

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

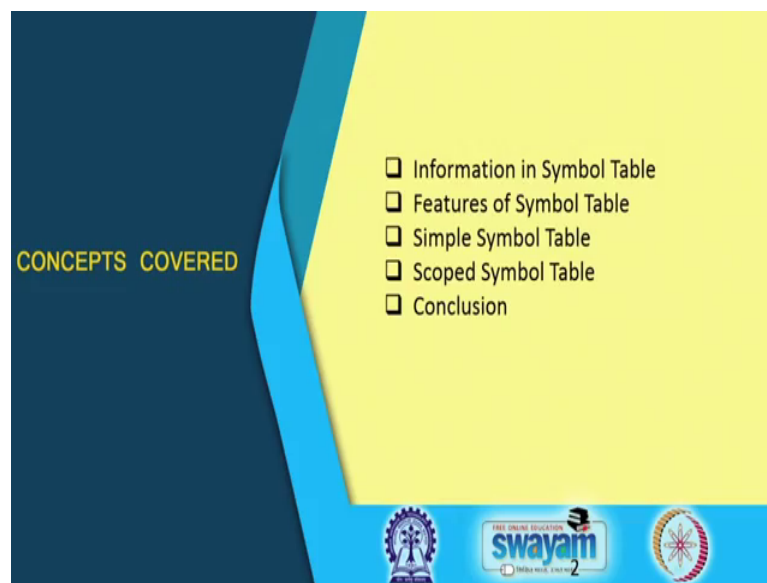
Lecture - 43
Symbol Table

So, next we will start with the Symbol Tables. So, the symbol table is another very important part symbol table construction and symbol table management in the compiler design. So, as we have discussed previously that it is not that symbol tables will become part of the final code that is produced, but it helps in the code generation process. And symbol table is one data structure in any compiler that will be accessed very frequently. So, if we can have a good representation of the symbol table so, that this access to it is fast then this compiler that we have; so, that will also be quite fast.

So, that is one objective, another objective is that so, depending upon the programming language it can define the scope of the variable; that is if when a in the in the program at some point we have defined a variable x. Now, where the; how do in what portion of the program that particular definition is visible. So, a typical example may be like say global variables versus local variables. So, global variables so, they are defined outside the all the functions and they are available throughout the program whereas, if I have got a local variable x defined within a function then it is that particular definition is available only within that particular function.

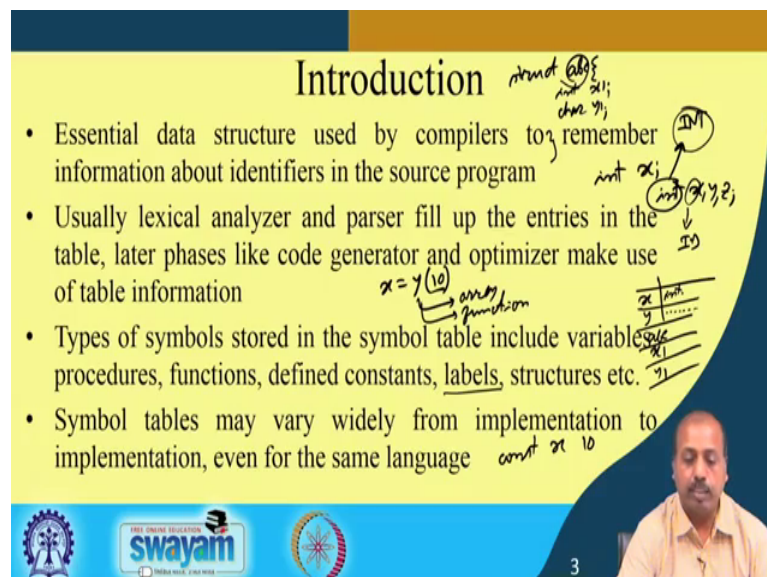
So, that way this symbol table design for the 2 cases will be different like, if we do not keep any differentiation between say global variables and local variables then whenever a definition comes. So, we have to search for the entire symbol table and then we have to decide like whether it is a global definition or a local definition. And, it becomes very combustion to check properly that whether a particular variable has been defined properly in the program or not. So, in this part of the lecture so, we will be looking into how to make that symbol table organization efficient and for our purpose.

(Refer Slide Time: 02:13)



So, we will be doing it like this; first will be looking into the typical information that are stored in the symbol table. Then the features of the symbol table, then we will be looking into some symbol table organization policies like some simple symbol table and some scoped symbol table and finally, we will come to the conclusion.

(Refer Slide Time: 02:33)



So, in it is essential data structure used by compilers to remember information about identifiers in the source program so, this is very important. So, because this identifiers may be typed type names, may be variables, may be some function name. So, like that

we can have different types of identifiers in a program and it may be that in the symbol table will store all those identifiers. So, usually the lexical analyser and parser they fill up the entries in the table and the later phases like code generator and optimizer. So, they will make use of those symbol table information. So, this is typically the situation and we have seen that in lexical analysis phase itself.

So, whenever this lexical analyser comes with a new token if it finds it is an identifier, it installs it in the symbol table and returns the index of that symbol table as an attribute to the token or ID. So, but parser also needs to fill up something because, that type of the symbol and all. So, the parser will know like in which line if it is defined; what I want to mean is something like this. For example, I can have a situation like this. So, let us defined something like say integer x or sometimes I have defined like say integer x y z. Now, what happens is that often getting this keyword integer the lexical analyser will written that type INT ok.

Now getting x so, it has written the type ID along with that in the symbol table it has installed the variable x identifier x, but it does not, but it could not know what is the type because this line the whole line is available to the parser. So, when the parser will be looking into this whole line so, it will understand that it started with the token INT. So, as a result its type should be integer so, that can be filled up. Similarly it is so, y when it comes so, the lexical analyser will put the y into the symbol table and the rest of the fields of that particular row so, they may be filled up by the parser.

So, this way this lexical analyser and parser are so, they will work together for from making the symbol table and later phases like code generation and optimization. So, they will be using this symbol table information because, it is very important to know that type of the individual symbols to allocate proper space for them. So, they will be done using the optimizer code generation or optimization phase.

Types of symbol stored in the symbol table that will include variables, procedures, functions, defined constants, labels, structures etcetera. So, all these can be stored in the symbol table. So, variables we understand that they must be in the symbol table. So, procedures are also needed because many a times if I have got a call to a procedure so, I have to see whether it is proper or not.

Typical situation may be like this in many programming languages what happens is that if I have got say x equal to y 10. So, this y it may be an array or it may be a function so, both are possible. So, if it is an array then this y 10 means I am referring to the 10th entry of the array a y. So, this 10 acts as an index of the array element. On the other end when it is a function so, I am passing this 10 as an as a parameter to the function and then this is a function call. Now how will you understand like whether y is array or function. So, for that purpose we need to refer to the symbol table.

So, looking into the symbol table we understand that either type of y is an array or type of y is a function. So, this procedures functions so, they are also put into the symbol table; so, that we can easily search them out. Some defined constants like we can have apart from these variables I can have something like constant x 10. So, that way these constants are to be defined. So, they are to be they are stored in the symbol table some in sometimes in the programs we have got these labels. So, these labels are also stored then structures are also stored because, structures may have some fields in them.

Like I can have a structure this way that structure a b c and it has got fields like say int x 1 character y 1. So, like that I can have a number of fields. Now this a b c is the name of the structure so, that also needs to be stored in the symbol table and then it has got fields like this x 1 y 1.

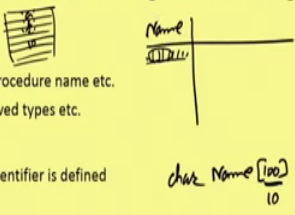
So, this x 1 y 1 they are also going to be stored in the symbol table with a proper attributes identifying that they are the fields of the structure a b c. So, this has to be done. So, this way there are many things that we have so we can store them in the symbol table and symbol tables may vary widely from implementation to implementation even for the same language.

So, symbol table the language the designer will tell nothing about the symbol table. So, language designer will tell about the scope of the variables and this syntax of the language and all. So, it is up to the compiler designer to figure out like what can be a good symbol table organization for the language. So, that is to be done.

(Refer Slide Time: 08:29)

Information in Symbol Table

- Name
 - Name of the identifier
 - May be stored directly or as a pointer to another character string in an associated string table – names can be arbitrarily long
- Type
 - Type of the identifier: variable, label, procedure name etc.
 - For variables, its type: basic types, derived types etc.
- Location
 - Offset within the program where the identifier is defined
- Scope
 - Region of the program where the current definition is valid
- Other attributes: array limits, fields of records, parameters, return values etc.



4

Now, what are the information that we are going; that are we going to store in the symbol table. So, these are the typical information that are stored the name. So, it may be the name of the identifier or may be stored directly or as a pointer to another character string in an associated string table. So, like if it is say if I say that I am storing the name of the symbols in the symbol table structure there is a field called name. Now what is the type of this field? So, this field has to be some character array, but how big. So, if I say it is character name 100; in that case I am giving 100 characters here.

Now, this may be too large or it may be too small because more if the program that we are compiling if the names ID names of the identifiers that are coming are only say 2-3 characters long then keeping 100 entries for each of them is a wastage of space. At the same time if the if the if I put here the value 10 instead of 100; so, if I make the value say 10 that will put a constraint that these identifiers can have at most 10 characters. So, if you look into the programming language it will tell you like how long can be these individual names.

So, if these names are if these names are may can be arbitrarily long then what we need to do is that we need to store somehow these names at some other place. So, in those cases what is done we have got a separate string table. So, this is a string table where it is storing all the strings and this string is not size limited because, this the individual entries are characters like a b c d and whenever the string ends maybe we have got the null

character there. So, that way this so, individual entries so, they are the characters and the last character is the null character of the string. Now so, this index of this string table may be stored as the name of the symbol.

So, that way we can do it so, then we can handle arbitrarily long names ok. So, they can be stored as pointer to the string table, then the type of the symbol. So, type of the identifier like variable, label, procedure name etcetera. So, what is the type that can be stored for variables we also need to store the basic types whether it is a basic type or a derived type etcetera. So, it is some basic type like integer, real, float etcetera or some derived types. So, that has to be stored in the type field; then location. So, offset within the program where the identifier is defined. So, at what distance what offset that particular identifier is defined that location has to be stored then the scope.

So, this scope is telling us like the region of the program where the current definition is valid. So, how long in which portion of the program the definition is valid. So, that will give us the scope and there are maybe other attributes like array limits, fields of records, parameters return values etcetera. So, they also need to be stored and they can be stored in the symbol table. So, all these if this is just a suggestion like these are the typical entries that we have in the symbol table. Now, at a particular design compiler designer may think about some other attributes or maybe think that some of these attributes are redundant.

So, you do not need those attributes. So, it also depends on the language for which you are doing this symbol table. So, so language may or may not support some of the features like say a language that does not support say record. So, in that case so, there is no point in storing the having the capacity of storing records in the symbol table. So, that way this symbol table design is depicted by this is depicted by the programming language and also it is a choice of the compiler designer.

(Refer Slide Time: 12:44)

Usage of Symbol Table Information

- Semantic Analysis – check correct semantic usage of language constructs, e.g. types of identifiers
- Code Generation – Types of variables provide their sizes during code generation
- Error Detection – Undefined variables. Recurrence of error messages can be avoided by marking the variable type as undefined in the symbol table
- Optimization – Two or more temporaries can be merged if their types are same

5

Now, where are we going to use this symbol table. So, we have seen that those are the information to be stored and we have also seen that symbol table is generated by the lexical analyser and the parser tool. Now, where are you going to use this. So, one point is the semantic analysis particularly that type check. So, whether we have when you are going to check types of some expression; say some assignment statements or some or some say operation like $x + y$ we are going to check that, we need to know the types of x and y . So, for that we need to refer to the symbol table to get the types of the identifiers.

Then there are code generation so, we for the variables that we have in the program. So, during code generation we have to proper spaces given. For example, if we have a program in which we have got say integer x character y the variables like that then; so, in the program the code that is generated so, if this is the say that object code. So, in the object code I must have the space is allocated to x so, this x is say 4 bytes. So, first 4 bytes of space they are reserved for x then the character is 1 byte so, this is for y . So, this way it goes so, this individual bytes; so, they are storing, they are stored there and this space is allocated to the individual variables.

So, this is to be done by at the code generation phase. So, type of the variables they can tell us how much space be reserved for those identifiers, then the error detection like they are undefined variable. So, how do you know like if and if your identifier is coming and

then it can see that ok that identifier is undefined. So, that particular variable is undefined. So, this parser or the lexical analyser can detect that situation that this is an undefined variable. Another important point is that suppose in my program I have so, in my program I have used the variable x several times. So, here I am writing x equal to something after some place in some expression I am writing x here.

So, like that suppose I am using x several times, but forgot to define x the statement like say `int x` this I have forgotten to write. Then what will happen at every place so, this compiler. So, this will give an error undefined x undefined x like that and that is a bit combustion and irritating. Because, the programmer will tell why are you why this compiler is giving you this message several times I know that x is undefined. So, why is it giving the same error message again and again. So, one way out for that is that when at this first place when it finds that x is undefined. So, it and find this particular it knows that x is undefined. So, in that case it can install this x into the symbol table.

So, it can install this x into the symbol table with its type as undefined. And, then it can then what will happen successive places. So, this x when it comes so, it will know that x is there in the symbol table and then it will not defy generate this error that x is undefined. So, it is in some sense fooling the compiler itself so, that it later points so, it does not repeat the similar error messages.

So, this type of error detection for undefined for undefined variables. So, this is occurring so, it can be installed. So, that recurrence of error message can be avoided then for the optimization purpose. So, what happens is that in the code generation process apart from the program variables that we have as defined by the user; then there are several temporary variables that are defined that that are generated.

So, particularly if you have got an expression which is quite long like say I have got an expression like x equal to y plus z into w. So, what happens is that so, it is converted into set of simple instructions like t 1 equal to z into w, then t 2 equal to y plus t 1 and then x equal to t 2. So, it is converted into these three statements. So, this t 1 t 2 so, these are the new variables or temporaries that are generated which were not there in the original program. But, you need to store; you need to store this t 1 and t 2 and give them some space also. In normal procedure what will happen like as you are defining the space for x and y. So, you also have to define space for t 1 and the space for t 2.

Now, if you do that so, if you are giving separate, separate space for all these temporaries. So, this 3 1 line code you see there are two temporary variables generated. So, if you have got thousands of lines of code then many many temporaries will get generated. So, whenever there is some expression so, it will generate set of temporaries. Now, all these temporaries they need not be given space simultaneously. So, they are they may be they may not be given separate spaces. So, maybe this t 1 and t 2 sometime later in the some other function. So, we will be again using some other temporaries. So, such that those temporaries do not coincide with this t 1 and t 2 so, they are not required simultaneously.

So, the same space t 1 t 2 can be given to the temporaries t 3 and t 4 also ok. Now, while doing this merging we have to know that these types of t 1 and t 3 are same. So, that the same amount of space can be allocated for them they can share the same amount of space. Similarly this t 2 and t 4 they are also of same type so, that they will share the same amount of space. So, this way we can have this optimization phase; so, that will try to merge two or more temporaries and this merging can be done only if they are of same type.

(Refer Slide Time: 19:06)

Operations on Symbol Table

- ✓ Lookup^{****} – Most frequent, whenever an identifier is seen it is needed to check its type, or create a new entry
- ✓ Insert^{**} – Adding new names to the table, happens mostly in lexical and syntax analysis phases
- ✓ Modify^{*} – When a name is defined, all information may not be available, may be updated later
- ✓ Delete – Not very frequent. Needed sometimes, such as when a procedure body ends

```
procedure P
{
  int x, y;
  ;
}
```

The slide features a yellow background with a blue header and footer. The footer includes the Swamyam logo and the text '6'. A small video inset of a man is visible in the bottom right corner.

Next important thing that we have are the operations on symbol table. So, what are the operations that we are going to do on a symbol table; the most frequent operation is the lookup operation. So, very frequently we need to check whether an identifier is there in

the symbol table, many times we need to get the types and offsets of those symbols. So, identifiers so, like that so, this lookup is the most frequent operation. Then insert is the second important operation because this is addition of new names into the table and happens mostly in the lexical and syntax analysis phases.

So, this lookup is the most frequent operation, insert is the next frequent operation. Then we have got modification sometimes a name is modified and this definite all the definite all information may not be available and then we need to update it later. As I was telling that this lexical analyser it might have put the name of the variable into the symbol table, but not the other information the type of set etcetera.

So, maybe calculated by this parser and accordingly the values maybe put there; so, this modification when it is being done by into the symbol table. So, that will require a modify operation and a delete a operation. So, delete operation is we want to delete the identifier from the symbol table.

So, delete is not very frequent so, this will occur particularly when any procedure ends. So, maybe I have a procedure and in this procedure P. In this procedure P so, we have defined some variables like x y etcetera, then when this procedure ends after that this x and y they do not have any meaning. So, for the outside this procedure this x and y they are not visible. So, we need to delete them from the symbol table to make the symbol table entries released. So, they can be used for some storing some other variables. So, this way we can have this delete operation. So, this lookup, insert, modify and delete.

So, these are the 4 basic operations that we have in the symbol table and out of that this lookup is the most frequent operation; then the next one is this insert and this modify and delete. So, they are then this modify. See if I put say 4 stars they are telling that they are very important then insert is 2 star, modify is 1 star and this is normal operation delete. So, that this delete is very very infrequent. So, they are only when a block ends then only this delete is done and this is often not very costly also. Because, we can know we can just release that space which is held by the symbol table for that portion of the definition. So, these are the operations on the symbol tables.

(Refer Slide Time: 22:11)

Issues in Symbol Table Design

- Format of entries – Various formats from linear array to tree structured table
- Access methodology – Linear search, Binary search, Tree search, Hashing, etc.
- Location of storage – Primary memory, partial storage in secondary memory
- Scope Issues – In block-structured language, a variable defined in upper blocks must be visible to inner blocks, not the other way

The slide features a yellow background with a blue header and footer. A video inset in the bottom right corner shows a man in a light blue shirt speaking. The footer includes logos for 'swayam' and 'INDIA RISE, EDUCATION RISE'.

Now, while handling this so, we have to next we are looking towards having some data structure which can be used for organizing this symbol table. And, then we can have there are several options lived there like when you are talking about the symbol table structure. And, we have to be concerned about the operations that we are going to do. So, based on that we can have the symbol table organized.

So, what are the issues? The first issue is the format of entries. So, there can be different types of formats for the symbol table, it can be a simple linear array or it can be a tree structure table. So, there can be different types of organizations of this symbol table. Then how are you going to access that table; maybe a linear search maybe a binary search tree search hashing etcetera. So, there can be different access methodology based on which this symbol table will be used. Now for example, if you are using a binary search and you have organized your symbol table as a list then of course, it is difficult to search for the entries.

On the other hand if it is organized as a table and you are storing the indices one after the; if you are storing the symbols one after the other then hashing is not a good strategy there. For hashing we need to store the entries at some particular places only in that table. So, this way this access methodology so, it will also tell you like how much is the will be; how much will be the cost of individual operations into the symbol table. Then the location of the storage it may be primary memory or it may be partial storage in the

secondary memory. So, typically this symbol table should be stored in the primary memory because whenever we are trying to access it so, it should be immediately available. Or, it may be that we are just talking about some partials to power a part of the symbol table being stored in the secondary storage.

So, that we can just so, the symbol table is very large so, it cannot be accommodated in the primary storage. So, we will do we will do put a part of it in the secondary storage. Then there are scope rules like in block structured language a variable defined in upper blocks must be visible to inner blocks and not the other way. So, what we mean is that for example, if we have got a C type of language. Now, if there is a block like this where you are defined say integer x and y and within this there is another block where we have got the integers a and b; so, here so, in this region so, you should be able to see both a b and x y.

Because, this x y is defined in the outside block and this x y you will should be visible to the block that it is encapsulating. So, this x y are visible here; however, this a b so, they are defined here and they should not be visible at this point. So, if there is a reference to a b at this point. So, either it is defined in again some higher level block or global variables or otherwise so, this is in this is this is an error ok; this is not available here. So, this is known as the scope rule. So, scope rule says that how much what is the scope of a particular definition in a program. So, how much of the program can see a particular definition. So, they are some variables that are defined in some block should be visible to the inner blocks, but not the other way ok.

So, that is these are the issues with the symbol tool. So, we will see how these are done in different symbol table organization how they are organized.

(Refer Slide Time: 26:08)

Simple Symbol Table

- Works well for languages with a single scope
- Commonly used techniques are
 - Linear table ✓
 - Ordered list ✓
 - Tree ✓
 - Hash table ✓

The diagrams illustrate four techniques for symbol table implementation:

- Linear table:** A vertical array with slots containing 'x', 'y', 'a', 'b', and 'c'.
- Ordered list:** A horizontal linked list with nodes containing 'x', 'y', 'a', 'b', and 'c'.
- Tree:** A B-tree structure with three nodes. The first node contains 'x' and 'y', the second contains 'a' and 'b', and the third contains 'c'.
- Hash table:** A bucket-based structure where 'x' and 'y' are in one bucket, 'a' and 'b' are in another, and 'c' is in a third.

Handwritten notes on the right side of the slide include:

- `int x, y;`
- `int a, b, c;`

Logos for 'swayam' and 'INDIA RISE, EDUCATION RISE' are visible at the bottom left. A small number '8' is in the bottom right corner of the slide.

So, the simple type of symbol table; so, they will work for languages in a single scope. So, single scope means so, everything is visible everywhere ok. So, it is typically the language where wherever you define a variable; so, they will be visible to the entire program. So, there is as in some sense you can say that the all variables are global in this way. So, at any point of time in your in your program you define some variable so, a b c like that. So, it does not matter so, whatever you define so, this will be visible here. This a b c will be visible here then x y will also be visible there. So, it is not dependent on the position of the program at which those variables are defined.

So, they should be visible every and for this type of languages the common structures that we have is the linear table that is a very simple structure; simple array type of structure. Then we have got ordered list where we can have something like say a part of the list is ordered in some fashion maybe the of the symbols that are occurring first. So, they will be kept earlier, then we have got 3 type of structure where this tree list is organized as a tree and there can be hash table. So, in hash table the elements are not put as they are coming. So, in this linear table what will happen is that this variables as they are getting defined; so, x y then a b c.

So, they will be stored one after the other. So, in ordered list so, there will be some ordering. So, how that ordering will come; so, it will be dependent on the language. So, it may be that we have got in a list structure x then y then a then b then c so, like that. So,

we have got an ordered list or it can be a tree type of structure. So, how this tree will be organized so, it may be a binary tree or something like that; so, it will be that way. And, hash table means that this x y a b c for each of them we will compute it index based on some function and then those symbols will be stored at the corresponding place only.

So, this way different types of organizations of this simple symbol table can be taken up. And, then when we are going to complex symbol table so, basically the scope rules they will define this complexity of the structure. And, accordingly they will be organized, but following these basic principles only, using this basic symbol table organization policy only.