delete the variable list of functions which we have function which we have just now parsed and analyzed why. And that is pointed to by active func p t r, the level of the function of course, is level itself which is; obviously, one at this point. If why should we delete the list well we have no use for the variable list after the body of the function has been completely analyzed. So, that is the reason we do not need the variable list anymore.

So, at this point I want to show you that the deletion is important, but later I will also show you how to keep this variable list for use in a two parse compiler. The assumption here is the intermediate code generation also happens along with semantic analysis and parsing. So, after the intermediate code has been generated we do not require the variable list. However, if you observe this we have not deleted the parameter list there reason is one the function body has been completed. We do not need the, we do not need the

variable list, but we are going to use the function at a later point in time. So, at that point we definitely require the parameter list to check against the parameters in the call.

So, we must be careful to delete only their variable list which is not needed anymore, but not the parameter list. So, we just reset the active function pointer to null and the level is made as 0 to make sure that we start all over again. So, here function head goes to `res i d` and then a declaration of the `PLIST` the parameter list. So, at this point in the production we have completed the analysis of this entire right hand side. So, the level can be raised to two to make sure that we now start processing variables. So, at this point where the level is 0 and then functions are entered with at level 0 parameters at level one and variables a level two or more the `res i d non terminal` expands to result followed by name.

So, now, there is some action we need to search the function table for this name if the name is already present. Then there is an error which is declared that the function is already declared and if the name is not present we enter the function with this particular name this particular type. And this name `p t r` is the pointer returned by the routine and this pointer points to this particular function entry in the symbol table. So, active function pointer is set as name `p t r`, because that is the function which we are now going to analyze level is set as one. So, now, level one implies parameter list. So, that is why we need to parse this next. So, result of course, either goes to `int` or `float` or `void` and the actions are very simple the result type is set as an integer real or void appropriately. So, now, let us look at the declaration `PLIST`.

(Refer Slide Time: 26:26)

The slide contains the following content:

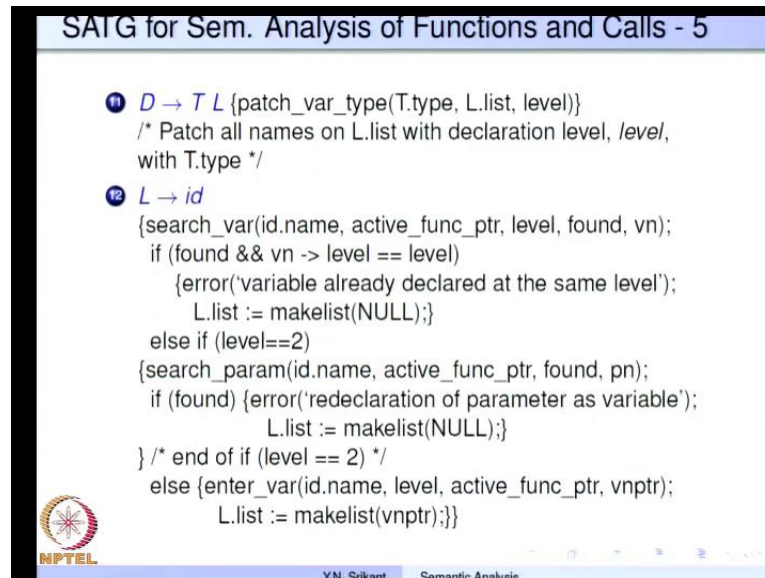
- 5 $DECL_PLIST \rightarrow DECL_PL \mid \epsilon$
- 6 $DECL_PL \rightarrow DECL_PL \ . \ DECL_PARAM \mid DECL_PARAM$
- 7 $DECL_PARAM \rightarrow T \ id$
{search_param(id.name, active_func_ptr, found, pntpr);
if (found) {error('parameter already declared')}}
else {enter_param(id.name, T.type, active_func_ptr)}
- 8 $T \rightarrow int \ {T.type := integer} \mid float \ {T.type := real}$
- 9 $VAR_DECL \rightarrow DLIST \mid \epsilon$
- 10 $DLIST \rightarrow D \mid DLIST \ ; \ D$
/* We show the analysis of simple variable declarations.
Arrays can be handled using methods described earlier.
Extension of the symbol table and SATG to handle arrays
is left as an exercise. */

At the bottom left is the NPTEL logo. At the bottom center, it says 'Y.N. Srikant Semantic Analysis'.

So, this declaration PLIST going to DECL P L or epsilon there is nothing to do similarly this also expands to DECL P L DECL PARAM or DECL PARAM there is nothing to do. But once we have a concrete declaration DECL PARAM goes to T i d we have an action as well. So, as usual we search whether this name has already been declared as a parameters using the routine search PARAM. So, if it is already found to found in the symbol table there is an error parameter already declared otherwise we enter the name into the symbol table using the routine enter PARAM.

The type of the name is T dot type the pointer at which the function corresponding to this parameter list exists is active func p t r. So, enter PARAM does its job and enters the name into the a symbol table at the appropriate place T expands to int or float which is trivial we have seen this. So, many times already var DECL going to DLIST or epsilon and DLIST list going to D or D list semicolon D these are quite simple there is nothing true really. Now, once we expand D we will have some action. So, we will show the analysis of simple variable declarations and handling arrays is let as a exercise.

(Refer Slide Time: 27:59)



The slide contains the following code:

```
11 D → T L {patch_var_type(T.type, L.list, level)}
/* Patch all names on L.list with declaration level, /level,
with T.type */

12 L → id
{search_var(id.name, active_func_ptr, level, found, vn);
 if (found && vn -> level == level)
   {error('variable already declared at the same level');
    L.list := makelist(NULL);}
 else if (level==2)
 {search_param(id.name, active_func_ptr, found, pn);
  if (found) {error('redeclaration of parameter as variable');
   L.list := makelist(NULL);}
 } /* end of if (level == 2) */
 else {enter_var(id.name, level, active_func_ptr, vnptr);
   L.list := makelist(vnptr);}}
```

MPTEL logo is visible in the bottom left corner of the slide.

D going to T L so, now, L has generated a list of names. So, we have seen this before, but let me just repeat it you know for the sake of continuity patch var type patches. The entire list, L dot list at the level L with all the names in this L dot list at the level L, with the type T dot type. So, that takes care of completing the information within the symbol table or the particular name then L going to I d. So, this generates either a single name or many names. So, let us see, what happens? When we search or a name. So, here so far you know we have already processed the parameters.

So, now we are processing variables. So, that is why we search whether the name has already been declared in the variable list at the appropriate level. So, if the name is found and the level is the current level then we are not allowed to declare the name again. So, the error message variable already declared at the same level is issued and we make the list as null. So, there is nothing more to do here otherwise if the level is two so; that means, we dint find the variable in the variable list anywhere. But we also have to check whether it has been declared before as a parameter which is also an erroneous situation.

So, search PARAM or this name and if it is found when redeclaration o a parameter as variable that is the error message we see which is issued. And we make L dot list as null if no there are no errors found and the name is not in the symbol table. We enter the variable with the name i d dot name at the appropriate level and the place where to enter is pointed to by active func p t r. So, this is the function in which we are analyzing the

variables. So, once the entry is made a pointer to that entry is returned in `vnptr`. So, `L` dot list now has make list `vnptr`. So, `vnptr` is one of the nodes in that list. So, this is a single variable.

(Refer Slide Time: 30:43)

SAIG for Sem. Analysis of Functions and Calls - 6

```

13  $L_1 \rightarrow L_2 . id$ 
   {search_var(id.name, active_func_ptr, level, found, vn);
   if (found && vn -> level == level)
       {error('variable already declared at the same level');
         $L_1$ .list :=  $L_2$ .list;}
   else if (level==2)
       {search_param(id.name, active_func_ptr, found, pn);
        if (found) {error('redclaration of parameter as variable');
                     $L_1$ .list :=  $L_2$ .list;}
        } /* end of if (level == 2) */
   else {enter_var(id.name, level, active_func_ptr, vnptr);
         $L_1$ .list := append( $L_2$ .list, vnptr);}}

14  $BODY \rightarrow \{ \{ level++ ; \} VAR\_DECL \ STMT\_LIST$ 
   {delete_var_list(active_func_ptr, level); level- -;} \}

15  $STMT\_LIST \rightarrow STMT\_LIST ; STMT \mid STMT$ 

16  $STMT \rightarrow BODY \mid FUNC\_CALL \mid ASG \mid /* others */$ 

```

Y.N. Srikant
Semantic Analysis

If there is a list of variables there is nothing very different we search the var list issue and error if found search the parameter list issue an error if found. And otherwise we simply attain the new name to `L2` dot list and pass it on as `L1` dot list. So, whereas, here we have just passed on `L1` dot list equal to `L2` dot list. So, whatever was previously found here is passed on to this the variable which was already declared is not added to the list again. So, here we add it and send it out to `L1` dot list. So, this `L1` dot list now contains all the names. And `D to TL` now the types information is available in this production.

So, patch var type operates properly and patches all the types of. These particular names body expands to begin block that is the parenthesis flower brackets. And then the var DECL statement list and finally, the parenthesis end you know now the block ending parenthesis. So, what do we do here we really have an action to increment the level, because we are now inside this block. So, the level has increased. So, increase the level then parse these and finally, we delete all the variables the level the current level that is after the incrementation decrement the level and get out. So, this is the action corresponding to body and statement list generates many statements, the statement itself can be any one of these.

(Refer Slide Time: 32:36)

The slide displays the following semantic rules:

- 17 $ASG \rightarrow LHS := E$
{if (LHS.type \neq *errortype* && E.type \neq *errortype*)
if (LHS.type \neq E.type) error('type mismatch of
operands in assignment statement')}
- 18 $LHS \rightarrow id$
{search_var(id.name, active_func_ptr, level, found, vn);
if (~found)
{search_param(id.name, active_func_ptr, found, pn);
if (~found){ error('identifier not declared');
LHS.type := *errortype*}
else LHS.type := pn -> type}
else LHS.type := vn -> type}
- 19 $E \rightarrow LHS$ {E.type := LHS.type}
- 20 $E \rightarrow FUNC_CALL$ {E.type := FUNC_CALL.type}

The slide also features the NPTEL logo and the text 'Y.N. Srikant Semantic Analysis' at the bottom.

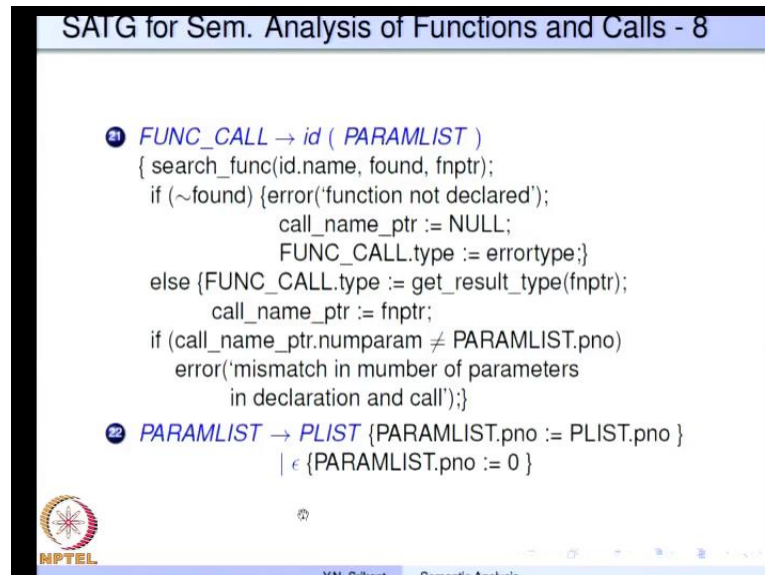
So, now let us see how the functions calls etcetera are processed. So, we have seen how to process the declarations and now we move on to the call this is an assignment statement. So, we have already you know seen this in great detail with coercion and all that possible. So, here we just do the most basic things L H S dot type equal not equal to error type and E dot type not equal to error type. And then if h L s dot type is not equal to E type we have an error type mismatch in operants otherwise if everything is there is nothing to do.

So, here of course, type conversion etcetera needs to be checked as well whether e's type can be converted. So, the L H S type has to be checked L H S expands to i d, because we have not considered arrays in this particular example. So, when we get an i d we search for that name at the current level in the variable list of the current function active func p t r. So, if it is found then you know see this is a, an assignment which has happened in a particular function. So, the unction is still active it has not closed that is why we are searching with this name in with the currently active function.

And if it is not found then we search whether it is a parameter. So, so if the PARAM (()) the parameter then find otherwise if it is a its an error identifier not declared. So, if the name is found as a parameter then L H S dot type is P n pointer type if it was found as a variable then L H S dot type is v n pointer type. So, that takes care of the L H S part which can be checked here again along with E e goes to L H S. There is only a type copy

E dot type equal to L H S dot type and if it is a function call. Then E dot type becomes function call dot type, because function call itself is yet to be expanded it returns a type.

(Refer Slide Time: 35:07)



The slide contains the following code:

```
21 FUNC_CALL → id ( PARAMLIST )
{ search_func(id.name, found, fnptr);
  if (~found) {error('function not declared');
               call_name_ptr := NULL;
               FUNC_CALL.type := errortype;}
  else {FUNC_CALL.type := get_result_type(fnptr);
        call_name_ptr := fnptr;
        if (call_name_ptr.numparam ≠ PARAMLIST.pno)
            error('mismatch in number of parameters
                  in declaration and call');}
```

```
22 PARAMLIST → PLIST {PARAMLIST.pno := PLIST.pno }
               | ε {PARAMLIST.pno := 0 }
```

The slide also features the NPTEL logo in the bottom left corner and the text 'YN_Silvart Semantic Analysis' in the bottom right corner.

So, what is function call? It is a name and a parameter list. So, when we have a name we must check whether it is a function. So, such func searches for this name and if it is found then it returns the pointer in f n p t r I it is not found we issue an error function not declared. And if it is found we assign function call dot type to the appropriate type obtained by executing get result type f n p t r. So, this function goes into the symbol table looks at the result type of the function at f n p t r and returns it as this result. So, call name pointer is f n p t r that is because this is the pointer we have just now obtained by searching. And we also need to check whether the number of parameters in the parameter list is the same as the number o parameters in the declaration.

So, if call name pointer dot num PARAM is not equal to parameter list dot p n o we have not yet seen parameter list. But it you can assume that it returns the number of parameters or number of expressions in this parameter list and p n o. So, if the number obtained from the symbol table is not equal to the number obtained through the parameter list. Then there is a mismatch in the number of parameters in declaration and call and this error message is printed out. What does parameter list expand to? Parameter list expands to PLIST. So, here there is nothing to do really parameter list dot p n o is

PLIST dot p n o or parameter list can also expand to epsilon in which case we there are no parameters then we set it as 0.

(Refer Slide Time: 37:10)

SATG for Sem. Analysis of Functions and Calls - 9

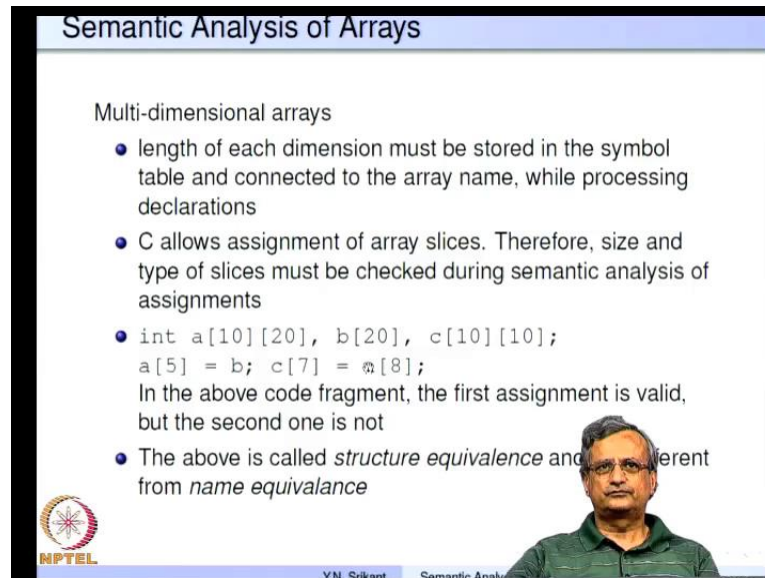
- 23 $PLIST \rightarrow E$ { $PLIST.pno := 1$;
check_param_type(call_name_ptr, 1, E.type, ok);
if ($\sim ok$) error('parameter type mismatch
in declaration and call');}
- 24 $PLIST_1 \rightarrow PLIST_2, E$ { $PLIST_1.pno := PLIST_2.pno + 1$;
check_param_type(call_name_ptr, $PLIST_2.pno + 1$,
E.type, ok);
if ($\sim ok$) error('parameter type mismatch
in declaration and call');}

NPTEL

YN, Siliguri Semantic Analysis

PLIST generates E. So, in this case this is the first parameter. So, PLIST dot P num p n o is set as one then we check the parameter type by using check param type unctioin. So, the function pointer is call name p t r, this is the first parameter and the type of the, you know a parameter is E dot type and is a flag. So, if the types are matching then is true otherwise is returned as false. So, if it is false then the error message parameter type mismatch in declaration and call is issued. So, otherwise everything is in this so; that means, the first parameter has matched with the declaration. Now this is the next production PLIST going to PLIST comma E. So, this says we have checked all these parameters and we need to check this. So, PLIST one dot p n o is PLIST 2 dot p n o plus 1. So, the plus one corresponds to this E now check the parameter type corresponding to this E. So, call name pointer PLIST 2 dot p n o plus 1 E dot type and if it is not then issue the same error message as before. So, this is how we actually process the parameter list in a call.

(Refer Slide Time: 38:41)



Semantic Analysis of Arrays

Multi-dimensional arrays

- length of each dimension must be stored in the symbol table and connected to the array name, while processing declarations
- C allows assignment of array slices. Therefore, size and type of slices must be checked during semantic analysis of assignments
- ```
int a[10][20], b[20], c[10][10];
a[5] = b; c[7] = c[8];
```

  
In the above code fragment, the first assignment is valid, but the second one is not
- The above is called *structure equivalence* and is different from *name equivalence*

MPTEL

Y.N. Srikant

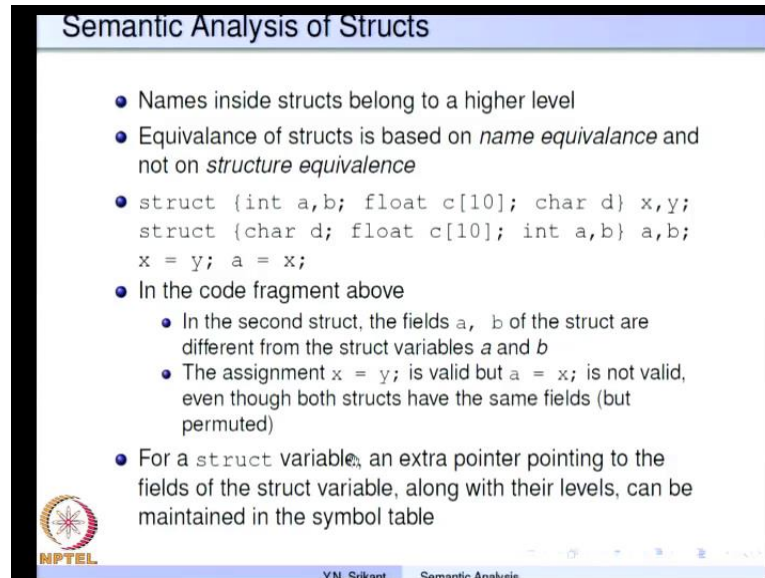
Let us move on and find out how we can process multidimensional arrays. So, as I told you I am going to give you only the basics. And leave the implementation as an exercise. length of each dimension must be stored in the symbol table and connect it to the array name while processing declarations because it is possible to assign array slices c allows it. So, therefore, size and types of slices must check during the semantic analysis of assignments. That is why this particular statement is very important we must store information about every dimension carefully. So, consider this example there is an array declaration a 10 b 20 and another declaration b 20 and third declaration c 10 10.

So, this is a two dimensional array this is single dimensional and this is a two dimensional. So, if you write a five equal to b let us analyze and find out the types. So, a of five is actually a array of size 20, because we have provided a subscript only or the first part. So, a each element of the this dimension is an array of size 20 integers. So, a f size is an array of 20 integers the right hand side b is an array of 20 integers. So, the types match and the first this assignment is definitely valid. So, this is an example of assignment to a slice of an array in the second case c 7 is an array of size ten whereas, a eight is an array of size 20.

So, the types do not match and therefore, this is an erroneous assignment. So, if getting down to the slices and checking the compatibility with respect to the size of the array the type of the elements etcetera is called as a structure equivalence. So, this is basically


structure equivalence, what we have done here is structure equivalence and it is different from name equivalence which will be explained shortly. So, for multidimensional arrays with assignment of array slices, we require structure equivalence.

(Refer Slide Time: 41:05)



**Semantic Analysis of Structs**

- Names inside structs belong to a higher level
- Equivalence of structs is based on *name equivalence* and not on *structure equivalence*
- ```
struct {int a,b; float c[10]; char d} x,y;  
struct {char d; float c[10]; int a,b} a,b;  
x = y; a = x;
```
- In the code fragment above
 - In the second struct, the fields *a*, *b* of the struct are different from the struct variables *a* and *b*
 - The assignment *x = y*; is valid but *a = x*; is not valid, even though both structs have the same fields (but permuted)
- For a struct variable, an extra pointer pointing to the fields of the struct variable, along with their levels, can be maintained in the symbol table

 YN_Srikant Semantic Analysis

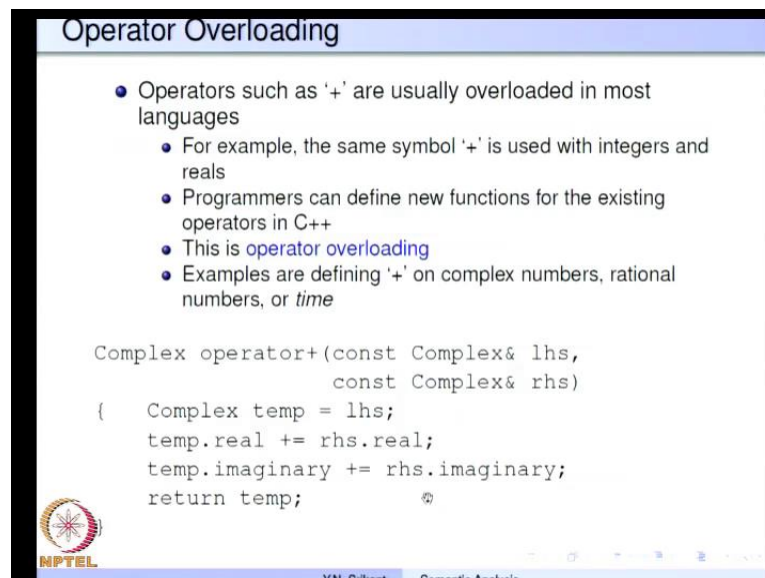
What about structs? The name inside structs belong to a higher level. So, the names inside can do not collide with the names outside, and equivalence of structs is based on what is known as a name equivalence and not on structure equivalence. So, let us take an example here is a structs int a comma b then float c ten char d. So, these are the 3 fields, 4 fields of this struct and x and y are variables of this type. The second struct has char D then float 10 and finally, int a comma b, a and b are two variables of this type. So, observe that this a and b and this a and b are different there is no collision between these two. Now, if you observe carefully the two structs are almost the same, you know this int a b is in the first two fields of the first struct. Whereas, it is in the last two fields of the second struct similarly char D is the last one here and the first one here float is in the middle in both of them.

So, when we write x equal to y we are actually looking at the same structure declaration. So, x and y belong to the same structure declaration and this you know the assignment is correct this is the name equivalence, because we are just looking at the same declaration corresponding to both x and y, if you had provided a name to this struct. Then you know we would really have said x and y are o that particular type name. Whereas, the second

one a equal to x require search to look at a permutation of the second struct. And then compare it with the first which is not really done by any compiler.

Because with large structures any number of permutations large number of permutations will be possible and checking all of them is very time consuming and possibly erroneous as well. So, when we checked the struct assignments we check whether they belong to the same declaration. So, if they are then we say yes they are equivalent otherwise we say they are not equivalent. So, this is basically the name equivalence and for storing the names of the fields, inside a struct an extra pointer pointing to the fields of the struct variable along with their level can be maintained in the symbol table. So, just to avoid and extra pointer and store it that should be good enough.

(Refer Slide Time: 44:00)



The slide is titled "Operator Overloading" and contains the following content:

- Operators such as '+' are usually overloaded in most languages
 - For example, the same symbol '+' is used with integers and reals
 - Programmers can define new functions for the existing operators in C++
 - This is **operator overloading**
 - Examples are defining '+' on complex numbers, rational numbers, or *time*

```
Complex operator+(const Complex& lhs,
                  const Complex& rhs)
{
    Complex temp = lhs;
    temp.real += rhs.real;
    temp.imaginary += rhs.imaginary;
    return temp;
}
```

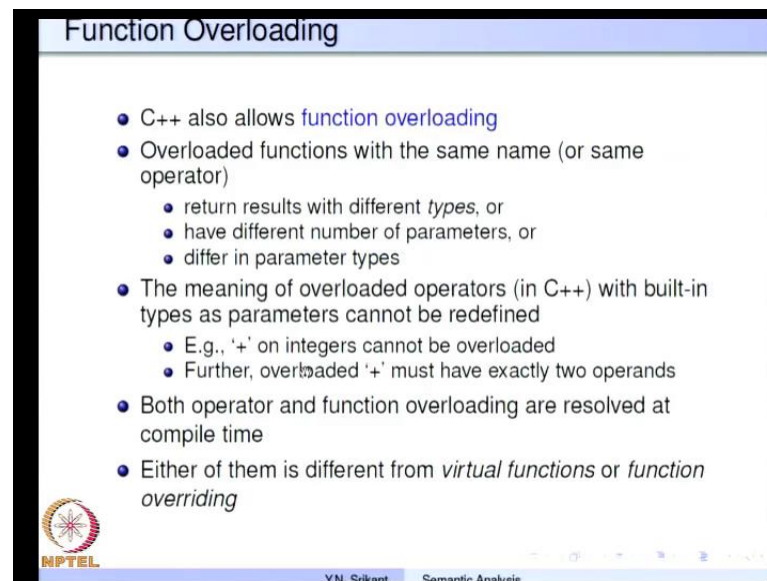
The slide also features the NPTEL logo in the bottom left corner and the text "YN-Srikant Semantic Analysis" in the bottom right corner.

So, let us look at operator overloading this is something know well known. So, operators such as plus are usually overloaded in most languages for example, we use plus with integers and we use plus with reals as well. But then in C plus plus operators such as plus can be given new or we can define new functions or the existing operators in C plus plus in other this is called as operator overloading. For example, we can define plus on complex numbers, we can plus on define plus on rational numbers. Or we can say at time you know two objects which are of time type time, etcetera.

So, here is an example complex operator plus this is the declaration of the overloaded operator plus its parameters are two complex quantities. So, we just want to add them.

So, we have a temporary then temp dot real is temp dot real plus L H S dot R H S dot real. So, we had initialized it to L H S and similarly we had the imaginary parts and return temp. So, whenever we write when there are say two complex numbers x and y. And we simply write x plus y automatically the result is produced to be of type complex and the appropriate addition function is called. So, this is called a operator over loading.

(Refer Slide Time: 45:30)



Function Overloading

- C++ also allows **function overloading**
- Overloaded functions with the same name (or same operator)
 - return results with different *types*, or
 - have different number of parameters, or
 - differ in parameter types
- The meaning of overloaded operators (in C++) with built-in types as parameters cannot be redefined
 - E.g., '+' on integers cannot be overloaded
 - Further, overloaded '+' must have exactly two operands
- Both operator and function overloading are resolved at compile time
- Either of them is different from *virtual functions* or *function overriding*

NPTEL

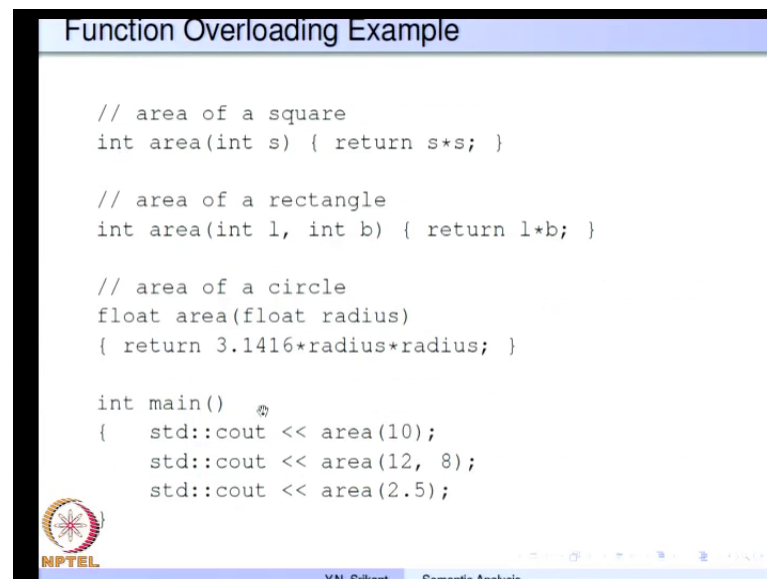
YN. Srikant Semantic Analysis

There is function overloading as well. So, whatever we did with operator we can do with functions as well now we can have the functions with the same name or. And then there must be some difference between these, they that various functions can written results with different types of or they can have different types of parameters or different number of parameters. So, something must be different between the functions which have the same name. So, so that the compiler can differentiate between the various functions. So, the meaning of the overloaded operators in C plus plus with built in types as parameters cannot be redefined. So, this is with respect to operator overloading. So, for example, we already have a plus on integers. So, now, we cannot redefine plus on integers to mean something else and we cannot say that you know plus will now have three parameters.

So, we must always have exactly two parameters for the overloaded operator plus irrespective of what type of parameters it takes. So, both function and operator overloading are resolved at to compile time and both are different from virtual functions or function overloading overriding function overriding really is useful in inheritance. So,

when there is a subclass and we redefine a function which was defined in the base class that is called as function overriding and virtual functions of course, are the methods inside the inherited class. So, at a point in time I could possibly call the function with the subclass or some other base class as well. So, this is these two are very different from the overloading that we are talking about.

(Refer Slide Time: 47:29)



```
Function Overloading Example

// area of a square
int area(int s) { return s*s; }

// area of a rectangle
int area(int l, int b) { return l*b; }

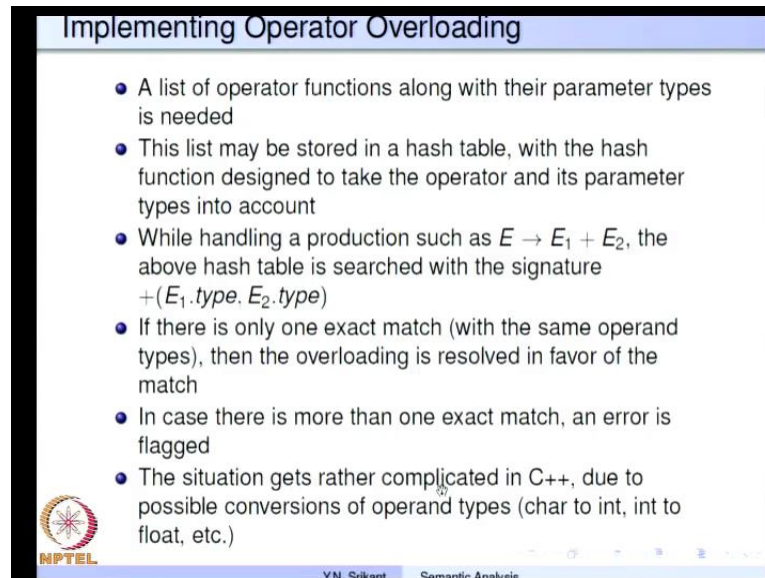
// area of a circle
float area(float radius)
{ return 3.1416*radius*radius; }

int main()
{
    std::cout << area(10);
    std::cout << area(12, 8);
    std::cout << area(2.5);
}

NPTEL
YN Srikant Semantic Analysis
```

So, here is the function overloading example. So, the same name area is used for square rectangle and circle in this case it return integer here also it returns integer r. Whereas, here it returns a float and the number of parameters it is 1 here 2 here and 1 here the types of parameters are also different. So, here it is float and here it is int. So, when we write something like this, this area 10 corresponds to square area 12 comma 8 corresponds to rectangle and area 2.5 corresponds to the circle.

(Refer Slide Time: 48:08)



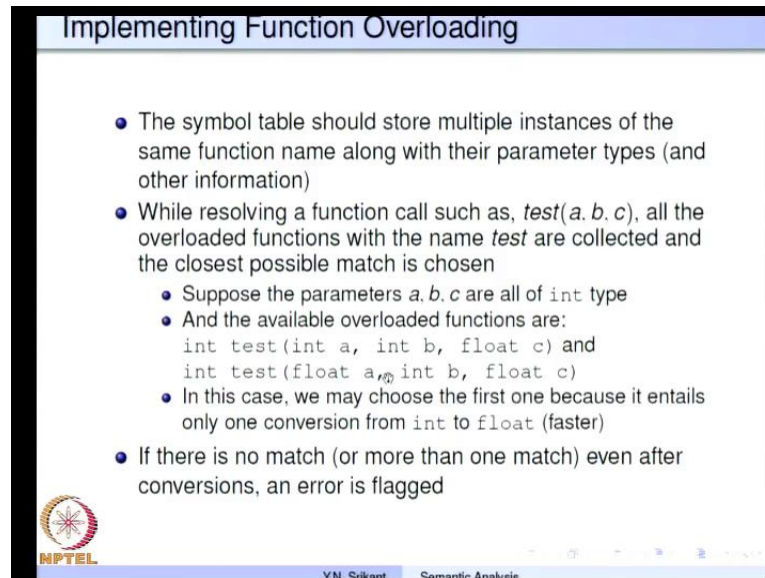
The slide is titled "Implementing Operator Overloading" and contains a bulleted list of six points. The first point states that a list of operator functions with their parameter types is needed. The second point says this list can be stored in a hash table with a hash function that considers the operator and its parameter types. The third point explains that for a production like $E \rightarrow E_1 + E_2$, the hash table is searched with the signature $+(E_1.type, E_2.type)$. The fourth point notes that if there is only one exact match, the overloading is resolved in favor of that match. The fifth point states that if there are more than one exact matches, an error is flagged. The sixth point mentions that the situation is more complicated in C++ due to possible conversions of operand types (e.g., char to int, int to float, etc.). The slide also features the NPTEL logo in the bottom left corner and the text "Y.N. Srikant Semantic Analysis" in the bottom right corner.

- A list of operator functions along with their parameter types is needed
- This list may be stored in a hash table, with the hash function designed to take the operator and its parameter types into account
- While handling a production such as $E \rightarrow E_1 + E_2$, the above hash table is searched with the signature $+(E_1.type, E_2.type)$
- If there is only one exact match (with the same operand types), then the overloading is resolved in favor of the match
- In case there is more than one exact match, an error is flagged
- The situation gets rather complicated in C++, due to possible conversions of operand types (char to int, int to float, etc.)

How is this really implemented? So, for operator overloading we need a list of operator functions along with their parameter types. So, we can store this as a hash table and the hash function must be designed very carefully it must be designed to take the operator and its parameter types into account. So, for example, when we handle the production such as $E \rightarrow E_1 + E_2$ the above hash table is searched with this signature $+(E_1.type, E_2.type)$.

So, in other words we hash using plus then $E_1.type$ and $E_2.type$ look at the entries in the hash table match them appropriately. So, if there is exactly one match with the same operand types then we have resolved the overloading and if there is more than one match an error is flagged. But in practice the situation is a little more complicated in C plus plus. Because conversions of operand types char to int int to float etcetera are also possible before we make a decision as to which function operator is relevant in this point.

(Refer Slide Time: 49:23)



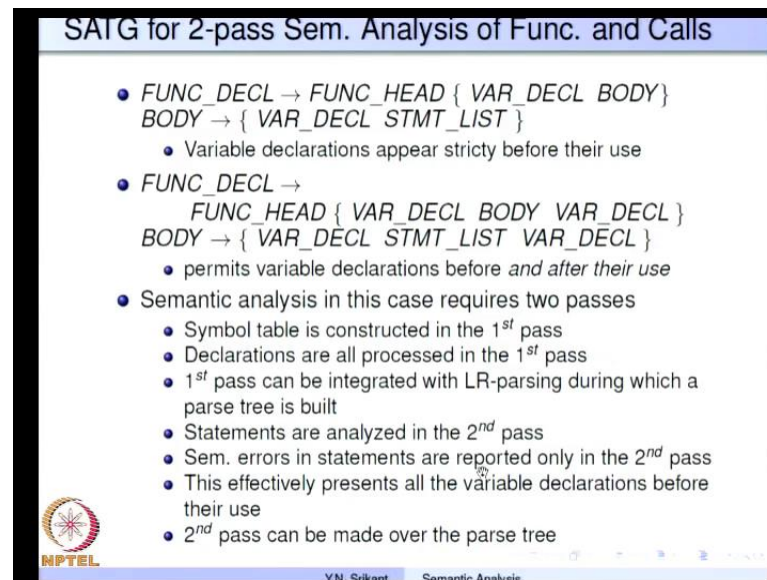
Implementing Function Overloading

- The symbol table should store multiple instances of the same function name along with their parameter types (and other information)
- While resolving a function call such as, *test(a, b, c)*, all the overloaded functions with the name *test* are collected and the closest possible match is chosen
 - Suppose the parameters *a, b, c* are all of *int* type
 - And the available overloaded functions are:
`int test(int a, int b, float c)` and
`int test(float a, int b, float c)`
 - In this case, we may choose the first one because it entails only one conversion from *int* to *float* (faster)
- If there is no match (or more than one match) even after conversions, an error is flagged

MPTel
Y.N. Srikant Semantic Analysis


Function overloading is also not so trivial to implement the symbol table should be able store multiple instances of the same function name along with their parameter types and other information. So, while resolving a function call say *test a b c*, all the overloaded functions with the name *test* are collected. So, and the closest possible match is chosen so, what is the closest possible match? Suppose the parameters *a b c* are all type *int* in this case and we have two of these tests routines, with first one with *int int* and *float*. The second one with *float int* and *float* if the first one is chosen, because we need to convert only you know one of the parameters possibly from *int* to *float* these are all *int*. So, whereas, for the second one we need to convert two of them to *float*. So, converting one is faster than converting both. So, the first one is chosen.

(Refer Slide Time: 50:30)



SATG for 2-pass Sem. Analysis of Func. and Calls

- $FUNC_DECL \rightarrow FUNC_HEAD \{ VAR_DECL \ BODY \}$
 $BODY \rightarrow \{ VAR_DECL \ STMT_LIST \}$
 - Variable declarations appear strictly before their use
- $FUNC_DECL \rightarrow$
 $FUNC_HEAD \{ VAR_DECL \ BODY \ VAR_DECL \}$
 $BODY \rightarrow \{ VAR_DECL \ STMT_LIST \ VAR_DECL \}$
 - permits variable declarations before *and after* their use
- Semantic analysis in this case requires two passes
 - Symbol table is constructed in the 1st pass
 - Declarations are all processed in the 1st pass
 - 1st pass can be integrated with LR-parsing during which a parse tree is built
 - Statements are analyzed in the 2nd pass
 - Sem. errors in statements are reported only in the 2nd pass
 - This effectively presents all the variable declarations before their use
 - 2nd pass can be made over the parse tree

 NPTEL

Y.N. Srikant Semantic Analysis

So, now let me also introduce you to semantic analysis on functions and declarations in a two pass compiler. Suppose we have we permit the variable declarations before the body and after the body instead of just before the body and the body itself has variable declarations then statement list and then variable declarations. So, we would actually be permitting declarations before and after the use. So, in such a case the symbol table must be constructed in the first pass declarations are processed in the first pass. But the statements are analyzed and errors are issued only in the second pass. The second pass of course, can be made over the parse tree whereas; the first pass can be integrated with L R parsing.

(Refer Slide Time: 51:21)

Symbol Table for a 2-pass Semantic Analyzer


block_table (indexed by blk.num)

blk. num	name	result type	param. list ptr	local var. list ptr	num. param	surr. blk. num
1						
2						
3						
4						

Parameter/Variable name record

name	type	parameter or variable tag	level of declaration	blk.num
------	------	---------------------------	----------------------	---------

- The symbol table has to be *persistent*
- Cannot be destroyed after the block/function is processed in pass 1
- Should be stored in a form that can be accessed according to levels in pass 2




YN. Srikant Semantic Analysis

So, let me show you an example here. So, along with the name we need to store you know the here we call it as a block table instead of a symbol table just to distinguish between the various types. And this is the indexed using the block number 1 2 3 4 and we also store what is known as a surround block number along with this extra information.

(Refer Slide Time: 51:47)

Symbol Table for a 2-pass Semantic Analyzer(contd.)

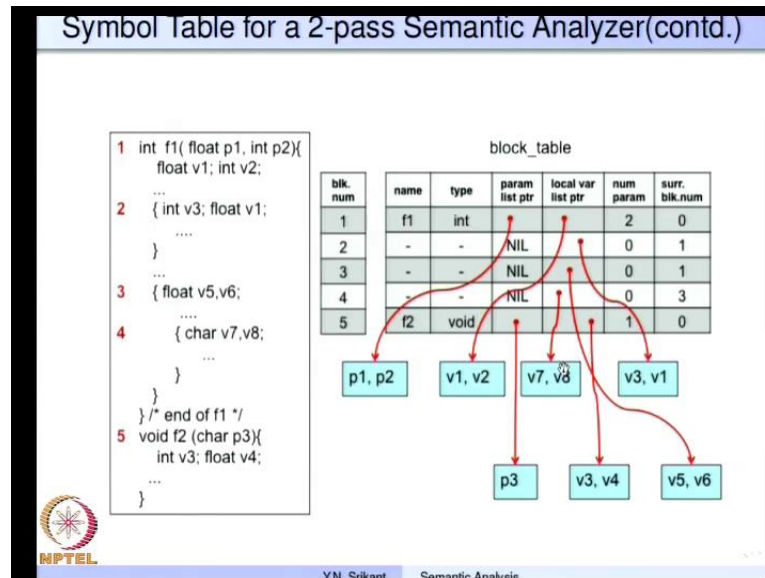
- The symbol table(ST) is indexed by block number
- In the previous version of the ST, there were no separate entries for blocks
- The surround block number (*surr.blk.num*) is the block number of the enclosing block
- All the blocks below a function entry *f* in the ST, upto the next function entry, belong to the function *f*
- To get the name of the parent function for a given block *b*, we go up table using surround block numbers until the surround block number becomes zero



YN. Srikant Semantic Analysis

So, let me show you an example first.

(Refer Slide Time: 51:49)



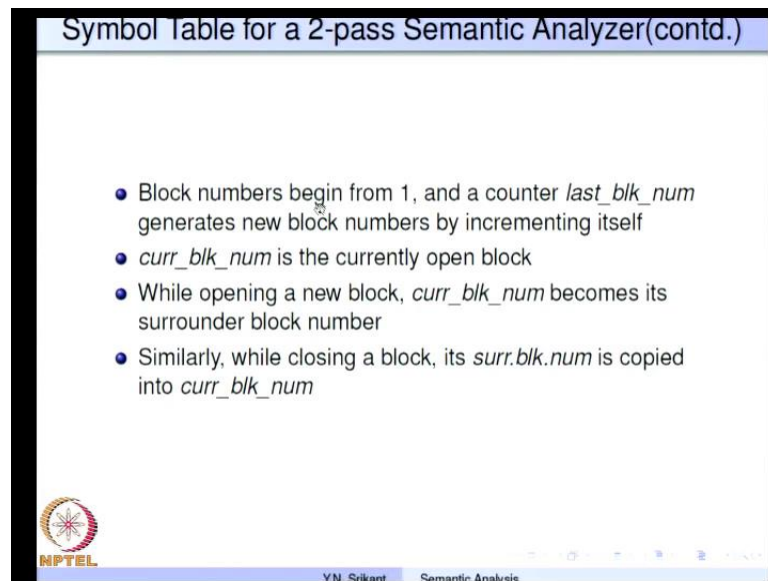
Here is the program in which we have int f 1 then we have the blocks and the, we have blocks inside again. And then the end of f 1 and f 2 the reason we want to show this is you know the variables etcetera. Here will have to be retained within the symbol table and cannot be disposed off. So, in other words we need the symbol table to be persistent it cannot be destroyed after the block or function is processed in pass 1. And it should be stored in a form that can be accessed according to levels in pass 2.

So, here these are the various functions. So, we have a function f 1 and then we entered it this is block number 1 function f 1 is block one whereas, block number 2 is within function 1. So, it has no type or parameters etcetera, but it definitely has a list of variables attached to it block number 3 again has a list of variables attached to it block number 4 is within block 4. And it has a list of variables attached to it, but the, and then function if f 2 corresponds to block number 5. The important factor here is the surrounder block number for example, block 4 is enclosed within nested within block 3, block 3 is nested within block 1.

So, corresponding to that we have surrounder block number as three and for 3, we have the surrounder block number as 1. And for 1 we have the surrounder block number as 0 otherwise the variables are all maintained as a separate list for equal block. So, this is the specialty in the case of the two pass compiler. So, the symbol table is indexed by block numbers and there were no separate entries for blocks in the previous case here there is a

surrounder block corresponding to the enclosing block. So, all the blocks below the function entry f correspond to the function f up to the next function of course. And to get the name of the parent function we just have to use the surrounder block numbers to until it becomes a 0. So, here for example, if you are in block 4, we just you know use the surrounder block number to go to 3 then to 1 and then if becomes a 0. So, f 1 is the function which nests these blocks.

(Refer Slide Time: 54:35)



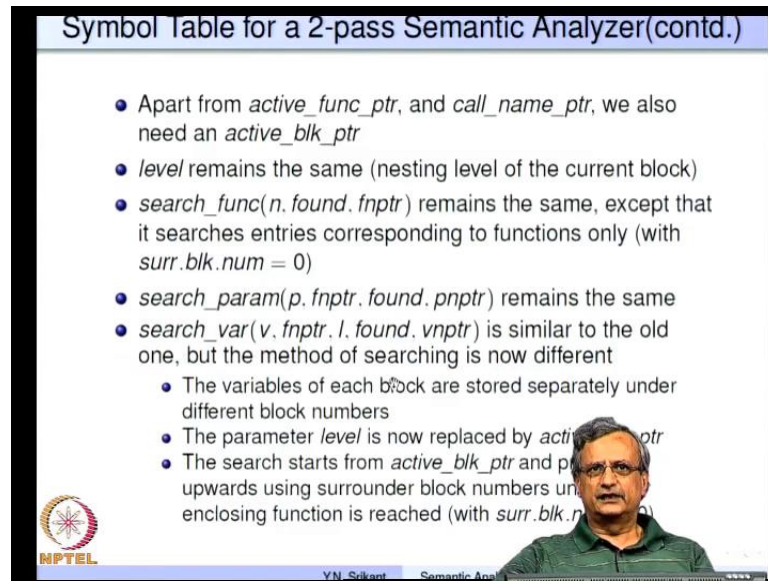
Symbol Table for a 2-pass Semantic Analyzer(contd.)

- Block numbers begin from 1, and a counter *last_blk_num* generates new block numbers by incrementing itself
- *curr_blk_num* is the currently open block
- While opening a new block, *curr_blk_num* becomes its surrounder block number
- Similarly, while closing a block, its *surr.blk.num* is copied into *curr_blk_num*

NPTEL
Y.N. Srikant Semantic Analysis



So, block number begins from 1 and there is a counter last block number which generates new block numbers and current block number is the currently open block. So, while we open a new block current block number becomes the surrounder block number and similarly while closing a surrounder block number is copied into the current block.

(Refer Slide Time: 54:55)



Symbol Table for a 2-pass Semantic Analyzer(contd.)

- Apart from *active_func_ptr*, and *call_name_ptr*, we also need an *active_blk_ptr*
- *level* remains the same (nesting level of the current block)
- *search_func(n, found, fnptr)* remains the same, except that it searches entries corresponding to functions only (with *surr.blk.num = 0*)
- *search_param(p, fnptr, found, pnptr)* remains the same
- *search_var(v, fnptr, l, found, vnptr)* is similar to the old one, but the method of searching is now different
 - The variables of each block are stored separately under different block numbers
 - The parameter *level* is now replaced by *active_blk_ptr*
 - The search starts from *active_blk_ptr* and proceeds upwards using surrounder block numbers until the enclosing function is reached (with *surr.blk.num = 0*)

YN, Srikant Semantic Anal

So apart from the current you know active function pointer we need call name pointer. And of course, we also need the active block pointer itself level remains the same. Search function remains the same except that the entry is corresponding to surrounder block number equal to 0 are the only one search. Search param remains the same search var is also almost the same it is just that we need to now search the blocks separately and instead of level we use the active block pointer.

So, the search starts from active block pointer and proceeds upwards using the surrounder block numbers until the enclosing function is reached. So, in other words here we search from the innermost block till we reach the function as in the previous case it is just that we have all the information about all the blocks stored separately. And therefore, in the second pass of the compiler you know searching these is possible. So, we will stop here. So, this completes the semantic analysis phase of a compiler. And from the next lecture, we will concentrate on intermediate code generation.

Thank you.