

```
document.getElementById('div').innerHTML = "Email Address  
is Invalid";  
  
else if (i==2)  
{  
    var atpos=inputs[i].indexOf("@");  
    var dotpos=inputs[i].lastIndexOf(".");  
    if (atpos<1 || dotpos<atpos+2 ||  
        document.getElementById('errEmail').  
        innerHTML != "")  
    {  
        document.getElementById('errEmail').  
        innerHTML = "Email Address  
is Invalid";  
    }  
    else if (i==5)  
    {  
        document.getElementById('errEmail').  
        innerHTML = "Email Address  
is Invalid";  
    }  
}  
else if (i==6)  
{  
    var atpos=inputs[i].indexOf("@");  
    var dotpos=inputs[i].lastIndexOf(".");  
    if (atpos<1 || dotpos<atpos+2 ||  
        document.getElementById('errEmail').  
        innerHTML != "")  
    {  
        document.getElementById('errEmail').  
        innerHTML = "Email Address  
is Invalid";  
    }  
}
```

## Compiler Design

**Prof. Santanu  
Chattopadhyay  
Computer Science and  
Engineering  
IIT Kharagpur**



# **INDEX**

<b>S. No</b>	<b>Topic</b>	<b>Page No.</b>
	<b><i>Week 1</i></b>	
1	Introduction	1
2	Introduction (Contd.)	16
3	Introduction (Contd.)	31
4	Introduction (Contd.)	47
5	Introduction (Contd.)	63
6	Introduction (Contd.)	78
	<b><i>Week 2</i></b>	
7	Lexical Analysis	87
8	Lexical Analysis (Contd.)	101
9	Lexical Analysis (Contd.)	115
10	Lexical Analysis (Contd.)	130
11	Lexical Analysis (Contd.)	149
	<b><i>Week 3</i></b>	
12	Lexical Analysis (Contd.)	163
13	Lexical Analysis (Contd.)	175
14	Lexical Analysis (Contd.)	187
15	Lexical Analysis (Contd.)	198
16	Parser	208
	<b><i>Week 4</i></b>	
17	Parser (Contd.)	221
18	Parser (Contd.)	233
19	Parser (Contd.)	249
20	Parser (Contd.)	265
21	Parser (Contd.)	281
	<b><i>Week 5</i></b>	
22	Parser (Contd.)	296
23	Parser (Contd.)	308
24	Parser (Contd.)	317
25	Parser (Contd.)	332
26	Parser (Contd.)	345
	<b><i>Week 6</i></b>	
27	Parser (Contd.)	357

28	Parser (Contd.)	370
29	Parser (Contd.)	383
30	Parser (Contd.)	397
31	Parser (Contd.)	410

### ***Week 7***

32	SR Latch and Introduction to Clocked Flip-Flop	421
33	Edge-Triggered Flip-Flop	436
34	Representations of Flip-Flops	446
35	Analysis of Sequential Logic Circuit	455
36	Conversion of Flip-Flops and Flip-Flop Timing Parameters	462

### ***Week 8***

37	Register and Shift Register: PIPO and SISO	470
38	Shift Register: SIPO, PISO and Universal Shift Register	478
39	Application of Shift Register	486
40	Linear Feedback Shift Register	499
41	Serial Addition, Multiplication and Division	513

### ***Week 9***

42	Type Checking(Contd.)	528
43	Symbol Table	541
44	Symbol Table (Contd.)	555
45	Symbol Table (Contd.)	569
46	Symbol Table (Contd.) and Runtime Environment	585

### ***Week 10***

47	Runtime Environment	599
48	Runtime Environment (Contd.)	613
49	Runtime Environment (Contd.)	627
50	Intermediate Code Generation	641
51	Intermediate Code Generation (Contd.)	655

### ***Week 11***

52	Intermediate Code Generation (Contd.)	670
53	Intermediate Code Generation (Contd.)	685
54	Intermediate Code Generation (Contd.)	699
55	Intermediate Code Generation (Contd.)	713
56	Intermediate Code Generation (Contd.)	728

### ***Week 12***

57	Intermediate Code Generation (Contd.)	744
----	---------------------------------------	-----

58	Intermediate Code Generation (Contd.)	757
59	Intermediate Code Generation (Contd.)	773
60	Intermediate Code Generation (Contd.)	781
61	Intermediate Code Generation (Contd.)	789

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

**Lecture – 01**  
**Introduction**

So, welcome to this course on Compiler Design. In this course we will try to see like the different phases in the design of a compiler. But to start with it apparently seems very much difficult that given a language, how do it translate it into another language. But at the end of the course we will see that certain parts of it can be automated and certain parts of it depends on the experience and the expertise of the programmer, to develop the corresponding portions of the tools.

So, there are different parts of this course, while some parts of it have got formal background on automata theory, other part they are guided by few rules. For some of the programming language constructs the rules are well defined. Now, if we come across a new programming language, can we need to you need to redesign those rules ok. So, in this course we will try to have an overview of this entire process.

(Refer Slide Time: 01:24)



So, to start with in this introductory lectures so, we will try to see like what do you mean by a compiler then, compiler applications, phases of a compiler, then challenges that we have in compiler design and compilation process. So, we will see an example by which

we will try to illustrate like, what are the different phases of a compiler and we conclude the lecture by summarizing what you have what we have in the introductory part.

(Refer Slide Time: 01:50)

**Introduction**

- Compilers have become part and parcel of computer systems
- Makes user's computing requirements, specified as a piece of program, understandable to the underlying machine
- Complex transformation
- Large number of options in terms of advances in computer architecture, memory management and newer operating systems

3

So, to start with compilers have become part and parcel of today's computer systems. Now if we look back the early computer system that we developed so, they have a huge computers in terms of the hardware and it required a good amount of space to house that particular computer.

Apart from that, from these electrical and electronic challenges the other part of it was to make it work. Make it work means to make it do some useful competitions. So, for that purpose we needed to write a programs and those programs are to be entered in some machine level language. So, as we know that computers ultimately they understand binary 0's and 1's. So, we need to write the program which is meaningful to the computer in terms of 0's and 1's. So, you can imagine like the amount of difficulty that you will have. So, if you have to write even a 10 line program into (Refer Time: 02:52) convert it into binary 0's and 1's, which are meaning full to the computer.

So, that so early generation computers this was the main problem and this translation process from this program that the program that is understandable by human being to a program understandable by the underlying computer was a big challenge, and it was done by means of manual methods by which we know the codes of each and individual instructions and we do translation by hand.

So, that made it very difficult and slowly the system improved and today we do not even understand that whatever we are writing and in high level language. So, it is not directly understandable by the computer, that is executing it. So, there are a lot of translations that are involved in the process. So, these translations are the essential part that we have in compilers. So, these compilers they make users computation requirements, which is specified by means of program understandable to the underlying machine.

So, user writes a program; that program maybe syntactically correct but is grammatically correct as per the language, and the program is hopefully syntactically correct, that is the meaning of the program is what the user wants to compute. For example, if you are trying to write a program which computes roots of a quadratic equations, then once I write a piece of program in some language maybe in C or maybe in some Pascal or C++, Java whatever it is.

So, there are two types of mistakes that I could have done; one is I have some grammatical mistake that is made for example, in C language you know that every statement it ends with semi colon. So, you may miss some semi colon at some place so, that give some syntactically error. Maybe I am using a variable which is undefined which is not defined so far. So, that may be also an error. But there is a difference between these two types of errors. In the first type when I say that I missed a semi colon. So, that is a grammatical error where when I say that I have not declared a variable. So, the machine does not know what to interpret for this particular variable, the type of whether it is integer real characters so, that is not understandable by the computer.

So, the type of operation that we are trying to do on the variable is not well defined. For example, if I am having two integer variables, I can do addition operation, but if the variables are of type character array. So, I cannot do the edition I can only do string concatenation or say string operations on them. So, there is a basic difference between the two types of errors that we understood that which one is syntactical error another when the meaning of the program is not correct. So, that is the semantic error. So, it is a compiler designer's job to catch both of these types of errors syntactic error and semantic error. So, when the user has specified his requirement. So, then the requirement itself we need to analyze whether the requirement has been specified properly syntactically, and or the meaning of the program is also clear. When everything is

alright, then only the compiler will do a translation. So, it will translate the program from whether high level language to the machine level language and that includes complex transformation.

So, in this course we will see how those transformation are taking place. So this transformation will happen accordingly and with the increase in the complexity of the computer architecture and operating systems. So, the challenge that the compiler designers face so, that is also increasing. For example, like say today if you look into any advanced computer architecture system. So, it has got many interesting feature; one typical example is the CISC versus RISC architectures. So, CISC architectures we have got instructions which are pretty complex and the RISC architecture we have got instructions machine level instruction which are very simple. But the main difference is that, in case of RISC the instructions are of equal size and they take more or less same amount of time for execution whereas, for CISC architecture so, it is a other way. So, it takes the instructions are of variable size and they take different amount of time for their execution.

In fact, while you are trying to do some work so, if I give you instructions which are very simple in nature. So, you may possibly do that operation in a many more compact fashion. Like if I am trying to find the roots of a quadratic equation then, if I have very simple instructions in my hand, I can possibly do that operation without creating much of extra executions. On the other hand if I am having complex instructions so, some of the instructions may not be a very much necessary whole part of the instruction may not be complete necessary for some operations, but still I have to take it.

So, that way the CISC instruction they are going to be difficult to handle as far as the compiler designers are concerned and in fact, this movement from CISC to RISC this happened, because of this compiler designers from the compiler designers perspectives. So, it was suggested that if I have got simpler machine level instructions, it is easier to generate efficient code. So, with that this is done. So, with the advances in computer architecture so these challenges that are faced by the compiler designer so, that is going up.

So, memory management policies like today you know that almost all computers operating systems they are supporting virtual memory. So, virtual memory supported

then so, we have to judge like what are the most relevant operations that we are going to do, and possibly we want to keep those relevant portions in some part of memory, which is not going to be swapped out to the secondary storage. So, that the operational efficiency is high.

So, this way this memory management plays an important role in determining the compilers generated codes performance. On the other hand the operating systems the new operating systems are coming. So, they are also having new and newer features. So, accordingly or the challenges that are faced by the compiler designers so, they are also increasing significantly.

(Refer Slide Time: 09:39)

What is a compiler

- A system software to convert source language program to target language program
- Validates input program to the source language specification – produces error messages / warnings
- Primitive systems did not have compilers, programs written assembly language, handcoded into machine code
- Compiler design started with FORTRAN in 1950s
- Many tools have been developed for compiler design automation

int x;  
x = 105  
x = (int)105

FREE ONLINE EDUCATION  
swayam

So, what is a compiler? So, let us try to understand what is it. So, compiler is a system software, that converts source level language program into target language program. The source language may be some high level language like say C, C++, Java, FORTRAN, Pascal.

Now, how many different languages have been developed so far that is innumerable. So, we cannot just go on giving examples, but we can say that. So, compiler for every language that he designed. So, if he wants that the corresponding program will be executed by the underline computer. So, there must be translated into some machine level code, and that has to be done by the compilers. So, compiler is a system software, that converts source language program to target language program.

So, here the target language means in this particular case we are assuming machine language, but very shortly we will see that it is necessarily machine level language. So, it may be something else also. Second important thing that the compiler does is that it validates input program to the source language specification. So, source language any language is specified by means of its grammar for example, if we look into the English language. So, English language has got its own grammar ok. So, any English sentence that we write. So, you can check whether it is grammatically correct or not by properly analyzing the sentence.

So, similarly when you write a program in for some in some language, so, we can consult the grammar rules of the language and accordingly we can say whether the program is grammatically correct or not. So, if the program is not grammatically correct, then the compiler should produce some error messages. So, that such as for example, semi column is missing then it can say that one semi column is missing at this point.

Sometime we also need to produce warnings. So, warnings are like this that, at some places may be the programmer has not retained some specific declaration or specific transformation that is needed, but it may not be an error. So, if the program will execute but the out outcome of the program may be unpredictable. So, we will take a small example and we will take a small example like for example, in C language program suppose I write say integer x, and then x equal to 10.5. So, if we do this. So, there is a problem because this 10.5. So, this is not an integer number. So, this is a real number and x is an integer as I have defined.

So, you see that when the system will try to do this 10.5 assignment to x. So, what will happen it is very much system dependent, because the 10.5 being a real variable maybe it is given 6 bytes of space or eight bytes of space whereas, integer x being an integer variable, depending upon the system it may be given 4 to 6 bytes. So, what I essentially means is that, the space allocated to an integer variable and the real variable so, they are not same.

So, naturally when you do this type of assignment, the value that will be copied into x is not very clear it is not very much sure .So, if you and it can vary from one computer system to another computer system. So, output produced in one computer may be different from what is produced on the other computer. So, ideally I should write if I

really want that this integer part should be assigned to 10. So, I should write like this; int casting the type casting has to be done and then I should write 10.5. So in this case I am explicit. So if I write like this then the compiler knows that what the user wants is the 10.5 value should be taken as an integer value and then assign to x.

So there are well defined rules by which this transformation will be done. So it will remove the fractional part and it will take the integer part only and assign it to x, but at this point what the compiler will do or what will happen when the program is executed. So, it is not sure. So in this case it will generate a warning. So, this is a warning that there is this incompatible type assignments. So, when you go to the type management or this type checking part of this course. So, we will see this thing in more detail. So, what you essentially mean is that, it is trying to assign some new value to a variable, and the value may not be comfortable.

So, that can give some warning. So, there are other types of warnings also that can come. So, many times when you are writing programs and compiling it so, you must have seen many such warnings and most of the time we say ignore the warnings. But ignore the warnings is not always good because as I have said that the value assignment may not be predictable. So as a result your program may not run correctly. So it is ideally desirable that you remove or take care of the warnings also and take appropriate actions in your program modify the program in such a fashion, that the compiler does not give the warning also.

See if there are errors in the program, then the compiler does not generate the final target code it gives the errors and come out. But if there are warnings, the compilers they generate the target code also, but the target generated code may not be very good say it may be erroneous. Then primitive system as I said the early computer systems that they did not have compilers. So at that time the electronic design itself was so, challenging, that people did not think about how to write programs in high level language. They thought that entering the values through some switches and all that should be good enough to judge the underlying hardware. And program written in assembly language and hand coded into machine code.

So after this machine language we have at the next level of program. So there was a assembly language code and when we discussed some in some courses on microprocessors you might have seen this assembly language programming. So this assembly language programs so they are they are they were hand assembled. So if you look into the processor designer's manual, you will find the corresponding code for each and every instruction and accordingly every instruction ultimately gives rise to some hexadecimal code and those hexadecimal code values are entered through manually, through the some card reader or some other mechanism. So, that the program is entered into the computer system.

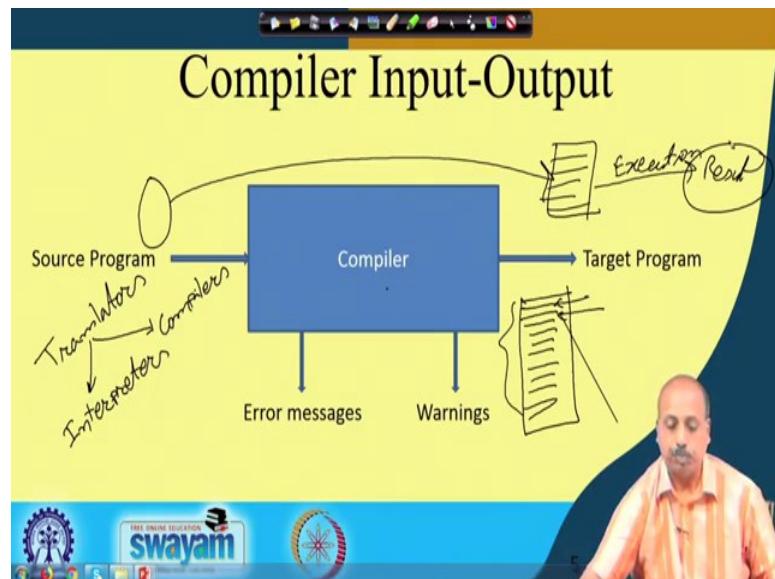
So, it was very common that ok. So, last night you enter your program. So, you typed punched cards a good , say may be a 80 for (Refer Time: 16:30) 89 program. So, there are 80 cards. So, 80 cards you punched on a card punching machine, give it to the system and next day morning you come just to find that, at card number 8 there is a syntax error. So, in and then again you go back again punch the cards correct it, and then again give it. So, that was the problem it was taking huge amount of time today you do not even think about that type of situation. So, as soon as we write the program even a part of it we give it for compilation and see whether there is any error in that part or not. So, that is the beauty of this compilers that can help us in deducting the bugs much earlier.

So, compiler design it started actually with a language FORTRAN in 1950s. So, FORTRAN is the language which is used for which was used for scientific calculations full form is for formula translation. So, and quite sometimes FORTRAN existed and this was also first compiler they were designed for FORTRAN language. And many tools have been developed for compiler design automation. So, as I said that a part of this course. So, they have got foundation on automata theory and accordingly we can come up with some tools automated tools that can generate the compiler given the given the grammar of the language.

So, it can generate the compiler in the sense that it can do syntactic checks ok. So, if there are some grammatical errors in the program. So, it can come up with those errors, but it cannot generate code ok. So, code generation part. So, we have to rely on some other translation mechanism, which are commonly known as syntax directed translation. So, which will be doing this translation part. So, one part of the course is to check whether the program is syntactically correct or not, the second part of the code is to see

how we can generate code for the programs which are syntactically correct. So, we will slowly go into this different parts, next we will see.

(Refer Slide Time: 18:49)



So, pictorially you can think of a compiler like this it us a box. So that the source program is the input and it the source program is taken as input and it the compiler analyses the source program, and it generates the some error messages if there is some error in the programs. So it can generate some error message. If there are some warnings so, it can generate warnings and if there are no errors then it also generates the target program.

So in the maybe in machine code maybe in some other form it also generates the target program. Now at this point of time so I would like to emphasize that there are the another. So in general I can see the compilers they are working as the translators. And these translators there are two types of translators that you can see in a computer system; some of them are known as interpreters interpreter and others are known as compilers ok. The role is more or less same in the sense that both of them translates the source program into machine language program, but interpreter is doing it like this. So if this is a program ok. So, it has got this program lines in it. So, interpreter will take one line at a time. So it will take the first line it will convert it into the machine code and it will execute it.

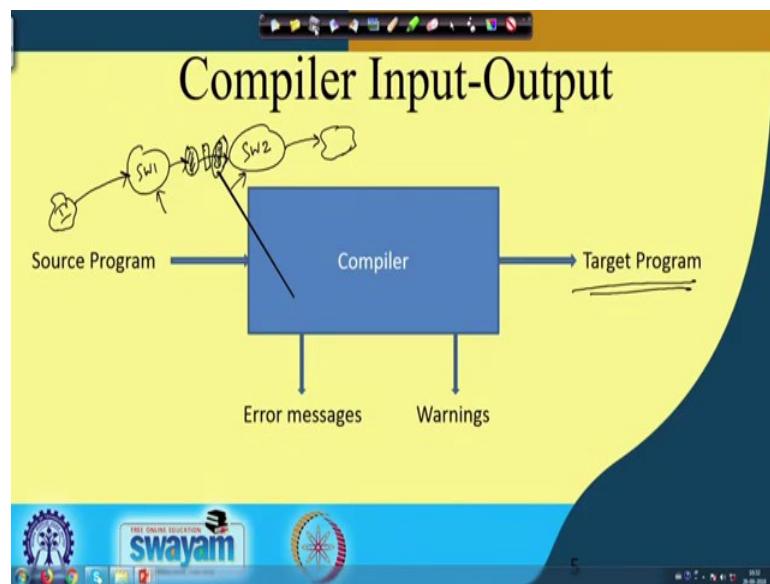
So, if the execution is fine then depending on the execution. So, then it will go to the next statement, it will translate the next statement into machine language and then again it will execute the program. So, it will execute that particular line. So, in case of interpreter as the name suggest. So, it interprets the programming language statements one by the programs statement one by one and executes it. Whereas, for compilers it takes the entire program together and as this diagram is showing this diagram is showing. So, it is generating the corresponding target program. So, given if this is the source language program so, for this it is translated into a machine language program by the compiler. And this entire program can now go into execution. So, it can go into execution and to produce the result it will produce the result. So, that is the works this compiler works.

So, this entire translation is done in one shot. So, this interpreter they are good in the sense that when you are designing the program initially, the interpreters may be good because it will go one line by line. So if there are some error in the program particularly the logical bugs in the program, then it is easy to catch those logical bugs by means of interpreters whatever point there is some problem raised. So we can we can check the variable values and you can try to see like what has done wrong. But at the same time the interpreted execution is also pretty slow, because it is doing one program line at a time so, it is going to be pretty slow unlike the compiled version.

So, it may be advisable that in initial phase of program development, we use the interpreters and at the final phase when the interpreted programs are running correctly, we are happy with its execution. So, we can give it to a compiler. So, that the compiler will generate the target code directly and this now the target code execution will be much faster compared to the interpreter interpretation of the source language program. So, the interpreter they are working at the source language level whereas, this compilers they are also working at the source language level but they are converting entire program into machine code. So as far as execution is concerned it is for the entire piece of code that the execution is being done.

So, this translation so now, we will try to emphasize on this line target program. Now this target program invariably we have assumed so, far that this target program is the machine language program, but it need not be so. So it may be something else also.

(Refer Slide Time: 23:12)



For example suppose I have got two pieces of software. So, suppose I have got two pieces of software: software 1 and software 2. So, some input that the that comes to the software 1. So, this is the input given to software 1 and it is necessary that this input will be should be processed by software 1 and software 2 to produce the final output. So, this software 1 produces some output that goes as input to the software 2 and the software 2 finally, produces the output. So, it can. So, happen like this.

Now, if this software 1 and software 2 are coming from two different vendors. So, it is very much possible that this format in which this software 1 produces the output is not compatible with the input format of software 2. So in that case it is necessary that we put a translator here also. So, that this program is this output of software 1 is converted into another output, which is compatible to software 2 which is understandable by software 2.

So, in this case this target program that we are talking about is this one this is a second program that we are these input or software 2. So, here also this compiler that we have so, it is doing a translation, but it is generating target program which is not for machine execution, but for understanding of another piece of software. So, that can happen. So, that is why this target program when you say. So it need not necessarily be the machine language program it maybe some other format also. So whenever we need any format conversion type of job. So this compilers so they can be utilized.

(Refer Slide Time: 24:59)

How Many Compilers ??

- Impossible to quantify number of compilers designed so far
- Large number of well known, lesser known and possibly unknown computer languages designed so far
- Equally large number of hardware-software platforms
- Efficient compilers must for survival of programming languages
- Inefficient translators developed for LISP made programs run very slowly
- Advances in memory management policies, particularly garbage collection, has paved the way for faster implementation of such translators, rejuvenating such languages

6

So, next we will see. So, what is the next answer very simple question like, how many compilers? Like if we say that how many compilers have been designed so, far can we enumerate and so, it this slide is just to make you understand that how difficult maybe the job of a compiler design ok. So, impossible to quantify number of compilers designed so, far. So, many compilers have been designed. So, nobody can tell like how many compilers have been designed so, far.

Large number of well-known, lesser known and possibly un known computer languages designed so, far. So, after we have gone through this code. So, you will understand that designing a new language is not that difficult. So once you are very clear in your mind that what are the requirements that I need to specify you will see that designing a new language is not difficult and whenever you design a language. So you need to have a compiler to translate it into some other form. So it may be machine code may be something else but we have to do that.

So, as a result what has happened is that, various people they have tried to design new languages and accordingly there are lots of languages that have been designed. So, which are lesser known and possibly unknown computer languages? So, need not known at all and similarly there are large number of hardware software platforms ok. So, hardware software platforms so, every day some new company they are coming up with new features they are adding new features to the operating system. So, if the compiler is

trying to exploit those features, then definitely it has to be it is a new compiler because it is targeting to a new target machine.

So, that way this how many compilers if I want to answer. So it is very difficult to answer in terms of numbers possibly you can say it is infinite. So efficient compilers are must for survival of programming languages, because if the compilers are not efficient then nobody will be using that particular programming language. For example if I design a language say l and I find that whenever I write a programs in language l. So, when it is translated into machine code, the program is much slower than the programs that we have in C language. So in that case the language l will not survive people will not use this language l ok.

So, that way these efficient compilers are very much necessary for these programming languages to survive; inefficient translators developed for LISP made programs run very slowly. So, LISP is a functional programming language. So, that is that was designed long back ok, but in it is it has got very nice features like programs that you write in LISP are very much smaller compared to the high level language programs other high level language program like C and all, but the problem is the translated version of the program. So, they were very slow.

So, if the translators designing efficient translator became a problem was a problem. So, the lisp program they use to run very slowly. So, it was not that much popular. However, with the advancement in memory management policies particularly garbage collection and all so, it has provided the avenue by which you can have a faster implementation of such translator and such languages are rejuvenated.

So, that way many old languages they are also coming back and because of the facilities that are provided in the operating system and the underlying hardware, and accordingly the programs they are they can be the efficient can be generated for the languages. So, this way answering the question how many compilers have been designed so, far is not at all very straight forward, you can only say it is not enumerable.

(Refer Slide Time: 28:55)

## Compiler Applications

- Machine Code Generation
  - Convert source language program to machine understandable one
  - Takes care of semantics of varied constructs of source language
  - Considers limitations and specific features of target machine
  - Automata theory helps in syntactic checks – valid and invalid programs
  - Compilation also generate code for syntactically correct programs

The slide features a blue footer bar with the 'swayam' logo and other educational icons. A video frame in the bottom right corner shows a man with a mustache, wearing a yellow shirt, speaking.

Applications of compilers; so, you have got first application is the machine code generation. So, it converts source language program to machine understandable one. So, that is the first the very first thing, that most compilers they are designed for this machine code generation takes care of semantics of various constructs of the source language. So, this is; obviously, I have to see that the program that is given to be compiled is syntactically correct. So, that is the syntactic semantic constructs and syntactical structures that we have so, that they are satisfied or not.

Then limitations and specific features of target machine. So, it has to see like what are the features like for example, say integers in some machine is allocated say 2 bytes some machines 4 bytes in some machine it is a little Indian architecture some machine it is a big Indian architecture. So, like that there are different architectural features that we have. So, I have to look into those limitations and features of this target machine to see how the compiler should generate the code.

Automata theory that helps in the syntactic check checks .So it can classify the program to be either a valid one or an invalid one, but it does not answer anything beyond that yes and no. But if the answer is no in that case also the compiler has to give enough indication that what exactly went wrong in the program ok. So that is the challenge. So it is not that compiler design the compiler designer designs compiler such that given a wrong program, it just says that the program is syntactically wrong should not be. It

should tell at line number 10 there is this error, at line number 15 there is this error, line number 30 there is this error.

So, you see that we get this type of messages. So, how this is actually done whereas, the basic theory of automata so, that will allow us to classify the programs only into two classes valid program and invalid problem. So, it does not give us hint on where exactly it went wrong.

So, it is the compiler designer's responsibility to see that to see that enough information is given back to the programmer to indicate how much where exactly the program went wrong and what what should be the action. Compilation also generates code for syntactically correct program. So if the code if program is grammatically correct then the compiler should generate the code. So this is the major operations that that are done by a compiler in the machine code generation process.

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

**Lecture – 02**  
**Introduction (Contd.)**

(Refer Slide Time: 00:15)

The slide has a yellow header bar with the title 'Compiler Applications (Contd.)'. Below the title is a bulleted list of applications:

- Format Converters
  - Act as interfaces between two or more software packages
  - Compatibility of input-output formats between tools coming from different vendors
  - Also used to convert heavily used programs written in some older languages (like COBOL) to newer languages (like C/C++)

Below the list is a hand-drawn diagram illustrating tool compatibility. It shows a flow from a source (labeled 'B') through a box labeled 'T1' (with an arrow), then through a circle labeled 'O' (with an arrow), then through another box labeled 'T2' (with an arrow). A question mark is placed above the arrow between 'T1' and 'O', indicating a potential compatibility issue.

The footer of the slide features the 'swayam' logo and other educational icons.

Another application that we have for compare for this compilers are the format conversion. So many times what happens is that we need to, as I was telling that we have got 2 different software packages, and between the software packages the output of one software package should go to the as a go as input of another package. So accordingly we need to give some translation. Because the, there may be the input output formats between the tools may not be compatible, I mean particularly if they are coming from different vendors.

So, in that case so the software output that we have from one tool. So, if this is say tool 1. It takes some input and it produces some output, and then we have got tool 2, but this tool one output I need to feed it to tool 2 but it is not compatible. So, tool 2 it expects input in some format. So, it is very much common particularly like maybe in tool 2 the input format. So, it does not take semicolons at the end of the lines whereas, tool 1 it produces semicolons at the end. So, like that so maybe tool 2 it requires the variables to be defined in some fashion whereas, tool 1 output is in some different format.

So, in those cases what is needed is that in between we put translators. So, this is another compiler, and this translator produces the output which is understandable by tool 2, so and it goes like this. So this way we have some format converters. And this is very much common, particularly if you look into the CAD domain, VLSI, CAD domain. So, you will find the many such tools where we need to do lot of format conversions. So, this is compatibility between input output formats between tools of different vendors.

Now, next thing is that also it is used for converting heavily used programs written in some older languages; like, COBOL to newer languages like C C ++. So COBOL is a language common business oriented language. So, that was used at one point of time; so they were used very much, particularly for these office automation purposes where we used to generate this payroll and all using this COBOL programs.

Now though this COBOL as a language is very much (Refer Time: 02:42), and it is the programs that are that runs in COBOL. So, they are not very much efficient as far as execution speed is concerned, but they are very efficient as far as file handling is concerned. On the other hand, today after that this language is like C and C ++ came, so now the new software they are be that are being developed.

So, they are not they are not being done in COBOL, buts in some next level programming languages or even in 4 generation language. But that point is there large number of programs which were existing. So, this payroll software that we had so, that is that was existing in some old language likes COBOL. So we need to convert we do not want to make though all those programs junk, rather we try to have some converter so that we can automatically generate the C C ++ programs from those old COBOL programs. So that way we have got, we have we have to have this programs which are written in some older languages translated into newer languages.

So, that is the role of format converter, so compare, so here also we have got the compilers in picture.

(Refer Slide Time: 03:57)

Compiler Applications (Contd.)

- Silicon Compilation
  - Automatically synthesize a circuit from its behavioural description in languages like VHDL, Verilog etc.
  - Complexity of circuits increasing with reduced time-to-market
  - Optimization criteria for silicon compilation are area, power, delay, etc.

Next we have got another application which is known as silicon compilation. So, silicon compilation is like this; that today suppose we are trying to design some new integrated circuit chip. For example suppose we want to generate a new processor. Now if we want to design a processor, now it if there are there can be 2 different avenues by which we do it. In one avenue, we start with the lowest level modules like, the lowest level digital circuits like adder, subtractor , multiplier, register design, then slowly go up and go up to the full system design.

So, but the problem in this particular approach is that it is time consuming, and if the user is not very much careful. So there is a possibility that some bug will get introduced into the process, and then the chip that is fabricated is not correct. On the other hand, so there can be another avenue by which we can handle this problem is start with the behavioral description of the system in some language hardware description language which is like VHDL, Verilog. So, these are popular hardware description language in which you can describe the hardware in different levels; so, behavioral level, structural level, functional level likes that.

So, at the top level we have got the behavioral level description. So, you can describe the behavior of the processor in terms of this language constructs. And then from they are we try to generate the circuit. Now you see here the input to the system is a description in some VHDL or a Verilog language. And then the output that we have, so if this is the

silicon compiler, if this is the silicon compiler, so here you have got this VHDL code as input and as an output, you do not have machine code, but rather you have rather you have got some other you have got another VHDL code which is basically a structural level description which we call netlist, which we call netlist.

So, this is the netlist of this library components that we have.

(Refer Slide Time: 06:21)

Compiler Applications (Contd.)

- Silicon Compilation
  - Automatically synthesize a circuit from its behavioural description in languages like VHDL, Verilog etc.
  - Complexity of circuits increasing with reduced time-to-market
  - Optimization criteria for silicon compilation are area, power, delay, etc.

So, that netlist sorry so, net that netlist that we are talking about. So, this netlist is nothing but it is a collection of hardware modules that will be making that will be making this whole system. For example, in your system you will need adder, subtractor, registers etcetera. And some connection pattern between them. So, this netlist will be a connection it will having all those module descriptions and it will it will have the connections between them. So, this thing is automatically synthesized. So, user does not design the individual adders. So, they are designed previously and available in the library and it is taken from there. So, that way it is done. So for very complex systems, so it is so this is the way that we should proceed, we should know it should start with the behavioral description. And then convert it into the netlist it depending upon some library modules.

So, this is, this is known as the silicon compilation process, and this is also some sort of compiler. So, complexity of circuits increasing with reduced time to market. So, thus so this is the pressure that we have. So, 2 motive every day we are coming up with newer

and newer systems electronic systems that have got more and more functionality into it, and the and that there is a very high pressure on time to market. Like, if a if you if we survey the market and see that we want to introduce a new cell phone into the market, then maybe if it is introduced with the next 6 months there will be some benefit some profit, but if it is introduced after one year maybe that profit will be half, and if we introduce after say one and half year maybe the profit will become 0.

So, that way there is a very high pressure on this really on reducing this time to market. So, here if you are starting at the lowest level and trying to generate the whole system very efficiently so it will take time. So, we go in the other way, so we start with the high level description behavioral description of the system and from there try to come to the circuit. And the optimization criteria they are also different. Like when you are writing programs for execution by a processor, then the optimization criteria there is the speed of execution, then you have got this what is the memory requirement. So, these are actually the optimization criteria.

But when you are doing silicon compilation, so the optimization criteria is the overall area required by the circuit, the power consumption of the circuit the delay that the circuit will have. So, we want to reduce this area power and delay. So, the measures that we have are totally different. So, a compiler which is targeted towards say machine code generation we will not do well, when if you put it onto a silicon composition process. So, the challenges are totally different.

(Refer Slide Time: 09:25)

## Compiler Applications (Contd.)

- Query Optimization
  - In the domain of database query processing
  - Optimize search time
  - More than one evaluation sequence for each query
  - Cost depends upon relative sizes of tables, availability of indexes
  - Generate proper sequence of operations suitable for fastest query processing

10

Next, the other application that we have is in the domain of database query processing which is known as query optimization.

So, in database you know that a major operation that we have is to answer the user queries. And once the database has been created, its content is more or less static. For example, if you look into the database of an organization in the employee database of an organization. So, initially there will be additions and all, but after sometime after the database has been stabilized. So, it is not that everyday's a lots of employees are joining the companies and leaving the company. So, that way that database is more or less stabilized or more or less static in nature. And the queries that you get on the database are like this maybe it is trying to find out the average salary, or it every month we have to do salary processing and all that.

So, the queries are much higher in terms of some information taking getting some information from the database, but the basic data value changes are much less, ok. So, it is very much important that whenever we have got this query. So, these queries are to be processed, and they are to be answered quite fast. Now how to reduce this search time? So, if we look into the database theory you will find that there are depending upon the size of the tables that we handle.

So, we can hand we can manipulate the tables in certain order to reduce the overall time requirement. So, given a query what is the exact sequence of operations that we should

do on the database for finding the answers quickly? So that is a big challenge, and that is known as the process of query optimization. So that is also some sort of compiler. Because you see, that the given the query in some high level language like say sql and all, so converting it into some file operations. So ultimately you are it is it is doing some file operations over the ultimately the operating system file routines are to be called.

So, how this translation will be done? So that is a big question and we have got these compilers to help us. As it is noted here that we want to optimize the search time; so, more than one evaluation sequence may be there for each query, and the cost depends on upon the relative sizes of tables availability of indexes etcetera. So the same query if the table sizes are different. So it may be if you process in different sequence of operations, then the timing requirement will be different. So we definitely do something. So, that a particular ordering has to be found. So the size of the tables will dictate like other with the sequence in which the query should be evaluated. So generate proper sequence of operations suitable for the fastest query processing, so that is very important.

So, here the optimization is not in terms of. So the space requirement or that execution time, but it is in terms of the retrieval from the database. Like if you know that these are these are file operations. So, whenever you have got file operations involved. So that file operation take very high amount of time compared to the processing. Like once the ones are an employee salary record has been retrieved; so, maybe we just want to increase the basic pay by some amount.

So that operation is just an addition operation or at most another multiplication operation. But the major time is spent in getting the record from the secondary storage. So, the optimization that we have is in terms of the number of disk accesses and all. So, the optimization criteria is totally different when we are going for going for database operation, compared to the situation that we have for say, this may code like say execution of normal programs, ok.

(Refer Slide Time: 13:15)

Compiler Applications (Contd.)

- Text Formatting
  - Accepts an ordinary text file as input having formatting commands embedded
  - Generates formatted text
  - Example *troff*, *nroff*, *LaTeX* etc.

Another very interesting application that we have for compilers is in the text formatting. So, there are many tools by which you can format the text files. Like, today you know that there are many such formatting tools. For example, we have some of these thumbs, so these formatting tools that we have. So, they can be classified into 2 categories, sorry. So, this formatting tools that we have so this text formatting some of these tools. So, they are what we are doing? So, we are providing some we have provided some sort of graphical interface, and in that graphical interface the font and everything is designed, and we just choose the appropriate font and enter the text in that format. So, immediately you get a feedback, like you are writing in such and such font and font size, and also you immediately get a and get an understanding like how is it looking like.

So, you can you can choose some other format, you can make some part bold and all by doing it immediately on to the text. In some other format, so it is like this that if you just write a file, and in that file you put your normal text, and before this takes you put some and also embed some formatting command in it, some formatting command you input into it in the same file.

So, in this case what will happen is that, so now, this whole thing is given to a compiler and the compiler; so it applies these formatting commands on this text, and then it will be converted into a formatted text. So, here in this in case of graphical thing what is what was happening is, it was that you are just you are immediately looking getting this

formatted text and as a visual output, so you are just looking into that. But in case of this one; so this file that we have is nothing but a simple text file. And this simple text file is converted by means of a compiler into a formatted text file where these formatting commands are taken care of. Like this formatting command maybe it will tell that the next few lines, I want to make it alex. So, accordingly it will do that.

So, it has got many advantages, because this they have the portability of this file is very easy. So, because this is a simple text file so, you can very easily transfer and all, and you can, so after some after some practice and experience. So, you will you may find that these formatting tools are much better compared to this graphics based tools, ok. So, this text formatting tools, so they come under this compiler. So, this is also some sort of compiler, because you are accepting input in some text file format and then as an output you are producing this graphical thing, so that is also a text formatting. So, this text formatting tools. So, they accept ordinary text file as input, having formatting commands embedded into it and it generates the formatted text.

And. So, these are some of the example like, so in Unix operating system you have got troff, troff, nroff and there is another very well-known package which is known as LaTex, ok. So, the where which are used for doing this type of translation so; they come under the broad heading of formatting tools. So, that is also a type of compilers.

(Refer Slide Time: 16:51)

The slide has a yellow header bar with various icons. The main title is 'Phases of a Compiler'. Handwritten notes include:

- 'cat' with arrows pointing to 'shred' and 'shred -'
- A mathematical expression:  $\text{English} \equiv \text{Alphabet} = \{A, B, \dots, Z, a, b, \dots, z\}$  with a circled 'P' below it.
- 'Does not exist any hard demarcation between the modules.'
- 'Work in hand-in-hand interspersed manner'
- 'Sentence' with an arrow pointing to 'collection of word'

- Conceptually divided into a number of phases
  - Lexical Analysis
  - Syntax Analysis
  - Semantic Analysis
  - Intermediate Code Generation
  - Target Code Generation
  - Code Optimization
  - Symbol Table Management
  - Error Handling and Recovery

At the bottom, there is a logo for 'swayam' and other educational icons.

Next we will look into the phases of a compiler. So, you see that if I have got a big job to do like this called translating from one language to another language, maybe high level say C language to target machine code, or say some a VHDL to this silicon output, or say this formatting tools text formatting tools, or whatever for whatever the application we think about.

So, this transformation process is quite complex. So, ideally I can have I have a monolithic piece of software; which is doing this translation the compiler appears to be monolithic, but in the design phase just to make this compiler design process simpler. So, we can we can think about it to be divided into a number of phases, though practically speaking, so there is no hard demarcation ok. So, these modules are they are they are not they are not demarcable clearly. But for our understanding for our discussion in the course, so we will be dividing them into number of phases ok. So, the first phase is known as the lexical analysis phase. So, to start with, so before going into this, so let us try to understand; like, if I have got some language, for example, if I have got the language English ok.

So, any language, so what we have immediately is the first thing that we have is the alphabet. Any language starts with the alphabet. Now you know that in English language the alphabet set is say this capital A, B, etcetera up to capital Z, then this small a, small b, small c, etcetera up to small z. Now there are many more symbols like this, 0, 1, to 9. Then whatever special symbols that we are allowing in say today's English language text. So, they all come under this alphabet set. So, for any language it starts with the alphabet ok. So, we have we have we are having the alphabet. Now once I know the alphabet set. So, this is commonly represented by the symbol sigma ok. And then this alphabet some of these symbols in the alphabet they are combined to form words.

So, this alphabet is from the alphabet we get words. Word is nothing but collection of alphabets, and it is separated by some special say separator for example, in most commonly we are using the separator blank ok. For English language sentence, so this word so word, so any collection of alphabet you take ok. So, that is a word now this word may be a valid word may be an invalid word. So, word you can classify into 2 categories one is the valid word and another is the invalid word. For example, for English language c a t cat, so this is a valid word as far as we know. But say c t a this

possibly not a valid word. So, I am not very much knowledgeable in English language that way.

So, I cannot say very clearly that whether cta is a valid word of English or not, but assuming that it is not a valid word. So, it is it goes into the invalid category. So, that is the second thing that we have. So, alphabet said, so if there is a, so if my if my English language is does not allowing the Greek symbols like alpha beta etcetera, then if I get the symbol beta somewhere in the text, then I will say this beta is a is not in the alphabet set. So, there is a problem with the alphabet, but even if the alphabet part is correct, this cta is not a valid word. So, that is the another level of invalidity. The next level of invalidity that comes like when we consider the sentences; when we consider a sentence now sentence, so it is a collection of word collection of word.

Now, if we consult the grammar of this English language grammar of the English language. So, there are certain rules which will tell us like what are the valid sentences what are invalid sentences and all. So, it is a very huge process or I should say very complex process to say whether a language whether a statement that I have made in English language whether it is grammatically correct or not. But anyway, so if we take some simpler language maybe the grammar rules will be simple, and it will be very it will be easy to tell given a sentence whether it is whether it is valid for the language or not. So, accordingly you can say that this language this sentence it can also be valid sentence or it can be invalid sentence. Now this if a sentence is valid sentence, we say that if the sentence belongs to the language. If the sentence is an invalid one, we say that it is not belonging to the language.

So, that way we can have this foreign language starts with alphabet, then goes to words, then goes to sentence, at each level we have got the issue of validity and invalidity. So, in the lexical analysis phase so, what we will try to do is to see whether we have got some valid word or not. So, alphabet we can check immediately. So, whether there is any foreign character that are present in the in the description, and if there is there is no if there is no foreign character.

So, it will try to see like what are the words of the language that are appearing in the disk in the program. So, accordingly it can determine the words. So, it can it will do the processing and it will find out the words. So, that is the job of lexical analysis. Once this

lexical analysis is done, we have got the phase of syntax analysis. So, in the syntax analysis phase we do the grammatical check, then we have got semantic analysis, where we try to find out whether there is any problem with the meaning like is maybe some variable is undefined and all that.

So, that is a semantic analysis phase. Then we have got some intermediate code generation phase so, up to this semantic analysis phase. So, you can or the syntax analysis phase, you can say, so this is based on automata theory. After that whatever we will have, so they are more they will be using the cost of the outcomes of this automata theory based analysis, but they will be augmented significantly by means of other techniques so that we can do the code generation part. The first phases of that is intermediate code generation. Then that intermediate code is translated into target code, so that is the target code generation. Then after the code has been generated, so we can stop there, but most of the compilers they will go into a code optimization phase.

So, since this is the entire process is an automated process. So, there is a very high chance that the code that is generated by this code generation process is not optimized significantly. So, it will be optimizing it, and then maybe we will be getting a faster code for that. So, that is the code optimization phase. So, apart from that, there are some other points which are also important for compiler design. One is the symbol table management. So, symbol table is a table of all the symbols that appear in the program. All the variables, procedures labels, that appear in the program. So, they come under the symbol table part. So, the symbol table has to be managed, because at many times the compiler module, so they will refer to this symbol table for the checking and code generation process, so they will do that.

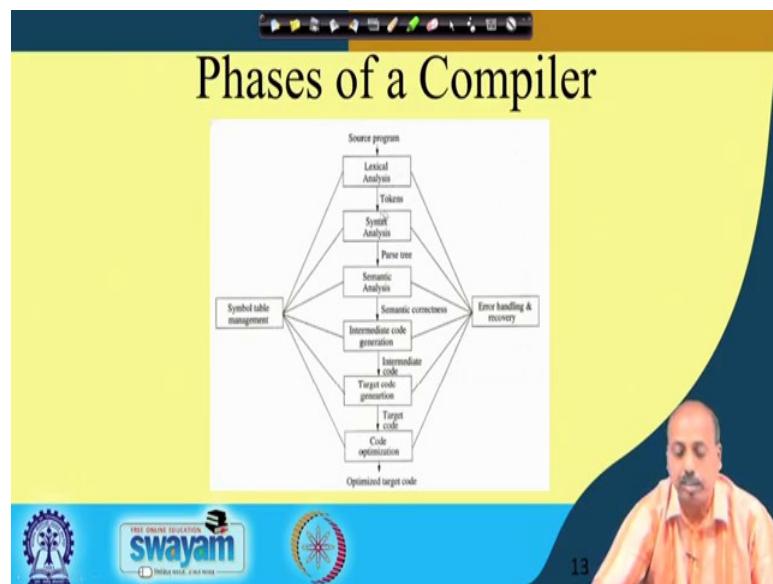
And we have got another very important phase which is known as error handling and recovery. So, whenever there is some error in the source language program, then this compiler has to detect it, and after detecting it has to proceed like, if there is an error at line number 10, then it is not advisable that the compiler just prints that there is an error in line number 10 and comes out. Rather it should check the remaining part of the program also, and come up with a list of all those lines. So, where the compiler thinks that there is an error maybe there are errors at line 10, 15, 18, 19, 20 like that. So, all those lines as many as possible so, if we can list it, then in the next go

the programmer may correct all those errors and then come up with the corrected version of the program.

So, that way the program works life will be simpler. Otherwise the programmer corrects line number 10 give you gives it for compilation and finds that again the compiler has given an error, that there is an error in guideline number 15, so that is not very much advisable. So, this error handling and recovery, so this is important so, the recovery part will be clear when you go into the details of this error analysis job ok. So, it is not very much understandable now. So, there is a caution, of course, that you should not think that all these modules are totally dealing from each other. It is not that the output of one module is given entirely the second module then the second module starts, so it is not like that. So, there is not much demarcation between the modules, and they work hand in hand interspersed when.

So, it is like this that whenever syntax analysis phase, it will find it may find that it needs some more words next word from the system from the from the program. So, it will ask the lexical analysis phase to give it that next word. Similarly, when the code generation phase, so it will try to generate the code, it may ask the syntax energy analysis phase to generate the to give it to tell it a what is the next rule by which I should proceed what is the next grammar rule by which I should proceed. So, that way all these modules they go hand in hand. The later part like code optimization and also they are a bit independent, but up to this intermediate code generation phase. So, I should say that they are all interlinked with each other.

(Refer Slide Time: 27:27)



So, next we have this diagram actually depicts the whole thing. Like, your source language program it enters into the lexical analysis phase. Now, lexical analysis phase, so it does many things. Like, there are normally the pro source language program they have got a lot of comments. So, the comments are early move, because comments are never translated into machine code. So, they are removed from the program. Then between 2 words somebody may put one blank somebody may put 10 blanks like that.

So, all those extraneous blank spaces they will be removed. And many times we have got some header files, we include some header files, and we say that the compiler will for example, in C language we have got the hash include directive, so they are all expanded ok. So, this is all this expansion will be done by the lexical analysis phase. So, ultimately this lexical analysis phase, so it will generate some tokens. So, these tokens will be used by syntax analysis phase. Syntax analysis phase generates parts tree which is used for semantic analysis. That is, the program is semantically correct, then it goes into the intermediate code generation phase. Then from there the intermediate code is translated into target code, goes to target code generation phase. And from the target code it goes into optimization phase.

And the symbol table management it is used by all the modules, from lexical analysis to code optimization. Some of the phases they will put entries into the symbol table, some of the phases. So, they will use the values that are present in the symbol table. And there

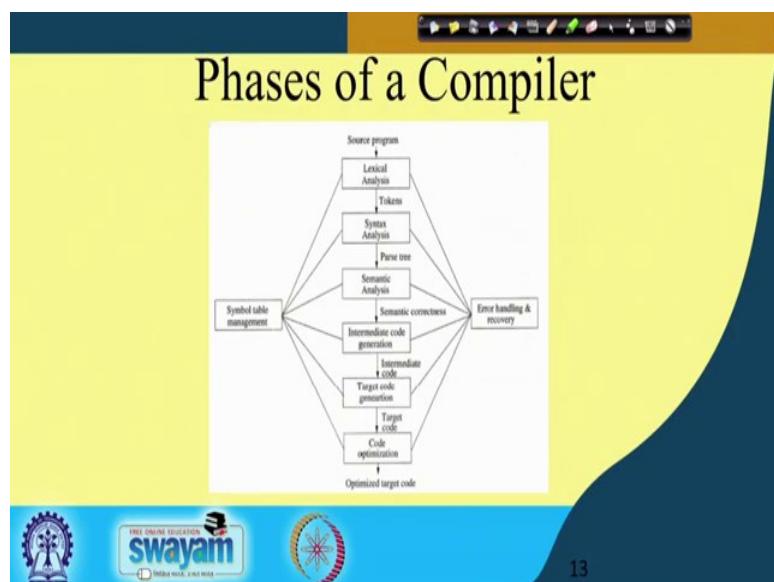
are error handling and recovery routines. So, which will be which are integrated with all these phases again, and if there is some error that has occurred? So, it will be trying to get trying to get some information from different phases and flash appropriate error message to the user so that the user can correct the program in a better fashion. So, these are the different phases. So, by which the compiler generates that I optimized target code from the source language program.

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

**Lecture – 03**  
**Introduction (Contd.)**

So, we are discussing on phases of a compiler so in that we can see that it starts with the source program and that source program goes through a series of transformations through various stages, and these stages they work hand in hand to produce the final optimized code.

(Refer Slide Time: 00:32)



So, to start with the source program goes through a phase called lexical analysis, so this lexical analysis phase it takes out, it identifies the words that you have in the program. Then so that formally they are called token because this is something more than the word so it is some identified corresponding to the words that we have plus some information like if it is a number; for example, then what is the value of that number. So, accordingly that the token may contain a token id part and one value parts.

So, we will see that later. Those tokens are used by the syntax analysis phase in the syntax analysis phase we check the grammatical correctness of the program. So the grammar rules of the language they are consulted. And whether the words appearing in a

particular sequence gives rise to a meaningful program or not or syntactically correct program or not so that is evaluated by the syntax analysis phase.

So, output of the syntax analysis phase is a part 3. So, that shows how the grammar rules of the program can be utilized to show that this program is syntactically correct or grammatically correct.

And so, grammatical correctness does not mean that the program is also semantically correct. For example, say one integer variable and the real variable both of them are identified as variable, but what we need is that we need to also sometimes we need the certain operations that can be done on integers certain operations can be done on real.

So, we need to categorize between these 2 types of variable. So, that those are done by the semantic analysis phase, and there we check whether the way the program constructs or are the components in the program had been used so whether that is correct or not. So if the syntax and semantics of the programmer correct, then it goes into the code generation process, and there it goes through a 2 phase code generation.

In the first phase one intermediary code is generated and from there the final target code is generated. So intermediary code is in some hypothetical language and from there whichever processor for usually want to generate the code that is the target processor so for the target processor the code is generated.

So, the target code it goes through optimization because of these automated process they are remains many scopes at which the program can be or the generated code can be made more efficient in terms of execution speed, in terms of area or the total storage requirement. So, that way so it goes into code optimization phase.

And output of the code optimization phase is optimized target code. Now as we have discussed in the previous classes that ok, so, it is not mandatory that the compiler will produce only optimized code so for silicon compilers it will produce optimize circuitry. So, they are also optimization is necessary because it may so happen for example, 2 inverters they come one after the other. So, an optimization may remove those 2 inverters because they replace them by a equivalent operations. So, that way so this type of optimizations are necessary.

So apart from this major flow so there are 2 more operations that are done in a compiler, one is the symbol table management so all the symbols that are defined in the program. They are kept in a table called symbol table. And this lexical analysis phase and syntax analysis phase they actually make the symbol table and the later phases they are actually going to use the symbol table.

And there is another module which is error handling and recovery of module. So, that actually tries to give hint to the user like what are the possible errors in the program, and accordingly it will tell the user will rectify the program and give it for compilation again. And while doing this error handling so it may happen that the compiler itself goes into a state from which it cannot proceed further ok. So, in that case some recovery reaction is necessary so that is why this stage is called error handling and recovery. So, recovery maybe it will discuss some of the last few words that it has been in the source program. So, that it can come to a valid state.

Typical example is maybe if there is some error in a line so if in the syntax of the programming language says that every statement should end with a semi colon, there it will skip all the symbols till it gets a semi colon because after that it knows that ok, I will be in a clean state where I expect a new sentence. So that way this error handling and recovery routines are going to be useful.

(Refer Slide Time: 05:33)

**Lexical Analysis**

- Interface of the compiler to the outside world
- Scans input source program, identifies valid words of the language in it
- Also removes cosmetics, like extra white spaces, comments etc. from the program
- Expands user-defined macros
- Reports presence of foreign words
- May perform case-conversion
- Generates a sequence of integers, called *tokens* to be passed to the syntax analysis phase – later phases need not worry about program text
- Generally implemented as a *finite automata*

*x = y; y = z + k; z = x + k*

*y = z + k; z = x + k; Add x;*

*Y = Z + K; Z = X + K; Add X;*

*Y = Z + K; Z = X + K; Add X;*

**SWAYAM**

So, we will see them in this part of the lecture we will try to have an overview. And later on for each of these phrases we will dedicate quite a few lectures discussing about their development. So, the first phase is the lexical analysis phase so this is the interface of the compiler to the outside world so any program or the component that we want to compile. So, it comes to the lexical analysis phase. So, the major job of the lexical analysis phase is to scan the input source program and identify valid words of the language in it.

So, as I was telling that at the lowest level of a language you have got alphabet set. So, every language will accept certain alphabet, and those alphabets are combined in some fashion to make the valid words. Now, if those if the next combination of this alphabet is such that these does not make any word of the language, then the lexical analysis phase is supposed to identify that, and then it can tell that this word is not known to this particular language so this that is an error.

So, that is the purpose of the second point that we are discussing, it will scan the input source program and identify the valid words of the language unit. Now apparently it seems it is a very trivial job, but it is not so really because when we are writing programs so, we are we are very much flexible. Like in the sense that somebody some programmer may write the entire program in a single line because it just says that after going from one line to the another line you have to put a semi colon. So, somebody may write a program in this fashion say,  $x = y$  semi colon  $y = z + k$  then  $x = x + k$  so, something like this.

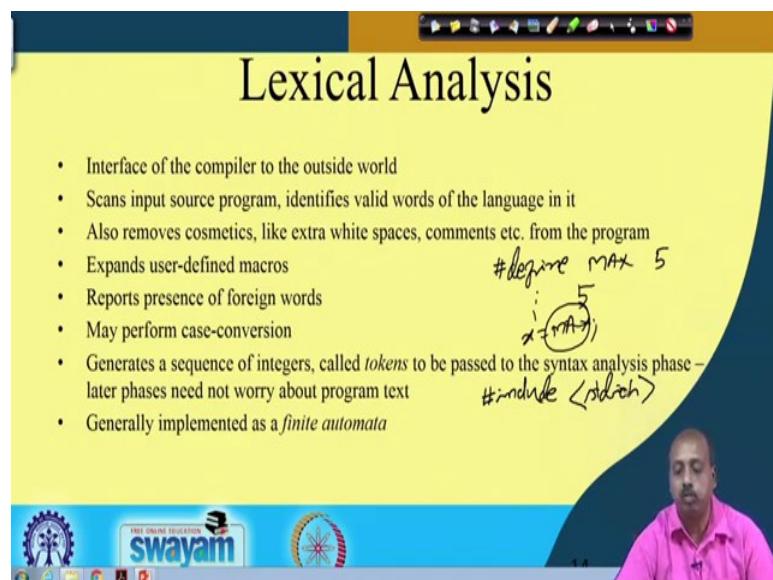
Somebody may put it in separate lines  $x = y$  in one line, then  $y = z + k$  in another line, then  $x = x + k$  in another line, somebody may like to put some comments also here, somebody may put a comment here at and this comments may also run in multiline fashion. And this somebody may write this line  $y = z + k$  as  $y$  then a quite a few blanks then  $z$ , then again quite a few blanks then  $k$ .

So, this type of variants variations in the source program can occur. So, we need to identify, you need to identify this situation somehow we need to ignore all this extra cosmetic things that are put into the line. Similarly, so whether the program is written in this fashion or this fashion or this fashion everything is correct ok, all of them are correct.

So, this lexical analysis phase it is responsible to take out the actual words meaning full words from the program so that is why it is job is a bit difficult. So, we will see that so it will remove cosmetics. So, as I said extra whitespace so whitespace means that blank, then tab, then new line characters.

So, they are all white spaces so it will try to remove it will remove all the extra white spaces comments, so you may put some comments in lines of comments are not compiled. So, they are not meaning full to the machine. So, they are to be removed. So, that removal is also done by the lexical analysis phase. So, these are some of the responsibilities, in many programming languages for example, in c language you will find that we have got macros like we say, hash define hash define say max 5.

(Refer Slide Time: 09:19)



And later on in the program wherever I write say x equal to max. So, that this max has to be replaced by 5. And that is we know that it is this translation or this transformation is done at the compile time. So, this max is replaced by 5 at the compile time.

So, this is also actually done by the lexical analysis phase. Similarly, we have got another compiler directive like hash include in c language. So, we say that #include stdio.h or some any other file.

So, what it means from the language we know that this stdio.h this particular file will be attached to your program before compilation. So, before so this stdio.h file has to be

taken and it has to be attached with your program. So these are all done by the lexical analysis phase.

So it expands the user defined macros, they have defines hash includes excreta so all those pre compiler direct this they are all expanded by the lexical analysis phase. Reports presents of foreign words, as I was telling that some words, some sequence of alphabets maybe put which is meaningless for the particular language.

So, it can report that such an such word is not in my language. So, that is the foreign word; may perform case conversion like sometimes we need to do case conversion upper case to lowercase or lowercase to uppercase.

So, sometimes a language is case sensitive sometimes it is not. So, accordingly this lexical analysis phase it may have to do some case conversion. Now as an output, this lexical analysis phase it generates a sequence of integers called tokens. So, all the before this lexical analysis phase what we have is a program which is a character stream.

So, it is a stream of characters, but at the output of the lexical analysis phase, what the lexical analysis tool has done, is that it has identified the words that are present in those character stream. And for every word it has got a predefined integer. The compiler has a predefined integer value and that value is passed from that point onwards.

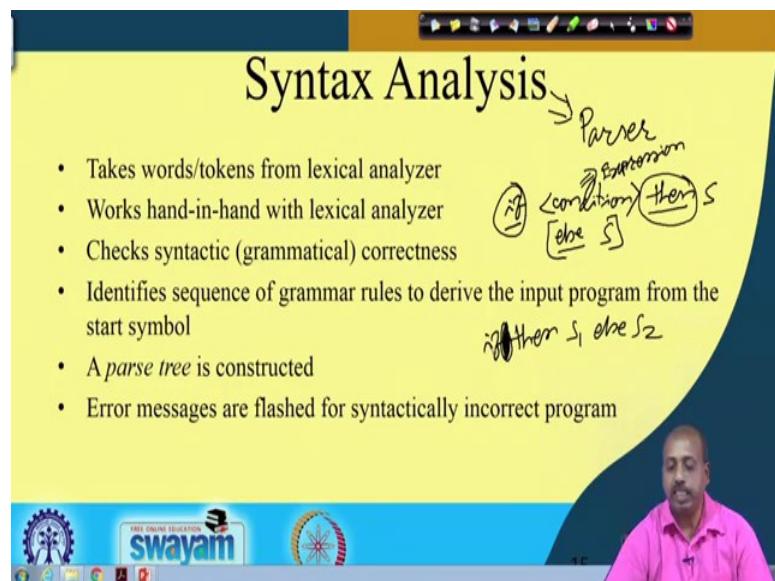
So, from the lexical analysis point onwards so, we can say my program is nothing but a sequence of integers so that is called a token. So, it generates a sequence of integers called tokens to be passed to the syntax analysis phase and naturally later phases did not worry about program text.

So, handling working with characters is a problem because there may be spaces and all. So, once it is a number it is sequence of numbers the handling then becomes easier. So, this is we will see that in detail those that is all the responsibility of lexical analysis phase.

So, it is generally implemented as finite automata, so as I said that compiler subject is very much dependent on the automata theory. And this lexical analysis tool, it uses the finite automata for design for its construction.

And we will see that there are different types of finite automata non deterministic finite automata, deterministic finite automata and also this lexical analysis tool can be built around those finite automata.

(Refer Slide Time: 13:00)



So, after the lexical analysis phase the next stage is syntax analysis. So, the syntax analysis phase it is also known as parser. So, this phase is also known as parser. So, we will be using these 2 terms interchangeably just remember that they mean the same thing.

So, this parser it takes words or tokens from lexical analyzer, and they work hand in hand, so it is not that this first the lexical analyzer will generate all the tokens that is there in the program; then the parser will start working on that it is not like that. So, the way it operates is the parser starts as and when it needs to know what is the next token.

So, it will ask the lexical analysis tool what is the next token, so accordingly that lexical analyzer will scan the input, and it will return the next probable token. So that way it goes, so they work hand in hand with each other this lexical analysis and syntax analysis.

The major responsibility of the syntax analysis phase is to check for syntactic correctness, grammatical correction. So, for example, if you look into any language so it will say that for example, I have got an if then else statement so it is said that first this q r if should appear, then some condition should come, then some condition should come

then the keyword then should come, then I can have some statements else some statements. So, out of that this else statements of this part is optional ok.

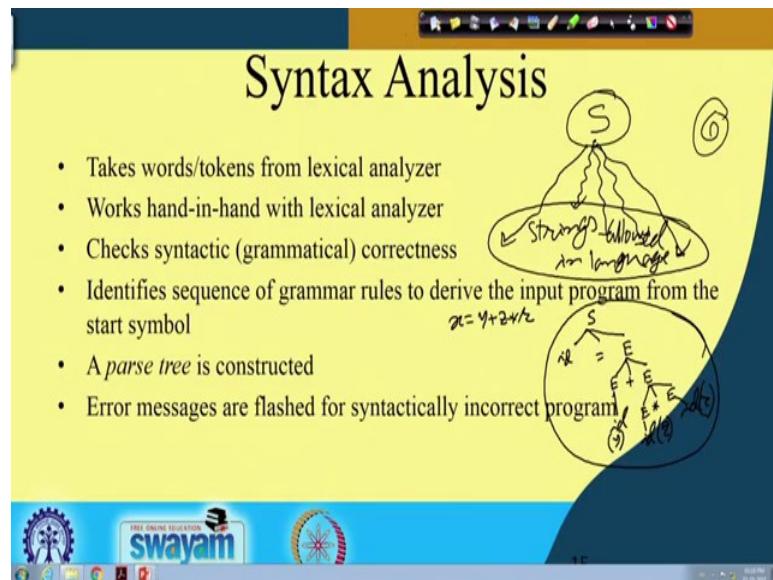
Now, if we see this if then else statement so this is a grammar so it starts with the keyword if then this condition. So, condition is nothing but some expression and what type of expression is supported in the language so that is also depicted by the language designer. So, they have told what can be the expression, so the conditional expression so that will be there. Then this keyword then should appear then we can then we should have some statement.

And after that we have an optional else part so if this optional part is present then this else q r must be present. So, that is the grammar rule so we do not have any grammar rule I cannot write like this if then S 1 else S 2. So, this cannot be written because grammatically this is correct incorrect because in between I needed a condition so that is missing.

So, the syntax analysis phase will actually identify this type of mistakes if the whether the program is syntactically correct or not or grammatically correct or not and if the program is grammatically correct if the input is grammatically correct then it will identify a sequence of grammar rules to derive the input program from the start symbol.

So, as I said that the language is specified by a grammar and every grammar has got a start symbol. So, that all strings that are allowed in that the possible in that language they can be derived from that start symbol using the grammar rules.

(Refer Slide Time: 16:21)



So, there is a special start symbol say S. So, if this is the set of all strings that are allowed in the language. So, these are the strings allowed in the language in the language L. Then, if you use the grammar rules so if G is the set of grammar rules so you can use this set G so that you can have derivation to all of them from this start symbol and in the derivation process it uses the grammar rules of G.

So, then this S is called the start symbol of the grammar. So, all valid programs they can be derived from the start symbol, on the other hand if the program is syntactically incorrect then you cannot derive the program from the start symbol of the grammar.

So, that is the idea so theoretical, the automata theory so it will allow us to do this particular exercise. It will give us a tool by which if a program is syntactically correct. So, we will be able to derive the full text of the program starting with the start symbol of the grammar. On the other hand, if the program is syntactically incorrect it will not be possible to derive the program text starting with the start symbol of the grammar.

So, that is the point or that is the advantage of this syntax analysis phase. And if the problem is syntactically correct so it will construct a parse tree. So, parse tree is basically telling us how this program can be derived for example, if suppose I have got an expression x equal to y plus z into r then you can say that as if this can be derived like this, that I have got a statement for an identifier equal to sum expression.

Where this expression is expression plus expression, this expression is giving me an id which is y and this expression is giving me another expression multiplied by expression and this expression is id which is z, and this expression is another id which is r ok.

So, this way we will see that this type of trees can be drawn, where starting with the start symbol of the grammar will be able to derive the whole string. So, this is called the parse tree so we will come to this again later when we go to the syntax analysis phase.

And in this apart from this generation of parse tree so, for the correct programs it will generate the parse tree, so what about incorrect program? For incorrect programs it will flash error messages that so that the user can rectify those errors and then you can again give the job for compilation.

So, this error message design error message display pointing the actual errors and identifying more number of errors in a single pass, so they at the challenges that we have in the syntax analysis phase. So, next comes the semantic analysis.

(Refer Slide Time: 19:50)

## Semantic Analysis

- Semantics of a program is dependent on the language
- A common check is for types of variables and expressions  
     $\{ \text{int } x, y, z; \}$   
     $x = y + z$
- Applicability of operators to operands
- *Scope rules* of the language are applied to determine types – *static scope* and *dynamic scope*

So, semantics of a program is dependent on the language, so for the same sentence or similar sentence to languages name in 2 different functions ok. So every program is nothing but some computational some function. So whatever input you are taking the program is transforming the input to some output so in some sense the program is a function. So that function is specified by means of the program statements.

So that meaning, that function that we are talking about. So that is the semantics of the program. So the meaning of the program, so meaning of the program is the function that it is executing that it is representing.

So, semantics of a program is dependent on the language, a common check is for types of variables and expressions. So, this is a very common sort of thing that we have you know almost all the programming languages where type check is mandatory. Why? Because so for certain operations you cannot do on a certain type of variable; for example, you have got this integer division and integer reminder operation so which are not applicable for real numbers.

On string variables you can have string concatenation you can have search for a substring and all, which are not applicable if you are taking an integer variable or a real variable like that.

So, if the user of the programmer has done some mistake and has got has written some something which is wrong from the point of view of types of the variables and expressions so that should be caught. So a common check is of the semantic analysis phase is for type checking, where we essentially check the applicability of operators to operands.

There are certain scope rules of language that are applied to determine the types, like say for example, at some points suppose I write say  $x$  equal to  $y$  plus  $z$  fine. Now it is desirable that these expression  $y$  plus  $z$  its type should match with the type of  $x$ , but how do we know what is the type of  $y$ , what is the type of  $z$ , what is the type of  $x$ .

So, you say that if this whole thing is within a function then somewhere earlier so, the  $x$ ,  $y$  and  $z$  have been they have been declared. For example, there may be declaration like integer xyz ok. So, you can take that so you get an idea like what is the type of  $x$ ,  $y$  and  $z$ . So, when these types of these variables can be obtained at the time of compilation itself. So, that is called static scope, the scope of the variable or a scope of a definition it is static.

So within this function the  $x$   $y$   $z$  all of type integer; however, there is another type of scope rules which says that it depends on the execution sequence in which the functions are invoke so, that will determine the values of  $x$ ,  $y$  and  $z$ .

For example, in c language you know that if I do not declare the type of a variable in a function and if the variable is available globally, then the compiler will take the type of the variable as whatever is defined globally. However, some programming languages that allow nesting of functions or nesting of procedures.

So, if something is defined some variable is used somewhere whose type is not available within the function, then it will go up and go up in the hierarchy and it will try to identify the block at which this particular variable has been defined. So, that way this nesting of this programming language structures. So, that will determine the actual value actual type of the variable that we are going to use.

So, that actually comes under the scope rules, so, we will discuss about these in detail when we go to the type checking and all. So, for the time being so you know that programming languages they define some scope rules in their definition in their specification and as a compiler designer we have to follow those scope rules for code generation.

(Refer Slide Time: 24:22)

## Intermediate Code Generation

- Optional towards target code generation
- Code corresponding to input source language program is generated in terms of some hypothetical machine instructions
- Helps to retarget the code from one processor to another
- Simple language supported by most of the contemporary processors
- Powerful enough to express programming language constructs

17

So next we will be looking into so till this much so if a program is semantically correct; that means. So, we there is no error which is done by the programmer at this point. So, it can go into the code generation phase.

So, this code generation phase it as I say that it goes through 2 different stages; one is called intermediate code generation, another is called the target code generation. So, in the intermediary code generation so we use some hypothetical language and code is generated in terms of that language. So, why is it done? So we will see it very shortly.

So, this is and as such this intermediate code generation is an optional states, so it is not necessary that you should always generate an intermediate code and then go to target code. So, it is an optional stage and the code that is generated it corresponds to input source language is generated in terms of some hypothetical machine instructions and that is up to the compiler designer to assume the language, the statements of that language. And we will see some types of this intermediary language later in our course.

So, you will see that we have got flexibility like we can say that my language will support this so type of constructs, and accordingly the code will be generated using that language. The point that helps it that will help us by having this intermediary code generation is that; it will help to retarget the code from one processor to another. So, it is like this, suppose I have designed a compiler which is targeted to say Intel x 86 architecture.

So, for that we know that x 86 has got an instruction set so we can generate code targeting that x 86 processor and the code for that. Tomorrow the same compile if I want to generate code for some say dec alpha machine so that will not possible because the same code will not run there so I have to again do the entire code generation phase.

And that we can save it difficult because now I have to most of the time this code generation is integrated with the parsing or the syntax analysis phase, and then we have to start modifying the syntax analysis phase the from that point onwards so that makes a difficult.

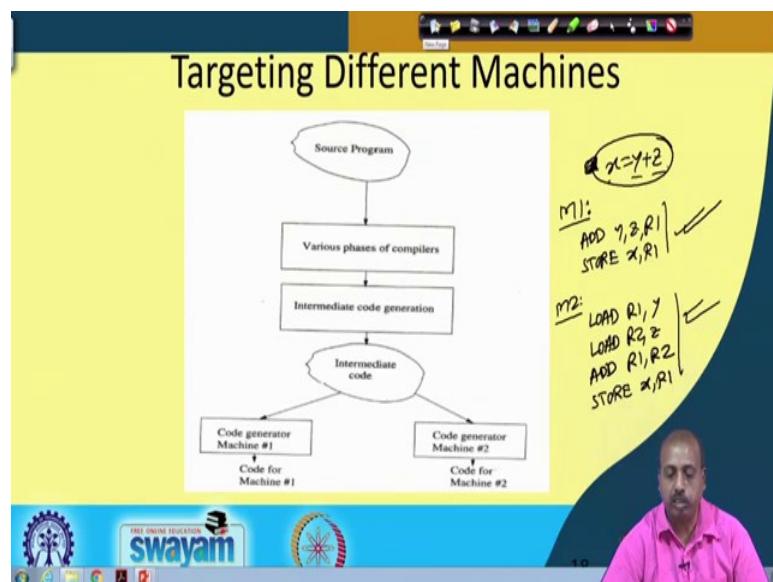
So, what is generally done; this intermediary code generation is that so which once you have a code in that intermediary format so from there it is just a template substitution sort of thing. So, for each intermediary language statement, you can have a template in terms of the target language code, targets language code and then you can just do this template substitution to get the target language program.

So, the retargeting the compiler becomes very easy, from one processor to another processor. Now what about the power of this intermediary language? So, it must be power full enough to express programming language constant.

So, we will that if it is far away it is very very if it is met to simple then it will not be able to catch the programming language constructs or it will not be able to represent the programming language constructs very easily so it will be difficult there. At the same time if it is at a very high level then also there is problem because it will be far away from the machine language code; that machine language of the processor. And then it is another compilers job to translate the intermediary code to the machine language code. So, both way we have got difficulty.

So, you have to do something so that we are add some intermediary stage which is equidistant you can say it is more or less equidistant from this source code and the machine code, so we will see them slowly.

(Refer Slide Time: 28:21)



So, this is the thing that I was talking about. So, we have got a source program that is source program passes through various phases of compilers. And after that it comes to intermediary code generation. So, this intermediary code generation phase produces an intermediate code.

So, this source language program after going through this compilation phase so it has come to an intermediate code representation. Now, from this point onwards so if I am trying for trying to generate code for machine 1.

So I can have a small piece of code which will translate this intermediary code into this code for the machine 1. If I want to retarget the code to target to machine 2. So, that can also be done by developing a small translated here as I was telling that this is simply a template substitution sort of thing. So for example, if I have got a statement like say x equal to say, say x equal to y plus z. So, this may be a statement in the intermediary language.

Now, when I am talking about machine 1 so machine 1 may be it is a machine where are this y and z can be kept in the memory and the x has to be a register. So, we can say that Add y z R1 and then store x, R 1 where as in M 2 it may so happen that it does not allow this memory operands directly so all the arithmetic operation so they are to be done on registers. So, in that case the code generation will be something like this that LOAD R1 comma y LOAD R 2 comma z ADD R 1, R 2 and then STORE x comma R 1. So, you see that if this is the program, if this is the intermediary language statement that we have.

So, if I have this type of small small templates for addition so this is the template in M 1 this is the template for M2. So, I can do a simple template substitution so this x equal to y plus z is substituted by these template for M 1 and this template for M 2, so you can generate the corresponding code easily. So, I do not need to regenerate the, redo the entire phase of the compiler I do not have to start designing the compiler from the scratch. So, that is the advantage that we have with this intermediary code.

(Refer Slide Time: 31:09)

## Target Code Generation

- Uses template substitution from intermediate code
- Predefined target language templates are used to generate final code
- Machine instructions, addressing modes, CPU registers play vital roles
- Temporary variables (user defined and compiler generated) are packed into CPU registers

Now, coming to the target code generation phase so target code generation is phase is nothing but template substitution from the intermediate code. Predefined target language templates will be used for generating the code, machine instructions addressing mode, CPU registers. So, they will play vital roles. So, here come the architectural input like the architecture designer have told ok.

This is these are the machine instruction, these are the addressing modes, these are the CPU registers which register can be used in which for which purpose and all. So, everything has to be documented and the compiler designer has to know that by heart and then accordingly do the code generation phase, so that is that makes it very critical.

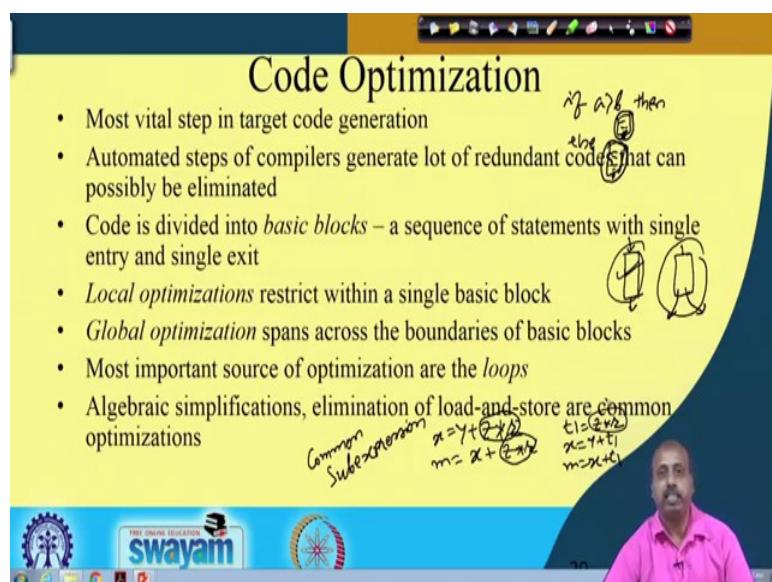
Temporary variables are user defined and compiler. So, they are generally packed into CPU registers so that this operation can be made faster. So, this target code generation is nothing but some template substitution and these templates will be designed very carefully by taking consideration of this machine details like machine instructions addressing modes CPU registers etcetera.

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

**Lecture – 04**  
**Introduction (Contd.)**

Once the code generation is done, I some sense the job of the compiler is over because, we started with the source code and accordingly we have generated the target code. So, that should be end of the compilation process. But, it really not so and there are many vital operations that goes on after this stage. So, that is known as the code optimization stage.

(Refer Slide Time: 00:40)



Now, why this is very vital? This is vital more so, because this whole compilation process is automated. And like if I am given a piece of source language program and I am given the opportunity do hand coding of each instruction in to machine language statements then possibly I can do much better work because we can visualize like forty where is the scope of optimization.

But, when I am doing template substitution sort of translation then there is very much possible, it is very much possible that some operation that we have done in the previous template are to be undone in the next template to do the next job done. So, for example,

suppose I have got, I have stored the value of y plus z in to a memory location x in the very next instruction I have to load that value x in to another CPU register.

So, instead of doing it like that maybe I can just store, I can keep a copy of that x in to the CPU register. So, that in the next instruction I do not need to copy it again from the memory, get it again from the memory. So, that way so we have to some sort of optimization. So, this most is a most vital step in target code optimization.

And if you look in to the type of research work that is going on in the domain of compilers so before these optimization remaining part of the process so they are fully automated so there not much scope exists for research on that. But, this code optimization there are many, there are lots of scope for improvement and lot of scope for doing further research and all. So, most of the research works are centered around this code optimization phase but to come to the code optimization phase we have to do the previous stages starting with lexical analysis up to the target code generation. So, that part has to be done, anyway.

So, this automated steps of compilers they generate lot of reddened codes that are that can possibly be eliminated. So, this will be more clear when we go in to the intermediate code generation. We will see that you generate lots of temporaries and those temporaries are just used in the next statement and possibly that temporary could have been eliminated ok. So, we can merge many of those temporaries together.

So, there are lot of redundancy in the code. So, many at many time what happens in that from one intermediately code that is generated it has got a branch statement which makes go to some level 1 1 and coming to 1 1 we find that there is another go to go to 1 2. So, in the first place itself, so we could have made it as go to 1 2 so that this is the another extra jump can be eliminated. So these are all sources of code optimization that can be done. But the code optimization is very difficult in the sense that it takes time. So, if you do not optimize a program then you will see that the compiler runs much faster compare to the case where we have got we ask the compiler to do the optimization.

And often compilers they have got different levels of optimization. So it is not uncommon to have say about 4 levels of optimization and the user can specify up to which level the compiler will do the optimization. Though in this particular course we do not have scope for go in to code optimization part, but you must keep in mind that this is

one of the basic job that almost all the modern compilers they will have in it the optimization part.

So, for the purpose of optimization the target code that we have so or the machine code that we have. So, they are divided into basic blocks. So, it is difficult like if this entire chunk of program is machine code is given to the optimizer and the optimizer is asked to do the optimization so, it becomes very difficult. Unfortunately this code optimization is a is an np heart problem. So, if you give it any code optimizers, so if you give it a big piece of codes it will not be able to do a good work. So, that is done is that we divide this whole program in to basic blocks.

For example, we can say that suppose I have got a statement like say if a greater than b then I have got a block of statements else. So here I have got a number of statements else another block of statements so, it may be like this. So if this block of statement and this block of statement, so if you look into them we will see that they are of this types so they have got a single entry point and a single exit point. Whereas, this if-then-else statement, so it has got its structure is like this. So, it has got a single entry point but it has got two different exit points. So, it can come up from the then branch it can come up from the else branch. So, it can come up from this point or it can come up from this point. So, this particular structure is more complex compared to say this structure.

So, when we have we have got a program or machine code program. So we analyze the program and identify this type of blocks the basic blocks ok. So they are called basic blocks and any basic blocks. So, you can say that code is divided into basic blocks and a basic block is nothing but a sequence of statements with single entry and single exit point. And we do local optimization within this within a single basic block. So, this total optimization process it can be divided in two phases, local optimization and global optimization. So local optimizations are much simpler and global optimizations are more complex.

So, first as I said that the compilers may have different levels of optimization at the lower values lower levels of optimization may be it will do only local optimization and as you go to higher levels. So it will do both local and global optimizations. So local optimization it will restrict its view within a single basic block only. So it will not look

across the basic blocks. So, that way if some variable is reused in some other basic block then that is not taken care of in this case.

So, it is all local optimization. So a typical example of this optimization is suppose I have got two statements like say  $x$  equal to  $y$  plus  $z$  into  $r$  and after that we have got another statement  $m$  equal to say  $x$  plus  $z$  into  $r$ . Now between these two statements is  $z$  into  $r$ . So, this part is common ok so this is known as common sub expression; so, this is known as common sub expression.

So, a typical optimization that the compilers do is to identify these common sub expressions and evaluate those common sub expressions only one. Like here if I have got a if I take a temporary variable  $t_1$  into which we compute this  $t_1$  equal to  $z$  into  $r$ , then we write like  $x$  equal to  $y$  plus  $t_1$  and  $m$  equal to  $x$  plus  $t_2$   $x$  plus  $t_1$ . Then what happens is that this multiplication we do only once. So, we save in terms of the CPU time. So, this is the, this was this common sub expression.

So, when we talk about local optimization we look for common sub expression within the basic block only and when we talk about global optimization we look across the basic blocks. So, across the boundaries of the basic block we try to see whether the variable common sub expressions are existing or not. So this is just a typical example, there are many other optimizations points at which this local and global optimizations can be tried out.

(Refer Slide Time: 08:53)

Code Optimization

- Most vital step in target code generation
- Automated steps of compilers generate lot of redundant codes that can possibly be eliminated
- Code is divided into *basic blocks* – a sequence of statements with single entry and single exit
- *Local optimizations* restrict within a single basic block
- *Global optimization* spans across the boundaries of basic blocks
- Most important source of optimization are the *loops*
- Algebraic simplifications, elimination of load-and-store are common optimizations

for  $i=1$  to 100  
   $\boxed{ } \rightarrow 100$

$x = 2x$        $x = 16x$   
 $x = 1 \leftarrow 1$        $x = 4 \leftarrow 4$

swayam

Now, so, most important source of optimization are the loops, because of the reason that say if I have a loop say I have got a say for loop say for  $i$  equal to 1 to 100 I have got a block of statements. Now, you see that if I can if I can optimize this portion ok.

And if I can reduce the execution of this block by say 1 millisecond then since this loop is repeated 100 times, so I have a total saving of 100 milliseconds whereas, if the loop was not there then my saving was only 1 millisecond. So, this way whenever there is a loop and particularly nested loops, so, if you can optimize at the innermost level of the loop then that has got the greatest impact on the overall optimization of the program or overall execution time of the program. So, that is why most important source of optimization are the loops. So loop optimization is very common and most of the compilers they will do some loop optimization and to come up with the optimized code.

Other, optimizations are like algebraic simplifications like algebraic simplification a typical example is suppose I have got an operation like say  $x$  equal to 2 into  $y$ . Now, this 2 into  $y$ , multiplying  $y$  by 2 you know that this multiplication by 2 can be implemented by shifting  $y$  by 1 bit, shifting left by 1 bit. Similarly, whenever it is a power of 2 like say  $x$  equal to say 16 into  $y$ , so you know that I have to shift  $y$  by 4 bits to multiply it by 16. So, this way we can, so whenever we have got powers of two for multiplication we can change it to shift operation. So that is one possibility so that is one algebraic simplification.

Then elimination of load store so sometimes we as I was telling that if in the previous instruction I have computed the value of  $x$  and stored in the memory location  $x$  and in the very next instruction I am again trying to load the value of  $x$  into a CPU register. So it may be better that we keep the  $x$  value of  $x$  in the CPU register itself so that I do not need to load it again. So this load store optimization so, this can also be done.

So, these are various sources of code optimization and there is a huge scope of doing something to the generated code or optimize the generated code so that we get a lot of performance enhancement for the generated code.

(Refer Slide Time: 11:44)

Symbol Table Management

- Symbol table is a data structure holding information about all symbols defined in the source program
- Not part of the final code, however used as reference by all phases of a compiler
- Typical information stored there include name, type, size, relative offset of variables
- Generally created by lexical analyzer and syntax analyzer
- Good data structures needed to minimize searching time
- The data structure may be flat or hierarchical

X

FREE ONLINE EDUCATION  
swayam

So next we will be looking into the module for symbol table management. So as I said that symbol table management is the symbol table is not directly part of the code that is generated. So apparently it seems that this is a something extra that we are doing but this helps in the compilation process.

So, by definition symbol table is a data structure that holds information about all symbols defined in the source program. So, it has got all info, all the symbols. So, symbols are like say the variables, constants, procedural names, function names so, they are all symbols or if they go to us, go to as are supported in the program then the levels, so all of them so they are symbols.

So, these symbols are kept in the symbol table they are noted in the symbol table so that whenever you are doing some later stage like code generation or semantic check things like that, so you can very easily refer to the symbol table and find out the value. So, typically the point is that, maybe while writing the program some time here I have defined integer x. And sometime later I am using like y equal to x plus 2, something like this.

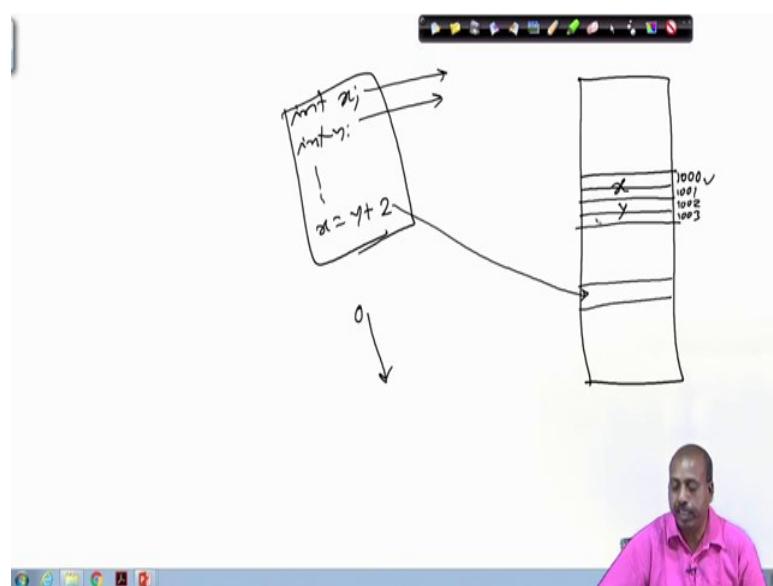
Now, when I am at this point trying to generate code or trying to check the validity of this line. So, I need to know what is the type of x? At that point it is not possible to go back to this line and check what the type of x was. So, instead what is done is that when this particular definition is seen so, in the symbol table we make an entry for x and

note down that it is type is integer. So that when I am at this line so I can refer to this symbol table this x is a symbol, so I refer to the symbol table to see the type of it. So I immediately get that it is type is integer.

So, that is the purpose of symbol table. So it is not useful no it is not required in the target code final code does not have the symbol table in it, but in the generation process all the references by all phases of a compiler so they will be using the symbol table. Typical information that we store in a symbol table so they are like this name of the symbol, type of the symbol, size of the symbol relative offset of variable. So relative offset means from the start of the program at what address or at what value the particular variable comes.

So, it is like this sorry, it is like this. Suppose I have got a variable suppose I have got a program. So, here it starts with the say integer x, then integer y and sometime later I write like x equal to y plus 2.

(Refer Slide Time: 14:34)



Now you see if this is my program then when the code is generated for this x y etcetera. So, there will be some valid memory locations. So if this is my memory where this program has been loaded, suppose this program is loaded starting from memory location 1000 ok. And if x is the first variable that we have here so up and assuming that x integer takes 2 bytes, so the bytes 1000 and 1001. So, they are dedicated for x. Similarly, 1002 and 1003 they are dedicated for y; 1002 and 1003 they are dedicated for y.

Now, at some point later, when I have got this x equal to y plus 2 suppose this statement is coded here now whatever code that I put here this day at this point. So, it has to talk in terms of these at this value 1000 of our 1000 and 1002. And, so that because x y does not have any meaning here so I have to talk in terms of this value 1000, 1001, 1002. So, this is true valid if the program is loaded from location 1000.

So, tomorrow if the program is loaded from location 5000 then these values are to be 5000, 5001, 5002 like that. So, what is done? The compiler at the time of compilation it is not possible to know that which address the program will be loaded. So what we do? We assume that the program is going to be loaded from address 0 and with risk so from this or the 0 is the start of the program. So from that point onwards where exactly is x defined. So what is that is called the offset.

So, since x is the very first variable in the program so a offset of x is 0. If the size of integer is 2, then y is defined at the second address 0 and 1 they are occupied by x. So, 2 will be occupied by y, 2 and 3 so, offset of y is 2. So this way we can compute the offsets ok. So this offset computation is done and accordingly we can the code generation stage it can refer to this.

So, symbol table can tell us like what are the offset values. So, this is the, so that is this information is stored in the symbol table. So, relative offset of the variable. Like with the starting with the program at which of which distance from the beginning of the program is the variable stored.

So, it is generally created by the lexical analyzer and syntax analyzer phases and sometimes this syntax analyzer also uses this symbol, symbol table. Lexical analyzer also sometimes uses it, like say suppose whenever it lexical analyzer whenever it finds say another definition of x, so whether the x is already defined or not so that has to be seen.

So, duplicate definition checking and also if that may be required. So, then need to refer to symbol table and we need to have good data structure to minimize the searching time because this symbol table search is very common. So most people but if as you know that whenever we have got any table the type of operations that we are doing in the table are insertion, deletion and search. So, more generally these are the 3 operations that we do.

And in case of symbol tables, you see that insertion is common but insertion happens and near the beginning of the program code and when you are analyzing the program. So, deletion is very rare because we are not going to delete any symbol from the symbol table possibly and but search is very much. Like whenever I am looking to any program; any program line that we have, so there will be some variables in the program in that line and those variables for those variables we need to identify the type offset etcetera. So, search is very important operation.

So, whenever the way that we represent symbol table or realize symbol table search operation should be made very efficient. So, if we need good data structures that can minimize this searching time whereas, this deletion time we need not be bothered much about it because we do not delete much. Insertion also to some extent it is important so, we need to do that but search is most important.

Data structure that we use for symbol table, so they may be flat or they may be hierarchical in nature. So, when you go to the symbol table management portion so, this will be more clear. So, it will be talking about different types of organizations or symbol table.

(Refer Slide Time: 19:228)

The slide has a yellow header bar with the title "Error Handling and Recovery". Below the title is a bulleted list of points:

- An important criteria for judging quality of compiler
- For a semantic error, compiler can proceed
- For syntax error, parser enters into an erroneous state
- Needs to undo some processing already carried out by the parser for recovery
- A few more tokens may need to be discarded to reach a descent state from which the parser may proceed
- Recovery is essential to provide a bunch of errors to the user, so that all of them may be corrected together instead of one-by-one *if and then some*

Handwritten notes are overlaid on the slide, including:

- A circled "Amrit"
- An arrow pointing to the text "Q=y+2;"
- An arrow pointing to the text "P=2x4;"
- A circled "X = Y + Z;"
- A circled "SOME"

The footer of the slide features the "swayam" logo and other educational institution logos.

Another very interesting job that the compilers do is the error handling and recovery. So, like how do you judge the quality of a compiler? So, quality of a compiler I mean the target code that is generated what is the quality of the code.

So, quality of the code can be judged by the execution speed of the code, how fast is the code executing, that is more or less depicted by the optimization that the compiler does on the program. Also to some extent it is dependent on the size of the code that is generated. So, how much memory space is needed? Though space may not be a very big criterion now because of all the computers now they have got large amounts for primary and secondary storage but at the initial times so this was a concern ok. So we can say we have got the memory size as a constraint.

Apart from that, compiler to a good compiler it should help the user in program development. So, for example, when the program is written initially there are lots of bugs in the program, both syntactical bugs and semantic bugs. So, the compiler should be able to catch those bugs very easily and report them to the programmer so that the programmer can correct them.

Now, it is an important criterion for judging the quality of compilers how much error handling it can do how much. So, for example, for first for example I can have you have a situation like this that in my program I am writing like  $x = y + z$ , then  $p = x$  into  $q$  like that. But, I have forgotten to define this  $x$ , so I assumed; it is for C program it is expected that there  $x$  should be declared somewhere before using them before using it. So, maybe this is missing as a result at this point, this point wherever  $x$  appears the compiler will give message that the error message that  $x$  is undefined. So, how many such situations the compiler can catch so, that is one problem ok.

Similarly, if I have got an internal statement maybe the growl the language grammar says that it should be like this if condition, then some statement else some statement. May be that I have missed this then the compiler should be able to catch it and then it should give me the information that ok, this is not there, then is expected. So, it should give a hint also. So, it is not that telling that there is an error at line number 30 so that does not help much. So, it should give me information that then that the compiler was expecting that token then where it called the error.

So, then the in that case the programmer can be we can very easily identify the situation, otherwise it is very common to look at the error list that a compiler produces and the programmer says no I have done it why the compiler is giving this error. So, it is definitely there is some problem but it often becomes very much confusing for the

programmer from the type of statement type of messages that the compiler generates. So, it becomes confusing for the programmer to identify the actual source of error.

A very common thing is that, some programming languages they if they want that all the statements they should end with a semicolon. Now, if I miss this semicolon and the next statement starts, now if the compiler ideally it should tell me that a semicolon is missing. But, if the compiler cannot pinpoint that situation then what will happen is, it shows some error at the next line because it takes that as if this assignment statement is continuing so it will try to take whatever tokens are coming after this. So, they are also part of the assignment statement. As a result the error message that is given may become confusing.

So, it is not a very easy job, it is not a very easy job to identify or to guess the type of errors that a programmer or a user can do but a good compiler designer should be able to do that guess ok. So, accordingly it should be able to can identify the proper messages that it should flash for to the programmer.

So, that is the thing that the compiler for semantic error, there is a type mismatch and also in that case it does not the program does not violate the grammar of the language. So, in that case it can proceed so, it can process the compiler can proceed but if there is a syntax error then we will see that the parser or the syntax analysis phase itself enters into an erroneous state from which it can it cannot proceed further. So, it needs to undo some processing which is already carried out by the parser for recovery.

(Refer Slide Time: 24:50)

Error Handling and Recovery

- An important criteria for judging quality of compiler
- For a semantic error, compiler can proceed
- For syntax error, parser enters into a erroneous state
- Needs to undo some processing already carried out by the parser for recovery
- A few more tokens may need to be discarded to reach a descent state from which the parser may proceed
- Recovery is essential to provide a bunch of errors to the user, so that all of them may be corrected together instead of one-by-one

After ... Then

FREE ONLINE EDUCATION SWAYAM

A man in a pink shirt is visible in the video player.

As I was telling that in that if statement that the 'then' keyword was missing. Now, what to do 'then' with the keyword or 'then' keyword is missing. So, we will see later on the say on when you go to the parser design that there are several strategies for handling it. So, one possible strategy is to go back a few tokens and you come to a state as if you have not seen the, if statement at all. So, this entire if statement you skip ok so, that may be one possibility.

So, that the parser which got stuck at this point it can come back to a descent state and from that point it can start again. So, the parser also enters into a erroneous state and in that case it needs to undo some processing already carried out by the parser for recovery.

So, some more for tokens may be needed to be discarded. As I said that if every sentence ends with a semicolon maybe the parser will simply look for a semicolon character until I until and unless it gets a semicolon token it can go on discarding whatever token the lexical analyzer gives it ok. So, that to come to a state where it has seen it to a semicolon token and from that point onwards the person knows that there is a new statement. So, the parser can start accordingly.

So, some tokens may be discarded to reach a decent state from which the parser may proceed. So, this is essential to provide a bunch of errors to the user. Why so much of complexity?

(Refer Slide Time: 26:22)

Error Handling and Recovery

- An important criteria for judging quality of compiler
- For a semantic error, compiler can proceed
- For syntax error, parser enters into a erroneous state
- Needs to undo some processing already carried out by the parser for recovery
- A few more tokens may need to be discarded to reach a descent state from which the parser may proceed
- Recovery is essential to provide a bunch of errors to the user, so that all of them may be corrected together instead of one-by-one

10 11 12 13

swayam

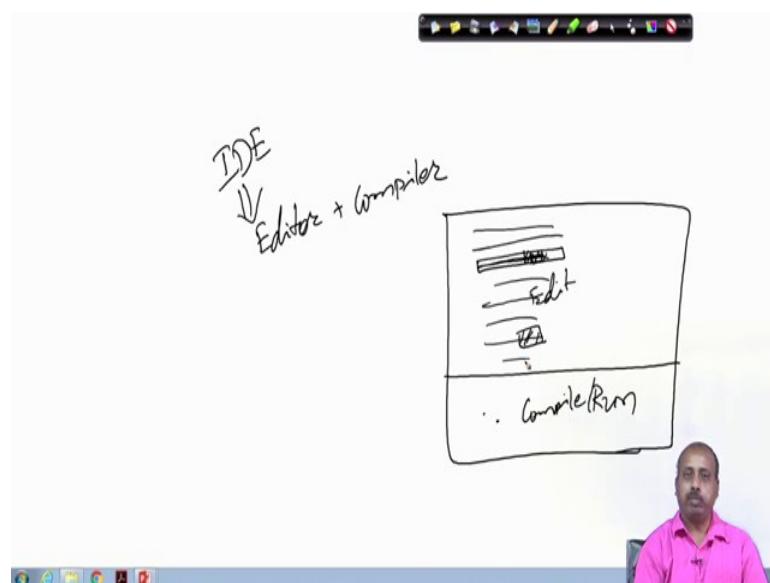
The point is like this, suppose I have I have written a program and this program is very long, so 10000 line code and there is an error at line number 10 ok. So, this is my line number 10, there is an error and when I give the program for compilation it gives me that there is an error in line number 10. I correct that error, again give it for compilation it comes up with a statement that there is an error in line number 12, again after correcting line number 12 it gives me the next error.

So, if it goes like this then you see the amount of effort that the programmer gives immediately the program will tell can the compiler not give me a full list of errors that is there in the problem. So, that I can correct all of them and they are submitted for compilation again. So, why so many times going back editing and then again submitting like that. So, that is very important ok.

So, what happened is that at this point maybe the compiler has come to a state the parser has gone to a state for which from which it needs to do a recovery. So, if it cannot do recovery, so it can stop at this point itself and then go back. Whereas, in case of but to show further errors, so before the user has corrected it. So, somehow the parson needs to recover it needs to recover from this state and do that and then again proceed so that it can catch these other errors also in the program. So, this is very common so, this recovery process is very important, so that it can the parser can come back and come to a decent state from which it can proceeds it.

So, if I say that every statement ends with a semicolon then the this line also ends with the semicolon. So, parser can simply ignore all the tokens steal the semicolon and then start working from the next line onwards. So, that looking into this semicolon it will understand that now the possibly the effect of the error is gone. So, I can start processing the next line. So, that way it can identify the errors that are there at line number 12 or some other future lines. So, this is very common when we have got these compilers which produces a list of errors.

(Refer Slide Time: 28:50)



Of course there are compilers which does not give this list which is normally in the situation where we have got this IDE or Integrated Development Environment where this editor and compiler they are together ok. So, this if this is comes up with a screen and in the screen there are portion, one is the editing part this is for the edit this is the edit window and this is the compile window or run window compile run etcetera; many tools have got this type of interface.

So, here I am right I have written the program and then I give it for compilation. And if may say so, maybe the if it identifies an error on this, this line maybe it will just come up with it will come up with this particular line, you highlight this line and maybe a few characters here which is the probable source of error. So, in this case it gives only one error at a time and the user corrects it and then it again goes for compilation and then maybe now it finds that there is an error at this point. So, this is also there so, instead of

producing a complete list, it can just show the next error position so, that can also happen.

So, but of course, if I am having a very large program then this is not a very good approach because there are large number of errors may be there. So, the many iterations of corrections will be, has to be they have to be carried out for getting the program syntactically correct. So, recovery is essential for a bunch of errors to the provider to provide a bunch of errors to the user so that all of them may be corrected together instead of one by one. So, this is very important that we should do.

(Refer Slide Time: 30:41)

## Challenges in Compiler Design

- Language semantics
- Hardware platform
- Operating system and system software
- Error handling
- Aid in debugging
- Optimization
- Runtime environment
- Speed of compilation

23

Now, what are the challenges that are faced by a compiler designer? Now, apparently it seems that this compiler design. So, it is not that difficult ok. So, if I tell you very frankly, suppose I have got a well designed language then design and I have got a very good idea about the target machine on to which I am going to therefore, which I am going to generate the code it should not take a good a very large amount of time. However, so the problem that we have is that languages they are also very complex they have got many interesting features many interesting functions can be specified there and the target machine also has got many important different type of facilities.

So, if we can exploit those facilities then we can have very good code generated. So, these are the challenges that are faced by the by a compiler designer, the language semantics like the functionality is that you can specify in the language. So, that makes;

the that makes the programmer capable of writing different types of programs with different meanings the compiler designer has to understand has to generate code accordingly.

The hardware platform that we are having, so, there is a hardware platform as we as we vary from one machine to another machine that the architecture itself changes so, naturally we have to do that. Operating system and system software; what is the underlying operating system? So, many of the operations that this programming languages have. For example, some programming language will allow you to do some file handling operation.

Now, the underlying operating system, so that we will also do that will implement this file handling by some methods ok. So, normally they are done by means of system calls and all. So, what are the system calls supported for this file and link? So, that has to be known by the compiler designer and it varies from one ways to another ways. So, the same code for one ways will not work for another one so, we have to be careful there. So, this OS and system software, they will play vital role in the compiler design process then error handling.

So, whether you are going to give a single error or multiple error like that as we have discussed just now. Then aid in debugging, the programs that we have, initial phase of program development they have got errors. So, apart from the syntactical error there are may be logical errors. So, the logical errors if you want to catch then the user will like to execute the program in a single step fashion that is one statement at a time type of execution environment. So, if we want to do that in a compiler has to provide some extra debugging information for that.

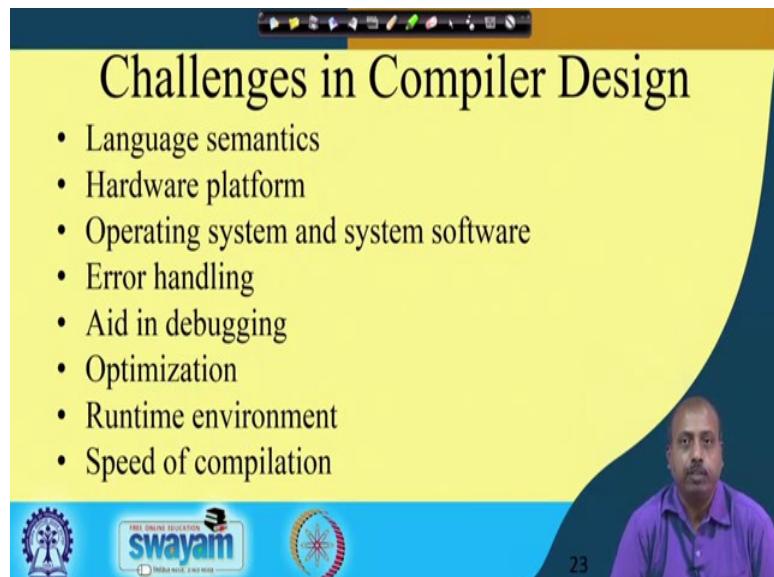
In the optimization, so that we have discussed in detail so, type of level of optimization and extent of optimization. Runtime environment; what is the type of procedure called, how are you going to implement the procedural call and also they will determine the runtime environment we will see that later. Speed of compilation; so, how fast is your compiler? So, if it is taking lot of time for compilation maybe if it takes 2-3 minutes then possibly we will say the compiler is very slow and then how to make the compiler fast? So, that is also a challenge.

So, you will see them in successive classes.

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

**Lecture – 05**  
**Introduction (Contd.)**

(Refer Slide Time: 00:15)



**Challenges in Compiler Design**

- Language semantics
- Hardware platform
- Operating system and system software
- Error handling
- Aid in debugging
- Optimization
- Runtime environment
- Speed of compilation

23

We are started discussing on Challenges in Compiler Design. So, there are several challenges, starting with the language semantics to hardware platform, then operating system concerns. Then the error handling, the powerful error handling will help in the will make the compiler more attractive. Then the debugging aids that is if I want to rectify the logical bugs in the program, then how can I do that, how the compiler can help in doing it.

Optimization; so I would definitely want more and more optimization, but that makes the compilation process more and more difficult; then the runtime environment management, so depending upon their facilities that we want to provide in the source language program like whether it supports, recursion and all; this runtime environment management will vary. And the speed of compilation that is how fast is the compiler, so it should we should not take lot of time for doing the compilation.

(Refer Slide Time: 01:09)

- “case” – fall through or not
- “loop” – index may or may not remember last value
- “break” and “next” modify execution sequence of the program

for  $i=1$  to 100  
  {  
    :  
    if (E) break  
    0  
  }

So, if you look into the language semantics, so here are a few cases, so which I would like to point out. One is that fall through case statement in most of the programming languages, they will support some sort of case statement. And this case statement in some programming languages it is a fall through case and sometimes it is not so.

(Refer Slide Time: 01:39)

So, we can take an example like most of the programming languages will have some sort of say switch statement switch or case. And some expression, some variable or

expression can be there, some expression E. And then we have got different cases. So, case 1, case 2, so like that we have got different cases.

Now, some programming languages they say that depending upon the value of E. So, it can enter into one of these cases, so this is a case K like that. Now, if E is E evaluates to two, so it will start executing from this point. So what happens, when it reaches the end of that particular case? So we can have two types of option. In one option it will keep the rest of the cases, and it will the end the switch statement ends at this point. Then after finishing this after finishing this line so it will start executing from this point.

In some other programming languages, so this is called a fall through case. In which ones this matches value matches, then it execute that part and it continuous so remaining cases are also executed from that point onward. So it is just the entry point. So, it is like this as if this all these cases, they form a piece of code a chunk of code, and you have got some entry points only. So, this may be the entry point for a 1, this may be the entry point for 2, this may be the entry point for K like that.

So, after you have entered, so this entire piece of code will be executed from that point onwards, until and unless there is a break statement in between, which will be taking it out of the case, so but this is true for languages like say C. But, in some other programming languages, so it is like this so this break is not there, but as soon as this is over.

So instead of continuing with the next case, so it will come out of the switch block all together. So, these are the two types of case statements that we can have been different programming languages if you look into different programming languages, you can find them. So, naturally the code that we generate for the two cases, so they are different two situation or two programming languages they are going to be different.

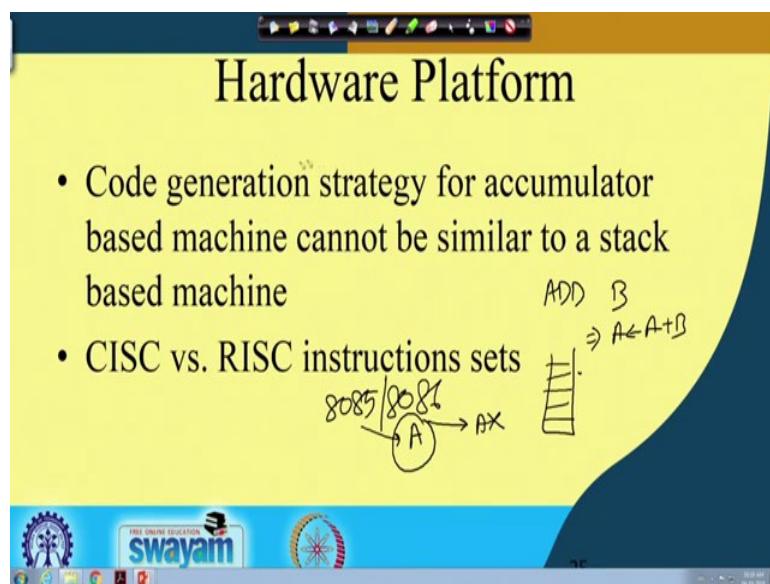
So, another concerned that we have is with say loop statement. So, loop statement, in some cases suppose I have got a loop like this. Say for i equal to 1 to 100, and in the body of the loop somewhere I have a break statement. So, suppose here there is a if some condition is satisfied, then we break from the loop. Then what is the value of i, when you come to this point ok.

So, there can be different types of semantics different type of logic about the value of i. Some programming languages say that when you are coming out with the way from somewhere in between without completing all the 100 iteration, so here you get the value of i, which it had at that point of time. In some other programming languages, it says that no i will be value of i is unknown. So, value of i may not be the value of i may not be predictable, it can have any arbitrary value.

So, this loop index. So, if the value may or may not be remembered beyond the last value or even if this finishes properly, i will become i it goes from 1 to 100 there is no break statement. Then what is the value of i, after the loop as completed. In some programming languages say, it will say that the value of i will be 101. Some programming languages, we will tell that it is unpredictable. So, depending upon that the code has to be generated right way.

Similarly, this break and next statement they modify execution sequence of the program. So, we have to take care of them. So this way the language semantics they have got many they put many compulsion or many conditions on the code that we are going to generate.

(Refer Slide Time: 05:57)



So, the next concern that we have is about hardware platform. So, code generation strategy for accumulator based machine cannot be similar to a stack based machine. So, in an accumulator based machine, what happens is that there is a special register,

which is called accumulator. And all operations that we do one of the operands and the destination is the accumulator.

For example, if you look into say 8085 type of architecture or 8086 type of architecture, all these architecture so they have got one accumulator register A in 8085, and in 8086 we have got AX. So, there any operation that you do so particularly for in 8085 architecture, so which simply say ADD B. So, ADD B means, the operation that is carried out is A getting A plus B, so that type of code that we that we generate, so it should be of this nature.

On the other hand this stack based machines so what is assumed that is that so there is a stack. So, whenever you do any operation so this arithmetic logic operations, so they just pop out the operands from the stack. Do the operation and push the result back to the stack, so that way the code generation for this stack based machine is definitely different from the code generation for the accumulator based machines.

Then this CISC and RISC architecture so they also play major role like CISC instructions so they are more complex. So, we have got a longer length instructions, complex instruction, they take different amount of time for execution. Whereas, the RISC instructions, so they are more or less similar. All the instructions are more or less similar in size and execution time.

So, as a result this CISC versus CISC and RISC instruction sets so they will also tell like what is the amount of what is the type of code that we generate for them. So, this way and they have got many other features like the RISC machine, so they have got large number of registers compared to the CISC machines. So, naturally will have more amount of the registered usage will be more. So, optimization and everything will be at different come in RISC machine compared to the CISC machine, so that is about the hardware platform.

(Refer Slide Time: 08:11)

The slide has a yellow header bar with the title 'Operating System and System Software'. Below the title is a bulleted list:

- Format of file to be executed is depicted by the OS (more specifically, *loader*)
- Linking process can combine object files generated by different compilers into one executable file

At the bottom of the slide, there is a blue footer bar featuring the Indian Space Research Organisation (ISRO) logo, the 'swayam' logo with the text 'FREE ONLINE EDUCATION', and a circular emblem. The number '26' is also present in the footer.

Operating system and system software, so format of file to be executed is depicted by the operating system. So, so what happens is that we have got this operating system will be having the tool called loader. And this loader module, so it will be finding out it will have some specific format of the object file ok. So, these object file that we are having so any compiler that you have. So, it should try to generate object file in that format. So, otherwise the loader will not accept it, so the program cannot be loaded into the memory.

Then the linking process or the linker tool, it also has got a major role to play, because they will combine many object file generated by different compilers into some executable file. So, this linking or linker tool, so it will also tell the of the acceptable object file format, so this is also there. So, of course there are many different link file formats. So, normally linkers support a number of different formats, but anyway so you it is this code generation process will be guided by the, this linking tool that we have.

(Refer Slide Time: 09:25)

The slide has a yellow header bar with the title "Error Handling". Below the title is a bulleted list of five points:

- Show appropriate error messages – detailed enough to pinpoint the error, not too verbose to confuse
- A missing semicolon may be reported as “<line no>: expected” rather than “syntax error”
- If a variable is reported to be undefined at one place, should not be reported again and again
- Compiler designer has to imagine the probable types of mistakes, design suitable detection and recovery mechanism
- Some compilers even go to the extent of modifying source program partially, in order to correct it

Handwritten notes are overlaid on the slide, including circled numbers 1, 2, and 3, and a diagram showing a flow from a variable assignment to a calculation.

The footer features the "swayam" logo and a small video frame showing a person. The URL "http://www.swayam.gov.in" is visible at the bottom left.

Then error handling. So, you have to show appropriate error messages. So, this is very important as I told in the last class. So, for the user to understand and to appreciate that this compiler is doing a good job. So, the user may be asking for user may be looking for more meaningful error messages rather than syntax error message like syntax error at line number 13, so that is very that is that is very cryptic. And the user will not be able to do much with that or even if at least the line number has to be mentioned. So, so if the line number is also not mentioned, then it is it may be a message simply that syntax error. So, syntax error where is the syntax error that you do so that makes it difficult.

So, they must be detailed enough to pinpoint the error. So, this error message that is the flash, so it should be detailed enough. However, it should not be very verbose to confuse, like say maybe the compiler tries to give two three different options by which the error might have occurred, so that way it is it sometimes it is more confusing. So, so those things are to be avoided.

So, this is actually up to the compiler designer to decide like what should be the appropriate error message for different types of errors and that way complier quality will be just based on the type of error message that is given. So missing semi colon may be reported as line number, and then followed by semi colon expected rather than syntax error. So, so the if you give this expectation part, what is the what was the expected

symbol here. See if you mentioned that, then definitely it is easier for the user to correct the program.

Another example is like this variable may be reported undefined at one place; it should not be reported again and again. So, what I mean is that suppose, I have got a piece of program, where at this point I have got the variable  $y$  equal to  $x$  plus  $z$ . And I have forgotten to put this declaration of  $x$  somewhere here. So, this declaration should have come before the usage of  $x$ , but somehow it is missing it is not there.

Now, in some later lines also I might have used this  $x$  equal to say  $x$  plus  $p$  that way  $x$  might have been used several more times in the piece of code. Now so it is so once this at this point we have detected, that there is an  $x$  which is undefined. So you have flash this message that  $x$  is undeclared or undefined. Then there is a no point repeating that message at this point as well ok.

So, so if a variable is reported to be undefined at one place, should not be reported again and again.

So to handle the situation what the compiler can may do is that compiler may make a temporary entry into the symbol table the with that indication and with a indication that this is actually undefined this is entered by the compiler. So, this accordingly the later part. So this when it searches the symbol table, so it will find  $x$  there so it will not be a undefined variable so like that so this can be done.

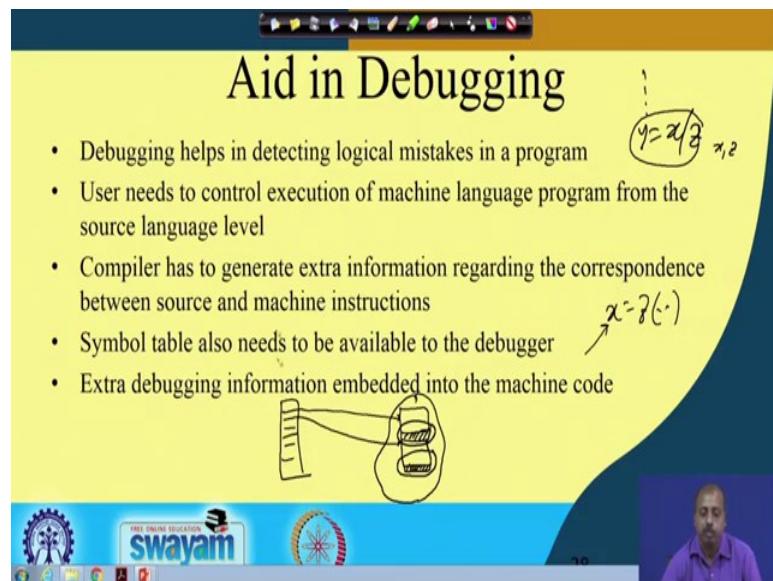
In compiler designer has to emerging, so this is the very important point that I wanted to emphasize. The designer has to emerging the probable types of mistakes, and design suitable detection, and recovery mechanism. And that is the expertise or experience of the compiler designer. So, what is the type of error that that any user may make while writing program in a particular language so that has to be imagined by the compiler designer.

And some compilers even go to the extent of modifying source program partially, in order to correct it. So, this is similar to say the corrections that we that is done by many of the word processing software's while we are entering the text. So it does some it replaces the wrong words by correct ones. So whether we like it or not that is the different issue, but sometimes it is very helpful. So similarly when we are writing a piece

of program for example, if I writing a C language program if at the beginning of a sentence if I write fi, then most possibly most probably I am actually I tried to write if and change it to I fi.

So, compiler can do this type of changes. So, it can change fi to if and when it finds that there is some errors some error is occurring then say fi, so fi will be taken as a variable and the variable if it is not available in the syntax in the symbol table; that means, fi is a fi is not a variable in that case may be the user has done a mistake and instead of writing fi if has written fi. So, this compiler can be can correct that fi to if. So, this type of corrections can be done by the compiler itself, so that is one possibility.

(Refer Slide Time: 14:27)



Then the other thing that we have is the help in debugging. So, debugging is suppose we have written a program. So, it is syntactically correct, but there are some logical errors in the program. So, it may so happened that in a program I am writing I have somewhere I have written like  $y$  equal to  $x$  by  $z$ . And we find that it is giving some result which is undesirable, and one possibly so maybe we want to we want to check at this point what are the values of  $x$  and  $z$ , so that whether the value of  $y$  is computed properly or not.

Or suppose we have called a function  $x$  equal to  $f$  some function, now after this function has been executed, we need to we may want to check the value of  $x$  to see whether the value computed is correct or not. Now, apparently we can do all these things by

introducing some print or write statements in my program at some intermediary places, but it is very cumbersome.

So, what the user can think about having a facility like I should be able to run my program line by line after every line the execution should suspend should get suspended, and I will be able to check the values of the variables, and then again give the continue signal. So, that this execution continues.

So, this way the user needs to control the execution of machine language program, but sitting at the source language level. So, the user does not know machine language level. So, what is the translation of each and every instruction? So, if the machine if the processor stops after every single program line execution, so that becomes difficult. But so that is of no use to the user because user will not be able to track at that level. So, user can track at the source language level, so that way we need to have some information where exactly to stop.

So, what you get is that if this is a source language program, and this is the corresponding machine language program, then after every statement that we have in source language I should. So, if this source language program translates to this much of code in the machine language then after this I should put some additional information here, so that I can suspend execution at this point. Similarly, after these are next piece of code starts here, then after that there should be some facility some code must be sitting here, so that I can ask the compiler ask the execution process to stop at that point. So, this is normally done by the help of a debugger this stopping execution in between and all, but the compiler needs to introduce these codes into the machine into the machine language program, so that is very important.

So, compiler can help in the debugging process by providing this information in terms of these that can help the debugger ok. So, compiler has to generate extra information regarding the correspondence between source and machine instructions. And symbol table also needs to be available for to the debugger, because debugger it will be the user will ask for values of different variables and debugger needs to find out the offset of those variables. And offsets are available only in the symbol table, so that way this offsets will be used by the debugger from the symbol table. So, symbol table is going to

be preserved in this case though previously we say it that in the target code symbol table does not have any role, but here it has got some role.

So, for if the program is to be in debug mode, then this symbol table is useful. So, this is particularly true when we have got initial phase of development. So, initially in the program there will be lots of bugs logical bugs. So, that way we need to debug our program mode and once we get the confidence that my program is now logically correct, so I can I can remove all this debugging information and then the I can generate a code which will be running more efficiently.

So, this there are two distinct situation in which the compiler is made to run; in one case it is generating lot of debugging information and it does not do any optimization there; in another situation it does not generate any debugging information, but does lot of optimization, so that the program runs efficiently. So, this so this debugging and optimization, so they go both of them are required in the program development phase, and the compiler should be able to handle that situation.

(Refer Slide Time: 19:19)

The slide has a yellow header bar with the title 'Optimization'. Below the title is a bulleted list of six items. To the right of the list, there are three handwritten mathematical equations:  $\theta = \gamma + 2$ ,  $p = xy$ , and  $\rho = x^2y^2$ . At the bottom of the slide, there is a blue footer bar featuring the 'SWAYAM' logo and other educational icons.

- Needs to identify set of transformations that will be beneficial for most of the programs in a language
- Transformations should be safe
- Trade-off between the time spent to optimize a program vis-a-vis improvement in execution time
- Several levels of optimizations are often used
- Selecting a debugging option may disable any optimization that disturbs the correspondence between the source program and object code

So my compiler should provide these facilities then optimization. So naturally so these are the series of transformation that may be necessary that may be beneficial for most of the programs on in a language. So the compiler designer must understand the general set of optimizations that are that are useful general set of transformation that are useful for most of the programs in the language. So guessing this particular say it is not very easy,

then the transformation should be safe. Safe in the sense that if I do some optimization, so it should not change is the meaning of the program.

For example, suppose I have got a statement like sorry I have got a statement like say  $x$  equal to  $y$  plus  $z$ , and then at different places I have got reference to this  $x$  ok. So, at this point, I have got a reference to  $x$   $p$  equal to  $x$  plus something. So, so then again at some point later I have got  $q$  equal to  $x$  into  $r$ . So, like that now this  $x$  is a variable. So, it is kept in the memory.

So if two in order to reduce the load on execution time maybe we do not load this  $x$  from memory again rather so this  $x$  may be copied into some CPU register as well and that at this point. So, we take the value from the CPU register, so that way it so one memory, memory access is set. So if I have got multiple access is to  $x$  in the program so and if I have a register that holds the value of  $x$  then that should be so will save so many memory accesses.

But the point is that the register that we are putting here remains reserved for the entire sequence. So, whether that is really the case or the register value can get modified due to some program execution sequence, occurrence of interrupts etcetera, so that may make this transformation unsafe ok so, that the program may not work correctly under those situations. So, that is what it says that the transformation should be safe. So, we should do optimize, but optimization should be safe.

Then the trade-off between the time spent to optimize a program and that time improvement in the execution time, so that is that trade-off is necessary. But this so the amount of time to optimize a program and this if you want to do more optimization, it will take more time, but it will lead to more improvement in the execution time.

Then there are several levels of optimizations are used. So you can have as I said that their, their, their different levels we can do optimization up to different extents. So, you can have different levels of optimization. And as we go deeper and deeper or higher and higher level of optimization, the compensation time increases, but the amount of optimization that we get the efficiency of the code that also improves. If we are doing some debugging if we selected debugging option, so that may divisible any optimization that disturbs the correspondence between the source program and object code, because optimization may change the sequence of instruction execution. So naturally if you have

your selecting debugging mode or debugging option, then the all the optimizations are to be turned off, so that is there. So these are the various issues that we have.

(Refer Slide Time: 22:57)

Runtime Environment

- Deals with creating space for parameters and local variables
- For languages like FORTRAN, it is static – fixed memory locations created for them
- Not suitable for languages supporting recursion
- To support recursion, stack frames are used to hold variables and parameters

$x = f(a_1, a_2)$   
function  $f(r_1, r_2)$   
int  $b_1, b_2$

FREE ONLINE EDUCATION  
swayam

Then the runtime environment. So deals with creating space for parameters and local variables. So, like say if I have got a piece of program, and at some point I am calling a function f with say write a x equal to f with some parameters say a 1, a 2 etcetera. Then in the function f in the function f I may have some more local variables. So, this is r 1 and r 2, these are the two arguments.

And maybe I have got some local variables I n t b 1, b 2 etcetera. Now, where are these a 1, a 2, r 1, r 2, b 1, b 2 stored that is the decision that we take in the runtime environment management, so that will be the reserving space for parameters and local variables. So, for languages like FORTRAN, so this is done statically. So we have got fixed in memory address memory locations were created for them from where it will take up all these values ok.

So, this is this makes the compiler design process simple, but at the same time it creates difficulty if you are going to support recursion, because recursion the same program may be called again. So the previous locations which we are reserved for r 1, r 2, b 1, b 2, a 1, a 2 etcetera they will get over written by the new call and the new creation of the next instance of the call to the function f.

So this makes it difficult for supporting recursion. For recursion supporting, so we have to have some sort of stack frames, so which are more complex in nature compared to this is static runtime environment management. So we can have to go for that. So when you go to the runtime environment management techniques, you will see that they are very efficient technique, that have been developed for handling this recursion.

(Refer Slide Time: 25:05)

The slide has a yellow header bar with the title 'Speed of Compilation'. Below the title is a bulleted list of four points:

- An important criteria to judge the acceptability of a compiler to the user community
- Initial phase of program development contains lots of bugs, hence quick compilation may be the objective, rather than optimized code
- Towards the final stages, execution efficiency becomes the prime concern, more compilation time may be afforded to optimize the machine code significantly

The footer of the slide features the 'SWAYAM' logo and the number '31'.

So next we have got a speed of compilation. So this is another important criteria to judge the acceptability of a compiler to the user community that is how fast is the compiler working. So it is a flash it gives the output is the give the compiled output or it takes lot of time. So that is that way it is difficult ok. So, initial phase of program development, it contains lots of bugs, hence quick compilation may be the objective rather than optimized code. So they are also most of mostly we have got syntax error, then this logical bugs and all. S, there is no point doing optimization there.

So, at that that time I should have an option by which I can tell the compiler see I do not want any optimization I want the compiled code fast ok so that should be generated quite fast or if there is any syntax error that should be reported fast. So, that parts the initial phase of program development. So we are we are will be looking for the compiler to work with the high speed of compilation maybe we forgo the optimization part.

But when you go to the final towards the final stages execution efficiency becomes the prime concerned, because there we are ready to spend more time, but we do not want the

compile that to the code generated to be inefficient, so that the code has to be very efficient. So we want to spend more time on the compilation process, but we want that the output or the outcome of the compilation, it should be very efficient, the code generated should be very efficient. So this is towards the final stage of compilation process we want optimized code.

So for the same compiler sometimes we want that it should be running very fast, sometimes will be we will want that it should be not it need not be that much fast, but it should generate very efficient code, so that is these are the issues that we have with a compiler. So the compiler designer should look into these angles. So, not only the code generation process, but this entire other issues that to be looked into and that makes the compiler design process complex.

So rather than a say if you do not bother about all this concerns, then it is pretty easy. So we have got automated tools by which we can generate the code for some for some processor, but as soon as you bring all this concerns into your purview so this entire code generation process becomes difficult.

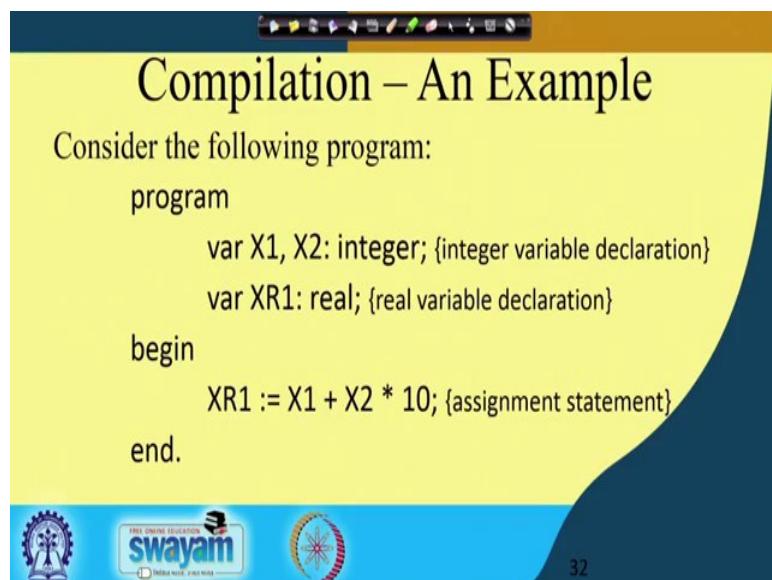
So in our course, so we will try to look into these aspects again and again in different chapters and that way it will go. So at the end we should be able to we should be able to design compilers within reasonable amount of time and taking care of all these issues.

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

**Lecture – 06**  
**Introduction (Contd.)**

Next will be looking into an example and with that example, we will try to explain this Compilation Process, how it processes, how it goes, and what are the outputs of various stages.

(Refer Slide Time: 00:25)



```
program
  var X1, X2: integer; {integer variable declaration}
  var XR1: real; {real variable declaration}
begin
  XR1 := X1 + X2 * 10; {assignment statement}
end.
```

So we take an example from of a program which is close to ah the language called Pascal, ok. So anyway it does not matter. So, the essentially this programs in that language starts with the keyword program, ok. Then comes the variable declarations, and each variable declaration is preceded by the keyword var, ok. So, we have got var X1, X2 then colon integer it says that X1 and X2 they are two variables of type integer. Then we have got XR 1 as a real variable. So you can have this var XR 1 real. So, these are the variable declaration.

So once the variable declaration is over, so we can start the main part of the program like in see language we have got this main function. So, in Pascal type of, Pascal language that we have considering here, so after program this program keyword so whenever you

are given this begin keyword it means that we are going to start writing the main body. So, the execution we will start from this point onwards.

So, here in this main body, so we have got a single statement `XR1 := X1 + X2` assigned as. So, this is the assignment operator colon and equality. So, `X1` plus `X2` into `10`. So, that is an assignment statement. And the program ends are this point, so this end is identified by end dot. So, this is the full program. So, this is the very simple program that we have and that is the whole and that is the whole program that we have taken. So, we will see what the outputs of various stages of the compiler are as it processes this particular input.

(Refer Slide Time: 02:08)

The slide has a yellow header with the title 'Lexical Analysis'. Below the title, there are two bullet points:

- Produces the following sequence of tokens:  
`program, var, X1, ',', X2, ':', integer, '/', var, XR1, '.', real, '.', begin, XR1,  
'=':, X1, '+', X2, '*', 10, '.', end, ''`
- Whitespaces and comments are discarded and removed by the Lexical Analyzer

The footer of the slide features the 'SWAYAM' logo and other educational icons. A small video window in the bottom right corner shows a person speaking.

So first stage of the compiler is the lexical analysis. So lexical analysis stage as I said that it will remove all this white spaces like this blanks, tabs, new line characters then the comments.

(Refer Slide Time: 02:23)

Compilation – An Example

Consider the following program:

```
program
    var X1, X2: integer; {integer variable declaration}
    var XR1: real; {real variable declaration}
begin
    XR1 := X1 + X2 * 10; {assignment statement}
end.
```

32

So if you look here, so these are the comment. So whatever we put within these curly brace. So they are all comment. So all these are removed. Similarly these new line characters, then these tabs, then these blanks, they will all be removed.

Ideally I can write this entire program in a single line, by just demarcating each word by a single blank space. But we can write in a fancy way also as we have done it here and the lexical analysis tool is suppose to if I remove all those cosmetic from the program.

(Refer Slide Time: 02:58)

## Speed of Compilation

- An important criteria to judge the acceptability of a compiler to the user community
- Initial phase of program development contains lots of bugs, hence quick compilation may be the objective, rather than optimized code
- Towards the final stages, execution efficiency becomes the prime concern, more compilation time may be afforded to optimize the machine code significantly

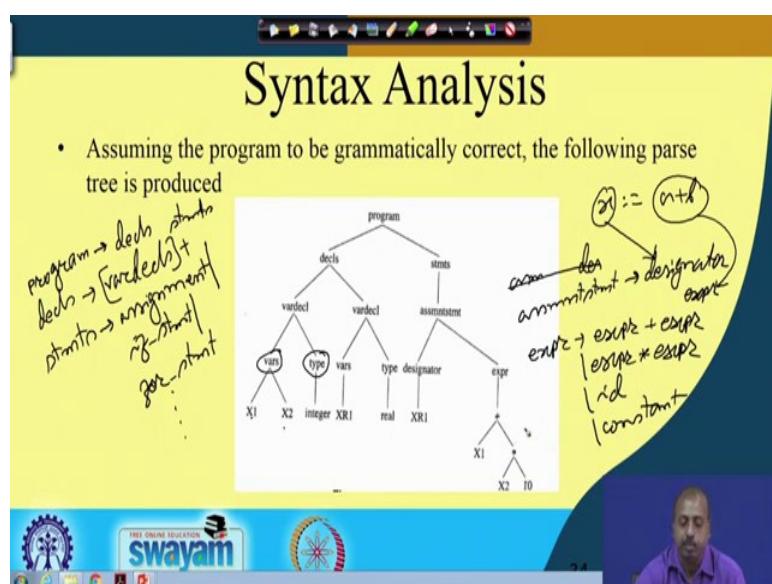
31

So, this is lexical analysis tool after analyzing the program. So, it produces a sequence of tokens. Like, so for the keyword program it outputs the token program. So, actually as I said that this, tokens are nothing, but some integer values. So, for the sake of our understanding we are not writing in terms of integer values, but we can assume that all these words like program, var, so they are having some corresponding integer value.

So, the lexical analyzer is actually passing those integers to the syntax analyzer or parser, but for our understanding we will write them as in terms of this tokens only program, var etcetera. So, this program token is returned. So, when it sees that this keyword program it returns the token program, then these were then X1, then comma, then these X2, then these colon, then these integer, then semi colon, var, XR 1, colon, real, colon, semi colon, begin, XR 1. So, these are the various tokens. So, when whenever the parser asked the lexical analysis tool to give the next token it gives one of these tokens which comes next, ok.

So, whitespaces and comments are discarded and removed by the lexical analyzer. So, that we have discussed previously that this all this whitespaces the cosmetics think that are removed from the program. So, at the output of the lexical analysis is nothing but this sequence of tokens, and the sequence that token is returned to the parser only when the parser asks for the token.

(Refer Slide Time: 04:44)



Now, coming to the syntax analysis phase. So, we will assume that the program, if we assume that the program is grammatically correct. So, it will generate a parse tree. So, I said that parse tree is a 3 type of representation that tells how the program can, how this entire program can be taken to, have been derived. So, this whole program is divide is this is actually depicted more by the grammar rule that we have in the program which you for the language which is not shown in our example, but assuming that I have a rule which says that the program can produce declarations and statements.

So, there is a rule which says that program is can produce declarations and statements. So, first whenever it sees the program keyword it divides it into portion declarations and statements. Similarly there is another rule in the language which says the declarations can be a number of variable declarations. So, we can have a number of variable declarations, ok. So, this is so when you come to this parser chapter. So, we will be discussing more on the grammar how to write it and all. So, it is like this.

Similarly the statements that we have statements can be of different types, like assignment is one statement. Then we can have say if-then-else, if statement is another state type of statement, then for statement is another type of statement. So, this way we can have different types of statement. So, that is; so we have got rule in the grammar which goes like this.

Then the one variable declaration it has got this is the list of variables. So, that is identified by vars and then the type. Similarly, this is the X1, X2, so this list of variables are identified by this is identifiers ended with a colon, ok. So, this that entire thing is retained here, so X1, X2, so integer. So, this way the the entire program that we have it can be represented in the form of a tree.

So, assignment statements, it has got a designator and expression. So, the thus designator the rule is like this that assignment statement, so this assignment statement it has got a designator and expression. So, if I have got say X assign to some a plus b then this X is the designator part and this plus b is the expression part. So, we have got an got an assignment to the variable XR 1. So, the XR 1 is the designator identifier and this is the expression that we have is X1 plus X2 into 10. So, this expression has got its own grammar, say expression is expression plus expression or expression star expression or it can be an identifier or it can be some constant.

So, here it is divided like this. This expression is expression plus expression like that, and then the right hand side is again expression multiplied by expression. Then the expression the light left hand side it gives me the identifier expression on the right hand side gives me the constant. So, this way we can have different we can we can generate the entire parse tree.

So if the program is syntactically correct then this syntax analysis tool. So this will generate a parse tree. So this parse trees is used by the code generation stage for generating the generating code for the correct program. And if the program was syntactically wrong then you this parse tree could not be generated by using the rules of the grammar, and accordingly the parser will give some syntax errors and at the other compiler designer can flash appropriate syntax error messages for the program, for the input now. So assuming that the program is syntactically correct, so it generates this particular program.

(Refer Slide Time: 09:24)

## Code Generation

- Assuming a stack oriented machine with instructions like PUSH, ADD, MULT, STORE, the code looks like

```

PUSH X2
PUSH 10
MULT
PUSH X1
ADD
PUSH @XR1  @ symbol returns the address of a variable
STORE

```

The diagram illustrates the state of the stack during the code generation process. It shows three frames on the stack. The bottom frame contains the value of X1. The middle frame contains the result of X1 multiplied by 10. The top frame contains the constant value 10. Arrows indicate the flow of data from the variables X1 and X2 into the stack frames.

Now, the code generation. So assuming that we are generating code for some stack based machine with the instructions like PUSH, ADD, multiplies STORE etcetera the code will look something like this. First I need to evaluate this expression in the, so the code is to evaluate the expression that is XR 1 equal to X1 plus X2 sorry, X1 plus X2 into 10. So, it first pushes this X2 into the stack. Actually the way it operates is like this. So, first it first

it has to evaluate this X2 into 10, and for doing that it has to PUSH X2 and PUSH 10 and then call the multiply operation.

So, as the stack based machine, you know that it pops out the two top most interest from the stack. So, in the execution what will happen is that sorry in the execution what will happen is that we have got in the stack first X2 pushed and then 10 pushed then this is multiply instruction comes accordingly the machine while executing it, it will pop out this stain, and X2 from the stack, do the multiplication and PUSH the result back to the stack. So, now, after doing this the stack will after this PUSH statement has been executed the stack will have X1, sorry it will have the a value of X2 into 10.

Then it says PUSH X1, so X1 is pushed into the stack and then it does ADD. So, when it does ADD, so again these two are popped out, the values are added and then; return value also this value has to be so now, my stack has got this result. So, this is X1 plus X2 into 10, X1 plus X2 into 10. So, this value is there in the stack. Then in the stack we PUSH the address of XR 1. So, the address of XR 1 is pushed into the stack and then we say STORE. So, when we say STORE it will again pop out to top most entrees the first entry will give me the address of the location where you want to store the value and the second entry gives me the actual value. So, based on that, it will be, it will be storing the value of this expression on to the variable XR 1.

So, first stack based machine. So, it will generate code like this what for some other machines, it will generate code in a different fashion. So, that explains the compilation process. So, starting with the program lexical analyzer we will identify the tokens. Parser will arrange those tokens in some fashion, so that it becomes the derivable program from the start symbol of the grammar. And if it can make that parse tree then this code generation face it can generate code by taking help of the parse tree the individual stages of the parse tree it can use and generate the code for the target machine. So, this way this compiler works.

(Refer Slide Time: 12:26)

The screenshot shows a presentation slide titled "Symbol Table". The slide has a yellow header and a blue footer. In the footer, there are logos for IIT Kharagpur, SWAYAM, and a circular emblem. The number "36" is also visible in the footer. The main content is a table with three rows:

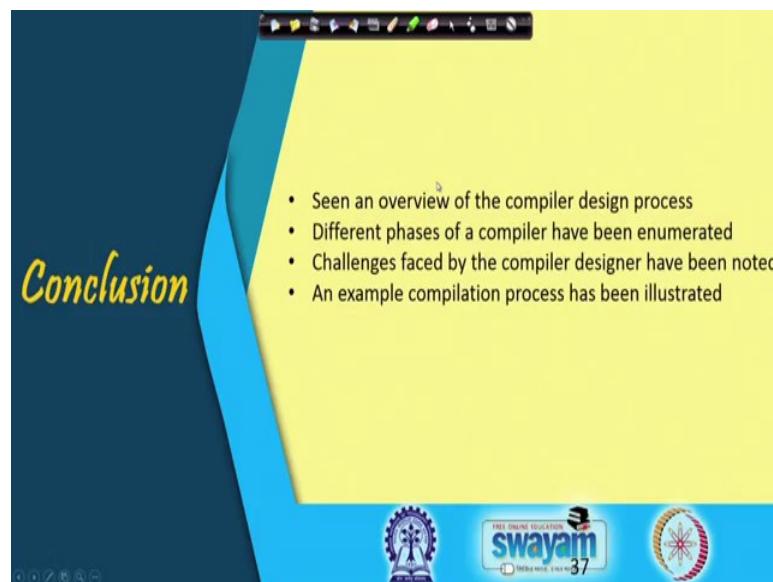
Name	Class	Type
X1	Variable	Integer
X2	Variable	Integer
XR1	Variable	Real

Then the symbol table as I said that there are three symbols in this program X1, X2 and XR 1. So, their class, the class is all of them are of type vary or of class variable X1 and X2 are of type integer and this XR 1 is of type real. So, that way we can have this X1, X2 and XR 1. So, there of time integer (Refer Time: 12:50) though.

Another field can be there which the offset which is not shown here is. I have not shown it explicable, purposefully because just to emphasize that it is not mandatory that all those information with their, all symbol tables be similar, so it can vary. So of to the compiler designer, compiler designer may feel that certain information be there in the symbol table certain information may not be there. So in this case the compiler designer might have thought that these are very simple. So I do not need to, I do not need the offset. So, what we are doing for this stack based machine? So we are pushing them on to the stack. So as a result we do not need the offset values. So this may be the symbol table structure.

Then after that we have got this code generation is done. So, the symbol table is done.

(Refer Slide Time: 13:35)



**Conclusion**

- Seen an overview of the compiler design process
- Different phases of a compiler have been enumerated
- Challenges faced by the compiler designer have been noted
- An example compilation process has been illustrated

• IIT Madras • FREE ONLINE EDUCATION SWAYAM • ISRO

So with this example we will like to conclude this part of our discussion. So, in this discussion we have seen an overview of the compiler design process. We have seen different phases of a compiler, how are they, what are they. So we have enumerated the phases we have also discussed in detail the challenges faced by the compiler designer. So as a successful compiler designer we have to take care of all those points, and we have also illustrated by means of an example compilation process.

So with that I will thank you. And in the next class, we will go to the lexical analysis phase and start in more detail like how to design a lexical analyzer for a compiler.

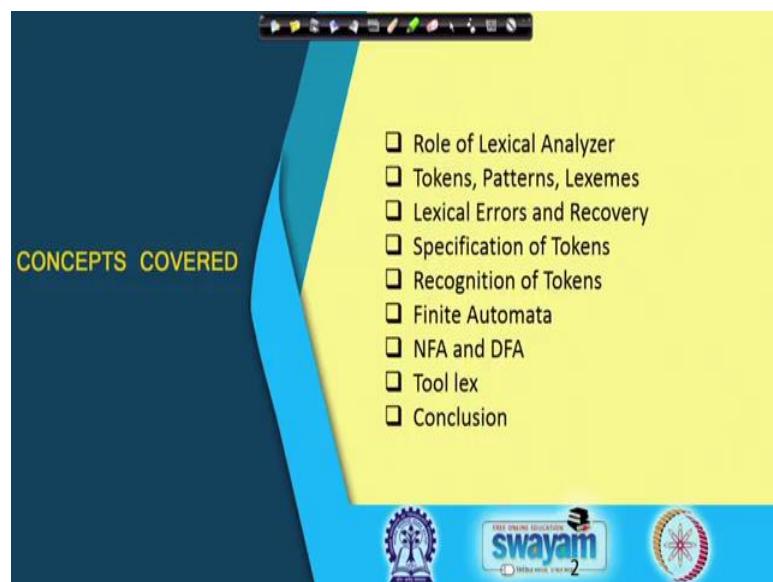
**Compiler Design**  
**Prof. Santanu Chattopadhyay**  
**Department of E & EC Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture – 07**  
**Lexical Analysis**

So today we will start with the first phase of a compiler which is known as the lexical analysis phase. So the task of the lexical analysis phase is to identify the valid words that we have in the input program for the language that you are considering. Now the major challenge that we have here is that, there may be different types of formatting and that formatting is often user specific. Like, one user maybe putting 2 blanks after every word, another user maybe putting one tab after every word.

So, like that the formatting is up to the user. And very often for a to a programmer we suggest that you should do a good amount of indentation so that the code is easily legible and all. So, this type of artistic things that are introduced in the program so, they are of no meaning when it comes to the machine code. So, they are to be handled by the first phase of the compiler, so which is no which the lexical analysis phase is. So, here we will look into that lexical analysis phase in detail.

(Refer Slide Time: 01:21)



The topics that we are going to cover a like this; first we will discuss about the role of this lexical analyzer. Then in the context of lexical analysis we will introduced some

definitions tokens patterns and lexemes. So, they will be used throughout our codes to determine to define the data items that are exchanged between different phases of the compiler.

So, these tokens are identified and generated by the lexical analysis phase. Then there are some lexical errors that may occur like, maybe some word which is least spelt. Like say somebody while writing say if if writes f i fi. So, were lexical analysis tool it can identify that type of mistakes and it can correct those errors, or at least suggest that possibly the user has user wanted to write f a if, but has written f i.

So, that way the user can be given some feedback. And not only identifying the error sometimes we need to recover from the error also. Like for example, getting the character f, so the lexical analysis tool will wait to see the next character. So, next character it finds i, then it understands ok f i is some error then it needs to convert it into i f, and then introduce those characters all to the input stream by replacing the original characters f i by that.

So, this way there maybe some error recovery scheme into that the compiler designer can think about and accordingly introduced into the compiler. Then how were these tokens specified? So, that is one important thing. And there can be different rules in the programming language that will tell us like how can we define different tokens. For example, the identifiers in a programming language it has got a particular format. If you considered say the URL of different websites so, it has got a particular format. So, that format or if I consider that as that u entire URL as a single word, that word structure is definitely different from the variable declaration stay that we have in some programming language.

So, that way we should be able to specify the desired tokens very clearly and that should be very unique. And second thing is that after the specification phase has been done, we have to have some recognizer for that. So, specification phase. So, it will define a set of rules by which the tokens will be defined, and in the recognition phase actually the lexical analysis tool it will scan the input program to see what are the tokens that are appearing in the, what are the words that are appearing in the program, and accordingly return corresponding tokens to the person.

Now, for the recognition part, so we will see the tool finite automata it is. So, the as I said that entire course hinges heavily on the automata theory. So, we will be using finite automata for to act as a tool that can identify the tokens that are there in the input program and to generate the words that are there in the input program and generate the corresponding tokens. And in there finite automata category so, we will look into 2 variants of for the finite automata that are used by the lexical analysis phase.

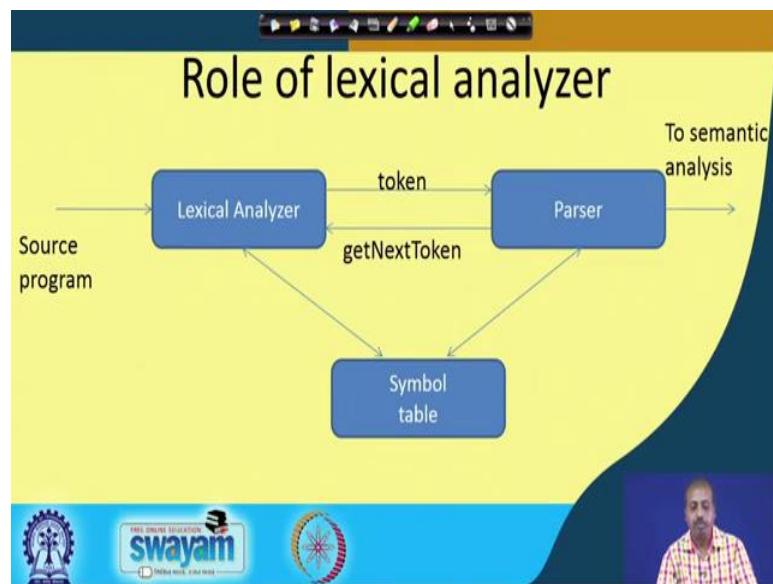
One is known as Non-deterministic Finite Automata or NFA, another is known as Deterministic Finite Automata or DFA. Both of them are equally powerful; however, specification strategy for NFA and DFA is slightly different. In general, we can find that NFA's are, NFA's will have less number of states compared to DFA, but to visualize a DFA maybe it is easier for us. But if we can visualize the NFA then we can get more compact representation for the token.

So, initially it seems that the DFA is the better one, but after some practice you will find that it is easier to construct NFA then to construct a DFA. And DFA often may often have large number of stakes compared to NFA. So, complexity of DFA maybe more, though their capabilities are same NFA and DFA they are perfectly equivalent to each other. Then we will look into one tool that was originally introduced with the Unix operating system known as lex.

So, which is given a specification, so it can generate a lexical analyzer program for some language. So, foreign language, so you can define it is the word the rules defining the words of the language and then feed it to the lex tool. And the lex tool will come up with a program lex dot y y dot c, which is a c program which can be used by the compiler designer to automatically generate the lexical analysis part. So, you do not need to write explicit code for the lexical analysis tool. Because the language maybe very complex the input programming language maybe very complex.

So, writing the corresponding lexical analyzer from scratch may be difficult. So, this tool actually helps us, and it has become so popular that nobody now nobody thinks in a different way apart from writing the lex specification and generating the lexical analyzer automatically. Finally, we will conclude the discussion.

(Refer Slide Time: 07:17)



So, start with the role of lexical analyzer. So, as you see this tool takes the input source program of some language and it will scan through this program and accordingly it will generate tokens. And as I said previously that this lexical analyzer and the parts are they work hand in hand. And they are they, so the demarcation line between the two is very thin ok.

So, the way it operates is that this parts are whenever it requires some further token, it will request the lexical analyzer tool to give it the token, and give it the next token. So, intern the lexical analysis tool, it will scan the input file wherever it was the input program wherever it was scanning previously from that point onwards. And accordingly it will identify the next valid word that appears in the input program of the language, and return it, returns the corresponding token to the person. So, what is token? So, we will come very shortly to the definition. For the time being you can just take it as if this is some quantity which corresponds to the valid words of the programming language and each valid word has got a different token identifier.

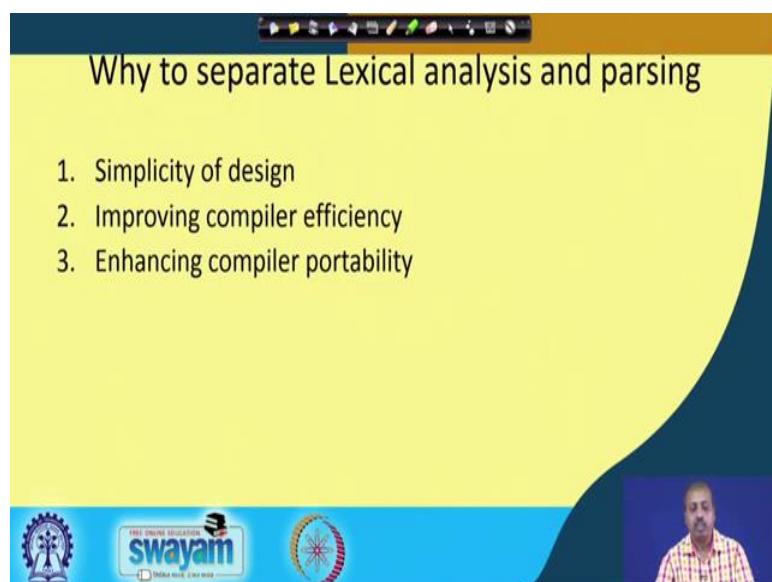
Now, these parts are intern. So, it gets the tokens and accordingly it understands the it checks whether the program is syntactically correct or not; the entire program. So, this lexical analyzer tool is giving it words, and the parts are is actually checking for sentences. So, if the sentence sentences are correct, then it will give it to the semantic analysis tool which will do the type checking, which will do say your other associated

actions meaning of the program it will try to identify, and then it may go to code generation phase.

So, both lexical analyzer and parser, they utilize this symbol table data structure. So, as I say previously, that symbol table stores all the identifiers that we have in the program. And so, this lexical analyzer whenever it finds a new identifier, it informs the symbol table it puts it into the symbol table, telling that this is the next symbol that we have, and it is put into the symbol table.

And the parser whenever it needs the reference to any symbol, so it will refer to the symbol table. So, who will populate the symbol table? So, that is often the choice of the compiler designer, sometimes it is done by some compiler designer will put the job on to the lexical analysis phase, some will put on the parser phase depending on the ease of programming and type of language. So, different approaches are taken by different compiler designers.

(Refer Slide Time: 10:11)



Now, as I said that this lexical analysis phase and the parsing phase they are working hand in hand. So why should we separate them into 2 different phases, why not design the entire thing together? So there are 3 basic reasons for doing it, first of all simplicity of design. So actually this whole compiler can be designed as a monolithic software, monolithic piece of program, where starting from the lexical analysis till code generation everything or even the code optimization.

So everything can be done together, but that will make the whole program very complex. So just to avoid it just to avoid that complexity so we have got simplicity of design so that this lexical analysis phase if it is separated, then some portion or some action, so they are separated into a different phase.

So designing that phase become easier for the compiler designer. It improves compiler efficiency because now these later phases of the compiler they need not work with the input stream or the input character strings that I have in the input program. So, they can be working with some tokens, and we will see very shortly the tokens are each token is given some integer identifier. So, we can say that the later part of the program of the compiler that the compiler that is designed, so they will be working with integers. And as you know if we have a program that does number clenching that is often much simpler to design compared to program that uses character strings and some manipulation around them.

So, that way this compiler efficiency improvement so, this is better if we have got this lexical analysis on a separate phase. It also enhances compiler portability. So, from one system to other systems if we want to port the compiler, then take the compiler to some other system then the formatting the input format may be slightly different on to the target system. So, they are so those things can be taken care of by the lexical analysis phase itself. So, the later phases of the compiler they can be kept as it is.

Only the front end part so that needs some changes and it can be done. May be from one for the same for the same language if we go from 1 platform to another platform, may be the syntax will the way in which words are defined may be slightly different. And those things can be taken care of by the lexical analysis phase itself.

(Refer Slide Time: 12:55)

The slide has a yellow header with the title 'Tokens, Patterns and Lexemes'. Below the title is a bulleted list of definitions. Handwritten notes are overlaid on the slide:

- A token is a pair – a token name and an optional token value.  
Annotations: 'name' is circled, 'token value' is circled and labeled 'attribute', 'char, numbers' is written next to it, and '(1234)' is circled and labeled 'value'.
- A pattern is a description of the form that the lexemes of a token may take.  
Annotation: '(N234)' is circled.
- A lexeme is a sequence of characters in the source program that matches the pattern for a token.  
Annotation: 'VAR' is circled and followed by an arrow pointing to '(45)'.

At the bottom of the slide, there is a blue footer bar with the 'swayam' logo and other icons. A small video window in the bottom right corner shows a man speaking.

Coming to the definition of tokens, patterns and lexemes; a token is a pair, a token name and an optional value. So, every token is given in name and it has got some value. The value is integer, and most often this token name and that we have, so it is given as it is also, given as integer and this token value part. So, it may be consisting of it may be consisting of some attribute of the token. A pattern in turn, so if pattern is a description of the form of a, form that the lexemes of a token may take. Whereas, a lexeme is a sequence of characters in the source program that matches the pattern of a token.

So, this lexeme is, so lexeme is the input string that is there that is read from the input file. And we have got some patterns that are valid for the program. For example, if I say that my identifier in my programming language is like this. Any sequence of character and any sequence of character and numbers, any sequence of characters and numbers, but the first symbol there must be a character. So, if I have got something like s 1 2 3 4. So, this is a variable in that as per my definition.

Then say, but say 1 s, so this is not a variable, this is not a variable. So, in my input string in the input program if I get s 1 2 3 4 so, this is the lexeme that you are getting. So, this is the lexeme, a sequence of character in the source program. And the pattern is I am telling I will say how do I specify this pattern, but pattern is actually telling a rule by which I can define what are valid a lexemes. Like here I am telling that any sequence of

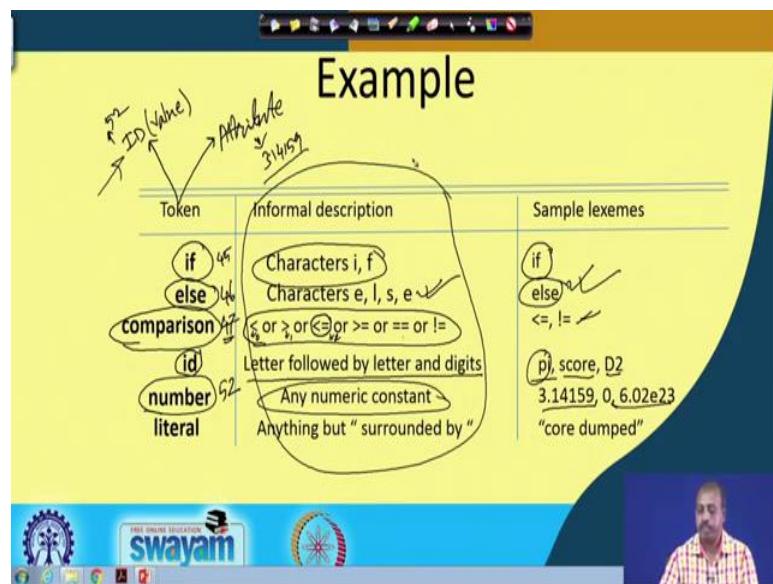
characters such that the characters and numbers such that first symbol is a character is we will define a variable.

So, any so as per my as per my definitions so, this is a valid definition valid variable, but this is not a valid variable. So, this is the pattern, so how do we write a pattern that will see a shortly, but somehow it will capture this rule. And now after getting this lexeme, matching with this particular pattern, which pattern is called the variable. So, after this lexeme s 1 2 3 4 matches with the pattern variable, so what is returned is the token.

So, this token name, so this is often taken as integer and this there can be an optional token value; so, which is often called attributes also. So, each token has got a value and a or a name and a value or attribute. So, in the compiler designer may give that variable the corresponding number may be say 45.

So, on getting this s 1 2 3 4 on the input string; since, it is matching the pattern of character number sequence for variable, so the compiler the lexical analyzer will return this token name as 45, and as a value it can return the symbol table location for the variable s 1 2 3 4. So, this way depending upon the token that we have we can have a value that is in our name and attribute or token value. So, that way it can that that pair can be there, so it can be token name and token attribute, and name is often it is an integer. But that is not mandatory some compiler designer may use token names as some other format also may not be integer. But in general, it is taken as integer. And the next tool that I am talking about, so that also takes it as integer.

(Refer Slide Time: 17:23)



So, coming to the some coming to some example like say this say this particular pattern, so characters i and f. So, this is a pattern. So, this is the lexeme, so that is appearing in my input string. And according to the token that is returned is if. So, though it is written as a character string i f, but you can understand that these what I want to mean a something different than this one.

Similarly, when it is say characters e l s e, the formal informal description of the lexeme is e l s e these 4 characters coming together. The sample lexeme is else e l s e that. So, these thing appears on the input file. So, this matches with this particular rule that is a character sequence e l s e. So, this matches with the, so this input sequence matches with this particular pattern, and then it will return this as the token.

And as I said that the may be, so all these tokens they are given some numbers. So, this may be say 45, this may be 46. So, like that randomly some values, some integer values unique integer values may be assigned. Similarly, so say all the symbols less than greater than less or equal greater or equal equality and not equality. So, these are this defines another rule ok. So, this defines another rule or another pattern, and some sample lexemes that appears in the input program maybe this less or equal, or not equal to, so these are these are appearing on the input file as sample lexeme.

And now the token that is returned is comparison. So, this comparison token may be given the number say 47 and the lexical analysis tool. So, when the parts are asked for

the token, if it gets on the input file less or equal the lexical analysis tool will return the value 47 to the parser. But that is not sufficient because I need to tell which comparison operator I am talking about.

So, accordingly it can have some attribute part the token can have some attribute part, and there somehow we tell that I am talking about this symbol. Maybe each of them is again given a code. So, this is 0, this is 1, this is 2, so like that each of them may be given a code. And that code may be returned as the attribute of the token comparison. Similarly, so for say for identifier, the rule maybe later followed by letter or digits. So, this is the rule, so this is the pattern that I am talking about. So, this lexemes that appears in the source program may be pi score D 2. So, these are some character strings appearing on the input program.

So, since the for lexical analysis tool, when it finds p i, pi. So, it find that there is no rule that defines lexeme the corresponding pattern pi, but this particular rule matches ok. So, this sells it because matches because it starts with the letter and it is it continues with letters or numbers letters or digits, so this part matches. So, it will return the token id, but it has to do something more, because this pi most probably pi is a variable. So, it will you put it into the symbol table. And as an attribute part, so it may return the symbol table offset to be used by the parser.

Some numbers, so this is a 3.14, so 14159, so this appears in the input program or say something like this. So, the rule that I have is any numeric constant. So, this is the pattern ok, and accordingly the token returned is number. So, here also token is number, but it the token does not talk about the value that is actually worked in there. So, this token has got 2 parts as I said. It has got a identity the token id and the attribute ok. So, token id or token value and then attribute part. So, this id is suppose this here the corresponding token number that is given is say 52.

So, this id value is 52 and in the attribute part we return the actual value. So, maybe 3.14159 so that parser if necessary, so it will it will it will check the grammar rule whether for checking the grammar rule it will look into the token id, but for doing some code generation and all maybe it will need this attribute value; so that also it can do.

So, this is how this parser and this lexical analyzer will work together. So, literal is anything that surrounded by double core. Anything, but is surrounded by this double

code symbols like core dumped. So, this is sample lexeme, so this is a literal. So, this way this lexical analysis tool so, it will be looking into the rules that we have. So, we have not yet seen how to capture this informal description part by means of pattern rules. So, that we are going to see very shortly. Once that is done, so this for the language I can define a set of rules that can define, that can guide valid lexemes that are there.

Now, the lexical analysis tool will scan the input file and it will identify lexemes that are occurring in the program, and accordingly it will return the corresponding token to the parsing phase. So, this is how this whole thing will work.

(Refer Slide Time: 23:21)

Like say when I talking about attributes of tokens. Suppose I have got this with this particular the string appears in my program E equal to MC square. So, E it appears in my program M into C then 2 star, then 2 this whole thing appears in the input program somewhere.

So, lexical analysis tool, so it will start scanning from. So, the when the parser as asked it to return next token at that point the lexical analysis tool might have seen up to this much in the program. So, it has seen up to this much. So, it scans the input and it finds the symbol E and after that there is equality. So, if it scans through the valid lexeme rules where valid pattern rules that are there in the program, that are there in the language and

it finds that E it can be an identifier. So, it will return the token id and as the value part or attribute part. So, it will return pointer to the symbol table entry e.

So the variable e must have been defined previously in the program. So, accordingly when that point it if the variable e was seen, so it was put into the symbol table now sometime later when this line e equal to MC square comes, then the lexical analysis tool gets the symbol e, and it identify e and it can search through the symbol table find out what is the offset of e in the symbol table and it can return that pointer to symbol table for e to the parser. So, this is the token value and this is the token attribute.

Now, it finds, so next time the parser asks the lexical analyzer to return the next token. So, at that time it will find this equality as the next token. And equality is the token the corresponding token name is assigned of. So, this it returns the token assigned of. So, we do not need any other attribute for this particular token. So, no attribute is mentioned. Next when it is asked, so it will find this M. So, it will return id and pointer to symbol table entry for M. Then it finds next token is star, so it will return the multiplication operator, then it will be finding the next time it will find this c, it will return the token as id and pointer to symbol table entry for C. Then it will find 2 stars, 1 star, 2 star

Now, it can, so what, so here there is a confusion, because here on getting one star it was it has returned mult op as the token, but here getting the first getting the first star itself, it could have returned mult op to the parser. But it does not do that, it actually what the lexical analysis tool does is that it finds the maximum match. So, there is a rule in my language which says that the if you get a single star that is a multiplication, but if you get 2 stars that is an exponentiation. So, this rule is there in my language.

So, the lexical analysis tool, so it will try to find maximum match at this point and it will find that the maximum match is this one not this one. Whereas, at other places the maximum match was this single star. So, at this point when I was looking for maximum match, so this was a single star. Because star C does not have any meaning for the programming language; so that is considered here.

So, naturally this here it is a maximum match, so star star it matches. So, it will return the exponential operator. And then this 2 is a number, now the as I said that it will have an attribute, so attribute value is integer value 2.

Now, you look into these 3 places, so here also it is returning id. So, here also it is returning id here also it is returning id. At all the 3 places it is returning the same token, ok. If this token id is given the number say 45. So, at all the 3 places it is returning the value 45 to the parser. But as an attribute it is differentiating between them. In the first place it is returning pointer to E, second place pointer to M and third place pointer to C. So, all these information are required like as far as the grammar checking is concerned, so these pointers are not necessary. But if you are trying to check what are the types of this variables, and if you are trying to generate code for that, then you need to have the corresponding symbol table pointers, ok.

So, for basic syntax checking it is not needed. But for later stages of this code generation and all, so these pointers will be useful. So, that is why this lexical analysis tool it not only identifies the words and returns are corresponding tokens, it also identify some attributes that may be meaningful for that particular word and returns them to the parser. Similarly, when it is considering this number so, it is returning the value of the number as 2. So, this is done here. So, this way all these tokens, so they have got some attributes and these attributes are returned to the parsing phase.

(Refer Slide Time: 29:05)

- Some errors are out of power of lexical analyzer to recognize:  
- a == f(x)) ...
- However it may be able to recognize errors like:  
- d = 2r
- Such errors are recognized when no pattern for tokens matches a character sequence

There can be lexical errors, so lexical errors as the situation is such that errors are out of power of lexical analyzer to recognize. So, this like say this one. Say some errors the tool

cannot identify, like say this one in some c language program we have written something like this.

Now, so we it cannot identify the situation because this fi; so ideally it seems that fi should be if, ok. But it should not do it automatically, because it may so happen that user has defined a function whose name is fi. And this a equal to equal to fx is an expression. So, value of this expression is parsed as a parameter to the function fi, and that that will be evaluating the function fi.

So this has got some other possible meaning also but it is, but actually if this is made if then that is the most probably that is the correct one. But the other one the existing one that fi and within bracket a equal to equal to fx, so that is also may be correct.

So we have got confusion. So the lexical analysis tool cannot handle this type of situation this type of errors; however, it can identify this as an error. Because in my programming language for the identifiers it is said that it should start with a symbol, it should start with a character and then it should continue with characters or digits.

Now, this, so no symbol can start with a 2. So this is not going to be a symbol. So, it, but at the same time it cannot take 2 as a number, because if it is a number when after that between 2 and r there should be an operator. So a number must be it must be separated by at least one blank space. So that way the lexical analysis tool can identify an error at this point. And it can tell the user that there is an error at this point.

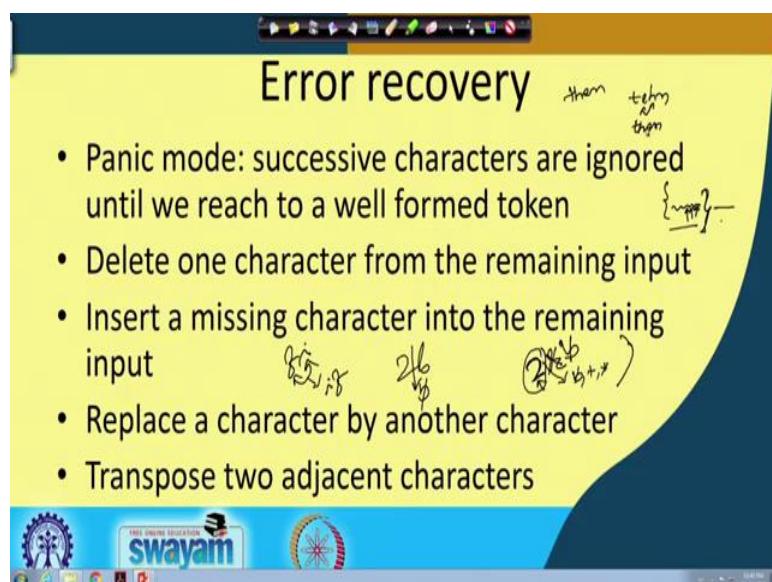
So such errors are recognized when no pattern for lexemes matches if character sequence. So if you do not find any match, then we can say that there are some there is some problem with this particular sequence of characters that are occurring, so that is an error. Another possibility is that there may be some foreign characters in my input string, and then the lexical analysis tool since it knows the alphabet, so it can inform that those foreign characters are there; so it is not in your language, so accordingly it can generate some error.

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

**Lecture – 08**  
**Lexical Analysis (Contd.)**

So if errors are there. So we can have we can think about some error recovery policy.

(Refer Slide Time: 00:19)



So, the problem with the lexical error is that, the input string input file that we have an input stream. So there is some problem in the input stream itself. So, if you are thinking about correction. So we have to correct the input stream and if there is no look back ok. So, lexical analysis tool so it is scanning the input file from one side and going towards other side. So, it has already scanned some portion which needs to be corrected.

So, the lexical analysis tool has to undo some operations that it has already done for correcting the text or doing some recovery. So, one possible recovery mode is the panic mode of recovery. So panic mode of recovery here it is said that a successive characters are ignored until we reach to a well formed token. So, some tokens are very unique for example, semicolon is a unique token, then say if you are considering its a C programming language then this symbols like open brace and closing brace so they are unique.

Similarly, if we get a closing parenthesis. So, that may be meaning end of an expression. So, like that certain tokens are certain character sequences are unique. So, if you find that there is some error so when the when it was going through this part. So, it can go on, it can go on ignoring all these characters still it gets a closing brace because after the closing brace it is I can assume that from this point a new correct token will start.

So, in a panic mode of recovery. So, it will ignore all these tokens and it will be starting a phrase with the next synchronizing token so or synchronizing symbol. So, this is good because I can the lexical analysis tool. So, it can come out of the error and continue processing of the remaining one. So, another possibility is that delete one character from the remaining input.

So, this is just one more one possibility that we you delete one character with the expectation that the from the next character I will get a valid one, as I was telling that the example that I took previously that its a 2 r. So, after getting 2 so, I was expecting. So, to return this as a number I expect that at this place I should have a blank or some operator say plus or multiplication some operator.

So, what I do? I just ignore the next character. So, maybe after r after this r there is a blank to if I delete this r when I will get this to as a valid token. So, this way I can delete one character from the remaining input and accordingly I can make a valid word out of the, whatever I have scanned or I can insert the missing character into the remaining input.

So, here also the same to same example like it is 2 b then I can insert one blank in between in that case this you will be returned as we have some number and after that we will b returned as an identifier. That will solve the problem for the lexical analysis tool. Now whether this is correct or not that is not the concern because it may or may not be correct, but the lexical analysis tool it got stuck at that this point.

So, it was unable to proceed further. So, if you do this then it will be able to proceed with the remaining input and maybe it will do something, maybe it will flash some error message and the user will correct it.

But if it can give a large number of error reported to the user otherwise it will stop at that point and it will come out from this compilation phase telling that there is a wrong, there

is an invalid character at this point patriotic only this much that is not there. Sometimes we replace a character by another character. So, that is also another recovery policy. So, if you so, there may be alternative words that are available. So, maybe one in some character in some words only one character is wrong.

So, what it does is that it modifies that word completely with a valid word. So, normally for this purpose we have got this text processing software which do this thing. So, they have got some dictionary of words and it whenever it finds that some word is coming which is not matching. So, will try to replace it by some valid word.

So, it will try to replace a character by another character or it can transpose to a two adjacent characters. So, two characters suppose that I have got this if I. So, if I do a transpose for these two characters will change the interchange their position and it will give me i f similarly. So, for while I am trying to write the qr then maybe I have actually written tehn and then if I transpose these two characters then it will become t h e n this type of corrections can be done by the error recovery phase.

So, whether so how many of them will be done and whether it is done always. So, it is up to the compiler designer. So, language design will not talk anything about error recovery. So, and it has got nothing to do with code generation also because if there is a mistake in my program. So, that is the mistake that is an error.

So, it is not generate the code, but the compiler designer I can think about this type of possible errors and flash good number of error messages good amount of error messages so, that the program can be corrected by the user. So, it is up to the user to it is up to the compiler designer to identify possible or visualize possible mistakes that the user may do and accordingly come up with some policy for the recovery.

(Refer Slide Time: 06:37)

**Input buffering**

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
  - In C language: we need to look after  $\text{C}$ ,  $\text{=}$  or  $\text{<}$  to decide what token to return
  - In Fortran: DO 5 I = 1.25
- We need to introduce a two buffer scheme to handle large look-aheads safely

$x \text{ } - \text{ } y \xrightarrow{\text{MINUS}} x = y$

E := M \* C \*\* 2 eof

Sometimes we need to do some buffering, some lexical analyzer it needs to look at the some symbols to decide about the token to return.

So, typical example maybe this symbols like in a C programming language we have got the symbol say minus then equality less than. So, we have to look head and then only you can see you can do something written the proper token for example, example if you have got say x minus y.

So, at this point if the input, if your input point are somewhere here you have got a this minus symbol should I return the token minus ok. So, it is not it may or may not be correct because it may be the user as written x minus minus plus y, user has actually written this and you are at this point now. Now getting the single minus you should not written a minus the lexical analysis tool it should look I head see the next minus and then this blank and then it should say that this minus might have. So, this is basically auto decrement mode similarly this equality looking into the looking into one equality. So, whether it is correct or not that is the question, because I may have two equalities so, x equal to y and x equal to equal to y.

So, both are valid in the C programming language and if the user has written like this then if you are returning that it is an equality operator. So, that is wrong. So, it has to you have to say that it is a equal check operated by see you both these equality symbols you should return a single token which is the equality check. So, this way this lexical analysis

tool its, it needs to look ahead similarly is less than we have got many variants like less than is there less than equal to is there then we have got this left shift.

So, all these are there in the programming language. So, depending upon the programming language you may need to look ahead for the next few characters to know what is the valid token and in general the policy followed by the lexical analyzer is that it returns the maximally matched token ok. So, it will write the maximally matched token. So, whichever token matches with the next maximum number of characters so, that token will be returned a very nice example that we have this is the Fortran language. So, Fortran language it has it has got a loop constructs.

(Refer Slide Time: 09:20)

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
  - In C language: we need to look after -, = or < to decide what token to return
  - In Fortran  $\text{DO } 5 \text{ I} = 1,25$
- We need to introduce a two buffer scheme to handle large look-aheads safely

So, where the structure is like this that you look. So, DO then a label and then we have got some variable I equal to say 1 comma 25 and then you write the body of the loop and then somewhere at this point. So, you have this label and here you say continue here. So, means that this body of the loop will be repeated, first with the I is equal to 1 then I equal to 2 etcetera up to I equal to 25. So, this is the, this is a valid so this is some example that we are doing.

So, actually the user wanted to write in this case DO 5 I equal to 1 comma 25, but by mistake this 1 comma 25 has been written as 1 dot 25. So, this 1 dot 25 so this is a real number which is allowed in Fortran language and Fortran language it does not does not recognize this blanks in between the symbols, it just ignore this blank. So, this entire DO

5 blank b I so, this whole thing is reduced to a variable DO 5 I. So, it thinks that this DO 5 I is a single variable that is equal to 1.25. So, this is this is the interpretation of this particular statement.

So, until and unless we have seen you so whether this comma or a full stop or dot. So, you cannot know whether you should written DO as a loop token or this DO 5 I as a identifier as an identifier. So, you see the level of complexity that this lexical analysis tool is going to face. So, it has to look ahead quite some, quite a few some characters to see whether a comma appears after the first number after the equality symbol. If it is not then it is the whole that the entire left hand side will be reduced to a variable name where as if it is a comma in that case it has to written only the do part and the input pointer should be pointing to the next character here; where as in this case if it is 1.25 then the input pointer will come to this point.

So, this type of things are there so, it designing lexical analysis part is not that symbol apparently it seems it is quite simple I will just do a simple string match and where ever I get the match I will written it. So, that is not the case because programming language are often very critical in nature. So, and that is up to the language designer they have done it like that.

So, we have to follow that. So, sometimes you need to introduce a two buffer scheme to handle large look ahead safely so that when one buffer pointer is advancing through one buffer, the other buffer holds the next symbol next input string. So, that way there is no possibility of over flow, sometimes we need to save some amount of buffer information on to some other so other buffer so that these symbols are not lost in the process any way so, they are all implemented in the legs tool. So, while designing the compiler we do not really feel that we have to do so complex things, but the legs tool that is there. So, it takes care of that.

(Refer Slide Time: 12:58)

The slide has a yellow header bar with the title "Specification of tokens". Below the title is a bulleted list:

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:  
letter | letter | digit  
↓  
letter = {a, b, c, ..., z}  
digit = {0, 1, 2, ..., 9}
- Each regular expression is a pattern specifying the form of strings

Handwritten notes are overlaid on the slide, including:  
a1 ✓ a2 ✓ a22 ✓  
a3 ✓ a4 ✓ a5 ✓  
 $\Sigma = \{0, 1\}$   
 $\Sigma = \{a, b, \dots, z, A, B, \dots, Z, 0, 1, \dots, 9\}$

The footer of the slide features the "swayam" logo.

Next we will come a very important part of this lexical analysis phase which is known as specification of tokens, like how do you tell what are the valid tokens in your programming language? So, any language or not only programming language any language. So, as I said any language has got an alphabet which defines the characters or symbols that the language can help.

So, if we are the, if we are talking about say the set of sentence is over binary symbols then my alphabet set is often represented as sigma. So, this has got 0 and 1 unit only two symbol, then if I say that we have got a say the English language text then all the symbols a, b up to z then this capital A, B up to Z then we have got these numbers 0, 1 up to 9.

Then all the symbols that I have in the English language so they will come as a alphabet set for the language. So, from this alphabet set we have to define some rules by which I will tell what are the valid strings or valid words for the language. So, here we will be using a particular structure known as regular expression to formalize the specification of tokens ok. So, we will be learning about regular expression for that purpose. So, regular expressions are means for specified regular languages. So, I will come to this regular language parts slightly later, when we go to the parser phase parsing phase and we will talk about the grammars and all for the time being we take it as accepted that the regular expressions can specify regular languages.

So, what is a regular language we learn it later. So, a typical example of regular expression is like this letter then with in bracket letter then vertical bar digit bracket close star. So, where this letter is a set of symbols so, any letter maybe I can say the letter set is, the set letter is all the lowercase letter that we have in the English language then digit. So, this is a set of symbols 0, 1, 2 up to say 9.

So, these are digits. So, it says that this whole thing is a regular expression which must start with a letter and after the letter I can have a letter or a digit and this letter or digit can occur any number of times. So, this star at the end means that with whatever portion on which we have applied this star that can appear any number of time.

So, if I if I like write a b so that is a valid that is a valid word for this particular regular expression then I can have say a 2 b or a 22 or a 3. So, a all a s anything that we can write like this, but anything that star does not start with a letter is not valid, like if I ask whether this two a is a valid word as per this rule or not. So, it will say no. So, this not correct, but all these are correct, all these are correct, but this is not. So, so we have got this particular rule where this. So, this is a regular expression this whole thing is called a regular expression.

So, time back I was telling how are you going to specify the patterns that are allowed for the programming language. So, this is actually the, this is done by means of the regular expression. So, you have to define the set of regular expressions for your language that will define all words that you have in your language.

So, each regular expression is a pattern that specifies the form of string for the language. So, the type of strings which will correspond to this regular expression so that will be identified by this regular expression. So, every token now you can understand that, each token has got a corresponding regular expression. So, for example, I have previously I talked about that token if ok.

(Refer Slide Time: 17:34)

Specification of tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
  - letter(letter | digit)\*
- Each regular expression is a pattern specifying the form of strings

if  
COMPARISON  
>  
=|>|<|>|=|<=| -

swayam

So, these that is the corresponding regular expression is simply if this is the corresponding regular expression similarly if I have got say comparison operator. So, the comparison, this particular token.

So, the corresponding the regular expression is equality or greater than or less than or greater equal or less equal. So, this way you can specify the token. So, the corresponding regular expression. So, each token will have got a corresponding regular expression and the task of the lexical analysis tool is to see which of these regular expressions maximally match the next part of the input. So, if this is your entire, if this is the entire input file that you have may be at present the lexical analysis tool is at this particular line and in this particular character. Now, if you ask when the parser asks it for the next word or next token what it does it scans from here and sees which token matches maximally ok.

So, we it finds the maximal match. So, may be the maximal match up to this much entire thing maximally matches with some token then this entire thing will be. So, that is checked by the regular expression. So, this entire part matches the corresponding token will be retuned. So, it is the task of the lexical analysis tool to see which token matches or which regular expression matches maximally from the given set of regular expressions and return the corresponding token to the parser. So, this is how this lexical analysis tool is going to what.

(Refer Slide Time: 19:28)

NULL

## Regular Expressions

- $\epsilon$  is a regular expression denoting the language  $L(\epsilon) = \{\epsilon\}$ , containing only the empty string  $\Sigma = \{a, b, c\}$
- If  $a$  is a symbol in  $\Sigma$  then  $a$  is a regular expression,  $L(a) = \{a\}$   $L(a) = \{a\}$
- If  $r$  and  $s$  are two regular expressions with languages  $L(r)$  and  $L(s)$ , then
  - $r|s$  is a regular expression denoting  $L(r) \cup L(s)$ , containing all strings of  $L(r)$  and  $L(s)$
  - $rs$  is a regular expression denoting  $L(r)L(s)$ , created by concatenating the strings of  $L(s)$  to  $L(r)$
  - $r^*$  is a regular expression denoting  $(L(r))^*$ , the set containing zero or more occurrences of the strings of  $L(r)$
  - $(r)$  is a regular expression corresponding to the language  $L(r)$  $L(r) = L(r) \cup L(r)$

So, we start with the definition of regular expression. So, epsilon so it is a regular expression denoting the language  $L$  of epsilon equal to epsilon containing only the empty string. So, throughout our discussion. So, this epsilon will be these will be calling as this a special symbol. So, will be call this as null ok. So, wherever you find this epsilon you take it as null no is a wide sort of thing you can say.

So, this is special regular expression so if your language. So, ai if your language contains only the null string that is very regimenterally language that has got only one string which is a null string. So, that is empty string. So, that is represented by the regular expression epsilon, then if you have got the symbol a small in the alphabet set sigma then a itself is regular expression corresponding to the language that has got the string which contains only a single a.

So, if my sigma in the alphabet set I may have all this symbols say a b and c fine then  $L$  of a  $L$  of a means it is talking about the language that has got the alphabet a only it does not have the other alphabets b and c. So, naturally so  $L$  of a is only the, it has got only a only a single valid string in the language which is a single a similarly  $L$  of b. So, each symbol that you have in the alphabet set constitutes a language consisting of a single occurrence of that symbol ok. So,  $L$  of b is a single b  $L$  of c is a single c.

So, it defines those language that contains the single b or single c in it now comes combination of two or more regular expressions. So, if  $r$  and  $s$  are two regular expression

with corresponding languages  $L$  of  $r$  and  $L$  of  $s$  then  $r$  vertical bar  $s$  is a regular expression denoting the language  $L(r) \cup L(s)$ . So, when I say  $L(r)$  so this is  $L(r)$  is nothing, but set of strings, set of strings in the language in the language with  $r$  as regular expression. So, that is  $L(r)$ . So, this is a set it defines the all the strings that are possible in the language identified by the regular expression  $r$ .

So, when I say  $r$  vertical bar  $s$  or we read it as  $r$  or  $s$ . So, if  $L(r)$  and  $L(s)$  are two languages then this  $r$  vertical bar  $s$  it will correspond to the language  $L(r) \cup L(s)$ . So,  $L$  of  $r$  or  $s$  is nothing, but  $L$  of  $r$  union  $L$  of  $s$ . So, any string which belongs to the language  $L(r)$  also belongs to the language  $L(r) \cup L(s)$ . Similarly any string which belongs to the, which belongs to the language  $L(s)$  also belongs to the language  $L(r) \cup L(s)$ . Next we look in to the next rule it says that if  $r|s$  is a regular expression so  $r|s$ ,  $r$  is a regular expression and  $s$  is regular expression or they are occurring side by side.

(Refer Slide Time: 23:21)

- $\epsilon$  is a regular expression denoting the language  $L(\epsilon) = \{\epsilon\}$ , containing only the empty string
- If  $a$  is a symbol in  $\Sigma$  then  $a$  is a regular expression,  $L(a) = \{a\}$
- If  $r$  and  $s$  are two regular expressions with languages  $L(r)$  and  $L(s)$ , then
  - $r|s$  is a regular expression denoting the language  $L(r) \cup L(s)$ , containing all strings of  $L(r)$  and  $L(s)$
  - $rs$  is a regular expression denoting the language  $L(r)L(s)$ , created by concatenating the strings of  $L(s)$  to  $L(r)$
  - $r^*$  is a regular expression denoting  $(L(r))^*$ , the set containing zero or more occurrences of the strings of  $L(r)$
  - $(r)$  is a regular expression corresponding to the language  $L(r)$

$r|s$        $rs$        $r^*$

$L(r) = \{a, ab, abc\}$   
 $L(s) = \{x, y, z\}$   
 $L(r)L(s) = \{(ax, ay, az, bx, by, bz)\}$

FREE ONLINE EDUCATION  
swayam

So, for example,  $a$  is a regular expression and  $b$  is a regular expression or I can say  $a|b$  is a regular expression and  $b|c$  is another regular expression. So, that way may be my  $r$  equal to  $a|b$  and  $s$  equal to  $b|c$  in that case  $r|s$  is this one  $a|b|b|c$ . So, if  $r|s$  is a regular expression then that will denote the language  $L(r) \cup L(s)$ .

So, any language any string that is created by taking one string from  $L$  of  $r$  and another string from  $L$  of  $s$  and concatenating the second string to the first string. So, you have got say one language  $L$  of  $r$  suppose it has got the language, it has got the strings  $a, ab$  and  $a$

c and L of s has got the string x and y then L of r s. So, this will have all these, all these strings like a x, a y, a b x, a b y, a c x and a c y that is you take the first part from the language L r and second part from the language L s and concatenate the two strings together. So, what you get is the set of strings that are allowed in this language.

So, if you have constructed individual, individual languages for the regular expression r and s. So, you can take concatenation of those strings of different languages and tell them that it is the regular expression for; it is the language for the regular expression r s. So, this is there then we have this special regular expression r star. So, r star means this star symbol it means that it is 0 or more occurrences of the regular expression r.

(Refer Slide Time: 25:33)

- $\epsilon$  is a regular expression denoting the language  $L(\epsilon) = \{\epsilon\}$ , containing only the empty string
- If  $a$  is a symbol in  $\Sigma$  then  $a$  is a regular expression,  $L(a) = \{a\}$
- If  $r$  and  $s$  are two regular expressions with languages  $L(r)$  and  $L(s)$ , then
  - $r|s$  is a regular expression denoting the language  $L(r) \cup L(s)$ , containing all strings of  $L(r)$  and  $L(s)$
  - $rs$  is a regular expression denoting the language  $L(r)L(s)$ , created by concatenating the strings of  $L(s)$  to  $L(r)$
  - $r^*$  is a regular expression denoting  $(L(r))^*$ , the set containing zero or more occurrences of the strings of  $L(r)$
  - $(r)$  is a regular expression corresponding to the language  $L(r)$

$L(r) = \{\epsilon, ab\}$

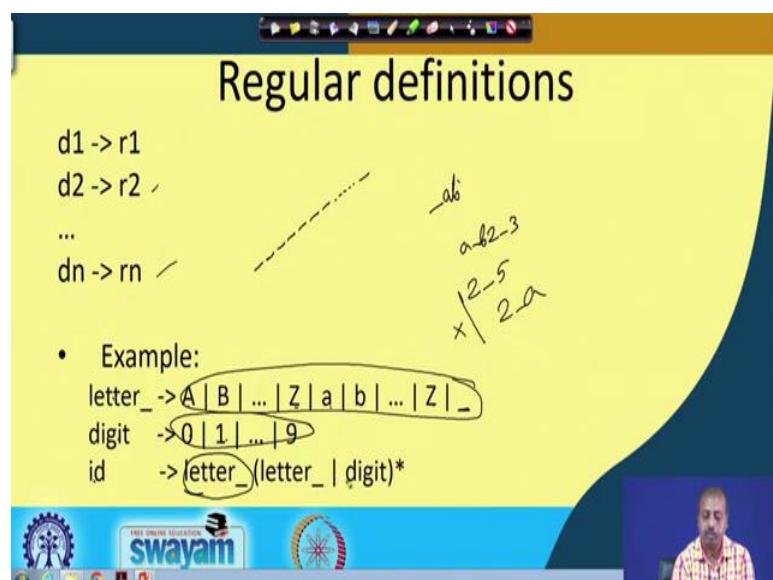
$L(r^*) = \{\epsilon, a, ab, aa, aab, aba, \dots\}$

So, if you have got say L of r, suppose I have got the two strings a and a b then L of r star L of r star. So, this will have all these all these words all these strings. So, 0 or more occurrences of this language this language words, if I take 0 occurrences. So, that is the null. So, epsilon if I take one occurrence. So, I can get a and a b if I take two occurrences then I can get a a or a a b similarly a b, a b, a b a. So, this way we can think about different words ok, different strings this set is infinite because this is 0 or more occurrence you can just go on augmenting taking more and more strings and connecting them together. In different number of times I have taken only two times, you can take it any number of times because this is a star.

So, this  $r^*$  star is a regular expression denoting  $L^r$  whole star, the set containing 0 or more occurrences of the string of  $L^r$  ok. So, and each whenever we have taking the string you are free to choose any string. So, that way they can be concatenated in arbitrary fashion.

So this is a very powerful operator that we have ok, unlike this or said say consecutive occurrence. So this star is a 0 or more occurrences of the symbol then with in bracket  $r$  is a regular expression which corresponding to the language  $L$  of  $r$  so it is nothing new. So, this  $r$  the regular expression  $r$  has got the language  $L$  of  $r$ . So if you put it with in bracket also we do not get anything new, we get the same language  $L^r$ . So, this way the regular expression, the regular expressions are defined. So this is definition of regular expression.

(Refer Slide Time: 27:45)



So here is some example suppose. So we have got this thing like regular definitions. So definition 1 will give me some regular expression  $d_1$ , definition 2 will give me some regular expression  $d_2$  definition  $n$  will give me regular expression  $r_n$ . So one such definition is the letter underscore so that means, I am talking about. So this is a definition which says that all these characters can appear A B capital A capital B upto capitals Z then small a small b up to small z and this underscore. Then this digit is a definition it is 0 1 up to 9 and id is another definition.

So, these are these are the specified in terms of regular expression only, you see this whole thing is a regular expression, similarly this whole thing is a regular expression.

Now I am use these two definitions to define an id how am I defining letter underscore followed by letter underscore or digit and this star. So I can start; that means, for a valid id for any character string to be called a valid id. So it may I start with letter or it may start with underscore character.

So maybe so I can write, I can have a character string like this underscore a b. So that is valid similarly I can have like a underscore b 2 underscore 3. So this is also valid. So they are all id they will all be identified as id. So, this can be done. So we can have this type of regular expressions formulated by first writing a few definitions and then writing the more complex one, building the more completes one based on the simpler regular expressions.

So these are called regular definitions. So you can we use these definitions to build more and more complex set and. So you can understand that anything so this is not valid like you cannot select 2 1 underscore 5. So that is not a valid one or 2 underscore a that is not valid one because it says that it should start with the letter or underscore character, but here for these two cases that is not true so they are not valid.

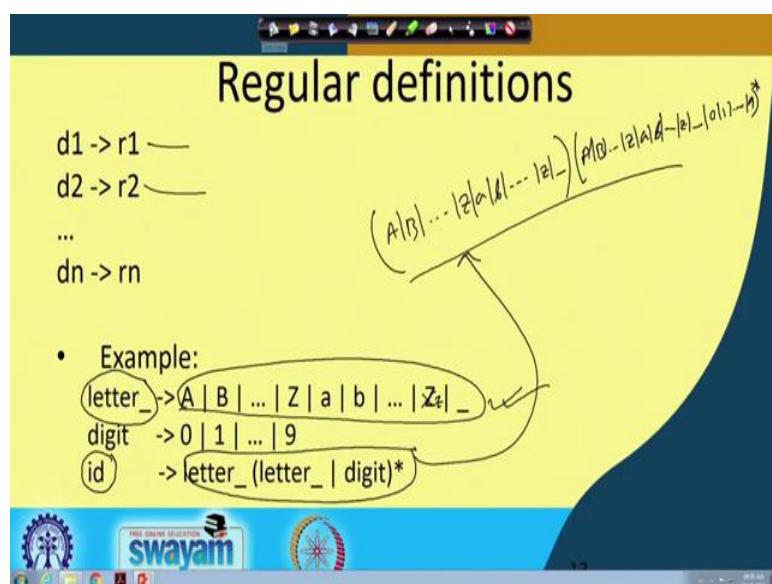
Whereas these cases they are fine and they can occur any number of times. So in the first character, even this is going to be a valid string. So all underscore and this can go up to infinite. So this is also valid because I am starting with an underscore that follow, that satisfies the first part of the definition and then I am starting with the then I am doing with it is the second underscore and that is satisfying any number of time. So this also a valid id in the as per in this particular language.

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

**Lecture – 09**  
**Lexical Analysis (Contd)**

So regular definitions so, we have seen few in the last class today we will continue with that at the beginning.

(Refer Slide Time: 00:23)



So definition each definition for we have got a corresponding regular expression. For example, this d 1 has got a regular expression corresponding to each which is r 1 d 2 has got r 2. So, we generally write it in this format d 1 then 1 arrow and then the corresponding regular expression. A typical example is say this letter underscore is a definition and that definition is governed by this particular regular expression. And as we have seen in the definition of regular expression so, this is A or B or up to capital Z than small a small b up to this should be small z not big Z, this should be small z and the underscore character

Then the digit is another definition is 0 1 0 or 1 or 2 or something up to 9. Then id an identifiers another definition which is composed of the previous two definitions like letter underscore followed by letter underscore or digit whole star. So, the definition says of this id this definition says that any character string that matches this particular

definition should start with the letter or underscore character and that is if you substitute this right hand side of this part here. So, what you get is A or B or Z or underscore character followed by this characters or digits. So, this part this whole thing is actually something like this. So, this is A or B or Z or a or b up to small z. Then you have got the underscore character.

So, this whole thing followed by again this one that A or B or Z or small a or small b or small z underscore or 0 or 1 or up to 9 and this whole star. So, this is the full thing so, this is written in a short hand form in the here. So, for making the expression shorter so we normally followed this type of strategy that we follow we define some regular definition for some bigger parts. And then sub parts and then use those sub parts or sub definitions to define a more complex expression. So, this is the way we will proceed so we will look in to some more examples like shape.

(Refer Slide Time: 03:03)

- One or more instances:  $(r)^+$
- Zero of one instances:  $r^?$
- Character classes:  $[abc]$

$$L((r)^*) = L(r^*) - \{\epsilon\}$$

- Example:
  - letter\_ ->  $[A-Za-z_]$
  - digit ->  $[0-9]$
  - id ->  $\text{letter}_(\text{letter}_|\text{digit})^*$

So before that we have got some other notations as well like if I have a regular expression r, then we say that r star is 0 or more occurrence of r. Similarly, r plus is another notation. So, this one it says that one or more instances. So, it is the only the epsilon is excluded so, all others are there. So, basically languages which is composed of r plus is equal to the language which is composed of r star so, language which is composed of r star minus the null string epsilon. So, then there r question mark so, this is

0 or 1 occurrence of r. So, this is it is not multiple occurrence so, it is 0 or 1 either 0 or 1 occurrence.

Then sometimes we define a character class for that we write like within square bracket we write abc. So, this means that I am talking about the characters a b and c and they are forming a class. So, this short hands will be used for writing complex expressions which otherwise becomes a bit ineligible to understand so, this short hands will be used. For example, this letter underscore previously you have seen a big definition. So, here it is shortened so, I am writing like A to Z so, this is a range A to Z then small a to small z. So, by character class definitions of this becomes one regular definition similarly digit instead of writing each and every digit so, it is we write as 0 to 9. Actually these are for human understanding so, as long as human being can understand the definitions so, it is fine. Of course, if you want to put it in a computer. So, or in some program then of course, you have to write explicitly a or b or c or d like that.

But for human understanding so, it is better if we use short hands so this is actually the usage of shorthand. Similarly, id is letter underscore followed by letter underscore or digit star. So, these are the some extensions that will be using while handling with handling the regular expressions.

(Refer Slide Time: 05:23)

Now, let us look into some examples, suppose I am considering strings over binary symbols 0 and 1. So, my in the language the alphabet set contains only two symbols 0

and 1. So what is the language corresponding to the first regular expression? So, 0 or 1 star so it means that if the star is not there so, it says 1 occurrence of 0 or 1 that is a single beat. Now, there is a star over this so; that means, that all binary strings including empty strings. So, everything will be acceptable in that language because, it says any occurrence of 0's and 1's with alternating 0's and 1's or the null string. So, everything is there so, this is the thing, the second regular expression. So, 0 or 1 followed by 0 or 1 start it is all non-empty binary strings. So, it so, this 0 or 1 star means 0 or more occurrences of the characters 0 and 1.

And before that there is another part of the regular expression which is 0 or 1 so; that means, the overall at least 1 0 or 1 1 must appear then rest of the characters they are may not be any more characters. So, single length string or maybe multiple length strings. So, you can understand that this actually equivalent to writing 0 or 1 plus so this is same as this expression. Then the next one say this one so, this expression says 0 followed by 0 or 1 star and followed by 0.

So, if we try to understand the meaning of this particular regular expression. So, this part means 0 or more occurrence of the characters 0 and 1, but the string must start with a 0 and end with a 0. So, the strings are having at least a length of 2 and it is starting with a 0 and ending with a 0. So, all binary strings of length at least 2 starting and ending with 0's are valid strings for this particular regular expression. Then the next one it is very tricky it says that 0 or 1 start 0 then the next three characters can be 0 or 1. So, all strings where the, which are having at least three characters in which the third last character is always 0. So, it is not three characters so, it is at least four characters so, this is lightly wrong.

So, this is at least four character in which the fourth last character, fourth last character is always 0. So, I can have a string like 0 0 1 0 0 something like that 1 and then this is a 0 and then I can have three more character. So, it maybe 1 0 1 for example so, this 1 will match with this part this 0 will match with this part, then this 1 will match with this part this 0 matches with this 0. And then remaining part so this one this entire thing sorry up to this one, up to this one so this will match with the 1st part. So, this way I can have this particular regular expression which corresponds to all binary strings of at least four characters in which the fourth last character is always 0. Then the next regular expression it is 0 star 1 0 star 1 0 star 1 0 star. . .

So, it says that there can be any number of 0's, but can be exactly three 1's in the string. So, for matching I must have 1 1 1 at the some places and in between I can have any number of 0's. So, I have three 1's 1 1 and 1 and in between I can have. So, may be at this phase I have got 2 0's at this place I have 4 0's after this I have got quite a few 0's so, like that.

So, any such string where I have got exactly three 1's will match with this particular regular expression. So, this way depending upon the language that we are talking about the word in it so, we can frame the corresponding regular expression. So, naturally so these are these examples are it are a bit synthetic. So, if we are looking for some programming language then we have to see like what are the valid words in that language and accordingly we have to define the regular expression.

(Refer Slide Time: 10:05)

**Example**

Set of floating-point numbers:

$$(+|-|ε) \text{ digit } (\text{digit})^* \cdot (\text{digit } (\text{digit})^* | ε) ((E (+|-|ε) \text{ digit } (\text{digit})^* | ε) | ε)$$

Annotations include circled parts of the expression and circled numbers like 5.23E25 and 5.23.

So, next we will look into another example say the set of floating point numbers. So, floating point number they have got several parts in it first there is a symbol. So, a number may be say a number may be say 5.23 into 10 to the power say 25 something like this. So, this is normally written as 5.23 E 25; now there can be signs also like at the beginning I can have a sign here that signs so, number may be positive or negative so, it may be plus or minus. Similarly this power so, it can also be plus or minus so, accordingly we define the regular expression the first part is defining, this first part is defining the first sign.

So, I start with the sign so, this sign part this plus or minus. So, is it is captured by this plus minus or epsilon. So, epsilon means there is no sign that is specified so it is taken as a plus number. Then the next part so, there is a decimal point and before decimal we have got some portion. So, here I have got a single digit so, that is digit, but it is not mandatory that there should be a single digit the number may be say 56 point something 57. So, this 56 will be captured by digit followed by digit star 2 digit and then this dot character is there. So, this dot appears and after that I can have the fractional part of the expression. So, this is digit followed by digit star or epsilon so, fractional part may or may not be there.

So, it can be that I may have say I may have minus 5 E 25 so, it may be there. So, at that point I do not have after 5 any digit. So, there is no fractional part or the number may be 5 minus 5.25 E 27. So, that way this 0.25 has to be captured. So, 0.25 will be captured by this part and here nothing is there so, that is captured by the epsilon part. So, this is there then the E character must be there the 10 to the power that exponent part. So, the E character must be there so that E is there then after that this exponent may have a signature may have a sign or may not have a sign. So, if there is a sign so, it may be plus or minus if there is no sign it is captured by this epsilon followed by again number. So, there must be at least 1 digit or there can be multiple digits. So, this whole thing can be there or this E part may be totally absent.

I may also like to represent a number like 5.45 without any exponentiation part this is a 10 power part. So, this part may be totally absent so, that is captured by this epsilon. So, this way this entire expression is taking care of the situation that set of floating point numbers in some programming language so, it may be captured by this particular regular expressions. So, you can formulate other regular expressions depending upon the words that are allowed in the language and accordingly come up with the recognizer for them.

(Refer Slide Time: 13:39)

Recognition of tokens

- Starting point is the language grammar to understand the tokens:

```
stmt -> if expr then stmt  
| if expr then stmt else stmt  
| ε  
expr -> term relop term  
| term  
term -> id  
| number
```

else  $x > y > z$

Now, how do you recognize that tokens? So, starting point is the language grammar to understand the token. So, what are the tokens in the language so, that for that we need to look into the grammar of the language. For example, this is some hypothetical language where we have got this if then else statement.

So, this statement so, we will see later when we go to the parser chapter. So, this is this particular some words which are not some words in the programming language so, they will called non terminal symbols. So, this non terminal statement it is producing this terminals involve if followed by some expression then there terminal symbol then and then another statement.

So, if expression then statement out of this if and then they are tokens of the language because they are some they correspond to some valid pattern that we have in our valid (Refer Time: 14:44) that we have in the program that we have that we take as input. Similarly, if in the second one if then and else they are the three patterns that are accordingly I should have corresponding tokens and or epsilon. So, statement may be if expression then statement or if expression then statement else statement or epsilon.

Similarly, an expression so I expect a conditional expression for the if then else statement. So, this is at some term followed by some relational operator and another term or it may be a simple term where term is actually an identifier or it is a number. So, I can have a statement like if a greater than b then some statement maybe x equal to y plus z

else x equal to y minus z. So, here this if will correspond to this token this (Refer Time: 15:45) in the input text file I have got these thing. So, this particular input stream portion so, they will give me the token if similarly this part will be reduced expression and how is it reduced.

So, you see that this is coming as term relational operators is this greater than and this b will be term. And this term will be represented by the identifier a and the second time will be represented by the identifier b. So, this will be more clear when you go to the parser chapter, but essentially what you want to emphasize at this point is to identify the tokens of the language. So, we have to start with the grammar of the language and then from there we can see like what are the important tokens that can come and accordingly we can proceed.

(Refer Slide Time: 16:35)

- The next step is to formalize the patterns:

```

digit -> [0-9]
Digits -> digit+
number -> digit(Digits)? (E[+-]? Digit)?
letter -> [A-Za-z_]
id -> letter (letter|digit)*
If -> if
Then -> then
Else -> else
Relop -> < | > | <= | >= | = | <>
  
```

- We also need to handle whitespaces:

$$ws \rightarrow (\text{blank} \mid \text{tab} \mid \text{newline})^+$$

So, the next step is to formalize the patterns. So, what are the pattern like we have to give the definition, like digit is anything a range 0 to 9 the characteristic characters 0 to 9. So, they form a digit then we can put another definition digits which is digit plus that is 1 or more occurrence of digit.

Then number is defined as the digit it should start it should it must have at least 1 digit and then dot digits, digits and question mark. So, digits and question marks so, this part we will mean that 0 or more occurrence of digits so, it may be 5.9. So, in that case this point dot digits so, they will match with this part and sometimes I may write simply as 5.

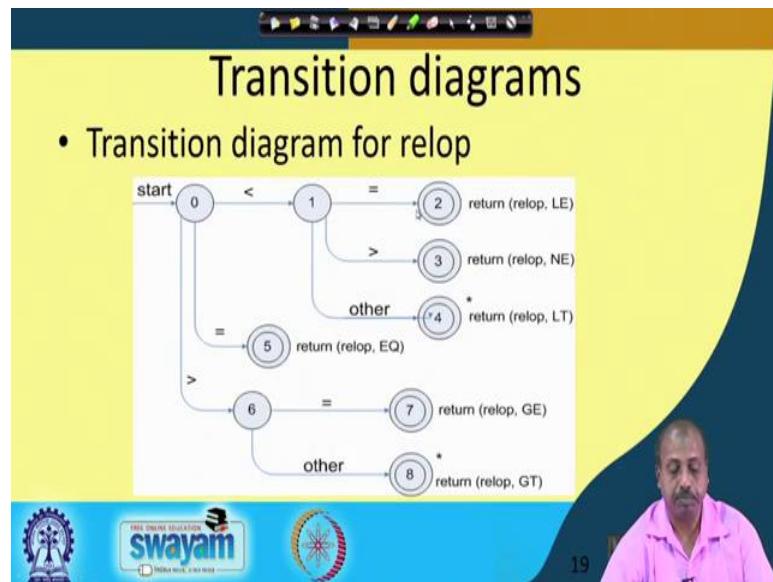
So, there is a integer number so, there is no fractional part. So, this question mark will mean 0 or 1 occurrence of digit. And after that I can have I will have E the exponentiation then there may be a question mark and this plus minus. So, occurrence of plus symbol or minus symbol and then digits so, this would be digits so, I can have more digits there.

So, this way I can actually this should be capital D capital digits ok. So, this should be so, this way I can we have already explain the floating point number so, on the same line this number maybe define. Similarly the letter so, we have got the A to Z capital uppercase letters A to Z then lowercase letters a to z and the underscore character. Then an identifier when you are defining so, it is letter followed by letter or digit star. Then this if is another definition which is actually the characters i followed by character f appearing on the input stream. Then I am defining another definition then so, which is the characters string t h e n this character. So, this way I can define this thing then say relational operator. So, I can have less than greater than less or equal greater or equal, equal not equal like that so, we can define the relational operators.

We also need to handle whitespaces. So, whitespace how do you define a whitespace? So, blank tab and new line character. So, these are the three whitespace characters we have now there can be 1 or more occurrence of such whitespaces. Somebody may write like say blank then may be put in 2 tabs and then a newline character so, that can happen.

So, this whole thing will be identified as a single whitespace and may be the lexicon analysis tool will remove all this whitespaces from program. So, that can be done. So, this whitespace so, this whitespace is another regular definition which is given by this regular expression blank or tab or new line plus so any number of them. So, this way we can define the patterns and the corresponding tokens.

(Refer Slide Time: 19:55)



Now, once we have done that so, we can you take help of some transition diagrams to identify the tokens. So, for example, if I am looking for the relational operator the definition was like this less than greater than less or equal greater or equal, equal and not equal. So, accordingly I can say that if this is my if the lexical analysis tool it starts at state 0 of these finite automata. Then or getting the less than symbol it will come to state 1 and then if it after that in the state 1 if it sees as next input symbolize equality then it comes to a state 2 which is a final state and identified by 2 concentric cycles and then it will return the token relop with the value or the attribute as LE. So, LE maybe some less or equal so, there may be some value or some constant define for the symbol for this symbol LE.

But, whatever it is so, conceptually we can say that the token delay relop. So, it can have values like LE NE LT etcetera so, it is returning the value LE. Similarly, after coming to state 1 if it finds is greater than symbol then it can say that it is relational operator not equal to. So, if it is any other symbol so, it is less then that so, it will return the relational operator LT. Similarly, at state 0 if it finds equality so, it will return relational operator EQ and then it is greater than equal to then it will return the relational operator greater there was it will return the relational operator greater or equal and here in state 8 it will return relational operator token with value greater.

(Refer Slide Time: 21:59)

## Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers

```

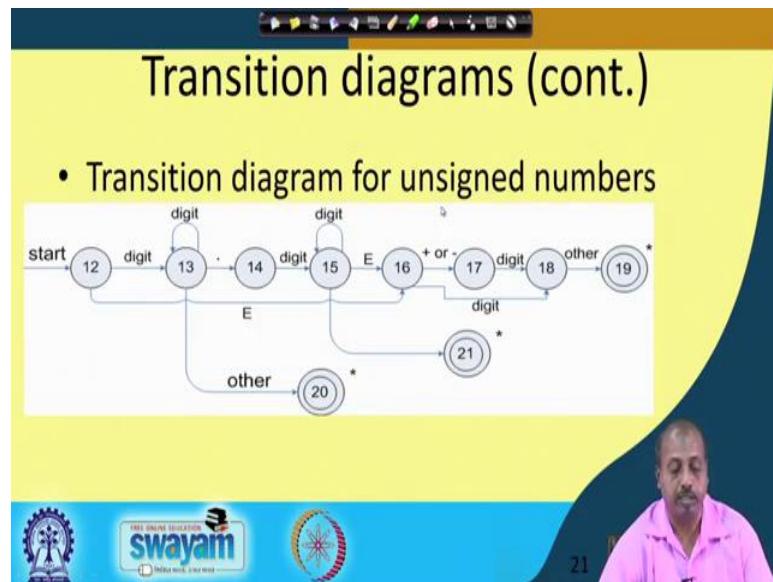
graph LR
    9((9)) -- letter --> 10(((10  
letter or digit)))
    10 -- "letter or digit" --> 10
    10 -- other --> 11(((11  
* return (getToken(), installID())))
    
```

The diagram illustrates a state transition for lexical analysis. It begins at state 9, labeled 'letter'. A transition labeled 'letter' leads to state 10, labeled 'letter or digit'. From state 10, if the input matches a letter or digit, it loops back to state 10. If the input does not match, it transitions to state 11, which is marked with an asterisk (\*) and the action 'return (getToken(), installID())'.

So, in this way you can define other such transition diagram also like this is the transition diagram corresponding to the reserved words and or identifiers. So, this is the starting state then it is coming to this letter followed by letter or digit. So, here and then if it is not matching with if it is not matching here then it will anything else. So, it will go to this other states. So, if it as long as it gets letter or digit fitted with looping there when it gets some other symbol it comes to the state eleven and it returns get token and install id.

So, this get token will get the corresponding token. So, depending upon this ID so, this will be returning the ID and this install ID. So, it will install the identifier into the symbol table and that symbol table off set will be returned to the partial. So, this way I can say that this transition diagram for reserved words and identifiers can be developed. So, we can use this in the lexicon analysis phase for defining these keywords.

(Refer Slide Time: 23:09)



Then we can have unsigned numbers. So, this is another possibility like from the start if you if you come to state 12 and if you get a digit there so, it comes to. So, it is digit then digit start then dot so, this number definition that we have seen previously so, it is for that purpose. Now, it may get only a single digit and that may be the end. So, it comes to state 20 it may be that from the beginning. So, you do not have any digits immediately starts with 10 power that is E. So, a straightway come to here come here. So, this way we can represent the valid unsigned numbers in the language in terms of a transition diagram. So, regular expression is a definition for the tokens that we have in the language.

Then their implementation can be in terms of transition diagram and once you have got a transition diagram. So, you can realize it by means of some program or you can realize it by means of some automata. So, this is some sort of automata because I have got a number of states and transitions between them. So, that defines 1 automata so, we can use that symbol that technique.

(Refer Slide Time: 24:29)

```
graph LR; start((start)) -- "delim" --> S23_1((23)); S23_1 -- "other" --> S24((24));
```

Next for white space so, it is delimiter followed by any number of delimiters and any other symbol it will come to state 24 where it will return the token delimiter so, this or white space. So, this is this is what this delimiter is this whitespace transition diagram may be developed like this. We can so, in this way so, these are a few examples only so, for depending on the programming language that we are handling.

(Refer Slide Time: 25:05)

```
TOKEN getRelop()
{
    TOKEN retToken = new (RELOP);
    while (1) /* repeat character processing until a
               return or failure occurs */
    {
        switch(state) {
            case 0: c = nextchar();
                if (c == '<') state = 1;
                else if (c == '=') state = 5;
                else if (c == '>') state = 6;
                else fall; /* lexeme is not a relop */
                break;
            case 1: ...
            ...
            case 8: retract();
                retToken.attribute = GT;
                return(retToken);
        }
    }
}
```

So, we have to see: what are the different regular definitions and accordingly we have to do it. Now, architecture of a transition diagram base lexical analyzer tool so, this is just a

just a typical example ok. So, how we will know how can we develop a corresponding code. So, this token get relop so, this is the relational operator for the relational operator part so, it will do it like this. So, it will first it will return a new relational operator as the token. So, while so, now, it will read the characters until a return or failure occurs.

And depending upon the state so, it is case 0. So, this relation actually we need to consult this relational operator table, a relational operator transition diagram so this one. So, you see that I so, it is depending upon the states it is 1 or if it is 1 means I have seen this less than symbol and then we are writing it like this sorry case 0 case, 0 means we have not seen anything now that is in state 0 so, that is in state 0 ok.

So, we are not seen anything so, we are correctly in state 0. So, the possibilities I will see a less than symbol, equality symbol or greater than symbol. Accordingly you can write this code the case 0 so, get the next character if the next character is also less than then it comes it comes to state 1. And if it is equal to then it goes to state 5 it is greater than goes to state 6 otherwise it is a failure. So, lexeme is not a relational operator; similarly at state 1 also you can write a few, write a few rules here. So, like here so, state 1 so, I can see the equality greater than other symbols so, like that.

So, similarly this sometimes will be requiring to come back. So, state 8 it will be doing a retract because state 8 is here. So, you are getting something else so, you need to come back so it is not there not a relational operator so you need to come back so, here so, this is attribute is GT. And so, it will return the token relational operator token; this is based on the transition diagram you can write a piece of code which will correspond to realization of the transition diagram.

(Refer Slide Time: 27:35)

The slide has a yellow header with the title 'Finite Automata'. Below the title is a bulleted list of definitions:

- Regular expressions = specification
- Finite automata = implementation
- A finite automaton consists of
  - An input alphabet  $\Sigma$
  - A set of states  $S$
  - A start state  $n$
  - A set of accepting states  $F \subseteq S$
  - A set of transitions state  $\rightarrow$  state

To the right of the list is a hand-drawn diagram of a finite automaton. It shows a large oval representing the set of states. Inside the oval, two smaller circles represent individual states, labeled  $s_1$  and  $s_2$ . An arrow points from  $s_1$  to  $s_2$ , labeled with the letter 'a'. Above the oval, there is a small sketch of a zigzag line with arrows pointing along it, labeled 'input'.

The bottom of the slide features a blue footer bar with the 'swayam' logo and other navigation icons.

Next we will look into something called finite automata because transition diagram is a it is a way of doing the thing, but it may not be very much suitable very much powerful in terms of the representation. So, we would like to represent them by means of finite automata. So, regular expression is the specification for the token and finite automata is an implementation of the token.

So a finite automata it has got an input alphabet set sigma a set of states  $S$  a start state  $n$  a set of accepting states  $F$  which is a subset of  $S$  and a set of transitions state to state on some input value. So it will if it is currently at state  $S_1$  if you have a transition from state  $S_1$  to state  $S_2$  and it is on some input symbol  $a$  then it can go from state  $S_1$  to state  $S_2$  by consuming this input symbol  $a$ . So what happens is that this automata is specified by means of a diagram like this where starting with the start state I have got all this transitions and there is an input stream ok. So at present maybe the input pointer is somewhere here.

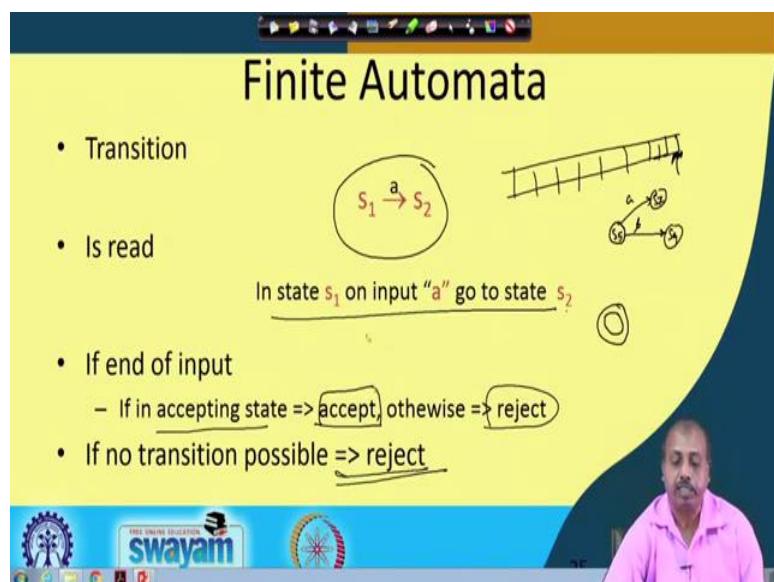
So if you are at the state  $S_1$  and the current input symbol is  $a$  then the input pointer is advanced to the next position and the current state of the system becomes  $S_2$ . So it consumes the next input and proceeds to the next states it transits to the next possible states. So this is the idea of finite automata. So you can use these finite automata to realize this lexical analysis tool for designing the lexical analyzer.

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

**Lecture – 10**  
**Lexical Analysis (Contd)**

So, any transition in finite automata which is written as this S 1 on input symbol a going to S 2.

(Refer Slide Time: 00:21)



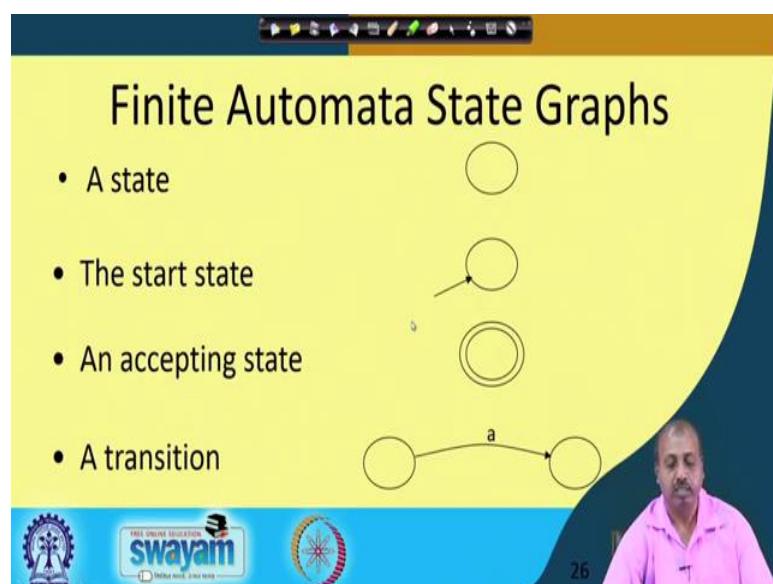
So this is a; this is a Transition. So, this Transition is read as in state S 1 on input a go to state S 2. So this is the meaning of this particular symbol S 1 arrow with a level a to S 2. So, this is read as this if you go if you find if you are at the end of the input. So this was the whole input stream as I was telling. Now, if it is so, if you have reach the end of the input. So, this point no more input is there. Then, we see whether the state at which the automata is at that point of time is an accepting state or not.

So, if it is an accepting state, then the decision is accept; otherwise the decision is reject and the accepting states are the final states and they are identified by this particular notation; the states that will have 2 are concentric circles in it. So they are the accepting states or final state. So that is the that may be the; so, if you are at this type at any of this type of state in that case so, it is accepted.

However, if there is no transition possible at some for some state like may be at present, I am at state say S 5 and S 5 the transitions are defined only on the symbols a and b, to say this is to say S 7, this is to say S 9 like that. Now, so if the next input symbol coming does not match with either a or b, then from S 5 the system is unable to go anywhere the automata transition is possible and that and in that case, so it will be a reject set.

So it will be it will go to a reject. So, the input stream is not accepted as per that regular definition. So it may so happen that some other regular definition will match. So, conceptually you can think that as if all the regular definitions are being matched parallelly by the lexical analysis tool and whichever rule gives the maximum match so, that is returned as the token. At the final state of it the token the match token for that is returned to the pastern.

(Refer Slide Time: 02:41)



So, this is the way, this operates the finite automata things. So, these are some of the notations like a state will be represented by a circle or ellipse whatever you do it both are same. The start state will have an arrow pointing to it. So, that will identify a start state and accepting state will have two concentric circles or concentric ellipses and a transition will be level from one state to another with a corresponding level a ok. So, this is a so, these are the notations that are used for defining a state transition graph.

(Refer Slide Time: 03:15)

The slide has a yellow background with the title 'A Simple Example' at the top. Below the title is a bulleted list:

- A finite automaton that accepts only "1"
- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

To the right of the list is a diagram of a finite automaton. It consists of two states: a start state (unfilled circle) and an accepting state (double circle). There is a transition from the start state to the accepting state labeled with the digit '1'. Above the accepting state are three input symbols: '1', '0', and '11'. At the bottom of the slide, there is a blue footer bar with the 'swayam' logo and other icons.

So this is one simple example. Finite automata that accepts only 1. So, this is the start state so we have got this is the start state and on getting the input one it comes to a final state. So, if you are giving as an input if you give a single one, then it will come here. But, if you give anything else for example, if you give a 0, then from the start state there is no definition, no transition defined with the level 0. So, it cannot do a transition so, it is in it remains in this state only and that is an error.

Similarly, if you are final state, then suppose I have given two 1's. So, on first one, it comes to this state. But, it does not know what to do on the second one as a result that is also an error. So, when the input is exhausted; so and you are if you are at an accepting state, then it is fine. If you are at some intermediary point; so, if you do not know where to go, transitions are not available for a for the current state so, in that case it is a reject.

So, a finite automaton accepts a string if we can follow transitions level with the characters in the string from the start to some accepting states. So, if you if it can lead you to some accepting state, then only it is the string is accepted.

(Refer Slide Time: 04:43)

## Another Simple Example

- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}

1111110  
1011

- Check that 1110 is accepted but 110... is not

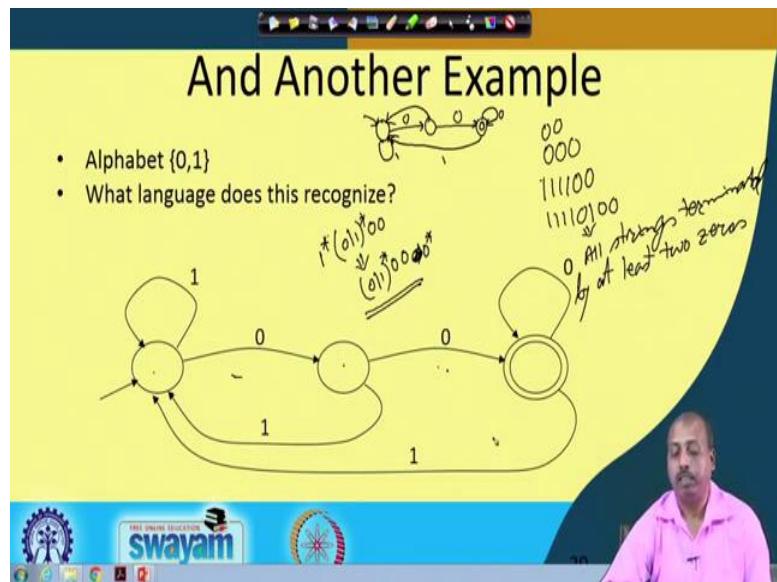
The slide also features a watermark for "swayam" and a video player interface.

Some other example like a finite automata accepting any number of 1's followed by a 0, single 0. So, all these strings are acceptable; like I can have say 1 1 0; I can have say 1 1 0. Similarly, 1 1 1 1 any number of 1's followed by a 0.

But this is not accepted 1 0 1 1 is not accepted because it is not ending with a 0. So, in as far as this language is concerned, the alphabet set consists of the symbols 0 and 1. Now, the transition the finite automata is defined like this, that this is the start state. From the start state as long as it gets 1, it remains in the start state; if it gets a 0, it comes to a final state.

So, that way it is doing the translations. So, in this case; so, this 1 1 1 0 is a valid string, but 1 1 0 if there is any more character, so that is not a valid string for this language. Because, once it once on a 0, it comes here from these state no transition is defined or neither on 0 nor on 1. So, as a result the second the 2nd one is not a valid string for the language. So, this way we can use this lexical this finite automata for representing the regular expressions.

(Refer Slide Time: 06:07)



What about this particular example? So, you see that if the string as long as it is long as so you start at this point, as long as you get 1, you stay in the start state only. Now, if you get a 0, it comes to the state and then, if you get a 1, it immediately goes back to the state, the start state. So, it can reach final state only when there are two 0's ok. One 0, two 0 and then, it remains there as long as there are 0's. Whenever it gets a 1, it comes back to the first state.

So, some typical language that are accepted by this are say 0 0 0 or simply two 0's; 1st 0 2nd first 0 and 2nd 0, it comes to the state or 0 0 and on 0 it remains here. So, 0 0 0 or it may be any number of 1's, then two 0's. So, that is also fine. So, wherever it gets a 1 1 1 1, then 0 then again 1. So, whenever it gets a 1, it comes back to the first state and to come to the end, the last state it must have at least two 0's at the end. So, to summarize I can say it is all strings terminated by; terminated by at least two 0's.

So, this particular finite automata, it corresponds to all the strings that are terminated by at least two 0's 2 then, because whenever it is getting a 1, it is coming back to the initial state. So, from that point it is again starting. So, then it can you can see that it will not be able to it will not be able to go to the accepting state until it gets two successive 0's at the end and there is no more character. So, that way it will do this thing.

So, given a given such finite automata, so you can trace it and find out the corresponding language and you can try to write down the corresponding regular expression also. Like

here, so I have to write the regular expressions. So, it can be in the state 1 star, then it is any other string over 0 or 1 star any number of 0's and 1's, but then there has to be two 0's that the end ok. In fact, this 1 star is not necessary.

So, you can forget about this 1 star also. So, this can be simplified as 0 or 1 star followed by two 0's. So, if it gets two 0's it will come there, otherwise it will not. So, you can also try to from this regular expressions so, you can try to draw the automata. So, this it has to if it starts at a particular state, this is the initial state; then getting one 0, it comes to this state; getting another 0, it comes to this state.

I am trying to derive this automata from this regular expression. So, for two 0's, it come here. Now, there can be any number of 0's after this so, I should say so this is also not correct. So, there should be a 0 star at the end. So, as long as they are 0's, so it will remain there; but if it is similarly, as long as there are 1's, it will remain here. So, if it gets a 1, it will come back to this point. If it gets a 1, it will come back to this point. So, this way you can draw the corresponding. So, this is the derivations. So, ultimately you see that we have come to the same diagram as we have it here. So, from the regular definition, you can come to the finite automata or from the finite automata you can come back to the regular expression. So, both of them can be done.

(Refer Slide Time: 10:31)

## And Another Example

- Alphabet still {0, 1}

- The operation of the automaton is not completely defined by the input

30

What about this one? So, you see that alphabet is a still 0 and 1. So, it says that as long as you get 1, you remain this state and then, on getting a single 1, you come to this state.

So, the valid strings if we try to enumerate 1 is a valid string, but say 1 1 is it a valid string? Like for the 1st one, it remains here; the 2nd one, it goes there.

So, that is possible then so, then there were three 1's so, that is also possible because two 1's, it will revolved here and third 1, it will come here. Of course, there is a problem in this particular case as you can see that on 1; so I can go to this state or I can remain in this state. So, there are 2 transitions 2 possible transitions that are their corresponding to the symbol 1 at the start state. So, this is there is some sort of non determinism.

So, apparently it seems that this is not correct, but as such. So, we will see that this defines a new type of finite automata which is known as non deterministic finite automata, where there may be multiple transitions that are possible from one state based on the same input symbol and if any of those transitions can lead me to a final state final accepting state, then we are happy. So, we can do it do this thing.

So, we will come to that as you proceed. Now, the operation of this automata is not completely defined by the input. So, because we have to we have because we have to take some non deterministic decision like at this point on getting on whether we remain here or we go from here to here.

So, there is a decision and that decision is not depicted by the finite automata completely. Whereas, in the previous examples so, here there is no non determinism like at every point. So, if there are two transitions going out from a state. So, they are labeled differently. So, on 0 you go here or 1 you come here. So, this is deterministic, but here we have got non determinism. So, the operation is not completely defined by the input.

(Refer Slide Time: 12:51)

Epsilon Moves

- Another kind of transition:  $\epsilon$ -moves
- Machine can move from state A to state B without reading input

31

Another thing that we have is something called epsilon moves. Sometime from state A to state B, we may have a transition whose level is epsilon. So, epsilon means it will no input symbol will be consumed for doing a transition from A to B. The machine simply transits from A to B, but does not consume any input.

So, machine can move from state A to state B without reading the input. So, if we have these two things in our hand that is epsilon moves and this non determinism; so non deterministic decision. So, they will be constituting a different type of automata which is known as Non Deterministic Automata or NFA.

(Refer Slide Time: 13:31)

Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No  $\epsilon$ -moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have  $\epsilon$ -moves
- *Finite automata have finite memory*
  - Need only to encode the current state

Finite Automata

So, right now we can see that this finite automata, it can be divided into 2 classes. So, this finite automata; so, this is called finite because number of states that we have in it is finite in nature. Now, this finite automata there are two different types; one is deterministic, another is nondeterministic.

So, there are different types of finite automata. So, in a deterministic finite automata so that we have seen at the beginning; so, there is one transition per input per state. So, from every state on some input symbol, there is a single transition that is defined. So, it will not happen that from a particular state; from a particular state, I will have two different transitions and going to two different states and labeled with the same input symbols.

(Refer Slide Time: 14:21)

Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No  $\epsilon$ -moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have  $\epsilon$ -moves
- *Finite automata have finite memory*
  - Need only to encode the current state

FREE ONLINE EDUCATION  
swayam

So, this will never happen for a finite deterministic finite automata and also there is no epsilon moves. So, there is no transition whose level is epsilon. On the other hand, this nondeterministic finite automata, so I can have multiple transition so, this is possible. So, from one particular state on getting the same input symbol, so it can go to two different states. So, while doing the operation, it will not deterministically select any of those two states and make a transition there.

So, you can have multiple transition for one input in a given state and also can have epsilon movements, epsilon moves can also be there. So, this defines the Nondeterministic Finite Automata or NFA. Finite automata have finite memory because it needs only to encode the current state. So, only the current state has to be remembered. So, it can forget what that how did it reach that particular state. Like if this is the start state and at present you are in this particular state.

So, you have come to this state where a number of state transitions. So, in between there are many states by which where input consumption we have come to this state. So, for the operation of the finite automata I do not need to remember this path; I do not need to remember this path, only remembering the current state is sufficient. So, that is the advantage that we have with finite automata. So, it can work with finite amount of memory and it needs to encode only the current state.

(Refer Slide Time: 16:01)

The slide title is "Execution of Finite Automata". It contains two bullet points:

- A DFA can take only one path through the state graph
  - Completely determined by input
- NFAs can choose
  - Whether to make  $\epsilon$ -moves
  - Which of multiple transitions for a single input to take

A diagram illustrates the difference: a DFA state graph shows a single path from state S1 to S2 on input 'a'. A NFA state graph shows multiple paths from state S1 to S2 on input 'a', including an  $\epsilon$ -move directly to S2.

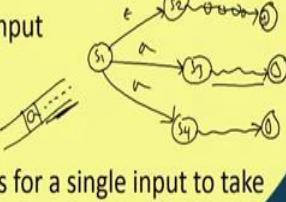
So, a deterministic finite automata, it executes like this it can take only one path through the state graph. So, it is completely determined by input. So, from 1 state, so there is no possibility that it can go to 2 different states on the same input symbols. So, it is operation is determined by the input. whereas, nondeterministic finite automata there is a choice like if there is a so, it may so happen that some epsilon movements are defined like from state say S 1, there is a transition to state S 2 which is marked as epsilon and there is another transition marked on input a to the state S 3.

Now, if the current input symbol is say a; your input pointer is here and the current input symbol is a. Now, whether you make this transition or you do not consume the input and go to the you follow this epsilon path? So, that decision is nondeterministic in nature. So, whether to take epsilon move or whether to consume the input and go to the next state? So, that is the decision question. So, so that is one problem whether to take epsilon move. Another point is if there are multiple transitions defined on a single input like it may so happen that from S state S 1. So, there is another transition defined.

(Refer Slide Time: 17:25)

## Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input
- NFAs can choose
  - Whether to make  $\epsilon$ -moves
  - Which of multiple transitions for a single input to take



So, from S 1 there is one epsilon move to state S 2. There is a transition on a to state S 3 and there is another transition on a to state S 4. Now, the situation is more complex because now on the input symbol a whether it should take epsilon transition or it should take the transition to state S 3 or S 4. So, that becomes non deterministic. So, this nondeterministic automata it can take some non deterministic decision and can and can take any of this transition.

Now, whether the anyhow whether this is valid or not. So, that depends like if there are some final states like this and then, suppose these are the final states and by taking this path may be it can eventually reach this one and by taking this path it can eventually reach here and by taking this path it can eventually reach there. Now, when you have taken this path whether you will reach this final state or not that depends on the transitions that you have on this path and the inputs that you have on your input stream after this.

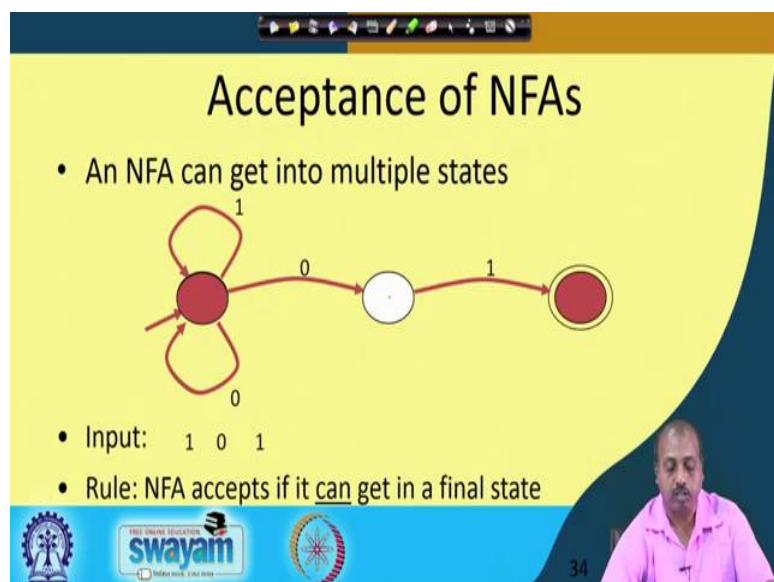
After whatever input symbols are available. So, if the by that you can reach this one, then it is ok. Similarly, on getting this path, so if it is on a you consume it, then the remaining part if it can be reached via this inputs by doing it in proper transitions, then that is also ok.

Now, it is designed in such a fashion that both of them are not possible. So, it cannot be that by taking epsilon path and this remaining input you can reach here as well as taking

a and using remaining inputs you can reach here. So, if that is not there if it is if the automata is design in such a fashion that only one of these final states can be reached by the next few input symbols, then you see that if you take a non deterministic decision which is wrong so, that will not lead you to a final state

And in that case you can come back, you can follow the other path to see whether that can lead you to a final state or not. So, that way we can take a decision about this which path to follow and by means of say backtracking and all and this way we can have automata operations. So, apparently it sees that this is a bit clumsy. But we will see that this is better in many cases because the total number of states in the system may be less compared to a deterministic finite automata. So, we will see that after sometime.

(Refer Slide Time: 20:03)



So, this is an example, like an NFA can get into multiple states like say for this is the start state as I was telling; once I get 0, it can remain in this state or it can go here. However, on getting one it definitely remains on this state. But on getting 0, it can go to the next state or it can remain in the next state. So, it is in this state so, if the input is 1, then definitely it will make the transition like this. If the input is 0, we have a question. So, it can either go here or it can go there. So, if it gets a 1 here, now you see that from this choice it can go to the final state. But from this choice, it cannot go to the final state or getting 1. So, NFA will accept the string if it can get in a final state.

So, this 1 0 1 is acceptable in this particular language because it can find a path by which it can reach a final state. Of course, in between it write some other path, but that did not reach to the final state so, they are not considered.

(Refer Slide Time: 21:15)

NFA vs. DFA (1)

- NFAs and DFAs recognize the same set of languages (regular languages)
- DFAs are easier to implement
  - There are no choices to consider

35

So, NFA versus DFA; so, apparently as I said that it appears the DFAs are better, but NFAs and DFAs can recognize the same set of languages. So, whatever whichever language is acceptable by an NFA it is also acceptable by a DFA and whichever language we can for whichever language you can construct a DFA, we can also construct an NFA. So, power wise both of them are equally powerful. So, both of them accepts can accept same set of languages. However, as per as implementation is concerned DFA's are easier to implement because we do not have any choice to considered.

So, we have got only for every state we have got the transition which are well defined. So, there is no choice to be considered. So, DFAs are easier to be implemented so, that is there.

(Refer Slide Time: 22:11)

NFA

DFA

All binary strings ending with two zeros  
1010100

- For a given language the NFA can be simpler than the DFA
- DFA can be exponentially larger than NFA

FREE ONLINE EDUCATION  
swayam

But, NFAs we have got choice so, we have got difficulty in the implementation because it may need back tracking. However, it may so happened that NFAs may be simpler compared to DFA. Like here I have got an; here I have got an NFA, where the language that we consider here is all strings that ends with two 0's. So, all binary strings ending with two 0's so, this is the for this particular case, we are trying to formulate the NFA and DFA.

Now you see this NFA as long as it there is 1 so, it remains in this state. If it gets a 0, there is a choice it can come to this state or it can remain here, but if it gets another so if it get two 0's, it will go there. So, if we if have a string like say 1 0 1 0 1 0 0, then getting the first one, so it will remain in this state because there is no choice. At the second 0, I have a choice. So, I can go like this or I can go like this. If I go like this and come to this state, then at the next time I get a 1, I find that this is the string becomes unacceptable.

So, that was not a good that was not a valid choice so, we come back ok. So, input pointer is retracted and we come to this state and so, you take this transition now and now after this if I get a 1. So, it will remain here, then again 0. So, there is a choice between this path and this path. So, this way, it can that the NFA can operate ok.

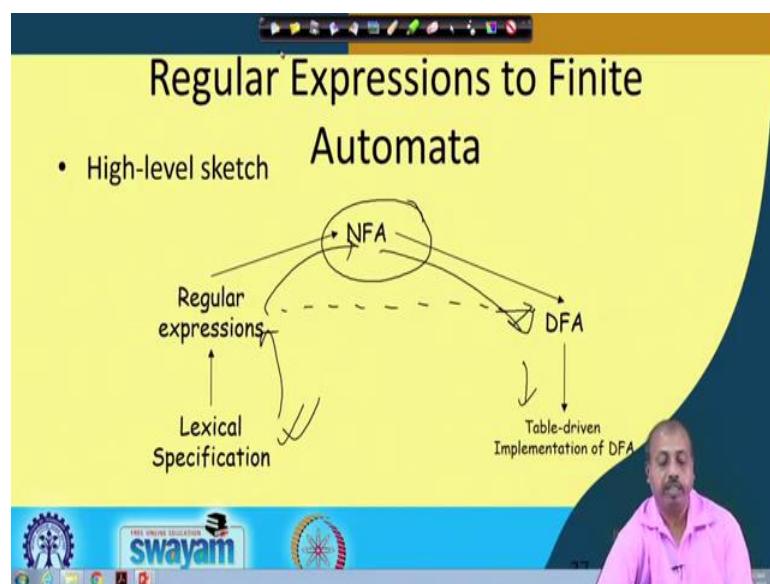
So, it can wherever whenever it is finding that I cannot proceed further, it can retract the input pointer and it can come back to the previous state from where it can it can just try out a new option; whereas, for DFA you do not have that thing. So, when the DFA's

structure is quiet complex. So, as you can see from the diagram itself. So, this also access the same language that is all strings ending with at least two 0's.

So, it is also accepting that, but the point is that this is in this case we have got that that the diagram is much more complex. So, DFA can be exponentially larger than NFA so, this is the thing. So, here of course, the number of states in NFA and DFA are same. So, it does not have much difficulty, but if we will see some example later where we will see that the number of states in a DFA can be much larger than the much much larger than the number of states in the NFA. So, that is why for the say for the representation of a regular expression.

So, it is better that we do it in terms of NFA and while writing the corresponding recognizer. So, if that the program can be made records if so, it can try out the alternatives, it can backtrack and try out the other part like that. So that way, the recognizer can be made complex, but space wise so it will be saving the space because I do not need to have. So, many states the number of states maybe much less compared to the DFA. So, that is the general observation.

(Refer Slide Time: 25:33)



So, regular expression to finite automata so, we have got we start with this lexical specification and then, from here we come to the regular expression. From the regular expression we converted into NFA and from NFA we convert into DFA and for DFA we have got table driven implementation of DFA that is how this whole thing works ok.

So, sometimes so of course it is possible that you do not make this NFA and go from regular expression directly to DFA that is also possible. But, normally it is not done. So, we follow the other revenue because there are there can be very well defined rule for converting regular expression to NFA. So, we will see some examples later.

(Refer Slide Time: 26:21)

The slide has a yellow header with the title "Regular Expressions to NFA (1)". Below the title is a bulleted list:

- For each kind of rexp, define an NFA
  - Notation: NFA for rexp A
- For  $\epsilon$
- For input a

Below the list are three diagrams of NFAs:

- For  $A$ : A state with a self-loop labeled  $A$ . An incoming arrow points to the state, and an outgoing arrow points from the state.
- For  $\epsilon$ : A state with a self-loop labeled  $\epsilon$ . An incoming arrow points to the state, and an outgoing arrow points from the state.
- For input  $a$ : A state with a single outgoing arrow labeled  $a$  pointing to another state.

The footer of the slide includes logos for Swayam and the Indian Space Research Organisation (ISRO), and the number 38.

So, like, so this is the so, for say the NFA for this regular expression A. So, for different type of regular expressions, we can have it is like this. So, it has got the input for some regular expression A. So, it has got an input pointer and this is the final states so, there will be a start say.

So, for different types of A, I can have different type of regular expression. For example, if the regular expression A is only epsilon, then it is represented by the NFA while we have got this initial state, then this epsilon telling it to the taking into the final step. If the regular expression A is equal to this small a, it has got only the single character small a in it.

So, it can go from this start state on this small a, it goes to the finals state. So, this way I can define regular expressions for individual symbols that we have for the in the regular expression what you are single symbols appears so I can make it like that.

(Refer Slide Time: 27:31)

Regular Expressions to NFA (2)

- For AB

```
graph LR; A((A)) -- ε --> B((B))
```

- For A | B

```
graph LR; S1(( )) -- ε --> S2((B)); S2 -- ε --> S3((( )); S1 -- ε --> S3;
```

Now, if you are trying to if you have got two such NFA's constructed for the regular expression A and B, then you can join them together by if the regular expression is say AB, Then, you can join them together by doing it like this. So, the final state of A, from the final state of A put an epsilon transition to the initial state of B; whereas, if the regular expression is A or B so and you have already constructed the NFA's for this regular expressions sub parts A and A and B separately. Then, you can add a few extra states like so you take this these 2 extra states and add epsilon transitions.

From the initial state, so you put an epsilon transition to the initial set of B. Similarly, put an epsilon transition to the initial state of A and from the final state of B, put an epsilon transition to the total final state of the whole expression. Similarly, from the final state of B, put an epsilon transition to the final state. So this way for A or B, we can make the NFA. So, what I want to mean is that depending upon the regular expressions. So, you can construct it to the small portions of it and then, combined them together for getting the overall regular expression.

(Refer Slide Time: 28:55)

### Regular Expressions to NFA (3)

- For  $A^*$

The diagram illustrates the construction of an NFA for the regular expression  $A^*$ . It features two states: an initial state (left) and a final state (right). An epsilon transition ( $\epsilon$ ) leads from the initial state to itself. A transition labeled 'A' leads from the initial state to the final state. Another epsilon transition ( $\epsilon$ ) leads from the final state back to the initial state.

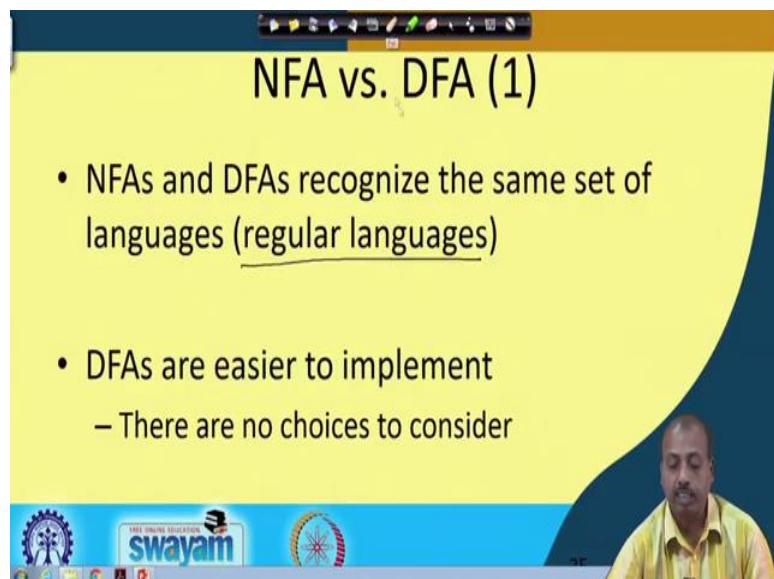
So this is another very interesting one for  $A^*$ . So I assume that I have already constructed the NFA for  $A$ . So this is the NFA for  $A$  that is already constructed. So, this part is there. Now, for getting  $A^*$  I should have 0 or more occurrence of  $A$ . So, this so, I take one initial state and a final state as extra and put an epsilon transition from the initial state to the final state. So that way this is this is actually correspond to the 0 occurrence of  $A$ .

Now, to make 1 occurrence, so you can go it like this to through an epsilon transition you come to  $A$ , make this  $A$  and then there is an epsilon transition for the any final state of  $A$  back to the initial state of initial state of the regular expressions. So you go back like this and again by another epsilon you come to the final state. So if you trace this diagram, then you can understand that this will realize the regular expression  $A^*$  ok. So, this way we can convert regular expressions to NFA and there is well defined rules like that for doing the conversion.

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

**Lecture – 11**  
**Lexical Analysis (Contd.)**

(Refer Slide Time: 00:15)



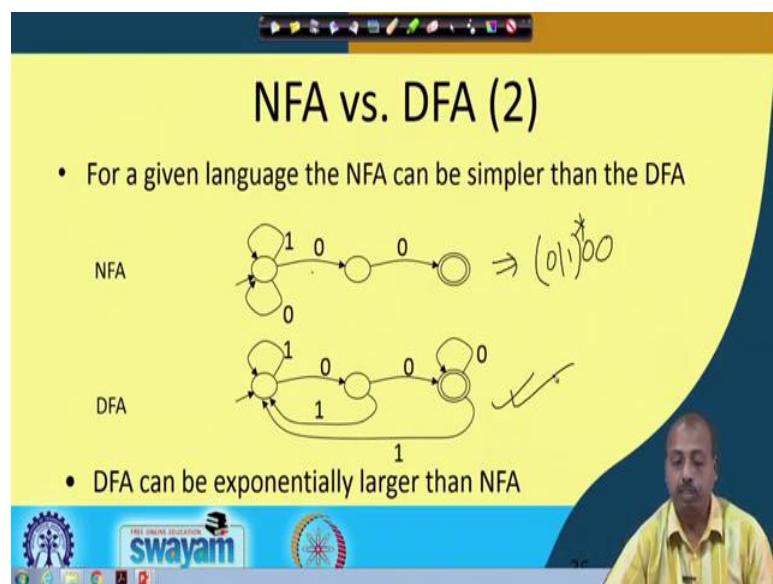
So, in our last class, we have seen two types of finite automata, Non – Deterministic Automata and Deterministic Finite Automata that is NFA and DFA. So, NFA is non-deterministic in the sense that at some state so we have got a choice, whether to make a transition on a particular input to a one state or the other, or also there is there were epsilon transitions. So, without consuming any input, whether the machine should make a transition from one state to the next; whereas, the deterministic finite automata did not have this type of non-determinism. So, there everything is deterministic.

So, from one state on a particular input pattern or input symbol, it will go to one of the next state. So, if there is no transition defined for a particular input symbol, then it is an error. So, the string gets rejected, but if we compare this NFA and DFA, apparently it seems that NFAs may be more powerful, because it has got non-determinism in it. So, or somebody may have say that DFA is better, because DFA there is no non-determinism. But theoretically it can be proved that this NFA and DFA, they are equally powerful as far as the set of languages that they can accept.

So, this set of languages will be called regular languages. So, we will define regular languages later, when we go to the in the grammar portion in the chapter on parser. But, these regular languages are a class of language, where it is a bit restrictive compare to many others, but this NFA and DFA both of them accept the same set of regular languages. So, any regular language will be accepted by a DFA, you can have a corresponding NFA.

And whenever you have got an NFA, so you can get a corresponding DFA; so, there is power wise, there is no difference. However, DFAs are easier to implement, because there is no choice to consider. So, we can have a we can do it very easily. So, we can have some sort of table driven algorithm by which you can make the transitions for the DFA, but there is other issues that we have a like this.

(Refer Slide Time: 02:31)

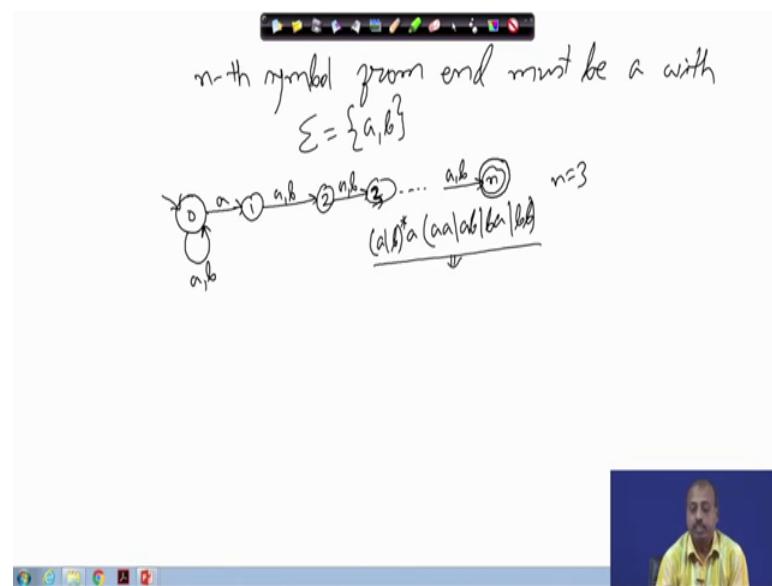


Other issues that we have are a NFA versus DFA. For a given language the NFA can be much simpler compared to DFA. So, here is an example that we see. So, we have got the in the so in the NFA. So, this is accepting the strings which are ending with two 0s. So, in this case it is it is ending with two 0s; so it is 0 0, it is going to the end state or final state, otherwise it is remaining in the initial state.

The same NFA or same regular expression; so, if you want to so this regular expression, corresponding to this is 0 or 1 star followed by 0 0. So, this regular expression so, if you want to realize by means of DFA, you will get a diagram like this. So, you see the

number of states in both the finite automata are same, however there are more number of transitions. And we will see an example, where we will find that this DFA can be exponentially larger than NFA. So, we will next look into an example which will be doing that.

(Refer Slide Time: 03:45)



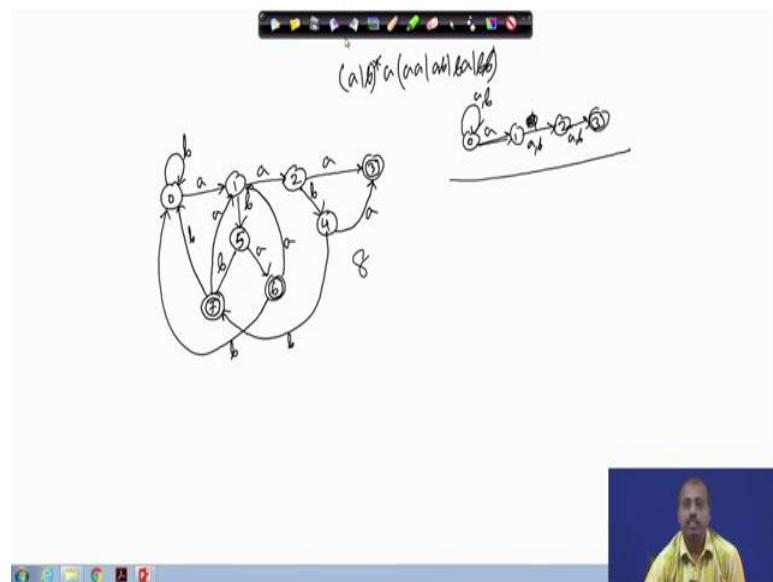
So, consider a regular expression for this particular language. Suppose, I have got it is written like this; so it is the n-th character or n-th symbol from end must be a with alphabet set, with the alphabet set sigma equal to a b. Now, if we try to construct the corresponding finite automata, the NFA construction may be like this. So, this is the start state 0. So, as long as the so, so you get a or b you remain in this state. So, I said that the n-th character must be a. So, if this is the n-th character, so on a; it makes a transition to state 1.

Then either on a or b, it goes to state to 2; then either on a or b, it goes to state 2 state 3 so this way it goes on ok. So, it is may be that the final state. So, this is our state n, this is our state n and on getting a, b it goes here. So, you see that here so, so this is the final state. So, this is a final state. So, you so this if the n-th character is a, then only the strings are going to be accepted, but for the same thing for the same regular expression. So, if we try to draw the corresponding DFA, then the situation will be very different.

So, let us take say n equal to 3 ok, let us take n equal to 3 and for that we try to construct the corresponding DFA. So, if n, n is equal to 3, then the corresponding regular

expression is given by a or b star followed by a; and then the next few characters may be a, a b, next two characters may be b a or b b, so this is the regular expression. In fact, this particular case where this is general that is there that the n-th character from n. So, you cannot have a regular expression for that ok, but when n becomes fixed so like n equal to 3, then you can write down this regular equation. Now, for this regular expression so if I try to draw the corresponding DFA, then it will be something like this.

(Refer Slide Time: 06:23)



So, let, let us take a new page and then do it. So, my regular expression is a or b star, then a, then a or b, a sorry, sorry. So, this a or b star, then a then you have got a or a b, or b a or b b. So, this is the regular expression. So, if you start at state 0, this is my state 0, as long as you get b you remain in this state. So, if you get an a, so basically I am trying to capture this a a a sequence; so you come to state 1, then if you get another a come to state 2, if you get another a come to state 3, which is a final state. So, this captures the sequence a a a.

Now, if you get after so after getting the first a, if you get a b that is you have to define the transition on b. So, one transition on b, you can may be from this from this state, if you get a transition b, you come to a state 4. And from this state 4, there can be several situations like if you get an a, if you get an a, then you see how did you come here is that the so if you get an a here, then you can go to this state, because then the third character from the end is really becoming an a. And if that is end of the string, then this is valid so

this is the thing. Similarly, from this state 1, if you get an, if you get a b, you come to state number 5.

And from state number 5, if you get an a, you come to state 6. And again state 6 is a final state, because the third character from the end is in a. So, to reach state 6, we have come via this path a, b, a. So, this a is the final these are the third, the third most character. So, this 6 is also a final state ok. Now, from this state 5, if you get a b, if you get a b, then also you come to a state 7, but this 7 is also a final state, because here also the last characters are b, b and a so that way it is correct. However, from state 7, if you get a b, so you go back to the first state, state 0 and from here also from state 6 also if you get a b, you go to state 0 ok.

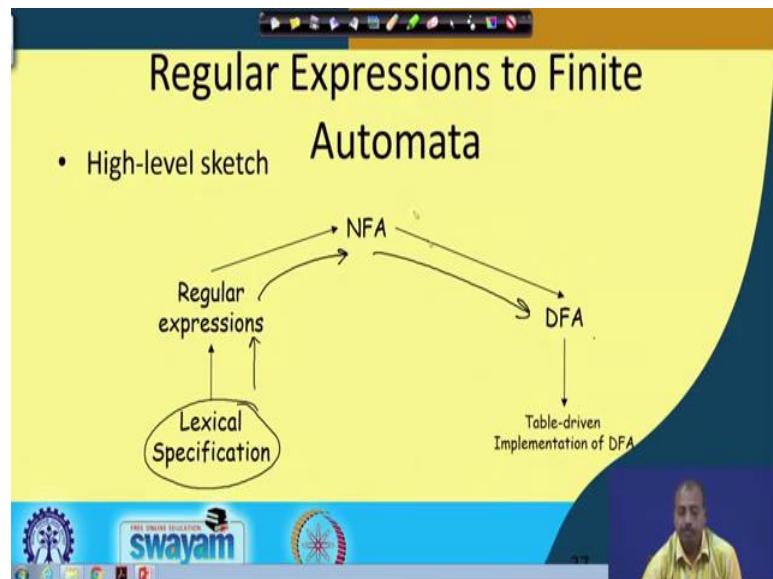
Now, from state 7, if you get an a, then you have to come to state 1, because now a a a, a b a and all those sequences can be managed. So, on a you have to come here. Similarly, from here also on getting a you have to come to state a. And from state 4, if you get a b, state 4, if you get a b, then it is like a b b, so then that is also a final state 7. So, you see that what is happening is the diagram is becoming very complex. Number of states in this case is equal to 8 compared to the previous like, if you are doing it using NFA, then you will do it like this. So, at this state it is a or b then on a you go here and then so this is 0 1, then on a or b you go there a or b so state 2, 3, and 3 being a final state so this was the NFA.

So, getting the third most character as a so it is going to a final state. So, here it was a three of only a 4 state machine, here I have got a 8 state machine. So, that way number of states is increasing significantly. So, that can happen ok. So, in a more pathetic situation so, it may go to the number of states they become exponential in the case of a DFA. So, DFAs are better, because if you are trying to develop a recognizer, then working with DFA is easier, because everything is deterministic. So, that the detection algorithm there will be non-determinism in it, but at the same time the size of the DFA maybe you have very large, so compared to the NFA.

So, for our understanding, we should we will work with NFAs most of the time and then do something so that the NFA can be converted to DFA for the sake of for the sake of implementation. So, this is exactly what is done by the lexical analysis tools. Like given the regular expression, so it first constructs the NFA; and from the NFA, it converts it

into a DFA. So, in the coming few, few slideshow you are going to see how this is going to happen.

(Refer Slide Time: 11:34)



So, regular expression to finite automata the avenue that is taken is like this that is we will start with the lexical specification from the lexical specification, which is an English language description like the third most character from the end should be a like that there is electrical specification, from there we write down the regular expressions.

So, after the regular expressions have been written. So, we convert these regular expressions to NFA and then from this NFA we convert it into DFA. So, there is a conversion algorithm that we will see that can convert non-deterministic finite automata to a deterministic finite automata. So, basically all the equivalent states of the non-deterministic finite automata. So, they are clubbed together and they have constituted one state in the DFA.

And for regular expression to NFA, so this parts so there are some well defined rules, which will convert, which will do this thing regular expression to NFA. And once we have got the DFA, then we can have a table driven implementation of the DFA, so which will be which will give us the recognizer for the language.

(Refer Slide Time: 12:50)

The slide has a yellow header with the title "Regular Expressions to NFA (1)". Below the title, there is a bulleted list:

- For each kind of rexp, define an NFA
  - Notation: NFA for rexp A

To the right of the list, there is a handwritten note "A = ε". Below the list, there are three diagrams of NFAs:

- For  $\epsilon$ : A diagram showing a single state with a self-loop arrow labeled  $\epsilon$ .
- For input a: A diagram showing a single state with a single outgoing arrow labeled  $a$ .
- For rexp A: A diagram showing a state with an incoming arrow and a final state with two concentric circles. There is also a handwritten note "A = ε" near this diagram.

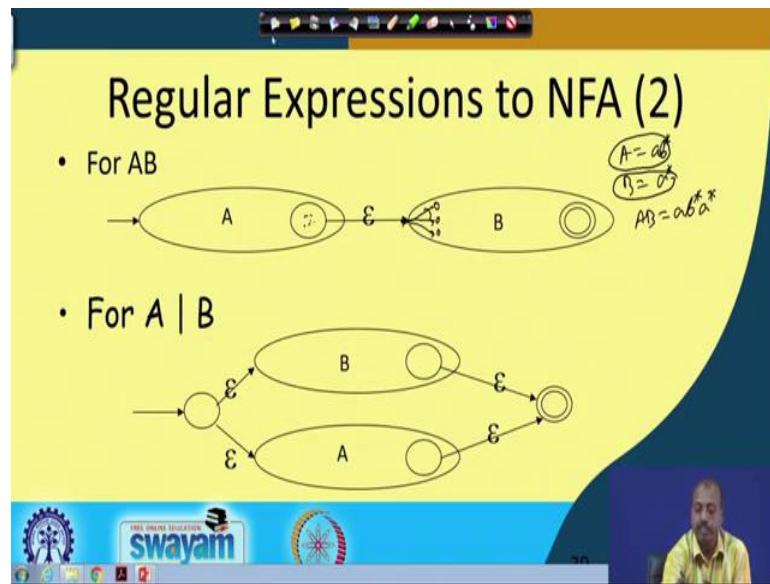
The footer of the slide features the "swayam" logo and other navigation icons. On the right side of the slide, there is a small video window showing a person speaking.

So, next we will slowly go into the portion that regular expression to NFA like say, we will be using this particular notation suppose we have got a regular expression a. So, the corresponding NFA will symbolically represent like this. So this a is basically there are a number of states, so which a there are number transition between them which constitute the NFA for a for the regular expression a. But for our for our construction purpose, we will represent it in this fashion that is as if there is a final state, and there is an initial state and this arrow means that this will go to the initial state of the NFA for a. So, this notation we will be using quite a, quite a few times in our discussion.

Next we will consider individual symbols like say epsilon. So, if your regular expression a is equal to epsilon, then a regular expression is converted into an NFA like this. So, this is the start state identified by arrow coming to it, then this is the final state identified by two concentric cycles circles. And there is an edge from initial state to final state whose label is epsilon. So, this is the regular NFA corresponding to the regular expression epsilon.

So, this is the regular NFA corresponding to the regular expression epsilon. Similarly, the for the regular expression a the corresponding NFA for corresponding NFA is represented like this that is from the initial state and the final state and then edge between them is labeled by a. So, this way we will be this will be doing the things for individual symbols that we have in the regular expression.

(Refer Slide Time: 14:33)



Now, what about two regular expressions coming together like  $A$  is a regular expression and  $B$  is a regular expression. Now we have got a new regular expression, which is  $A$  followed by  $B$  like say your  $A$  maybe the regular expression say  $a b^*$  and say,  $B$  maybe the regular expression  $a^*$ . So,  $A, B$  is the regular expression  $a b^*, a^*$ .

So, how to construct the corresponding NFA with the situation in which you have already constructed the NFA for  $A$  and we have already constructed the NFA for  $B$ . So, that part is the NFA for  $A$  second part is the NFA for  $B$ . So, from the final state of  $A$ , we add an epsilon transition to the initial state of  $B$ . And this state no more remains the final state, because this is not the end of the regular expression. So, this way we do it. So, if there are multiple initial states in this multiple initial states, in this, then there will be multiple such epsilon transitions or we can say of course, normally we have one initial state.

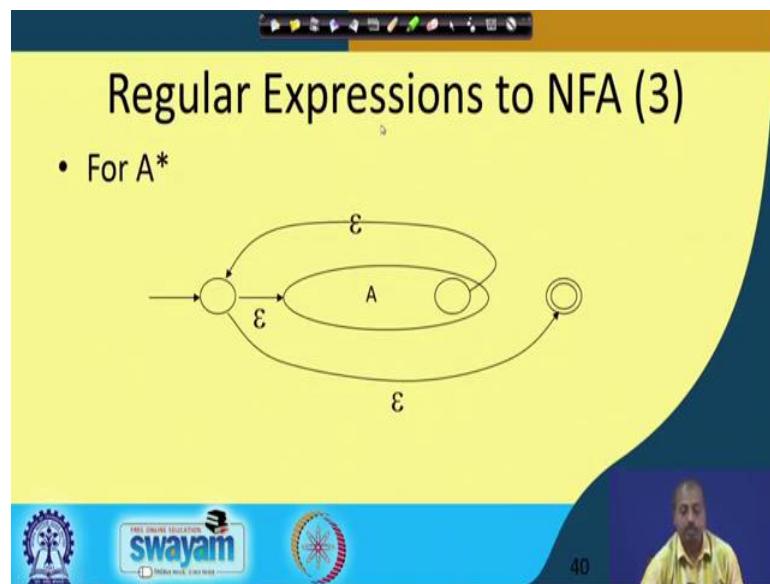
So, the possibility is that there are many final states in  $A$  and this is the initial state in  $B$ . So, which we have edge coming from all of them and all of them will be labeled epsilon. So, this way we do it ok. So, for  $A, B$  so, one regular expression followed by another regular expression. So, we take the final states of the first regular expression from there, we add epsilon transitions to the start state of the next NFA for next regular expression. Next we will see how to do the regular expression for  $A \mid B$ . So,  $A$  and  $B$  are two

regular expressions and the new regular expression formed is the regular expression A or regular expression B.

And as our assumption so, we have already constructed the NFA for regular expression A, which is this part, we have already constructed the NFA for regular expression B, which is this part. Now, for constructing the NFA for A or B what we do, we introduce two new states one initial state and one final state and the initial state or the start state becomes the start state of the whole regular expression A or B.

We add epsilon transitions to the initial states of A and b. So, like this. So, here we add epsilon transitions here and add epsilon transitions to this the initial states. And from the final states of A and B, we add epsilon transitions to the final state of the, the newly created final state the final state for the whole expression. So, this is how we do it for A or B. So, these are quite intuitive, because you can go either by this path or you can go by this path, but and in the first case when it is AB. So, you have to go through like this. So, A followed by B going from A to B, we do not consume any new symbols. Similarly, while taking a decision like whether, we should go to A or B, we do not consume any input ok, so that way this part is realizing the regular expressions AB and A or B.

(Refer Slide Time: 18:02)



So, the more complex situation comes, when we have got the regular expression A star. So, we already have the NFA for A. So, this is the NFA for A, then we add two new states to the system one is the one is the start state, and one is a one is a start state, and

one is a final state, and we add epsilon transition. So, from the start state we add epsilon transition to the start state of A; and from the final state of A, we add epsilon transition to the newly created start state. And also we add epsilon transition from the start state the newly created start state to the newly created final states. And the final state of A, it is no more the final state ok so, because the transitions are being added ok.

So, you see in this way if we do it, then the advantage that we get is suppose there is A zero occurrence of A because, it is A star. So, suppose we have got zero occurrence of A then it will be going by this epsilon transition. If it is one occurrence of A, then it will go like this with epsilon it will come to A, it will go like this then through epsilon, it will come back. Here then by epsilon it will come to the final state one occurrence. If it, if there are two occurrences of A, then it will go like this, then come back by epsilon like this. And then again go through A one more cycle coming via this epsilon, and again come here and then via this epsilon, it will go to the final state.

So, as many time you want, so depending upon the number of times this A appears, so you can just go on moving around this NFA and finally, when you have reached the end, so you can come by this epsilon to the final state. So, this way you see that we have got a few set of rules for constructing. The NFA for individual types of regular expression individual symbols epsilon, then consecutive appearance of two regular expressions or of two regular expression and this 1 or 0 or more occurrences of the regular expression. So, this way we can construct the NFA.

(Refer Slide Time: 20:13)

## Example of RegExp -> NFA conversion

- Consider the regular expression  
 $(1 \mid 0)^*1$
- The NFA is

```
graph LR; A((A)) -- ε --> B((B)); A -- ε --> C((C)); B -- ε --> C; B -- ε --> D((D)); C -- ε --> E((E)); C -- ε --> F((F)); D -- ε --> F; E -- ε --> G((G)); F -- ε --> G; G -- ε --> H((H)); H -- ε --> I((I)); I -- 1 --> J((J)); J --- J
```

FREE ONLINE EDUCATION **swayam**

Let us take an example suppose, we have got this particular regular expression 1 or 0 star 1. Now, how do we do it? So, if see if you look into individual components, then we have got a regular expression 1, regular expression 0 and regular expression 1. Now, how do you construct a regular expression for 1? So, it is like this. So, this is this is a final state this is the start, start state, and it is labeled with 1. Similarly this 0 regular expression for 0, so you can construct it like this on 0 it goes here.

Now, once we have done that the next part is to construct the regular expression for 1 or 0. Now, for 1 or 0 what is we so, so we take the so this is our A ok. So, this is our A this is our B, this is our B. And then we add two new states like B and G there and from B, we add epsilon transition to C and an epsilon transition to D and from E and F, we add epsilon transitions to G, so that way.

So, now, this whole thing it represents the NFA corresponding to the regular expression 1 or 0 after that I have to take a star over this. Now, for taking star over this what we do, we add a new state H here. Now, from and add a new star state a here. And from the star state A, we add an epsilon transition to the initial state of 1 and 0, then from the final state of the NFA for 1 or 0. So, we add a transition epsilon transition back to A and also from A, we add an epsilon transition to the state H the final state H. So, that gives me. So, at this point we have got the regular expression.

So, till this much we have got the regular expression NFA for the regular expression 1 or 0 star with that we have to add the NFA for regular expression one and this is the regular this is the portion of regular expression 1. So, from the final state of the first NFA we add an epsilon transition to the initial state of the, the other NFAs the NFA for 1. So, this is done and epsilon transition is added. So, that way this whole thing becomes the NFA corresponding to this regular expression.

Of course there is lot of scope of optimization, because you see that there are epsilon transitions like from E to G and then again another epsilon transition to A. So, ideally I should be able to do it like this. So, this part this state G can be eliminated and both of them from E and F, I can take them back to a directly. So, this type of optimizations can be done, but going by the set of rules, we get this particular NFA ok.

(Refer Slide Time: 23:40)



Now, once this NFA is there what can we do with this. So, the next thing that we can do is. So, we are at this point. So, we have we started with the lexical specification from there the regular expression from there we got the NFA. The next thing to do is to convert this NFA to DFA and once we are done with that then we will have a table driven implementation of the DFA. Now, how to convert the NFA to DFA, so that part we have to consider see now.

(Refer Slide Time: 24:10)

The slide has a yellow header bar with the title 'NFA to DFA. The Trick'. Below the title is a bulleted list of steps:

- Simulate the NFA
- Each state of resulting DFA  
= a non-empty subset of states of the NFA
- Start state  
= the set of NFA states reachable through  $\epsilon$ -moves from NFA start state
- Add a transition  $S \xrightarrow{a} S'$  to DFA iff
  - $S'$  is the set of NFA states reachable from the states in  $S$  after seeing the input  $a$ 
    - considering  $\epsilon$ -moves as well

On the right side of the slide, there is a small diagram of an NFA with states A, B, C, D, E, F, G, H, I. State A is the start state. Transitions are labeled with inputs: A to B (0), B to D (0), D to F (0), F to G (0), G to A (0), and G to H (1). State H is a final state.

The footer of the slide features the 'swayam' logo and other navigation icons.

So, the basic idea is that for NFA to DFA conversion, the idea is to simulate the NFA. So, we simulate the non-deterministic finite automata. And each state of the result of resulting DFA is equivalent to a non-empty subset of states for the NFA. So, you start with the NFA and start simulating for a particular input symbol A and see how far it is going for with without consuming any more input.

Like here, so if you see this one if you, see say this example, so you see if you start with say 0. So, without if you are starting at state A, then without consuming any more input with the just an input 0, you can come from A to B to D to F to G, and then it is not adding anything more, because now it is going to come back to A only. So, this A, B, D, F, G, so all these are all these are can be simulated for the input symbol 0. So, to construct the start state, we consider, we consider the set of NFA states, which are reachable through epsilon moves from the start state.

So, in this particular case, A is the start state. So you take all the states, which can be reached from a using epsilon transitions only. So the set of states are A, B, C, D, H and this I. So all of them can be reached by the epsilon transitions only from state A. So all of them taken together, so that set will constitute the start state of the DFA; so, this is what is said that this the start state of the DFA is equal to the set of NFA states reachable through epsilon moves from the NFA start state.

Then from any state  $S$  we added transition to transition to state  $S$  dash in the DFA on input  $A$  if and only if so this condition is satisfied.  $S$  dash is the set of NFA states reachable from the states in  $S$  after seeing the input  $A$  considering epsilon moves as well. So, what has happened is I have started with the start state. So, a number of NFA states number of NFA states, so they have given me the DFA state, DFA start state  $S$ .

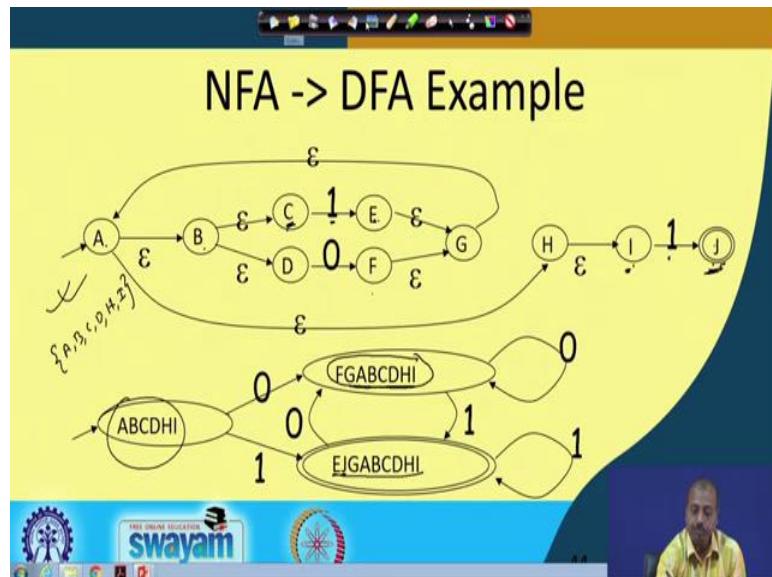
Now from these states so from these individual states on  $A$  you see where can you go. So maybe this state on  $a$  it takes here, so this state on  $a$  takes here. So others they do not have transitions on  $A$ . In that case these two plus all the states which can be reached from here via epsilon transitions. So all those states that can be reached from these two states where epsilon transition, so that will constitute the state  $S$  dash. So, in that DFA, I will have a state  $S$ ; I will have a state  $S$  dash, and the transition will be labeled with  $A$ . So, this is how we will add transitions to the DFA.

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

**Lecture – 12**  
**Lexical Analysis (Contd)**

So next we will consider an example for this NFA to DFA conversion, suppose we take the NFA that we have just constructed the before so, this is the NFA ok. So, now, it is this NFA if we are trying to run a DFA construction algorithm then I said that first you take from the start state using epsilon wherever you can go.

(Refer Slide Time: 00:41)



Like from this start state using epsilon you can come to the start state itself A then B C D then this H and I. So, this A B C D H I so, they there so, that is the if I say that is the set. So, this is the set that can be reached A B C D H I. So, this set constitutes the start state of the DFA. So, for the sake of our understanding we just give the name of the state at as A B C D H I. So, we can give any other name 1 2 3 4 like that, but for our understanding we write it as A B C D H I. Now, I have to see that on inputs where they can go from the states A B C D H I say it from state A on 0 it is not defined.

Only from state D on 0 you can come to F and if you come to F from F on epsilon you can go to G and using epsilon you can come back to A and from A on epsilon now you can go to all the states A B C D H I as we have constructed. So, I can say that from this

state go by 0 you can come to the state F and from F on epsilon transitions you can go to all the remaining states.

So this new set F G A B C D H I it constitutes the next DFA state. Similarly, on from this state A B C D H I if you see the transitions on 1. So, you can see that from state C on getting 1 it can come to state E and from there are no other transition of course, there is another transition from I you can make a transition on 1 to J. So, these are the two cases in one case your C can take you to E and I can take you to J. So, naturally so, this E and G E and J so, they are the new states that have been identified. Now from E and J on epsilon where can you go from E on epsilon you can come to J from G and from G on epsilon you can come to A and from A you can reach all these states.

Similarly, from J there is no epsilon transition defined so, from J you cannot go anywhere. So you see that this is the new state that I can reach from this previous state on getting a 1. So this since this state is not yet identified so, we just make it, we make it a new DFA state and you see that since the state J is included here where J is a final state of the NFA J is included here. So, J becomes the J becomes so, this becomes one of the final states for the DFA.

Now, from this state or let us come back to this F G A B C D H I now what happens on a 0. So, since 0 transition is defined only from D to F so, from D it can go to F and once you are in F from epsilon transition so, you can come to G from G you can come to A. So, essentially it gives rise to the same subset of states so natural so, that way from 0 for 0 the DFA remains in the same state.

From the state E J G A B C D H I so, if you get a 0 then from this state D on 0 it will go to this state F and from the F on epsilon transition so, you can go to all the remaining states G A B C D H A H I. So, that way I get the same so, I get the same state for on 0. So, from the state if you get a 0 you come to this state and from this state E G E J G A B C D H I. So, if you get a 1 then it remains in the state. So, because on 1 from state C it will come to E and from state E and from state I it will come to stage J and then on epsilon transitions. So, it will give rise to the same subset of states.

So, if this is the same state that is created for the DFA. So, this way we can convert one NFA into DFA using some algorithm. And you see that whether this DFA is more complex or not than that NFA so, that depends on the type of regular expression that we

are considering. In this case it is not complex because, now I have got only 3 states in the DFA compared to 1 2 3 4 5 6 7 8 9 10 states in the NFA.

And the number of transitions are also less; however, as we have seen previously that this may not always be the case in many cases in many situations so, it can give rise to the exponential number of states the DFA. So, but for algorithms to the detection algorithms to run so, it is better if we have the DFA implementation. So, this way we will we will be converting the regular expression to NFA and from the NFA we will be converting it into DFA.

(Refer Slide Time: 06:19)

**NFA to DFA. Remark.**

- An NFA may be in many states at any time
- How many different states ?
- If there are N states, the NFA must be in some subset of those N states
- How many non-empty subsets are there?  
–  $2^N - 1$  = finitely many, but exponentially many

Diagram: State A has two outgoing arrows labeled 'a' to states B and C. A box highlights "Current = {B, C}".

Some remarks on this NFA to DFA conversion and NFA maybe in many states at any time. So, what I mean is that suppose I have got a transition in the NFA from one state a there is a transition on input symbol A to state B and another transition on input symbol a to state C. So, if you are simulating this NFA and you are currently at state A if your current state is A and the next input symbol is a small a, then the next the current state becomes the set BC.

So, this is the meaning that you have if you try to stimulate the NFA then you have to consider in terms of current set of states in which the system may be there the NFA the auto motor may be there. So, then NFA may be in many states at any time, but how many different states if there are N states the NFA must be in some subset of those N states and how many subsets can be there?

So, there can be  $2^N$  minus 1 finitely many, but exponentially many subsets because it can do many of those any of those subsets like. So, any of these like if I have got say what I want to mean is suppose my NFA has got set of states  $S_1 S_2$  up to  $S_N$ .

(Refer Slide Time: 07:47)

**NFA to DFA. Remark**

- An NFA may be in many states at any time
- How many different states?
- If there are  $N$  states, the NFA must be in some subset of those  $N$  states
- How many non-empty subsets are there?  
 $- 2^N - 1$  = finitely many, but exponentially many

Now, while you are doing the transitions so, it can be in any of the subsets because from  $S_1$  it may be from  $S_1$  it may go to it may be going to  $S_2$  on  $a$  and it may be going to another state  $S_3$  on  $a$  or it may be that on  $S_1$  it goes to  $S_2 S_3$  and  $S_4$  on  $a$ .

So, this way so, these are the different subsets. So,  $S_2 S_3$  becomes the subset than  $S_2 S_3 S_4$  becomes the subset so, these are the possibility. So, in the worst case so, you can have  $2^N$  minus 1 different subsets that are possible and it can go to any of those states any of those state of state of subsets of states on a particular input symbol. So, and in fact, when you are doing this conversion from NFA to DFA so, what we are essentially doing is that we are constructing the equivalent set of equivalent subsets of the NFA.

And then for each of these set of sub subsets so each of these the subset of states so, we are calling it to be a state in the DFA. So, DFA that is why DFA number of state can become exponential. So, in this case so, this can happen. So, you can it can give rise to exponentially many number of a subsets so, that create that makes it difficult for simulating the NFA.

(Refer Slide Time: 09:29)

The slide has a yellow header with the title 'Implementation'. In the top right corner, there is a small diagram of a DFA with states  $S_1$  and  $S_2$ , and a transition  $S_1 \xrightarrow{a} S_2$ . Below the title, there is a bulleted list:

- A DFA can be implemented by a 2D table  $T$ 
  - One dimension is “states”
  - Other dimension is “input symbols”
  - For every transition  $S_i \xrightarrow{a} S_k$  define  $T[i,a] = k$
- DFA “execution”
  - If in state  $S_i$  and input  $a$ , read  $T[i,a] = k$  and skip to state  $S_k$
  - Very efficient

At the bottom of the slide, there is a blue footer with the 'swayam' logo and other navigation icons.

Whereas, for DFA we do not have this problem a DFA if you are trying to implement it can be realised by a two dimensional table  $T$  one dimension is the states and the other dimension is the is the input symbol. So, what I say is that you maintain a table for a DFA that this is the current state or present state this is the present state and you have got the input symbols say this a b so, these are say c d suppose I have got 4 input symbol.

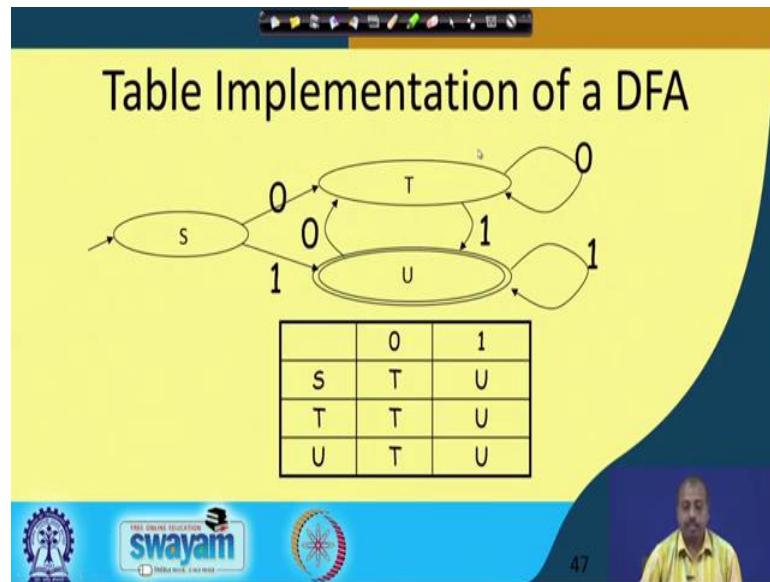
So, these are the different states  $S_1 S_2 S_3$  etcetera. So, you just note down on say from  $S_i$  to  $S_k$  say from  $S_1$  to  $S_2$  there is a transition on input symbol  $a$ . So, I write here the next state is  $S_2$ . So, accordingly we define that in this table  $i \ k \ i \ a$  equal to  $k$  that is from state  $i$  on getting input  $a$  it will go to state  $k$ . So, I can maintain a 2 dimensional table where this  $T[i][a] = k$  or this. So, a table corresponding to the state  $S_1$  and input  $a$  is  $S_2$ .

So, this is how I can represent the DFA and for the DFA execution for simulating the DFA if we are in state  $S_i$  with an input  $a$  with a read  $T[i][a]$  equal to  $k$  and skip to the state  $S_k$ . So, so, this is how the simulation will be done. So, we also have got an input string so, this is the input string that we have and at present the input pointer is somewhere here.

So, if this symbol is  $a$  at the present state is  $S_i$  and we will consult this table and find out what is the entry for  $T[i][a]$  in the table. So, and if the entry is say  $k$  then we will tell we will update the current state to state  $k$  and we will advance the input pointer to the next

position. So, that we can have a very efficient algorithm for simulating the DFA where as for simulating NFA you need to remember the current set of states. So, there can be a number of present states at which the system may be there so, it makes it difficult.

(Refer Slide Time: 11:47)



So, for example, this is the DFA that we have constructed and we if we name the states as S T and U S T and U so, I can have a table like this. So, there only in my alphabet there are only two symbols 0 and 1. So, from state S on input 0 it goes to state T on say input 1 it goes to state U.

Similarly, from the state T on input 0 it remains in state T on input 1 it comes to state U from state U on input 0 it remains in the state it comes to state T and on 1 it remains in state U. So, this way I can have a very simple table that can represent the DFA and then this DFA that can be simulated as we have discussed previously.

(Refer Slide Time: 12:39)

## Implementation (Cont.)

- NFA → DFA conversion is at the heart of tools such as lex, flex or jflex
- But, DFAs can be huge
- In practice, lex-like tools trade off speed for space in the choice of NFA and DFA representations

So, NFA to DFA conversion is the heart of tools such as lex flex or jflex. So, this lex flex and jflex so, they are some automated tools that have been developed by compiler designers. So, even these they come as some utility in many of the operating systems. So, they were originally proposed to come with the UNIX operating system and later on many other operating systems. So, so they have integrated them as some tools for system development. So, if you are trying to develop a compiler for a language so, if the lexical analyzer part can be designed using these tools.

And these tools so, they have the capability to convert NFA to DFA. So, they first take the regular expressions for the language for which you are going to get the lexical analyzer and then convert those regular definite regular expressions to their corresponding NFA and convert those NFA's to DFA.

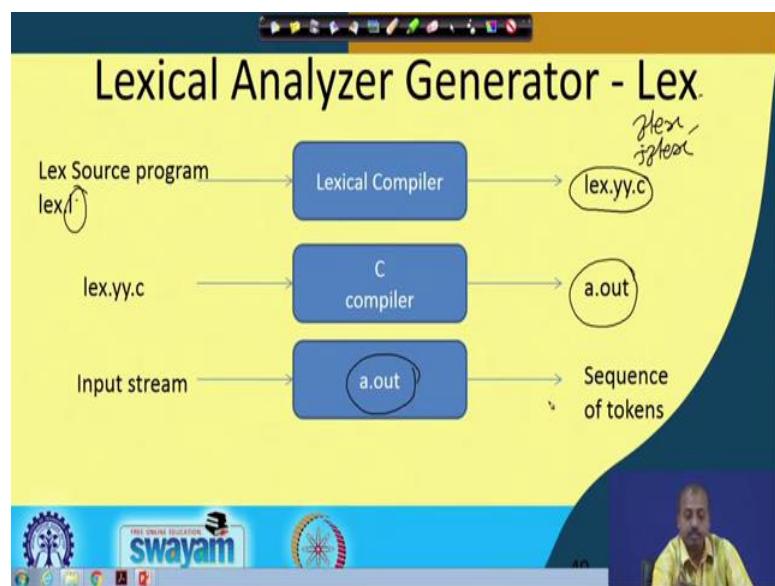
However, the DFA's can be large can be huge because them and they normally it is produced as a text file as a C program or C ++ program or java program and the program size becomes very large and it is very difficult to understand like what is there in the code, but we most of the time we take it blindly that this is the code produced is correct and since the tools are running for quite some time over the years.

So, they are time tested so, there is very I should say every little chance that there are some bugs in those tools. So, we can use those tools with a lot of faith on them and most of the time what happens is if there is some problem the problem is due to the regular

expressions that we have written. So, there is some problem in the regular expressions themselves.

So, this lex like tools so, the trade off speed for space in the choice of NFA and DFA representation. So, we have there is a trade off between space and speed of operation. So, they do this as they do this a trade off to see like what can be a reasonably good speed and reasonably good representation the NFA or DFA.

(Refer Slide Time: 14:47)



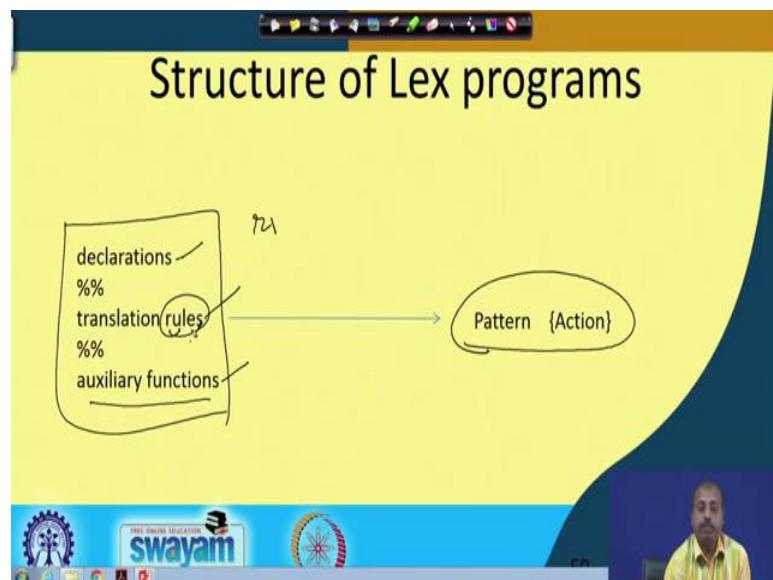
So, next we will look into one particular tool lexical analyzer generator called lex. So, this is the original tool as I said that this was proposed in the UNIX operating system this came with the UNIX operating system. So, this is a lexical analyzer generator that it can give generate ul lexical analyzer given the source specification file.

So, for the source specification file so, you should have this extension dot l there is a lex source program. So, that is given to a lexical compiler the tool is a. So, we have got the 2 lex and there are some more tools like some improved version flex and jflex like that so, there are some improved version. But, what they essentially do is that they generate a c program lex dot yy dot c this lex and flex they generate the c program. So, that lex dot yy dot c and this lex dot y y dot c being a c program you can pass it through a compiler c compiler to generate the executable version a dot out and this a dot out is a lexical analyzer for the source a lex specification file.

So, if you give some input to this program which follows the above the specification that we have in this lex file then it can accept it can detect the tokens that are appearing in the input stream and return those tokens. And like many other jobs can be done some format manipulation and also everything can be done. So, this is basically a lexical analysis tool.

So, this input stream will be coming they will be given this a dot out will analyse this input stream and it will give you the sequence of tokens. So, it will act as the lexical analyzer.

(Refer Slide Time: 16:43)



So, we will see some sample form sample structure of this lex program. So, there is no we do not have scope for going to a detailed discussion on this lex tool. But, what we will do is that we will see some overview that how this lex tool operates or the specification file.

So, this is the lex specification file. So, this is the lex specification file. So, we have got it it has got 3 parts the declaration part the translation rules part and the auxiliary functions part. So, this declarations part so, these 3 parts they are separated by this double percentage marks so, this is the format. So, all the regular definitions and declarations so, they should be put in the declaration part.

Then what are the translation rules at what do you do when you see a particular token particular lex in matching with a rule then what do you have particular definition. For example, here I have put a regular expression say r r 1. So, here I have to tell for r 1 what are you going to do? So, these rules part we will be telling what to do with the when we see this particular thing. For example, if you find say some variable declaration and then maybe the action for that is to install the regular install the corresponding variable in to the symbol table.

So, that way we can do it. So and there are some auxiliary functions so, which may be used by this transition rules that we are writing here. So, these actions are written in a c code and those c codes get integrated into the lexical analyzer tool. And as an output so, it produces the patterns and the corresponding actions. So, this ultimately generates a switch case type of statement big switch case statement for each of these translation rules. So, it will have the corresponding actions which are given here so, this way this a lex programs will be structured.

(Refer Slide Time: 18:43)

```
%{
/* definitions of manifest constants
LT, LE, EQ, NE, GT, GE,
IF, THEN, ELSE, ID, NUMBER, RELOP */
}

/* regular definitions
delim [ \n]
ws (delim) +
letter[A-Za-z]
digit [0-9]
id (letter)([letter])*(digit)*
number (digit)+(\\.(digit)+)?(E(+|-)?(digit)+)? */

%%
{ws} /* no action and no return */
if {return(IF);}
then {return(THEN);}
else {return(ELSE);}
id {yval = (int)installID(); return(ID);}
number {yval = (int)installNum(); return(NUMBER);}

int installID() /* function to install the lexeme, whose first character is pointed to by ytext, and whose length is ylen, into the symbol table and return a pointer thereto */
int installNum() /* similar to installID, but puts numerical constants into a separate table */

}

```

So, this is a typical example. So, the first part the definition part is up to the first up to this much is the definition part ok. So, here you can say you can definitions of a manifest constant like LT LE EQ so, this way we can define all the tokens basically. So, these definitions they can they can be useful for this parts are for the time being they do not have any meaning.

So, this then comes this is this regular definitions. So, this one is so, do we are defining a regular definition delim delimiter which is blank tab and newline character. And they are put with put within this square bracket means it is the alternatives. So, we blank the blank character the tab character and the new line character. So, they will constitute the set of delimiters.

Then the white space is a regular definition whose actual definition is like this delim plus. So, that is 1 or more occurrence of this delimiter characters so, that is a white space. Then we are defining letter so, letter is the definition and the corresponding stream sequences is it can be this A to Z and then and then small a to small z.

So, that we can have this sequence of characters giving this regular definition then digit is 0 to 9 now an ID can be letter followed by letter or digit whole star. So, I do not have any other symbols. So, it says that the identifier it starts with a letter and the remaining characters are have to be letter or digit and the that is the star. We can define the number so, number is defined like this that I should have at least 1 digit followed by a dot or then digit plus.

So, if you have a dot then you have got a digit in the 1 or more digits then there is a question mark. So, question mark means this entire thing this part there may be 0 or 1 occurrence. So, that will take care of the situation that is at sometimes we do not have the fractional part after decimal point. So, it will be taking care of that and if you are having fractional part then definitely has to be 1 digit before that.

So, you cannot write like 0.25 like that. So, you have to write like 0.25 because, this 0 in that case will match with this digit plus part and this then this point will be there and this digit plus will match with this 25 ok. But, if you write like 0.25 then this first digit is not there so, it will not be matching. So, this type of minute details may be there. So, that you have to see and this whenever we are writing the specification the lex specification we have to be careful at those points. Then comes the exponentiation part that is at 10 to the power part so, this symbol E must be there then you can have plus minus and there is a their question mark means it is 0 or 1 occurrence.

So, we can have a sign or we may not have a sign. So, if we have a sign that can be plus or minus or there if there is no sign then that is also fine in that case it is taken as plus. Then followed by digit plus so, you should have at least 1 digit available. So, whenever

you are writing say 0.25 into 10 to 10 to the power 5. So, we have to write like 0.25 then E 5. So, you have to do it like this.

So, now we have to come to the action part. So, for whitespace we do not have any action so, no action and no return. So, it is simply written as no action. Then if you have the token i if you have the symbol like a regular definition if. So, if it matches with the if the input is like i if state if word then it will return the token if ok. So, it is there so, if it gets the if it gets the sequence t it is sequence then and then this particular regular expression will match and in that case the action part is to return the token THEN.

So, in this way whatever regular definitions we have. So, you can put it here the corresponding regular expressions and in the action part we have to give the corresponding c code. So, here it is returning else as we know that the parser will call the lexi analysis tool for the next token and this is how that tokens will be returned. So, it is then else etcetera so, they are all defined as tokens in the language and so, it will be it is in the parser will know these tokens and the lexical analyzer will also know the tokens. So, it will be returning those tokens to the parser.

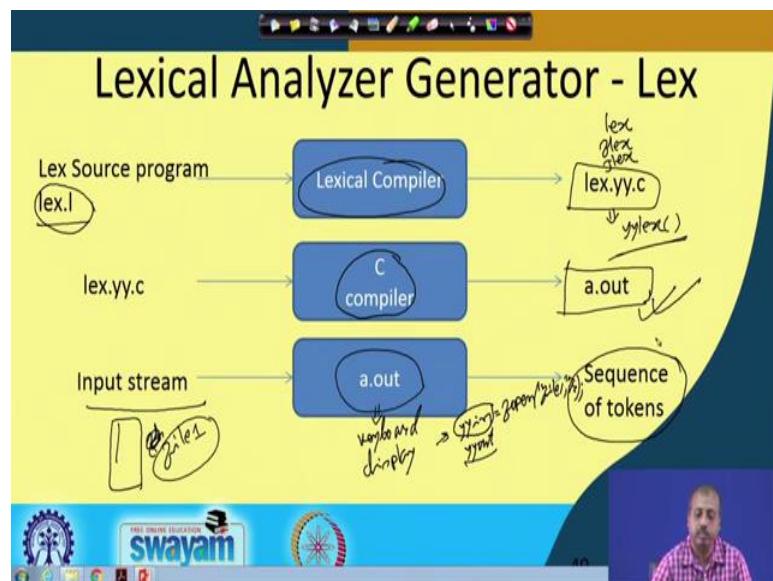
Now, this id so, this id whenever you are putting within this brace; so, it mean that this will be this definition will be replaced. So, it will be taking this definition and it will be replaced here. So now it is doing it like this that yy 1 val. So, this is a special variable that this lexical analyse analyser tool and parsing tool they will understand. So, it is a some value of it is the value of the token that is returned.

So I said that every token has got some attributes. So this value is one of the attribute for the token and the value is returned by the variable y y 1 val. So, we are setting the yy 1 val to be the install ID function will be called. So, that will be installing the corresponding identifier into the symbol table and it will return the corresponding index of the table and it will return the identifier. And in case of number so, it will be converting it will install the number into the number table and it will return the token number.

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

**Lecture – 13**  
**Lexical Analysis (Contd)**

(Refer Slide Time: 00:15)



So, next we will discuss on a Lexical Analyzer generator tool called lex. So far whatever techniques we have seen like say from the regular expressions. So, we can construct NFA from NFA you can convert into DFA and all that. So, we can always write some programming language using some programming language. So, we can write programs for implementing those lexical analyzers, but many times it becomes cumbersome. Particularly if the language for which you are designing the lexical analyzer is huge. So, there are number of tokens or identifiers in that language then it becomes very difficult to construct or write the corresponding program.

So, actually this tool lex it came with the UNIX operating systems. So, UNIX operating system designers they found that this is a very useful tool if we can have a set of application software. So, which in some sense became a part and parcel of the UNIX operating system that can be used for writing the compilers automatically.

So, in that direction the first tool that has been developed is this lexical analyzer tool. So, here it says that for a particular input language. So, you can write down a specification

file for the tokens that you have in the language and so, corresponding regular definitions and all and put them into a file called this lex dot l. So, this lex dot l is a is a file is a text file where you write down the regular definition and corresponding actions. Like when that particular regular definition is found in the input stream, what the program is suppose to do.

So, it may for example, if it is an identifier so it may like to install that identifier into the symbol table, if it is say some number so it may try to get the corresponding integer value right. If it is an integer number ultimate the input program is nothing but an ST sequence. So, from that it may want to convert it into a proper number. So, that way so, those things are done by this lexical analysis tool.

Apart from that so, we can write down some portions or some actions so, that this white spaces are skipped. So, like that all those regular definitions that you can come in the source language programs. So, their definitions are written in this file lex dot l, and there is a tool called lexical analyzers. So, original UNIX operating system it came up with the tool called lex. And later on so, there are many updated versions of it one is the known as flex another is for java enabled things, so there is lex and all so, these are there.

But, essentially all of them are same in the sense that they are generating a lexical analyzer. And if you pass this your lex dot l file through this lexical compiler then it generates a c program lex dot yy dot c. This lex dot yy dot c it has got built in function in it is. So, if you open this file you will find that there is a function yy lex. So, this yy lex function so, you can call it repetitively and it will return the next token that is available on the input stream.

So, if you just write a main program where it repetitively calls this yy lex function till end of file is reached. So, if you do something like this then it will be returning you all the tokens that are appearing in the input program. So, that is one possibility, but the way this lexical analyzer is used by in a compiler design is that the parser. So, it in turn calls this yy lex function. So, whenever it needs a token it calls this yy lex function and it works like that. So, this lex dot yy dot c as I have said that this is a c program. So, this has to be passed though a C compiler for generating the object file.

So, this is written here as a dot out because in the UNIX operating system the default object file name of the object file is a dot out so we can give some other name also.

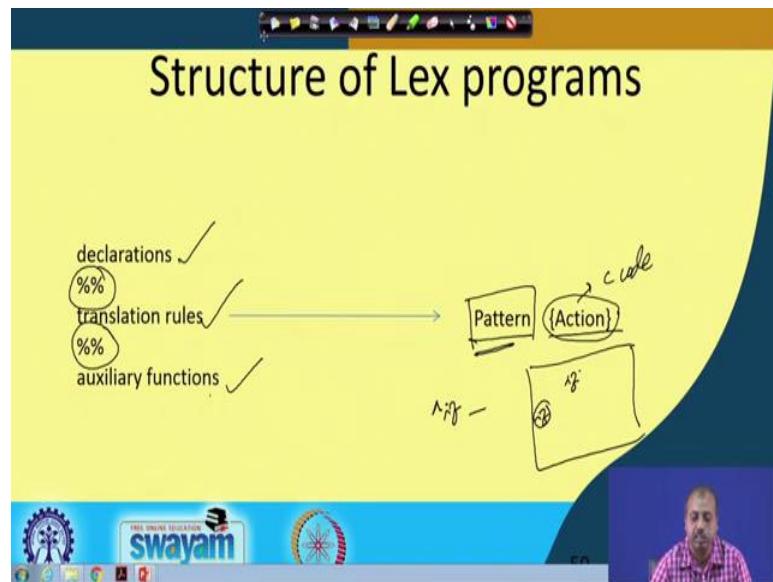
Whatever it is so, this is the executable file that is produced. So, this is actually the lexical analyzer this a dot out is the lexical analyzer. So, if you run this program a dot out and a you give it an input stream then so, this a dot out program. So, it accepts the input from the keyboard. And it produces the output to the display.

And there are some variables so, yy in and yy out. So, this y y in so, this is the input stream. So, this is the input stream pointer from where it will read the input. So, if you do not mention anything. So, by default this yy in is set to the keyboard and yy out is set to the display so, that way it operates. So, if you want that the input should be read from some other file some input some of the you are that for example, you might have return your input in some file the input stream maybe in the file say file dot say this is a name of the file is a file 1.

So, what you can do you can open this file and set yy in to this file. So, you can write like yy in equal to f open say file 1 in a read mode etcetera. So, this is the standard c program for a opening a file in the read mode. So, in that case from that point onwards this is yy in so, it will be accepting input from this file 1. So, whenever this yy lex function is called it will scan through this file and whatever be the next token that is available it will return it. So, this a dot out so, this lexical analyzer tool it will it will generate the sequence of tokens ok.

So, that is how this lexical analysis tool generator. So, that can be useful for compiler design or you can be if you have some very simple text formatting sort of application then it can also be done using this lex tool. So, you for certain input pattern input stream you want to generate something else so that can be done. So, you can do it using my some other tool that way you can use some action. So, we will see how the actions may be written so, that we can do something with for them.

(Refer Slide Time: 07:15)



So, structure of a lex program is like this. So, it is divided into three sections as I said earlier so, it is divided into three sections declarations translation rules and auxiliary functions. And they are separated by this double percentage characters ok. So, in the declaration part so, you can write down audio all your regular declarations etcetera so that you can have, that you may use with that you may be using while writing the regular expression in the translation rules.

So, in the translation rules part; so, you have to write the regular pattern and the corresponding action. So, we are not writing it a regular expression here because this pattern is a more powerful than the regular expression. So, in regular expression we can write something, but sometimes what is required is that we want that a particular input stream should match at the beginning of a line or should match at the end of a line.

So, like that so, if you want that sort of modification, so this lex is more flexible like. If for example, if I write a write something like this, this hat symbol and then I write i f. In that case if this is my input file then in some line if i f is the first two characters of the line then only it will find a match for this particular pattern. Whereas, if this if appears somewhere here then it will not find a match with this particular pattern because it is it is not starting with the beginning of the lines.

So, this way this pattern is more powerful compared to the regular expression that we are familiar with it theoretically. So, that is why it is written as pattern and in the pattern you

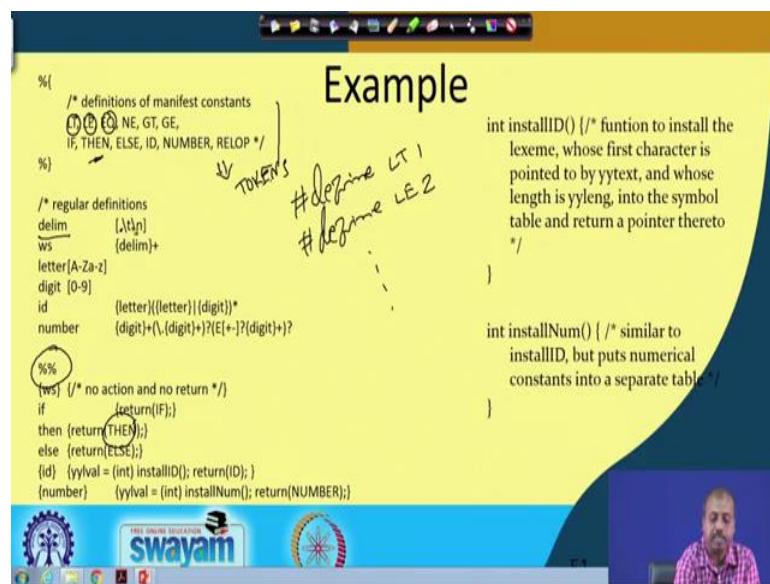
can give the corresponding action. So, this translation rules it will have all the patterns that the programs should recognise as a valid input patterns for the language and then it will have the corresponding action. And while writing the action; so action may be action is basically some c code. So, this is some again some c code and these actions are made part of the file lex dot yy dot c. So, whenever some pattern is found so the corresponding action is executed. So, this action maybe just printing the token that it has found or it may be returning the token to the parser. So, that way it can you can think about different actions like a if it is an identifier you can think about my installing the identifier into the symbol table. If it is a number then you may think about installing it into the number tables. So, like that way we can think about different actions.

So, those actions are taken care of by this action part. And while for writing this action we may need some additional function for example, like a for installing a symbol into the c identifier into the symbol table we may have a special function which is a installation routine.

So, that routine may be written as part of the auxiliary function and from this action part you can just call this routine.

So, that is how this lexical this lex is going to be helpful. So, we will see some example by in which this lex has been used.

(Refer Slide Time: 10:37)



So, here in this part sorry in this part you say that that the first percentage symbol it appears at this point. So before that whatever you have so, they are actually coming to the definition part. So, here you can you can define all the constants. So, they are manifest definitions of manifest constants so, they are actually the tokens that we have in the language. So, these are the tokens; all tokens that you have. So, you have to put it in this within this percent open braced percent close brace symbol.

So, this is again another standard that is introduced by this lex and a parser generator tool so they used this; so, within this symbol whatever you put so, they are taken as token. And what the system will do is that for each of these symbols that I have you are so, it will put a number ok. And later on so, if you look into the action part so, when it is returned then, so, it is actually the corresponding number that is assigned by this lexical analysis tool so that will be returned. So, that is so, they if you look into this lex dot yy dot c file. So, you will find that there are has defined lines that have been generated has defined LT 1 has defined LE 2. So, like that it has generated all those has defined for this part.

Then comes the regular definition part: so, in the regular definition parts so, we will be writing some definitions which will be used as pattern while we are going to the 2nd part.

So, here I have got the delimiters. So, delim is a regular definition so, we have got this is a blank this is a blank space tab and newline character. So, to just to say mean that this is a tab not the character t. So, it is written as reverse slash and then t. So, reverse slash in the UNIX operating system it is taken as an escape character or escape sequence. And the slash t it stands for the tab symbol similarly slash n stands for the new line symbol.

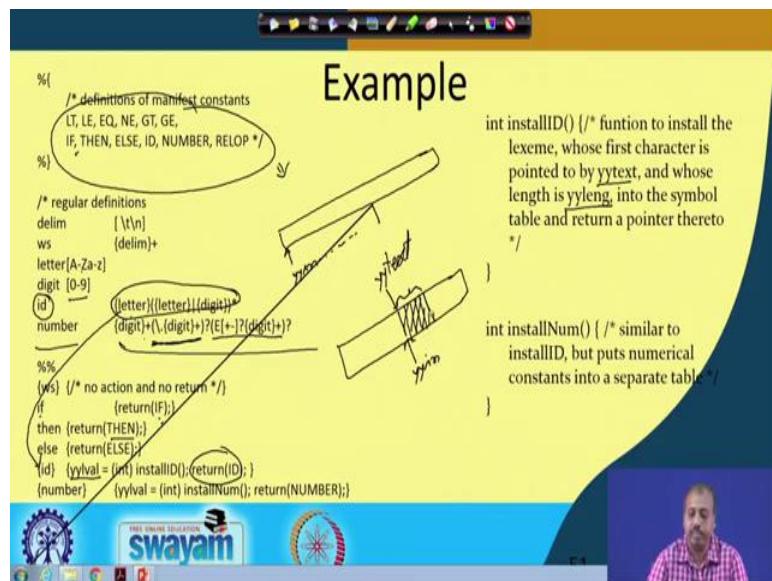
And as I said previously so, this is an aggregation of symbols so, this blank tab and newline they will be called delimiter. So, the white space is at a another definition so, this is within brace means that this is going to be replaced here. So, this delim definition of delim is going to be replaced here and then there is a plus so, any number of occurrence of those symbol. So, this is basically the white space that are occurring.

Then for the letter we say there then A to Z capital A to Z and small a to z these are the range that are there. So, these are the augmentations that are done with the basic regular

definition ok. So, in basic definitions so, this a to z was not used only for our understanding so, it has been introduced and lex has adopted that.

So, lex has adopted that a to z as a range small a to small z as the range. Similarly the digit is another definition for 0 to 9. Now it defines the id so, id it says it is a letter followed by letter or digit and then star. So, this is similar to regular definition. Then number: so digit plus so, one or more number of digits then we have got this thing this dot. And so, this dot has got some special meaning in case of a lex tool that is why there is escape sequence here the escape character back slash and then this dot. And then we have got digit plus. So, that after that I can have any number of digits so and this whole thing that is there is a exclamatory question mark. That means, this whole part and this a this part is optional.

(Refer Slide Time: 14:59)



So, you may or may not have the fractional part in a number. So, that is why it is like this. But at least one digit will be there then there can be a dot and any number of digits and after that there can be 10 to the power the exponent part. So, for again this part is optional this whole part is optional.

So, you have got E then after E we can write the digits directly 10 power 25 30 like that or I can have plus minus I can write like 10 power minus 30 or 10 power plus 25 like that. So, that is captured by this plus minus and this question mark. So, this question mark means that 1 or 0 or 1 character. So, that way we can have at most one occurrence

of those symbols. So, this number is the regular definition for a this part. So, any it defines the both integer and the floating point numbers ok.

Then after that it comes to the third part of the regular expression; 3rd part of the lex tool that is this portion. So, what we are doing here is that we are so for whitespace there is no action and no return. So, if that yy lex function is called and the yy lex function finds that the yy in pointer is at some white space. So, it will simply advance the yy end pointer, so it will not return anything.

Then if it finds the characters i and f in that case, so it matches with this particular definition i f and in that case the action that is taken is return if. So, as I have told you that each of these definitions so, they are converted into some constants, so that constant value will be returned. So, you have to interpret what is this constant. So, if you are calling it from the main program and in the main program you are just printing the return value from yy lex so, you will find that it is nothing but a number,

But, if you once you know this thing the values here and these values are the available in some header file and if we include that header file in your program and then you can just do a check and print the corresponding values. So, this if it finds that characters i and f then it will return the value which response to this is a capital I F symbol that constant. If it finds this four characters t h e n then it will be return the constant corresponding to THEN. If it finds e l s e then it will find ELSE. Then this id,id since it is within this place so this is the entire thing will be replaced here. So, this id will be replaced by whole thing. So, this is just this or that definition of identifier will come.

And then the action part you see there are two things; one part is it is returning the constant corresponding to id the token corresponding to id. But, before that it is setting another variable yy l val to this return value of this function install ID. So, install ID is a routine it is an auxiliary routine that this lexical analysis this specification file will have. So, this install ID routine is written here. So, this is the code is not written, but essentially what it will do it is to a this function is to install the lexeme whose first character is pointed to yy text and whose length is yy leng into the symbol table and return a pointer there to.

Now, there are say few special variables that we have come across the yy text and yy leng. Now, if this is your input stream as I said that this yy in pointer is initially pointing

to the first character and then as you are calling this yy lex function this yy in pointer advances. Suppose, at this point of time the yy in pointer is somewhere here. And then we have given a call to yy lex.

And here this is the next few characters that matches with some pattern. And after it has found the match what is done this yy text will point to this sequence of characters. So, this will be yy text will be pointing to this and yy leng variable will contain how many characters are there in this part ok. So, this information are available in these two variables. So, after your you have returned from yy lex function if you just print the character string yy text then you will find the a string that has been matched with a token that is returned.

And the yy leng it will have the length of the string; like how many characters were there in that particular string that has matched. So, this way you can think about a writing specification files. Similarly you see that this number; so this number when it matches with the number then it is the action that is taken is return number so that is the token that is there. But it also calls a function yy 1 val equal to install number. So, this yy 1 val so, this is another important variable yy 1 val.

(Refer Slide Time: 20:51)

The slide shows a YACC grammar definition:

```
%{
/* definitions of manifest constants
LT, LE, EQ, NE, GT, GE,
IF, THEN, ELSE, ID, NUMBER, RELOP */
}

/* regular definitions
delim    [ \n]
ws       {delim}+
letter[A-Za-z]
digit [0-9]
id       {letter}{letter}{digit}*
number   {digit}+{digit}*?{E+}-{digit}+?

%%
{ws} /* no action and no return */
if      {return(IF);}
then   {return(THEN);}
else   {return(ELSE);}
{id}   {yyval = (int)installID(); return(ID);}
{number} {yyval = (int)installNum(); return(NUMBER);}

int installID() /* funtion to install the
lexeme, whose first character is
pointed to by yytext, and whose
length is yylen, into the symbol
table and return a pointer thereto */

int installNum() /* similar to
installID, but puts numerical
constants into a separate table */
}
```

Handwritten notes on the slide:

- A large bracket groups the entire grammar definition under the heading "Example".
- An arrow points from the word "yyval" in the notes to the variable "yyval" in the grammar definition.
- A handwritten note "yyval is attribute value of the token" is written near the yyval in the grammar.
- Another handwritten note "yytext is attribute value of the token" is written near the yytext in the grammar.
- A handwritten note "yylen is attribute value of the token" is written near the yylen in the grammar.
- Arrows point from the explanatory text in the notes to the corresponding parts of the grammar definition.

So, it is you can read it as the value attribute of the token you can say it like this is the value at value attribute of the token return. So, the token returned is number, but what is

the corresponding value. So, if you want to return it so, then this yy 1 val variable can be used.

So, normally this parser it has got access to all these variables like yy text, y y leng, yy 1 val and all that and yy 1 val is going to be a very important attribute that you can see later. So, here for the number you have given a call to yy this install numbers. So, the number is installed into the number table you can say so, this is the corresponding function install num. So, puts the numerical constant into a separate table.

So, you assume that it is put on to a table. And the index of that table where it is put into so, that index is returned into the return from this function and that is assigned to yy 1 val. Similarly, for this install ID function also they wherever the place at which the particular lexeme was installed; this in the symbol table that index is returned and this yy 1 val is made to contain that particular value. So, this index values will be available to the lexical and the parser tool or some high level program which is going to use this lexical analyzer as part of it.

So, in this way you can write down a specification for different regular definitions. So, for the entire programming language you see that the designing lexical analyzer is so easy because you do not have to do any of those NFA DFA etcetera by hand. So, all that you need to do is to write down the corresponding regular expressions and think about the actions that should be taken if those the patterns are found on the input string. So, based on that you can design this lexical analyzer and rest of the thing is taken care of by the lex tool. So, it can automatically generate code for that portion of the tool.

So, this is how this lexical analysis tool works.

(Refer Slide Time: 23:27)

The slide has a dark blue header and a yellow main content area. The word 'Conclusion' is written in yellow on the left side. A bulleted list is present in the yellow area:

- Words of a language can be specified using regular expressions
- NFA and DFA can act as acceptors
- Regular expressions can be converted to NFA
- NFA can be converted to DFA
- Automated tool lex can be used to generate lexical analyser for a language

At the bottom, there is a video player interface showing a man speaking. The Swayam logo is visible on the right side of the video player.

So to summarize our discussion so, in this chapter so, we have seen several things. First of all words of a language can be specified using regular expression. So that is the first thing that we have seen, if we do not do that, if we cannot do that then it becomes a very clumsy because there are so many different types of words that will be possible. And it becomes the very difficult for the compiler designer to figure out what are the valid words and what are the invalid words. And so this regular expression based formalism it helps us to write down the, to tell what are the valid words of the language.

And once this regular expressions have been written. So we can design non deterministic finite automata and deterministic finite automata that can act as acceptors. So if suppose corresponding to a regular definition we can design finite automata. And then the input stream based on the input stream. So it can scan through the input and going from one state to the other when the input is exhausted if it comes to final state or an accepting state then that particular word is valid otherwise it is not valid, so that they can act as acceptors.

And we have seen techniques by which regular expressions can be converted to NFA and the after that the NFA can be converted to DFA. So, this way this whole process there exist a very sound theoretical foundation for determine valid words of a language. And after that as an add on so this with the from UNIX operating system. So we have got that tool lex that can be used to generate lexical analyzer for a language.

And then you can also in the other operating systems also these tools now have been put it. So if they are in other systems also they are available and several augmentations have also been done, but it will require quite some time to discuss on them. So, I would suggest that you refer to some manual for this lex tool to get a detailed understanding of how this tool can work ok.

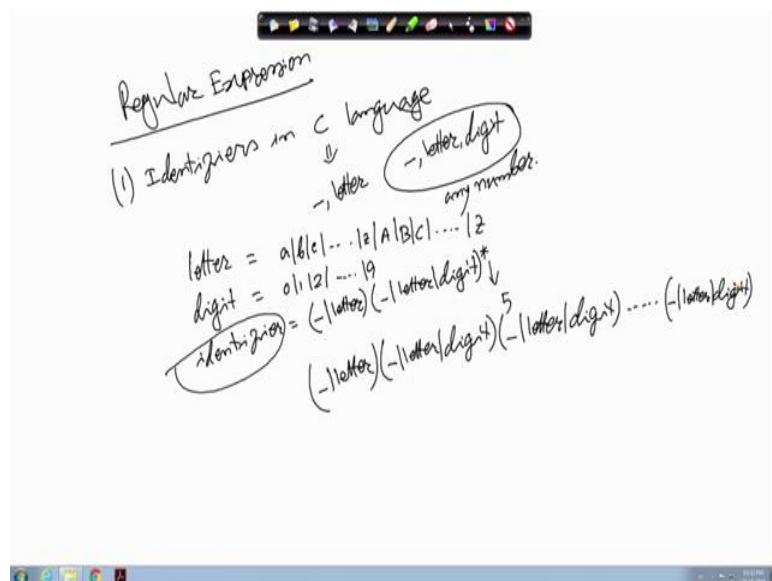
So that way we can design this lexical analysis tool. So, with that we conclude this part; of course we will look into quite a few examples in the next classes.

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

**Lecture – 14**  
**Lexical Analysis (Contd)**

So, next we will be looking into a few examples of this regular expression and what do they mean; how to write the regular expression; how to convert them to NFA DFA etcetera. Because that should give us a good practice and I would suggest that you practice on your own taking many other examples available in different sources ok. So, to start with we will be looking into we will try to write a few regular expressions. We will try to write a few regular expressions.

(Refer Slide Time: 00:45)



The first one that we will try to write is for identifiers in C language; identifiers in C language. So this is very common. So, almost all programming language it will have some identifier. So if you are designing something some compiler for some programming language. So this is the first step that you have to solve. So you have to write down some regular expression for identifier. So, first you need to consult the language manual and if you consult the C language manual, it says that it has to be the first character can be underscore or a letter and it can be followed by underscore letter or digit.

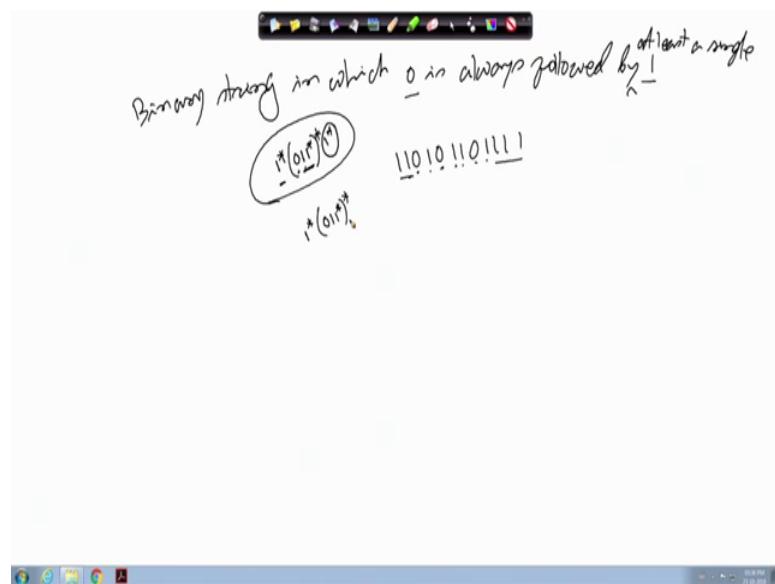
So, this can be any number of digits. So, this can be any number. So, that if you want to design. Then, what you have to do is that first we have to put the put some regular definitions; like the letter is defined as the a, b, c this characters till z; then, capital A, capital B, capital C till capital Z that is letter. Then, you can say digit is 0, 1, 2 going up to 9 ok; 0 to 9. After that you can define identifier; you can define identifier.

So, it can start with the underscore character or a letter. And after that the next character, next all of them can come underscore, letter or digit and they can come any number of time. So, there should be a star on that. So, this defines the identifier for the programming language. Some programming language is they have got a limitation on the number of characters that you can have. For example, some of them has got restrictions say.

So, it is very easy if the size is not restricted; if the identifier size is not restricted. But if I tell you that the in some language; so the instead of star, so this has to be 5 only. So, that is totally 6 characters. So, first character is underscore or letter and next 5 characters can be underscore letter or digit. But total is only 6 characters. Then of course, it is very difficult because you have to write it explicitly. So, you have to write like first character, their first underscore or letter and then, you have to write explicitly underscore, letter, digit; then again, underscore, letter, digit. So, you have to go on doing like this you have to go on doing like this for 5 times.

So that way, you see that it is a blazing that this language is very flexible in terms of the length of the identifier and it helps the compiler designer, rather than putting restriction on the compiler. Apparently it seems that I have to support a very large number of characters in the identifier, but it is not really so. And of course, with the help of the tool lex if we do, then it is the minimum amount effort that we have to put. So, next we will be writing another regular expression. So, that will be binary strings such as a 0 is always followed by 1.

(Refer Slide Time: 05:11)



So, Binary string in which 0 is always followed by 1. So, for writing this, so one possibility is that there is no 0 ok so it is and even the strings are of 0 length that is null string. So, they are also valid because they do contain neither 0 nor 1. So, one possibility is that I have got only 1's. So, any number of 1's can be there.

But if there is a 0, if there is a 0; then, it has to be followed by a 1 and after that I can have any number of 1's not only a single 1. So, whenever there is a 0 it is followed by so, at least a single 1 you can say. So, 0 1 1's at least 1 1's 1 symbol is there and then any number of them and then, this can go on any number of times and at the end you can have any number of 1's.

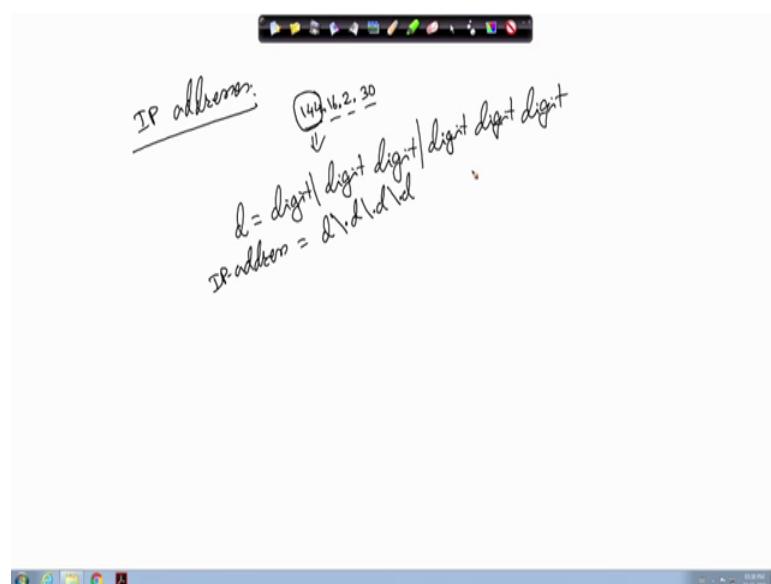
So, this regular expression this will be capturing all those say for example, if I have got to say 1 1 0 1 0 1 1 0 1 1 something like this, then you see that it is a first two 1's will be matching with this 1 start then this 0 will be match with this 0. Then, this 1 will match with this 1. Then there is 0. So, this 1 star will be taken as 0 and then, for matching this 0. So, we will take another occurrence of this particular sub expression. So, this 0 will be matching with this 0, then these two 1's will match with these two 1's; then again, there is a 0.

So, another round of this symbol will be another this regular expression will be taken sorry. So, that will match with 0, 1 and then 1 star. So, 1 star will match with this. So, after that we have got 1 star and all. So, that way can take this 1 ok. So in fact, whether

this 1 star is necessary or not? So, that is also a question like if there are 1's at the end. So, that is taken care of by this part ok. So, I think. So, this was a this 1 star can be neglected also.

So, you can write it as 0 1 1 start whole star. So, that can also be written. Of course, putting the 1 star at the end does not make it more number of strings to be accepted. So, that way there is no problem, but that is redundant. Next we will be looking into another regular expression for IP addresses.

(Refer Slide Time: 08:23)



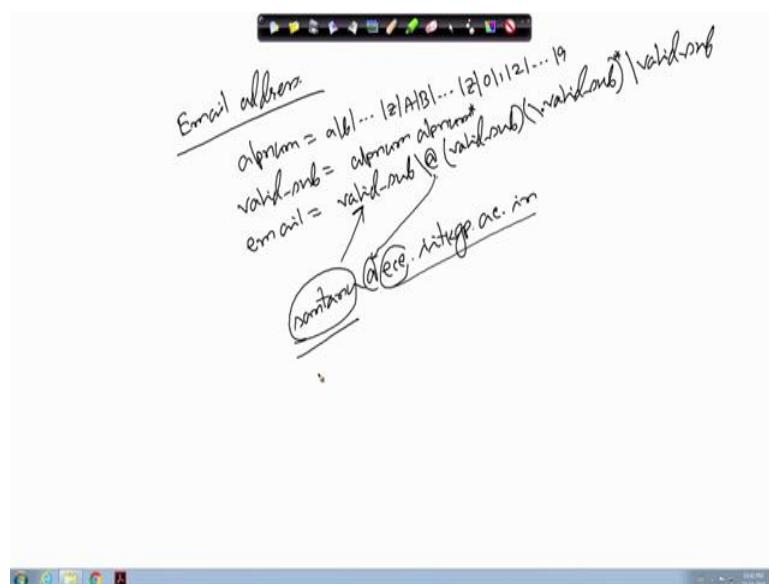
So, an IP address is so, in any IP address, so it has got 4 parts like if an IP address is say 144 16 2 30. Then, what happens is that so individually they are some numbers and there are some dots in between ok. So, there should be 4 such portions, 4 such parts and each part will be separated by a dot and each of them is a number.

So, what you can do? So, and this numbers; so there is a limit because this is each of them is only 8 bit. So, you cannot have more than 3 digits in it ok. So, you cannot have more than 3 digits in it. So, we can say my d is equal to digit or digit 2 digits basically or 3 digits; digit digit digit and once it is done, then I can write that IP address is equal to d and then, this dot character has to be there and following the lex convention see we put this escape character d, then d, then d.

So, this is the regular expression for IP address. Of course, I did not check whether this values that are coming are valid or not; that is not the purpose of lexical analysis tool that will be done by parser or some high level program which will be having the values. And at what the lexical analysis tool can do? It can identify the corresponding number the matching number and it can return that part. So, that as an attribute and then the program high level program can check; whether those values are correct or not.

So, that way we can have some regular expression for this IP addresses. Next, we will be looking into say another example say email address. So, email address.

(Refer Slide Time: 10:59)



So, for email address so, we have got alphabetic character and numbers. So, we say that we define alpha num as this lowercase characters a, b going up to z and the uppercase characters A, B going up to capital Z and then the digit 0, 1, 2 going up to 9 ok.

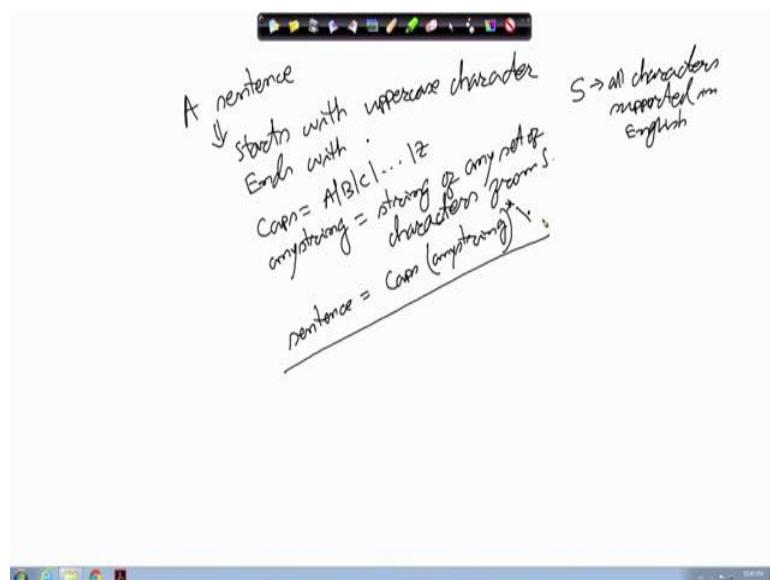
Then, we can say that a valid part valid sub part valid sub is equal to alpha num 1 of them followed by alpha num star. So, 1 or more occurrences of these alphabetic alpha alpha alphanumeric characters and then, the email can be defined as valid sub; then, at the rate symbol we have put purposefully this escape character to make it clear.

Then, I can have valid sub, then after this we can have valid sub. Basically, what I want mean is that suppose my email id is a santanu@ece.iitkgp.ac.in. So first before this at the rate character, so this part will be matching with this valid sub.

Now, this at the rate part matches with this at the rate part and after at the rate I can have any number of such sub parts. So, in this particular case, we have got 4 sub parts. So, you we have any number of sub parts. So, we can have this thing, this star. Sometimes, we are in email id, so we may not be mentioning this all this symbols. So, we can be it can be that it is just 1 1 valid sub part after that. So, we can put a or with this with valid sub.

Telling that there is no at the rate I just write santhanu; I do not write anything else. So, that also valid because by default it will attach some other parts to it. So, that will also become a valid part. So, this way we can we can write the regular expression for email address. Next we will be looking into another example. Say we are trying to design a system that can check whether it is a sentence or not.

(Refer Slide Time: 14:43)



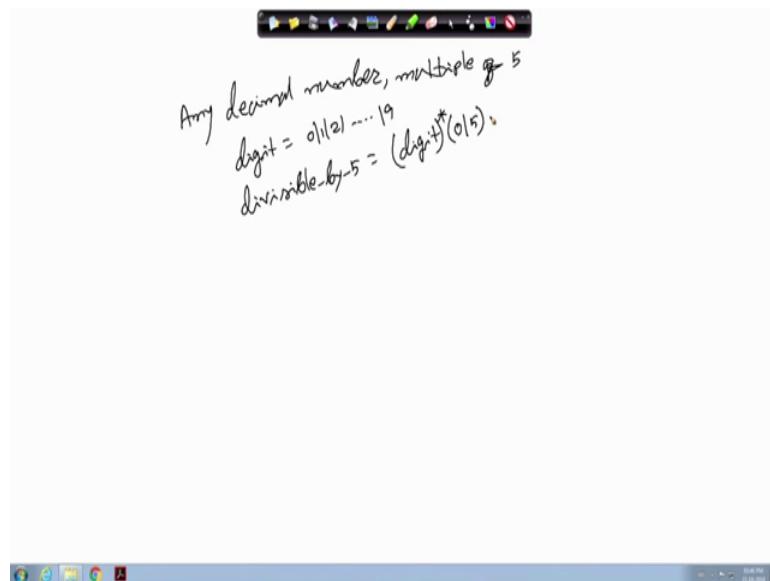
So, a sentence so, in English language a sentence means it starts with the uppercase character starts with uppercase character and it ends with a full stop. So, these are the bare minimum things that we need to have that we need to have in a English language statement. So you can.

So, you can define like caps equal to all the upper case symbol A B C going till Z and then, any string you can define another definition any string; of course, I it is very difficult to tell what are the possible strings in the English language. So, it is so whatever the set of characters that English language supports. So, if I start writing it will be enormous.

So, I do not do that I assume that there is a set S which contains all characters supported by all characters supported in English language supported in English ok. Then, in that case the any string is string of any symbols, any set of characters, any set of characters from S; then, we can write that sentence is equal to caps followed by this any string and that can come any number of times. So, any number of strings, any number of words can come and then it should end with a full stop ok.

So, that way it can. So, this may be the regular expression for sentence. This is not very rigorous because of this definition of any string is not very clear, but anyway so we can proceed in that direction. Next, we will be taking another example, where it is a decimal number multiple of 5.

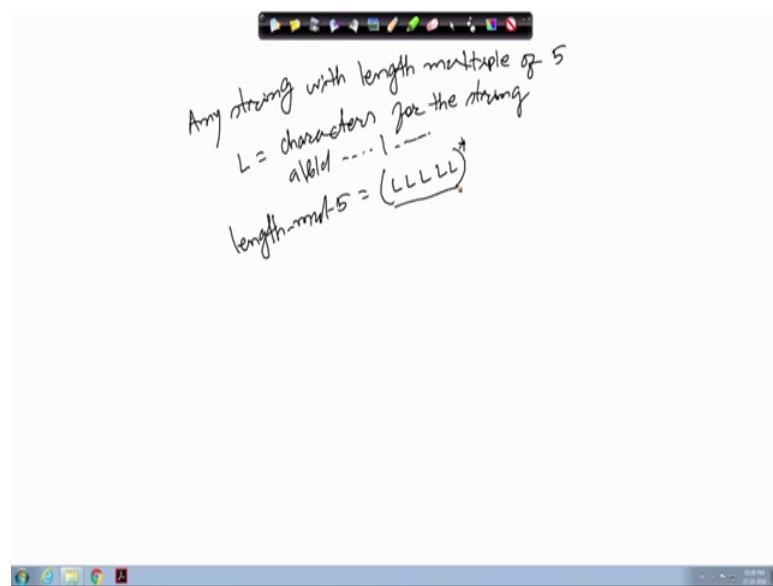
(Refer Slide Time: 17:27)



Any decimal number multiple of 5; so, if I need to do that, then the number decimal numbers which are multiples of 5, they should end with a 0 or a 5 ok. So, I can say I can first define the digit as I did previously for numbers 0 or 1 or 2 or 9.

And then, I can I can define the regular expression divisible by 5. I can define this regular definition divisible by 5 to be digit star and then, it should end with a 0 or 5. So, any number of digits can appear, but after that the last digit should be 0 or 5. So, in that case the number resulting number is definitely divisible by 5. So, we can have a regular definition like that.

(Refer Slide Time: 18:53)

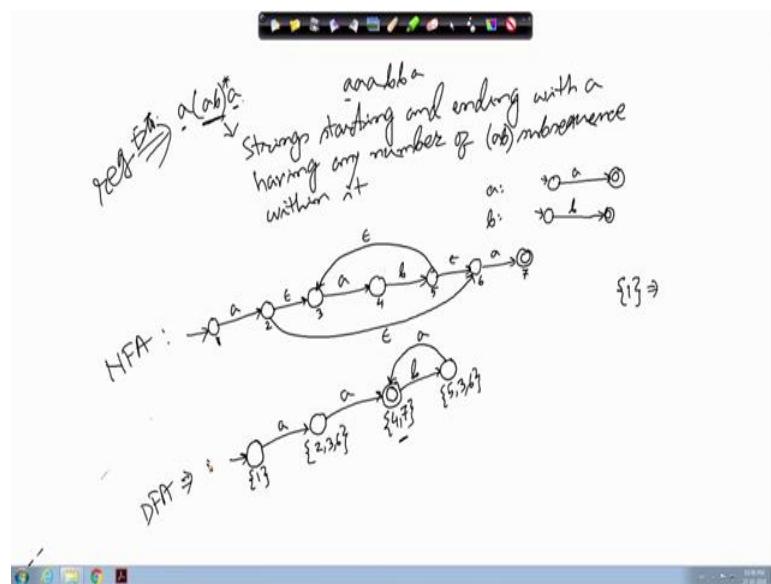


So, then any number who any then we will write another one any string, any string whose length is multiple of 5 whose length is multiple of 5; so if you want to do that then we can set the first I define L to be the set of characters that can come.

So, I can define all characters for on which we are going to form the string. So, it is the characters for the string. So, you can you can write it like say a or b or c etcetera. So, all the characters that are allowed for the language so that you can put in L and then, it says that length should be multiple of 5. So, that definition length multiple of 5 so, it can be L L L L and L and then, whole star. So, that means, so 0 occurrences. So, 0 is also multiple of 5.

So, that way it is a 0 length string. So, but I cannot have length to be equal to 1, 2, 3, 4 because then this part has to much and there are 5 such Ls. So, I will have length must be equal to 5 ok. So, this way we can define regular expressions for different patterns. Next, we will be looking into some regular expression and then, try to understand what is the corresponding pattern.

(Refer Slide Time: 20:45)



Say I have got this one  $a \ ab^* a$ . So, what is this regular expression? What does it mean? So, you see that all the strings, they start with  $a$  and end with  $a$  and in between I can have  $a$  is this  $n$  number of  $ab$ 's ok. So, if I have got a string like  $a a a b b$  ok. So, that say  $a$ . So, is it a valid string? So, this  $a$  the first  $a$  will match, then this  $ab$  has to match; but it will not match with a  $b$ . So, that is a problem.

So, it will be the string that is that it will recognize that strings starting and ending with  $a$  having any number of  $a b$  subsequence within it ok. So, this is the regular expression. Now, for this if we try to construct the NFA ok, then for  $a$  we know for  $a$  I have got the regular expression the NFA like this; for  $b$  I have got this one. This is for  $b$ . Now, I have to do it for  $a b$  ok.

For doing for  $a b$ , I can do it like this that this is the state from where it goes on  $a$  to this is the final state and from here on  $b$  I can go to the final state there. Now after that I have got  $a b^*$  and for doing  $a b^*$ , the construction is that I have to add 1 epsilon transition from here. Similarly, from here I have to add an epsilon transition to this ok, then I have to have another epsilon transition taking to the final state and then, from this state there should be an epsilon transition there.

So, this is the construction of star that we have done previously and then so, this construct the  $a b^*$ . So, this portion and before that I have got this  $a$  and ending with another  $a$ . So, I will be having transition like this  $a$  and this is the start state and from

here, I will have another going to this as the final state fine. So, this way I can have the regular expression converted into NFA. Now, this NFA.

So, if I number the states this is state number 1 2 3 4 5 6 7; then, I can construct the corresponding DFA. I can construct the corresponding DFA like this. So, I can take. So, initially I have to see that from state 1 what are; so, on state from state 1 on this on this state 1, from the state 1 how can I go to other states so on epsilon transition what are the other states on which to which I can go.

And here you see that on epsilon transition you cannot go anywhere. So, your state number 1 is the start state of the DFA. So, this is the start state of the DFA. Now from state from this state if you go on a, you will be reaching the state 2 and from state 2 on epsilon transition, you can go to the state 3 and 6. So, this from this one, I can come to a state which is 2, 3, 6 on a and from this 2, 3, 6 states, so if you go on a, say from 2 on a; you cannot go anywhere from 3 you can come to 4.

And from 6, you can come to 7. So, from this I can say that on a. So, on a I will be able to go to a state which is 4 and 7. From 3, you can come to 4 and from 6, you can come to 7 and from 4 and 7, there are no epsilon transitions defined. So, this is one of the final. This is this is the state where you can go. Similarly, from 2, 3, 6 if you try to go on b; so there is no transition define from 2, 3, 6 on b. So, on b there is no transition. Now from state 4 and 7, there are no transitions defined on a ok, but on b from 4 you can come to 5 so on.

So this from 4, you can come to 5. So the state is 5. Similarly, from 7, you cannot go anywhere on b. It is not defined. Now from 5 on epsilon transition you can go to 3 and 6. So, this 5, 3, 6 so, this 5, 3, 6 and here since 7 was a final state of the NFA. So, this also happens to be a final state of the DFA. Now this 5 3 6 from the 5 3 6, if you try so this is on b and from 5 3 6 if you get a ok. So, from 5 from 5 on a there is no transition; from 3 on a it can come to 4 and from 6 on a it can come to 7. So that gives us this state only.

So you see that this is a transition which is on a. So, this way we can construct the DFA ok. So this is the corresponding DFA. So this is the DFA. So, this is the regular expression. This is the regular expression. This is the corresponding NFA and you can have the corresponding DFA made here. So this way we can have these regular

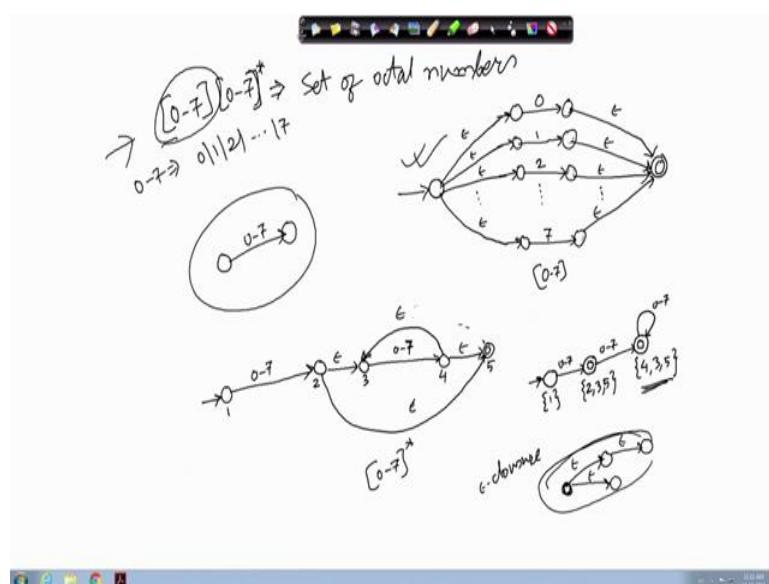
expressions for several strings and then, we can determine the corresponding NFA and DFA.

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

**Lecture – 15**  
**Lexical Analysis (Contd.)**

So, next we will be looking into some more examples of this construction of NFA and DFA from regular expressions.

(Refer Slide Time: 00:29)



So, the next example that we will look into is a regular expression, which is given by 0 to 7 followed by 0 to 7 star. So, basically we have got the digits 0 to 7 followed by 0 or more occurrences of the digit 0 to 7.

So, what is this regular expression? So this is the regular expression that corresponds to the set of octal numbers, set of octal numbers, because the numbers can start with any of the digits 0 to 7. And there can be one digit or multiple digit, but the digits are restricted to 7 only, so it cannot be more than that. So, this represents the octal numbers.

So, if you want to construct the corresponding NFA for this one, so for each of the digit we will have NFA like this. So, this is for the 0 ok, so then this is for 1 this is for 1, then we for 2 we will have another such NFA. So, this way we can have up to 7, we can have the NFA's created. Now, for 0 to 7, which actually means that regular expression 0 or 1

or 2 or up to 7, so for that we have to join all of them. So, we have to have one initial state, and one final state. And from this initial state, I have to add epsilon transitions to each of these states ok. So, I have to add epsilon transitions. So, these are all epsilon transitions. And this is the final initial state.

And so these states that we have, so they will no more be final state ok. So, they will be normal state now. And then we will have this, we will have from here we have to add transitions to the final state ok. So, these are again epsilon transitions, and this is a final state.

So, this way we have the NFA corresponding to 0 to 7. Now, for our sake of writing, so for our gravity; so what we will do, we will simply represent it by something like this, we will write here 0 to 7. Essentially, we mean that this whole stuff is actually this one.

Now, from this if we once we have done this, then you can draw the regular expression for 0 to 7 star. So, this part is for 0 to 7, if I take it like that so it is 0 to 7, now for making 0 to 7 star. So, we have to have some epsilon transitions added ok. So, we will have to have some epsilon transitions added.

So, there we have to introduce one initial state from here, the epsilon transition will come. Similarly, from this one I will have one epsilon transition to the final state ok. And from this state, there should be an epsilon transition back to the there will be an epsilon transition coming back to their; sorry there is a problem here, so this is not correct so ok. So, from this state if I number this state, numbers are 0, 1, 2, 3 etcetera, then from state number 1 I have got a transition on that digits 0 to 7. Now, on epsilon it should come back to the states, so that will capture that 0 to 7 star.

And then I have to have epsilon transition here, and from this first state to the final state there should be an epsilon transition, so that way it will capture 0 to 7 star. So, this part will be 0 to 7 star. Now, before that we have got this 0 to 7. So, for that I will have to add that 0 to 7 portion, so this will be like this 0 to 7 ok. So, then if we number the states, then this is the final this remains the final state ok, this remains the final state.

Now, if we want to construct the corresponding DFA, then we have to convert this NFA to DFA by clubbing the states as we did previously. So, let us renumber the states

for this clubbing purpose. So, let us say that the states are numbered as so this is say numbered as 1, this state is 2, this is 3, this is 4, and this is 5 ok.

Now, for the original NFA, this is the start state of. So, when we are trying to construct the DFA, we have to start with state number 1 and take epsilon transitions from state number 1 to have the initial state of the DFA. Since there is no epsilon transition from state number 1. So, in the DFA the initial state is consisting of a single state of NFA, which is 1. Then from 1 on 0 to 7 you can go to state-2, and from state-2 on epsilon transition, we can go to state 5. So, this way it on 0 to 7, it goes to the state, which is consisting of the NFA states 2, 3, and 5; 2, 3, and 5.

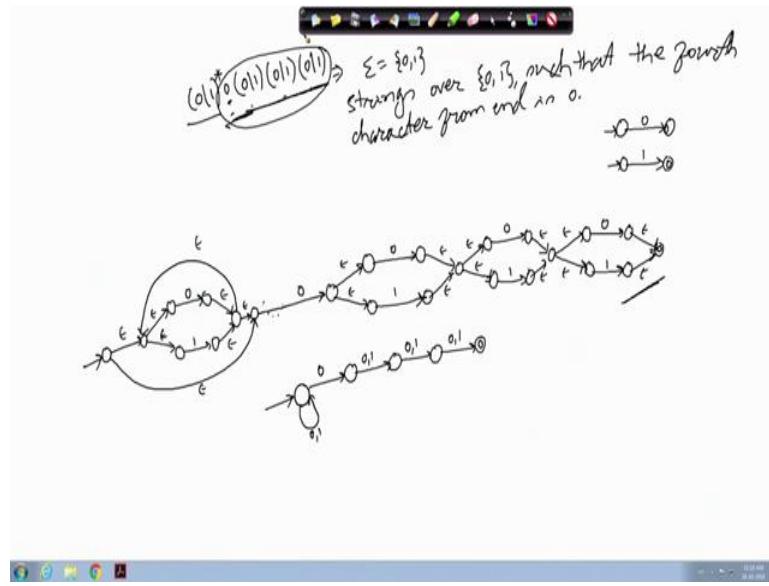
Now, from 3 on epsilon transition on 0 to 7, you go to state number-4. And from state number-4 on epsilon transition, it can go to state-3 and state-5. So, from here on 0 to 7, it can go to a state, which is actually consisting of the NFA states 4 from 2, 3, 5. So, from 3 this is it is going to state 4. And from 4 on epsilon transition, it is going to 3 and 5, so 4, 3, and 5. And you see that here this here this state number-5 is a final state for the NFA. So, they also constitute the final states for the DFA ok.

Now, from state number 3 from state number-3 on getting 0 to 7; so like if I if I consider this particular DFA state, then from state number-4, there is no transition on 0 to 7, for 5 also there is no transition, but for 3 there is a transition to state number-4.

And after coming to 4, so if you take the states reachable, where epsilon transition with which is which is also known as epsilon closer, so this is the technical term. So, from one states, so all the states that are reachable by epsilon transitions only. So, if this is a state, then from this state, where epsilon transition I can reach all the states. So, all the states they will come in the epsilon closure of this state.

So, basically we are trying to compute that epsilon closure. So, from here that epsilon closure will give me the same state, because from 3, it will come to 4. And from 4 if I take the epsilon closure, it will get 3 and 5. So, as a result you will get a loop here for 0 to 7. So, this is the DFA corresponding to the octal numbers in a regular expression that we had ok.

(Refer Slide Time: 09:32)



Next, we will take another example. Say the regular expression, which is given by 0 or 1 star 0, and then followed by 0 or 1, 0 or 1, 0 or 1 fine. So, if we what is this regular expression. So if we look at it a carefully, then here the alphabet set is 0, 1. So, it is actually strings over the symbols 0 and 1.

And then we are if you look into it carefully, then the 4th (Refer Time: 10:06) character is always 0, all the strings where the 4th class character is 0. So, they are the valid strings of this language. So, there is strings over strings over the alphabet set 0, 1 such that the 4th character from end 4th character from end is 0. So, this is the meaning of the regular expression.

So, you can try to construct the NFA for this one. And the NFA for this can be constructed like this. First of all for 0 and 1, so we know that this is the regular this is the NFA. This is 0 and this is 1. Now, for 0 or 1, so we can do it like this, so if this is on 0 and this is on 1. So, we have to add one state before this, so that there are epsilon transitions. And then one state after this; and these are also epsilon transitions. So, this is the part of the NFA corresponding to 0 or 1.

Now, you see that there are 3 such 0 and 1's. So, you can say that as if I have to make it I have to make 3 such structures, and put them one after the other. So, let us put 3 such structures. So, this is on 0 going here, these are epsilon transitions. Then again from here

on epsilon it comes to this state, and then this is the final state. So, this constitutes this part of the regular expression.

Now, we have got before this 0, so for this 0 you see that we should have a transition like this fine. And before that we have got this 0 or 1 star. So, for 0 or 1 is represented like this ok, so this is the 0 or 1. Now, for making a star, what I need to do is that I have to have one more I have to have one more state initial state here. And add one epsilon transition.

And similarly, have another final state here, which will be an epsilon transition. And then add epsilon transition from here to here, and add epsilon transition from here to here. So, this is going to be the case, which is 0 or 1 star. Now, I have to concatenate it with the remaining part of the regular expression. So, I can concatenate it like this, and that is also an epsilon transition.

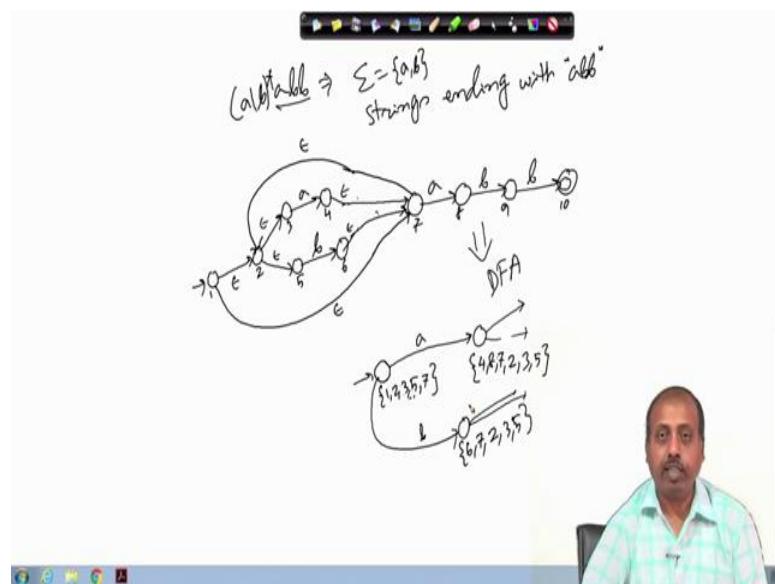
So, you can do it like this or for optimization purpose, since there are two epsilon transitions. So, you may say that I can have only one of them. And you can go this epsilon this state all together, and you can say that I will make it from here to there by means of 0, so that can be done. Just to simplify the diagram a bit ok. And this is the initial state for the overall NFA.

So, this is one possibility and you can, so that is by the construction the construction rule that we have. So, we can do it like this ok. So, on the other hand, so we can also do it in a different fashion like you can draw an NFA directly, like you can make it like this that as long as it is 0 or 1, it remains in this state. And then when it gets a 0, it makes a non-deterministic move to the next state.

And then you have got this 0 or 1, there are three such states three such transitions 0 or 1 ok. And this is the initial state this is the final state. So, this way also you can draw the NFA. So, this is also NFA, because it has got non-deterministic decision at the first state. So, either it is on getting input 0, either it is remaining in this state or it is going to the next state, so that way we can have a non-deterministic choice. So, this is also an NFA, but you see the NFA that we have constructed here is very complex, but it is but the process is automatic. So, we have followed a rule book for constructing the overall NFA ok.

So, now you can try to convert it into DFA, but convert into DFA we will take good amount of effort because of this reason that at the end you have got something like this, this 4th character from end is 0. So, in our class we have discussed previously that whenever you have got this type of situation, then there is a possibility of state explosion. So, this is a typical example, where the state explosion may occur ok.

(Refer Slide Time: 16:47)



Next we will look into another regular expression, we will be looking into another regular expression say something like this  $a$  or  $b$  star, then  $a$  then  $b$   $b$ . So this is a regular expression. So, when the language corresponding to this is that any string, so here this alphabet set is  $a, b$ . And this realizes it specifies the words, which are strings ending with  $a b b$ , all strings ending with the substring  $a b b$ , so that is going to be valid strings for the language.

Now, here also if you try to construct the NFA, then for this  $a b b$  this portion, the NFA will be something like this on  $a$  it comes to the state, on  $b$  it comes to the state, then again on  $b$  it goes to this state. And before that we will have to have  $a$  or  $b$  star for  $a$  or  $b$  star the NFA is like this, so this is  $a b$ . Now, all of them so for that I have to have epsilon transitions add it. So, I have added a epsilon transitions, so this is  $a$  or  $b$ .

Now, I have to make  $a$  or  $b$  star. So, for making  $a$  or  $b$  star, so I have to take another final state. And from here, I have to add epsilon transition. And I have to have one initial state from where there will be an epsilon transition. Now, from this state, it should come back

here that is an epsilon transition and from here it should go there, which is an epsilon transition.

Now, I have to merge these two. So, essentially what happens is that these two states becomes same ok. So, I can just redraw this part, so that this transition. So, this state becomes the final state ok. So, it can come here. And so all these transitions can be merged now sorry so all these epsilon transitions they can be modified, so that they can point to this state epsilon transitions. And from here on epsilon, it can go back there. So, this will also realize this regular expression  $a \cup b^*ab$ . So, you can always convert this regular expression into this NFA into DFA.

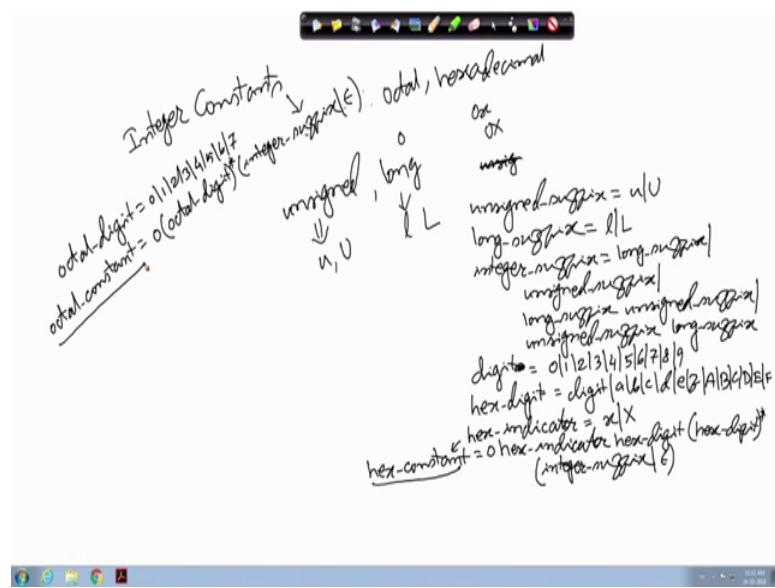
So, you can convert into DFA by following the technique that you can start with say symbol was state number 1, 2, 3, 4, 5, 6, 7, 8, 9, 10; you can number them like this. Then from state number-1 on epsilon transition, it can go to state-2 and state-7. So, you can construct the initial state of the DFA consisting of the states 1, 2, and 7 of the NFA.

Then from this states on 1, 2 sorry 1, 2, 3, 5 will also come. It is not only 1, 2, 7, 1, 2, then 3, 5 and 7 they will come, so that will be the initial state. Now, from here on getting a you can go from 3, you can go to 4 on getting a and from 7, you can go to 8. So, I can say that on a I can go to a state, which is the state 4, 8 and their epsilon closure, so 4, 8 will be there.

Now, from state 4 on epsilon, you can go to 7. And from 7, you can come to 2. From 2 you can go to 3, and you can go to 5. So, this way you can go on constructing the further states on different symbols a and b. Similarly, from state from this state on getting b, you can come to a state on getting b. So, it will come to 6, because from state 5 was there. So, from state 5, 1, b, it will go to state 6. And from state 6 on epsilon transition you can go to 7, then 2 and then from 2, you can go to 3, 5. So, this is the state on this is the transition on state on symbol b.

So, now you can explode further from here what happens on a, what happens on b, so that way you can just go on finding the further transitions, and that will complete this DFA ok. So, you can construct the DFA like that ok. So, this way for from regular is once you have constructed the regular expression from the English language statement of the problem. So, you can make the corresponding NFA, and then convert the NFA to DFA.

(Refer Slide Time: 23:05)



So, once the DFA has been made, you can go for realizing the corresponding lexical analyzer using the tool Lex ok. So, we will take one more example from the lexical analysis portion, so that will be for Lex specification. So, we will consider the constants that you can declare in say C plus plus language; so constant integer constant integer constants.

So integer constants, so they can be decimal can be decimal, can be octal or can be hexadecimal ok. So, these are the similarly so you can have; so this octal is identified by the numbers starting with a 0, and hexadecimal is identified by the number starting with 0 x or 0 x like that.

Similarly, towards the end of the number, you can have some more qualifiers like you can have unsigned, and you can have long. So, these are the other two qualifiers that we have for the integers and unsigned is identified by u or capital U, and long is identified by a l or capital L ok. So, we need to define integer constant some regular expression that will represent integer constants in the C plus plus language.

So, lets us see how we can do this thing. So, we will write like this that first of all we will write, we will write some sub parts of this definition, which is the first one is unsigned suffix unsigned suffix can be equal to u or lowercase u or uppercase U. Then long suffix can be lowercase l or uppercase L.

Now, integer suffix integer suffix so you can have them in any order. So, it may be that it is a long suffix that is the number is ending with a with the symbol l or capital L small l or capital 1 long suffix or unsigned suffix unsigned suffix or maybe long-long suffix followed by unsigned suffix long suffix followed by unsigned suffix or it may be the other way that is unsigned suffix followed by long suffix. So that is the other way unsigned suffix followed by long suffix. So, these are the way by which the suffix can come after the integer.

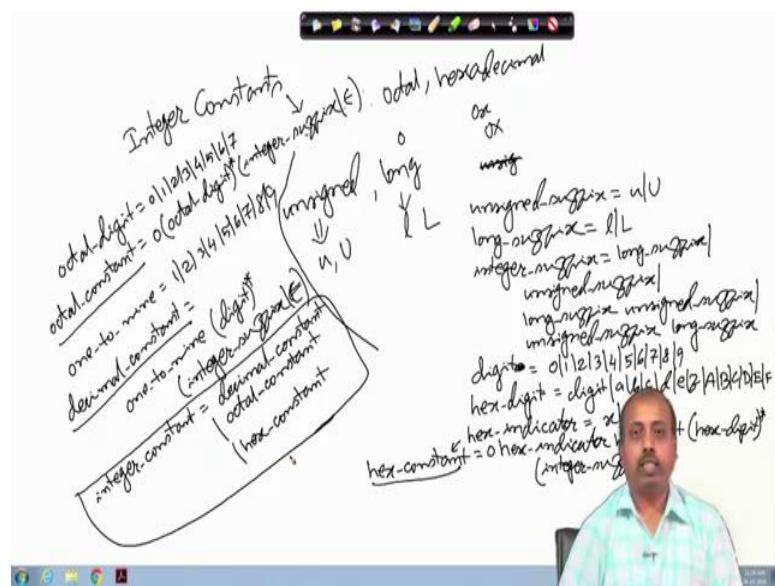
Then I will define digits. So, digits are a single digit is 0 or 1 or 2 or 3 4, 5, 6, 7, 8, 9. So, these are the digits. Now, I will have hexadecimal digits or something more than that. So, hex digit is equal to digit, and there are some more options like a, b, c, d, e, f also the capital versions capital A, capital B, capital C, capital D, capital E and capital ok. So, this is about the hexadecimal digit.

Now, we can have the indicator the hex indicator hex indicator is small x or capital X ok. Now, for the octal numbers, I can have octal digit octal digit. So, it does not include all the digits. So, it is 0 or 1 or 2 or 3 or up to 7, 4, 5, 6, 7. So, I can have eight different digits for that.

And octal constant, now after we have defined this octal digits. So, we can define octal constant octal constant, it can be 0 ok. So, octal constant should start with is 0, then this octal digit any number of octal digits can come now octal digit star. And then I can have an integer suffix after this. I can have some integer suffix integer suffix or epsilon. So, integer suffix may or may not be present, so that way we can have this thing.

Similarly, the hexadecimal constant the hex constants like here. So after this I can define hex constant hex constant equal to 0 followed by hex indicator 0 followed by hex indicator, then hex digit at least one digit followed by hex digit star. And that should be followed by integer suffix or epsilon integer suffix or epsilon. So we have defined hex constants, we have to define octal constant.

(Refer Slide Time: 30:38)



Now, I can define the digit the decimal constant, but for decimal constant so we will not take anything that starts with a 0 ok so 0 it will go to octal. So, to differentiate between them, so we define another the regular definition 1 to 9, which is given by 1 or 2 or 3 or 4 or 5 or 6 or 7 or 8 or 9. And now decimal constant decimal constant is defined as this 1 to 9 1 to 9 followed by digit star. And after that I can have integer suffix integer suffix or epsilon integer suffix or epsilon. So, this way I can have decimal constant. So, decimal constant have been defined also, all the constants are been defined.

Now the overall integer constant is nothing but or of all them. So integer constant can be defined as decimal constant or octal constant or hex constant ok. So this is the final definition of integer constant. So it takes care of all the issues like decimal constant or octal constant, hexadecimal constant ok so all of them. And then it also takes care of these suffixes a long unsigned, etcetera ok.

So, this way we can define regular expression for different programming language constructs and accordingly, you can design the corresponding lexical analyzer.

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

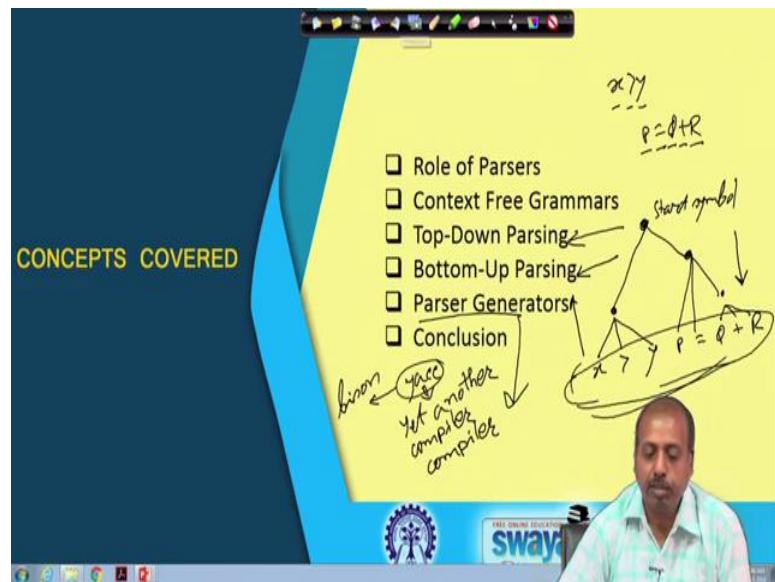
**Lecture – 16**  
**Parser**

The next phase in the compiler generation process or the Compiler Design process is designing a tool for syntax analysis purpose. So, we have seen so far the lexical analyzer. So, that can return the tokens that are appearing in the language so, valid words of the language. Now, these words are to be connected or there are they need to be sequenced in a proper fashion for some meaningful constructs of the language. So, whatever be the language starting with English to some programming language.

So, there is some sequence in which this token has to appear. So, lexical analyzer so, it cannot determine that sequence. So, it can just give you the tokens that are coming or the words that are coming in the input source file. And, the next job is to try to see whether these words are appearing in a proper sequence or not. So, if they are appearing in a proper sequence then they are following the grammar rules of the language. And, accordingly the entire program is accepted as a valid program, entire input file is accepted as a valid input for the language that we are considering.

So, this phase so, we will be discussing on this syntax analyzer design or the task of syntax analysis and it is very complex. In the sense that depending upon the programming language constructs so, which are quite varied in nature? So, it is often very difficult to come up with some syntax analysis tool. And, depending upon the constructs that we have sometimes we can have a very simple type of syntax analyzer whereas, if the language is quite complex, the grammar is quite complex in that case the construction also becomes difficult and many times we need to do a trade off. Like some so, we have to use the knowledge or the intuition of the compiler designer to resolve between different issues that are appearing in the reduction process.

(Refer Slide Time: 02:23)



So our discussion will flow in this way, first we will discuss about role of parsers. So, parser is the technical term for this syntax analyzer. So, it parses the input file into the grammar of the language. So, if it is successful; if it is successful in the sense that if the grammar is the if the input file is grammatically correct.

So, it follows a certain pattern by which this entire program can be derived starting with the start symbol of the grammar. So, that is the role of the parser. So, parser will try to give a proof that this particular input file follows the grammar of the language. And if it is successful in proving it so, it will give it will return a parse tree which is constructed by being in terms of the terminals and non-terminal symbols that are used for specifying the grammar. And, then after that based on that parse tree the compiler designer can try to do so, take some actions.

So, that it can generate code or it can do some other transformation to the input pro input file, so on. So, that is the basic rule of parser. So, as I think now you can follow that it is it has to check all the grammar constructs of the language. So, as a result it is going to be quite involved, then there are different types of grammars. So, we will be discussing mostly on the context free grammars because, most of the programming language construct so, they belong to this category of; this category of grammars ok.

So, most of the programming language they follow their constructs can be specified by means of context free grammars. So, we will be mostly discussing on context free

grammar. After we have a after introducing these grammars and all we will be looking into is 2 class of parsing technique: the 1st class is known as top down parsing and the 2nd class is known as bottom up parsing. So, it is like this that suppose I need to; suppose I need to develop some, suppose I am given an input line say  $x$  greater than  $y$  then say if as a  $P$  equal to say  $Q$  plus  $R$  something like this is given.

And there are some rules in the language, where there if there are some rules in the grammar that are used for specifying the language. Now, one possibility is that you start with this basic symbols like  $P$  equality  $Q$  plus  $R$ ,  $x$  greater than  $y$  etcetera. So, you start with those symbols  $x$  greater than  $y$ , then  $P$  then equality symbol  $Q$  plus symbol  $R$  and all that. So, you start with that and then try to combine them in some fashion to such that in a tree type of structure; maybe you try to combine say these 3 together into some symbol. Similarly, maybe you try to combine so, these 3 together into some symbol. Then you try to combine say these 3 into some symbol then finally, you try to combine these 2 into some symbol where this final one is the start symbol of the grammar.

So, this is the start symbol of the grammar. So, what is happening is that ultimately we are producing a tree and the way I have described it; I have started with the at the lowest level the input file itself and from there I am trying to reduce that input file to the start symbol of the grammar. So, this type of approach so, they are known as bottom up parsing because, I am starting with the symbols or the words of the language and then trying to combine them together towards the start symbol of the grammar. Just the reverse approach that is given a program, we can start with the start symbol and then try to see like what may be the possible set of rules.

So, that ultimately it boils down to this particular program. So, it is actually looking in this direction. So, previously we were going in this direction and now we are going in the opposite direction, starting with the start symbol we are trying to go down and reach the final program. So, that particular approach will be known as top down approach. So, in general top down approach they are simpler than bottom up approach. However, there are restrictions also like we will see that for certain languages; so, it may be difficult to construct top down parser because, of the nature of the grammar whereas, it may be possible to construct the bottom up parsers for them.

And in general so, we will see that the bottom of parsers often they make a superset of this top down parsers. So, for whatever any language if you can construct a top down parser you can also construct a bottom up parser, but the reverse is not true. So, this way we will be looking into both top down and bottom up parsing strategies because, bottom up parsing is costly ok. It is so, constructing the parser is difficult. So, it is costly and it takes more time for constructing the parser. So, many simple grammars; so will be we can very easily construct the top down parsing technique.

And then we can they have a parser for that. So, we will look into some examples from both the categories, then we will be looking into some parser generators. So, as we go through the chapter then we will see that the there are algorithms and techniques so, which are fine. So, that they are all automated that is so, once you understand that theory. So, it is possible for us to write down the corresponding program which will act as the parser, but due to the sheer volume of the programming language grammars that we have in today.

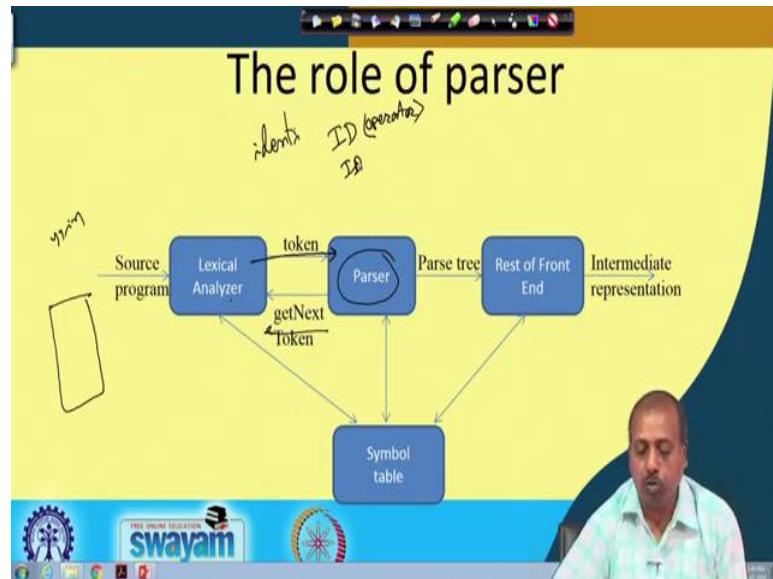
So, it is very difficult to construct those parsers by hand. So, we will be looking into some tool which will be used for this which will be acting as this parser generator. Just like for lexical analysis tool so, we have a lexical analysis job so, we had the 2 legs. So, here also for parser generator so, we have got tools called yacc. Then so, the yacc is yet another compiler generator, compiler compiler Yet Another Compiler Compiler.

So, what does it mean? Why compiler is coming twice, is because there because of the fact that it is a compiler for a compiler. So, it has you specify a grammar and for that it will generate a compiler. So, it is based that is why it is a compiler compiler. So, it compiles a compiler already makes a compiler. So, that is why it is like this is yet another compiler compiler and the abbreviation is yacc. And, just because the abbreviation is yacc some other versions of the tool that has come up one way one of them is known as bison and there are many others I believe.

So, the bison does not have any full form. So, this is basically taking yacc as an animal. So, it is taking bison as another tool, but whatever it is so, basic principle remains same. So, they are all parser generators and they can be used for automatically generating parser from the specification. So, our job is to see that the specification is correct and we

understand how this parser generated so, work actually. So, once we do that so, we will be able to write our own parsers. So, throughout the chapter we will see this thing.

(Refer Slide Time: 10:15)



So, let us start with the role of a parser. The role of a parser is like this. So, the parser actually is sitting somewhere here. So, this is talking to the lexical analyzer tool for tokens. So, it whenever it needs a token to proceed so, it will ask the lexical analyzer get next token. So, to provide the next token to it and as we understand as we have seen in the lexical analysis tool so, lexical analysis tool so, it actually scans the source file. So, if this is the source file so, it has got the pointer y y in and that pointer from that pointer it will try to see what is the maximally matched token for this for the next inputs part.

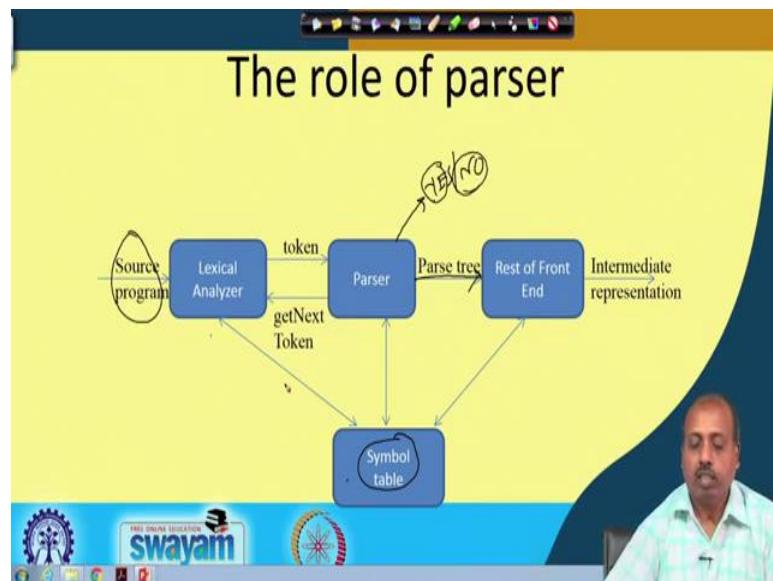
So, accordingly it will find out the token and return it to the parser. So, parser getting this token will try to see like what may be the corresponding action. So, it may try to follow a certain grammar rule for deriving the statement or deriving the line of text like that or it may be that it finds that the token that it has got is not meaningful in this at this point of time. So, that in that case so, that is an error. So, that is a grammatical error. So, typical examples may be that suppose it has seen an identifier ok, suppose it has seen an identifier as a token.

So, ID is the token returned by lexical analyzer. Now, if I consider a language that consists of only arithmetic expressions ok. Then after ID so, it is expected that I will get an operator here ok. So, it is expected that I will get an operator, but instead if the so,

after getting the token ID from the lexical analyzer the parser asks for the next token. And, the next token if it happens to be again ID; that means, then the parser will understand that there is some grammatical problem or grammatical mistake in the input. So, it can flag that particular error that at this point there is a problem, there is an error because it was expecting an operator and it is getting an identifier. However, the lexical analyzer tool so, it could not understand this particular problem.

So, lexical analyzer tool so, it just scanned the input and whatever is the maximally matched token at that point of time so, it has returned to the parser. So, this lexical analyzer and parser they are working hand in hand. So, they are the whenever parser is requiring token so, it is asking the lexical analyzer tool and then it is proceeding. Then it is trying to see like which grammar rule may be applied for deriving the input and then if it is successful then the process will go on. And, if it is if it can do it completely for the entire program then it will generate a parse tree. So, outcome of this parser is basically two things: one is one is an YES NO answer.

(Refer Slide Time: 13:19)



One is an YES NO answer that is whether the source program that is given is following the grammar or not. So, if the source program is following the grammar then the parser will answer YES, if it is not following the answer grammar so, it will answer NO. So, if it answers YES in that case so, we can make it to generate another output which is known as parse tree. So, this will show how the grammar rules have can be applied to

have the input source program derived from the start symbol of the grammar. And then the later stages of the compiler so, they can take help of this parse tree and can generate appropriate output ok. So, it may be some code, it may be some something else whatever it is so, some action. So, whatever it is so, it can do that part and also if the answer is NO, in that case the parser can tell us at which point it found the problem. As I was telling so, it was expecting an operator while it got ID. So, it can flash that message that I was expecting an operator so, there is a mismatch. So, it can flash it can identify that situation. So, it is basically the job of the compiler designer to see that to ensure that those erroneous conditions are checked and those erroneous conditions are appropriately inform to the user ok.

Then the user will correct the source program and again give it for compilation so, it will go like this. On the other hand this parser also takes help of this symbol table. So, as I said as you said that whenever this lexical analyzer finds a new identifier. So, it installs this identifier into the symbol table and returns the index of the identifier to the parser as an attribute in the attribute yy lvl or something like that. So, this parser can again use this symbol table to get further attributes for the symbol.

So, from y from the lexical analyzer so, it has only got the index of the symbol table where the particular identifier is defined, from the symbol table it has got more information like type of the identifier or that that is the type of operators that can be applied on it and all. So, that way the parser can check all those things. So, we have got this particular role of the parser. So, it is central to the compiler overall operation of the compiler. So, it is the main part of the compiler that we have. So, we will see how this parser can be designed.

(Refer Slide Time: 16:11)

- A 4-tuple  $G = \langle V_N, V_T, P, S \rangle$  of a language  $L(G)$ 
  - $V_N$  is a set of nonterminal symbols used to write the grammar
  - $V_T$  is the set of terminals (set of words in the language  $L(G)$ )
  - $P$  is a set of production rules
  - $S$  is a special symbol in  $V_N$  called the start symbol of the grammar
- Strings in language  $L(G)$  are those derived from  $S$  by applying the production rules from  $P$
- Examples:

$$\begin{array}{l} \{\emptyset\} > E + \emptyset \mid T \\ T \rightarrow T * F \mid \emptyset \\ F \rightarrow (E) \mid id \\ \\ E \rightarrow TE' \\ E' \rightarrow +TE' \mid E \\ T \rightarrow FT' \\ T' \rightarrow *FT' \mid \epsilon \\ F \rightarrow (E) \mid id \end{array}$$

$E \rightarrow E + T \xrightarrow{E \rightarrow T} E \rightarrow E * T$

Parse Tree:

```

graph TD
    E --> T
    E --> F
    T --> T
    T --> *
    F --> E
    F --> T
    E --> E
    E --> T
  
```

To start with we define a grammar ok. A grammar is a 4-tuple  $G$  consisting of 4 say 4 parts  $V_N V_T P$  and  $S$ . So, the for a language  $L$  so, if a grammar is  $G$  then the language for it is known as is denoted by  $L(G)$ . So,  $L(G)$  it denotes the language corresponding to the grammar  $G$  and that language has got a and this grammar  $G$  definition. So, it has got 4 parse in it  $V_N V_T P$  and  $S$  where,  $V_N$  is a set of non-terminal symbols used to write the grammar. So,  $V_N$  is that so for writing the grammar so we need some special symbols; so, they will construct the non-terminal symbols whereas,  $V_T$  is the set of terminal symbols which is a set of words in the language.

So, since there this the symbols that are appearing in  $V_T$  so, they are appearing in the final language. So, they are called terminal symbols whereas, the symbols which are belonging to non-terminal set  $V_N$ . So, they are not appearing in the final program or the final input so or the final language. So, that is why they are called a non-terminal. So, we should be the final derivation that we have so, there should not be any non-terminal left. So, everything has to be replaced by terminal and then these terminals can be; terminals can be combined in some fashion to using these grammar rules to get the overall parse tree. Then apart from this  $V_N$  and  $V_T$  the set of non-terminal and terminal symbols, we have got a set of production rules for the grammar.

So, production rules actually tells like how can we proceed with in the non-terminals, how can we replace non-terminals by set of terminals or non-terminals etcetera.

So, we will see some example and S is a special symbol in the set V N set of non-terminal symbols. It is called the start symbol of the grammar ok. Now so, we will take an example. So, this is a grammar this is a grammar; so, in this grammar so, you see that some symbols are termi non-terminal symbols, like symbols like E then T F ok.

So, they are terminal symbols non-terminal symbols whereas, symbols like plus star open parentheses close parentheses id. So, these are called these are the terminal symbols because so, this particular grammar is for arithmetic expressions that has got multiplication addition as the operator and the parentheses is also there. So, you can have an expression like say 2 plus 3 into 5. So, this is supposed to be a valid string of this language. So, how can I have this grammar? How can I have this string?

So, starting with say E so, I can use the first rule E E producing so, this this part. So, we read it as E producing E plus T or T. So, this is actually combination of 2 rules E producing E plus T and another rule E producing T. So, there are 2 rules so, for the sake of brevity so, we just write them together and write it as E producing E plus T or T. So, by combining these 2 so, we will write it as E producing E plus T or T. But for, but you should keep in mind that the these actually mean that there are 2 rules: one rule is E producing E plus T another rule is E producing T, since their left hand sides are common for the rules.

So, we are writing them together. So, let us go back to the point like say for say E I can use this rule E producing E I can use the way I can have a derivation like this E producing T. And, then T producing T star F and then this T producing F producing E and this E produce F producing within bracket E and then this E again giving the remaining part. So, this is actually giving E plus T ok, let me draw it a fresh because it is not.

(Refer Slide Time: 21:21)

**Grammar**

- A 4-tuple  $G = \langle V_N, V_T, P, S \rangle$  of a language  $L(G)$ 
  - $V_N$  is a set of nonterminal symbols used to write the grammar
  - $V_T$  is the set of terminals (set of words in the language  $L(G)$ )
  - $P$  is a set of production rules
  - $S$  is a special symbol in  $V_N$  called the start symbol of the grammar
- Strings in language  $L(G)$  are those derived from  $S$  by applying the production rules from  $P$
- Examples:
  - $E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow (E) \mid id$
  - $E \rightarrow TE'$   
 $E' \rightarrow +TE' \mid E$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT' \mid E$   
 $F \rightarrow (E) \mid id$

The parse tree diagram shows the derivation of the expression  $(2+3)*5$ . The root node is  $E$ , which derives  $T$  and  $T'$ .  $T$  derives  $T$  and  $F$ . The left  $T$  derives  $2$  and  $+$ . The right  $T$  derives  $3$  and  $*$ .  $F$  derives  $(E)$ , which further derives  $id$  (with value  $5$ ). The left  $T$  also derives  $E'$ , which derives  $+TE'$ . This part of the tree is enclosed in a dashed oval.

So, the expression that I have is 2 plus 3 star 5 ok. So, I start if I start with E then E producing T first of first of all I have to derive this part. So, this multiplication of 2 expressions so, the this is expression 1 this is expression 2. So, I have to do it a multiplication of 2 expressions. So, that is what we are trying to do. So, T producing E producing T and this T producing T star F ok. Then this from this F I will have id which is 5. Now, for the left T from this left T I have to derive this part, in this 2 within bracket 2 plus 3.

So, from this T we write F and from F we write open bracket E closing bracket and then from this E we have got E plus T and from this E we get T to F to id; which is 2 in our case and then this T gives F gives id which is 3 in our case. So, this is called the parse tree right. So, this will be more clear as we proceed through the remaining part of this course; so, this will be more or less it will be this is what we are exactly going to do in our future lectures. So now, you see that this particular grammar is used for deriving expressions, that involves addition and multiplication.

Now, this is another grammar ok. So, apparently it seems that they are very different from each other, but this is also same as this one. So, we can we will see later that these 2 grammars are equivalent. So, it is not mandatory that for a particular language there can be only a single grammar. So, different people can come up with different grammars and

if you use this particular grammar then the way we should do the derivation is like this that so, we have got 2 plus 3 star 5.

(Refer Slide Time: 23:49)

**Grammar** (2+3)\*5

- A 4-tuple  $G = \langle V_N, V_T, P, S \rangle$  of a language  $L(G)$ 
  - $V_N$  is a set of nonterminal symbols used to write the grammar
  - $V_T$  is the set of terminals (set of words in the language  $L(G)$ )
  - $P$  is a set of production rules
  - $S$  is a special symbol in  $V_N$  called the start symbol of the grammar
- Strings in language  $L(G)$  are those derived from  $S$  by applying the production rules from  $P$
- Examples:

```

E -> E + T | T
T -> T * F | F
F -> (E) | id

E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> *FT' | ε
F -> (E) | id
    
```

The slide also shows two parse trees for the expression  $2+3*5$ . The first tree starts with  $E$ , branches into  $E$  and  $T$ , where  $E$  branches into  $2$  and  $+$ , and  $T$  branches into  $3$  and  $*$ . The second tree starts with  $E$ , branches into  $+$  and  $E'$ , where  $+$  branches into  $2$  and  $3$ , and  $E'$  branches into  $*$  and  $F$ , where  $*$  branches into  $5$  and  $F$  branches into  $id$ .

So, these you have to start with  $T E$  and then  $E$  we have got only one rule  $T E$  dash. And, from this  $T$  I have to derive this within bracket 2 2 plus 3. So, for doing that so, we replace this  $T$  by  $F T$  dash and then this  $T$  dash can give rise to epsilon using this rule  $T$  dash giving epsilon and from this  $F$  we do it like within bracket  $E$  and from this  $E$  I have to get that 2 plus 3. So, from this  $T E$  is from this  $I$  again do  $T$  and  $E$  dash and from this  $T$  from this  $T$  I will be going to  $F T$  dash and this  $T$  dash will give epsilon, this  $F$  will give  $id$  which is 2.

Now, from this  $E$  dash I will do plus  $T E$  dash to plus  $T E$  dash and the now this  $E$  dash part can be made to epsilon. And, then this  $T$  will be giving me  $F T$  dash and then this  $T$  dash will give me epsilon. Then this  $F$  can give  $id$  and which is equal to 3 in our case ok. Now, for this  $E$  dash part from this  $E$  dash I have to derive this star that star 5 ok. So, for that so, I have to so, for this sorry not from here actually from this  $T$  dash I have to do that, that star  $F T$  dash that star will come from here.

So, this is star  $F T$  dash and this; this  $T$  dash will give me epsilon and this  $F$  will give me  $id$  which is 5 and this  $E$  dash will give me epsilon. So, this way I can have another parse tree for the same expression, but using a different language. So both the parse trees are correct only because, the grammar is different. So there the trees are appearing to be

different and it appears to be very difficult to draw the parse tree. So I was just trying to do it intuitively starting with the; starting with the grammar rules and trying to derive the final string.

So it is very much possible that we get mislead and we go to some other derivation which does not lead to the final string. So this our discussion on this parser generator will actually avoid all these things. So it will guide us so that we can select the proper production rules at different points ok. And, we can operate, we can derive the final parse tree which is which will be correct ok. So next we will be looking into some error handling mechanisms, but before that so there can be different types of grammars that I would like to introduce; one is that one type of depending upon their capacity.

(Refer Slide Time: 27:41)



So, there are different types. So, type-0 type-1 type-2 and type-3; out of them type-0 is the most flexible. So this is the most flexible or most powerful, most flexible and powerful and type-3 is the least flexible least flexible and it is naturally power the power is also less. So power in the sense that thus the type of languages for which it can construct a grammar ok. So if you try to construct a type-3 grammar then you will find that for many languages you cannot do that ok.

And type-1 and type 2 they are some intermediary types ok. So, this type-3 languages so, they are also known as regular language. And the corresponding grammars are known as regular grammar and incidentally the regular expressions that we have seen in our lexical

analysis discussion so they fall into this particular category, they fall into this class of regular grammar. Then this type-2 so this is known as context free grammar; context free grammar so most of the programming language constructs they will belong to this context free category ok.

So, most of the constructs that you have in normal programming languages, that we are familiar with; so they belong to the context free grammar. So most of the time our discussion will be centered on context free grammars only. Then type-1 it is known as context sensitive grammar. So they are known as context sensitive grammar because, we will take some examples later. So where it will show that it can be more powerful than context free grammar and type-0 grammar so, they are known as free grammar ok.

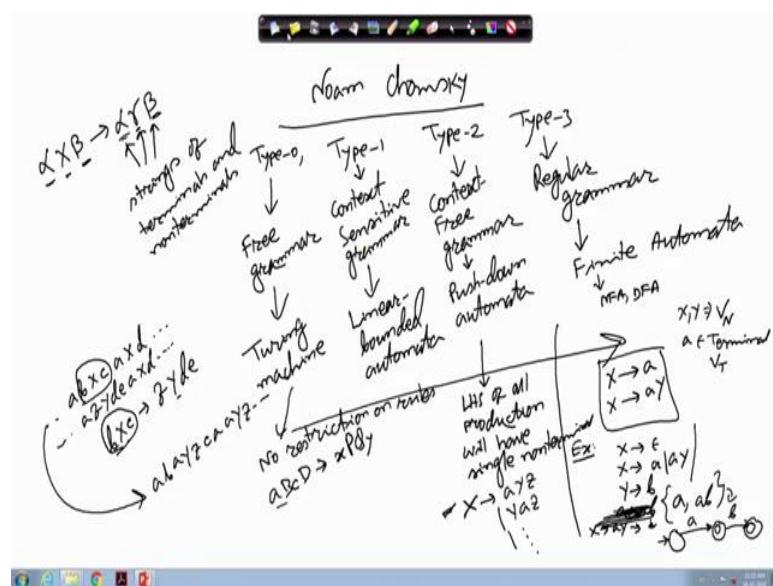
So accordingly so if you try to design a recognizer then designing recognizer for type-3 is the simplest job that we have already done in terms of finite automata NFA and DFA. Type-2 is more complex where you will need a stack apart from the finite automata. So that there it is known as pushdown automata so that we will be able to do that. Then if the constructing the other acceptors for type-1 and type-0 they will be more and more difficult ok. So most of the time we will be restricted to context 3 type type-2 grammars only, that is context free grammars because programming language constructs they belong to this particular category.

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

**Lecture – 17**  
**Parser (Contd.)**

There is a language hierarchy that is known as Chomsky's hierarchy.

(Refer Slide Time: 00:20)



So, it is defined by Noam Chomsky. So, according to Chomsky there are four types of languages they are known as type 0, type 1, type 2 and type 3. So, out of this type 0 is the most flexible class of languages because the corresponding grammar that we have is known as free grammar. So, this is called free grammar and the successive classes or successive types.

So, they are more and more restrictive in nature. So, type 3 is the most restrictive version which is known as the regular languages and the corresponding grammar is called regular grammar then we have got type 2. So, type 2 is known as context free grammar, context free languages and the corresponding grammar is called context free grammar and this type 1 is more or (Refer Time: 01:46) more less the constant compared to type 2. So, they are known as context sensitive grammar or context sensitive language.

So, from the name we can understand that this free grammar it can accommodate all other types of languages, but as we are going from type 0 towards type 3 the languages are becoming more and more restrictive in nature. So, for each of these classes of languages so we can have different types of accepters available from the automata theory other or the computation theory and this for the regular grammar we have got the finite state automata. So, other the finite state machine or finite automata. So, this is the class of this particular tool can accept type 3 languages. So, we have examples like this NFA, DFA etcetera.

So, all the regular expressions that we have seen in our discussion during lexical analysis they belong to this type 3 language and we can have the corresponding regular expression. So, we can draw the responding NFA and DFA and also we can write down the grammar in some forms. So, I will welcome to that slightly later, then in the context free category. So, here the corresponding acceptor is known as push down automata. So, this is basically apart from the finite machine.

So, you will need a stack for designing and acceptor for the type 2 languages. So, they are known as push down automata. Similarly in context sensitive category we have got linear bounded automata, linear bounded automata which can be used for designing acceptor for this type 1 language and this type 0 or free grammar. So, here we have got Turing machine ok. So, as the acceptor machine that can detect whether a string belongs to the particular language or not so that can be done that can be constructed using Turing machine for free grammar.

Now, out of all this machines Turing machine is the most generic one and naturally we can so that is going to be most powerful; however, constructing Turing machine as you know the Turing machine is a hypothetical machine. So, we cannot construct it because practically because the tip size is going to be infinite. So, this is a theoretical machine only, but it is the most powerful computation platform that we can think about. And as we go from this towards these finite automata they become more and more destructive in nature and the corresponding acceptor day becomes easier to design.

Now, let us look into the corresponding grammar like will start with the regular grammar and then slowly go to the other category. So, regular grammar we can have grammar rules of the form  $X$  producing  $a$  or  $X$  producing  $a Y$  where  $X \neq Y$ . So, these are non

terminal symbols. So, these are they belong to the class V N that is the class set of non terminals and a is a terminal symbol so a is a terminal symbol belonging to the set V T ok. So, the all grammar rules will be of this form X producing a or X producing a Y. If you can take an example life suppose I have got a got an example like this say X producing epsilon X producing a or a Y and Y producing b. You see that here all the production rules on the left hand side we have got a single non terminal, on the right hand side we have got a single terminal symbol or a single terminal followed by a non terminal. So, either these two combination.

So, if a grammar is like this then will tell that the grammar is a regular grammar. So, we can quickly try to understand what is this grammar. So, you can I see that this grammar it will accept all those languages that can have at most one b and the b if the b is there. So, it will appear only at the end like I can have strings likes say a a, a b. So, this can be derived starting like this x producing a y then. So, sorry it is not a a, a b. So, I can have the strings are X producing a or a Y and Y may be replaced by b. So, I can have a single a or the string a b. So, only these two.

So, these two is the languages, these two are the belonging to the language now. So, this is an example of context free sorry this is it is an example of regular grammar and their type 3 grammar and the corresponding language is a type 3 language. Next will look into context free grammar. So, in a context free grammar; so all production left hand side it will have a single non terminal. So, here the idea is that left hand side of all production will have single non terminal, single non terminal symbol.

So, typical examples are like this, I can have say X producing a Y Z like this where X Y Z. So, these are non terminals and a is a terminal. So, on the right hand side there is no restriction. So, another rule maybe say Y a Z. So, like that I can have a number of rules so, on the right hand side it do not have any restriction, but on the left hand side there is only a single non terminal symbol. So, if the grammar can be specified by means of production rules like this, then the grammar will be called a type 2 grammar or context free grammar and the corresponding language will be context free language.

And as I have told in the last class that most of the programming language construct they belong to this type 2 category. So, there most of the grammar, that will be using for this different different programming languages, so they follow this context free category. So,

that is why these has become very popular for the compiler design codes and everywhere will find that will be talking about this type of languages; then comes this context sensitive grammar. So, as the name suggests the context sensitive. So, every rule has got a context. So, context it is like this if I have a.

So, maybe I have a rule like alpha X beta producing alpha gamma beta where this is gamma is a alpha beta gamma. So, they are all strings of terminals and non terminals strings of terminals and non terminals. So, you see that here the thing is that this X has been repaired has been we can be replaced by gamma only if it is preceded by alpha and followed by beta.

So, that is why it is there is a context. So, the context is that x is preceded by alpha and followed by beta, then only we can apply this rule to change the x to gamma and alpha beta they remain un change. So, they just hold the context that is why it is called a context sensitive grammar. So, if you are having a string like say a b X c a then say X d like that and suppose I have a rule which says that b X c can be replaced by say f Y d E where X and Y they are non terminal and the b c d, a b c d E F so they are all terminal symbols. Now you see that this rule b X c matches with this part ok. So, I can replace this part by this right hand side.

So, I can from this I can derive the string a f Y d e, but this x cannot be modified because the context does not match. So, here the context is a d, but here it is b c. So, contest does not match so I cannot do this replacement so, it remains as it is had it been a context free grammar. So, like this and it been a context free grammar I would have replaced both axes by these rules. So, I could have written like a b and then X replaced by Y Z, a Y Z then c a then again this X replaced by a Y Z. So, I could have done like this, but in a context sensitive grammar since there is a context. So, that this left hand side has got a context. So, you cannot replace it freely, only when the context matches you can do the replacement.

So, it is even better in the sense that you can more precisely frame the grammar rules ok. So, that is why this is more powerful than type 2 languages and type 0 does not have any restriction. So, there is no condition regarding this context like while writing grammar for context sensitive language.

So, you cannot change from this alpha and beta over the rule; so, you cannot. So, this alpha and beta they remain unchanged from the left hand side to the right hand side of the rule, but if in case of type 0. So, that restriction also does not exist. So, there is no restriction on the grammar; no restriction on grammar.

So, you can have rules like  $a \rightarrow B c D$  producing  $x \rightarrow P Q y$ . So, you can have some rule like this. So, it is totally independent of this context. So, I am not maintaining the context of this left hand side or to the right hand side they totally modified.

So, we can have total freedom in the writing down the corresponding rules for the grammar. So, this is the type 0. So, type 0 is the most flexible one as a result none of the automata they can be used for accepting this type 0 languages. For type 0 you have to reward to Turing machine for designing the acceptor and as I have said that we will be mostly concentrating on this type 2. Because type 2 is the context free grammar and most of the programming language constructs they will be based on type 2.

So, we have already seen type 3 which is regular grammar and we have seen that a regular expression. So, you can always construct a finite state machine for that for example, for this language you can very easily make a DFA. So, if this is the start state on  $a$  it goes to these state this is the final state and on  $b$  also it goes to another final state

So, that is  $a$ . So, you can draw on DFA which will be doing this operation ok. So, this type 3 can be there for the regular grammar and that is for regular expression.

(Refer Slide Time: 14:32)

# Grammar

- A 4-tuple  $G = \langle V_N, V_T, P, S \rangle$  of a language  $L(G)$ 
  - $V_N$  is a set of nonterminal symbols used to write the grammar
  - $V_T$  is the set of terminals (set of words in the language  $L(G)$ )
  - $P$  is a set of production rules
  - $S$  is a special symbol in  $V_N$ , called the start symbol of the grammar
- Strings in language  $L(G)$  are those derived from  $S$  by applying the production rules from  $P$
- Examples:
 

$\begin{array}{l} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid id \end{array}$	$  \begin{array}{l}  \textcircled{E} \rightarrow E + T \rightarrow E + T * F \rightarrow E + T * (E) \rightarrow E + T * (T \rightarrow E + T * (T * id)) \\  \rightarrow \textcircled{E} + T * (T * id) \rightarrow E + T * (F * id) \\  \rightarrow \textcircled{E} + T * (id * id) \\  \rightarrow \textcircled{E} + F * (id * id) \\  \rightarrow E + id * id \\  \rightarrow T + id * id \\  \rightarrow F + id * id \\  \textcircled{id} + id * (id * id)  \end{array}  $ <p style="text-align: right;"><math>2 + 3^*(4+9)</math></p>
--	---

So, with this we will be going back to our discussion on grammar. So, grammar as I have said that it say 4 tuple  $V N$ ,  $V T P$  and  $S N$  is the set of non terminal symbols  $V T$  is the set of terminal symbols  $P$  is a set of production rules. And we have seen that depending upon the structure of  $P$ , we can have different types of different class of grammars different types of grammars and  $S$  is a special symbol in the set of non terminal  $V N$  which is called the start symbol of the grammar.

So, so, and the strings of the language they will be represented by  $L_G$  then they are the strings that can be derived from  $S$  by applying the production rules or from  $P$  ok. So, any grammar for which we have got for which we have written the production rules, many times what will be happen is that we will not be explicitly mentioning the start symbol in that case the first, the very first non terminal symbol that appears in the grammar specification.

So, that is the start symbol for example, for this particular grammar E is the start symbol because E is the first non terminal among the set of rules that we have written. For the second grammar also E is the start symbol, now starting with E you can derive strings like this. So, you can apply the first rule E producing E plus T, then you can replace this T by T star F, then you may decide that this F I will be replacing by within bracket E. So, E plus T into within bracket E and then this within bracket E.

That can be modified to say T by applying this rule E producing T from this you can have E plus T star T producing F ok. So, say do not if say T star F then from this thing you can derive like E producing E plus oh deriving is already there. So, E plus T into T star id from this T can be again be replaced by say F giving us E producing T star F star id, giving E producing T star id star id, giving E producing E producing part is not necessary. T this T T can be replaced by F F star oh sorry this was E plus. So, this E plus part is there F, F star F into id star id then that can give us E plus.

This F can be made to given id then this E can be made to give a T T plus id star id star id. So, this T can be replaced by F so you can get it like this then this F can be replaced by id. So, id plus id star id star id. So, I write these an identifier. So, maybe I have got an expression like 2 plus 3 into within bracket 4 into 5 ok.

So, this shows the derivation starting with the start symbol of the grammar. So, finally, we could derive this string. So, this string we could derive this string using the grammar rules. So, this string belongs to the language accepted by the grammar. So, this appears to be very cumbersome; however, the whole process is automatic. So, once we can write down a piece of code that can do this transformation by suitably selecting the rules from the grammar, then the whole process can be automated very easily.

So, this compiler, this parts are designing task or the two the knowledge that will gather during this parts are design a discussion. So, that will help us to design this type of automated tools.

(Refer Slide Time: 19:40)

The slide has a yellow header with the title "Error handling". Below the title is a bulleted list:

- Common programming errors
  - Lexical errors
  - Syntactic errors
  - Semantic errors
  - Lexical errors
- Error handler goals
  - Report the presence of errors clearly and accurately
  - Recover from each error quickly enough to detect subsequent errors
  - Add minimal overhead to the processing of correct programs

On the right side of the slide, there are some handwritten notes and diagrams. One note says "if ac [unclear] else [unclear]" with a circled "2F" above it. Another note says "if c S1 else S2". There are also some small diagrams of boxes and arrows.

The footer of the slide shows the "swayam" logo and other navigation icons.

So, coming to the error handling; now one also it is very common that the input specification that is given the input file that is given for parsing it has got some errors. So, particularly they are written by for programming languages. So, users are writing programs and programmers often do mistakes ok.

So, those mistakes can lead to different types of errors, first one is the lexical error. So, lexical error are those errors which are ah some which do not constitute any valid word of the language. So, for example, if I have got say if I have got say a one symbol that that is a 2 F. So, 2 F also this is not valid because the it does not fall into any of the regular definition for the language. So, that is a lexical error, then we have got syntactic error. So, syntactic error means there is some grammatical mistake.

Like say if then else statement it says that if some condition, if some condition then some statement else some other statement say S 1 and S 2. Now, if I have not put this then so, if I have written like if c S 1 else S 2. So, as far as the lexical analysis tool is concerned. So, it will identify this individual tokens if else then that tokens that will constitute this condition, tokens that will constitute the statement. So, it will find those tokens, but when it comes to the syntax analysis phase so, it will be taken as a it will be detected as a syntactic error because this does not make a correct statement of the language.

So, it does not follow any grammar rule. So, that is the syntactic error. So, another class of error at the semantic errors. So, it is the program is or the statements are syntactically

correct, but semantically there is some problem this is particularly ah visible for the programming languages where the variables I have got some types and the types are all predefined. So, for example, in the language c so if you have got an assignment like x equal to y plus z then this x y and z they must be predefined variables.

So, in your program you must have define somewhere for example, integer xyz that must have been declared somewhere. So, if this is not there then x y z will not be there in the symbol table and when parser we will try to parse this thing. So, lexical analyzer will written x as id, y as id, z as id. So, that way the, it is an expression id plus id then this is an assignment statement everything is correct syntactically it is absolutely correct. But I cannot derive the type of this variables and whether this plus operator is applicable on y and z that is also not known for example, if y and z are say character arrays then y plus z does not have any meaning.

So, these are semantic errors. So, the semantic errors are also serious and they are to be detected, but that that is not a grammatical error because grammatically it is correct, grammatically we have got this identifier plus identifier. Now meaning of those identifiers or the types of those identifiers we will tell us whether the construction is semantically correct or not. We have got semantic errors then they there are so of course, the lexical errors are there. Now how to do error handling?

So, error handling means somehow the compiler. So, it should detect those errors and not only that the detection. So, it should tell it a, the user that here is the error. So, it should pin point the error like what is the error and at which point in the program the error has occurred. So, error handler it will have the goals like it should be able to ah report the presence of errors clearly and accurately. So, that is the first thing that the some error is there.

So, what is the error and at which place. So, that should be told very clearly, it should be able to recover from each error quickly enough to detect subsequent errors. So, as I was telling in some classes earlier that error handling is important because if you are detect one error and if you just on detection of the first error if the compiler quits the compilation job and it will it ask the user to correct it and maybe user we will correct it and then give it for compilation, just to find that after two three lines again there is another error. So, typically compilers work in a fashion in which it produces all the error

messages together. So, that user can correct all those errors and then give it for compilation.

So, but for doing that it is very difficult because particularly for synthetic error since this compiler or the parser is working based on some automata it may go to a state from where it is very difficult to come to a clean state. So, so that is the, that is a very important job. So, somehow the compiler must be able to decide that I have to discard the next few tokens and come to a state which is clean and I can start looking into the new tokens from that point. For example, most of the programming languages so they have got the feature, that any statement that you have it ends with a semicolon.

Now, while doing the parsing operation so suppose there was an error at this point then somehow the compiler must keep through all these tokens and it should come to the next semicolon and from this point onwards it is expected that the effect of that error is gone ok. So, from this point onwards the compiler should start doing the parsing face and try to see whether it can detect more errors and not. So, if the recovery is not proper when the compiler we will produce arbitrary error messages that will mislead the user.

So, this is very important that the user, that the compiler can come up to a decent state from which it can produce proper error messages for the subsequent errors as well as. So, it should add minimal over it to the processing of correct programs. So, if the program this error handling part. So, it should not be such that it creates the code generate makes the code generation process for correct programs to take long. So, it should not happen like that. So, the compilation should be first at the same time it should be able to decide on those ah on those errors and it should be able to recover from there. So, this is a very important, this is a very important thing to do.

(Refer Slide Time: 26:54)

**Error-recovery strategies**

- Panic mode recovery
  - Discard input symbol one at a time until one of designated set of synchronization tokens is found
- Phrase level recovery
  - Replacing a prefix of remaining input by some string that allows the parser to continue
- Error productions
  - Augment the grammar with productions that generate the erroneous constructs
- Global correction
  - Choosing minimal sequence of changes to obtain a globally least-cost correction

Handwritten notes:

- S → it is a set of synchronization tokens.
- If C is in S, then S2 is found.

SWAYAM

So, we will see how this can be done. There are several error recovery strategies followed by these compiler designers one is known as panic mode recovery. So, panic mode recovery is like this suppose we are at same place and there is a crowd and in the crowd there is a there is some sort of news comes that something some something has happened maybe there is a fire or something like that.

When the everybody wants to come out of the building for example. So, if people who if the people are inside a building all of them try to come out and we say that is a panic mode may be, it is not mandatory that we should come out so early, but this we most of the people will all the people will do like that. So, this is known as panic mode recovery. So, what the compiler does is that it will discard input symbols one at a time until one of the designated set of synchronization tokens is found.

So, what is the synchronization token? So, as I have said that depending upon the programming language you can define when a particular block ends for example, for the C language. So, you know that say every statement ends with a semicolon. Now if discarding up to semicolon is also not sufficient you find that still the compiler the parser could not come to descend state another synchronization token is the closing brace. Like (Refer Time: 28:18) you normally you whatever we have. So, every block of statements so that is enclosed within a brace, pair of brace.

So, if there is some error in this statement and by discarding this statement we could not come out of the error then what the compiler we will do is that it will skip the success statements also till it comes to the brace. So, that it has fifth and entire block. So, that is one possibility. So, some programming language so they will allow you to have individual procedures and the begin end block markers begin procedure, end procedure markers. So, in that case the compiler can skip up to that point. So, depending upon the programming language that we fall which you are designing the compiler. So, we have got different time synchronization tokens by analyzing the program you can understand what are the, we can what are the synchronization token. So, this is known as panic mode recovery..

Then there is a phrase level recovery. So phrase level recovery it will try to replace a prefix of remaining input by some string that allows the parser to continue. So some part of it may be there is some there is something like this for example suppose my grammars, my language says that there is an if then else statement if condition then statement else statement. Now it may so happen that this then token is then this it is then part is missing.

So what will happen the parser or it will see this if part condition part then it will see the statement part. So, what the parser can do so it can be it can intelligently determine that here what is missing is a then token. So it can insert this then token into the input string. So, that way it will replace a prefix of remaining inputs by some string. So this is the remaining string from this point onwards. So this is the remaining string. So it will replace this part by this it introduces this then at this point. So the program did not have, the program that you are considering has got something like this if C S 1 else S 2. So this part is not yet seen so it has got S 1.

So it will understand that this I have to put a prefix then at this point then the whole thing will become meaningful. So this is the phase level recovery. So it tries to introduce some phrases at the beginning of this of this input part and then it can continue. So this way we can have different error recovery policies ok. So there are error production and global correction that we will see in the next class.

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

**Lecture – 18**  
**Parser (Contd.)**

Another type of error recovery strategy is by means of error production. So, error production says that the language designer has given as a state of production rules that tells how the valid statements of the language can be designed , can be generated. Now, compiler designer can think about the possible errors that the user can do and accordingly add some few more productions into the system which will be called error productions.

So, if a particular, going back to that example that we are taking of if then else statement so it is.

(Refer Slide Time: 00:53)

**Error-recovery strategies**

- Panic mode recovery
  - Discard input symbol one at a time until one of designated set of synchronization tokens is found
- Phrase level recovery
  - Replacing a prefix of remaining input by some string that allows the parser to continue
- Error productions
  - Augment the grammar with productions that generate the erroneous constructs
- Global correction
  - Choosing minimal sequence of changes to obtain a globally least-cost correction

So, it was like this the S producing if some condition then statement else statement. Now, this is a rule from the language, this is a production rule from the language. Now compiler design can interpret or can try guess that the type of error, type of mistakes that the user can do is forgetting these then.

So, accordingly the compiler designer can add another rule to this language. So, if C S else S then what will happen if a program has got a line where we have got this where the then party is missing then what will happen is that this rule will not match, but this rule will match, but the compiler designer knows that this is an error production. So, it should not try to generate code based on this.

But the parser it will be able to proceed with this rule and it will not fall into an error condition. So, that is basically used in the error recovery of the parser. So, this is the error productions. So, you can augment the grammar with productions or that generate the erroneous constructs. So, this is the other possibility and also you can do some global correction. So, we can find out some minimal sequence of changes to obtain a globally least cost correction. So, this is a very I should say costly approach. So, what it is trying to do is that there is an input program.

(Refer Slide Time: 02:38)

## Error-recovery strategies

- Panic mode recovery
  - Discard input symbol one at a time until one of designated set of synchronization tokens is found
- Phrase level recovery
  - Replacing a prefix of remaining input by some string that allows the parser to continue
- Error productions
  - Augment the grammar with productions that generate the erroneous constructs
- Global correction
  - Choosing minimal sequence of changes to obtain a globally least-cost correction

So, there is an input program say P and there is that has got some errors in it. So, what it will try to do is to transform this P into if another program P dash where all those errors have been corrected and that connection is global. So, for example, if you have got say in your program suppose you are using a variable x and at various places in the program so you are writing like x equal to y plus z, P equal to x into k like that and if this variable x is undefined then you see that at every place it will generate a message that x is undefined, x is undefined like that. So, you can transform this program into another

program where at the beginning you guess what is the type of x, since I am doing this addition and multiplication integer operation. So, I can define x as integer ok. So, this line may be introduced into the program.

Rest of the thing remains as it is only this extra line has been introduced. So, what will happen this program will not generate any semantic error. So, this that way the parser will be able to process the compilation process will be able to proceed or say. So, x will get installed into the symbol table as a result it will that lexical analyzer will be able to return proper token that x is an identifier token so like that. So, it will help in the compilation process, but there can be many ways there can be many interpretations.

So, why do we replace x by a constant number. So that is also a correction. But so, that way what is the minimal number of changes that can be done in the program to obtain a globally least cost correction. So, that is the challenge of this global correction strategy. So, this is particularly useful where you have got text formatting type of application where, we try to show the output to be as correct as possible to the user and if the user is not satisfied with the output we will perhaps go to the correction.

But many times what happens is that the correction done by the formatting tool is good enough and we do not plus we do not do any further corrections. So, this global correction has got applications in text formatting type of cases.

(Refer Slide Time: 05:04)

Context free grammars

- Terminals
- Nonterminals
- Start symbol
- Productions

expression -> expression + term  
expression -> expression - term  
expression -> term  
term -> term \* factor  
term -> term / factor  
term -> factor  
factor -> (expression)  
factor -> id

2\*3

FREE ONLINE EDUCATION  
swayam

So, next we will be looking to the in more detail the grammar. So, this is coming back to the context free grammars. So, as I was telling that most of the programming languages so they have got this context free grammars, they can be specified in using context free grammar. And, as I said that it has got four set series terminals, non-terminals, a special start symbol which is a which is a member of the set of non-terminals and the set of production rules.

So this is a typical example of a grammar. So this is known as expression grammar. So, throughout our course many times we will be referring to this particular grammar. So, we read it like this, expression produces expression plus term then or expression produces expression minus term or expression produces term. So, term is basically product of two sub expressions and expression can be sum of two sub expressions or the sum or subtraction of two some sub expressions or it may not have any such addition subtraction.

So if it is say and only products. So 2 into 3 in that case this whole expression is considered as a term using the third rule. And, then this term is taken as it is take a term is can be term multiplied by factor or term divided by factor or a term can be simply a factor and factor can be within bracket expression or a or a single identifier. So, this way we have since previously some derivations.

So, this is a grammar and we will be calling this as expression grammar and it takes care of expressions, arithmetic expressions having the operators like plus minus multiplication division and over the parenthesis. So, it takes a it can detect parenthesized expressions involving identifiers and the arithmetic operators plus, minus, multiplication, star and slash addition, subtraction, multiplication and division. So, this is a good example and very often we will be referring to this example.

(Refer Slide Time: 07:24)

Derivations

- Productions are treated as rewriting rules to generate a string
- Rightmost and leftmost derivations
  - $E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid id$
  - Derivations for  $\neg(id+id)$ 
    - $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$
    - $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$

So, derivation; so we have already seen something called something like derivation like see suppose we have got an expression like this a grammar like this. So, it has got a set of rules if producing E plus E or E star E or minus E or within bracket are id fine. So, this is the grammar. So, we can have a derivation for this one, say suppose this is the expression given to us minus id plus id. So, how can we derive it? So, we can do it like this first we can we apply this rule.

E producing minus E so that this unary minus is taken care of then this E is replaced by, this E is replaced by within bracket E. So, this is open parenthesis this is close parenthesis this is parts have been generated, then this E is replaced by E plus E is because I need to generate this id plus id, then this first E is replaced by id and in the next case the second E is replaced by id. So, this is one possible derivation of the string minus of id plus id starting with the start symbol of the grammar E.

Now, somebody may say that I have a choice at this point. So, when I am looking into this E plus E I have a choice whether I can replace this E by id or I can replace this E by id, in the first case we have replace the first E by id. So, there can be another derivation where the second E is replaced by id first and in the next generation the first E is replaced by id.

So, in general I can say that if I have got a string ok. So, I have started with the start symbol of the grammar and from there I am driving the strings like this, than at any point

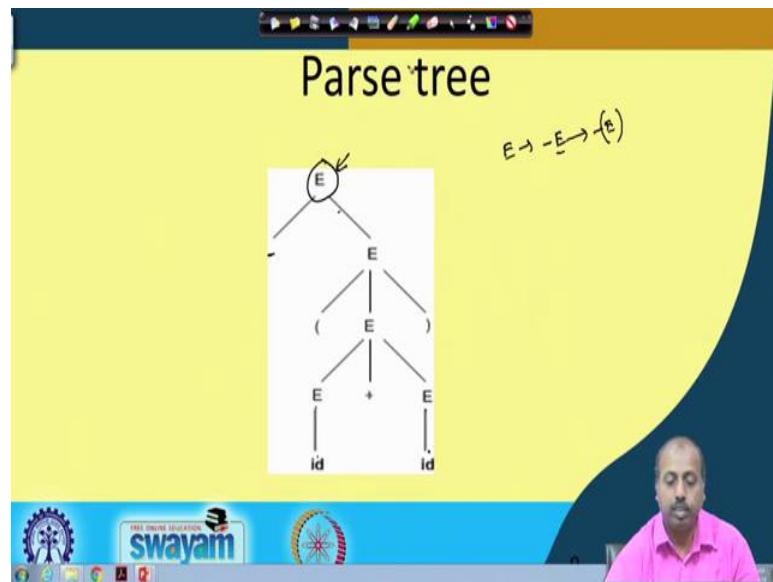
of time suppose I have got some string like id plus E star E plus id like this plus E, from this I want to generate my next string ok. Now there are possibilities like I can replace this E I can replace this E or I can replace this E I have got three choices.

Now, there are two derivations that have been followed into a by the compiler designers, one is that you replace the leftmost non terminals that is this one. So, that is known as leftmost derivation. So, in the next derivation so if I write it like id plus this E producing id star E etcetera etcetera. So, that will be a left most derivation because, I have replaced the left most non terminal by the right hand side of the corresponding production rule or I can replace this E.

So, I can have the derivation like E star E plus id plus this E producing say within bracket E that can be done. So, I have replaced the last E the rightmost E in this case. So, we can have two different derivations, one is called a left most derivation another is called a rightmost derivation. So, in the rightmost derivation from the intermediary string the rightmost non terminal symbol is replaced by the corresponding right hand side of the production rule and in the leftmost derivation the left most non-terminal is replaced by the right hand side of the corresponding production rule.

So, these are this is known as derivation. So, productions are treated as rewriting rules to generate a string. So, they are they are basically productions are nothing, but rewriting rules. So, we can replace one string by another string by applying those rules and there are derivations like right most derivation and left most derivation.

(Refer Slide Time: 11:37)



We can think about parse tree like the derivation that I have drawn I have done in the last slide. So, you can think of it to be like this. So, E is the start symbol of the grammar and from there we have got; E is the start symbol of the grammar and then I have replaced the rule E E I have used the rule E producing minus E.

So, minus so it has got two children nodes, one is the minus the terminal symbol minus another is the terminal symbol E and the next step this E is replaced by within bracket E. So, it so the derivation that we had is minus within bracket E; so, this E is replaced by within bracket E. So, that is the next part of the tree, now from this I can have this E producing E plus E. So, this E is replaced by E plus E. So, that is and the next phase of this tree that we have, next level of the 3 E plus E and the next level this E produces id and this E produces id. So, this way I can have a tree representation of the derivation.

Where the first derivation that I have done, so it constitutes the first part of this tree and the route of this tree is the start symbol of the grammar and from there the rules that I am applying. So, based on that we can have different we can have different branches in the tree. So, this is the parse tree, parse tree type structure.

(Refer Slide Time: 13:21)

Next we will have sometimes so there can be ambiguities in a grammar for example, suppose we consider this particular specific this particular statement or example id plus id star id. Now, I am following the grammar, I have got a simple grammar like E producing E plus E or E star E or id since I do not have parenthesis and all in this particular string. So, I do not need to consider the other constructs that the grammar may have, other rules that the grammar may have. So, we have got only these rules. So, one derivation for this id plus id star id may be like this, we start with E then replace it with E plus E then the first E is replaced by id second is replaced by E star E and then this E is replaced by id and this E is replaced by id.

So, this is one particular tree, another parse tree can be like this. So, here I first I replace this thing by E E star E. So, instead of doing it like this so in plus. So, I am starting with this star. So, E producing E star E and then this E it produce, it gives me E plus E and this E gives me id this E gives me id and we have got. So, the derivation sequence in this case is E giving E plus E from their giving id plus E from their id plus E star E from their id plus id star id. So, id plus id star E and this is giving id star id.

And in this case what has happened? I have started with E and from there I have got E star E, from there I have got this E replaced by E plus E. And then star id star E. So, let us do it like that it is then this E is replaced by id plus E star E then this is giving me id plus id star E and that will give us id plus id star id. So, both in both the cases I have got

the final string id plus id star ids in both the cases and both of them are doing left most derivations ok. So, here also I am substituting the left most non terminal first in the sub string. So, here also I am doing the same thing. So, if I do that, but even. So, both of them are parse trees corresponding to left most derivations, but even if I do that you see that we have got two distinct parse trees for the same strings of the language. Now which one is correct?

So, sometimes so, whenever you have got more than one such derivations. So, we say that the grammar is ambiguous in nature because it has got more than one interpretation of the same string. In fact, if you will be you if you look into it more carefully then in this case you see that here this if you if you go in a bottom of fashion then this is the id so this is the id is replaced by E and then E star E is replaced by e. So, you can understand that this expression multiplications is done first and then it is added to the first id.

Whereas in this case this id plus id. So, they will be the so this thing is the. So, ultimately this E plus E is done first. So, addition will be done first and then the result will be multiplied by this is E plus E is giving E. So, as if this addition is done first and then it is multiplied by the second E. So which is, if we considered the presidents of this addition and multiplication; so, multiplication has got higher presidents than addition. So, naturally this is not a valid interpretation or of the statement, but as long as the presidencies are not mentioned. So, both of them are correct it ok.

So, the language designers I have they have they have to tell the presidents and as if compiler designer either you have to design the grammar such that this ambiguities are not there ok. Will, we can see shortly like if I have got that E T F grammar where I have got the production rules like E producing E plus T or T then T producing T star F or F and F producing id or within bracket E.

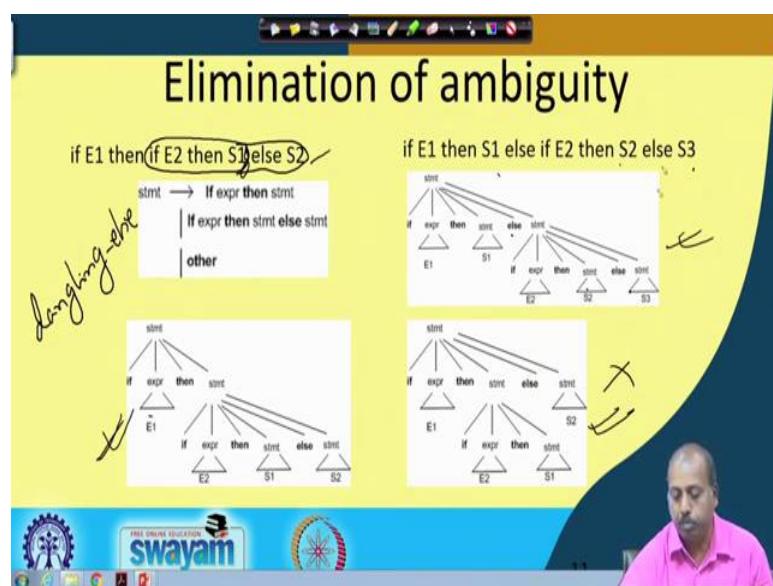
So, this particular grammar so, this is known as E T F grammar E T F grammar. So, this E T F grammar does not have this ambiguity E T F grammar does not have any ambiguity because it says that for reducing for applying this rule first this t has to be derived. So, this E plus so and the T we will take care of this multiplication first. So, addition cannot be done earlier than the multiplication. So, this will be more clear when we go to the code generation chapter, but this we can just take if view that this E T F

grammar cannot be cannot be ambiguous and. In fact, you will not be able to derive this parse tree using E T F grammar.

For example if we use E T F grammar then you will always have you always have do it like this E plus T then this E giving T giving F giving id and then this T giving T star F and this giving F and this giving id and this F giving id. So, this is the only parse tree that you can have. So, you cannot have any other parse tree for this E T F grammar, on the leftmost derivation. So, it does not have any ambiguity whereas, the this particular grammar where this T and F non terminals are not there so they are known as E grammar or expression grammar and this is ambiguous.

But many times this ambiguity is kept because in this case I have three non terminal symbols needed for writing the grammar et and f and here there is a single non terminal and you will see that the size of the parser. So, it will vary it will be more if there are more number of non terminals. So, though ambiguity is a difficult, but it is a problem for this parsing process. So, many a times you will allow this ambiguity so that the size of the parser is less compared to the situation where you have got a totally unambiguous grammar. So, you will see as we as we proceed through this chapter in the successive portions of this course.

(Refer Slide Time: 21:22)



So, this is another source of ambiguity like say this if then else statement. So, suppose we have got this statement if this rule like it says that a statement is like this, that if E 1

then if E 2 then S 1 else S 2 ok. So, the corresponding grammar is this one. So, it says that a statement can produce if expression then statement or if expression then statement else statement or other statement. So, this is where this is other actually it encompasses all other statements that may be there. So, we are not bothered about other statements. So, we have just kept it as other. Now suppose we have got suppose we have got this thing that if E 1 then if E 2 then S 1 else S 2 now. So, this one can be derived like this. So, this if E 1 so this is the expression.

So, it follows the first rule if expression then statement, then the expression will be giving me E 1 and this statement is again broken down into if expression then statement else statement where this second expression part. So, this corresponds to this E 2 and this S 1 and S 2 so they are corresponding to then part and else part. So, another possible derivation of the same thing can be like this, say I have by start with statement now if expression. So, I take the second rule. So, for the first level of derivation I take the second rule and it derives like statement producing if expression then statement else statement and for this expression gives me E 1 that is fine. Now this statement gives me this part.

If E 2 then S 1. So, this part so it does it produce a like if expression then statement and E 2 and S 1 fine. So, so this is the situation. So, what has happened is that in this case, in the in the first case this in the first case this else S 2 part this else S 2 is the confusion ok. So, in the first case in this diagram so this has been taken with this particular if statement as you see if E 2 then S 1 else S 2 whereas, in this case this S 2 the else part has been taken with the outer if ok. So, it is it has been taken like this that as if I have got this whole thing as a as the then part ok.

If E 1 then this statement else S 2. So, if E one is false it will go to S 2 whereas, in this case if E 1 is false it will go to the next statement and if E 1 is true then it will come to check E 2 and then it accordingly it will go to S 1 or S 2. So, there is a the two different derivations are possible for the same statement, whenever you have got this type of nested else we have got this particular problem. What about this case like if E 1 then S 1 else if S 2 then S 2 if E 2 then S 2 else S 3. So, here I have got in the then part I have got E 1 and S 1 else and in the second part I have got if E 2 when S 2 else S 3.

So, here I do not have any confusion because there was no. So, the here the problem occurred because one of the, if statement had has the else part the other one does not have any else part. So, in the nested if where does this else go so that is the ambiguity, that is the problem. So, in one case in the first case we have taken else with the inner motive in the second case here. So, we have taken else with the outer motive if, but if there are too such else's. So, both the statements are if condition then statement else statement and nested like that then there is of course, no problem. So, this is always fine. So, it there is no confusion here, but this is the confusion and you remember that most of the programming languages they will tell that this else is always attached to the innermost if.

So, this is known as the problem of dangling else this is known as the problem of dangling else. So, so the most of the programming languages why which allows this type of nested if then else statement it will tell that the if it is a mention if it is written simply like this then this else S 2 is always part of the innermost if so it is actually like this the. So, the statement to be taken as if E 1 then this whole thing; so, this is this is valid, but this is not valid ok, but as far as the grammar is concerned both of them are valid ok.

So, we have to do something to the grammar. So, that it is taken care of. So, you do not have the problem of this type of dangling else. So, otherwise the parser we will find a confusion here and there are several ways by which it will report it and as if proceed through the portion you will see that there are different ways in which this will be reported by the parser I am sorry.

(Refer Slide Time: 27:08)

```
stmt → matched_stmt  
|  
open_stmt  
  
matched_stmt → If expr then matched_stmt else matched_stmt  
|  
other  
  
open_stmt → If expr then stmt  
|  
If expr then matched_stmt else open_stmt
```

So, next will be looking into how to eliminate this dangling else problem; so, it one possibility is that a statement appearing between a then and then else must be a matched statement. So, this is the thing that thus I we say that there are two types of statements, one is called a matched statement another is called an unmatched statement or open statement. So and man a matched statement so it can give me all other statements of the programming language, but as for when it comes to the if part sorry, when it comes to the if part then it is always if then else.

So, I do not have the if expression then statement as the producible from the matched statement. So, so if expression then statement can be produced only by the open statement. So, with the open statement can also produce if expression then statement else statement, but this first then it must be a matched statement. And so that this else becomes a part of this. If the else is there then it will become a part of this if it is not there so, it will be a matched statement so that if there is an else inside. So, it will be matching with the innermost if. So, that way this solves this dangling else problem.

Now, so if we do some derivation then there was. So, this in the previous diagram what will happen is that. So, this part; so this part can be produced only by matched statement. So, as a result I cannot have this type of structure. So, only this structure will be possible because this will be a matched statement this is open statement and this is the

matched statement and the matched statement we will have the else part with it as a result it will not be able to derive the first string.

(Refer Slide Time: 29:23)

- A grammar is left recursive if it has a non-terminal A such that there is a derivation  $A \Rightarrow A\alpha$
- Top down parsing methods can't handle left-recursive grammars
- A simple rule for direct left recursion elimination:
  - For a rule like:
    - $A \Rightarrow A\alpha \mid \beta$
  - We may replace it with
    - $A \Rightarrow \beta A'$
    - $A' \Rightarrow \alpha A' \mid \epsilon$

So, next will be looking into some work that talks about eliminating the left recursion. So, many a time a grammar has got rules such that starting with a non terminal you can derive a string which is  $A A$ . So, which is which is having again that-non terminal as the left most symbol. For example, if I have a grammar rule like this say S producing say A a and there is a rule A which says that it can give me S b, then what will happen if you apply this rules then this gives me A a and from this a is again replaced by S.

So, S b a, so what is happening; so, starting with A you have produce so, this whole thing is an is alpha. So, alpha is a string of terminals and non-terminals. So, the first character sorry this is not alpha. So, alpha is this part. So, this is giving me S ba. So, this ba part is alpha. So, as per this convention A producing A alpha. So, this S is giving me S alpha. So, that way the symbol the non-terminal symbol that is there on the left hand side is appearing again as the first symbol in the right hand side. So, as if in some sense it is you and you can understand this is the some type of recursion ok

That is the same non-terminal symbol appearing as the first symbol on the right hand. So, this is the parsing methods that we have so, parsing methods cannot handle this type of left recursive grammars because, per the top down top down parsing methods what they will do? It will start with S and it will try to derive a string from here say in after

sometime it comes to the situation where it is S ba. So, basically we it is back to this is S. So now, again so, it will fall into a loop here and it will try to do that again. It will try to again apply the same set of rules to replace S and that way it falls into an infinite loop.

So, if you are looking for parse the top down parsing strategies then this left recursion is a problem. So, there can be some modification, simple modification to the grammar that can convert a left recursive grammar to a non-left recursive grammar. So, the rule is like this suppose, we have got a rule like this  $A \rightarrow A\alpha$  or  $B\beta$  where  $\alpha$   $\beta$  are strings of terminals and the non-terminals. So, we can replace it by a producing  $B$   $\dashv$  and  $A \dashv$  producing  $\alpha$   $A \dashv$  or  $\epsilon$ .

(Refer Slide Time: 32:21)

## Elimination of left recursion

- A grammar is left recursive if it has a non-terminal A such that there is a derivation  $A \Rightarrow A\alpha$
- Top down parsing methods can't handle left-recursive grammars
- A simple rule for direct left recursion elimination:
  - For a rule like:  
 ~~$A \rightarrow A\alpha | \beta$~~   $\rightarrow$   $A \rightarrow Ad \rightarrow Ad\alpha \rightarrow Ad\alpha d \rightarrow \dots$   
 $A \rightarrow \beta A' \rightarrow \beta A'\alpha \rightarrow \beta A'\alpha d \rightarrow \dots$
  - We may replace it with
    - $A \rightarrow \beta A'$
    - $A' \rightarrow \alpha A' | \epsilon$

So, you see that whatever strings that can be derived from the first grammar that is this rule can also be derived from this rule like here. So, you can see that A can produce so, all the strings that this A can produce.

So, A can be applying it can go on producing A alpha. So, it can do it like this A alpha alpha A alpha alpha alpha alpha. So, like that so, it can do it like this and ultimately this alpha has to be replaced by beta. So, it will finally, give a string which is beta then alpha alpha whatever. So, you see that using the second rule also so, you can do like this. So, you start with A first we do beta than A dash and this A dash can give me beta alpha A dash and then again this A dash can give me beta alpha alpha A dash. So, I just go on applying

this rule and finally, this A dash can be replaced by this epsilon. So, that you have got the beta alpha alpha alpha like that ok.

So, whatever string is derivable from this rule is also derivable from this set of rules, but the advantage that we have is the second set of rules it does not have any left recursion in it ok. Unlike the first rule where there was a left recursion, the same non terminal was appearing as the first symbol on the right hand side. So, it does not happen in this case. So, here it is simply that situation does not occur. So, the top down parser it can take a decision whether it should follow rule number 1 or rule number 2 that is whether to follow this rule or this rule for A dash.

So it will see that if the remaining part of the string it starts with alpha then it will follow this rule, if it finds at the end of the string then it will apply the second rule a dash producing epsilon to resolve the issue. So that way the top down parser will not get stuck at that point, it will be able to progress. So, this way a left recursion elimination is a very fundamental step whenever we are going to discuss about top down parsing strategy. So this is going to be one of the most important thing that we should do, you should scan through the grammar and find out where this is the left recursions are there and you eliminate all those left recursions from the grammar.

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

**Lecture – 19**  
**Parser (Contd.)**

In our last class, we have been discussing on left recursion. And left recursion is a problem particularly for top down parsers because top down parsers the way they work is that whenever it takes a decision, how to expand in non-terminal; it looks on the right hand side of the non terminal. So whichever terminal symbols comes at the beginning, so based on that among the alternatives of a non terminal so it will select one option.

(Refer Slide Time: 00:43)

The slide has a yellow header bar with the title "Elimination of left recursion". Below the title is a bulleted list:

- A grammar is left recursive if it has a non-terminal A such that there is a derivation  $A \Rightarrow A\alpha$
- Top down parsing methods can't handle left-recursive grammars
- A simple rule for direct left recursion elimination:
  - For a rule like:
    - $A \Rightarrow A\alpha | \beta$
  - We may replace it with
    - $A \Rightarrow \beta A'$
    - $A' \Rightarrow \alpha A' | \epsilon$

Below the list is a diagram showing the transformation of a production rule. On the left, a production  $A \Rightarrow A\alpha | \beta$  is shown with a curved arrow pointing from the first part to the second part. An annotation says "We may replace it with". To the right, the transformed rules  $A \Rightarrow \beta A'$  and  $A' \Rightarrow \alpha A' | \epsilon$  are shown with arrows indicating their relationship. The professor is visible in the bottom right corner of the slide frame.

For example; if we look into this particular production, A producing A alpha or A producing A alpha or beta where alpha, so here what happens is that this A again appearing on the right hand side for the first production. So it cannot take a decision whether to proceed with a producing A alpha whether to if in a string I have got say somewhere in the string A appears. So, in the next level, so whether this should be replaced by A alpha or not. So, that is not known because this will go on in a recursive fashion because the after this A alpha it will be replaced by A alpha, alpha and it will go on like this without consuming any further input.

So, it will be expanding like that. So, that creates the difficulty. So, what is done is that we have to resolve this left recursion and in our last class, we have seen that we can replace this particular production by means of a pair of production; A producing beta A dash where A dash is a new non terminal that is introduced into the said v n and A dash produces alpha A dash or epsilon.

So, we have seen that these two are equivalent. So, this single rule and these pair these pair of rule involving A and A dash they are equivalent. So, today we will first look into an algorithm by which it will be; by which will be able to solve resolve this left recursion from a grammar.

(Refer Slide Time: 02:11)

**Left recursion elimination (cont.)**

- There are cases like following
  - $S \rightarrow Aa \mid b$
  - $A \rightarrow Ac \mid Sd \mid \epsilon$
- Left recursion elimination algorithm:
  - Arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
  - For (each  $i$  from 1 to  $n$ ) {
    - For (each  $j$  from 1 to  $i-1$ ) {
      - Replace each production of the form  $A_i \rightarrow A_j \gamma$  by the production  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$  where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all current  $A_j$  productions
  - Eliminate left recursion among the  $A_i$ -productions

So, we will be able to remove this left recursion completely. So actually problem the type of left recursion that we have seen in the previous slides. So, here this is known as immediate left recursion.

(Refer Slide Time: 02:33)

The slide has a yellow header and a blue footer. The title 'Elimination of left recursion' is at the top. The main content is a bulleted list:

- A grammar is left recursive if it has a non-terminal A such that there is a derivation  $A \Rightarrow A\alpha$
- Top down parsing methods can't handle left-recursive grammars
- A simple rule for direct left recursion elimination:
  - For a rule like:
    - $A \Rightarrow A\alpha | \beta$
  - We may replace it with
    - $A \Rightarrow \beta A'$
    - $A' \Rightarrow \alpha A' | \epsilon$

Handwritten note on the slide:

$A \Rightarrow A\alpha$   
↓  
Immediate Left Recursion

The footer features the Swayam logo and the text 'FREE ONLINE EDUCATION'.

So, this A producing A alpha this the A producing A alpha so, this is known as immediate left recursion because it is appearing immediately. So, in the immediate right hand side of A, so this is appearing, but it may be the case that it is it does not happen immediately, but when it goes via a number of non terminals or number of rules, it jointly they create this left recursion. So, we will see one example for that. So, here you see that S producing Aa or B and A producing Ac or Sd or epsilon. So, here this, so, this left recursion this A, S producing Aa. So, there is no immediate left recursion here, but after that. So, I can do an expansion like this that S produces Aa and after that I can apply this rule a producing Sd, so, giving Sda.

So, effectively the left recursion has got re introduced into the set of rules. So, not only immediate left recursion, so immediate left recursion can be resolved using the strategy that we have seen in the previous slide, but this type of; this type of collective or non immediate left recursions. So, they also need to be eliminated. So, for that purpose, there is an algorithm that we can frame. So, this is for; this is for left recursion elimination.

(Refer Slide Time: 04:17)

So what it does is that it first arranges the non terminals in some sequence A 1, A 2, A n. So in this particular case, we have got two non terminals S and A. So, you can order them arbitrarily. So, there is no preference of ordering so, somebody may order like this, somebody may order like this but it does not matter whatever way we ordered it.

Then it says for i equal to 1 to n, for j equal to 1 to i minus 1 so, first i equal to 1. So, it will be picking up the first production rule and if there is a there is a production like a producing A j gamma. So, will be whenever we have got a producing A j gamma will be replacing it by the production A j producing delta one gamma delta 2 gamma etcetera where A j produces delta 1, delta 2, delta k.

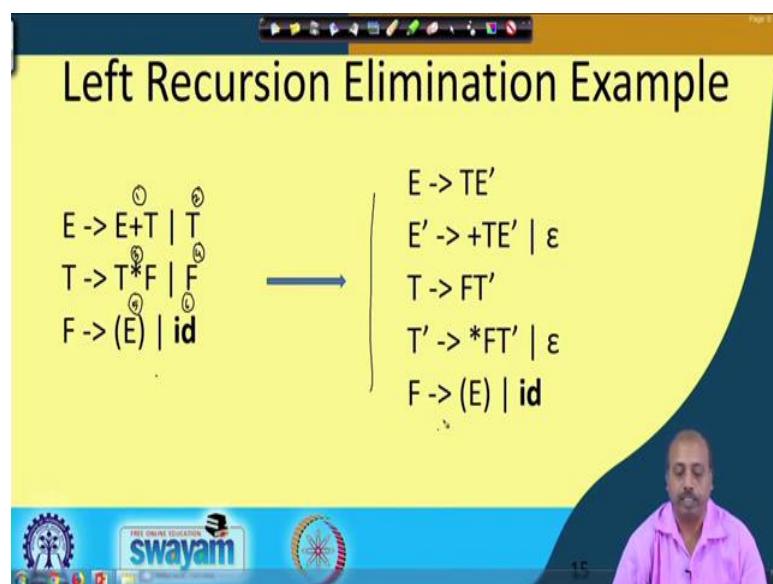
So, after that, so essentially what we are doing we are replacing this A j here by this delta 1, delta 2 etcetera. So, this process may introduce some immediate left recursion. So, after doing, this process may immediately introduce immediate left recursion like here we have got say S producing Aa and there is a production rule which says A producing S d

where this Aa Aa. So, this is our Aj so, this with respect to this particular rule, so, this capital A is our Aj and our delta we have got only delta 1, delta 1 is S d.

So, if I follow this rule, so what it will happen is that. So, S producing A j will be replaced by S d and gamma is A ok. So, as far as this example is concerned gamma is A. So, it will be replaced like this. So, it will bring in immediate left recursion. So, if there is any in the set of rules then it will eliminate this left recursion by the technique that we have seen previously as we have seen in the last slide.

So, that way of for all the; for all the production rules, so it will eliminate the left recursion. So, that, the resulting grammar it will not have any left recursion as for as complexity of this algorithm is concerned. So, this is you can understand that is be go n square time algorithm because where n is the number of the production rules that we have in the system. So, let us take an example and try to see how this left recursion elimination can work.

(Refer Slide Time: 07:29)



So, this is the standard ETF grammar; E producing E plus T or T, T producing T star F or F and F producing within bracket E id. So, if we number the rules, so this is my E producing E plus T. So, this is rule number 1, E producing T is rule number 2, T producing T star F is rule number 3, T producing F is rule number 4 and we have got this as rule number 5 and this as rule number 6.

Now, you see we have got here E producing E plus T. So, so here, so this will be this is having a immediate left recursion. So, this will be replaced by this set of rules E producing T E dash and then E dash producing plus T E dash or epsilon. So, this is by immediate left recursion. Similarly we have got T producing T star F. So, T star F again this is having and immediate left recursion so, that will that way it will be release. So, in this particular grammar, so we do not have any non immediate left recursion. So, all the left recursions that we have they are all immediate in nature.

So, we do not have any; we have we do not have any such production rule which will introduced some new left recursion by the by means of substitution. So, here everything is immediate so, and this right hand side, so here all the rules we have eliminated the left recursions from the immediate case and as we have seen previously also like this grammar and this grammar they can generate parser tree for the same expressions for the they actually represents the same language. And as for as the left recursion is concerned, so we have seen that they are also doing the same thing the grammar will the power of the grammar does not change or the set of language is accepted by the grammar does not change by doing this left recursion elimination ok.

(Refer Slide Time: 09:39)

**Left factoring**

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.
- Consider following grammar:
  - Stmt  $\rightarrow$  if expr then stmt ~~else stmt~~ ✓
  - | if expr then stmt ✓
- On seeing input if it is not clear for the parser which production to use
- We can easily perform left factoring:
  - If we have  $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$  then we replace it with
    - $A \rightarrow \alpha A'$
    - $A' \rightarrow \beta_1 \mid \beta_2$

So, next we will see another important issue which is known as left factoring.

So, left factoring is another problem like if we have got a situation like say this or we have got a rule where statement produces; if expression, then statement; else statement or

if expression, then statement now when you have. So, in your input sequence suppose I have got the token if.

The lexical analyzer has return to the parser that the next token is if. Now, the parser has to take a decision whether it should follow this rule or this rule. So, if it expands via this rule; that means, it is assumed that in future else we will definitely be there, but if this turns out to be a normal if then statement it does not have any else part. So, later on the parser will be in a problem. So, it will not be able to parse that particular statement.

On the other hand, if it assume the parser assumes that the statement will be if expression then statement, then we have the difficulty that if actual statement is if then else statement, then later on when that else token will be written by the lexical analyzer to the parser. So, parser will not be able to accommodate that. So, in any way, so since we are going in a top down fashion. So, all this left factoring left recursion this problems occur for the top down parsers because, top down parser have to has to take a decision looking on the next token that it has receipt which rule to follow.

So, getting the token if the parser cannot decide whether it should follow the if then else rule or the if then rule. So, this is another important problem and this has to be resolved. So, left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top down parser parsing so, this is an example. So, on seeing input if it is not clear for the parser which production to use. So, what is the way out? Way out is that we take up to this much. So, up to statement part so we take it as a proper rules and this else statement part will take as another rule so that the parser up to this much a single rule will follow and after this if else comes. So, a new rule will be used it is not that I have to take a decision looking at the first if itself.

So, or so, this is the example like if we have this alpha beta 1 or alpha beta 2. So, far as per as this example is concerned this alpha is if expression, then statement beta 1 is else statement and beta 2 is epsilon. So, what we do, we replace it by a producing alpha a dash ok. So, up to whichever is common so, take it take it out. So, alpha A dash and A dash producing produces beta 1 or beta 2. So, that is the obvious thing that we can do. So, in this particular case, so we can write it like this.

(Refer Slide Time: 12:41)

The slide has a yellow header with the title "Left factoring". The main content is a bulleted list:

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.
- Consider following grammar:
  - Stmt -> if expr then stmt else stmt
  - | if expr then stmt
- On seeing input if it is not clear for the parser which production to use
- We can easily perform left factoring:
  - If we have  $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$  then we replace it with
    - $A \rightarrow \alpha A'$
    - $A' \rightarrow \beta_1 \mid \beta_2$

On the right side of the slide, there is a handwritten note:

$\Rightarrow \begin{matrix} \text{stmt} \rightarrow \text{if expr then stmt} \\ \text{stmt} \rightarrow \text{else} \end{matrix}$

At the bottom of the slide, there is a video player interface showing a person speaking. The Swayam logo is visible at the bottom left.

Statement produces if expression; if expression, then statement and then statement dash where statement dash produces else statement or epsilon ok so, do it like that. So, one thing one notation that will follow throughout these lectures is that whenever we are writing in say in type font like say whenever writing in the slides. So, wherever we have got a terminal symbol we have tried to put it in boldface like this to distinguish it from the non terminal symbols.

Like here expression is a non terminal, but if is a terminal similarly statement is a non terminal. So, we will try to write this if then else this tokens in bold face and other terms like expression statement etcetera which are non terminal symbols, they will try to write in normal font. So, whenever the it is whenever there is a scope of confusion. So, if there is no scope of confusion, then of course, any font can be used. Similarly, when writing in by our hand. So, you will underline the tokens like this if then else. So, these are tokens so, we will underline the tokens by to distinguish them from non terminals we will underline the terminals to distinguish them from non terminals wherever it is confusing say if it is not confusing, if it is straight forward, then we will follow the normal convection ok.

So, the rule followed is you take out this alpha part separately the alpha A dash and then A dash produces beta 1 or beta 2. So, you can do it like this so, that is left factoring.

(Refer Slide Time: 14:43)

**Left factoring (cont.)**

- Algorithm
  - For each non-terminal A, find the longest prefix  $\alpha$  common to two or more of its alternatives. If  $\alpha \neq \epsilon$ , then replace all of A-productions  $A \rightarrow \alpha \beta_1 | \alpha \beta_2 | \dots | \alpha \beta_n | \gamma$  by
    - $A \rightarrow \alpha A' | \gamma$
    - $A' \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$
- Example:
  - $S \rightarrow iEtS | iEtSeS | a$
  - $E \rightarrow b$
- Modifies to
  - $S \rightarrow iEtSS' | a$
  - $S' \rightarrow eS | \epsilon$
  - $E \rightarrow b$

So, this is the general algorithm for left factoring for each non terminal A, find the longest prefix alpha common to 2 or more of its alternatives. If alpha not equal to epsilon, then replace all A productions by A producing alpha beta 1, alpha beta 2 up to alpha beta n by this one; A producing alpha A dash or gamma and A dash produces beta 1 or beta 2 or beta n.

So, this simple rules takeout the longest prefix between a number of; between a number of alternatives and take them out. So, this is that if then else gamma that is written in short. So, S produces if expression then statement or if expression, then statement else statement or a and E produces b.

So, where a b these are terminals indicating some expression and all. So, it that is not a concerned here, so here this there is a left factoring of this parts i E T S. So, this part is common between this first rule and the second rule. So, this grammar can be modified to something like this S produces if expression then statement dash S dash or a where S dash produces e S that is else S or epsilon and E producing b. So, this part remains as it is.

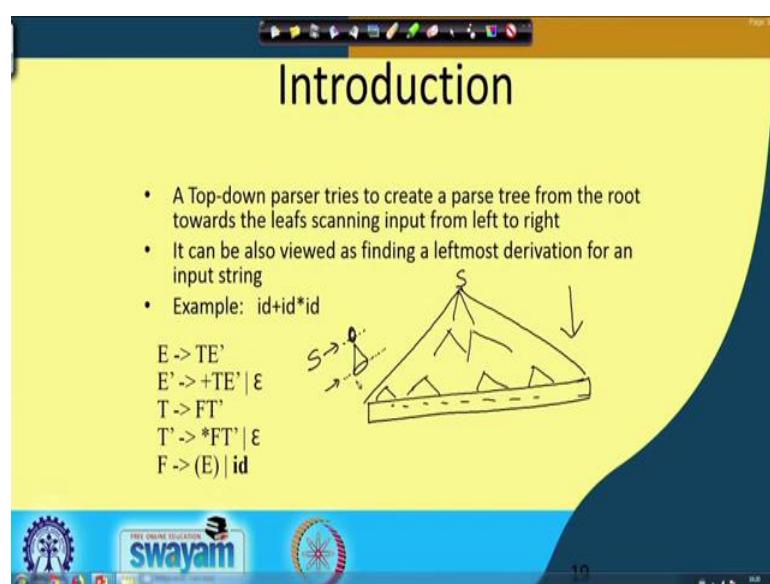
So, this is the way we will do this left factoring. So, that the grammar becomes suitable for top down parsing or predictive parsing.

(Refer Slide Time: 16:23)



So, next we will be looking into one of the parsing strategies known as top down parsing. So, top down parsing it will try to derive the whole stream whole given stream starting with the start symbol of the grammar and in the process it will generate the parse stream and that parse stream may be used for other purposes, but it will be it will try to come from the top it is starting with the start symbol of the grammar, it will try to derive the whole stream.

(Refer Slide Time: 16:55)



So, how it will do? So, we will see that. So, a top down parser it tries to create a parse tree from the root towards the leaves scanning in foot from left to right. So, if this is the so, you can conceptually view it like this if this is your input stream; if this is the input stream that you have. So, it will be ultimately covered by tree whose root will be the start symbol and it will be going like this.

This entire tree will get covered and ultimately, at the lowest level you will have this thing and this is done in a top down fashion ok. So, that is a top down parser. So, it can also be viewed as finding left most derivation for an input string. So, this perform say left most derivation so, it replaces the left most non terminal symbol by means of a terminal symbol. So, it is by means of a corresponding rules so, it starts with S, the starts symbol then uses some rule to produce some go towards the input string. Then wherever is the first non terminal symbols, suppose this is the first non terminal symbol, so in the next line next state so, this thing will get expanded. So, this part will remains. So, this will get expanded to something like this and this will be there then again it will find out the left most non terminal and it will expand it.

So, it will it does a left most derivation of the input. So, suppose I have got this id plus id star id. So, in a leftmost derivation how will it be done?

(Refer Slide Time: 18:37)

**Introduction**

- A Top-down parser tries to create a parse tree from the root towards the leafs scanning input from left to right
- It can be also viewed as finding a leftmost derivation for an input string
- Example:  $\text{id} + \text{id}^* \text{id}$

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

E  $\rightarrow$   $\epsilon$   $\rightarrow$  FT'  $\rightarrow$   $\epsilon$  FT'  $\rightarrow$  idFT'  $\rightarrow$  id+idFT'  $\rightarrow$  ...

So, it will be starting with E as the start symbol, then there is only one rule that involves E. So, it is T E dash ok. So, T E dash, so we have got, so next terminals in the next

production; so I have to replace this T because this is the leftmost non terminal. So, this T will be replaced by F T dash. So, it is because F T dash E dash and in the next rule this F will be replaced and F will be replaced by id so, it is id T dash E dash.

Then this next this T dash has to be replaced then this T dash has to be replaced by this rule T dash producing star F T dash. So, this is id star T dash E dash so, it will go like this and ultimately, it will derive this thing. So, that is. So, we will see some examples again but it will go it will proceed like this and do the whole derivation.

(Refer Slide Time: 20:07)

- A Top-down parser tries to create a parse tree from the root towards the leafs scanning input from left to right
- It can be also viewed as finding a leftmost derivation for an input string
- Example:  $\text{id}+\text{id}*\text{id}$

$E \rightarrow TE'$   
 $E' \rightarrow +TE' \mid \epsilon$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT' \mid \epsilon$   
 $F \rightarrow (E) \mid \text{id}$

```

    E   ⇒ E   ⇒ E   ⇒ E   ⇒ E   ⇒ E   ⇒ E
      \|     \|     \|     \|     \|     \|     \
    T   E'   T   E'   T   E'   T   E'   T   E'   T   E'
      |       |       |       |       |       |       |
    F   T   F   T'   F   T'   F   T'   F   T'   F   T'   F
      |       |       |       |       |       |       |
    id   id   id   ε   id   id   ε   id   ε
  
```

So, this shows the derivation E producing in a left most derivation, T E dash then in the next step this T is replaced by F T dash and E dash in the next step, this F is replaced by id and this is T dash, then the in the next step this T dash is; T dash is producing epsilon. So, that is it is replaced by epsilon and then this T produces so, this part is done now this it goes to this E in this E it this E dash is expanded by plus T E dash and then it will go on like that. So, after that it is not shown here.

So, this plus T up to we have derived up to this id and this is plus T E dash. So, it will be going on like that producing the whole string. The rest of the derivations are not shown here ok.

(Refer Slide Time: 21:05)

Recursive descent parsing

- Consists of a set of procedures, one for each nonterminal
- Execution begins with the procedure for start symbol
- A typical procedure for a non-terminal

```
void A() {  
    choose an A-production, A->X1X2...Xk  
    for (i = 1 to k) {  
        if (Xi is a nonterminal  
            call procedure Xi();  
        else if (Xi equals the current input symbol a)  
            advance the input to the next symbol;  
        else /* an error has occurred */  
    }  
}
```

A  $\rightarrow x_1x_2x_3\dots x_k$   
 $y_1y_2y_3\dots y_m$

So, there are two top down parsers that will look into. The first one in the category is known as recursive descent parsing. So, in a recursive descent parsing it consists of a setup procedure. So, as the name suggests, so it is a recursive parser. So, it is a collection of a recursive routine and it will try to recursively descent. As I said, that it starts with a start symbol of the grammar and tries to derive the remaining strings the final string. So, it actually in some sense it descents from the start symbol towards the final string. So, that is why it is called a descent parsing and it is recursive descent parsing because it does a recursively by means of procedures which are recursive in nature.

So, it consists of a set of procedures one for each non terminal. So, for every non terminal, we have got a procedure. Execution begins with the procedure for start symbol so, whatever be the start symbol so, with that there execution will begin. So, from the main routine you call give a call to the procedure corresponding to start symbol of the grammar, a typical procedure will look like this.

Suppose, A is a non terminal symbol in the grammar, so, A is a non terminal symbol in the grammar. So, for that we have a corresponding procedure A and no parameter needs to be passed here so, because it is and it does not return any value also. Now, this a may have several alternatives like my production rule may be A producing x 1, x 2, x 3, x k or y 1, y 2, y 3, y, y m. So, like that there may be several alternatives.

So, what this parser is trying to do it is try, it will first try to see whether using this rule it can reach to the final string or not. If it fails, then it will try with the alternatives next alternative. If it fails, then it will try with the next alternatives. So, that way if all the alternatives are exhausted and the final string cannot be derived then there is an error in the input. So, input the syntactically wrong input is syntactically wrong otherwise it will say that it has got a derivation for the input string starting with the start symbol of the grammar. So, it will do like this. So, it will for a for a particular non terminal A it will select A production rule A producing X 1, X 2 up to X k and for i equal to 1 to k, if X i is a non-terminal, it will call the corresponding procedure X i.

(Refer Slide Time: 23:57)

**Recursive descent parsing**

- Consists of a set of procedures, one for each nonterminal
- Execution begins with the procedure for start symbol
- A typical procedure for a non-terminal

```
void A0 {
    choose an A-production, A->X1X2..Xk
    for (i = 1 to k) {
        if (Xi is a nonterminal
            call procedure Xi();
        else if (Xi equals the current input symbol a)
            advance the input to the next symbol;
        else /* an error has occurred */
    }
}
```

The slide also features a hand-drawn diagram on the right side showing a stack of recursive calls. It shows a stack frame for 'E' with a pointer to 'T E''. Inside 'T E'' is another stack frame for 'E' with a pointer to 'T E''. This illustrates how multiple levels of recursion are managed.

So, as I said that say this particular rule, so, E producing say T E dash for example, so this is a rule. So, what it what will happen is the in the procedure E, it will first call the procedure for T, then it will call the procedure for E dash so, this is the whole routine. So, that is what is written here for if X i is a non-terminal like T is a non-terminal here, then call the procedure X i. So, it will call the procedure T.

When it returns from the procedure T, then it will come to this point. Now, I advances to the next point so, now, it is E dash so, it will call the procedure E dash. So, that is for the non terminals. So, if X i happens to be a terminal; for example, if I have got a rule like E producing say plus T E dash in that case, it will first this X 1 happens to be a non-terminal terminal symbol plus, then it will check with the current input ok.

The lexical analysis tool, so the lexical analyzer will be informed by the parser to get the next token. So, if the lexical analyzer also finds a plus at the input point, then it will return the token plus and the parser will see that these two are matching ok. So, if it is a terminal, else if  $X_i$  is a terminal symbol the input symbol a, then advance input to the next symbol. So, if  $X_i$  equals the current input symbol a so, this is also plus this is also plus, then we advance the input pointer to the next point.

So, now, so naturally now, it will be call now it comes to this loop again and as a result it now it will be calling the routine for T. So, it will go like that otherwise there is an error. So, there is an error that has occurred so, it will be flashing that error that there is a syntax error.

So we are not showing it for explicitly for all the rules. So this will go for all the rules. So this is a generic way we have written it. So it will be there for each and every non terminal that you have in the grammar. So you see that this recursive this and parsing algorithm is very simple. So, you have to just have a collection of recursive routines and then those recursive routines. So they will be they will be collaborating with each other to see whether the input string can be derived from the start symbol of the grammar.

(Refer Slide Time: 26:39)

Recursive descent parsing (cont)

- General recursive descent may require backtracking
- The previous code needs to be modified to allow backtracking
- In general form it cannot choose an appropriate production easily.
- So we need to try all alternatives
- If one fails, the input pointer needs to be reset and another alternative has to be tried
- Recursive descent parsers cannot be used for left-recursive grammars

21

So general recursive descent parsing may required backtracking. So, this is one important issue as I was telling that I may have several alternatives like say A producing X 1, X 2 say E of.

(Refer Slide Time: 26:57)

The slide also features a handwritten diagram on the right side showing the derivation of a grammar rule  $E \rightarrow E + T \mid T$ . It shows the state  $E \rightarrow E + T \mid T$  at the top, followed by a downward arrow pointing to  $E \rightarrow E + T \mid T$ . Below this, another arrow points to  $E \rightarrow E + T \mid T$ , with a handwritten note '1/2' indicating it's the first step. Further down, another arrow points to  $E \rightarrow E + T \mid T$ , with a handwritten note '2/2' indicating it's the second step. This illustrates how the parser explores multiple derivations of the same non-terminal.

For example, I can have two rules like  $E$  producing  $E$  plus  $T$  or  $T$  or say if I if solve the immediate left recursion, so this will be giving rise to  $E$  producing  $T$   $E$  dash or epsilon. So,  $E$  producing  $T$   $E$  dash and sorry  $E$  producing plus  $T$   $E$  dash where  $E$  dash produces  $E$  or epsilon says something like this. Then I may have some alternative like if may say  $E$  dash for the rule for  $E$  dash. So, it may call the routine for  $E$  or it may be calling it may do nothing.

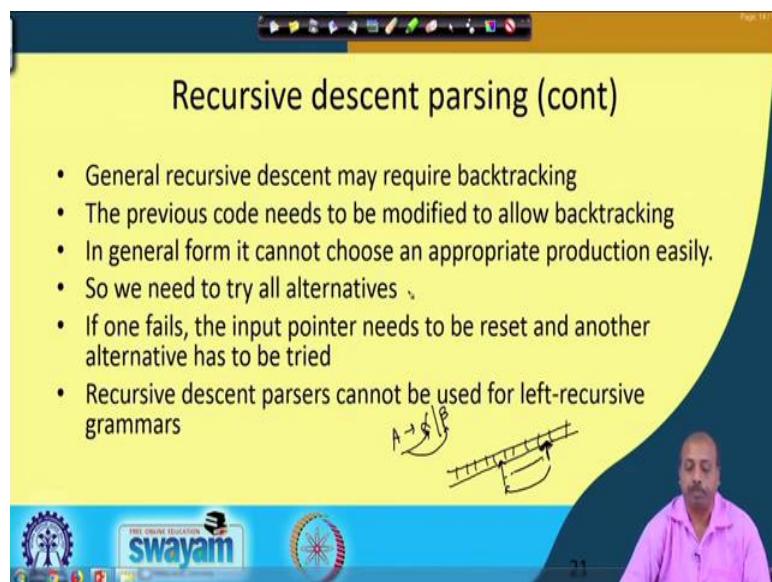
So that way I can have several alternatives. So if this alternative fails, then I will be go walking with the next alternatives. So that way I may have to have different recursive calls and one so one if one of the alternatives fail, then I have to call recursively the next alternative. So this is this is that is why it is recursive in nature but this recursiveness create some problem also because implementing recursive programs sometimes it is difficult. Now it takes more time, then a non recursive version. So we will see how to go to a non recursive version but first to understand the concept. So this recursive parsing, so that is very easy and given a grammar, so you can very easily make it.

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

**Lecture – 20**  
**Parser (Contd.)**

So, the recursive descent parsing technique that you are discussing; so there are some problems. The first problem is that the general recursive descent parsing so this requires backtracking.

(Refer Slide Time: 00:26)



So, the point is that as I said that there may be alternatives. So, I suppose I have got 2 alternatives A producing alpha is an alternative and beta is another alternative. So, first I will try with this rule A producing alpha now while doing this so, the input that we have so, input pointer was say. So this is my input sequence and the pointer was somewhere here, when I was when I started the procedure for A.

Now, while trying out this alternative alpha, this input pointer has advanced to some extent. So, it is gone to this and then suppose, we find that no A producing alpha is not the correct option to try out. So, it does not derive the final string. So, we need to come back and try the next alternative a producing beta, but how do we do this? Because by this time the lexical analyzer so it has advanced its input pointer already to this point.

So, it has to be taken back to this point before, we start this A producing beta alternative. So, this is difficult because, we have to take back the input pointer to come back to this point. So, that is what we you are telling it here, that the last code that we have seen for the procedure A.

So, that needs to be modified to allow backtracking. In general form, it cannot choose an appropriate production easily because, we may have this left recursion, left factoring and also though it may be a left recursive grammar or they it may require left factoring. So, if those are not done then I cannot make a top down parser strategy, recursive descent parsing strategy using that grammar.

So, we cannot choose an appropriate option very easily whether to try out alpha or beta. So, there are no immediate guidelines. So, even if this left factoring and left recursion elimination has been done still, it may be difficult to make a wise choice between alpha and beta. So, only when alpha fail so, we try out beta. So, exhaustively all the alternatives maybe may have to be tried out. So, you need to try all alternatives; if one fails the input pointer needs to be reset and another alternative has to be tried.

So, these I have just discussed. So, you need to take back the input pointer for the lexical analysis tool to the point from where, we have to started the procedure A. So, this is another issue and the recursive descent parsers they cannot be used for left recursive grammar. So, if the grammar has got left recursion then we cannot use this recursive descent parsing.

(Refer Slide Time: 03:12)

S->cAd  
A->ab | a

Input: cad

S  
c A d

S  
c A  
a b

Handwritten notes:

- S() { consume c }
- A() { consumed }
- A() { consume b } X
- A() { consume a }

So, we will see some solutions to this strategies, this drawbacks in our next slides, but before that we take an example suppose, we have got a set of production rules  $S$  producing  $c A d$  and  $A$  producing  $a b$  or  $a$ . So, you see that we do not have any left recursion here because, here it starts with a terminal symbol  $c$ , here also it starts with a terminal symbol  $a$ . So, there is no left recursion. Similarly, there is no left factoring is also required because, because we do not have any non-terminal big coming on the right hand side.

So, it is not that I have got this  $a$  then something that you can take out  $a$  and like that suppose the input that we have is  $c a d$ . So, the way it proceeds it first starts with first try out we start with the star symbol of the grammar and I do not have any option here there is only one rule  $c A d$ .

So, it expands it by  $c A d$ , then it tries out the first alternative  $S$  producing  $a b$ . So, if it does this  $S$  producing  $a b$  then if you see that it has derived the strings  $ca bd$ , which is not  $cad$ . So, it cannot be converted to  $cad$ . So, it will fail the parser this when it tries does this. So, it cannot go to the input string  $cad$ . So, it will not match. So, it will when procedure for  $A$  is called, it will first call the actually when the procedure for  $S$  is called. So, the procedure  $S$  is something like this.

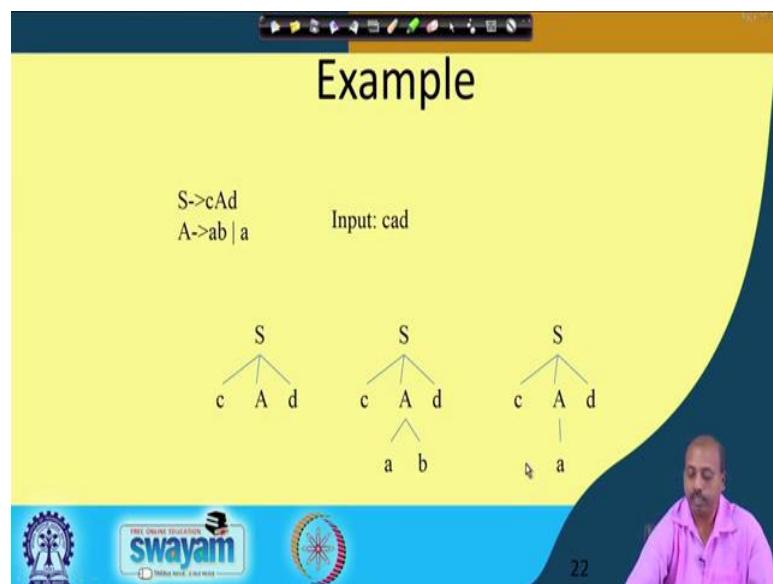
So, consume  $c$ . So, it is consume  $c$  then it will called procedure  $A$  then it will be consume  $d$ , it will be something like this and the procedure for  $A$  is something like this

there are 2 alternatives, first alternative is consume a and then consume b. So, this is one alternative and another alternative that we have is consume a this is a second alternative. So, first it will try out the first alternative for a. So, this will be successful because, when it is c a d initially the input pointer is here.

So, procedure for S is called so, it will consume c. So, input pointer will advance it will come to a now it calls the procedure A so, it comes here consumes a and then pointer is advanced. So, it tries to consume b, but it finds that there is a d there. So, there is a mismatch. So, it understands that this alternative is not valid for this particular derivation. So, input pointer is taken back by one position from where this procedure was called and then it will find the other alternative says consume a.

So, input pointer will be advanced and then A part is done. So, it will come back to this point and now it will say consume d from the procedure a. So, this will be consumed. So, that will be the third thing and then it. So, it can proceed like that.

(Refer Slide Time: 06:29)



So, it fails at this point. So, it tries out the other alternative A producing a and giving c A d.

(Refer Slide Time: 06:37)

Predictive parser

- It is a recursive-descent parser that needs no backtracking
- Suppose  $A \rightarrow A_1 | A_2 | \dots | A_n$
- If the non-terminal to be expanded next is 'A', then the choice of rule is made on the basis of the current input symbol 'a'.

23

So, this way this recursive descent parsing works, but sometimes we do not like this type of backtracking because, then we have got the difficulty, because we have to see like how much has to be taken back and all. So, this predictive parsing so, this is a recursive descent parsing that needs no backtracking. So, it will be a recursive descent parsing only, but without any backtracking. Suppose, I have got a rule A producing A 1, A 2, A n. So, if the non-terminal to be expanded next is A then the choice of the rule is made on the basis of the current input symbol. So, if we have got a number of suppose, these are the alternatives A1, A2, A n. So, these are the alternatives.

So, which alternative to follow so, based on the current input symbols. So, it will take a decision ok. So, if we can take the decision properly then we can make a predictive parser for the grammar. So, if we cannot make a decision for all the productions then predictive parser designed for the particular grammar is not possible.

(Refer Slide Time: 07:45)

## Procedure

- Make a **transition diagram** (like dfa/nfa) for every rule of the grammar.
- **Optimize** the dfa by reducing the number of states, yielding the final transition diagram
- To parse a string, **simulate** the string on the transition diagram
- If after consuming the input the transition diagram reaches an **accept state**, it is parsed.

24

So, in many cases it will be possible and in many cases it will not be possible. So, we will see the situations for both. So, procedure for technique for designing this long recursive version or predictive parsing mechanism is to make a transition diagram for every rule of the grammar like dfa or nfa deterministic finite automata, non deterministic finite automata. So, we make a transition diagram and then we try to optimize the transition diagrams by reducing number of states yielding the final set of diagram.

So, we first draw a set of diagrams based on some optimization of the diagram. So, we come to the final stage of diagram, whenever you are trying to parse a string, we simulate the string on the transition diagram. So, as if we are making transitions to the transition diagram using the inputs from the string, if we have consumed the input transition after consuming the input, when you are consuming the entire input if the transition diagram reaches an accept state then it is taken as parsed ok.

So, if we so, transition diagram like dfa and nfa it will have an initial state and a set of final states. So, if it reaches one of the final states after consuming all the inputs then you will say that the parser has accepted the input.

(Refer Slide Time: 09:17)

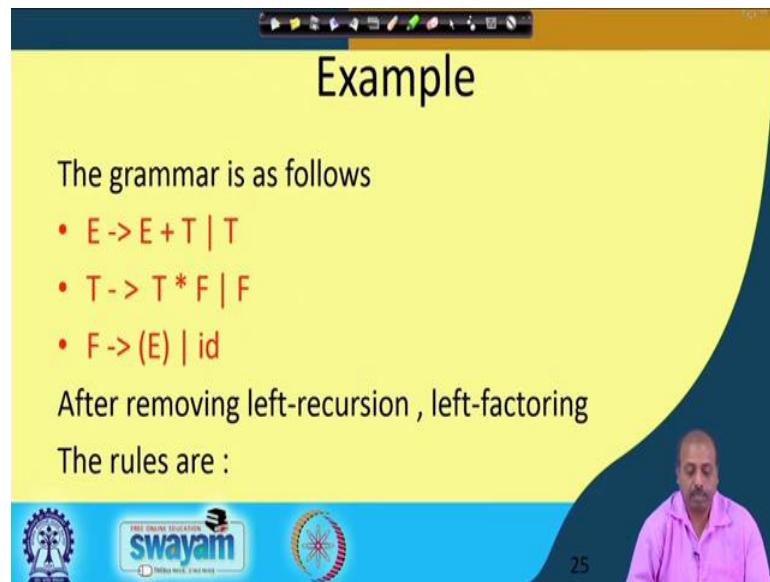
## Example

The grammar is as follows

- $E \rightarrow E + T \mid T$
- $T \rightarrow T^* F \mid F$
- $F \rightarrow (E) \mid id$

After removing left-recursion , left-factoring

The rules are :

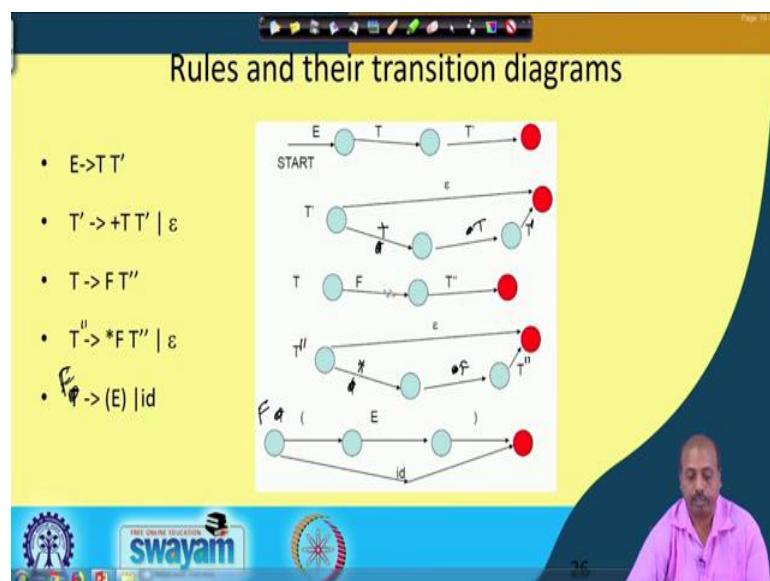


So, we will take some example. So, suppose we have again take that expression grammar E producing E plus T or T, T producing T star F for F F producing within bracket E or id then after removing this left recursion and left factoring the rules will be something like this E producing T T dash then T dash producing plus T T dash or epsilon F producing T producing FT double dash and T producing star FT double dash epsilon.

(Refer Slide Time: 09:33)

## Rules and their transition diagrams

- $E \rightarrow TT'$
- $T' \rightarrow +TT' \mid \epsilon$
- $T \rightarrow FT''$
- $T'' \rightarrow *FT'' \mid \epsilon$
- $F \rightarrow (E) \mid id$



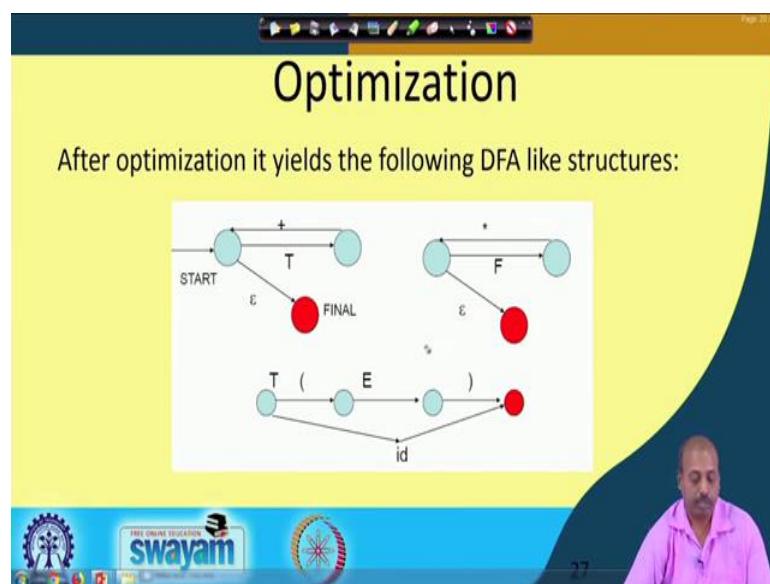
And then T producing within bracket E id now, so this is so, this is the for the first one. So, we have got a diagram like this E this is for E. So, produces it uses T and T dash. So,

on getting these 2 symbols, it will come to a final state similarly T dash. So, this is the red is the red is the final state. So, on getting a epsilon. So, it can come to final state or it can be it can be. So, this is our T. So, this is FT double dash then this is T dash.

So, T dash is actually this is something wrong. So, this should be plus this should be T and this should be T dash and then these T dash, this is FT double dash then, these T T dash producing this should be T double dash. So, this is T double dash producing. So, this should be T double dash producing a star FT double dash or epsilon and then this should be F producing within bracket E or id.

So, this is for F within bracket E or id. So, this is T dash. So, this should be plus this should be T and this is T dash then T producing FT double dash that is all right then this is for T double dash, T double dash producing epsilon and star this should be star FT double dash and F producing within bracket E and id. So, this is our final set of transition diagrams that we can have and then.

(Refer Slide Time: 12:50)



So, you can do some optimizations like starting with the star symbol. So on getting T, it comes to this state and so, this is the rule for this is the rule for T. So, that T is again coming back. So, you know coming back to the start state. So, if there is no expression after this. So, it can come to the final state or if there is a plus symbol so, it can come to this. Similarly, if for the for this f it can come to on getting epsilon, it can come here, but

if it gets F. So, it can come to this state and getting star, it can come back to this point and again proceed on epsilon to the next state. So, this can be done.

(Refer Slide Time: 13:37)

## SIMULATION METHOD

- Start from the start state
- If a terminal comes consume it, move to next state
- If a non-terminal comes go to the state of the "dfa" of the non-term and return on reaching the final state
- Return to the original "dfa" and continue parsing
- If on completion( reading input string completely), you reach a final state, string is successfully parsed.

28

So, how do you simulate it? We start with the start symbol of the set of diagram start with the start state and if it is a terminal, we consume it and move to the next state, if it is a non terminal. So, we go to the state of the dfa of the non terminal and return on reaching the final state, when we return the final state of that non-terminal transition diagram, we return to the start state and we return to the original dfa and continue parsing.

So, this way when we are whenever you reach the any final state so, we come back to the original dfa and continue parsing on completion of input when the input has been seen completely. So, if we find that we have reached a final state then the string is successfully parsed. So, otherwise it is not.

(Refer Slide Time: 14:30)

Disadvantage

- It is inherently a recursive parser, so it consumes a lot of memory as the stack grows.
- To remove this recursion, we use LL-parser, which uses a table for lookup.

leftmost derivation      left-to-right scan

So, the problem that we have is that inherently it is a recursive parser. So, it consumes lot of memory as the stack grows. So, we are calling the routines again and again. So, as a result we are doing it by means of transition diagram, but it is taking lot of time it may take lot of time, because of this recursive nature and to remove this recursion, we can use one type of parser, which is known as LL parser, which will use a table for the lookup procedure.

So, this LL so, these 2 L stand for the first L stands for leftmost derivation and this is the this is stands for left to right scan of the input, this will do a left to right scan and it will produce a left most derivation for the parsing.

(Refer Slide Time: 15:41)

The slide has a yellow header bar with the title "First and Follow". Below the title is a bulleted list of definitions:

- $\text{First}(\alpha)$  is set of terminals that begins strings derived from  $\alpha$
- If  $\alpha \Rightarrow^* \epsilon$  then  $\epsilon$  is also in  $\text{First}(\alpha)$
- In predictive parsing when we have  $A \rightarrow \alpha \mid \beta$ , if  $\text{First}(\alpha)$  and  $\text{First}(\beta)$  are disjoint sets then we can select appropriate A-production by looking at the next input
- $\text{Follow}(A)$ , for any nonterminal A, is set of terminals a that can appear immediately after A in some sentential form
  - If we have  $S \Rightarrow^* (\underline{\alpha} A \underline{\beta})$  for some  $\alpha$  and  $\beta$  then  $\beta$  is in  $\text{Follow}(A)$
  - If A can be the rightmost symbol in some sentential form, then  $\$$  is in  $\text{Follow}(A)$

Below the list are two hand-drawn diagrams:

- The first diagram shows a derivation tree starting from a root node S. The path from S to a terminal node is labeled  $S \Rightarrow^* (\dots \Rightarrow^* (\dots \Rightarrow^* (\dots \Rightarrow^* \text{terminal}))$ . Dashed arrows point to nodes labeled  $A \Rightarrow^* \alpha$  and  $A \Rightarrow^* \beta$ , with the text  $\text{Follow}(A) = \{\alpha, \beta, \dots\}$  written below.
- The second diagram shows a horizontal line with several symbols: a circle, a square, a triangle, a rectangle, and a diamond. Annotations include  $A \in \text{First}(A)$ ,  $A \in \text{First}(\beta)$ , and  $A \in \text{Follow}(A)$ .

So, we will see how it can be design, but to come to that. So, we need to have some definition one is known as first set, another is known as follow set. So, these 2 definitions we have to learn by heart because, many times in our compiler course. So, will come to these 2 sets first and follow and we should be we should be able to compute them very confidently and particularly this follow set computation is slightly tricky. So, we have to be careful there ok. Let us try to understand what does it mean. So, first of alpha is a set of terminals that begins strings derived from alpha, as you know that alpha is a any string of terminals and non terminals.

So, if it is whatever the string be. So, from starting with this alpha, whatever string after whatever strings you can derive ok. So, in that for whichever symbol appears as the first symbol; so, that is the thus that is particular terminal symbol so, will be included in the first set. So, if alpha in 1 or 0 or more derivations give epsilon in epsilon is also in the first of alpha ok. Now, if I have got so, how does it, how is it going to help? It is going to help because, if I have got 2 productions, A producing alpha and A producing beta and at present suppose, I am looking in the current input symbol is A.

So, based on this we will be able to take a decision like which rule to follow. So, we look into the 2 sets first of alpha and first of beta. Now if the symbol A belongs to the set first of alpha and A does not belong to first of beta. So, in that case there is there is no point trying out the A producing beta alternative for this particular string.

So, we should try with A this A producing alpha type of rule. So, this way it will be helping us in predicting like which rule to follow. So, in predictive parsing whenever we have got A producing alpha or beta, if first of alpha and first of beta are disjoint sets then we can select appropriate a production by looking at the next input ok. So, this way it is going to help. Another definition is follow of A, so where A is a non terminal symbol. So, follow of a is a set of terminals A that can appear immediately after a in some sentential form.

So, sentential form means anything that you can derive with the start symbol of the grammar. Suppose, S is the start symbol of the grammar and you are applying some rules and. So, ultimately this rule leads to set of terminals. So, this is the final input string now, all the this intermediary things that you have so they will be called sentential forms because, they are actually derivable from S and will lead finally, to sum a string of the language.

So, any string of any string of symbols terminals and non terminals that can finally, lead to the finally, lead to some input string or involving terminals only and this is derivable from the start symbol of the grammar then that mix of mixed string of terminals and non terminals. So, there will be it will be called a sentential form so, and in that sentential form.

Suppose, I have got a sentential form here, I have got the symbol A and after that suppose this small a character, the terminals small a appears then we say that small a belongs to the follow of A. In another sentential form maybe, this A is followed by may be followed by B. So, B is also in the follow set of A. So, A can be followed by A or B. So, the so, the follow set of A will have these 2 symbols a b, some more may be there, but these 2 must be there.

So, if we have starting with S in 0 or more derivation steps. So, if you come to this type of sentential form alpha A A beta and the for some alpha beta then a is the follow of A. So, this particular small a is in follow of capital A, if a can be the right most symbol in some sentential form then dollar is in the follow of A. So, if you have if you start with S S and see that we can derive something. So, that a is the last symbol of that sentential form then it is assumed that this may be followed by the dollar symbols and dollar matches with the end of strings.

So, whenever we have got a string. So, the end of string is marked by the dollar symbol. So, that is a convention followed by these compiler designers. So, it assume that the last symbol of the input is the end of string symbol which is dollar. So, if this A happens to be the last symbol of any sentential form then this dollar will also be in the follow of A.

So, these are the definitions of first and follow, first is any string that can be derived from alpha, whatever can come in the first whatever terminal symbol can come at the beginning so, that is the first of a alpha. Follow means, you take any derivations from the start symbol of the grammar and all the intermediately derivations in all the intermediately derivations, if this any non terminal a can be followed by another terminal symbol small a then the small a is going to be in the follow of follow set of capital A. So, that is all these first and follower define.

(Refer Slide Time: 22:02)

**Computing First**

- To compute  $\text{First}(X)$ , apply following rules until no more terminals or  $\epsilon$  can be added to any  $\text{First}$  set:

- If  $X$  is a terminal then  $\text{First}(X) = \{X\}$ .
- If  $X$  is a nonterminal and  $X \rightarrow Y_1 Y_2 \dots Y_k$  is a production for some  $k \geq 1$ , then place  $a$  in  $\text{First}(X)$  if for some  $i$   $a$  is in  $\text{First}(Y_i)$  and  $\epsilon$  is in all of  $\text{First}(Y_1), \dots, \text{First}(Y_{i-1})$  that is  $Y_1 \dots Y_{i-1} \Rightarrow \epsilon$ . If  $\epsilon$  is in  $\text{First}(Y_j)$  for  $j=1, \dots, k$  then add  $\epsilon$  to  $\text{First}(X)$ .
- If  $X \rightarrow \epsilon$  is a production then add  $\epsilon$  to  $\text{First}(X)$ .

Handwritten notes on the slide include arrows pointing from the text to various parts of the rules and from the rules to the production  $X \rightarrow Y_1 Y_2 \dots Y_k$ .

Now, how is it going to help us? So before going to that so, let us see how can you compute this first set and follow set. For computation of the first set, we apply the rules until no more terminals or epsilon can be added to any of the first set, what is the rule? So, if  $X$  is a terminal symbol. So, we are trying to compute the first of  $X$ , where  $X$  is a grammar symbol, it can be a terminal, it can be a non terminal.

So, if  $X$  is a terminal then first of  $X$  is definitely the  $X$  itself, because nothing more can be derived from a terminal symbol. So, that is there in the first of  $X$ , if  $X$  is a non terminal symbol and  $X$  producing  $Y_1, Y_2, Y_k$  is a production for some  $k$  greater or equal

1 then we place a in first of X, if for some i a is in first of Y i and epsilon is in all of first of Y<sub>1</sub> first up to first of Y<sub>i</sub> minus 1 that is Y<sub>1</sub> Y up to Y<sub>i</sub> minus 1, they produce epsilon.

So, let us try to explain this part. So, what it says suppose I have got the X is a non terminal symbol and we have got a rule which says it is Y<sub>1</sub>, Y<sub>2</sub>, Y<sub>k</sub>. So, anything that you derive from X suppose, this from this Y<sub>1</sub>, I can get a derivation like a alpha then Y<sub>2</sub> etcetera, this Y<sub>1</sub> can be expanded to a alpha then definitely a has is there in the first of X. So, that is what the first part says, if Y<sub>1</sub> whatever is there in the first of Y<sub>1</sub> will be in the first of X ok. So, if we just take i equal to 1 then. So, this whatever is in first of Y<sub>1</sub> will be in first of X other. So, what about the terminals belonging to the first of Y<sub>2</sub>? So, I will have this thing if Y<sub>1</sub> can give me epsilon.

So, if Y<sub>1</sub> can be reduced to epsilon then. So, I had got this Y<sub>1</sub>, Y<sub>2</sub>, Y<sub>3</sub>. Now, if this Y<sub>1</sub> can be reduced to epsilon then after sometime, I am getting the configuration like Y<sub>2</sub> Y<sub>3</sub> provided Y<sub>1</sub> can be reduced to epsilon by means of some steps ok. So, then this whatever is in first of Y<sub>2</sub> will also be in first of X provided Y<sub>1</sub> can give me epsilon.

So, in general so, if I have got say X producing Y<sub>1</sub>, Y<sub>2</sub>, Y<sub>i</sub> minus 1 Y<sub>i</sub> Y<sub>i</sub> plus 1 up to Y<sub>k</sub> then whatever is in first of Y<sub>i</sub> will come to the first of X provided each one of them can give me epsilon, that is first of all of them first of Y<sub>1</sub>, Y<sub>2</sub> up to Y<sub>i</sub> minus 1, all of them have got epsilon in them or I can say that is this Y<sub>1</sub>, Y<sub>2</sub> up to Y<sub>i</sub> minus 1 can give me epsilon. So, this is the second rule and it says that if X producing epsilon is a production then we add epsilon also to the first of X. So, that is if there is a production rule X producing epsilon then it will come.

(Refer Slide Time: 25:46)

	First	Follow
F	{(id)}	{+, *, ), \$}
T	{(id)}	{+, ), \$}
E	{(id)}	{), \$}
E'	{+, ε}	{), \$}
T'	{*, ε}	{+, ), \$}

So let us see how this first can be computed. So this is say this is a set of gamma rules E producing T E dash E dash producing plus TE dash or epsilon T producing FT dash T dash producing star FT dash or epsilon F producing within bracket E or id for some of the rules. So the first rule says that whatever is in first of T will be in first of E and if T can give me epsilon then whatever is in first of E dash will also be in first of E. Now, T does not give me epsilon. So naturally I do not have that liberty.

So whatever is in first of T will be in first of T. Now what is in first of T? First of T is equal to first of F because, by this rule you see that first of T is equal to first of F and what is first of F? So this rule will tell that open parenthesis in first of F and id is also in the first of F. So first of F has got open parenthesis and id and from this rule it says that, whatever is in first of F is in first of T. So first of T also has got open parenthesis and id and what about first of E? First of E by this rule it says that, whatever is in first of T is in first of E, first of T is open parenthesis id. So first of E will also have open parenthesis and id now first of E dash from this rule you see it is plus TE dash.

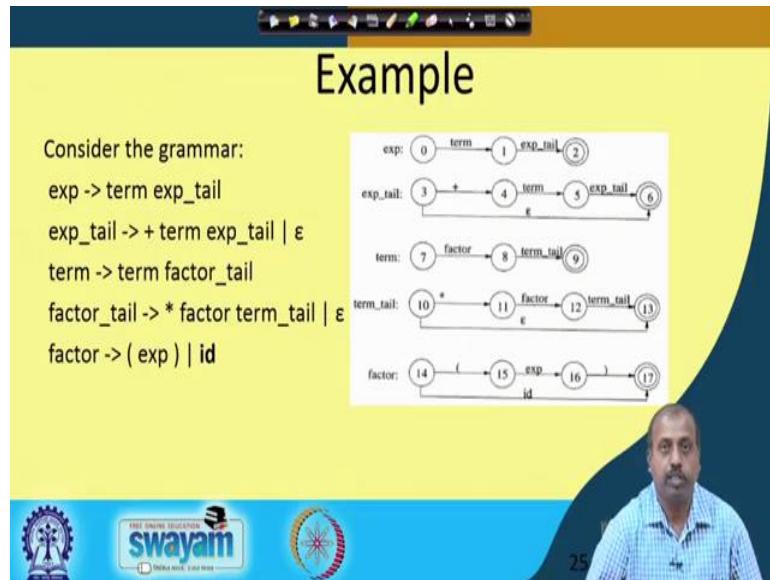
So it starts with a terminal symbol. So it cannot have anything else. So this plus is in first of E dash and E dash producing epsilon is a rule. So epsilon is also in the first of E dash. So, E dash so, first of E dash has got plus and epsilon and finally, this T dash it can give me star and epsilon because, this can give me star and this is epsilon. So star

and epsilon can come. So in this way we can compute the first set for the gamma rules that we have here.

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

**Lecture - 21**  
**Parser (Contd.)**

(Refer Slide Time: 00:15)



So, today we will first look into another example of grammar from which we can make transition diagrams for predictive parsing. So the grammar that we considered is like this that it; it is similar to that expression grammar. So that expression produces a term and expression tail where expression tail is plus then term then expression tail or epsilon. So, it is similar to that ETF grammar for we have eliminated that left recursion and we are writing it explicitly. So, T E dash we are writing like expression tail then T dash we are writing it as a factor term tail or factor tail like that.

So, this term produces term or followed by factor tail and factor tail is star factor; then term tail or epsilon and factor produces within bracket expression on id. So, as the transition diagrams that we make are like this that first we have got this further expression. So, it is state 0 from there it goes on term it goes to state 1 and from there an expression tail; it will go to state 2 and state 2 is a final state.

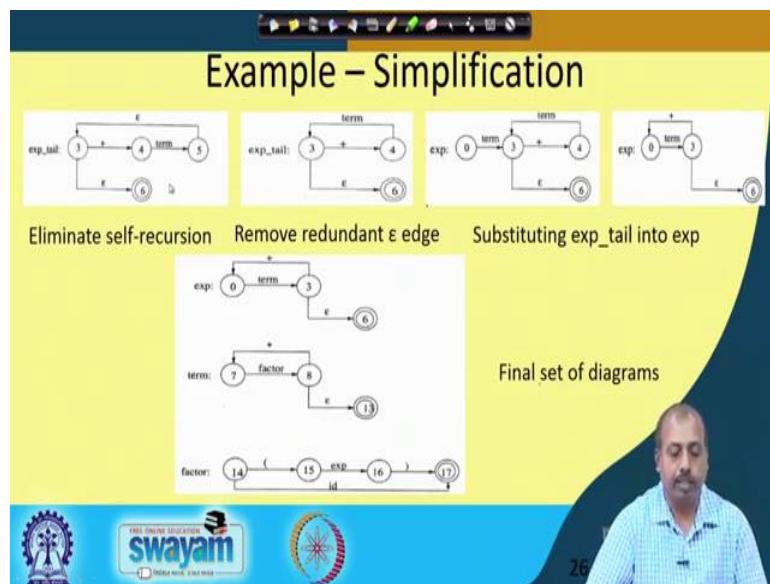
Then these expression tail; so expression tail produces plus term then expression tail epsilon or epsilon. So, it is like this that expression tail. So, it is starting at state 3; so, 1

plus it will go to state 4 and then on term it will come to state 5 and on from 5 on expression tail it will go to 6. And there is an epsilon production, so from 3 on epsilon it will go to state 6. Similarly term produces term and then factor tail, so it is represented like this.

So, then this factor tail producing star factor term tail; so it is. So, it is term tail producing that is that is basically this is; this should be actually factor tail yeah this should be factor tail. So, this is producing like this the term tail producing star factor and term tail. So, this way this whole like expression is made and this factor produces within bracket expression and id.

So, with bracket start expression then bracket close or this is id. Now we can simplify this expression this set of transition diagrams by like this.

(Refer Slide Time: 02:47)



For example this expression this expression tail we are; we can eliminate the self recursion, like here you see in the expression tail; so it is calling the routine expression tail again; so instead of that from 5, we can come back to state 3.

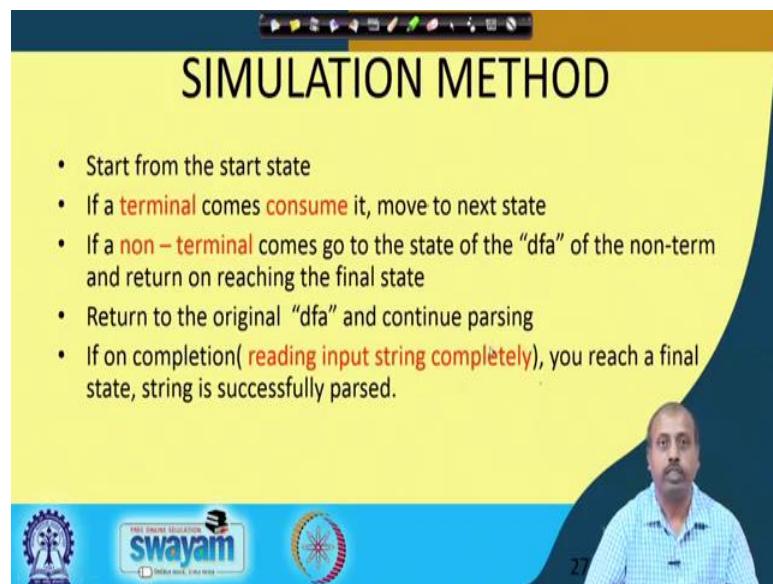
So, that is done here from 5 on epsilon it comes back to state 3. So, that is calling the expression tail. So, by this way we can and simplify the diagram a bit; we can and remove the redundant epsilon adjacent like this is a redundant epsilon edge. So, we can

all together remove this state 5 from state 4 on getting term I can come to state 3; so that is there that is possible.

Similarly, from then if we substitute this expression tail in to the diagram for expression; so in the diagram of expression was like this there we had expression tail now we have our modified the expression tail. So, we just substitute that expression tail diagram there. So, this is giving us expression the; this is giving us that this term then this expression tail diagram is a substituted; so, it is like this after that you can simplify.

Like say from here on term it is going from 0 to 3 and then on plus and term it is coming back to state 3. So, instead of that on plus I can take it back to stage 0; so it is done like this and then on a epsilon it goes to state 6. So, that we can simplify the expression the; these diagrams. So, this is the final set of diagrams where we have got in the diagram for expression term and factor where we have substituted and eliminated the redundant edges and states and this is the final set of states. So, we can use this diagrams for purpose of predictive parsing; so, as we have seen in the last class.

(Refer Slide Time: 04:43)



So, we can do a simulation and then we can see whether a given string is acceptable by the set of diagrams.

(Refer Slide Time: 04:51)

**Computing First**

- To compute  $\text{First}(X)$ , apply following rules until no more terminals or  $\epsilon$  can be added to any  $\text{First}$  set:
  1. If  $X$  is a terminal then  $\text{First}(X) = \{X\}$ .
  1. If  $X$  is a nonterminal and  $X \rightarrow Y_1 Y_2 \dots Y_k$  is a production for some  $k \geq 1$ , then place  $a$  in  $\text{First}(X)$  if for some  $i$   $a$  is in  $\text{First}(Y_i)$  and  $\epsilon$  is in all of  $\text{First}(Y_1), \dots, \text{First}(Y_{i-1})$  that is  $Y_1 \dots Y_{i-1} \Rightarrow \epsilon$ . If  $\epsilon$  is in  $\text{First}(Y_j)$  for  $j=1, \dots, k$  then add  $\epsilon$  to  $\text{First}(X)$ .
  1. If  $X \rightarrow \epsilon$  is a production then add  $\epsilon$  to  $\text{First}(X)$

Next we were discussing on this first and follow computation because we want to have a non recursive version of this predictive parsing.

So, the first computation we have seen in the last class where we say that; so, if I have got a general production rule like this  $X$  producing  $Y_1, Y_2$  upto  $Y_k$  then so this whatever is in first of  $Y_1$  will be in first of  $X$  accepting epsilon. And then whatever see if first of  $Y_1$  contains epsilon then whatever is in first of  $Y_2$  accepting epsilon will also be in first of  $X$ .

And so if all of them have got epsilon in their first  $Y_1$  to  $Y_k$ , then only epsilon will be there in the first set  $X$ . So, this way we can compute the first set and we have seen an example in the last class that shows us how to compute the first set.

(Refer Slide Time: 05:55)

- To compute  $\text{Follow}(A)$  for all nonterminals  $A$ , apply following rules until nothing can be added to any follow set:
  1. Place  $\$$  in  $\text{Follow}(S)$  where  $S$  is the start symbol
  2. If there is a production  $A \rightarrow \alpha B \beta$  then everything in  $\text{First}(\beta)$  except  $\epsilon$  is in  $\text{Follow}(B)$ .
  3. If there is a production  $A \rightarrow \alpha B$  or a production  $A \rightarrow \alpha B \beta$  where  $\text{First}(\beta)$  contains  $\epsilon$ , then everything in  $\text{Follow}(A)$  is in  $\text{Follow}(B)$

So, today will next look into the computation of follow. So, follow means the terminal symbols that can follow a particular non terminal in any sentential form. So, sentential form we say that thing starting with the start symbol of the grammar.

So, if you try to derive strings then all possible have to think about all possible strings of terminals and non terminals that can be derived from the start symbol of  $S$ ; then in any of those strings if a terminals symbols appears after in non terminal symbols, then we say that the follow set of the non terminal symbol will have this terminal symbol. So, a typical; so the rule the you can understand that the rule that we are going to follow.

Like if this is a production rule  $A$  producing  $\alpha B \beta$  then whatever I can derive from  $\beta$  and whatever be the first set of that; so that their those symbols will be following  $B$ . For example, if first set of  $\beta$  contains the symbols of a small  $a$  and small  $b$ , then small  $a$  and small  $b$  will be in the follow up  $B$ .

So, accepting epsilon of course, if epsilon we cannot take because in a may be the actual string that we have is  $\alpha B \beta$ ; then some  $\gamma$  etcetera. So, even if this give me epsilon; so these first on  $\gamma$  whatever symbols are there will be coming to the follow up  $B$ . So, this way we can we can compute the follow up  $B$ ; so, whenever we have got a rule do like this;  $A$  producing  $\alpha B \beta$ . So, whatever is in first set of  $\beta$  will be in the follow set of  $A$ ; so, that is one rule. The other rule that we have is if there is a

production A producing alpha B or A producing alpha B beta. So, I will come to this part later.

(Refer Slide Time: 07:47)

Computing Follow

- To compute  $\text{Follow}(A)$  for all nonterminals A, apply following rules until nothing can be added to any follow set:
  - Place  $\$$  in  $\text{Follow}(S)$  where  $S$  is the start symbol
  - If there is a production  $A \rightarrow \alpha B \beta$  then everything in  $\text{First}(\beta)$  except  $\epsilon$  is in  $\text{Follow}(B)$ .
  - If there is a production  $A \rightarrow \alpha B$  or a production  $A \rightarrow \alpha B \beta$  where  $\text{First}(\beta)$  contains  $\epsilon$ , then everything in  $\text{Follow}(A)$  is in  $\text{Follow}(B)$

$S \rightarrow \dots \rightarrow a_1 a_2 A_1 a_3 \dots A_2 \Rightarrow a_1 a_2 A_2 a_3 \dots a_3 B a_4 \dots$   
 $\rightarrow a_1 a_2 A_2 a_3 \dots a_3 B a_4 \dots$   
 $a_1 a_2 A_2 a_3 \dots a_3 B a_4 \dots$

I will come to this, but later; consider this rule A producing alpha B. Then where there then everything in follow of A will be in follow up B.

Why? Suppose starting with the start symbol of the grammar; we are deriving strings at a some point of time, I have got a string where say this is say a 1. So, these symbols are there this time in terminal and non terminal symbols are there ok. Say they after some time there is one A and after that I have got a symbol say small a ok.

Then in the next production what I can do? So this a may be replaced by alpha B. So, what can happen is that the string will get transformed into something like this. So, this is alpha B a so; that means, if a can come in the follow of capital A; if small a can come in the follow of capital A; then small a will also come in the follow of capital B that is obvious because from the in the next stage I can replace this A by alpha B; so, whatever symbol was following A will now follow B.

So, this is the; so that is the first part ok. Now if the rule is likely complex like instead of B alpha B; if the rule is alpha B beta in that case; so, what are the symbols in follow of B? So, whatever is in first of beta is in follow of B that we have seen in this rule. Also if this first set of beta contains epsilon if first set of beta contains epsilon; then what will

happen is that say from this I can rewrite this as a 1, a 2 then capital A 1, capital A 2, then b 1 etcetera. And then this A can replaced by alpha B beta then small a and it can go like this.

Now if first of beta contains epsilon; that means, from this beta this of there was through some derivational steps this beta can be replaced by epsilon. So, that the string gets transformed into something like this a 1, a 2, capital A 1, capital A 2, b 1, then alpha B; this beta part reduces to epsilon; so a comes immediately; so that can happen.

So, actually these two rules are similar if you say that the first of beta contains epsilon then these two rules are similar. So, naturally I can whatever is in follow of A will be in follow of B; so that is the second rule. So, these are the 3 rules and the so for the star symbol of the grammar there is a special rule that dollar will be in the follow of S. So, using these 3 rules; so you could able to construct the follow set. So, let us take some example and try to see like how this follow computation can be done; say for this grammar.

(Refer Slide Time: 10:59)

	First	Follow
F	{(), id}	{+, *, ), \$}
T	{(), id}	{+, ), \$}
E	{(), id}	{), \$}
E'	{+, ε}	{), \$}
T'	{*}, ε	{+, ), \$}

Now, for this grammar; so for applying the first rule that says that the if something is the star symbol of the grammar, then dollar will be in the follow set of E.

Now start symbol of the grammar is E; so the dollar will be in the following set of E. So, dollar is included now you scan through this grammar; try to apply the first the try to apply the rule that you whenever we have got A producing; A producing alpha B beta,

then whatever is in fast of beta is in follow of B. So you if try to apply this; so apply to the first rules alpha is epsilon, B is T and beta is E dash. So, whatever is in first set of E dash can be in the follow of T. So, first of E dash already computed plus and epsilon.

So, out of that epsilon I will not take; so this plus will go to the follow of T; so follow of T has got the plus symbol. Then by apply for second rule also if you have tried to apply that first principle alpha B beta. So, alpha is plus, B is T and beta is E dash as this does not add anything; so it remains as it is. Then if we apply in this rule then whatever is in first of T dash will be in the follow of A. So, first of T dash has got star and epsilon; so follow of F, so follow set of F has got star there.

From this rule nothing new comes because whatever is in follow of T dash will be in the first of T dash will be in the follow of F; so that we have already done. And here you see; if we apply this rules alpha B beta; so this is alpha, this is B and this is beta. So, whatever is in first set of beta is in follow of E; so first set of beta that is close parenthesis. So, first set of it contains close parenthesis only; so in the follow of E will have close parenthesis; the follow of E has got close parenthesis is there.

Now, we will apply the second rule; so, whenever you have got A producing alpha B; then whatever is in follow of A is in follow of B. So, whatever is in so follow of E will be in follow of E dash by this rule whatever is in follow up T will be in the. So, follow up E dash. So, follow of E dash follow E has got say close parenthesis and dollar. So, they will be added to the follow of E dash; so close parenthesis and dollar added here.

Similarly, if you apply this rule; so E dash producing plus T E dash for first of E dash contains epsilons. So, by the second rule applies for the follow set and then this whatever is in follow of E dash will be in the follow of T. So, follow of E dash has got close parenthesis and dollar. So, close parenthesis and dollars they are added to the follow of T.

So, coming to the third rule; so whatever is in follow of T will be in follow of T dash. So, follow of T has got plus close parenthesis and dollar. So, they are added to the follow of T was follow of T dash. So, follow of T dash; so, follow of T dash has got plus close parenthesis and dollar. So, this way we have to go on applying the rules again and again till the follow sets do not change.

So, it is slightly involved the process is slightly involve because you have a you really do not know like when you are going to terminate; so you have to go on trying the 3 rules that we have listed for follow computation and whichever rules; when there is no more addition to any of the follow sets, then only the computation will terminates. So, this way it becomes a bit difficult to compute the follow set. But you can very easily write a computer program and for doing this thing iteratively till we are coming to the all the all the sets they have got the maximum possible increase in them. So, this way we can compute the first and follows sets. So, once you have computed the first and follow sets we can go for constructing the parser for this predictive parser which is a non recursive parser and they are known as LL 1 grammars.

(Refer Slide Time: 15:35)

## LL(1) Grammars

- Predictive parsers are those recursive descent parsers needing no backtracking
- Grammars for which we can create predictive parsers are called LL(1)
  - The first L means scanning input from left to right
  - The second L means leftmost derivation
  - And 1 stands for using one input symbol for lookahead
  - More general one is LL(k), with k symbol lookahead
- A grammar G is LL(1) if and only if whenever  $A \rightarrow \alpha \mid \beta$  are two distinct productions of G, the following conditions hold:
  - For no terminal  $a$  do  $\alpha$  and  $\beta$  both derive strings beginning with  $a$
  - At most one of  $\alpha$  or  $\beta$  can derive empty string
  - If  $\alpha \Rightarrow^* \beta$  then  $\beta$  does not derive any string beginning with a terminal in  $\text{Follow}(A)$

So, this type; so as I said that the it is a left to right the first L, so there are 2 Ls here there are 2 Ls here; the first L the first L will mean that the parser will take a left to right scan of the input. And second L means it will produce a left most derivation.

So it will the parser are the compiler that will be designed based on a LL 1 strategy. So, it will be scanning the input from left to right and it will produce a left most derivation of the input strings. And in general we have got LL k; so this in this particular case k is equal to 1; that will tell how many symbol look ahead we use for a; parser to proceed.

So look ahead means that at present so we are looking at the current symbol. So, if it is looking only at the current symbol then will be calling it LL 1 grammar or LL 1 parser. Now if we see that; so now the decision is taken based on the current symbol only.

So, if you need to check k more board symbols ahead to take a decision; in that case the parser will be called and LL k parser. So, the grammar will be called LL k grammar and this is the parser will be called LL k parser. So, there is a rule which says that this they grammar G is LL 1; if and only if whenever A producing alpha or beta are two distinct productions of G; the following conditions hold.

For no terminal a do alpha and beta both derive strings beginning with a. So, it should not be that for both of them. So, if you think about the strings which are derivable from alpha.

(Refer Slide Time: 17:45)

- Predictive parsers are those recursive descent parsers needing no backtracking
- Grammars for which we can create predictive parsers are called LL(1)
  - The first L means scanning input from left to right
  - The second L means leftmost derivation
  - And 1 stands for using one input symbol for lookahead
  - More general one is LL(k), with k symbol lookahead
- A grammar G is LL(1) if and only if whenever  $A \rightarrow \alpha \mid \beta$  are two distinct productions of G, the following conditions hold:
  - For no terminal a do  $\alpha$  and  $\beta$  both derive strings beginning with a
  - At most one of  $\alpha$  or  $\beta$  can derive empty string
  - If  $\alpha \Rightarrow^* \epsilon$  then  $\beta$  does not derive any string beginning with a terminal in  $\text{Follow}(A)$

Alpha is a string of terminals and nonterminals; so if you derive further strings from alpha. So, its suppose it starts with symbol a the string starts with symbol a. So, with beta if I do the same thing it should not be the case that is beta also I can come to a situation where the derived string starts with a.

Naturally, in that case there will be confusion. So, if you look into the first of alpha and first of beta if both of them contains a; so you cannot decide that you whether on getting a, whether you should follow the production a producing alpha or the production a

producing beta; so that is one thing. Second point is that at most one of alpha or beta can derive empty string; so that is epsilon production.

So, that is also there like; so if you find that at some point of time I need to consume entire capital a in my sentential form. Without consuming any further input, so I should try to replace it by alpha where alpha can give me epsilon or beta if beta can give me epsilon, but it should not happen that both alpha and beta can give epsilon; so, at most one of them can give epsilon.

So then we do not have any problem of choice; otherwise there is a choice. So, I can either produce by a producing alpha or a producing beta. So, I really do not know which one will be correct; so that can lead to some erroneous action by the parser. And if a in if alpha in a number of steps produces epsilon then beta does not derived any string beginning with the terminal in the follow of a; so, this is another condition.

So, it says that you; so I have got to something some string at this I have got A and then I have got something else. Now I have got a choice I can replace A by alpha or A by beta; so if I replace A by alpha, if I replace A by alpha and alpha can give me epsilon; alpha can give me epsilon, then from beta I should not be able to derive any string that has got a will terminals in follow of A.

So, that is this if I could have gamma with beta and it can give me a valid string where there is a there is a symbol which is in follow of b that can comes; so b is in follow of capital A; so that can come. So, then there will be confusion again. So, these are the three conditions that have that are to be satisfied for a grammar to be LL 1. So, the detail proof can be obtained in some formal language theory book, but we are not looking into that. So, we would be following these rules to determine whether a grammar is LL 1 or not. Even without doing these checks; so we can decide a like whether is a grammar LL 1 or not by constructing the corresponding parsing table.

And if the grammar is not LL 1, then there will be multiple entries in the parsing table for a particular value. So, naturally whenever there are multiple values at some entries; so we have got there we say that there is a the grammar has got problems. So, it is not LL 1 grammar; so we will see that.

(Refer Slide Time: 21:21)

## Construction of predictive parsing table

- For each production  $A \rightarrow \alpha$  in grammar do the following:
  - For each terminal  $a$  in  $\text{First}(\alpha)$  add  $A \rightarrow \alpha$  in  $M[A,a]$
  - If  $\epsilon$  is in  $\text{First}(\alpha)$ , then for each terminal  $b$  in  $\text{Follow}(A)$  add  $A \rightarrow \epsilon$  to  $M[A,b]$ . If  $\epsilon$  is in  $\text{First}(\alpha)$  and  $\$$  is in  $\text{Follow}(A)$ , add  $A \rightarrow \epsilon$  to  $M[A,\$]$  as well
- If after performing the above, there is no production in  $M[A,a]$  then set  $M[A,a]$  to error

The slide has a yellow header with the title 'Construction of predictive parsing table'. Below the title is a bulleted list of steps for constructing a predictive parsing table. At the bottom of the slide, there is a blue footer bar with the 'swayam' logo and other navigation icons.

So, how do you construct a predictive parsing table? So, what the predictive parsing strategy will do is that it will have a parsing table; so like this.

(Refer Slide Time: 21:31)

### Example

Input Symbol					
Non-terminal	id	+	*	(	)

	First	Follow
F	{(, id)}	{+, *, ), \$}
T	{(, id)}	{+, ), \$}
E	{(, id)}	(), \$}
E'	{+, ε}	(), \$}
T'	{* , ε}	{+, ), \$}

	id	+	*	(	)	\$
E	$E \rightarrow TE'$	ERROR	ERROR	$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

The slide has a yellow header with the title 'Example'. Below the title, there is a list of grammar rules:  $E \rightarrow TE'$ ,  $E' \rightarrow +TE' \mid \epsilon$ ,  $T \rightarrow FT'$ ,  $T' \rightarrow *FT' \mid \epsilon$ , and  $F \rightarrow (E) \mid id$ . To the right of the rules is a table with columns for 'First' and 'Follow' and rows for non-terminals F, T, E, E', and T'. Below this is a parsing table with columns for 'Input Symbol' (id, +, \*, (, ), \$) and rows for non-terminals E, E', T, T', and F. The table entries indicate which rules can derive the input symbols. At the bottom of the slide, there is a blue footer bar with the 'swayam' logo and other navigation icons.

So, it will have a parsing table and looking into the also looking into the; grammar symbol as that is there on some stack and the next input symbols; so it will decide like what action to take. So, is the stack top contains E and the next input symbols is id; then this table says that you go by this rule E producing T E dash, so it will go by this rule ok.

(Refer Slide Time: 22:01)

**Construction of predictive parsing table**

- For each production  $A \rightarrow \alpha$  in grammar do the following:
  1. For each terminal  $a$  in  $\text{First}(\alpha)$  add  $A \rightarrow \alpha$  in  $M[A,a]$
  2. If  $\epsilon$  is in  $\text{First}(\alpha)$ , then for each terminal  $b$  in  $\text{Follow}(A)$  add  $A \rightarrow \epsilon$  to  $M[A,b]$ . If  $\epsilon$  is in  $\text{First}(\alpha)$  and  $\$$  is in  $\text{Follow}(A)$ , add  $A \rightarrow \epsilon$  to  $M[A,\$]$  as well
- If after performing the above, there is no production in  $M[A,a]$  then set  $M[A,a]$  to error

Diagram: A grammar rule  $A \rightarrow a$  is shown with arrows indicating the derivation from non-terminal  $A$  to terminal  $a$ .

SWAYAM

So, I think there is a so there; so we will see how to construct this parsing table. So, for each production  $A$  producing alpha in the grammar so we will do like this.

For each terminal  $a$  in first of alpha we add a producing alpha in the table  $A$   $a$ . So this table is 2 dimensional table in one dimension we have got all the non terminals; so, this is the table  $M$ . So, here I have got all the non terminals from the set  $M$ , this set I have got all the terminals ok. So, that way; so this  $M$   $A$   $a$  is corresponding to this particular non terminal and this particular terminal. So, what is the entry here; so that entry is decided by this.

So, if  $A$  is in first of alpha where if  $A$  producing alpha is a rule, if  $A$  producing alpha is a grammar rule and first of alpha contains  $A$ , then we add this rule  $A$  producing alpha to this particular entry. Then it says that if epsilon is in first of alpha then for each terminal  $b$  in follow of  $A$ , we add a producing alpha to the table  $M$   $A$   $b$ . So, if epsilon is in the first of alpha; so this can give me epsilon then we for every terminal which can give me follow of  $A$ , which have which there is in follow of  $A$ ; so, I will like to reduce this by epsilon.

So, this will be added to  $M$   $A$   $b$  and finally, if epsilon is in first of alpha and dollar is in follow of  $A$ , then we add  $A$  producing alpha to  $M$   $A$  dollar as well. And after doing this whatever you go whatever entries are undefined; we mark them as error entry. So, how does it also if I look into the example like say how is it constituted? So, this is the

grammar rules that we have ETF grammar modified by eliminating left recursion and all and this is the first and follows sets that we have computed.

Now let us look into this; let us look into this rule E producing TE dash. Now as far our as far our rule is concerned. So, it says that if A producing alpha; so you have to see the first of alpha. So, first of alpha first of TE dash is equal to first of T; so it is bracket start and id. So, in bracket start and id we add this rule. So, E producing T dash is added here and here fine; then we have to have this whether this first of T dash; parser T dash contains epsilon not that has to be seen.

So, as for the second rule it says that if epsilon is in first of alpha. So, whether epsilon is in first of alpha for not; so that has to be seen. So first of TE dash; so first of TE does not contain epsilon. So, there is no question of first TE dash containing epsilon; so second rule is not necessary. Now come to the rule E dash producing the plus E dash; so first of plus TE dash is plus, so I will add this rule E dash producing the plus TE dash in this particular entry.

Now look into this E dash producing epsilon. So, bye that second rule it says that by the second rule; it says that whenever I have got epsilon A producing epsilon; then and whatever we have in the follow of A, there I should add this particular rule. So, follow of E dash the follow E dash is the bracket close and dollar.

So, E dash bracket close I have added E dash producing epsilon and dollar I have [vocalized-noise given E dash producing epsilon; so this is added to these 2 rules. Then this next coming to this rule T producing FT dash; so FT dash first of a FT dash will is equal to first of a; that is bracket start an id. So, bracket id and bracket start; so we have added this T producing FT dash.

Now comes T dash producing star FT dash. So, by the say by the similar rule like T dash produces star; so, the first of the star FT dash is equal to star. So, at the step at T dash star; so we add this particular rule T dash produces star FT dash. And then T dash produces epsilon, so whatever we have in the follow of T dash plus bracket close and dollar; so there I will add T dash producing epsilon. So, plus bracket close and dollar we have added T dash producing epsilon.

Then finally, so this particular rule F producing within bracket E; so first of this contains bracket start. So, F bracket start is F producing within bracket E and this id; so this id part is very slowly this is applying other rules a producing id. So, first of this a part is id only; so, this F id, so we have added this thing. So, the simple rules for you have got the first look into the first set. And whatever 5 symbols are coming in the first set; so you add the rules at those points and if the first produces epsilon; then whatever is the follow of the point on terminal then for all of them we add the rule. So, that is the straightway rule for producing the predictive parsing table.

So, we will see how we can use it for this passing job. Now we can produce the other entries like all these entries which are left as blank; so they are actually error entries. So error entries means the; so if you have got a non terminal E and the next input symbol is plus; so on E you cannot get it plus ok; this is an error. So, all the entries that we have here so they are; they can be marked as error; they can be marked as error.

So whenever the parser comes to the state this particular table entry so; that means, there is an error in the input string; as a result it has come to this. So, the input stream has to be rejected it is not there is some syntax error and the parser needs to recover so that it can continue parsing with the remaining strings.

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

**Lecture – 22**  
**Parser (Contd.)**

(Refer Slide Time: 00:14)

The slide title is "Another example". It contains the following information:

- Grammar rules:  
 $S \rightarrow iEtSS' \mid a$   
 $S' \rightarrow eS \mid \epsilon$   
 $E \rightarrow b$
- First and Follow sets:  
 $\text{First}(S) = \{i, a\}$   
 $\text{First}(S') = \{e, \epsilon\}$   
 $\text{First}(E) = \{b\}$   
 $\text{Follow}(S) = \{\$, e\}$   
 $\text{Follow}(S') = \{\$, e\}$   
 $\text{Follow}(E) = \{t\}$
- A parsing table with columns for Non-terminal (S, S', E) and Input Symbol (a, b, e, i, t, \$). The table shows which non-terminals derive specific input symbols.

You take another example of this predatory parsing table construction. So, this is the grammar that we have that is  $S$  producing this is basically that if then else grammar. So,  $i$  for if so,  $i$  is a terminal symbol  $t$  is a terminal symbol like that  $i$  is the, if  $t$  is then and  $E$  is expression so, like that. So, we have got. So, we the expression part we make it  $b$  for the sake of simplicity. So, that is also taken as a terminal. So, to make the parser small. So, we have taken as if I can have only this sort of thing I can have if then the expression is some  $x$  then we have to write like this and this if is also  $i$  then is  $t$ . So, like that anyway so, this is the grammar.

Now, so, first we have to for constructing the predictive parsing table the first thing that we have to do is to compute the first and follow sets. So, first of  $S$ , if you go by this rule and so, the first rule. So, it says that  $I$  will be there in the first of  $S$  by the second rule it says that  $a$  should be there in the first of  $S$  similarly first of  $S$  dash it will have  $e$  and this epsilon. So, these two are there and first of  $E$  will have  $b$ .

So first computation is straightforward now the follow computation. So, follow S is the start symbol of the grammar. So, dollar must be in the follow of S now in the whatever is so. So, now, this follow of E follow of E is t. So, from this rule you can see that e may be followed by E t. So, that will have that can have t. Now what if so, by now we can now look into this part? So, S S dash. So, S S dash means whatever is in first of S dash can be in the follow of a.

So, first of S dash has got e and epsilon. So, epsilon we cannot take, but e you can take. So, follow of S follow of S can be e. So, follow of S we have added e and dollar was already there and by this rule is producing i S. So, often i E t S S dash so, whatever is in follow of S can be in the follow of S dash. So, follow of S has got dollar and e. So, follow of S dash will also have dollar and e.

Now,. So, this way we can go on arguing like what are the elements that can be there in the follow sets and apply those rules for all the productions and ultimately when the set does not change you stop at that point. So, we so, you can check that after this no more symbol can be added to the follow sets of the non terminals and once we have done that. So, we can try to construct the, this predictive parsing table.

(Refer Slide Time: 03:26)

Another example						
Non-terminal	Input Symbol					
	a	b	e	i	t	\$
S	S → a			S → iEtSS'		
S'			S' → ε S' → eS			S' → ε
E		E → b				

The first thing that we have is say this rule S producing a. So, this rule says that the, whatever is in first of a they are I have to add this S producing a. So, first of a is a only. So, S producing a is added here similarly from this side it says that whatever is in first of

this  $i E t S S$  dash there I have to add this rule. So, first of this is  $i$  only so, this is added here  $S$  producing  $i E t S S$  dash. So, that is done so, first rule is done.

Now, the second rule it says that  $S$  dash producing  $e S$ . So,  $S$  dash producing  $e S$  it says that whatever is in first of  $e$  first of  $e S$  there  $i$  have to add this rule. So,  $S$  dash produces the first of  $e S$  is  $e$ . So,  $S$  dash producing  $e S$  is added here. Now  $S$  dash producing epsilon is there. So, that says that whatever is in follow of  $S$  dash there I should follow of put this rule  $S$  dash producing epsilon. So, follow of  $S$  dash has got dollar and  $e$ . So, in dollar  $i$  put  $S$  dash producing epsilon and in  $e$  also  $i$  put  $S$  dash producing epsilon.

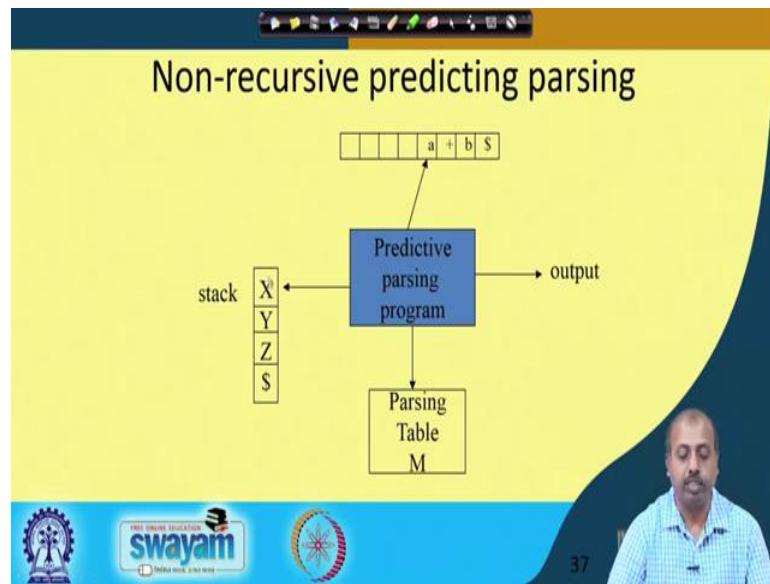
And then this  $e$  producing  $b$  and  $b$  has got those on the first of  $e$  contains first of  $b$  contains  $b$  only. So, there I add this rule the  $e b$  has got  $e$  producing  $b$  this particular rule. Now look into this situation like for this particular cell. So, what has happened is that there are two rules and by which we can proceed. So, this is actually happening because the grammar that we have taken is that that if then else grammar and this if then else grammar it is ambiguous grammar.

So, that is why. So, this situation has occurred. So, it says that on  $S$  dash. So, if you have seen up to  $S$  dash then if you currently seen  $S$  dash now if you see an  $e$  part then if you see an  $e$  part then one action says that you reduce this  $S$  dash by epsilon. So, that this  $S$  dash part does not come. So, it is taken as  $a$  is the previous statement is taken as if expression then statement.

The second rule is telling you take it you expand it as  $S$  dash producing  $e S$ . So, that the expression is taken as the statement is taken as  $e$  I if condition your, if expression then statement else statement. So, it is telling that you follow the second option. So, both of them are correct as far as the parse tree generation is concerned because this is an ambiguous grammar. So, we can have this type of problem.

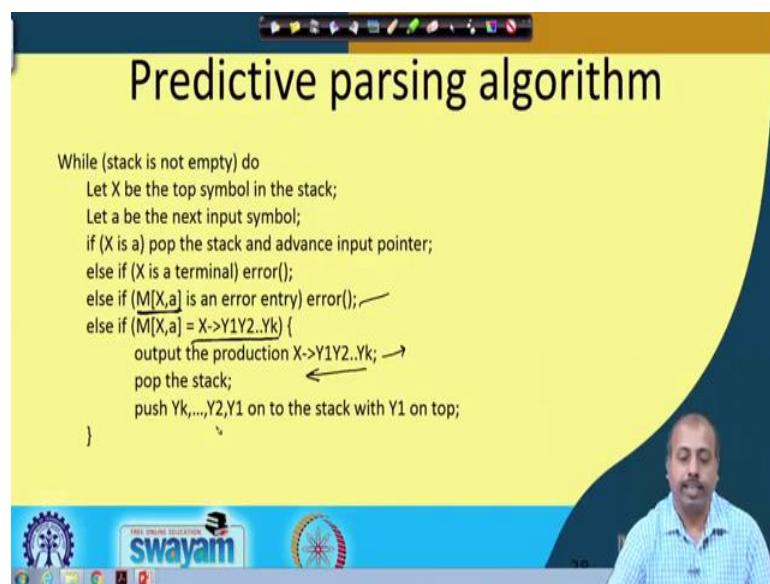
So, so, as I was telling so, once you construct the predictive parsing table. So, if the if some of the entries are multiplied defined, then the grammar is not 1 1 1 grammar so; however, for many grammar. So, it may be that case that they are 1 1 1 particularly the expression grammars and also they are 1 1 1. So, we can design parsers for using this 1 1 1 policy otherwise the policy is very simple accepting that follow computation part. So, remaining part remaining parts are quite simple.

(Refer Slide Time: 06:47)



So, how do we use it this predictive parse cells? How the how actually the parsing takes place? So, it has got a few components in it one is the stack. So, stack is a stack of non terminal symbols and it there is a dollar here and then we have got these production rules sorry we have got the input and then we have got a parsing table in. So, all of them come to this predictive parsing program and it produces output whether it is correct or not. So, stack can contain both terminals and non terminals. So, this XYZ they are actually grammar symbols. So, they can contain both terminals and non terminals.

(Refer Slide Time: 07:30)



So, how do we proceed? So, so, we look into the top of the symbol at the top of the stack. So, let the symbol be X and we also check the next input symbol suppose it is a. So, like the situation like top of the stack contains X and the next input symbol is a in that situation if X is a. So, if this top of the stack is a terminal and that is equal to the next input symbol then we pop out it from the stack and advance the input pointer.

So, if this X happens to be equal to a then this is popped out from the stack and input pointer is advance to the next position. As if there is a match between what we expect on the string so, that is available in the stack and what we actually get on the input string. So, since these two are matching so, we are just adverb we are just consuming that input and adopting out the symbol from the stack and advancing the whole parsing process.

So, that is their else if X is a terminal then error that is if the stack of top of the stack contains a terminal and it does not match with the terminal that you have as the next input ; that means, so, there is an error. So, you are expecting a. So, you are expecting a say b here and you are getting a, at the next input. So, there is an error so, that way it is that we it will be an error condition. So, if so, the so, we have excluded the possibility that the top of the stack contains a terminal symbol. So, if the terminal matches if the input it is fine if the terminal does not match with the next input then that is an error.

So, what remains is the top of the stack contains a non terminal. So, in that case we need to consult this particular entry MX a. So, the parsing table will be consulted and then if this MX a is error is an error entry. So, if the no rule is defined in that case the parser will also tell error. Otherwise if the rule if the entry says that entry says has this particular rule. So, X producing Y 1 Y 2 Y k, then we will output this production rule. So, for the sake of telling that by which rule, we are progressing like that.

So, we will be printing this particular rule will pop out X from the stack. So, X is taken out from the stack and these symbols are put into the stack this it is in it is in the reverse order. So, Y k Y 2. So, Y k Y k minus 1. So, that way they will be pushed so, that y one is on the top of the stack. So, that way this parser will proceed ok. Ultimately when your top of the stack will contain dollar and the input will also pointing to a dollar; that means, we have reached the end of the string and the whole parsing process was correct.

(Refer Slide Time: 10:33)

Stack	Input	Action
E	id+id*id\$	Parse E -> TE'
E'T	id+id*id\$	Parse E' -> FT'
E'T'F	id+id*id\$	Parse F -> id
E'T'id	id+id*id\$	Advance input
E'T'	*id\$	Parse T' -> FT'
E'T'F*	*id\$	Advance input
E'T'F	id\$	Parse F -> id
E'T'id	id\$	Advance input
E'T'	\$	Parse T' -> E
E'	\$	Parse E' -> +TE'
E'T+	+id*id\$	Advance input
E'T	id*id\$	Parse T -> FT'

So, we will see some example by which we can do this parsing consider that that each expression grammar that we have taken. So, this was the parsing table that m table that we have thought about now suppose we have got this particular string to be checked whether it is syntactically correct or not and we have to see how the parsing can proceed. So, we start with i d plus i d star i d so, this is the explained input.

So, that my input pointer is somewhere here and the top of the stack we put the start symbol of the grammar. So, the stack top has got E in it. So, as per our rule is concerned. So, it says that you have to check so, we have to check if X is a terminal do something in this case X is not a terminal because top of the stack contains E which is a non terminal. So, X is not a terminal so, we have to see the corresponding entry in the table MX a.

So, this MX a that is ME i d. So, ME i d is E producing plus TE dash. So, what will happen the action is we have to pop we have to take out this E from the stack. So, we have to take out E from the stack. So, the stack had e. So, this e is taken out of the stack now this TE dash. So, they will be put into the stack. So, that E dash is at the bottom and T is at the top. So, this whatever symbol comes first. So, that will be at the top.

So, this T E dash. So, so, here this stack is shown. So, this is the bottom this side is bottom and this side is top. So, top of the stack now contains T and input pointer is fixed at i d only. Now I have to consult T i d, now if you say T i d. So, it says T producing F T

dash. So, this T will be popped out from the stack and T dash and F they will be put into the stack they will be pushed into the stack.

Now, it contains a from the top of the stack and i d as the next input symbol. So, F i d says that you follow this rule F producing i d. So, F is taken out of the stack and i d is put into the stack. Next time the parser takes that top of the stack contains a terminal which is i d and the input pointer is also at i d. So, these two are matching so, this is taken out of the stack and the input pointer advances. So, input pointer now comes to this one the plus the input pointer now comes to the point plus.

Now, so, this is the situation so, input is like this. Now T dash and plus. So, T dash plus it says that T dash producing epsilon. So, right hand side replacement is done. So, nothing has to be pushed into the stack because it is the epsilon. So, T dash is going out of the stack input remains as it is now E dash and plus. So, E dash and plus says that you go by E dash producing plus T E dash. So, E dash is coming out of the stack and plus T E dash they are put into the stack, next this plus is checked and it matches with the next input symbol plus.

So, these two are matching so, they are taken out of the. So, this is taken out of the stack this is input pointer is advanced. So, that input is now pointing to the i d so, it is now pointing to the i d. So, we have got i d star i d dollar. So, this part is remaining and then this T and i d. So, T and i d it says that you follow T producing F T dash. So, T is going out and F T dash is coming into the stack input remains as it is.

Now, F and i d says that you go back F producing i d. So, this F is taken out an i d is pushed into the stack then i d i d matches. So, it is taken out of the stack and input pointer advances so, that this is left as the input. Now T dash and star so, T dash and star so, it goes by this rule T dash producing star F T dash. So, it is that it is replaced it S is replaced by star F T dash now star and star will match in the next iteration of the loop. So, they will be taken out so, it will have E dash T dash F and i d. So, F and i d says F producing i d. So, it is replaced F is replaced by i d now i d i d they match ok. So, they are i d is taken out of the stack this i d with the input pointer is an advanced.

So, now you have got a situation T dash and dollar. So, T dash dollar says T dash producing epsilon. So, this will be replaced by T dash producing epsilon. So, T dash will go out of the stack now you will have E dash and dollar. So, E dash and dollars is E dash

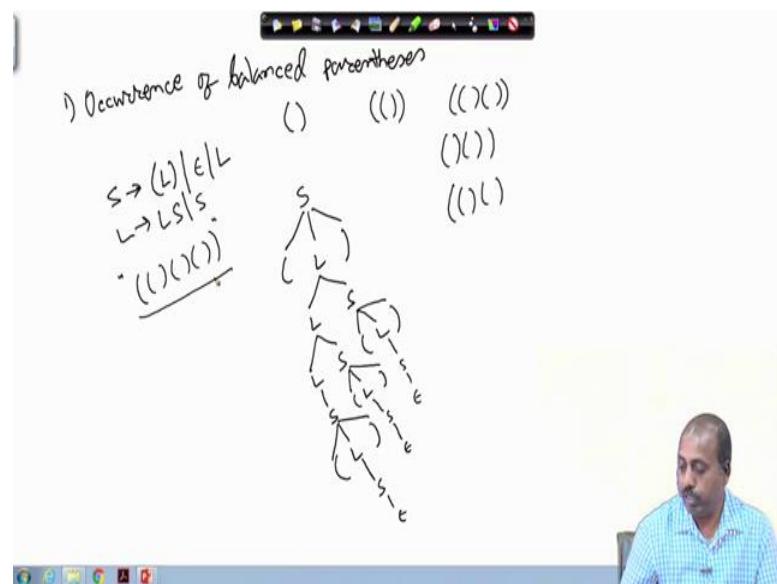
producing epsilon. So, that also goes out and then the input pointer is also a dollar. So, stack is empty and input pointer is also containing in the pointing to dollar symbol. So, that is the end of the string. So, we could successfully parse the input sequence.

So, this is the action in the action part we have purposefully written like how do we proceed and from this you can construct the parts tree. For example, so, you can if you follow these action sets first it says that parts by E producing T E dash. So, if I try to draw the words tree first I will do it like this T E dash next it says the parts using E dash producing F T dash. So, this is this is F T dash then it says that you follow rule F producing i d. So, F producing i d then advance input so, no action then it says T dash producing epsilon.

So, follow this rule T dash producing epsilon. Then it says that parts using this rule that. So, this is no actually from this it is slightly difficult. So, let us take some other example. So, these are the actions. So, these actions will tell us like how it will proceed, but from there constructing the parse tree your directly will be a problem ok. So, constructs we will take some example and then explain it again.

Now, so, we will take a few examples to work out and we will see how can we make a few grammars and we can write a few grammars and see like what we can how we can construct some parsers different types of parsers and all. So, for that purpose so, let us first try to write a few per gram example grammar.

(Refer Slide Time: 18:17)

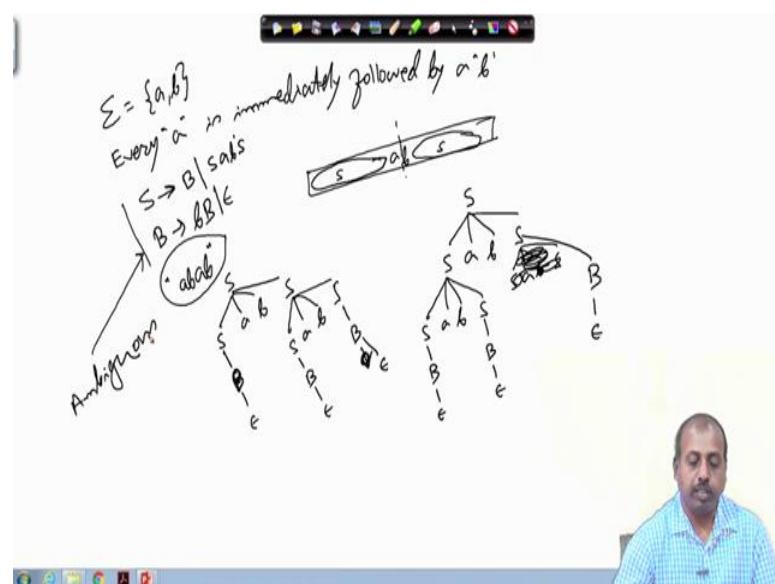


The first example that we write is occurrence of balanced parenthesis occurrence of balanced parentheses. So, balanced parentheses means so, this is a set of balance this is a balanced parenthesis then this is also a balanced parenthesis then this is also a balanced parenthesis but only got unbalanced so, maybe if somebody is writing like this. So, this is unbalanced or somebody may write like this ok. So, this is also unbalanced.

So, this way we can have some unbalanced parenthesis. So, how can I write a grammar for this purpose? So, the grammar for this balanced parenthesis is like this. So, S producing within bracket L or epsilon or L and this L gives L S or S ok. So, for example, suppose we try to derive the string some parentheses parenthesized string like say this one suppose we want to derive this particular string.

So, we can do it like this S producing brackets start L bracket close and then this L giving me L S and from this L S we can say this L can again be made to give L S, this S can give me like this within bracket L. Then this L can give me S and then this S can be made to give S L bracket close then this is S and this can be made to give epsilon similarly this S. So, that gives us the first parenthesis the first part of the parenthesis, then this can give me bracket start L bracket closed and this L can give me S and that can give me epsilon similarly this L can give me S giving me epsilon. So, this way you can draw the parse tree for this particular expression.

(Refer Slide Time: 21:27)



Now, so, next we will be looking into another example where we have got we have got the alphabet set as a b got the alphabet set as a b and every a is immediately followed by a b. In the string in the language every a is immediately followed by a b. So, that is the language that we are looking into.

So, what are the possibilities that the string may be simply a string of B is or it may be S then a is followed by b and then any other string. So, S is the start symbol of the grammar. So, S can derive all strings, where a is immediately followed by b. So, what we can have either the string does not contain any B or I can say the left part of the string I can I can a. So, if this is the whole string as if I can break at a point where one of them is a and other one is b ok.

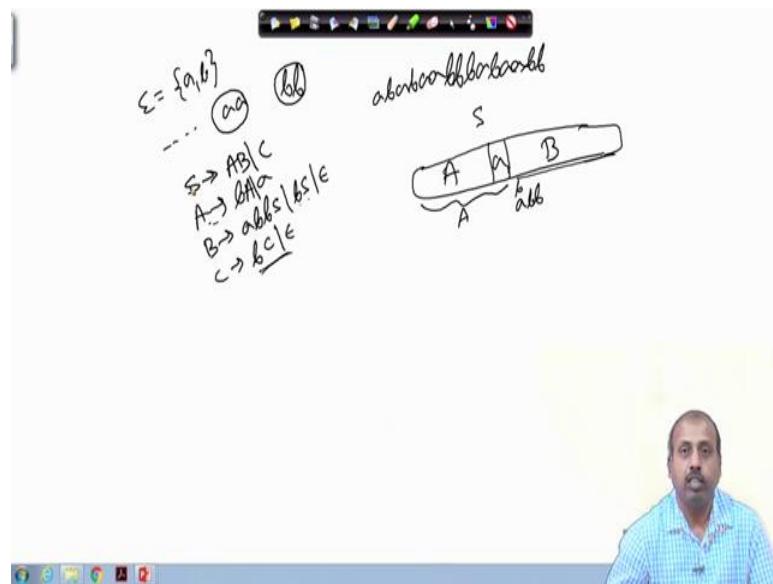
Now, what about this part? So, this part can be any string over a and b such that a is followed by b. Similarly this part also I can have any string of a and b such that a is immediately followed by b. So, we have got this thing that S. So, this is also type of S this is also type of S. So, we can write it like this is producing S a b S and where this B it can be any number of B is. So, that string can be generated by B or epsilon.

Now, let us look into a string a b a b if this is a string. So, one possible derivation for this is like this S a b S then this S gives me epsilon and this S gives me S a b S and then over. So, this S gives me sorry this is cannot give me epsilon. So, this can give me B and B can give me epsilon. So, this gives me B leading to epsilon this is gives me B leading then giving me epsilon.

So, that is one possibility; the other possibility that we have is like this S producing S a b S then this S can give me S a b S and this can give me B then epsilon this can give me B then epsilon and this can give me S a b S or sorry this is already there. So, this can give me B followed by epsilon. So, that way we can have we can have the two different parts trees for the same string a b a b. So, this particular grammar that we have written. So, this is an ambiguous grammar this is an ambiguous grammar.

However, ambiguous grammar. So, if I try to construct the 1 1 1 parsing table then naturally there will be there will be duplicate entries or duplicate values for some of the table entries. However, many cases it is permitted it is done because the table becomes smaller even if it is ambiguous the table becomes smaller. So, that way we can proceed.

(Refer Slide Time: 25:50)



Now, we will take another example where strings are over a and b. So, we will take an example where we have got this alphabet set as a b and such that any string that contains a a will be followed immediately followed by b b. So, in the previous example, that we took. So, was single a was followed by single b here any pair of a is will be followed by pair of b. So, you have got this type of strings a b a b a, but if you have two as then immediately there will be two bs again you can have a b you can have something like this, but as soon as you have two as then again two bs should appear. So, the language should be like that.

So, the production rules that we have for this particular grammar is like this S producing AB or C or A produces b A or a then B produces a b b S or b s or epsilon and C produces b c or epsilon ok. So, a c is broken up into two parts a and b such that. So, having the first this a part is having the first occurrence of a and then b is now the remaining part and this b must be giving me this thing it must start with the say next a and then b b s or so, this the S is broken also if this is the whole string. So, it is broken at the position where we have got the first a.

Now, the situation may be that after this a a b is occurring. So, then that is fine, but if it happens that an a is occurring then there must be two b. So, this so, this part we are taking as A so, this part we are taking as B. So, up to this much is taken as capital A and this part is taken as capital B. So, if it is after if in capital B we have got this thing. So, if

it if there is an a then this will be b b S on the other end or it may be that there is a single a so, there is no a.

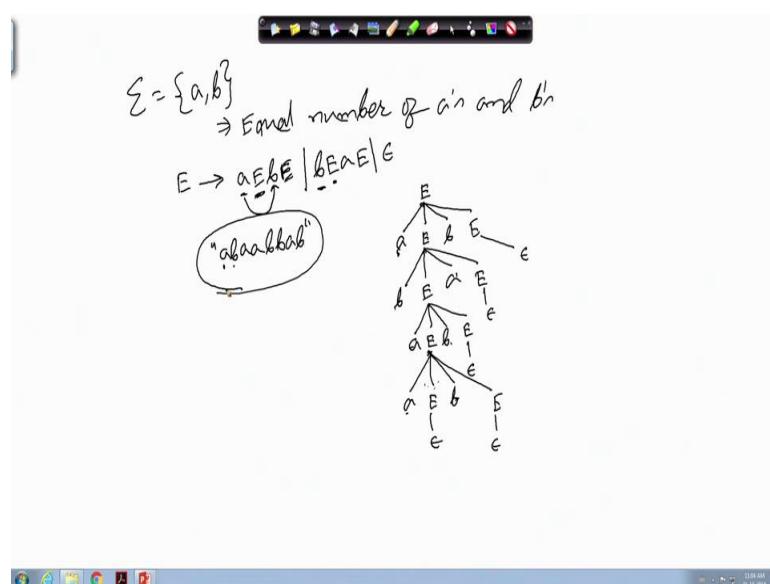
So, there was a single a at this point taken care of by this A rule and then it is simply another b can come and then S or epsilon and C can give the other part so, b c or epsilon. So, this is basically the strings of bs only bs that can come in. So, that is taken care of by this. So, this way this grammar is slightly complicated it requires some thinking; however, you will be you can see like how this grammars can be written. So, it requires some practicing so, you can just try out some more language specifications and write down the corresponding grammar.

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

**Lecture – 23**  
**Parser (Contd.)**

So, next we will be looking into some more examples of this grammars and how to construct the Predictive Parsing table for that.

(Refer Slide Time: 00:28)



So, the next example that we look into the sigma set the alphabet set is having the symbols a and b. And then we would like to construct language the grammar for the language such that it has got equal number of a's and b's in it. So, the language is having all the strings that have equal number of a's and b's ok

So, for that purpose, so you can write down the grammar like this. So, if I have that non terminal E, it produces a E b E, or b E a E or epsilon. So, here by E we represent the strings that have got equal number of a's and b's. Now intuitively you can understand that the string can start with a, or can start with b. So, if it starts with a then the situation is that there should be a matching b which correspond to that ok.

So, in between we have got some string. So, that have got equal number of a's and b's and then after that the next b coming matches with this a. And after that we can again

have a substring that has got equal number of a's and b's or the string can start with a b and then there can be substring having equal number of a's and b's then the character a. So, that the symbol a so that this b matches with this a and then we again have a substring that has got equal number of a's and b's in it or epsilon. So, this is a possible grammar for that.

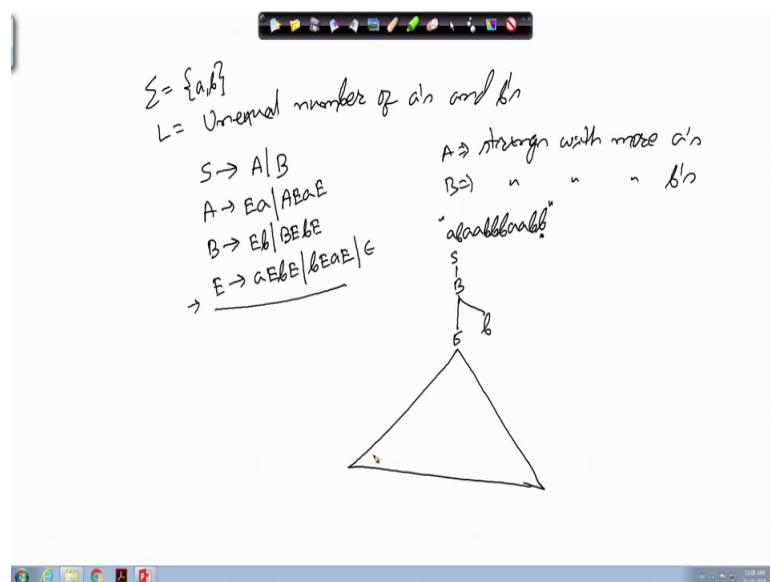
Now, let us see how can we make a string out of that how can we have an example for that. Consider the string say a b a a, then b b a b. So, it is expected that we will have this string should be accepted because there are 4 a's and 4 b's. So, the way the parse tree can be constructed is since its starts with a, so we have to have this left most symbol as a. So, you have to follow this rule b and E.

So, we have got this matching a's and b's. So, this a is here and this b is, this b is this one. So, this way we can have it like this or we can so we can match this b is some so somewhere else also. So, let us see that one possible parse tree may be like this. So, we make it like b E a E. So, that this a matches with this b and this E gives me epsilon and then this E gives me again equal number of a's and b's that part. So, this is a E, and b E where this E gives me epsilon, and this E gives me also epsilon.

So, we have got a b a then again a then. So, this is this E can give me epsilon ok, so number of a's we have is 1, 2, 3, 4. So, here we have got how many a's 1, 2, 3 a's. So, there should be one more a. So in fact, this E so this E should be giving me another derivation should be there. So, they should I have to follow another break here. So, this is a E, b E and then this a. So, this E should give me epsilon this a should give me epsilon.

Now, we have got 4 a's 1, 2, 3, 4 a's and there are 4 b's 1, 2, 3, 4 b's are there. So, that way so this parse tree can give me a proof that this particular string is a belonging to the language having equal number of a's and b's in it. So, that is one example. So, next example that we will take is where it is just the complement of the other one.

(Refer Slide Time: 05:35)



So, the sigma the alphabet set remains like a b, but the language that that will have unequal number of a's and b's; unequal number of a's and b's. So, it is just the complement of the previous one; so, for doing this so the grammar that will construct will be something like this. So, we will take the S as the start symbol of the grammar. So, this gives me S equal to A or B; where A stands for the situation where we have got more a's than b's in the string. So, A this will represent strings with more a's and B will represent the situation with first strings with more b's ok.

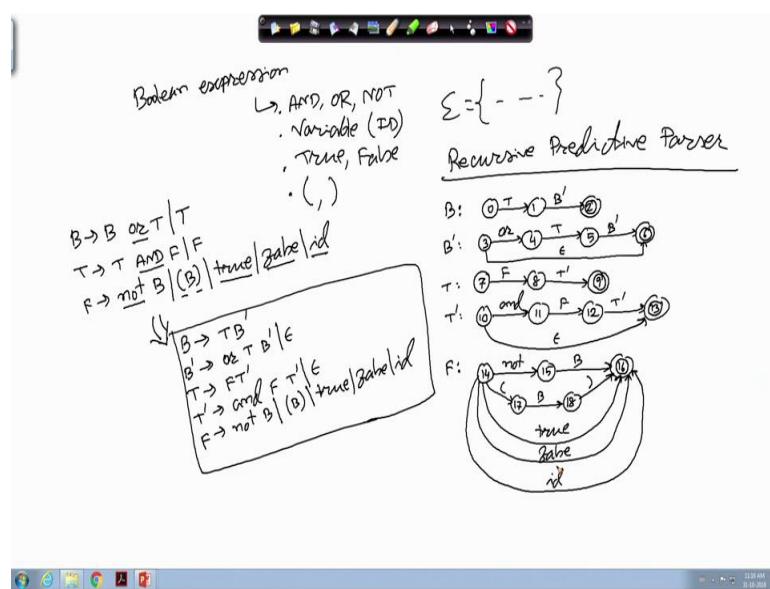
Now, so for A the derivation can be E a, where E is the strings having equal number of a's and b's or it may be A E a E ok. So, it may be equal number A and B followed by A that is we have got one more a here, or it is having more number of a's more number of a's and b's then equal number then one more a and then equal number of a's and b's, so one extra can come.

Similarly the B can be written in a similar fashion like E b or B E b E, where E is what we are written previously having equal number of a's and b's. So, this is a E b E, or b E a E or epsilon ok. So, this is the grammar you can try to derive some string that has got unequal number of a's and b's in it. So, let us try to see the derivation for the string say a b, a a, b b b a a, b b. So, this has got a's 1, 2, 3, 4, 5 and there are 1, 2, 3, 4, 5, 6 b's. So, there are 5 a's and 6 b's. So, they should be acceptable by the grammar ok. So, let us see how it is the how the derivation is done. So, S gives S can give a.

So, it has got more number of b's than a's. So, it should follow S producing B and then this S producing B and then we can allow this rule like it can have equal number of a's and then b's. Then this equal number of a's equal number of a's and b's, so this is last b is has been derived. So, this equal number of a's and b's so that can be derive from here. So, this is the E part.

And now we can just follow this particular derivation to get it, so the way we have done in the last example ok. So, you can same set of rule has been used here. So, you can see that we can have a derivation tree below this that will give me the string that has got equal number of a's and b's in it. So, this way we can construct the corresponding parse tree and giving the proof that this is having equal unequal number of a's and b's in them. So, next we will be taking one example which is towards a predictive parsing and it says that it is the you have to construct the grammar for Boolean expressions.

(Refer Slide Time: 10:01)



So, Boolean expression so it can have the operators like AND, OR, and NOT. It can have some variable or identifier and there are some constant values true and false, true and false and they definitely there can be parenthesis. So, there can be open parenthesis and close parenthesis. So, these are the various symbols that we can have in the terminal set for the Boolean expression. So, this sigma has got all these symbols in it.

Now, we can try to construct the corresponding grammar. So, the grammar for Boolean expression will be like this [laughing] B producing B or T or T; T giving T AND F or F;

and F can give me not of B or within bracket B or true, false or I d where these are all terminal symbols, OR, AND, NOT open parenthesis close parenthesis true false id.

So, these are all terminal symbols and B T and F they are non terminals. So, if you try to construct the predictive parsing or say recursive descent parsing in general the top down parsers then the first thing that I have to do is to convert it into a grammar. So, that there is no left recursion.

So, here we have got left recursion. So, that left recursion has to be eliminated. So, the left recursion can be eliminated by applying the that set of rules that we had done previously, B producing T B dash, B dash producing or T B dash or epsilon, T producing F T dash. And then T dash producing and F T dash or epsilon, where F there is no left recursion for F so this remains unaltered not of B within bracket B or true or false or id. So, this is the grammar that we have got finally, for the top down parsing.

So, next we will be look into that recursive version of the predictive parser, recursive predictive parser. And for this recursive predictive parser we know that we need to make a set of transition diagrams. So, the transition diagram for B will be like this. So, this is the state number 0, from here getting T it comes to state number 1, from here getting B dash it goes to state number 2 and that is a final state.

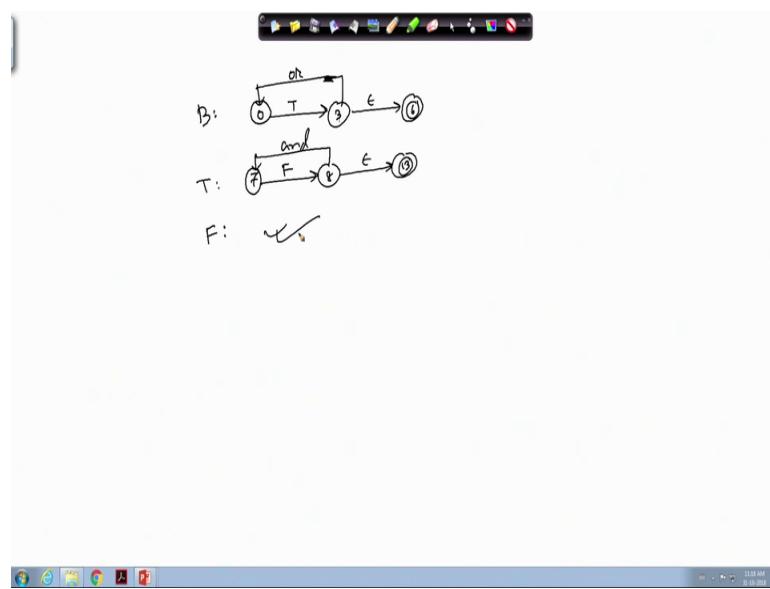
Similarly, for B dash we can have the transition diagram like this. So, up to 2 we have done. So, this will start with 3 on getting or it will come to state 4, and getting T it will go to state 5. And from here getting B dash it will go to state 6 and there is an epsilon production. So, we will have another epsilon transition added like this. So, this is the epsilon transition.

So, that is about these two things that is we have got this B and T next for T dash for T dash we can have F T dash. So, this will start at state number 7 with F it will come to state number 8 and on T dash it will come to state number 9 and that is a final state for T dash. Then we have got so this is for T sorry this is for T. Next we will do it T dash, so T dash is and F T dash. So, that is state number 10 from 10 on and it will go to state 11 and then F it will go to state 12 and then T dash it will go to state 13 and 13 happens to be a final state and then we will have a epsilon transition from 10 to 13.

And finally, we have got F we have got F and then this F will start at state number 14 and from 14 we can have on not it goes to state number 15. And from 15 on B it will go to state 16 and 16 is a final state. Then this open parenthesis so this will take it to state number 17, then B it will go to state number 18 and then close parenthesis. So, it will take it there.

Then true false and id, so these transitions are to be added. So, this is for true, this is for false, and this is for id. So, you have constructed the transition diagrams for this particular grammar. Now we can use these transition diagrams for doing the parsing job. However, we can do some simplification also as we have done previously with other examples. So, this after if we do the simplification as we did with the E T F grammar. So, if we do the simplification then we will get a refined set of transition diagrams.

(Refer Slide Time: 17:50)



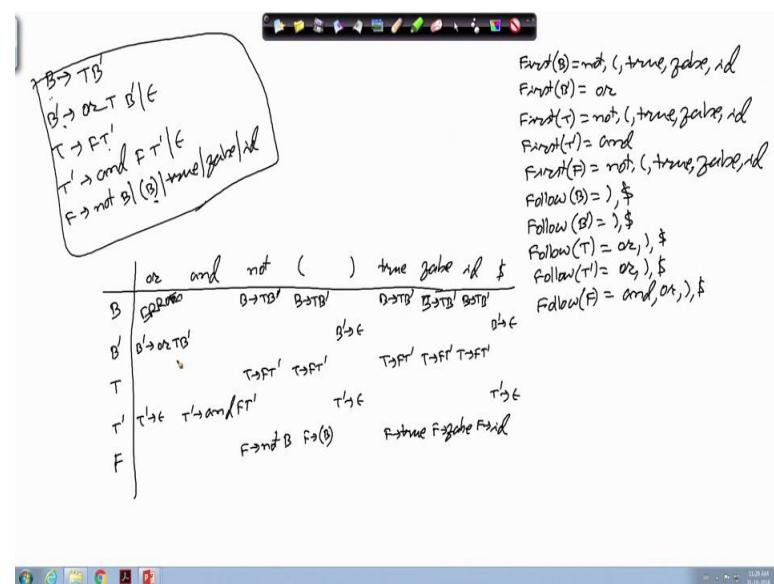
And I am just drawing the final set of transition diagrams you can just check it that is that B transition diagram will be something like this on from state 0 on T it will come to state number 1. And from state number 1 so this will this is actually this is a state number 1 will get eliminated and what will remain is the state number 3 ok.

And from here on or it will come to state 0, this is on or and then on epsilon it will go to state 6, for T the transition diagram will be something like this. So, it will start at state number 7, on F it will go to state number 8, from state 8 on getting and it will come back

to state 7. And if it gets epsilon then it will go to state 13 fine. And the F diagram will remain unaltered. So, whatever we had previously so the F diagram will be remaining as it is. So, this is the final set of diagrams that we will have for these things ok.

So, next we will be trying to make the predictive version, non recursive predictive parsing version. So, that will be based on this 11 philosophy, so that 11 predictive parser. So, we will try to make so for that the first thing that we need to do is to write down the fast and follow sets ok.

(Refer Slide Time: 19:43)



So, for the sake of our understanding; so, I am just writing the grammar once more. So, I have got B producing T B dash, then B dash producing or T B dash, or epsilon, then T produces F T dash, then T dash produces and F T dash, or epsilon. And this F has got not of B within bracket B true false and id. So, this is the grammar that we have. So, first thing that I have to do is to compute the first and follow sets for the individual non terminal symbols. So, the first of B first of B is equal to first of T, the first of T is equal to first of F and first of F has got not then open parenthesis true, false, id.

Now, first of B dash, first of B dash so; this will be having only or the first of T first of T equal to first of F. So, this is the same set that we have written previously true, false, and id. Then first of T dash; first of T dash will have and in it and this first of F will have all these rules all these symbols, not open parenthesis true, false and id. Now, what about the follow sets? So, follow of B equal to so follow of B you see that B can be followed

by close parenthesis. So, it is close parenthesis is there and B is the start symbol, so dollar is there. So, what about follow of B dash follow of B dash?

So, if you have this rule B producing T B dash. So, by the second rule you know whatever is in follow of B will be in follow of B dash. So, follow of B has got close parenthesis and dollar. So, this will also have close parenthesis and dollar then follow of T follow of T. So, by this rule whatever is in first of B dash will be in the follow of T.

So, first of T dash has got or, so or will be there and then follow of t. So, since so by so follow of T by this rule B dash can give me epsilon. So, whatever is in follow of B will be in follow of T, because B dash can give me epsilon. So, follow of B has got open close parenthesis and dollar. So, those two symbols will be there close parenthesis and dollar.

So, similarly you can find out that follow of T dash will be there equal to or close parenthesis and dollar. And follow of F follow of F so it is equal to first of T dash by this rule or this rule. First of T dash has got and so and can be there then follow of F will have. So, whatever is in say not of B, so whatever is in follow of B will be in the follow of whatever is in follow of B will be in the follow of whatever is in follow of F will be in follow of B. So, that is so whatever is in follow of B can be in the follow of F.

So, by applying the rules so you can see that we can see that this and can be there then this or actually by this rule. So, you can so, whatever is in first of T dash we have taken it here and whatever you have what by this rule whatever will be in the follow of F will be in the follow of B. So, that way so and by this 1, so this T dash producing and F T dash. Now you see that T dash can give me and so that and has come in the follow of F. And the other symbols that can come in the follow of F is basically this or then this close parenthesis and dollar. So, these symbols can come in the follow set of F.

Now, once we have constructed the first and follow sets. So, we can try to make the corresponding parsing table. So, the parsing table that will be making it will have entries like B, B dash, T, T dash and F on the on the as per as the rows are concerned. And on the columns will have or, and, not, open parenthesis, close parenthesis, true, false, id and dollar.

Now, we have to apply individual table construction rules and see like what can go which, what can go where. So B producing T b dash, so whatever is in follow of T first of T B dash there I have to apply with this particular rule and first of T B dash is equal to first of T. So, not then this open parenthesis, true, false, id, so they will have this rule. So, B producing T B dash, this rule will come here then it will come here also ok. And B producing T B dash it will come there also ok.

So, then we can have say this rule. So, if we if we say the next see the next rule B dash producing or T B dash. So, or T B dash the first is having or, so B dash or will have this rule that B dash produces or B or T B dash. So, this rule will be there now B dash producing epsilon is there. So, whatever is in follow of B dash there I have to add this particular rule. So, B dash follow set has got this close parenthesis. So, here I should have B dash producing epsilon and here I should have B dash producing epsilon ok.

So if you do this way then for T producing F T dash. So, first of F first of F has got all this symbols. So, there I should put this T producing F T dash. So T producing F T dash comes here then open parenthesis. So, T producing F T dash comes there, then this true, false, and id. So T producing F T dash T producing F T dash T producing F T dash. So, all of them come at this place and then you have got T dash producing epsilon.

So this T dash producing T dash T T dash T dash producing and F T dash. So, this T dash, so T dash and so this should have this rule T dash producing and F T dash and T dash producing epsilon in the follow set of T dash we have got or. So there I should put this rule T dash producing epsilon then close parenthesis. So T dash producing epsilon and dollar.

So there also I should have T dash producing epsilon then F producing not B. So this rule should be added here F producing not of B, then F producing within bracket B. So this should be added here because the corresponding first set has got this thing has got the open parenthesis. Similarly, F producing true should be added here, F producing false should be added here in the false and F producing id should be added there ok.

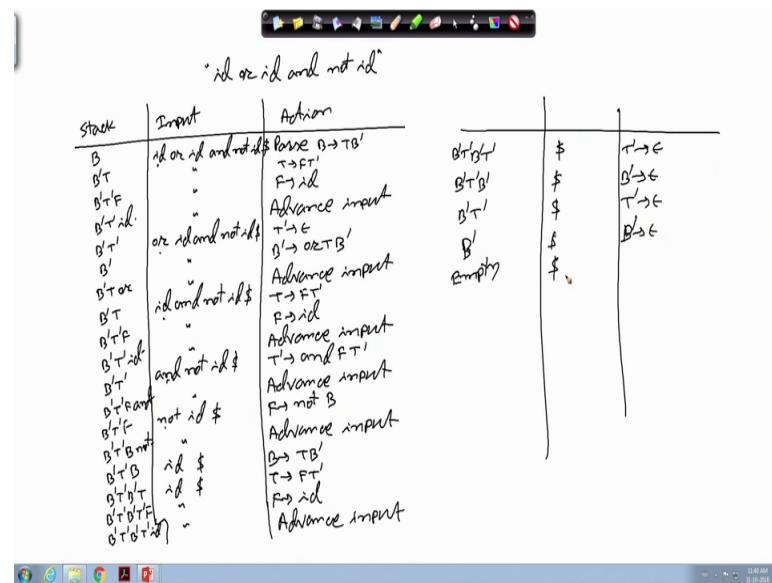
So this completes the table formulation and whatever entries are undefined. So they are parsing errors. So they are all error entry so all these are errors ok. So you can formulate parsing tables like this and from there we can go to the parsing process to see how this parsing can be done for different symbol not a string.

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

**Lecture – 24**  
**Parser (Contd.)**

So, next we will be taking on Parsing example of this Boolean expression.

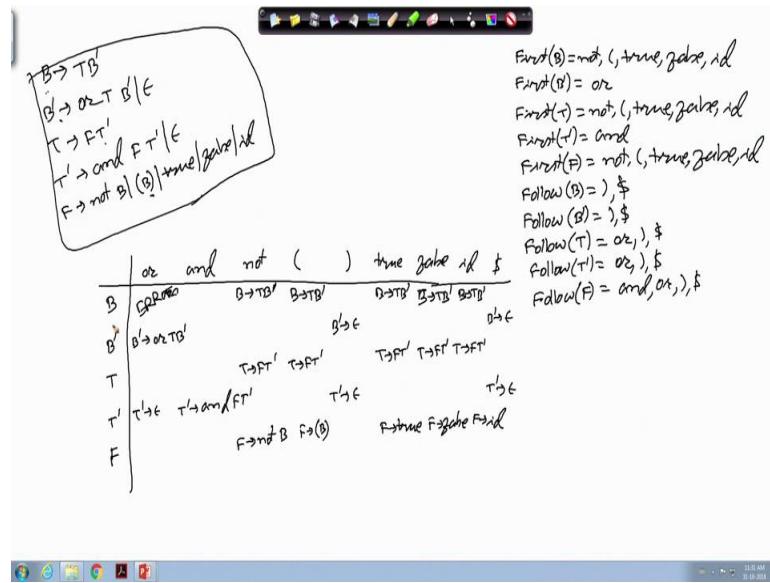
(Refer Slide Time: 00:25)



Stack	Input	Action	
$B$	"id or id and not id\$"	Parse $B \rightarrow TB'$	
$B'T$	"	$T \rightarrow FT'$	
$B'TF$	"	$F \rightarrow id$	
$B'Tid$	"	Advance input	
$B'T'$	"	$T' \rightarrow E$	
$B'$	"	$B' \rightarrow id$	
$B'or$	"	Advance input	
$B'T$	"id and not id\$"	$T \rightarrow FT'$	
$B'TF$	"	$F \rightarrow id$	
$B'Tid$	"	Advance input	
$B'T'$	"and not id\$"	$T' \rightarrow id$	
$B'TB$	"	Advance input	
$B'TB$$	"	$B \rightarrow not B$	
$B'TC$	"	Advance input	
$B'TBnot$	"not id \$"	$B \rightarrow TB'$	
$B'TB$$	"	$T \rightarrow FT'$	
$B'TB$T$	"id \$"	$F \rightarrow id$	
$B'TB$T$$	"\$"	Advance input	

Suppose we take the exam id or id, id or id and not of id. Suppose this is the Boolean string that is given B or id and not of id ok. Now as we know that this predictive parsing processes it maintains stack and the input. So, we will be writing it like this, so there is a stack part there will be the input as it is there and then the corresponding action part ok. So, initially stack top will contain the start symbol of the grammar B and the input will be the whole string id or id and not id, this whole thing will be there in the input.

(Refer Slide Time: 01:37)



Now we have to see what was the rule for B id? So, if you look into this grammar this table it says B id says you have to go by B producing TB dash by this rule ok; so, by B going by B producing TB dash. So, it says parse B producing TB dash and they will be put into the stack. So, B is popped out from the stack and the stack will now contain B dash and T and input remains unaltered same as this one.

Now, I have to see what is T id? So, T id T id is stilling go by T producing F T dash ok. So, it will be so it says that parse by T producing F T dash. So, this T is going out of the stack now the stack will have B dash, T dash and F input remains unchanged. So, B dash T dash F now stack top is F and the input symbol id ok. So, F id you see F id says that you go by F producing id. So, it says that you go by F producing id and as a result it will be B dash T dash id, this remain same and then this. So, these two ids will now match this and this id will match. So, naturally the corresponding action is advance input pointer.

So, now the situation that we have is B dash T dash on the stack and or id. So, there is a dollar at the end we input is assumed to be ended by a dollar and not and dollar, so that is the situation. Now I have to see what is T dash or. So, T dash or is T dash producing epsilon. So, it says that the corresponding thing is T dash producing epsilon. So, T dash will go out of the stack, so now the stack will become B dash and input will be like this only.

Now B dash or so B dash or is telling me go by B dash producing or TB dash. So, it says the action is B dash producing or T B dash. So, they will be put into the stack. So, this B dash goes out and this B dash comes in, so B dash T or input remains same. So, now these 2 ors match, so we advance the input advance input. So, this or goes out now the stack contains B dash T and this has got id, input has got id and not id dollar fine.

Now, T and id, so T and id you see T and id says go by T producing F T dash this rule. So, it says go by T producing F T dash. So, this T goes out of the stack and T dash and F they are put into the stack input remains unchanged. Now F id, F id is we have seen previously that it will tell me to go by this rule F producing id. So, F will be popped out from the stack and this id will come in input will remain unchanged. Now these 2 ids these 2 inputs will these 2 ids will match. So, the action will be advance input it will be advance input. So, this stack will become B dash T dash and the input will now have and not id dollar.

Now, I have got T dash and now T dash and it says it go by T dash producing and F T dash. So, it says go by T dash producing and F T dash. So, this T dash is popped out from the stack then this new T dash F and they are put into the stack input remains unchanged. So, again these 2 ands match, so we have to advance input we have to advance input. So, it becomes B dash T dash F and then this and is has been consumed. So, it is not id dollar.

Now, F and not F and not says you go by F producing not B. So, it says go by F producing not B. So, this F is taken out from the stack. So, B dash T dash B and not input is as it is. Now these 2 not's will match, so it will be advancing input; it will be advancing input. So, this will become B dash T dash B and then it says that I have got id and dollar fine.

Now, B id, so B id says that go by B producing TB dash this rule. So, it says go by B producing T B dash. Now so this B will be going out, so the stack will contain B dash T dash B dash T and then this is id and this is dollar. Now T id so, T id says go by T producing F T dash ok. So, it says go by T producing F T dash.

So, if we do that then the stack content will become something like this. This T goes out and this T dash F comes to the stack and this remains the input remains unchanged. Now I have got F and id and F and id will tell me to go by F producing id. So, this will be

modified to B dash T dash B dash T dash id input will be this one. So id, id will match so I will be advancing the input advance the input pointer.

So, my new stack will be B dash T dash B dash T dash this will be the new stack and here the input will be dollar. So, T dash dollar, so T dash dollar says go by T dash producing epsilon. So, it says go by T dash producing epsilon. So, T dash will go out now the stake will be B dash T dash B dash and dollar and this will take this is a B dash dollar tells me that go by B dash producing epsilon. So, this goes by B dash producing epsilon, so this B dash goes out. So, I have got B dash T dash and dollar.

Now, again T dash dollar will tell me T dash producing epsilon. So, this says you go by T dash producing epsilon. So, T dash will go out now I will have B dash and dollar and here it says B dash dollar says go by B dash producing epsilon. So, go by B dash producing epsilon, so this B dash goes out. So, stack is now empty. So, this stack is now empty and this has the input pointer is also at dollar; that means, the given string is a valid string ah. So, by using this predictive parsing method, so we can construct the corresponding parse tree.

So, this way given a grammar, so you can first construct the first the first and follow sets and from there you will be able to construct the parsing table and once the parsing table is made, so this parsing process is automated. So, you whether to check whether a given string belongs to the language or not? So, you can have this parsing algorithm. So, which will be run and then you can find out whether it is input is come in to dollar input entire input is consumed and the stack is also empty.

So, if you a come if you can come to that configuration staring with the configuration that stack has the start symbol in it and the entire input is there with the pointer at the beginning of the string. So, if you can come to that configuration then you can be you can tell that the string is accepted by the language. So, next we will be looking into a new topic which is known as bottom up parsing. So, bottom up parsing, so this top down parsing that we looked into. So, this actually try to construct the parsing parse tree starting with the start symbol of the grammar and from there it was trying to go in a top down fashion to derive the final string. So, another approach that we can have is that we can start constructing from the bottom. So, starting with the input, so we try to

constructing from the bottom and then we can ultimately merge onto converge onto the start symbol of the grammar.

Now, the problem with the top down parsing was that so you have to predict like at some point of time looking at the next input symbol. So, if it is 111 grammar, so looking at the current input symbol only you have to tell like which rule to follow that sometimes becomes difficult. And if you are if you try to modify that look ahead if you want to make it 11k then the parser will become very complex.

So, that way bottom up parsing is in some many cases bottom up parsing will be better because it will be able to construct from the bottom so it knows the input string that is there and it tries to construct from there. So, that way many many a time you will find that for certain grammars. So, you will be able to formulate the bottom up parsing algorithms, but not the top down parsing. So, the parsing table you can you can construct for bottom up parsing but not for top down parsing. So, that makes bottom up parsing more powerful than the top down parsing.

(Refer Slide Time: 12:49)

The slide has a yellow header bar with the title 'Introduction'. Below the title is a bulleted list:

- Constructs parse tree for an input string beginning at the leaves (the bottom) and working towards the root (the top)
- Example:  $id^*id$

On the right side of the slide, there is a parse tree diagram. The root node is labeled  $E$ . It branches into  $T$  and  $T$ . The left  $T$  branches into  $F^*$  and  $F$ . The  $F^*$  node branches into multiple  $F$  nodes, each of which further branches into  $id$  nodes. The right  $T$  branches into  $F$  and  $F$ . The  $F$  node branches into  $T^*$  and  $F$ . The  $T^*$  node branches into multiple  $T$  nodes, each of which further branches into  $F$  nodes, which then branch into  $id$  nodes. The  $F$  node branches into  $T^*$  and  $F$ . The  $T^*$  node branches into multiple  $T$  nodes, each of which further branches into  $F$  nodes, which then branch into  $id$  nodes.

Below the parse tree, the following grammar rules are listed:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

The footer of the slide features the 'swayam' logo and other educational icons. A small video window in the bottom right corner shows a person speaking.

So, to start with this bottom up parsing strategies to the construct parse tree for an input string beginning at the leaves that is at the bottom most level of the parse tree. So, parse tree at the leaf level it has got only the terminal symbols that is the given the input string. So, it will let us start with that bottom level thing and then try to construct the parse tree and finally, reaching the root so which is supposed to be the start symbol of the grammar.

So if you can do this, so if you can have a if you can show that the entire string can be reduced to the start symbol of the grammar then we say that the string is accepted by the language. So, an example suppose we have got that expression grammar which is given by this E producing E plus T or T T producing T star F or F and F producing within bracket E or id. And we try to see whether id star id is a valid string of this language or not.

So, we will do it in a bottom of fashion. So, first id will be replaced by F ok. So, we get this part then this F can be replaced by T ok. So, this is we have got T into id then this second id is replaced by F and then this T into F. So, that part is replaced by F then T into F is that F F is replaced by T it should be F should be replaced by T here and there is a jump step jump at this point. So, ideally they should not be like this. So, this E should give me T and T should give me F because E cannot give me F. But anyway so this is (Refer Time: 14:45) are make the space less. So, it has been shown like that, but they should be ideally like this E to T to F.

So, you see starting with the input string. So, we can construct the entire parse tree and go to the root the start symbol of the grammar. So, if you can do this thing then we say that we have got a bottom up parsing strategy for the grammar.

(Refer Slide Time: 15:15)

**Shift-reduce parser**

- The general idea is to shift some symbols of input to the stack until a reduction can be applied
- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply
- A reduction is a reverse of a step in a derivation
- The goal of a bottom-up parser is to construct a derivation in reverse:
  - $E \Rightarrow T \Rightarrow T^* F \Rightarrow T^* id \Rightarrow F^* id \Rightarrow id^* id$

FREE ONLINE EDUCATION  
**swayam**

So, next so they are so this bottom up parser. So, they are also known as shift reduce parsers. Why it is named? Because these parsers they work on two operation mainly on

two operations one is called a shift operation, another is called a reduce operation. So, that is a general idea is to shift some symbols of input to the stack until we can find a situation where reduction can be applied.

So, we go on shifting some input symbols on to the stack and then try to replace that part replace some part of the stack by applying some reduction. And each and in each reduction step a specific substring matching the body of a production is replaced by the non terminal at the left hand side of the production.

For example, if I have got a in the stack if you find that you have got say T star F available in the stack and in the grammar there is a rule like E producing T star F. Then what we do? So we can replace this part of the stack by the symbol E, so that is exactly what is said here. So, we have at each so that is a reduction. So, we reduce some part of the stack by a non terminal.

So, whatever comes on the left hand side of the rule by which we are doing the reduction, so the stack now will now contain that symbol. So, stack will now contain that symbol and the portion on the right side. So, they will be they will be used to replace the portion of the stack. So, naturally the decisions the key decisions during bottom up parsing are about when to reduce? And what which production we should apply for reduction? So, like shifting so I can just get the next input and shift it into that the step, but I have to take a decision that at some point of time I will apply the reduction operation also the reduced operation also.

So, when to you apply this reduce? So, if you are not, but doing it correctly then you will be reducing at arbitrary point. So, that it will not be able to generate the parse tree even if the string is correct. So, that so I have to do it judiciously at which point I do the reduction and also there may be several rules by which we can do a reduction ok.

For some for some point of time so it may so happened that if I take set top most 3 symbols there can be reduction. And there can be two different rules by which we can do that reduction or if I look into top most 5 symbol so I can do a reduction. So, that way I have to take a decision like of which production rule we apply for doing the reduction and a reduction is a reverse of a step of derivation.

So, in a derivation we are going in a top down fashion. So, we are starting with the start symbol and then we are deriving the sentential forms till we are at the input stream and in the reduction process. So, this is just the reverse we are starting at the input stream and converting some parts of it into non terminals and ultimately the entire thing reduces to the starts symbol.

So, this is just the reverse of the derivation process and the goal of bottom up parser is to construct a derivation in reverse. Like say actual derivation is say for this id star id the derivation is E to T to T star F to T star id to F to id to id star id. So, but the bottom up parser it will do it in this the direction. So, it will start with id star id then it will find this reduction of id to F, then it will find this reduction of F to T, then it will find this reduction of id to [FL], then it will reduce this T star F by T and then finally, it will reduce this T by e, so that is the thing. So, does it in the reverse direction compare to a top down parsing.

(Refer Slide Time: 19:23)

The slide has a yellow header with the title "Handle pruning". Below the title is a bullet point definition:

- A Handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation

Right sentential form	Handle	Reducing production
$\text{id}^* \text{id}$	$\text{id}$	$\text{F} \rightarrow \text{id}$
$\text{F}^* \text{id}$	$\text{F}$	$\text{T} \rightarrow \text{F}$
$\text{T}^* \text{id}$	$\text{id}$	$\text{F} \rightarrow \text{id}$
$\text{T}^* \text{F}$	$\text{T}^* \text{F}$	$\text{E} \rightarrow \text{T}^* \text{F}$

The footer of the slide includes the Swayam logo and the number 43.

There is a concept called handle pruning. So, what is a handle? So a handle is a substring that matches the body of production and whose reduction represents one step along the reverse of a rightmost derivation. So, for example if I have got say this particular right sentential form  $\text{id}^* \text{id}$ , then this  $\text{id}$  can be a handle because there is a production rule which says  $\text{F} \rightarrow \text{id}$  and this right hand side matches with this  $\text{id}$ . So, this  $\text{id}$  part is a handle.

Similarly this F producing F F into id then this F is a handle because there is a rule which says T producing F. So, this id is also can be a handle by because there is a rule like F producing id, but the problem is. So, it will give rise to something like F star F and then it may not be giving the actual a rightmost derivation. So, a handle it is with respect to a string.

So, it is not only that it should be the right hand side of some production, but also it should be part of the derivation process, it should be part of the rightmost derivation process arbitrarily we can take. So, if we can identify proper handles then we will be able to identify the proper derivation in the reverse. So, this rule, so this will be using this LR parsers or this bottom of parser; so, they will be using this handle pruning strategy ok.

(Refer Slide Time: 20:55)

- A stack is used to hold grammar symbols
- Handle always appear on top of the stack
- Initial configuration:

Stack	Input
\$	w\$

- Acceptance configuration

Stack	Input
\$\$	\$

So, this is the general philosophy of a stack reduce shift reduce parsing policy. So, we will be using a stack to hold grammar symbol. So, grammar symbols means the symbol the terminals and non terminal, so both are grammar symbol. So, we will be having a stake that will hold the grammar symbols and as a result handle will appear on the top of the stack. So, if we have it like this, so if this is the stack. So, it has the grammar symbols in it may I have got say a, b, then some e, s then say c, e, so like that again another e like that.

Now, you see the where this capital uppercase letter. So, they are say non terminals and the lowercase letter, so they are say terminals. So, this may be say some d this may be a

like that. So, they are at, so this stack can contain both terminals and non terminals. Now what can be and handle like say this handle it will always appear on top of the stack. So, I can have a handle which stands over so these three entries so a, b, e. So, if there is a production rules like rule like a, b, e then we will say that this is matching the handle.

(Refer Slide Time: 22:29)

**Shift reduce parsing**

- A stack is used to hold grammar symbols
- Handle always appear on top of the stack
- Initial configuration:

Stack	Input
\$	w\$

- Acceptance configuration

Stack	Input
\$S	\$

So, initially the stack will contain dollar and the input will be the entire input string w and dollar. So, initially the stack has got dollar and this stack has got dollar and this input id this is the input string w. So, we will assume that there is a dollar character at the end of it, there is a dollar at the end. So, if the whole parsing root in it will start with this particular configuration.

And then it will be an acceptance it will go to an acceptance configuration if at the end you find that the stack has got dollar S in it and it the input is here the input pointer is here. Initially input pointer was here and after this parsing process, so input pointer has come to the end of the string and the stack contains S followed by dollar. So, if this configuration can be reached.

So, starting with this configuration by taking actions corresponding to the input symbols that you have in w; so, if you can come to this configuration then we say that this is this is a valid string the given string w is a valid string of the language. So, in this way the shift reduce parser server the basic actions are it will sometimes we will shift the next input symbol on to the stack, sometimes it will pop out some entries from stack and

apply a reduction operation on them the reduce operation on then. So, these two things will be doing. So, how do when do you take a shift decision and when do you take a reduce decision. So, that will be depicting the design of the parser or the parsing policy.

(Refer Slide Time: 24:09)

	Stack	Input	Action
• Basic operations:	\$	(id*id\$)	shift
– Shift	\$	*id\$	reduce by F->id
– Reduce	\$F	*id\$	reduce by T->F
– Accept	\$T	*id\$	shift
– Error	\$T*	id\$	shift
• Example: id*id	\$T*id	\$	reduce by F->id
	\$T*F	\$	reduce by T->T*F
	\$T	\$	reduce by E->T
	\$E	)	accept

So, this is a simple example. So, we have got 4 basic operations. So, we have just now we have talked about shift and reduce there are two more operation one is accept and another is error. So, accept is the situation where you have reached a configuration where the stack top contains the start symbol of the grammar, and the input is also input pointer is also pointing to dollar. So, that is the accept configuration.

And in between so if you are in a state where the parser does not know what to do the parsing table or the parsing policy does not tell clearly like what to do in that case that is an error operation. So, error operation so you can have some error message flashed or you can take some error recovery measure, so that the parser can continue parsing reporting the error.

So, let us take the same example the id star id. So, the initial configuration will be like this the stack will be having only dollar and this id star id dollar. So, this will be the input. So, right now we do not know how do we take the shift and reduce decision, but suppose we know that we can we can choose we can make this actions we can make the actions shift and reduce and the choice is done in this fashion as it is shown here.

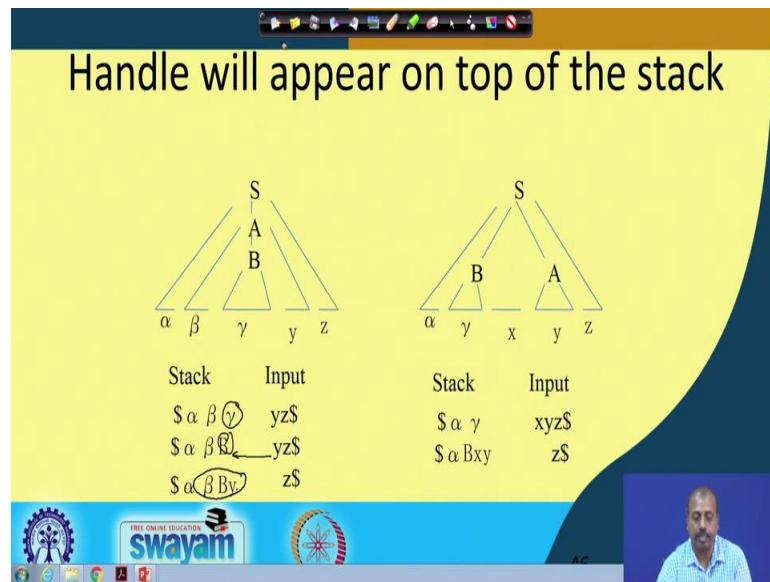
So, suppose we take a decision to do a shift then what will happen this id will come to the stack. Now the stack has got dollar and id with id at the top of the stack and the input will have star id dollar. Then somehow we come to a decision that will be following this reduction rule reduced by a producing id. So, this id will be popped out from the stack and then this left hand side that is F so it will be pushed into the stack.

So, stack now contains dollar F, then looking at a F and star we take a decision that we will be doing a reduction by T producing F. So, this F is taken out from the stack and T is pushed into the stack. Now getting T and T at the top of the stack and star at the next input so, somehow we take a decision that will shift this star into the stack. So, the star is shifted into the stack and then this looking into this star and id this parser takes a decision that I will do a shift operation. So, this id is shifted into the stack.

Now, id and dollar then somehow we takes a decision that I will be following a reduction by F producing id. So, this is id will be replaced by F, then this T star then by F and dollar looking at F and dollar. So, it will see it will take a decision that will reduce by T producing T star F. So, ultimately so this way it proceeds. So, ultimately we are at a configuration where we have got this start symbol E on the top of the stack and dollar at the input pointer. So, at that point we accept the input ok. So, that is the corresponding action is accept.

So, if there was an error in the expression they need some points so we will not be able to have any legal activity. So, at that time so we will flash that there is an error at this point. So, this can come to an error as quickly as possible. And in fact, this bottom up parsing on advantages that. So, it can come to these errors quite fast. But of course, how do you take this decision so that is the question ok. So, we will look into that as we proceed through the discussions.

(Refer Slide Time: 27:49)



Now, handle will appear on top of the stack like say save this y z. So, this is the input; so, this y so this stack has got this thing this alpha, beta, gamma so this is the stack. Now it will find that this gamma. So, gamma will be replaced by B, so this gamma will be replaced by B. So, there so you see that gamma gamma comes at the top of the stack and that has been replaced by B.

Similarly, we have got this in the next step this y is shifted. So, this is B y, so this becomes it become B y and then this B y this beta B y. So, this gives me a handle and this beta B y is replaced by gamma. So, that way it can it can proceed ok. This is just an example how this handles can come at the top of the stack.

(Refer Slide Time: 28:53)

Conflicts during shift reduce parsing

- Two kind of conflicts
  - Shift/reduce conflict
  - Reduce/reduce conflict
- Example:

stmt → If expr then stmt  
| If expr then stmt else stmt  
| other

Stack                      Input  
...if expr then stmt      else ...\$

So there can be conflicts in the shift reduce parsing and there are two types of conflict like one is called shift reduce conflict, another is called reduce reduce conflict. So, shift reduce conflict means looking at the next input symbol and the top of the stack. So, there may be two possible actions. So we can shift it shift the next input or we can apply some reduce rule we can reduce operation following some production rule and if the grammar is such that we cannot identify a single operation.

So both the operations appears to be valid the shift and reduce operation then that is called a shift reduce conflict. Another thing is reduce reduce conflict so here the parser does not know which rule to apply for doing the reduction. So it say it find there some reduction has to be done, but there may be more than one rule which qualifies for the reduction that is the right hand side of more than one rule matches with the handle. Then which rule to follow so that is reduce reduce conflict.

So, typical example is say this one. So statement producing suppose we have got this particular rule so this is the grammar that we have. Now in the stack if we have this situation if expression then statement and in the input I have got else part. Now what to do? So, if you are if so one possibility is that so it will be lead to if expression then statement else statement type of thing so that is this one. So, in that case I should shift this else into the stack. So, that is a shift operation.

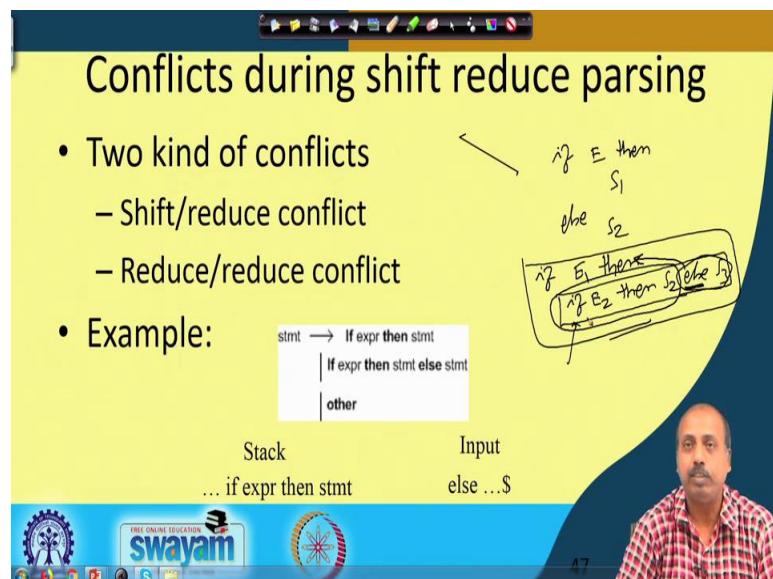
Other possibility is that this. So, this itself maybe the and if then statement. So if we so we may try to reduce we may try to reduce by this rule statement producing if expression then statement. So this is a shift reduce conflict on the input symbol else. If the input symbol is else and the stack contain something like this then it say shift it can give rise to a shift reduce conflict.

So we will see how to take care of this as we proceed through the classes. And this is we will see that there are some of them can be handled or some of them can be resolved by taking help of the precedence of the operators. Some of them can be handled by modifying the grammar a bit and all that thing, but ultimately when you are designing the parser the parser should not have any conflict. So it is the compiler designer's responsibility to do something so that the parser does not face this type of conflicts.

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

**Lecture – 25**  
**Parser (Contd.)**

(Refer Slide Time: 00:15)



Conflicts during shift reduce parsing

- Two kind of conflicts
  - Shift/reduce conflict
  - Reduce/reduce conflict
- Example:

stmt → If expr then stmt  
| If expr then stmt else stmt  
| other

if E then S<sub>1</sub>  
else S<sub>2</sub>

if E<sub>1</sub> then S<sub>1</sub> (else S<sub>2</sub>)  
if E<sub>2</sub> then S<sub>2</sub> (else S<sub>1</sub>)

Stack                      Input  
... if expr then stmt    else ...\$

So, in our last class we were looking into the shift reducing parsing policies and we have seen that there are four actions that can occur with shift reduce parsing, shift reduce, accept and errors. So, these were the four operations that a shift reduce parser will do. Now, many a time we will see that it may happen that there will be conflict in because the as depending upon the grammar the parser may not be able to decide uniquely whether to do a shift operation or a reduce operation. So, that is known as a shift reduce conflict.

So, this is that shift reduce conflict. So, parser cannot decide whether to shift or to reduce and there is another conflict which is known as reduce reduce conflict where we can there are more than one rule by which we may try we may do the reduction. Of course in future what can happen is that as you see more tokens, so maybe one of these reductions are valid.

So, as a result if you do not do enough look ahead. So, there will be conflicts and those conflicts cannot be resolved at the first level it itself. So, that type of situation

will give us reduce reduce conflict. So, if we want to modify the grammar for removing this conflicts that is better if not then we have to take some default action. And the default action for shift reduce conflict is doing a shift and for a reduce reduce conflict the default action is whichever reduction rule comes first in the set of production rules.

So, that will be taken as the rule by which to do the reduction. So, there cannot be any shift shift conflict because in both the cases we are going to shift to the next input symbol. So, there is no problem with that. So, now, there is nothing like shift shift conflict and shift accept or reduce accept. So, this sort of conflicts also cannot occur because there we are already in the accept state. So, there is no further action to be taken

Now, let us take some example and try to see how this shift reduce conflicts it can occur like in this case. So, what we have is say this particular grammar then if else grammar. So, if statement producing a producing if expression then statement or if expression then statement else statement or other statements; now see at some point of time the situation may be like this that in the stack we already have these tokens.

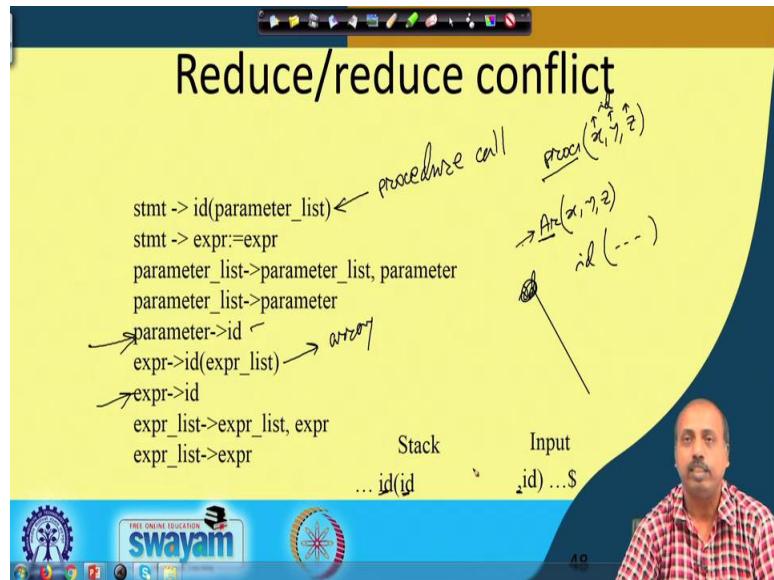
If expression then statements so up to this much we have seen, so in this expression and statements. So, these are non terminals and if and then they are terminal. So, as we know that the stack can contain the all grammar symbols both terminals and non terminals. So, suppose at some point of time this is the situation where statement is at the top of the stack. And then the next input that we have is the else, next token that we have is else.

Now what to do? So, one possibility is that we so this else is a part of this if then statement. So, this else has to be shifted into the step. Other possibility is so it is so there was a nested if. So, there the situation is like this. So, there was a there is a nested if. So, I have got some if expression then S 1 else S 2, so this is one possibility. Other possibility is that if E then say if E 1 then S 1 if E 2 then S 2 else S 3.

So, this is the situation on which there will be a shift reduce conflict because when we see these particular else. Now what to do? So, whether it should be shifted because whether it is giving to it is going to give me this if then else statement or. So, I will reduce up to this, I will reduce up to this, and these else becomes a part of this outer E outer if. So, thus so that is the conflict ok. So in that case the parser will not be able to take a unique decision whether to do a shift or a reduce.

And as I said that default action is to do shift. So, if you shift it then what happens is that this else S 3 so, this becomes a part of the inner most if. So, and most of the programming languages they also tell that way that the else is always associated with the inner most if and in that case shift is a valid action. So, that is one example of shift reduce conflict. Next we will be looking into the reduce reduce conflict.

(Refer Slide Time: 05:01)



So, like this suppose I have got a grammar that has got both that has got both so, this is a procedure call. So, this is a procedure call and then this may be some array list, this may be an array. So, if there is a procedure call so say proc 1 there I can pass this parameters say x, y and z maybe it has got 3 parameters. Now there may be another array Ar and there also I have got the, I have got the arguments or the array indices they are also expressions. So, that is also say x, y, and z some expression.

Now, in so this x, y, z as far as tokens are concerned so, all of them will be taken as id. So, in one case you have got this situation that id followed by id followed by ideally I id followed by this parameter list where this parameter list is again an expression. So, this is parameter list giving parameter and the parameter is ultimately giving id, so ultimately it is giving id comma id like that. And other situation is that we have got this array and that after that array also we have got this array name is coming as id. And then we have got this expression list which is the array indices. So, these array indices so they also come as id.

Now, if you are at a situation like this. So, we have seen id within bracket id and the next symbol is a next token is a comma. Now what to do? So, there can be one possible reduction like this by this one parameter giving id and there is another reduction expression giving id. So, which one to do we really do not know. So, until and unless we know that very recently we have seen a procedure call then in that case this reduction should be by this.

And if you on the other hand if you assume that very recently we have seen one array call array array name as the portion before the open parenthesis then we know that this is going to be an expression list. So, you should do go by expression producing id. So, there is a confusion so just by looking into this top of the stack and this comma you cannot take a decision by which rule to do the reduction. So, this gives rise to reduce-reduce conflict.

So, these conflicts are to be avoided because if a grammar has got this type of conflicts. Then in the parsing process then the parser will not be able to proceed properly and there will be difficulty in parsing the input sequence. So, default rules are there, but it is not mandatory that default rules will always be applicable. So, depending upon the language, so they may not be applicable also. So, we have to be very careful about these conflicts. So one of the basic responsibilities for this parser designer is to modify the grammar so that these conflicts can be resolved.

(Refer Slide Time: 08:19)

The slide has a yellow background and a dark blue header bar. The title 'Bottom-Up Parsing' is centered in the header. Below the title, there is a bulleted list of parsing techniques:

- Operator Precedence Parsing
- LR Parsing
  - SLR
  - Canonical LR
  - LALR

Handwritten notes are present on the slide, specifically pointing to the 'LR Parsing' section:

- A handwritten arrow points from 'LR Parsing' to 'SLR'.
- A handwritten arrow points from 'LR Parsing' to 'Canonical LR'.
- A handwritten arrow points from 'LR Parsing' to 'LALR'.

At the bottom of the slide, there is a blue footer bar with the 'swayam' logo and other navigation icons. A small video window in the bottom right corner shows a man speaking.

So we will be looking into two types of bottom up parsing strategies one is known as operator precedence parsing and another is a class of parsers known as LR parsers out of these two operator precedence parsing this is very simple. So, for a particularly for language that just accept expressions; so, basically the expressions expression grammar, so they can be parsed using this operator precedence parsing.

So, we will see there are certain rules that will define what is an operator grammar and all. And in general other parsing method so we have got this LR parsing and this LR parsing we will see that it can further be divided into number of categories. We will look into something called SLR or simple LR parsing then we will look into something called canonical LR canonical LR, so which is more generic in nature.

So, SLR is pretty simple out of these three alternatives that we have in error parsing SLR is going to be pretty simple and many a times. So, we can we can construct the parser by hand. On the other hand this canonical LR. So, it is difficult to construct by hand and it has got a large number of states compared to an SLR parsers. So, it has got a large number of states.

On the other hand this a LALR there is another parser known as LALR which is a full form is look ahead LR. So, this parser, so this will have less number of states in fact, the number of states that LALR parser will have is same as the number of states that you have in SLR, but it can be more powerful than the SLR, so that way it is better.

So, most of the automated tools that we have that we have talked about previously like Yak, Bison etcetera; so, they generate LALR parser for a grammar; however, LALR is difficult to learn for our class. So, we will be we will be first learning SLR and then go towards the other categories. So, let us start with this operator precedence parsing.

(Refer Slide Time: 10:31)

Operator Grammar

- No  $\epsilon$ -transition
- No two adjacent non-terminals

Eg.

$$E \rightarrow E \text{ op } E \mid \text{id}$$
$$\text{op} \rightarrow + \mid ^*$$

The above grammar is not an operator grammar  
but:

$$\underline{E \rightarrow E \pm E \mid E^* E \mid \text{id}}$$

So, operator precedence parsing is applicable for operator grammars. So, a grammar will be said to be an operator grammar. If it does not have any epsilon transition and in no production rule right hand side you will have two adjacent non terminals like say this particular grammar, so  $E$  producing  $E$  of  $E$ . So, you see this operator so, this is this operator we can modify. So, this grammar whether it is an operator grammar so it is not an operator grammar because you see this  $E$  and  $op$ . So, they are two non terminals. So, they are appearing side by side. So, this is not an operator grammar. So, this is not operator grammar

However we can modify this grammar a bit we can modify this grammar a bit and we can write it like this. So if we substitute this  $o p$  in the first rule, so you get a grammar like this and here you see we do not have this situation that is two adjacent non terminals so that condition never occurs. So, it is always separated by a terminal symbol. So, this is an operator grammar and also it does not have any epsilon transition. So, this is an operator grammar. So, this is fine. So, we can use this grammar for operator precedence parsing.

So, once given a grammar you first check whether this is an operator grammar or not if it is not an operator grammar we have to check by doing some simple modification to the grammar is it possible to convert it into an operator grammar. So, if we can do that

then we can try to frame the operator precedence parsing table and follow the operator the operator. We can follow the operator precedence parsing.

(Refer Slide Time: 12:25)

The slide has a yellow background with a blue header bar at the top containing various icons. The title 'Operator Precedence' is centered in a large, bold, black font. Below the title, there is a bulleted list of three items:

- If a has higher precedence over b;  $a \rightarrow b$
- If a has lower precedence over b;  $a \leftarrow b$
- If a and b have equal precedence;  $a \doteq b$

Below the list, the word 'Note:' is written in black. To the left of the note, there is a handwritten-style diagram showing two horizontal lines with arrows pointing from left to right. The top line has an arrow above it, and the bottom line has an arrow below it. To the right of the note, there is a list of three points:

- id has higher precedence than any other symbol
- \$ has lowest precedence.
- if two operators have equal precedence, then we check the **Associativity** of that particular operator.

At the bottom of the slide, there is a blue footer bar with the 'swayam' logo and other navigation icons. On the right side of the footer, there is a small video window showing a man with a mustache wearing a red and white checkered shirt.

So, what do you mean by precedence of operators? So, from our mathematics classes we know that whenever we have got some operator, so there are some precedence. For example, addition and multiplication out of that in general multiplication has got the higher precedence than addition. So, in case of grammar so we will be talking about precedence between the terminal symbols; so, suppose a and b they are two terminal symbols. If a has higher precedence over b we will denote it like this. So, this is just a notation. So, we will read it as a a a has higher precedence than b.

Another possibility is if a has lower precedence over b. So, we will be writing as a less than then a dot then b made and we will read it as a has lower precedence than b. And if a and b are of equal precedence then we will be writing like this a with a dot on the on top of an equality sign and then b. You see that many a times for our for the sake of simplicity we will simply write simply say a greater than b or a less than b like that or a equal to b.

But in general we will be we will be following we will be meaning this thing that is when I say greater than b. So, we I really mean that a is of higher precedence than b. So, so these are certain rules like I identifier has got higher precedence than any other symbol, dollar has the lowest precedence. And if two operators have equal precedence

then we check the associativity rule of that operator. So, this is in general for expressions. So, this is true for expressions that I any identifier it will have higher precedence over any other symbol and the dollar will have the lowest precedence and we have to follow associativity to decide the precedence.

So, if two operators are equal precedence like say a plus b plus c. So, a plus b plus c then this a b a a. So, this plus and this plus they are of equal precedence. So, we have to follow associativity rule in that case. So, anyway so for grammars that involves only arithmetic expressions. So, these rules are valid, but in general how to decide this precedence and also that we will see as we proceed in the lecture.

(Refer Slide Time: 14:59)

	id	+	*	\$
id		>	>	>
+	<.	>	<.	>
*	<.	>	>	>
\$	<.	<.	<.	>

Example:  $w = \$id + id * id\$$   
 $\$ <.\ id > + <.\ id > * <.\ id > \$$

So, this is a precedence table; so, following the previous rule that we have so we can say that this is the precedence rule. So, I we said that id has identifier has got higher precedence than any other symbol. So, here is so identifier is having higher precedence than any other symbol. Then it says that in case of plus so plus has got lower precedence than identifier and then this is I have to follow associatively. So if I have got a plus b plus c then we do a plus b first and then do plus c.

So, plus has got the higher president over plus, similarly plus has lower precedence than star and plus has got higher precedence than dollar. So, all the terminal symbols they have got higher precedence than dollar. Similarly star it has got multiplication it has got lower precedence than identifiers, higher precedence than plus, higher precedence than

star, and higher precedence than dollar. And dollar has got lower precedence than everybody accepting dollar. So, this is the precedence table. So, this particular table we have framed by taking into consideration the arithmetic expression rules.

Now we will see later how to do it for a general grammar. Now, suppose we have got an example to be parse. So, id plus id star id. So, what we do? So, we put a dollar at the beginning and a dollar at the end and between any two terminals. So, we insert the corresponding precedence where precedence value.

So, if you consider this rule dollar and id. So, dollar is less than id. So, we are put less than then id and plus. So, id is greater than plus. So, this is greater than so that way we do it. Now it is said that anything that comes within this less than and greater than. So, that is a handle. So, this is a handle similarly if we have got this thing. So, this part will be a handle. So, that way it can identify the handles in the language in the thing.

(Refer Slide Time: 17:03)

## Basic Principle

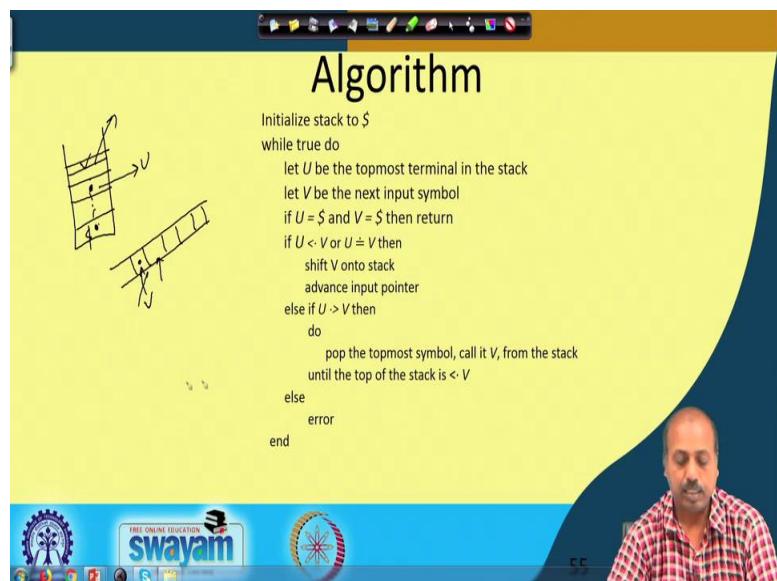
- Scan input string left to right, try to detect  $\cdot >$  and put a pointer on its location.
- Now scan backwards till reaching  $\cdot <$ .
- String between  $\cdot <$  and  $\cdot >$  is our handle.
- Replace handle by the head of the respective production.
- REPEAT until reaching start symbol.

54

So, so basic principle for this parsing is like this we scan the input string from left to right and we try to detect the greater than. So, this string that we have so I scan from left to right and try to see where first this greater than occurs. So, this is the first greater than. So, we detect the greater than and put a pointer on its location. So, we keep a note that we have seen a greater than at this point. So, keep a note will give a pointer here and then now we have to scan backwards till we reach the previous less than.

So, this one is done so we scan backwards to see what is the, we scan backwards to see where this less than appears. And once we find it the string between this less than and greater than is the handle we replace this handle by the head of the respective production. So, this handle will be replaced by E now so this is this. So, because E producing id was a rule, so E producing id was a rule. So, this handle will be replaced by E now. So, and then we repeat this process until we reach the start symbol. So, this way the parsing process will continue like this.

(Refer Slide Time: 18:31)



So, this is the overall algorithm. So, initialize the stack to dollar and we then we consider at any point of time what we do is that we have got the stack. So, say we look into the top of the stack and if this is the input stream so we look into the input symbol. So, we compare this top of the stack element with the first input symbol. So, here we have put a dollar.

So, we compare dollar with the first input if U be the topmost terminal in the stack. So, this is the so what is the topmost terminal in stack so that we find out and V is the next input symbol. So, there may be some there may be some non terminal symbols, but suppose this is the topmost terminal symbol that we have here. So, this is our U, and this is the V. So, we compare between U and V so, if U is of equal precedence than dollar, if U is dollar or V is dollar in that case we have successfully parsed the string. So, we will come out.

If not if U is less than V U is of lower precedence than V or U is of equal precedence than like V then we will be shifting V into the stack. So, in that case V will be put into the stack and we will be advancing the input pointer to the next place and if U is greater than V precedence of U is more than the precedence of V then we pop out the topmost symbol ok.

Then we were from the stack call it V from the stack until the top of the stack is less than V. So, whatever symbol we pop out. So, until that popped out symbol is having a less than relationship with top of the stack has got a less than relationship with V. So, till that much we pop out. So, that will be popping out enough entries to that is that will pop that will be popping out a complete handle. Otherwise so if that also does not happen then there is an error.

So, error condition occurs when we some table entries are undefined like say here in this table, so id id is undefined, so and as we can intuitively understand. So, if there are two identifiers coming one after the other. So, that is meaningless because in an expression between two identifiers some operator must be there. So, if the operator is not there; that means, there is some error that is there is some error. So, this entry is a is an error entry. So, we can have many such error entries. So, if you find that there is no precedence relationship between U and V; that means, there is an error.

(Refer Slide Time: 21:11)

**Example**

STACK	INPUT	ACTION/REMARK
\$	id + id * id\$	\$ < id
\$ id	+ id * id\$	id > +
\$	+ id * id\$	\$ < +
\$ +	id * id\$	+ < id
\$ + id	* id\$	id > *
\$ +	* id\$	+ < *
\$ + *	id\$	* < id
\$ + * id	\$	id > \$
\$ + *	\$	* > \$
\$ +	\$	+ > \$
\$	\$	accept

	id	+	*	\$
id	>	>	>	>
+	<	>	<	>
*	<	>	>	>
\$	<	<	<	>

The diagram illustrates the state of the parser. On the left, a vertical stack of tokens is shown with arrows pointing down to each token: 'id', '+', 'id', '\*', 'id', '\$'. To the right, a partial parse tree is drawn with three main nodes: 'id', '+', and '\*'. Arrows point from the stack to each node. Above the stack, the word 'Example' is written. Below the stack, there is a logo for 'swayam' and other educational icons.

So, let us see how this operation parsing process works. So, this is our input string, so id plus id star id. So, this is the input string so we have put a dollar at the end. So, we have put a dollar into the stack and then the rule says that you compare dollar with id and since you compare dollar and id. So, this is the a parsing table. So, dollar is less than id. So, dollar less than id so in that case the action is to shift id, so id has been shifted into the stack now it is id plus.

So, id plus so id is of higher precedence than plus. So, you so it is I so it is of higher precedence than plus. So, in that case it will be popping out the entries from idea from stack. So, id is popped out and the popped out symbol id has got top of the stack is dollar and that popped out symbol is id. So, if you look into this thing so top of the stack should have a less than relationship with the element popped out ok; so, that should be that case.

So, here you see when this id is popped out. So dollar will be on the top of the stack and dollar has got less than relationship with id. So it stops so this id is just popped out now between dollar plus. So, dollar plus dollar is of lower precedence than plus. So, plus will be shifted into the stack then we have got id. So, plus id so plus is of lower precedence than id. So, id is pushed into the stack. So, we have got star id here so, this id and star so id is of higher precedence than star.

So, we have to pop out entries. So, we pop out this id and so when we pop out between id and plus. So, id is of a higher prep, so as top of the stack now is now containing plus. So, plus is a of lower precedence than id, so it this popping out operation stops. So, we have got plus here and star here. So, between plus and star plus is of lower precedence than star so star is pushed into the stack then between star and id, star is of lower precedence than id, so id is pushed into the stack.

Now, between id and dollar so id is of higher precedence than dollar. So, so it will be id is of higher precedence. So, I have to pop out some entries so id is popped out between our now star is on the top between star and id star is of lower precedence than id. So, popping out operation stops now it is now the top of star and dollar so between the star and dollar. So, star is of higher precedence than dollar. So, it will be popping out the star star from the stack.

So, top of the stack contains plus and the symbol popped out is star and plus is of lower precedence than star. So, the popping out stops so it comes to this configuration between

plus and dollar plus is of higher precedence plus is of higher precedence than dollar. So, it will be popping out the plus symbol from the stack.

Now the top of the stack contains dollar and popped out symbol is plus. So, dollar is less than plus, so the popping out stops. So, at that point top of the stack contains dollar and the input is also dollar. So it comes to an accept state and the whole parsing process ends. So, if when you come to a situation that the top of the stack contains dollar, and the input is also having dollar; that means, we have seen the complete string.

So, it is called it is a shift reduce parsing because sometimes we are shifting the symbol whenever the next symbol that is coming on the input is whenever the top of the stack is of lower precedence than the next input symbol. So, we are pushing the symbol into the stack and whenever so that is a shift operation and whenever we are getting a situation where top of the stack is of higher precedence than the next input to come.

So basically in the conceptually you can view it like this that as if. So, if this is my stack if this is my stack at some situation. So, we have got these symbols and there is that less than and greater than relationships. So, whenever the top of the stack is of lower precedence than id so it has been pushed into the stack. So, at any point of time if you have this all these symbols so they are of so there we have got the situation that this top of the stack is of lower precedence than id. So, this id the next symbol that has come, so this is of higher precedence; so, you have got a situation like this ok.

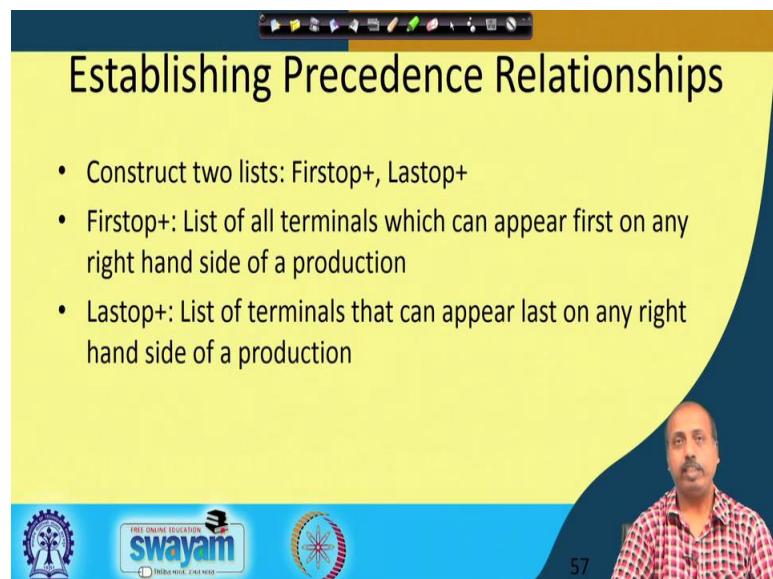
So as if they are all of higher precedence then at some point of time if you get a symbol that is going to do it like this. So whatever comes in between with all these equalities and all so this entire part is going to be one handle. So this entire part is going to be one handle. So this portion is popped out this portion is popped out from the stack and it is so that that is the reduction that we do. So we can output it is explicitly not written that we do a reduction by that.

So essentially it is doing a reduction by that rule and then we can, but then we can again proceed from this point. So this way where so we basically try to figure out the situation where we have got in the stack a condition like this and in between we have got all equality things. So this whole part becomes a handle and that handle is pruned by doing the reduction by the corresponding grammar rule.

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

**Lecture - 26**  
**Parser (Contd.)**

(Refer Slide Time: 00:15)



Establishing Precedence Relationships

- Construct two lists: Firstop+, Lastop+
- Firstop+: List of all terminals which can appear first on any right hand side of a production
- Lastop+: List of terminals that can appear last on any right hand side of a production

So, next we will be seeing how to establish this precedence relationships. So, the previous example that I have we have taken. So, that is from our knowledge in our arithmetic class's ok. So, we know the precedence between the operators and accordingly we could make that table. But in general given a language so, we have to establish precedence relations between the terminal symbols of the grammar. So, how to do that?

So, we first construct two lists; one is known as fir stop plus another is known as lastop plus. So, firstop plus is a list of terminals which can appear first on any right hand side of a production. So, you take any if you take any derivation from one particular non terminal symbol.

So, you take any grammar rule so, you so what from the starting with that grammar rule, whatever terminal symbol can appear first on the right hand side so, that is that will be called firstop plus. And similarly lastop so this is a list of terminals that can appear last on any right hand side of a production. So, that will be the lastop plus.

(Refer Slide Time: 01:27)

**Firststop and Lastop**

- For  $X \rightarrow a \dots | Bc$  put a, B, c in Firststop(X)
- For  $Y \rightarrow \dots u | \dots vW$ , put u, v, W in Lastop(Y)
- Compute Firststop+ and Lastop+ using Closure algorithm
  - Take each nonterminal in turn, in any order and look for it in all the Firststop lists. Add its own first symbol list to any other in which it occurs
  - Similarly process Lastop list
  - Drop all nonterminals from the lists

FREE ONLINE EDUCATION  
swayam

So, how to construct this lastop plus and firststop plus so, before going to this lastop plus firststop plus so, we have to consider firststop and lastop. So, firststop is like this suppose we have got a grammar rule which is say X producing a etcetera etcetera where a whatever I am writing is in lowercase letter. So, they are terminal symbols and whatever is written in uppercase so, they are non terminal symbol.

So, in this particular rule I have got a as a terminal symbol so, and this B is a non terminal c is a terminal like that. So, what you do you take from the rule the first terminal symbol that can come so, the firststop of X. So, this has got a capital B and c. So, what is the first operator or first symbol that can come first top terminal symbol that can come on the right hand side derived from X ok. So, this is the definition of firststop.

Now, you see if you follow so, from this rule we understand that I can I can get a string where the first character is a first symbol is a. And from this rule it says so, whatever will be there in the firststop of B will be in the firststop of X ok. So, that is why we include b in the firststop of X and later on when we convert in to firststop plus so, those replacements will be done.

Similarly, from the for the lastop so, if there is a rule like this so, Y produce a last symbol is a terminal symbol and here the last symbol is v and W. So, this is this is in that case u v and W will be putting in the lastop of Y.

Now, we compute firststop plus and lastop plus using the closure algorithm. So, closure algorithm is like this that we take a non terminal and then you so, we can take them in any order. So, we order the non terminals in some fashion take the first non terminal and look for it in all the firststop lists. So, in so then we check whether that particular non terminal appears in the firststop list of any other non terminal.

So, if it happens to be so then you add this firststop symbol list to any other in which it occurs. So, like here if you know what is the firststop list of B, then you can add the that firststop list of B to the firststop list of X. So, this way we can compute the firststop plus. Similarly we can create the lastop plus lastop list ok.

So, that will give us lastop plus and then from these two lists so, we drop all the non terminal symbols. Because by the definition of firststop plus and lastop plus so there is no such terminal and non terminal symbols will not be there only terminal. So, we drop all the non terminal symbols from there.

(Refer Slide Time: 04:35)

**Example**

$E \rightarrow E + T \mid T$	$\text{Firststop}(E) = \{E, +, T\}$	$\text{Lastop}(E) = \{+, T\}$
$T \rightarrow T * F \mid F$	$\text{Firststop}(T) = \{T, *, F\}$	$\text{Lastop}(T) = \{*, F\}$
$F \rightarrow (E) \mid \text{id}$	$\text{Firststop}(F) = \{(\), id\}$	$\text{Lastop}(F) = \{(\), id\}$
$\text{Firststop+}(E) = \{E, +, T, *, F, (\), id\}$		$\text{Lastop+}(E) = \{+, T, *, F, (\), id\}$
$\text{Firststop+}(T) = \{T, *, F, (\), id\}$		$\text{Lastop+}(T) = \{*, F, (\), id\}$
$\text{Firststop+}(F) = \{(\), id\}$		$\text{Lastop+}(F) = \{(\), id\}$
$\text{Firststop+}(E) = \{+, *, (\), id\}$		$\text{Lastop+}(E) = \{+, *\}, id\}$
$\text{Firststop+}(T) = \{*, (\), id\}$		$\text{Lastop+}(T) = \{*, (\), id\}$
$\text{Firststop+}(F) = \{(\), id\}$		$\text{Lastop+}(F) = \{(\), id\}$

FREE ONLINE EDUCATION  
**swayam**  
Digitize, Democratize Education

60

So, it has take an example and see how this thing happens, say I have got this grammar E producing E plus or T or T, T producing T star for F. F producing within bracket your id. So, firststop of E if you go by this rule it says, E will be there then this plus will be there because this is a first terminal symbol that can come and this is a T, from these rule I can get T. Similarly from T star T producing T star F I will get T and star and from T

producing F I will get F so that is the firststop of T. And firststop of F will be it can be open bracket start or id. So, open parenthesis and id so, these are the firststop of F.

Similarly, if you try to see lastop of E so, it can a lastop can be E plus or this T. So, this plus and T are in the lastop of E; lastop of T is star and F and lastop of F is close parenthesis and id. Then we have to consider the you have to construct the firststop plus so, firststop plus of E is it says that since T appear. So, whatever is appearing in the firststop list of T so, that should appear in the firststop list of E. So, apart from this E plus T so we add this we add this T star and F on to this.

And then in the next round so, since F appears so whatever is in the firststop list of F so, that will appear in here. So, as a result this open bracket open parenthesis and id say they also appear in the firststop plus list of E. Similarly we can construct the firststop plus of T so, which is T star F open parenthesis and id and firststop plus of F which is open parenthesis an id. Then similarly we can construct the lastop plus list. So, lastop plus of E will be whatever is there plus and T, then whatever is there in the lastop of T so, they will appear like star F and from the F I will get this close parenthesis an id so all of them will come.

So, this way we can go on constructing the firststop plus and lastop plus for individual non terminals. And then from this list so, you have to drop the non terminal symbol. So, this E T F are dropped so, you get it like this. Similarly from this you get star open parenthesis an id here you get open parenthesis and id. So, so here also you get the corresponding lastop plus. So, once you have constructed this firststop plus and lastop plus for each of the non terminal symbols. So, we can find out some precedence rule that can that we can be the that can be used for getting an operator precedence parser for the language.

(Refer Slide Time: 07:29)

Constructing Precedence Matrix

- Whenever terminal  $a$  immediately precedes nonterminal  $B$  in any production, put  $a < \alpha$  where  $\alpha$  is any terminal in the First<sub>0+</sub> list of  $B$
- Whenever terminal  $b$  immediately follows nonterminal  $C$  in any production, put  $\beta > b$  where  $\beta$  is any terminal in the Last<sub>0+</sub> list of  $C$
- Whenever a sequence  $aBc$  or  $ac$  occurs in any production, put  $a \doteq c$
- Add relations  $\$ < a$  and  $a > \$$  for all terminals in the First<sub>0+</sub> and Last<sub>0+</sub> lists, respectively of  $S$

$A \rightarrow aBx$

$a < \alpha$

$\alpha > \beta$

$a \doteq c$

So, so this is the rule to construct the precedence matrix. So, it says that whenever a terminal  $a$  immediately precedes non terminal  $B$  in any production. So, if you have got something like this. So,  $A$  producing say  $a B$  so, here this terminal  $a$  sorry. So, this  $A$  producing say  $A B X$  then what happens is that this  $a$  or say instead of  $X$  so, I can write like alpha so, any symbol any string of so, alpha is any string of gamma symbols.

Now, this  $B$  is a non terminal and  $A$  is a terminal. So,  $A$  is immediately preceding non terminal  $B$  in a production then whatever you have in first<sub>0+</sub> of  $B$ . So, whatever you have got in first<sub>0+</sub> of  $B$  so, so let us make it some other symbol so grammar. So, first<sub>0+</sub> suppose it contains the symbol alpha, then I have to make a less than alpha ok.

Then it says that the second rule tells that if a terminal  $B$  immediately follows non terminal  $C$ . So, if you have got a rule like this  $A$  producing something like say  $C$  then  $b$  something like this. Then you look into the last<sub>0+</sub> list of  $C$  and wherever you have got any beta belonging to the last<sub>0+</sub> of  $C$ , make them to be higher precedence than  $b$ . So, by this rule I have to make this last<sub>0+</sub> and first<sub>0+</sub> these two things.

And if there is a sequence like this so,  $a B c$  or  $a c$  where  $B$  is a non terminal in that case we make the for any production then we make them  $a$  and  $c$  to be of equal precedence so, this is a third rule. And the fourth rule says that dollar the end of string character. So, that

will that will have lower precedence than all terminals in the firststop plus and lastop plus list of the start symbol of the grammar. Similarly it should be greater than dollar for all the terminals in the firststop plus and lastop plus list of S.

So, it takes the start symbol of the grammar from there we get it. So, using these rules so we can formulate the precedence matrix that can be used by the parser. So, let us see for the example that you are considering how they are going to occur.

(Refer Slide Time: 10:21)

	\$	(	)	id	+	*
\$	<-		<-	<-	<-	
(	<-	±	<-	<-	<-	
)	>		>		>*	>
id	>		>		>*	>
+	>	<-	>	<-	>*	<-
*	>	<-	>	<-	>*	>

See this is the firststop of plus list for E T F this is the lastop plus list for E T F. Now let us consider this rule like E producing E plus T F E plus say E producing E plus T,. Then the first rule that we had it says that if you have a rule where a terminal a precedes non terminal B, immediately precedes non terminal B. So, that way what qualifies is this plus if we considered the first rule. So you have to see this plus and T. So, it says that whatever you have got in the firststop plus of T the rule set that whatever you have in the firststop plus of T.

So, this is a should be of lower precedence then that. So, you see here so plus is made to be lower precedence then this plus is to be made lower precedence, then this star open parenthesis an id. So, plus is made lower precedence then open parenthesis id and star fine. And similarly if you come to this rule T producing T star F, then star should be of lower precedence then the firststop plus of F. So, star is of lower precedence then open parenthesis and this id.

Then coming to this rule so, this open parenthesis an E so open parenthesis should be of lower precedence, then all the of all the symbols in the firstop plus of E so, open parenthesis is of lower precedence than this one so this open parenthesis so, plus star open parenthesis an id. So, plus star open parenthesis an id so it is less than all of them. So, that finishes complete applying the firstop rule, now with the lastop rules the lastop rules.

So, what will have? So, it says that if you have got a production like this so E so this production so E producing E plus. So, this E and this plus when you consider these two, then the rules said like this that so, this if you take care the lastop plus symbol beta. So, beta should be of higher precedence than b. So, beta should be greater than b. So, here this E so, so plus star bracket close an id, they should be of higher precedence than plus.

So, plus is a higher precedence than plus is of higher precedence than plus sorry so, plus is a higher precedence than plus is of higher precedence than star sorry, star is of higher precedence than plus. Because this is actually the set from where I am taking beta and my rule is beta should be greater than b. So, beta is greater than plus here. Similarly star is greater than plus here, then we have got close parenthesis.

So, close parenthesis is higher than plus and id is also greater than plus so, all the greater, then so I have taken care of. So, this way we can do all these things so for similarly lastop plus of T. So, if from the second rule I will get this T and star ok. Getting this T and star T and star, now lastop plus of this thing has got this star close parenthesis an id.

(Refer Slide Time: 14:25)

The slide shows the following LR(0) items:

- $E \rightarrow E + T \mid T$
- $T \rightarrow * F \mid F$
- $F \rightarrow ( E ) \mid id$
- $\text{First}_{\text{stop}}(E) = \{+, *, (), \text{id}\}$
- $\text{First}_{\text{stop}}(T) = \{*, (), \text{id}\}$
- $\text{First}_{\text{stop}}(F) = \{(), \text{id}\}$
- $\text{Last}_{\text{stop}}(E) = \{+, *, (), \text{id}\}$
- $\text{Last}_{\text{stop}}(T) = \{*, (), \text{id}\}$
- $\text{Last}_{\text{stop}}(F) = \{(), \text{id}\}$

A handwritten note  $\beta > *$  is written above the parse table.

The parse table is as follows:

	\$	(	)	id	+	*
\$	<-		<-	<-	<-	
(	<-	=	<-	<-	<-	
)	>	>	>	>	>	
id	>	>	>	>	>	
+	>	<-	>	<-	>	
*	>	<-	>	<-	>	

Handwritten notes on the left side of the table indicate transitions: from \$ to (, from ( to ), from ) to id, and from id to +. There are also arrows pointing from the first row to the second and from the second row to the third.

So, this so I should if this is said beta from where I am taking beta. So, beta should be greater than star that should be the thing. So, star should be greater than star so, that is done here star should be greater than close parenthesis star should be grater than close parenthesis done here then star should be greater than id. So, so, id should be greater than star. So, id should be greater than star. So, that way it is getting the thing. So, we can see that we can formulate this firststop plus and laststop plus and from there we can make this table.

Now, this particular rule this particular so, F producing [with/within] within bracket E. So, this is of the form that a B c and then it says that a and c should have equal precedence. So, this open parenthesis and close parenthesis so, that is of equal precedence. But it does not mean that close parenthesis so, this particular entry. So, they should also be equalities that is not that is not true ok. So, it should be like a situation is like this. So, if there is open parenthesis there will be a matching close parenthesis, but there cannot be an expression which is like this ok.

So, this is actually excluding that so the if you get an open parenthesis so, in this region so you can get an expression. But here there is a problem so, there must be some operator in between that can that should come. So, there that so the whatever entries are not defined so, they are all errors.

(Refer Slide Time: 15:59)

**Example (Contd.)**

$E \rightarrow E + T \mid T$	$\text{Firstop}(E) = \{+, *, (), \text{id}\}$	$\text{Lastop}(E) = \{+, *, (), \text{id}\}$
$T \rightarrow T * F \mid F$	$\text{Firstop}(T) = \{*, (), \text{id}\}$	$\text{Lastop}(T) = \{*, (), \text{id}\}$
$F \rightarrow (E) \mid \text{id}$	$\text{Firstop}(F) = \{(), \text{id}\}$	$\text{Lastop}(F) = \{(), \text{id}\}$

Operator Precedence Table:

	\$	(	)	id	+	*
\$	error	<	/	<	<	<
(	/	<	=	<	<	<
)	>	/	>	/	>	>
id	>	/	>	/	>	>
+	>	<	>	<	>	<
*	>	<	>	<	>	>

Bottom of the slide shows the Swayam logo and other navigation icons.

So, this is error so, this is also error so all these entries that are undefined so, they are all error entries. So, if you if you come to this error entry; that means, there is some problem with the input string. So, that has to be discarded.

(Refer Slide Time: 16:21)

Handwritten notes on operator precedence:

- $\text{Firstop}(S) = \{(), a\}$
- $\text{Firstop}(L) = \{(), a\}$
- $\text{Lastop}(S) = \{a, *\}$
- $\text{Lastop}(L) = \{a, *\}$
- $\text{Firstop}+(S) = \{a, *\}$
- $\text{Firstop}+(L) = \{a, *\}$
- $\text{Lastop}+(S) = \{a, *\}$
- $\text{Lastop}+(L) = \{a, *\}$

Operator Precedence Table:

	\$	(	)	a	+	*
\$	<	<	<	2		
(	<	=	<	2		
)	>	/	>	2		
a	>	/	>	2		
+	<	/	<	2		
*	<	/	<	2		

Bottom of the slide shows the Swayam logo and other navigation icons.

So, so next we will be taking an example another example for which we will be constructing the this operator precedence table and operator precedence, how this operator precedence table will work for parsing will work we will take an example.

Suppose let us take an example grammar which is like this. So, S producing within bracket L or a, then L producing L comma S or S so, this is the grammar that we have.

Now, so, first we have to consider at the firststop of this S and L firststop of S and firststop of L. So, firststop of S so this has got open parenthesis and a. Similarly firststop of firststop of L it has got L I will not write this commas, because that will confuse with the comma that we have there so, I will not be writing this commas.

So, let us delete these commas ok. So, let us delete this commas so, I will be as whenever writing separated by a space; that means, that is the next thing that next character that I am writing. So, this has got L then comma and S from this rule L producing S I will have this thing.

Similarly, I can write the lastop of S to be equal to lastop of S to be equal to a and close parenthesis and this lastop of L can be made equal to comma and S lastop of L is comma and S. So, from this I have to construct the firststop plus and lastop plus. So, firststop plus of S will be equal to so, if does not have any non terminal. So, this is the list firststop plus of S and firststop plus of L so, this has got L comma S. So, since S is there so, this bracket start a they will come and the comma will come.

So, this is the firststop plus of L. Similarly lastop of S will be equal to a and close parenthesis and lastop plus so, this is lastop plus of S and lastop plus of L will be equal to comma, then lastop of S will came that is a and close parenthesis. So, they will came so, this is the firststop plus and lastop plus rule. Now, based on that we can make the parsing table the or the precedence table sorry. So, you can make the precedence table which will be like this. So, we write down the terminals dollar open parenthesis, close parenthesis, a and comma.

And this side also you have got dollar open parenthesis, close parenthesis, a and comma. So, if I have to fill up this individual entries here ok, now if you apply the rule fist rule so, you have got open parenthesis L. So, this first fist one it says open parenthesis L. So, whatever you have in the first of plus of L so a should so open parenthesis should be less than firststop of L ok.

So, firststop of a firststop plus of L has got open parenthesis a and comma so, open parenthesis should be less than open parenthesis sorry. So, open parenthesis should be

less than open parenthesis, then we should get this thing this less than so, firststop L has got comma so, comma should be less than open parenthesis.

If this a should be this open parenthesis should be less than comma. So open parenthesis is less than comma open parenthesis is less than comma. So, these two are done these two are done because this is L so, open parenthesis the and a. So, open parenthesis should be less than a. So, the other entries this one open parenthesis less than a.

Similarly, we have to get the next one so, this L producing L comma S so, between this comma and S. So, comma comes before S so this comma should be less than firststop of S. So, firststop of S firststop plus S has got open parenthesis and a. So, comma should be less than open parenthesis comma is less than open parenthesis and comma is less than a. Comma should be less than a so this should this way I can do this.

Now, this equality these two open parenthesis and close parenthesis coming in this rule S producing L within bracket L. So, that will make this open parenthesis close parenthesis to be of equal precedence. So, that will be done like that. And the rest of the entries like say we have to have this rule say L close parenthesis. So, this L close parenthesis so this will tell that this laststop of laststop of L, they should be of higher precedence than close.

So, laststop L laststop plus of L has got comma and a. So, comma should be higher precedence than bracket close, comma should be of higher precedence than bracket close and this a should be of higher precedence than bracket close and this bracket close should be of higher precedence than bracket close. And then we have got this rule say comma S, so, from this second rule comma S. So, this for laststop S it should be greater than comma.

So, laststop S laststop plus of S has got a and close parenthesis. So, a should be of higher precedence than comma, a should be of higher precedence than comma. And close parenthesis should have higher precedence than comma. So, this is the other rule that we have. Now this one L comma so, L comma. So, this will tell us that this this comma so laststop of L laststop of L this should be of higher higher precedence than comma.

So, laststop of L has got comma. So, comma should be of higher precedence than comma, comma should be of higher precedence than comma then comma should be of higher precedence than comma, then this laststop of L. So, this has got a, a should be of higher

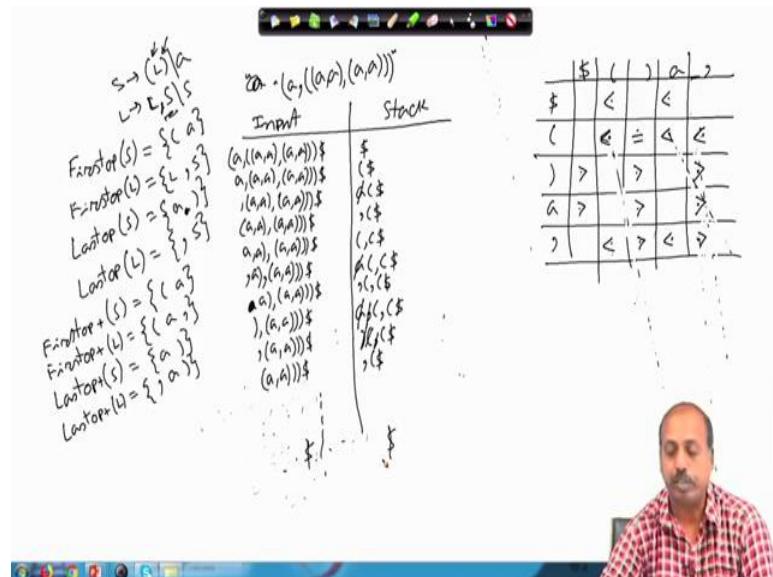
precedence than comma so, that is already done, then close parenthesis should have higher precedence than comma so, that is also already done.

Now, for the dollar part so it is says that since S is the start symbol of the grammar. So dollar should be less than the open parenthesis. So you have to see the start firststop plus of S so, that is open parenthesis and a. So, there I should have this relation. And for the laststop plus of S, I have got a and close parenthesis. So, I should have a to be of higher precedence than dollar and close parenthesis to be of higher precedence than dollar. So, using these rules so, we can formulate this table. And we can use this parsing table now for doing the operator precedence parsing.

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

**Lecture – 27**  
**Parser (Contd.)**

(Refer Slide Time: 00:15)



So, next we will be looking how into an example of how to use this particular table for doing a parsing operation. So, let us consider an input string which is given by this expression. So, this a, then comma then a comma a comma, then a comma a, then two close parenthesis. So, this is the string. So, we will see whether we can derive this string using parse this string using this operator precedence parsing table.

So, as you know that we will have two parts in our formulation; the first part will be the input and the stack. So, initially input is this whole string. So, this is a comma within bracket a comma a comma a comma a and it is terminated by dollar and the top of the stack is also dollar. So, we have got open parenthesis and dollar. So, top of the stack is dollar. So, dollar is less than open parenthesis. So, it will tell that it tells that I have to shift this open parenthesis into the stack. So, the configuration changes to something like this. Then, open parenthesis and dollar and then, this next input symbol is a and top of the stack is open parenthesis.

So, open parenthesis is less than a; open parenthesis is less than a. So, this will also be shifted into the stack. So, this input now becomes comma a comma a comma a dollar and this becomes a open bracket dollar. Now by this rule, so comma and a; so, comma so top of the stack is a and a is of higher precedence than comma. So, a will be popped out from the stack. Now, last top of the stack is now open parenthesis and a is the symbol popped out. So, open parenthesis is open parenthesis is of lower precedence than this one, open parenthesis lower precedence than symbol a.

So, the popping process stops. So, it is at this situation. Now we have got comma and open parenthesis and comma is of lower precedence than open parenthesis. So, it will be shifted. So, you will be the input string will be like this. So, it will be comma, open parenthesis and dollar. Now, we have got this open parenthesis. So, comma and open parenthesis, then comma is of lower precedence than open parenthesis. So, it will be shifted. So, the input will be like this.

So, open parenthesis will be shifted, then comma then another open parenthesis and dollar. So, this will be the situation. Now, open parenthesis and a; so open parenthesis and a, open parenthesis is of lower precedence than a. So, a will also be shifted. So, the situation will become, so, a will be shifted. So, a open parenthesis comma open parenthesis dollar. So, that is the stack. Now, between a and comma, a is of higher precedence than comma. So, a is popped out.

Now, top of the stack is open parenthesis and open parenthesis is of lower precedence than a. So, the popping process stops. Now you have got the situation; a now we have got the situation where comma is the next input symbol and open parenthesis is the top of the stack and this open parenthesis is of lower precedence than comma. So, comma will be shifted. So, we have got this situation comma is shifted fine. Next I have got comma and a and comma is of lower precedence than a. So, a will be shifted; a will be shifted ok.

Now, I have got a and closed parenthesis. So, a and closed parenthesis; a is of higher precedence than closed parenthesis. So, a is popped out from the stack. Now comma and a, so, comma and a; comma is of lower precedence than a. So, this pop process stops. Now, I have got this closed parenthesis and comma. So, closed parenthesis is of higher precedence sorry comma and closed parenthesis. So, comma is of higher precedence than

closed parenthesis. So, comma will be taken out from the stack and now I have got open parenthesis and comma.

So, open parenthesis is of lower precedence than comma. So, the popping process stops. Now, I have got this closed parenthesis and open parenthesis. So, closed parenthesis is of higher precedence than open sorry, open parenthesis and closed parenthesis. So, open parenthesis and closed parenthesis, they are of equal precedence. So, they are so, it will be pushed into the pushed into the stack. So, this becomes comma a comma a; then this dollar, now I have got this open closed parenthesis open parenthesis comma open parenthesis dollar on to the stack.

Now, top of the stack contains closed parenthesis and the next input symbol is comma. So, closed parenthesis comma higher precedence. So, this is popped out. Now top of the stack contains open parenthesis and last symbol popped out is closed parenthesis. So, open parenthesis is of equal precedence as closed parenthesis. So, this is also popped out. Now top of the stack contains comma and symbol popped out is closed parenthesis. So, closed comma is of higher precedence than closed parenthesis. So, this comma will also be popped out. This, comma is also popped out.

Now, you have got this plus and dollar. So, now, this open parenthesis, this open parenthesis and comma. So, open parenthesis and comma. So, open parenthesis of lower precedence than comma. So, the process stops. So, we have got the next situation as open parenthesis and comma. So, open parenthesis and comma say it should be less than. So, it will go to a comma a, then dollar then this would be comma open parenthesis dollar.

So, it will proceed like this ok. So, this way you can continue in this table. So, ultimately we will find that the top of the stack input will also be a dollar and top of the stack will also be dollar. In that case it will be in an accept state. So, you can just continue few more steps to get the final table.

(Refer Slide Time: 08:49)

The slide title is "Example (Contd.)". It displays a parser state transition table with columns for \$, (, ), id, +, and \*. The rows represent symbols: \$, (, ), id, +, and \*. The table entries show transitions between states, indicated by arrows pointing left or right. Below the table, there are three sets of grammar rules:

$E \rightarrow E + T \mid T$	$T \rightarrow T * F \mid F$	$F \rightarrow (E) \mid id$	$First_{\text{top}}(E) = \{+, *, (), id\}$	$First_{\text{top}}(T) = \{*, (), id\}$	$First_{\text{top}}(F) = \{(), id\}$	$Last_{\text{top}}(E) = \{+, *, (), id\}$	$Last_{\text{top}}(T) = \{*, (), id\}$	$Last_{\text{top}}(F) = \{(), id\}$
------------------------------	------------------------------	-----------------------------	--	---	--------------------------------------	---	--	-------------------------------------

The Swayam logo is visible at the bottom of the slide.

Next we will be looking into the LR parsing process which is even a better version than operator precedence because operator precedence parsing the basic difficulty that we had is that it cannot give us for parser for many of the grammars which do not have this operator grammars structure ok.

(Refer Slide Time: 09:03)

The slide title is "LR Parsing". There is handwritten text above the list: "Left-to-right" with an arrow pointing down, and "Rightmost derivation in reverse" with an arrow pointing right.

- The most prevalent type of bottom-up parsers
- LR(k), mostly interested on parsers with  $k \leq 1$
- Why LR parsers?
  - Table driven
  - Can be constructed to recognize all programming language constructs
  - Most general non-backtracking shift-reduce parsing method
  - Can detect a syntactic error as soon as it is possible to do so
  - Class of grammars for which we can construct LR parsers are superset of those which we can construct LL parsers

The Swayam logo is visible at the bottom of the slide.

So, this LR parser. So, this is the most prevalent type of bottom up parser and LR k is mostly; so, in general we will call LR k. So, where, k is the number of symbols that we do a look ahead. So, this LR; so, this L and R, so these two letters L stands for L stands

for this left to right scan; L stands for left to right scan and then this R stands for rightmost derivation in the reverse; rightmost derivation in reverse. So, it is in reverse because it is a bottom up approach. So, it is that is why it is in reverse.

So, it produces rightmost derivation unlike that LL, so which was producing a left most derivation. So, this will give a right most derivation. Both LL and LR they will do a left to right scan, but the which symbol to be replaced, so that will differ. So, we in general we will have got LR k parser. So, where k is the number of tokens that you do a look ahead or number of symbols that you look ahead. Now, in general we will have k value less or equal 1. So, will be looking into LR 0 and LR 1 grammars. So, will be looking into LR 0 and LR 1 grammars.

So, LR 0 will give us the SLR parsing table; whereas, this LR 1 will give us the canonical LR type of structure. Now, why should we go for this LR parser because most of the parsers that we will find, so they will be of this nature. They will be LR parser. So, why should we have this? So, first of all this is a table driven parsing. So, it is there is no recursion involved in it and can be constructed to recognize almost all programming language constructs. So, here though it written here as all, but it is it need not be also it is almost all.

So, and it is most general non backtracking shift reduce parsing method. So, it if there is no backtracking involved, so, you do this coding for this is pretty easy and there is no it follows the shift reduce parsing policy that is fine. So, can detect a syntactic error as soon as it is possible to do so. So, this is the fastest method to detect syntactical errors. Main reason is that it is following a bottom up approach and it is trying to identify the handles in the stack.

So, that is why as soon as it finds that there is possible handle, but it cannot be derived from any of the inputs any of the sentential forms; in that case it can flash the error. So, this way it can find out syntactic errors as fast as possible and this is the most important observation that the class of grammars for which we can construct LR parsers are the super set of those for which we can construct LL parsers.

So, if for a language you can construct LL parser, so you can also construct LR parser for that. But the reverse may not be true. So, you may be having only the you may not be

able to formulate LL parser, if we have got if we for language that can support you getting LR parser.

(Refer Slide Time: 12:47)

The slide has a yellow header bar with a navigation menu. The main title 'LR Parsing Methods' is in bold black font. Below the title is a bulleted list of three methods:

- SLR – Simple LR. Easy to implement, less powerful
- Canonical LR – most general and powerful. Tedious and costly to implement, contains much more number of states compared to SLR
- LALR – Look Ahead LR. Mix of SLR and Canonical LR. Can be implemented efficiently, contains same number of states as simple LR for a grammar

At the bottom of the slide, there is a blue footer bar with the Swayam logo and the number 63. A small video frame shows a man with a beard speaking.

So, there are three different methods of this LR parsing that will be discussing in this course. One is the SLR parsing or simple LR, they are easy to implement and it is less powerful. So, powerful in the sense that the class of languages for which you can construct a parser without any shift-reduce conflict or reduce-reduce conflict.

So, without any conflict so the set of languages for which you can construct a parser. So, if that set is pretty large, we will say that the parsing strategy is powerful. If it is not that, so we will say it is less powerful. Then we have got Canonical LR, it is a most general and powerful. So, this; so, this is the I should say the most general so for whatever language you can construct a shift reduce parser. So, we can do a we can construct a canonical LR parser for that. However, the difficulty is it is tedious. So, you will see that there are large number of states that will be created in the parser. So, that way it is difficult to make to make a CLR or Canonical LR parser like that.

And costly to implement because since the number of states are more, so you will have this policy of the program that will be rise that will be used for getting this parser will be difficult. So, contains much more number of states compared to the SLR parser. So, when we look into some example, it will be more clear. Then another class of parser

which is known as LALR which is look ahead LR. So, it will do some look ahead and try to see whether it can do better in terms of resolving the shift reduce conflicts.

It is a mix of SLR and canonical LR can be implemented efficiently and most importantly it contains same number of states as simple LR for a grammar. So, it is it has got same number of states as your SLR parser, but power wise it is same as the Canonical LR parser. So, what we normally do is that we construct the Canonical LR parser and from there by doing some equivalency analysis. So, we try to merge the states and come reduce that Canonical LR parser to a SLR to an to an LALR parser and the resulting LALR parser will have same number of states as the SLR parser

However it will be much more powerful than the SLR parser and its power will be same as that of the canonical LR parser. So, that is why most of the automated tools that we have. So, they will try to construct an LALR parser though the construction process is difficult. So, if you if you are trying by hand you using a pen and pencil, pencil and paper type of approach.

So, doing it by hand, then you can most of the time. So, you will find that you can make the SLR parser by hand, but doing the canonical LR or the LALR parser by hand is a very tedious job. So, excepting for very simple grammars, so will not be able to do it by hand.

(Refer Slide Time: 16:01)

**States of an LR parser**

- States represent set of items
- An LR(0) item of G is a production of G with the dot at some position of the body:
  - For A->XYZ we have following items
    - A->.XYZ
    - A->X.YZ
    - A->XY.Z
    - A->XYZ.
  - In a state having A->.XYZ we hope to see a string derivable from XYZ next on the input.
  - What about A->X.YZ?

So, all this LR parsers, so you have got a concept of State and a state represent a set of items. So, what is an item that we will define and LR 0 item. So, when I say LR 0; that means, it does not do any look ahead's. So, it just looks into the current input symbol and based on the state in which the parser is and the next current input symbol, it will take a decision like what to do. So, what which state to go next or what should be the action.

As we know that there all are four action shift, reduce, accept and error. So, which action to be taken? So, it will be deciding on that. So, if there is a production a producing XYZ. So, like this then we can have. So, all these are items. So, an item you can say that it is like a production rule with a dot somewhere on the right hand side. So, here, so this is an item where dot is at the beginning; this is an item where dot is after the X symbol. So, this is after the Y symbol and this is after the Z symbol.

So, that is the notation, but what do we really mean by that? So, when we say that we are in a state that has got this as an item. So, we are in a state where we have got this a producing dot say dot XYZ as an item; that means, in this state I am expecting to see a string which is derivable from XYZ on the next input. So, whatever be the next input symbol say A, so that means, if you if I am currently in this state and the next input symbol is A. Then it will be it will be leading to derive a string. So, from this point I will be I will expect that I will find a string which is derivable from XYZ.

So, this way it will try to do a prediction like it will try to figure out which rule which type of derivation I am going to see and that is what it will be doing. Now what about this one? So, Y producing X dot YZ. So, this means we have already seen a part of the string we have already seen a substring which is derivable from X. So, as if this is the next if this is the input, then we are at this point a. Then, this part in this region I have already seen a string which is derivable from X and then, the next input symbol is a and from this point I am expecting to see a string which is derivable from YZ ok.

So, that way, so this dot is very important. So, dot means the portion before dot is I have already seen that. So, in some state, so if I have; so if I have what say number of items like say A producing X dot YZ or say B producing say PQ dot R like that. So, if I have got say two items that means, I could have I have arrived to this state either by either of these two cases; like I have seen a string which is which is derivable from X and now I

am expecting to see a string which is derivable from YZ or I have seen a string which is derivable from PQ and now I am expecting to see a string which is derivable from R.

So, this way, so all the items that you have in a state. So, that they will tell us that they will give us enough hint about the type of string that we have seen so far and the type of string that we are going to see are expecting next to see. So, based on that we have to take a decision, like at this point; so if the next input is such that so this one survives. So, X producing YZ survives. So, will be going to another state where it will be it will be advancing input and it will expect some string that way.

On the other hand, if the next input is such that so this alternative survives ok. So, this is the this is not in that case, it will be coming to a state where it will be doing like this. Now if the parser cannot take any decision like, then of course, there will be an error. So, shift-reduce conflict or reduce-reduce conflict can occur. So, we will see that as we proceed through this slides. So, this is the meaning of A producing X dot YZ and that is we have already seen is a string derivable from X and then, we are expecting to see a string which is derivable from YZ.

(Refer Slide Time: 20:39)

## Constructing canonical LR(0) item sets

- Augmented grammar:
  - G with addition of a production:  $S' \rightarrow S$
- Closure of item sets:
  - If I is a set of items, closure(I) is a set of items constructed from I by the following rules:
    - Add every item in I to closure(I)
    - If  $A \rightarrow \alpha B\beta$  is in closure(I) and  $B \rightarrow \gamma$  is a production then add the item  $B \rightarrow \gamma$  to closure(I).
- Example:
 
$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

$I_0 = \text{closure}(\{[E' \rightarrow E]\})$
$E' \rightarrow E$
$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

Now, how do we construct this LR 0 items ok. So, canonical LR 0 items, item sets, because it is not item. So, item sets because each of them each of this thing like A producing dot something. So, that is a, that is an item as we have seen in the last slide. So, the all of these are items. So, all of these are items.

(Refer Slide Time: 21:01)

- States represent set of items
- An LR(0) item of G is a production of G with the dot at some position of the body:
  - For A->XYZ we have following items
    - A->.XYZ
    - A->X.YZ
    - A->XY.Z
    - A->XYZ.
  - In a state having A->.XYZ we hope to see a string derivable from XYZ next on the input.
  - What about A->X.YZ?

So, when we say a set of items, so, this is basically a collection of items that we had like A producing X dot YZ; so, B producing P dot QR. So, these are all; so, the individually they are item. So, this collection of items so that is called set of items and that will constitute a state.

So, we see that um. So, how to construct this LR 0 item sets? So, first thing that we have to do is to augment the grammar by another rule S dash producing S ok. So, why do we do this thing? So, previously we have a, we had a grammar that started with S and all these rules were there. So, in the ultimate production or that parse tree that is produced. So, at the root we had got S and then from here everything is derived. So, this is there are lowest levels. So, we have got all the terminal symbols. So, what we are doing? So, we are adding another rule is just producing S.

So, why do we do this? We do this because when the parser will try to do this reduction S to S dash; then, we know that we have seen the we see we have been successfully parse the whole string. So, that is the idea of adding this extra production is just producing S into this grammar set G and call it an Augmented grammar. So, we augment the grammar G by adding the rule is just producing S; then we construct the closure of item sets. So, if I is a set of items, closure I is a set of items constructed from I by this rules.

So, at every item of I to closure of I and if A producing alpha dot B beta is in closure of I and B producing gamma is a production, then add B producing dot gamma to the closure

of I. So, this is how will be doing the closure. So, suppose we have got a grammar like this E dash producing; so, this is that augmented ETF grammar. So, we have got this original grammar had these three rules E T and F and we have added another rule E dash producing E add as the extra rule.

Now, say I 0 that is items item set 0. So, this is this was having the only one rule E dash producing dot E. So, this is for the whole graph this is for the situation where I have not yet seen anything. So, I am expecting to see a string which is derivable from E. So, E dash producing dot E. So, when I take closure of this, so by applying this rule. So, you see we have got E dash producing dot E. So, if you compare with this.

(Refer Slide Time: 24:07)

## Constructing canonical LR(0) item sets

- Augmented grammar:
  - G with addition of a production:  $S' \rightarrow S$
- Closure of item sets:
  - If I is a set of items, closure(I) is a set of items constructed from I by the following rules:
    - Add every item in I to closure(I)
    - If  $A \rightarrow \alpha B\beta$  is in closure(I) and  $B \rightarrow \gamma$  is a production then add the item  $\alpha B\gamma$  to closure(I).
- Example:
  - $E' \rightarrow E$
  - $E \rightarrow E + T \mid T$
  - $T \rightarrow T^* F \mid F$
  - $F \rightarrow (E) \mid id$

$\alpha = \epsilon$   
 $B = E$   
 $\beta = \epsilon$

$I_0 = \text{closure}(\{[E' \rightarrow E]\})$ 

```

E' > E
E > E + T
E > T *
T > T * F
T > F
F > (E)
F > id
  
```

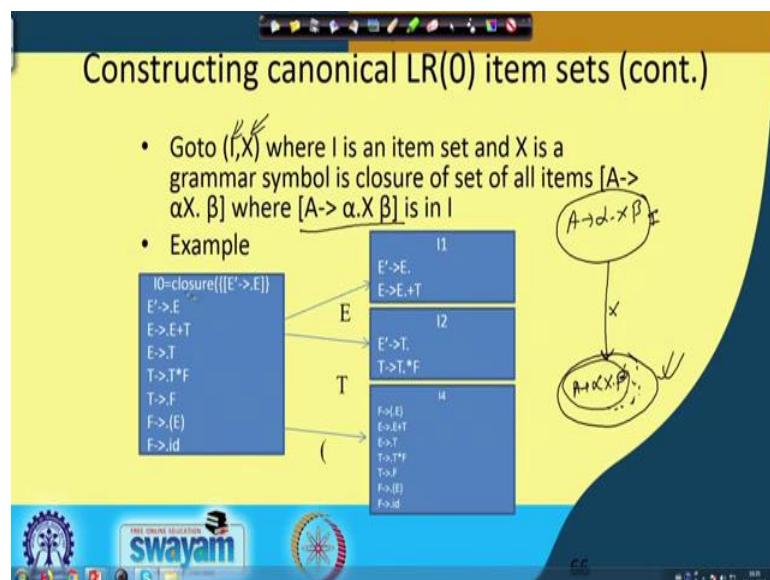
Then, alpha is epsilon; then B is E; B is equal to E and beta is also epsilon.

So, comparing with this, now it says that whatever we have so B producing gamma; so, B producing gamma; so, this rule survive. So, E producing E plus T and. So, if I do that, then it says that B producing dot gamma should be added to closure of I. So, E producing dot E plus T should be added to the set. So, E producing dot E plus T is added, then this E producing T, this also survives by the B producing gamma as per for this rule. So, E producing dot T is also added. Now, as soon as E producing dot T is there ok, Then again so this rule has to apply.

So, here alpha equal to epsilon B equal to T and gamma equal to epsilon and then, sorry beta equal to epsilon; beta equal to epsilon and then it says that B producing dot gamma. So, T producing dot T star F and T producing dot F. So, these two are added to the set and as soon as this F T producing dot F is added. So, based on that this F producing dot within bracket E and E at the F producing dot id. So, they are also added to the set. So, this way I can compute the closure of this particular set E dash producing dot E and that will constitute one state of items.

So, I 0 is the initial state of the parser where it starts with E dash producing dot E and takes the closure of that particular item to get the whole all the items in that set. So, in his way, so we can construct the items.

(Refer Slide Time: 26:05)



The another part of is to construct the Goto. So, Goto  $I, X$ . So, this is defined for an item  $I$  for an item  $I$ . So, on some grammar symbol  $X$ , we can define a Goto. So, if  $I$  is an item set and  $X$  is a grammar symbol is a closure of all the items a producing  $aX$  dot beta, where  $a$  producing alpha dot  $X$  beta is in  $I$ . So, it is basically since  $a$  producing alpha dot  $X$  beta is in the item. So, from this if  $I$  get  $X$ , then what we are planning to do so that is the thing.

So, if you if you get an  $X$  naturally you will be coming to a state where  $a$  producing alpha  $X$  dot beta will be an item because if you have seen  $X$ , then you will be expecting to see a string which is derivable from beta. So, this will be calling Goto  $I, X$ . So, the

Goto. So, this is the set I; this is X. So, Goto I, X is this set and you take the closure of this. So, to get all the items in this state. So, we have to take the closure of that one. So, this is an example like say this I 0 is the state which is I 0 is the state which is E dash producing dot E and the closure of that. So, this is the set of items that we have.

Now, from here on E, so from this rule you see the E dash producing dot E. So, if you see E, then you will be going to an item E dash producing E dot. So, E dash producing E dot. So, that is one possibility and from these item, if you get an e you will come to a configuration E producing E dot plus T ok. So, that is the thing. No other rule has got a dot before E. So, they will not be coming in the set I 1. So, in; so, now, in the set I 1, you see after dot there is no non terminal. So, naturally I told nothing will be added even if I take closure of I 1.

So I one remains this set only. What about I 2? So I 2 is on T. So, on T if we; so on T if we go so similarly we will get E. So, this should not be E dash. This should be E. This should not be E dash, it should E. So, E producing T dot and from this rule, I will get T producing T dot star F and again the same thing that we do not have any non terminal after dot. So, the closure does not have anything; only these two item survive and then I can say on so any other symbol like this open parenthesis is there. So, F producing dot E; so on open parenthesis, so it will come to a configuration where a producing open parenthesis dot E closed parenthesis.

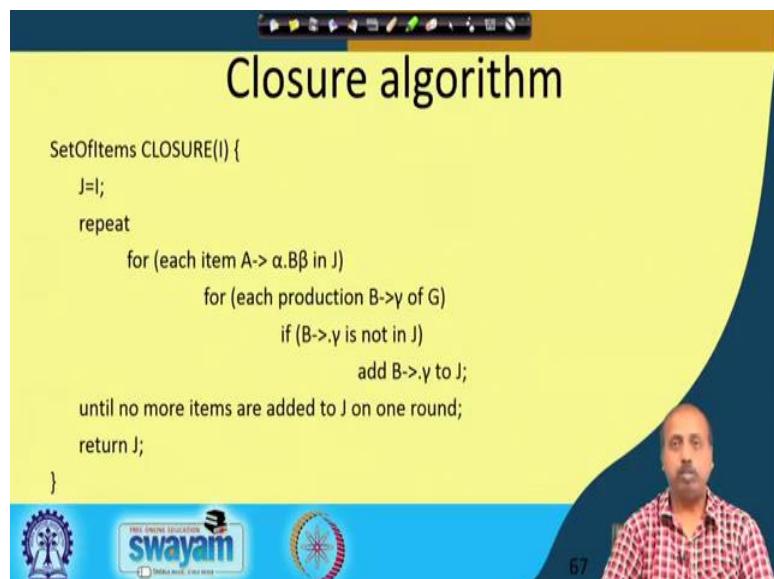
So this is the item and since now you have got a dot before this E. So, you have to again scan the grammar rules to see what may be the situations. So, they as you know that the grammar had this production rule E producing E plus T. So, this has got dot E. So, I have to add this E producing dot E plus T in the closure set; then E producing dot T in this set and as soon as e producing dot is there. So T producing dot T star F will be there, T producing dot F will be there, F producing dot within bracket E will be there and F producing dot id will be there. So, in this way you can construct the set I 4 ok.

So, this way for all this from a particular state. So you can define your Goto's and on different terminal and non terminal and accordingly, you can derive new states that the parser can go.

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

**Lecture – 28**  
**Parser (Contd.)**

(Refer Slide Time: 00:15)



```
SetOfItems CLOSURE(I) {
    J=I;
    repeat
        for (each item A-> α.Bβ in J)
            for (each production B->γ of G)
                if (B->γ is not in J)
                    add B->γ to J;
    until no more items are added to J on one round;
    return J;
}
```

So, we can look into a closer algorithm. So, set of items of closure I. So, it will return a set of items which is the. So, the closure I is the function. So, we start with a set J; J is initialized to I. So, whatever you have in thus item I here. So, J will have their those items and then for each item A producing alpha dot B beta in J and for each production B producing gamma of J. So, if B dot gamma is not in J, then we add B dot gamma to J. So, this algorithm we have already we intuitively seen like while discussing on the examples.

So, this is a formal way of writing the algorithm. So, until no more items can be added to J on one round. So, if you try with all the rules and then find there no more production can be added, then we will be stopping the algorithm and it will be returning the set of items J.

(Refer Slide Time: 01:11)

GOTO algorithm

```
SetOfItems GOTO(I,X) {
    J=empty;
    if (A-> α.X β is in I)
        add CLOSURE(A-> αX. β ) to J;
    return J;
}
```

FREE ONLINE EDUCATION  
swayam  
India's first e-university

68

So, that is the closure algorithm. Similarly, the GOTO algorithm. So, this also we have seen the GOTO operation. So, this is very simple. So, GOTO I, X from item I on input on the input X where do you go? So, input maybe a terminal or a non terminal or a basically some grammar symbols. So, its start with J equal to empty and this if A producing alpha dot X beta is in I, then we add closure of A producing alpha X dot beta to J. So, this is alpha dot X beta is in I. So, on X it will go to alpha X dot beta and then we take the closure of it and then add that item to J. So, it will be returning J.

So, this is the GOTO part.

(Refer Slide Time: 01:59)

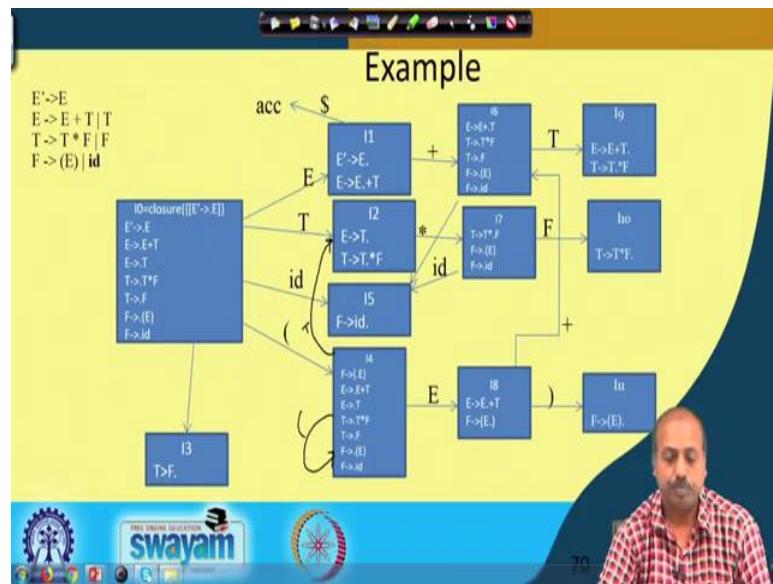
```
Void items(G') {  
    C= CLOSURE({[S'->.S]});  
    repeat  
        for (each set of items I in C)  
            for (each grammar symbol X)  
                if (GOTO(I,X) is not empty and not in C)  
                    add GOTO(I,X) to C;  
        until no new set of items are added to C on a round;  
}
```

The slide has a yellow header with the title 'LR(0) items'. Below the title is the pseudocode. At the bottom, there is a blue footer bar with the 'swayam' logo and other icons. On the right side of the slide, there is a video feed of a man with a beard and mustache, wearing a red and white checkered shirt, sitting at a desk.

Now, the algorithm for constructing the LR 0 items. So, we start with. So, we start with the augmented gamma G dash; then this are this is the grammar is passed. So, we start with the initial set which is the closure of this particular rule S dash producing dot S. So, this as you remember that this S dash producing dot S is the additional rule that has been put into the set into the grammar G to get the augmented grammar G dash. So, C equal to closure of S dash producing dot S. Now, for each set of items I in C and for each grammar symbol X. So, if GOTO I, X is not empty and not in C; then we add GOTO I X to C.

So, we have already seen how to make this GOTO I, X function and the closer function. So, if it is if you if it can give us some new item then will be adding it to C.until no new set of items can be added to C on a round. So, this way it will be constructing the LR 0 items. So, this is this is simply the procedure.

(Refer Slide Time: 03:17)



Now, once we have done that. So let us take an example of grammar and how this items can be constructed. So, if this is the grammar  $E$  dash producing  $E$ . So, this is the originality of grammar augmented with the rule  $E$  dash producing  $E$ .

And the initial item is constructed by closure of  $E$  dash producing dot  $E$  this item. So, this  $E$  dash producing dot  $E$ , you will have  $E$  dash producing dot  $E$ . So, this we have already discussed. So, whatever you have got dot before  $E$ . So, all these rules is survive  $E$  dash  $E$  producing dot  $E$  plus  $T$  and  $E$  producing dot  $T$  from  $E$  producing dot  $T$ . So, if this rule will come  $T$  producing dot  $T$  star  $F$  and all. Now from these, so we take the GOTO. So, you look into the symbol that comes after dot. So, after dot we have got  $E T F$  open parenthesis and  $id$ . So, for each of these cases, so I have to find what the corresponding GOTO is.

So, if I do it on  $E$ . So, I will get  $E$  dash producing  $E$  dot and  $E$  producing dot  $E$  plus  $T$ . So, that is there. Then, you have got this thing  $T$ . So, if you get  $T$ , then this  $E$  producing  $T$  dot and  $T$  producing  $T$  dot star  $F$  they will come; from  $id$ , I will get  $I$  will get  $F$  producing  $id$  dot and from this open parenthesis, I will get  $F$  producing dot  $E$  and since dot  $E$  is coming, so all the rules will come. So,  $E$  producing dot  $E$  plus  $T$ , then  $E$  producing dot  $T$ ; then  $T$  producing dot  $T$  star  $F$ . So, all these rules will come. So, this is all these four sets that we have consider seen. So, they are all new sets and also the some

F. So, it will go to. So, this labeling is missing. So, this should be on F. So, this is on F, it comes to the state 3, I 3. So, this is E produce T producing F dot.

Because nothing more has to be done, so it is like this. Now, from this one from I 1 you see that I have got a dot before plus. So, on plus it will go to the state E producing E plus dot T and from this E plus dot T. So, this dot is there before T. So, again all these rules will come T producing dot T star F T producing dot F producing dot within bracket E and all that. So, all of them will come and from this one there is a dot before star. So, based on that it will come to this rule, T producing star dot F and then the since there is a dot before a F. So, it will be all this F rules will come F producing dot E dot within bracket E and F producing dot id like that. Similarly, from this I cannot have any more production because this is this is no symbol after this dot. From this for this one I will get this dot E. So, dot E it will be adding this thing. So, dot E will be. So, on E it will go to dot so, E plus dot T.

And this F producing E dot within bracket E dot like that and. So, from this one I will have also rules like on T, on T it will go to go to E producing T dot. So, it will have E producing T dot and T producing T dot star F. So, these 2will come and that will give me the rule I 2 only; this is the set of items I 2 only. So, that does not give us any new set of item, similarly, on F within bracket. So, from this state if we were try to take a transition on F producing, so on this GOTO I 4 open parenthesis, then it will give me this rule F producing open parenthesis dot E close parenthesis. Now since so this is same as I 4 because I 4 was like that only. So, this will be giving us I 4 only. So, if you were try to draw explicitly, then you can you can draw these type of transitions. So, like this. So, this is on open parenthesis.

Similarly, this is on T; this is on T. So, like that you can draw other cases ok. So, this is not they are not showed here, so, apart. So, if you if you explore this algorithm you will find that accepting these set of items no new items are generated. So, all others are just duplicate of the previous ones. So, that way you can just write down the set of items that are generated from the grammar rules.

(Refer Slide Time: 08:19)

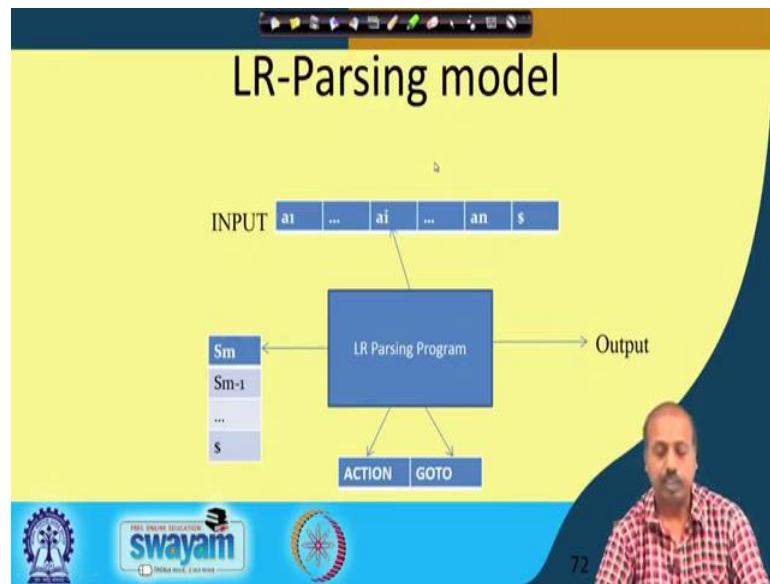
Line	Stack	Symbols	Input	Action
(1)	0	\$	id*ids	Shift to 5
(2)	05	sid	*ids	Reduce by F->id
(3)	03	\$F	*ids	Reduce by T->F
(4)	02	\$T	*ids	Shift to 7
(5)	027	\$T*	ids	Shift to 5
(6)	0275	\$T*id	s	Reduce by F->id
(7)	02710	\$T*F	s	Reduce by T->T*F
(8)	02	\$T	s	Reduce by E->T
(9)	01	\$E	s	accept

Now, so, next we will be looking into how to use this LR 0 automaton for identifying the strings. So, initially the stack contains the state 0 and this as the symbol is dollar and input is id star id dollar. Then, it will say that once it is getting this id. So, it will be shifting to state 5. So, how does it shift? So, that decision we will see later. Suppose, if the action is the like shift the next input and go to state 5. So, it shifts the next input id into the stack and go to state 5.

So, this 5 state is put into the stack. Now it says now the next input is id and the sorry star and the current state is 5. So, current state is 5 and the input is star. So, it finds that there is a rule like F producing id dot. So, based on that it understands that it has seen an seen a handle. So, it will reduce these by F producing id. So, this id will be taken out from the stack and now, the new state will be this F will be put into the stack the left hand side and the new state will be determined by the GOTO part.

We will see that algorithm. So, it will come to state 3. So, this way this algorithm will proceed and it will finally, come to this accept state. I am not going to explain it in detail because it is not possible to understand it until unless we have seen in more detail how this parsing algorithm works with the help of this basic set of items and the transition diagrams.

(Refer Slide Time: 10:01)



So, this is how the input works that this parsing process works. So, we have got this LR parsing algorithm. So, it has got a stack which has got these states in it. So, this stack will have that state. So, initially this is dollar and it has got the other states in it and then, this is the input a 1, a 2 up to ended with a dollar.

And there is a parsing table. So, which has got two parts in it; one part is called Action part, another is called GOTO part. So, this Action part and GOTO parts; so, these two are there. So, this LR parsing program, so, it will take that it will consider the current input; it will consider the. So, it will consider the current state, the current input and it will consult the corresponding table and based on this table it will take some action whether to shift the next input or whether to do a reduction; whether to accept and all accordingly it will output may be the production rule by which it does the reduction or the declaration that the input is accepted or error. So, like that. So, that is the output part.

(Refer Slide Time: 11:13)

The slide title is "LR parsing algorithm". The pseudocode is:

```
let a be the first symbol of w$;
while(1) /*repeat forever */
    let s be the state on top of the stack;
    if ACTION[s,a] = shift t {
        push t onto the stack;
        let a be the next input symbol;
    } else if (ACTION[s,a] = reduce A->B) {
        pop |β| symbols of the stack;
        let state t now be on top of the stack;
        push GOTO[t,A] onto the stack;
        output the production A->β;
    } else if (ACTION[s,a]=accept) break; /* parsing is done */
    else call error-recovery routine;
}
```

The slide also features two diagrams:

- A diagram showing a stack with a pointer at the top, labeled with a symbol.
- A diagram showing a stack with a pointer at the bottom, labeled with a symbol.

The footer of the slide includes the "swayam" logo and other navigation icons.

So, this is the L R parsing algorithm. So, if let a be the first symbol of the input stream w. I repeat now what we do and let s be the state on the top of the stack. Now, if action of s a is shift t. So, if the if the action is shift t, so, it will put it will push t onto the stack and let, now let a be the next input symbol. Otherwise if the action is by reduce a producing beta. So, it will we have pop out beta symbols from the stack and now if the state t is on the top of the stack, then it will go to it will push go to t a onto the stack and output the production a producing beta; else it the action is accept then it will accept it, otherwise it will call in a recovery routine.

So, what does it mean? So, if this is the input, so, at any point of time suppose this is the symbol a that we have and the current state so that is on the stack, we have got the states and the current state is say s. Now it will consult the table, the parsing table for the state S and input a and the if the corresponding input is shift t. So, we will represent it normally by s t. So, it will be doing. So, part identifies the that it is a shift action. So, it will be a shift operation and the new state will be t. So, in that case what it will do? It will push t onto the stack. So, it will be pushing t, so, this s. So, t becomes the state on the stack and the input pointer will be advanced to the next one. So, that way this pointer is advanced.

On the other hand, if action is reduced by a producing beta then it will be popping out beta number of entries from the stack and then, it will be if t is on the top of the stack

now; then it will be finding out this GOTO t, A these state and it will be putting it into the stack. So, we will see some example and then it will be more clear ok.

(Refer Slide Time: 13:35)

The table shows the following data:

STATE	ACTION						GO TO
	id	+	*	(	)	\$	
0	S6			S4			R1
1		S6			S7	R2	Acc
2		R2	S7	R2		R2	
3	R4	R4	R4	R4			
4	S5		S4				8 2 3
5		R6	R6	R6	R6		
6	S5		S4				2 3
7	S5		S4				10
8	S6		S11				
9	R1	S7	R1	R1			
10	R3	R3	R3	R3			
11	R5	R5	R5	R5			

**Handwritten notes on the right:**

- (0)  $E' \rightarrow E$
- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow id$
- $id * id + idS$

**Diagram:** A stack diagram showing tokens being pushed onto the stack. Labels include 'state 5' and '2 > production'.

So, we do not know how do we construct this table ok, this parsing table how do we construct suppose that is not known. So, let us see how this suppose this table is given to us and the meaning the way we read this table is say in these state. So, this state 0, so if you get an input id; then this is the corresponding action. It says that you do a shift operations; so input will be shifted and it will be coming to a state 5 ok. That is the that is how we will read it.

Similarly this is S 4; that means, if the current if the in state 0 if you find open parenthesis. So, you will do a shift operation and it will come to state 4. Now this one. So, this reduce. So, this one. So, R 4, it tells that you have took in state 3 if the input is star; if the input is sorry input is plus, then you have to apply reduction rule number 4 for doing a reduction. So, for that purpose this grammar rules and numbers like 0 1 2 3 4 like that. So, it says that you have to apply a grammar rule and do the reduction by that ok.

So, let us see how this thing proceeds. So, let us taken consider an example say this is the string. So, id star id plus i id. Now initially my input pointer is here and this stack contains state 0 and there is nothing in nothing symbols in the stack. Now, it will be finding. So, this is the input i d. So, this will be from state 0 and id state 0 id it says that it should be state 5. So, 5 the state 5 is pushed into the stack and then, this symbol id is also

put the we have seen the symbol i d. So, this is on the this is also on the stack. Now it says the next is star. So, this is star is. So, now, state 5 star. So, it says reduced by rule number 6 and rule number 6 is F producing id. So, it says. So, it will. So, a reduction. So, this id will be taken out. So, id will be replaced by F and the new state will be identified from this.

So, to identify the new state, what we will do? For a new state whatever if t is now the top of the stack, then GOTO t, A will be the new state. So, now after taking it so 3 is on the top of the stack, so 3 F; so, 3 F sorry. So, sorry this 3 is also popped out because from state 3. So, it has found F. So, this is a we are in at this point. So, this is this is be reduced by F producing id. So, this state 5 has gone out. So, this state 5 has gone out. So, this top of the stack now contains 0 and this 0 F. So, 0 F says that the new state is 3. So, this F is put into the stack and this 3 is put into the stack.

So, we have F is remembered as symbols next symbol and 3 is the stack. Now the next at state 3 I have got this star. So, state 3 star says that you do a go by rule number 7; reduced by rule number reduced by. So, this is basically. So, this is not rule number 7. So, this is rule number 4, there is a mistake here. This should be rule number 4. So, reduce by rule number 4; so, this 3 1 getting star. So, it will be on state 3 on getting star. So, it will be a reducing by T producing f. So, T producing F is rule number 4. So, this should be rule number 4. So, it will reduce by that. So, this F will be going out of the out of the symbol set and this T will be coming.

And the new state will be 0 T. So, 0 T is state number 2. So, this 2 is the new state and now on 2 star; so, 2 star, it says that shift and GOTO state 7. So, this it has done a shift. So, star is shifted and this 7 is there on the stack. Now state 7 is on the stack. Now from there it will find that now it is 7 and I d. So, state 7 id, it says shift 5. So, this id is shifted and this new state becomes 5. So, that is put into the stack. Now 5 and plus. So, 5 plus it says that reduce by rule number 6. So, reduce by rule number a rule F producing I d. So, it will be taking out id from the stack, replace it by F and then F is put into the stack and then from this 5 is also gone.

And from this 7 F; so, 7 F will tell that the GOTO is GOTO 10. So, this 10 is put into the stack. So, new state is the state 10 ok. Now from state 10 on plus from state 10 on plus, it says reduced by rule number 3. So, reduce by rule number 3 is this one, T producing T

star F. So, this T star F will be going out of the stack and then, this state 10 will also be going out. So, the new state is 7 and the left hand side is T. So, 7 T 7 from actually this state 7 will also go. So, it will be the state new state should be 2. So, where is 2? So, this will be actually when it is popping out when it is taking out these T star F. So, all these states will also go out. So, we have popping out 3 in entries from the symbol. So, this 3 states will also go. The 10, 7 and 2 they will go and 0 on F, so it will say that from state 0 on T, it will say that it will go to T; so, 0 T. So, this state two comes here. Actually the point is the here I am popping also here; I have to replace by the right hand side has got 3 star F. So, 3 symbols.

So, 3 symbols are taken to be symbols taken from these symbols and 3 states all also to be taken from this stack because for each symbol there is a corresponding states. So, if I am taking out 3 steps also 3 symbols from here. So, 3 states are also to be taken out from here. So, this takes it to 0 state; it makes state 0 at the top and from state 0 on getting on the next input T. So, GOTO 0 T is 2.

So, it goes to state 2. So, it 0 to 2 and T. So, now, from a state 2 on plus state 2 on plus it goes to reduce by rule number 2. So, it will follow rule number 2 that is E producing T. So, it will be removing this symbol from the stack, it will be removing these state from the stack and now this left hand side is E. So, this is replaced by E and this 0 E. So, 0 E is 1. So, this state is 1. Now 1 and plus. So, 1 plus says shift by shift and go to state 6. So, this plus is shifted and it goes to a state 6 and now it will be 6 and i d. So, 6 and id tells it is shift 5. So, it will be it will be shifting. So, it is E plus id and this is 5 and next it will be 5 and dollar ok.

So, 5 and dollar. So, is reduced by rule number 6; so, F producing i d. So, this id goes out this 5 also goes out. So, F comes in and 6 F is you know 6 F is 3. So, accordingly the state becomes 3. Now, 3 and dollar; so, 3 and dollar tells 3 and dollar tells reduce by rule number 4 that is T producing f. So, this F is taken out, T is put into the stack and the now thus, this 3 is also gone; so, 6 and T. So, 6 T is 9. So, it puts the state 9 into the stack and from 9 dollar; so, 9 dollar it says reduce by rule number rule number 1. That is E producing E plus T. So, it will reduce it by this rule. So, E is. So, this E plus T is taken out and so, 1 2 3 symbols. So, 1 2 3 states are taken out.

Now 0 and E, so 0 E says 1. So, it is 1 here; E here and this is dollar. So, this one dollar says accept. So, it will come to the accept state. So, in this particular case; so you can think that as if I have got for understanding this is in a better fashion. So, you can we will we can assume that as if we have got a stack of states. So, this is state stack and we have got a symbol stack. So, whenever we are popping out something from the symbols stack, we have to pop out equal number of entries from the state stack and when whenever we are doing a push operation or shift operation, the state is shifted here and the symbol is shifted on to these stack.

In some sometimes, so we can also visualize it like this that we have got a single stack, we have got a single stack where we push both the symbols and stack; then the another stack states ok. Then alternately I have got a symbol. So, this is this maybe say id and this maybe some state number 0. This maybe plus; then this maybe state number say 2. So, like that. So, I have got alternate between the symbol and the state. So, that can also be done. So, in many books we will find that the convention followed is that way. So, it is a single stack where we have got the alternation of symbols and stack the symbols and states and so that whenever we are talking about popping out, so we pop out twice the number size of the right hand side; twice the size of the production. So, like this.

So, if you are applying say for example, say this rule T producing T star F. So, in the stack you have a situation where you have got this F after that some state number, then star; after that some state number and then T; after that some state number. So, then you are so you have to pop out all these entries from the stack ok. So, that way it is 2 into size of the right hand side. So, 2 it; so, total number of entries popped out is 1 2 3 4 5 6. So, 6 entries are to be popped out. So, many books we will find that it will say you take out 2 into number of symbols on the right hand side of the production from the stack.

So, both are correct. So, either you can consider two different stacks; one for state; one for symbol or you can take it as if there is a single stack where both symbols and stacks are put into symbols and states are put into and it just whenever you are popping out will be taking out twice of that. So, many times we will be following the other convention as well in our later classes for doing this comparison taking while we take some examples maybe we will be doing the other way.

(Refer Slide Time: 27:03)

Constructing SLR parsing table

- Method
  - Construct  $C = \{I_0, I_1, \dots, I_n\}$ , the collection of LR(0) items for  $G'$
  - State  $i$  is constructed from state  $I_i$ :
    - If  $[A \rightarrow \alpha \cdot a \beta]$  is in  $I_i$  and  $\text{Goto}(I_i, a) = I_j$ , then set  $\text{ACTION}[i, a]$  to "shift j"
    - If  $[A \rightarrow \alpha \cdot]$  is in  $I_i$ , then set  $\text{ACTION}[i, a]$  to "reduce  $A \rightarrow \alpha$ " for all  $a$  in  $\text{follow}(A)$
    - If  $[S' \rightarrow S \cdot]$  is in  $I_i$ , then set  $\text{ACTION}[i, \$]$  to "Accept"
  - If any conflicts appears then we say that the grammar is not SLR(1).
  - If  $\text{GOTO}(I_i, A) = I_j$  then  $\text{GOTO}[i, A] = j$
  - All entries not defined by above rules are made "error"
  - The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow \cdot S]$

So, for constructing SLR parsing table, so first of all we have to construct the collection of LR 0 items for  $G'$ .

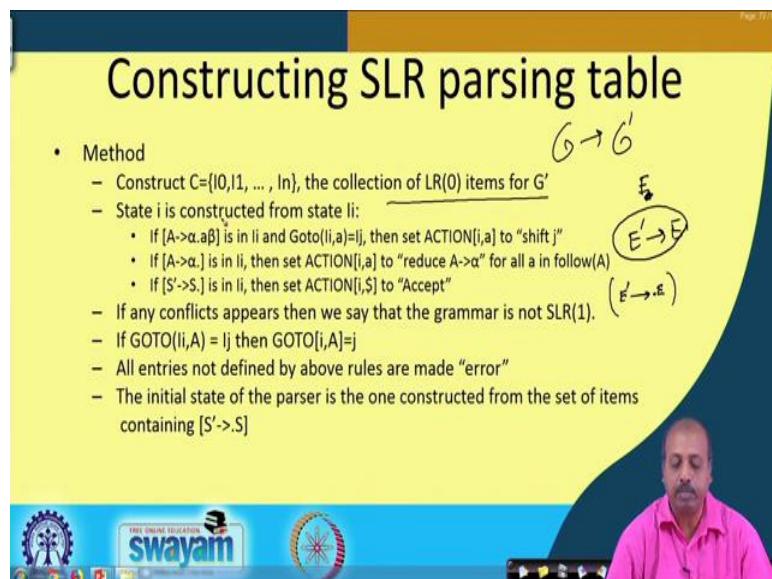
And this state is constructed like this. So if  $A$  producing  $\alpha \cdot a \beta$  is in a particular item and  $\text{Goto } I_i \text{ a is } I_j$ , then we have to set action  $i, a$  to shift  $j$  and now we have if we have got  $A$  producing  $\alpha \cdot a$  production, then that is in the set of item in the item  $I_i$ , then one action  $i, a$  so, it will be reduced by  $A$  producing  $\alpha$  for all  $a$  in the follow of  $A$ . So this basically it is like this that if you have got some in some sentential form where in this portion, you have got other string  $\alpha$ . That is you have already seen something which is derivable from  $\alpha$ . Now if you are attested where  $A$  producing  $\alpha \cdot a$  is there; then, after this you are expecting to see something which is which can follow the symbol is.

So this part whatever comes. So it is in the follow set of  $A$ . So, that is why this rule that is you look into the follow set of  $A$  and for all those symbols you add the action reduced and this and if this is the case where it is the when you are going to do this particular reduction is  $S'$  producing  $S \cdot$ . So, it is going to be the accept state. So we will discuss about this parsing table construction strategy in detail in the next class.

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

**Lecture – 29**  
**Parser (Contd.)**

(Refer Slide Time: 00:15)



**Constructing SLR parsing table**

- Method
  - Construct  $C = \{I_0, I_1, \dots, I_n\}$ , the collection of LR(0) items for  $G'$
  - State  $i$  is constructed from state  $I_i$ :
    - If  $[A \rightarrow \alpha, a\beta]$  is in  $I_i$  and  $\text{Goto}(I_i, a) = I_j$ , then set  $\text{ACTION}[i, a]$  to "shift  $j$ "
    - If  $[A \rightarrow \alpha, \cdot]$  is in  $I_i$ , then set  $\text{ACTION}[i, a]$  to "reduce  $A \rightarrow \alpha$ " for all  $a$  in  $\text{follow}(A)$
    - If  $[S' \rightarrow S, \cdot]$  is in  $I_i$ , then set  $\text{ACTION}[i, \$]$  to "Accept"
  - If any conflicts appears then we say that the grammar is not SLR(1).
  - If  $\text{GOTO}(I_i, A) = I_j$  then  $\text{GOTO}[i, A] = j$
  - All entries not defined by above rules are made "error"
  - The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow S]$

So, next we will be looking into how to construct the SLR parsing table. So, first we have already seen the technique. So, we will formally put it in the form of an algorithm. So, first we construct the LR 0 set of items sets of items. So, this is the first step. So, it first convert the grammar  $G$  into an augmented grammar  $G$  dash by adding the extra start symbol, like if the grammar start symbol is say  $E$  then we add another extra production  $E$  dash producing  $E$ .

With the idea that when the whole when the parser will try to do a reduction by this particular rule. So, it will mean that the string has been parsed successfully. So, that way we start with the item like  $E$  dash producing dot  $E$  and that is and we take the closer of this. So, that will make the set  $I_0$  and from this set  $I_0$ . So we will be seeing on various input symbols input symbols and grammar non-terminals also, so will be having other items produced. So,  $I_0, I_1$ , suppose when we construct the items. So we get these items like  $I_0, I_1$  up to  $I_n$ . So, that is there though they that is the collection of LR 0 items for the grammar.

Now, from this, so from the state  $i$  will be constructing the set  $i$ . So, for each of these  $i$  for each of these  $I_0 I_1$  etcetera, so they will constitute one state. So, this will constitute the state  $S_0$ , this will constitute the state  $S_1$  so like that.

(Refer Slide Time: 01:54)

## Constructing SLR parsing table

- Method
  - Construct  $C = \{I_0, I_1, \dots, I_n\}$ , the collection of LR(0) items for  $G'$
  - State  $i$  is constructed from state  $I_i$ 
    - If  $[A \rightarrow \alpha \cdot a \beta]$  is in  $I_i$  and  $\text{Goto}(I_i, a) = I_j$  then set  $\text{ACTION}[i, a]$  to "shift  $j$ "
    - If  $[A \rightarrow \alpha \cdot \beta]$  is in  $I_i$ , then set  $\text{ACTION}[i, a]$  to "reduce  $A \rightarrow \alpha$ " for all  $a$  in  $\text{follow}(A)$
    - If  $[S' \rightarrow \cdot s_i]$  is in  $I_i$ , then set  $\text{ACTION}[i, \$]$  to "Accept"
  - If any conflicts appears then we say that the grammar is not SLR(1).
  - If  $\text{GOTO}(I_i, A) = I_j$  then  $\text{GOTO}[i, A] = j$
  - All entries not defined by above rules are made "error"
  - The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow \cdot S]$

Page 73/10

And we will have a from this state once we call this the states, so we will be looking into the go to parts. So, we have previously seen how to construct the go to of symbol of a particular state on some terminal and non-terminal symbol and we have also seen like how to make get the closer.

So, if this  $A$  producing alpha dot a beta is an item in a particular set of items then that is in  $I_i$ . So, if I have got a item like  $A$  producing alpha dot a beta and the go to  $I_i$  says it is a new item  $I_j$ . Then in the action part of the table at the corresponding to state number  $i$  and input  $a$ , we will at the action shift  $j$ ; that means, it will do a shift operation and it will come to a new state which is  $j$ . So, that is the meaning of this rule. So, we can, we can look into a, we can look into a particular example like say this one.

(Refer Slide Time: 03:13)

The screenshot shows a parser table and a stack trace. The table has columns for State, Action, and Goto. The stack trace on the right lists actions taken:

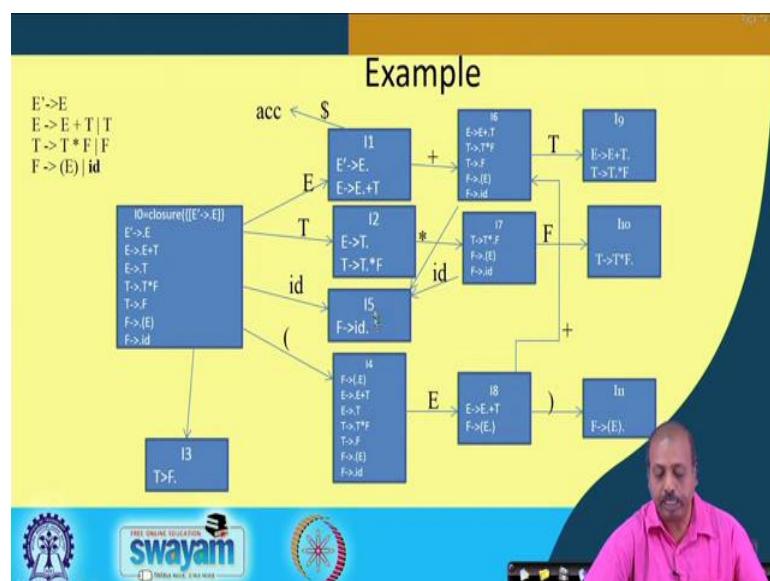
- (0)  $E' \rightarrow E$
- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T * F$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow (E)$
- (6)  $F \rightarrow id$

A handwritten note on the slide says  $f\cup(T)=\{*, +\}$ .

STATE	ACTION					GOTO			
	id	+	*	(	)	\$	E	T	F
0	S5						1	2	3
1		S6							
2		R2	S7		R2	R2			
3		R4	R4	*	R4	R4			
4	S5		S4				8	2	3
5		R6	R6		R6	R6			
6	S5		S4				9	3	
7	S5		S4						10
8	S6		S11						
9	R1	S7	R1	R1					
10	R3	R3	R3	R3					
11	R5	R5	R5	R5					

So, while constructing this table from I from the state I 0, so I0 is having E dash producing dot E and the closer of that. So, if from there we will get it is a dot E will have the id also? So, on id it will do a shift and it will come to state 5. So, if we just go back a few slides say on I0 on id it comes to the state I5 comes to the item I5. So, accordingly I will be adding this entry in the table shift and the new state is 5. So, that is the way this table is constructed for the shifting part.

(Refer Slide Time: 03:36)



Now, if you have got a rule like this a producing alpha dot if you have an item in the set Ii like A producing alpha dot then we will first have to see what is the follow of this non-terminal A and for each of the symbols small a in the follow set of A. So, we will be adding this production reduce by A producing alpha. To understand this thing, so let us look into how do you for example, how do you get this particular rule, how do we get this particular reduction.

So, this as I 3 and on plus it is R 4. So, I 3, so it means the reduced by rule number 4. So, rule number 4 is this one, T producing F. So, I have to see what is the follow set of T. So, follow set of T, follow of T is given by. So, T is followed by this star because by this rule, so this is followed by star and by this rule E producing T whatever is in follow of E is in follow of T. So, I will get plus and here E is followed by closing parenthesis, and also since E is the start symbol. So, dollar was there in the follow set of E.

So, as a result the follow of T will have the star plus close parenthesis and dollar. So, you see for this plus star close parenthesis and dollar we have added the action reduced by rule number 4. So, this is done here. So, this way we can we have to we solve for all the symbols that are coming in the follow set of the terminal non-terminal on the left hand side of the production. So, we have to have this we have to add the corresponding rule there.

Next, for particularly for this S dash producing S dot, where S is the actual start symbol of grammar G and S dash is the start symbol of the augmented grammar G dash. So, if S dash producing S dot giving S dot if this item is there; that means, I have I am trying to do a reduction by this particular, we are trying to do a reduction by this rule S dash producing S. So, if in a particular item you have this thing then action i dollar is set to accept, means action i dollar means I am currently in state I, state i and the next input symbol is dollar. So, input is or the input string is also exhausted and I am trying to do this reduction. So, that is naturally the accept state.

So, so the main difficulty here is one thing is that you have to calculate this items carefully, this item calculations, so that is one job and the other job is to compute the follow of all the all the symbols or the of all the non-terminal symbols that we have in the grammar wherever you will be requiring this A reproducing alpha this type of rule, so you will need to have the follow set of A.

Now, if there is any conflict like while doing this putting this actions shift and reduce. So, if there is a conflict then the grammar is not an SLR grammar. So, like say in the previous example that I that we took. So, if we look into the action part you see there is no entry where there are two actions are defined. So, everywhere it is either a shift operation or a reduce operation or there is no operation. So, no operation means it is an error, but for the action is accept, so those things are there, but there is the no entry in this table where the actions are multiply defined.

So, this table, this particular grammar is an LR SLR grammar because we could construct the SLR table for them without any problem. And then for the go to part. So, if you look into this table. So, there are two parts, one part is the action part and the other part is the go to part. For the go to part I have got the non-terminals E, T and F, and from individual items individual sets of items. So, on the non-terminal with the go tos have been calculated. So, here the corresponding go tos are added.

For example, from state say I have from an item I<sub>0</sub> on E, T and F let us see where we are going. So, if you look into this particular diagram for I<sub>0</sub> on E it was going to state I<sub>1</sub>, on T it was coming to this one 12 and here it is not marked the level is not marked, but it has come from this T producing F dot, so this is F. So, this I<sub>1</sub>, I<sub>2</sub> and I<sub>3</sub>. So, these are the 3 states to which we can go from the state I naught. So, that is done here. So, you see from the state 0 on getting this E T or F. So, it is moving to the states 1, 2 and 3. So, in this way the go to part will be filled up.

So, once this action part and go to part has been met then we can use the algorithm that we have seen previously this parsing algorithm for parsing the from for passing the input string.

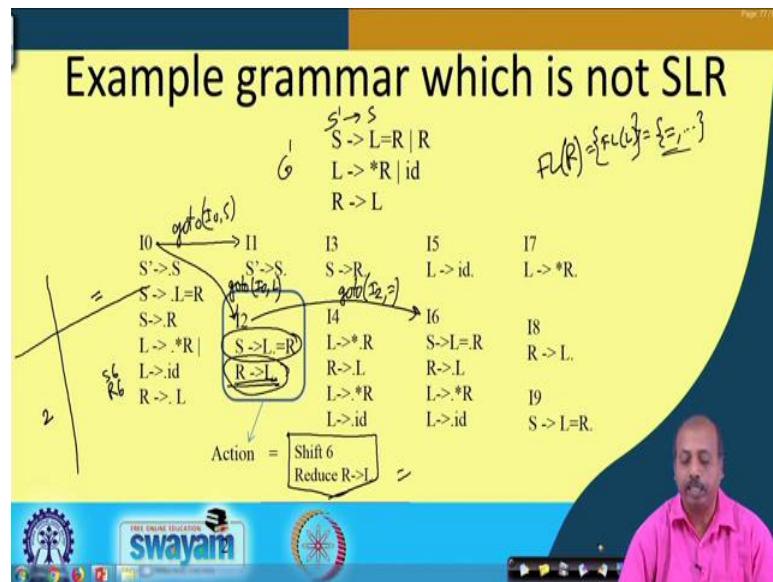
(Refer Slide Time: 09:21)

```
let a be the first symbol of w$;
while(1) /*repeat forever */
    let s be the state on top of the stack;
    if (ACTION[s,a] = shift t) {
        push t onto the stack;
        let a be the next input symbol;
    } else if (ACTION[s,a] = reduce A->β) {
        pop |β| symbols of the stack;
        let state t now be on top of the stack;
        push GOTO[t,A] onto the stack;
        output the production A->β;
    } else if (ACTION[s,a]=accept) break; /* parsing is done */
    else call error-recovery routine;
}
```

And at the end; so, if you are finding that ok, we are at a state where the with the action is accept then parsing is over. If there is an error in the input stream that at some point of time you will be landing into one of these entries which is blank in this table and if the entry is blank that means, the input there is some error, ok. And of course, if you come to this dollar and this things, so the accept state then it is then it is accepted.

And if there are multi some integer are multiply defined then there is problem, as I said that the grammar is not a SLR because the parser will be confused whether to do a shift operation or a reduce operation. Or if may it may so happen that two action both the actions are reduce of actions, but they are by two different rules. So, the person will not to know clearly like which rule it should use. So, that way it can give rise to the shift reduce conflict and reduce conflict. So, those conflicts are to be ideally they should not be there in the in the in the parsing table. So, if for a grammar we have a situation where these conflicts are not there then we will tell that the grammar is an SLR grammar.

(Refer Slide Time: 10:39)



So, let us look into an example which is not SLR, ok. So, let us look into this grammar where we have got this thing this S producing L equal to R o R, L producing star R id, R producing L. So, here this star id and equal to so these are terminal symbols, S, L and R they are non-terminal symbols.

So, initially I will have the state. So, with this I will be adding this additional rule is just producing S to get the augmented grammar G dash. Now, on this augmented grammar we try to construct the sets of items. So, this I0 is S dash producing dot S then the closer of that. So, S producing dot L equal to R, S producing dot R they will be coming from this rule, then L producing dot star R or L producing dot id and again since dot R is there, so R producing dot L will come. So, that is I0. I0 to I1, so S dash, so it is go to I0 S. So, go to I0 S is I1, so which is this one?

Similarly, from go to I0 L, so this is the I0 to I1. So, this is go to I0 S. And similarly, from I0 to I2 it is go to I0 on L. So, that way is the others other items are constructed accordingly. So, I am not going to that explanation but let us look into this item more carefully say this item I2. So, S producing L dot equal to R; that means, address of the first item says that I have seen a portion of the string where from which I can derive L and I am expecting the next input symbol to be equal to.

Now, what is the action? Now since the since it is equal to; so the if the next input symbol is equal to I will do a shift. So, the first part we will of will tell us, first item we

will tell me that. Second item telling me know if you even if you see equal to, so you better do a reduction by this rule, a better do a reduction by this rule R producing L dot. So, you have to look into the follow set of R, so you have to look into the follow set of R and for all the terminals in the follow set of R you have to reduce by R producing L. So, what is the follow set of R? So, follow set of R is equal to; so here by this rule L producing star R. So, whatever is in follow of L is in follow of R. So, follow of L is equal to follow of R follow of R will contain follow of L it may contain something more, but it will have this follow of L and follow of L has got this equality symbol. So, you see among the other symbols this equality symbol we will definitely be there. So, follow of L contains equality.

So, when I am doing the action the first one, the first rules says the, so by this rule S producing L dot R L dot equal to R. So, from I2 on equality it goes to I6. So, to from I2 to I6 this is go to I2 equality, ok. So, this is the shift action. So, so those naturally it tells me if I if I get an equality I should do a shift and I the new state will be 6. So, that is coming from this particular rule.

Now, this rule telling me that you should do a reduction, for which symbols? For all the symbols which are coming in the follow set of R and the follow set of R contains equality, so that means, if you see an equality then you should do a reduction by R producing L. So, this is giving rise to a shift reduce conflict on the symbol equality.

So, whenever if you are trying to draw the parses that have a parsing table then for the state 2 and for the input symbol equality, I will have two entries, one is shift 6 another is reduced by rule number 1, 2, 3, 4, 5, 6, reduced by rule number 6 ok, that way I will have two actions. So, so this is the, this particular grammar that is given is not an SLR grammar because it has got multiply defined entries for some of the multiply defined values for some of the entries. So, that has to be taken care of.

So, what is the solution? Like they, so one solution is that you modify the grammar. So, you modify the grammar whether it will catch the language syntax properly or not that is a question. If it if you can modify the grammar and make it SLR, it is well and good. So, if we cannot then we have to go for more powerful parsing strategy like say we have to go for something called say LR parser, called canonical LR parser which is more

powerful than SLR parser, but it will have more complexity. The complexity of this SLR parser we will see that it is going to be much more compared to this SLR parser.

(Refer Slide Time: 16:35)

The slide has a yellow header with the title "More powerful LR parsers". Below the title are two bullet points:

- Canonical-LR or just LR method
  - Use lookahead symbols for items: LR(1) items
  - Results in a large collection of items
- LALR: lookahead symbols are introduced in LR(0) items

A small diagram shows a horizontal line with a bracket above it containing the symbol "A\$". An arrow points from the "A" in the bracket to the "A" below it, indicating the lookahead symbol and the current input symbol. The slide footer includes the Swayam logo and navigation icons.

So, they are known as canonical LR, sometimes it is called C LR or sometimes it is called just LR. And this SLR parser is often called LR 0 parser, ok. So, in case of LR this canonical LR parser we use some look ahead symbols for items. So, they are called LR 1 items. So, if they are not LR 0 items, so they are called LR 1 items. So, from the current position you do not only look at the current input symbol. So, what it means is this thing that if this is your, if this is your input stream then at present suppose you are at this position, so your input pointer is here.

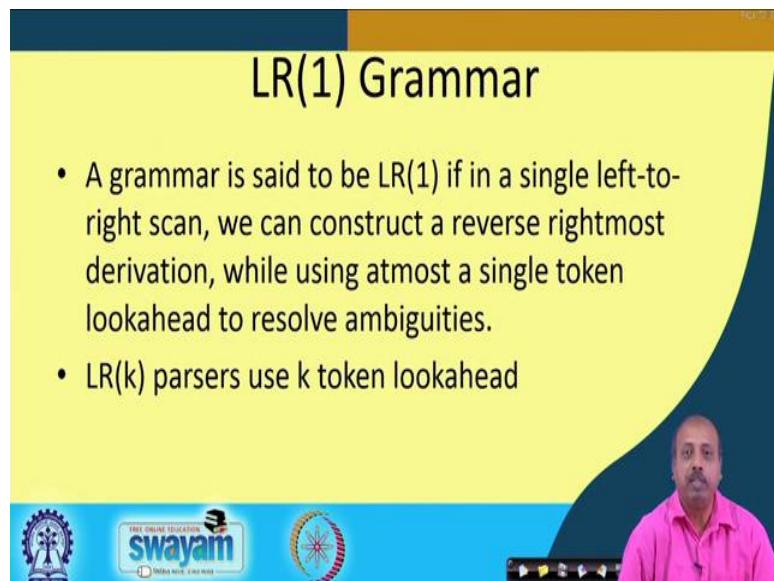
So, in case of LR parser, SLR parser we were taking a decision based on this particular input symbol only but in case of LR 1 parser will be looking ahead by one more symbol. So, we will be looking into the next symbol also and accordingly take a decision. So, if we do that, then many of these conflicts that we are getting say shift reduce or reduce conflicts, so they may get dissolved as we are doing a lookahead. So, but the difficulty is that it will result in a large collection of items. So, that way the complexity of the parser will be very high.

And there is another variant which is known as LALR parser while lookahead symbols are introduced in the LR 0 items. So, here also it is there, but here the number of states will be much less compared to this canonical LR parser. So, SLR parser foremost of many

grammar so you can do the parser construction by hand. Canonical LR parser, so it becomes difficult because with as you are increasing the complexity of the grammar, so number of states will become pretty high.

And LALR also since it is depending on these LR 1 items, so it will be again be difficult, it will be difficult to construct by hand, but more, but they can be automated very easily. So, we have we will see that the automated tools for this parser generator, so they will generate LALR parsers because of the reason that LALR parser ultimately the number of states will be same as LR 0 or SLR parser. But power wise it will be as powerful as the canonical LR or LR 1 parser. So, we will see how they are done.

(Refer Slide Time: 19:12)



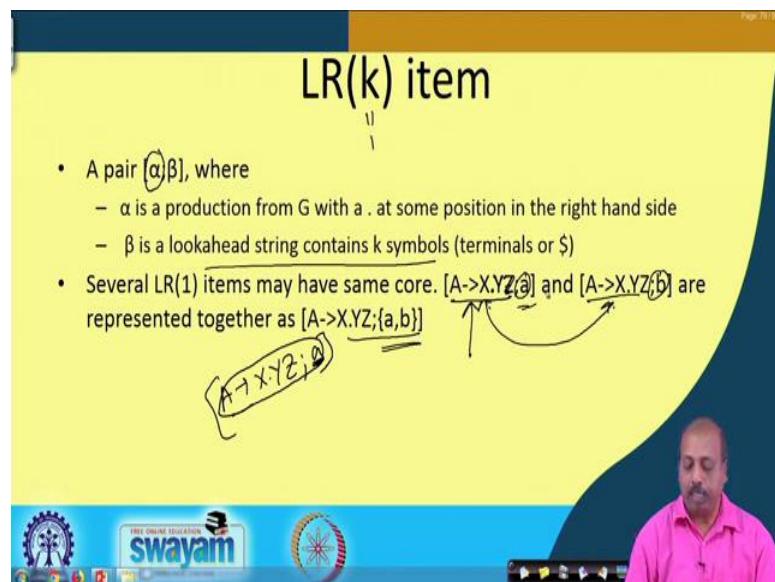
So, let us look into the LR 1 grammar. A grammar will be said to be LR 1 if in a single left right scan we can construct a reverse rightmost derivation while using at most single token lookahead to resolve ambiguities. So, in case of LR 0, so we were not doing any lookahead. In case of LR 1 we are doing lookahead by at most one token. So, you may or may not require to take a your may no may or may not need to base your decision on the next input, on the next input symbol may be at current input symbol only you can take a decision. But in case of confusion, ok. So, you can you are allowed to look into the next token and then take a decision.

So, and it will just like any other this LR parsers, so it will be doing a reverse rightmost derivation. So, it is a bottom up parser and it will be doing a rightmost derivation. So, at

every state the rightmost non-terminal symbol will be replaced by the corresponding right hand side of the production rule.

In general LR k parsers that will they will be using k token lookahead. So, they will be more and more powerful. So, as you are trying to do by allowing more and more lookahead in the parser becomes more and more powerful. However, the parser complexity we will also increase significantly. So, normally we go with this LR 1 parser, we do not go beyond that because most of the programming languages and the constructs that we have, so LR 1 is good enough. So, we do not need to go beyond that, ok.

(Refer Slide Time: 20:50)



So, what is an LR k items. So, for all our discussion, so we will be taking k equal to 1. So, you can just generalize it to other values of k. So, this LR k item, so it is a sphere alpha semicolon beta, where alpha is a production from G of the grammar G with a dot at some position in the right hand side. So, this alpha part is similar to what we have in the LR 0 item. So, this alpha part is same as that one.

For example, if you have got say a producing X dot Y Z. So, this was an item in case of LR 0 parser. But in case of LR 1 parser there will be another look ahead string that contains k symbols, at most there, so lookahead string that will contain k symbols. So, if I am using k equal to 1; that means, I will have a single set terminal symbols after this. So, this is this is this is an item, A producing X dot Y Z semicolon a. So, what is the meaning of that will see.

So, basically this part will be giving me the lookahead. So, what is what we are going to do a lookahead. Now, as I am doing this since the items they have got two parts one part is a rule with a dot somewhere and the other part is a terminal symbol or a terminal string then there may be several such items where the rule part is same, ok, rule with dot, so that part is same what differs is the beta part. So, though they will they are said to said to have same core. So, this one, so this is called the core and maybe. So, these two items if you see both of them having are having X producing, A producing X dot Y Z as the core. So, they have got the same core, but the next item that next lookahead token is different here it is A, here it is B.

Now, if I have got say a situation where this thing occurs then many a time will be writing in a short hand form like this A producing X dot Y Z semicolon then within brace. So we write all the alternative strings for which this is an item. So, a in this particular case, so we have got two such items a producing X dot YZ semicolon a, A producing X dot Y Z semicolon b. So, we combine them together while writing as A producing X dot Y Z semicolon a comma b.

So, this way we will be representing LR k items, where we remember that this part is meaning of the core part is same as what we have done what we have in case of LR 0 item. So, this means that we have for examples say this one A producing X sot Y Z this means we have already seen a string which is derivable from X and after that this dot is there; that means, we are expecting to see a string which is derivable from this Y Z, ok. So, that was the meaning. So, this is fine.

So, this lookahead part, lookahead token, so this will be clear after sometime. So, we will be explaining why this is real, what is this and how is it going to help us.

(Refer Slide Time: 24:43)

## Usage of LR(1) Lookahead

- Carry them along to allow choosing correct reduction when there is any choice
- Lookaheads are bookkeeping, unless item has a . at right end
  - In  $[A \rightarrow X.YZ; a]$ , a has no direct use
  - In  $[A \rightarrow XYZ.; a]$ , a is useful
  - If there are two items  $[A \rightarrow XYZ.; a]$  and  $[B \rightarrow XYZ.; b]$ , we can decide between reducing to A or B by looking at limited right context

So, as such for most of the time. So, this is like a baggage that we carry with the items. So, this look ahead item, so we carry them along to carry them along to allow choosing correct reduction when there is any choice. So, this is particularly, this look ahead will be particularly useful when we are going to have this reduction rules. So, in that case when there is a confusion in the reduction process, so then we will be taking help of that. So, we will either using that reduction based on the two or more these tokens, this look ahead tokens; so based on that we will take a decision. Whether to reduce by one rule or the other or whether to do a reduction or not.

Then, so look heads are basically bookkeeping, so that way. So, they are bookkeeping. So, unless we have got an item that has got a dot at the writing. So, like if you consider say these two rules these two items, so for this item there is no directive. So, from this from this item, so we can try to generate say go to the item and y. So, we will be simply where may we will be simply getting an item XY dot Z and this part will be carried forward, this a part this token part. So, this will be carried forward. So, it is a lookahead. So, it is just a bookkeeping, whereas, for this particular case where this dot is at the end then it is going to be useful.

How is it going to be useful? Suppose I have got I am at a state say I5 where I have got these two items. So, A producing XYZ dot semicolon a and B producing XYZ dot semicolon b. So, previously what we were doing? So, if we have to do a reduction then

this rule tells me that for all the symbol all the terminal symbols for the in the follow set of A, I have to do reduction by this rule and this rule is telling me that for all the all the terminals in the follow set of B I have to do by this reduction.

Now, if the grammar is such that this follow set of A and follow setup of B, they have got some terminals common between them then this will immediately give rise to a reduce conflict. Now, what this A and B is going to do is that it is trying to modify the context. So, it will see whether the next input symbol is A or not and this will see the next input symbol is B or not, that is a look ahead token. So, the lookahead token is a then only this reduction will be done. If the look ahead token is B then this reduction will be done. And since these two tokens are different they are not same token then we can definitely take a decisions, that read that reduce conflict will get resolved in this fashion.

So, we can decide between reducing to A or B by looking at limited right context. So, we are not doing very large amount of look ahead, we are looking ahead by a single symbol single token and then we are taking a decision. So, that way it is going to be powerful. So, if the if the follow set of A and B are totally disjoined from each other then of course, we were the it would have been an LR 0 grammar, so it would have been an SLR grammar. But if they are not disjoint the A and B, follow set of A and B if they are not disjoint then this grammar will not be an SLR grammar. But with this is if this look if this look ahead tokens are different then this grammar is definitely an LR 1 grammar. So, this way it will be helpful, in this look ahead token they can be utilised for resolving the reduce conflicts between a number of rules.

So but the problem is that you see that similar items, so they will go on coming only the look ahead part may be different, as a result they will give rise to large number of states or large number of items and that is the thing. So number of states that you get with this LR 1 parsing policy it will be much larger compared to the LR 0 parsing policy.

And this is exactly the reason why this thing happens, because if we had relied only on this rule part this dot this left side. So, the A producing X dot YZ then number of states in SLR and this canonical LR they would have been same the same, but because of this look ahead token so they are going to be different and they are going to have more number of states in case of this canonical LR parser.

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

**Lecture – 30**  
**Parser (Contd.)**

So, next we will be looking into the closure algorithm. So, for this LR 1 items the closure algorithm will be similar to what we have the closure algorithm for this LR SLR parsing.

(Refer Slide Time: 00:29)

```
SetOfItems CLOSURE(I) {
    J=I;
    repeat
        for (each item [A-> α.Bβ;a] in J)
            for (each production B->γ of G and each terminal b in First(βa))
                if ([B->.γ;b] is not in J)
                    add [B->.γ;b] to J;
    until no more items are added to J on one round;
    return J;
}
```

So, here but only thing is that these items have got this additional part. So it has got this look ahead token part, so, this look ahead tokens will also appear in the closure state. So, for an item I so, if we are trying to get a closure so, for I we are trying to get a closure we initialize this set J to I and for each item a producing alpha dot B beta semicolon a in J in. So, this should be in I this is not in J in I. So, we have to see whether there is a grammar rule which has got the form B producing gamma.

So, if there is a rule like B producing gamma then each terminal for each terminal B in the first of beta a. So, when should we go by this rule? When should we go by this rule? So, this go to B; so, this go to B we will see we will like to see if we can we see that there is something starting with beta followed by a. So, if there is something like that then only we will be using that particular transition in the parser. So, that is what is done

here, so, for every production rule so, which is of the form B producing gamma then will be so for each then for each terminal B in the first of beta a so we are doing this B producing dot gamma semicolon b it is not in J this is not yet added into the into the set J. So, we add this B producing dot gamma B to semicolon b to J.

So, that way it is just otherwise it is same only thing is that for. So, previously we did not for the LR 0 item we did not have look ahead, so, there was this part was absent, so, this part was absent. So, we were just doing it like this for each production B producing gamma of J we were adding this B producing dot gamma into J. So that is what we were doing, but now I have to tell what the look ahead part is also. So, just to do that, so, look ahead part is decided by this by you compute the first of beta a and whatever symbols are coming in the first of beta a, so, we have to add this thing into this set J.

So, this way the process will continue till no more items can be added to J then will be returning J. So, that is the closure computation algorithm, so, the we can use this for getting the closure of items.

(Refer Slide Time: 03:08)

The slide has a yellow header with the title "GOTO algorithm". Below the title is the pseudocode:

```
SetOfItems GOTO(I,X) {
    J=empty;
    if ([A-> α.Xβ;a] is in I)
        add CLOSURE([A-> αX.β;a]) to J;
    return J;
}
```

The slide footer features three logos: the Indian Institute of Technology Madras logo, the Swayam logo ("FREE ONLINE EDUCATION SWAYAM"), and the Ministry of Human Resource Development (MHRD) logo.

And then we can see the GOTO part. So, GOTO part is also a similar. So, there is initially, so, we are trying to find this GOTO of I X. So, I is an set of items and X is the grammar symbol then goes initially J is equal made equal to empty. Then if there is an item like A producing alpha dot X beta semicolon a in I, then we take the closure of A producing alpha X dot beta semicolon a and then we add that item to J. So, this way we

take the closure and add all those items to J and then it will be returning J. So, this way we can constitute the closure set the GOTO part for the this LR 1 items.

(Refer Slide Time: 04:05)

**Constructing LR(1) Parsing Table**

- Method
  - Construct  $C = \{I_0, I_1, \dots, I_n\}$ , the collection of LR(1) items for  $G'$
  - State  $i$  is constructed from state  $I_i$ 
    - If  $[A \rightarrow \alpha, a; b]$  is in  $I_i$  and  $\text{Goto}(I_i, a) = I_j$ , then set  $\text{ACTION}[i, a]$  to "shift j"
    - If  $[A \rightarrow \alpha, a]$  is in  $I_i$ , then set  $\text{ACTION}[i, a]$  to "reduce  $A \rightarrow \alpha$ "
    - If  $[S' \rightarrow S, \$]$  is in  $I_i$ , then set  $\text{ACTION}[i, \$]$  to "Accept"
  - If any conflicts appears then we say that the grammar is not LR(1).
  - If  $\text{GOTO}(I_i, A) = I_j$  then  $\text{GOTO}[i, A] = j$
  - All entries not defined by above rules are made "error"
  - The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow S, \$]$

*Journal ( $S' \rightarrow S$ )*

Page 30 of 30

FREE ONLINE EDUCATION  
**swayam**

Next we will see how, next we will see how to construct the LR 1 passing table. So, for constructing LR 1 parsing table so, the policy is same as we have done in LR 0 parsing table. So, we first construct the collection of LR 1 items C is the  $I_0, I_1$  up to  $I_n$  for the augmented grammar  $G'$  and then state  $i$  is constructed from this item  $I_i$  like this. So, if  $A$  producing alpha dot a beta comma semicolon b is in the item is in the set  $I_i$  and  $\text{GOTO } I_i \text{ a is } I_j$  then action  $i, a$  is said to be equal to shift  $j$ . So, this part there is no change, so, this is exactly same as what we have for the LR 0 parser, the SLR parser.

But, this one it says that if you have got a rule like  $A$  producing alpha dot semicolon a is in  $I_i$  then set action  $i, a$  to reduce by  $A$  producing alpha. So, here you note that we are not looking into the follow set of  $A$ . So, follow set of  $A$  whether it can contain small a or not etcetera those things are not need not be computed. Definitely the follow set will if it is a valid thing then the follow of  $A$  will contain small a, if the if my string is valid then the follow of capital  $A$  will definitely contain small a that is why this after doing this reduction. Because the string was something like this so, this part was alpha and then one look ahead token was a and we said that we will reduce by this whole thing by  $A$ ; that means, in some sentential from this small a will appear after capital  $A$ .

So, small a is definitely there in the follow set of A, but a may be some symbols for some cases it will be something different. So, we will be doing this reduction only when we are getting this i, a. So, in case of LR 0 parser so, this reduction rule was put for all the symbols which are in follow of A follow of capital A, but in this LR 1 parser. So, we are adding the rule only to the case where the input symbol is small a.

And then S dash producing S dot semicolon dollar; so, if it is there in some i sets of set of item then, we said this action i dollar to accept. So, this is the; this is the simple way the same it is same as that SLR parsing and if there is any conflict that appear then we say that the grammar is not it is not a SLR so, this is not LR 1. So, S should not be there, so, if there is any conflict even after doing this if there is a conflict then the grammar is not LR 1 grammar.

So, but unfortunately we do not know grammars which are parsers which are more powerful than this S the canonical LR parser. So, if this thing fails; that means, you really need to work with your grammar and try to do something or bring some context sensitivity into the parsing process. So, you by means of context free grammar, so, you will not be able to generate a parser for the particular language.

So, we will see that if you look into this parser generated tool Yacc then Yacc, bison etcetera. So, those tools so, they apart from this LALR or LR 1 parsing table generation. So, it they also allow several context dependent features and those context dependant features may be used for resolving the conflicts that are occurring in an LR 1 parser. But, anyway with a for the time being we assume that once we have come to this LR 1 parser since it is very powerful so, it has it will be able to resolve whatever constructs we have in our programming language.

And then the GOTO part is same as what we have in a LR 0 parser. So, if GOTO Ii, A is equal to Ij then we say this GOTO i A equal to j and all other entries which are not defined. So, they are they will be marked as error and initial state of the parser is the one that is constructed from this one, S dash producing dot S semicolon dollar. So, this dollar is the um end of string character or the so, then this is the previously in case of LR 0 parsing table construction so, we took the item S dash producing dot S and we took the closure of that. But, here I have to tell the look ahead token also, so, here the look ahead token is dollar. So, that is the additional thing that we have.

So, this way we can have this LR 1 parsing table constructed and we can have otherwise the parsing algorithm is same, so, that is same with the what we have in the LR the parsing 1.

(Refer Slide Time: 09:28)

Non-terminal	LR(1) items
goal -> expr	$I_0 : [goal \rightarrow expr\$]$ , $[expr \rightarrow term + expr\$]$ , $[expr \rightarrow term\$]$ , $[term \rightarrow factor * term, \{+, \$\}]$ , $[term \rightarrow factor, \{+, \$\}]$ , $[factor \rightarrow id, \{+, *, \$\}]$
expr -> term + expr	$I_1 : [goal \rightarrow expr\$]$
expr -> term	$I_2 : [expr \rightarrow term\$]$ , $[expr \rightarrow term + expr\$]$
term -> factor * term	$I_3 : [term \rightarrow factor, \{+, \$\}]$ , $[term \rightarrow factor * term, \{+, \$\}]$
term -> factor	$I_4 : [factor \rightarrow id, \{+, *, \$\}]$
factor -> id	$I_5 : [expr \rightarrow term + expr\$]$ , $[expr \rightarrow term + expr\$]$ , $[expr \rightarrow term\$]$ , $[term \rightarrow factor * term, \{+, \$\}]$

So, let us take an example suppose we have got a set of grammar rules like this. So, this goal producing expression producing term plus expression producing term and then this term producing expression a factor star term produce a factor produce id. So, this is the grammar G and we have added this additional rule goal producing expression to make the grammar G dash the augmented grammar G dash.

Now, I set that the first rule the first state or the first set of item I 0 is the closure of this goal producing dot expression and the symbol the look ahead token is dollar. So, I am expecting a string I am expecting to see a string expression and that will be followed by dollar. So, that the next input symbol will be dollar; so that way this is done and then you have to take the closure of that.

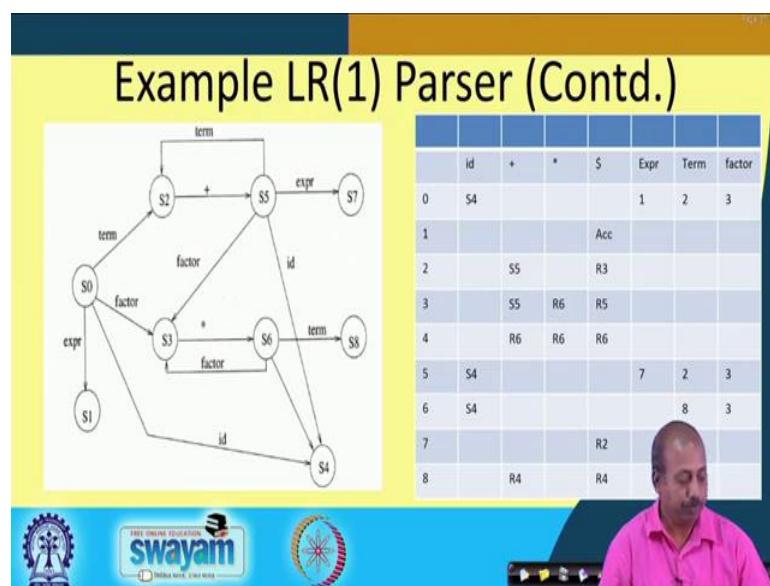
So, this dot expression is there. So, expression producing dot term plus expression so, this will be created these dollar is carried forward then this expression producing dot term is taken, so, this dollar is carried forward. Term producing now expression producing dot term is there. So, based on that this term producing dot factor star term and but here now this if you look into this one this expression producing dot term factor dot term comma dollar then this from this dot term you will see that it will be producing

dot factor star term and then from this dot factor it will be giving me this also it will be giving me a term producing factor and then this term producing dot factor and then this expression producing term. So, this will be giving me expression producing dot term, so, this one will come.

And many of these symbols like plus and dollar. So, they will come both of them will come because another item will be created where this plus will also be coming because their core part will become same that way it will come.

Similarly, from I1, so, this is goal producing expression dot will come. From I2 it will be expression producing term dot and then. So, from this rule expression producing dot term dollar, so, it will give me expression producing term dot and this expression it will give me from this item. So, it will give me expression producing term dot plus expression dollar, so, this way these items the LR 1 items will be constructed for this particular grammar.

(Refer Slide Time: 12:26)

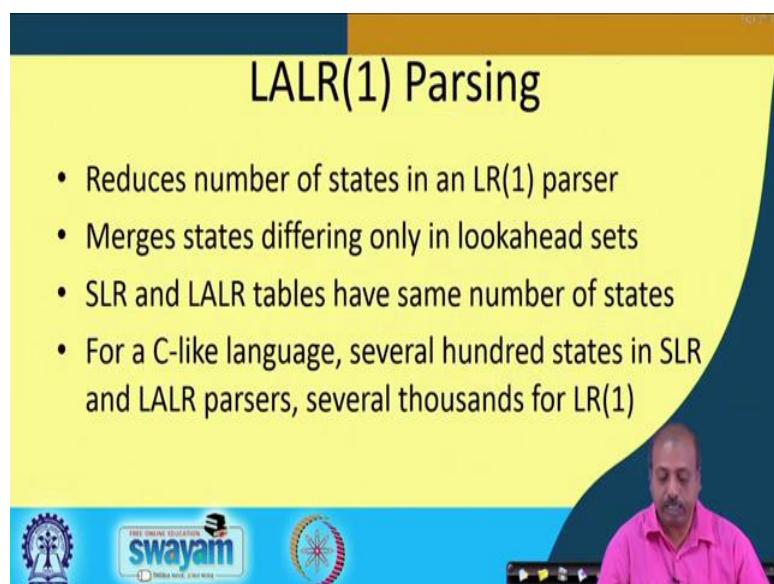


Next once these items have been constructed so, we can construct the corresponding table so, its corresponding automata will be like this. So, S0, S1, S2 so, these are the various states and then based on this rule that we have seen previously that parsing table construction rule so we can construct this table. So, this way so, for from state 0; so, on id so it is so, it from so it is going to state it is on id it is coming to state 4. So, this is the

that is id it is shift 4 and then on this term factor and expression so, it is going to the states 1, 2 and 3, so, accordingly this GOTO part is put like this.

So, this way we can construct this table. So, this table construction is same as what we have in the previous case like this the SLR parsing table for that how we whatever way it is constructed so, that they will be coming like that.

(Refer Slide Time: 13:29)



So, once this is done so, we can this table is ready, but you see that the number of states will be much more compared to this LR 0 parser or SLR parser. Of course, for this particular case we do not have much number of states, but it may so happen in many a cases that number of states are much higher.

So, this is LALR parsing algorithm, so, that is also doing a look ahead by one token. So, they are called LR 1 they are basically LR 1 parsing, but with look ahead they are called look ahead LR that is why it is called LALR. It reduces number of states in an LR 1 parser by merging states that differ only in the look ahead sets. So, if there are a number of states that they differ only in the look ahead sets then it will merge all those states together and call them as one state.

So, as a result this SLR and LALR tables they may they will have the same number of states. Though we are not proving this result formally so, it can be shown that this SLR and LALR parsers they will have more number of states whereas, the LR 1 parser will

may have large number of states. So, for a C-like language; so, there are several hundred states in the SLR and LALR parsers whereas, for LR 1 parser so, it may be one order of magnitude mode. So, several thousand states may be there in the LR 1 parser.

So, this way we can have this thing we can have this LR 1 parsers converted to LALR parser. So, we will see how to do that so that the number of states are reduced, ok.

(Refer Slide Time: 15:22)

$S' \rightarrow S$   
 $S \rightarrow CC$   
 $C \rightarrow cC \mid d$

LR(1) items

- $I_0 : [S' \rightarrow S, \$], [S \rightarrow CC, \$], [C \rightarrow \cdot cC, \{c, d\}], [C \rightarrow \cdot d, \{c, d\}]$
- $I_1 : [S' \rightarrow S, \$]$
- $I_2 : [S \rightarrow C \cdot C, \$], [C \rightarrow \cdot cC, \$], [C \rightarrow \cdot d, \$]$
- $I_3 : [C \rightarrow c \cdot C, \{c, d\}], [C \rightarrow \cdot cC, \{c, d\}], [C \rightarrow \cdot d, \{c, d\}]$
- $I_4 : [C \rightarrow d, \{c, d\}]$
- $I_5 : [C \rightarrow CC, \$]$
- $I_6 : [C \rightarrow c \cdot C, \$], [C \rightarrow \cdot cC, \$], [C \rightarrow \cdot d, \$]$
- $I_7 : [C \rightarrow d, \{c, d\}]$
- $I_8 : [C \rightarrow cC, \{c, d\}]$
- $I_9 : [C \rightarrow cC, \$]$

States  $I_4$  and  $I_7$ ,  $I_3$  and  $I_6$ ,  $I_8$  and  $I_9$  can be merged

$I_4 : [C \rightarrow d, \{c, d, \$\}]$   
 $I_{36} : [C \rightarrow c \cdot C, \{c, d, \$\}], [C \rightarrow \cdot cC, \{c, d, \$\}], [C \rightarrow \cdot d, \{c, d, \$\}]$   
 $I_{89} : [C \rightarrow cC, \{c, d, \$\}]$

$10-6 = 4+3 = 7$

So, next we will see how an example of how to do this thing. So, first of all suppose this is the grammar that we have and for this grammar  $S$  dash producing  $S$ . So, this is the new rule that is added and this is  $S$  producing  $CC$  and  $C$  producing small  $c$  capital  $C$  and  $d$ . So, this small  $c$  and small  $c$  and small  $d$  so these are the terminal symbols of the grammar and rest are all non-terminal.

So, the set  $I_0$  is constructed by which we are for which we are doing this thing. So, this we are we take this production  $S$  dash producing this item  $S$  dash producing dot  $S$  and dollar and then we have to take the closure of that. So, closure of that gives me  $S$  producing dot  $CC$  dollar. So, now dot  $CC$  dollar so, what I have to do is for taking the closure of this item so I have to consider the first set of.

So, you remember that if I have got a production like  $X$  producing dot  $X$  dot  $YZ$  semicolon alpha then what I have to do is I have to take the set first  $YZ$  alpha and then I have to see what is coming there. So, for then the this as this they this look ahead token I

have to do it like this. So, if I have got a production rule like Y producing gamma then for that I have to see what is the first set of this YZ alpha and accordingly I have to see I have to add them to the first to this item.

So, here also so, this dot CC and there is a rule like C producing cC. So, when I am trying to use this rule, so, I have to see what can I what can be derived. So, what is the first set of this capital C? So, this first set of capital C contains this small c and small d, so, they are added. So, I have to see what is there in the first of this C dollar and this first of C dollar capital C dollar will have small c and small d, so, this is the item that is created.

Similarly, by this rule it will be C producing, so, I am looking into the C producing d, but I have to see like what are what are the first of this C, C dollar and the first of C dollar again having C and the small c and small d so, they are added to the look ahead token. In this way this look ahead tokens are calculated. So, this is a bit cumbersome, but that is the technique ok, so, we have to do it like this.

So, you see for the small grammar having only to say three grammar rules S producing C cC producing small c, capital C and or d, so, the three grammar rules. So, there are 10 states that I created that way it is a large number of states that have been created.

Now, what we look into this for converting into LALR consider the states I 4 and I 7, so, I 4 and I 7. So, you see that they are the core part is same. This is also C producing d dot, this is also C producing d dot. So, what is differing is only the look ahead part. So, here the look ahead part is C d, here the look ahead part is dollar. So, we will be combining them together and call them a new state I 47 and this has got C producing d dot semicolons that look ahead's will have look ahead's of both the states c, d and dollar. So, c, d and dollar they have been taken into consideration.

Then the I third then this 3 and 6; so, 3 and 6, so, this is 3 and this is 6. So, here also the look aheads are this a core part is same. So, it has got core like C producing small c dot capital C, so, this is matching; then C producing dot c capital C this is matching, C producing dot d this is also matching. So, again what is differing are the look ahead tokens. So, this look ahead tokens so, they will be merged together and we will be getting this I this state I 36 and this 8 and 9. So, here also this 8 and 9 so, they are matching. So, their core part is matching, so, we just merge them together.

So, essentially from this 1, 2, 3, 4, 5, 6 states so, we could reduce them to 3 states. So, total originally there were 10 states and now from the 6 states. So, 10 minus 6, 4 and 3 new states created, so, total number of states turns out to be 7. So, if we construct the SLR parsing table then you can check that this also gives rise to seven states. So, that way this is going to be this LALR parsing. So, this is going to help us of course, it is cumbersome in the sense that we are. So, we are the way that we have done it we have first produced the LR 1 items so and then we are trying to see whether there is any merging possibility between the items.

So, that way the constructing the LR 1 items itself is costly and then if you so, then checking for the merging and all. So, that is why this LALR parsing construction parts of construction is going to be a costly affair though automation is no problem. So, automation can be done, so, rules are well defined, so, we can do the automation very easily.

(Refer Slide Time: 21:01)

### LALR Construction – Step-by-Step Approach

- Sets of states constructed as in LR(1) method
- At each point where a new set is spawned, it may be merged with an existing set
- When a new state S is created, all other states are checked to see if one with the same core exists
- If not, S is kept; otherwise it is merged with the existing set T with the same core to form state ST

So, what can be the other way of constructing the LALR parser? So, this is known as step by step approach for this LALR parsing table construction this item construction.

So, sets of item sets of states constructed as in LR 1 method only; however, so, we do not construct the entire LR 1 set of items and then try to do the reduction, then try to merge the items instead of that at each point whenever a new set is spawned. So, we may see that whether it can be merged with an existing set or not.

So, in the previous example you see that when we had so, when we had generated this say for example, this state number this particular state 7. So, whenever we are generating this 7. So, whenever this is generated we see that it is matching with 4. So, we do not generate a new item or new set 7 rather we add this dollar to this set, so, that way it is going to be done. So, with the it will not generate large number of new state, so, number of states generated will remain same as the S LR parsing.

So, that is how this parsing table for LALR this set of items for LALR parser will be created that at each point when a new set will be spawned. So, you will see whether it can be merged with an existing set or not. Whenever a new state S is created all other states are checked to see if one with the same core exists. And if so, if it is not if there is no such thing exists then S will be kept otherwise it is merged with the existing set T and we name the core as name with the same core to form the state ST.

So, previously you have seen that we have named the states as 47; 47 the I 4 and I 7 were merged, so, we told the item to be I 47. So, similarly this 3 and 6 was merged, so, we called it I 36 or I 36 like that. So, that way this merging is done and after doing the merging so, we can get the after doing the merging we get the new set of states and the new set of states will be number of states will be same as the SLR number of states. So, this way we can step by step we can construct the LALR parser.

(Refer Slide Time: 23:26)

The slide has a yellow header with the title 'Using Ambiguous Grammars'. Below the title is a table of grammar rules:

$E \rightarrow E+E$	$E \rightarrow E^*E$	$E \rightarrow (E)$	$E \rightarrow id$	$Follow(E) = \{+, *, (), \$\}$
$I_0: E' \rightarrow E$	$I_2: E \rightarrow (E)$	$I_4: E \rightarrow E+E$	$I_6: E \rightarrow (E)$	$I_8: E \rightarrow E^*E$
$E \rightarrow E+E$	$E \rightarrow E^*E$	$E \rightarrow (E)$	$E \rightarrow id$	$I_{10}: E \rightarrow E$
$E \rightarrow E^*E$	$E \rightarrow E+E$	$E \rightarrow (E)$	$E \rightarrow id$	$I_{12}: E \rightarrow id$
$E \rightarrow (E)$	$E \rightarrow E^*E$	$E \rightarrow (E)$	$E \rightarrow id$	$I_{14}: E \rightarrow E+E$
$E \rightarrow id$	$E \rightarrow E+E$	$E \rightarrow (E)$	$E \rightarrow id$	$I_{16}: E \rightarrow E^*E$
$I_1: E' \rightarrow E$	$I_3: E \rightarrow id$	$I_5: E \rightarrow E^*E$	$I_7: E \rightarrow E+E$	$I_9: E \rightarrow (E)$
$E \rightarrow E+E$	$E \rightarrow E^*E$	$E \rightarrow (E)$	$E \rightarrow E+E$	$E \rightarrow id$
$E \rightarrow E^*E$	$E \rightarrow E+E$	$E \rightarrow (E)$	$E \rightarrow E+E$	$E \rightarrow id$
$E \rightarrow (E)$	$E \rightarrow E^*E$	$E \rightarrow (E)$	$E \rightarrow E+E$	$E \rightarrow id$
$E \rightarrow id$	$E \rightarrow id$	$E \rightarrow id$	$E \rightarrow id$	$E \rightarrow id$

Handwritten notes on the right side of the slide include:

- A diagram showing state transitions:  $E \rightarrow EAT(T)$ ,  $T \rightarrow TAP(F)$ ,  $F \rightarrow (E)id$ .
- An arrow pointing from  $E \rightarrow id$  to a circle labeled  $\emptyset$ .
- A box labeled  $M \times 3$  containing  $M \times 1$ .

So, after this so, since it is very difficult to do some exercise on this SLR and this canonical LR and LALR parsing. So, rest of our discussion will be constructing will be restricting ourselves to SLR parsers only. But, whatever discussions we do with SLR parser so, they will also be valid for this canonical LR and LALR parsers. So, the strategy will remain same only thing is that maybe for some grammar where this SLR parser is giving rise to conflicts this canonical LR and LALR will may not give you conflicts. But, there may be situation where the grammar is such that it also gives conflict there, so, in that case we have to resolve conflicts accordingly.

So, how to so, next important issue that we will be looking into is the usage of ambiguous grammars for this parser construction. So, ambiguous grammar means that there is ambiguity. So, like this is that famous ETF this expression grammar previously we had written like E producing E plus T or T then T producing T star F or F and F producing within bracket E or id, so, this was the grammar.

Now, what do we do we do not take this extra non-terminal symbols this ETF. So, why this non terminals T and F were introduced they were introduced to ensure the precedence of the operators because until and unless you have done this until and unless you have reduced to a T so, you cannot reduce to E, but for reducing to T so, this multiplications are already taken care of. So, with this you can so, whenever you are doing a reduction to E either it can be a an expression that does not contain any multiplication or it is an expression where there is a addition operation and one part of it and the two parts of it they can contain addition or multiplication, but at the top most level I have got multiplication addition.

So, that means, the addition it has got the lowest precedence compared to this multiplication. So, this way similarly this F is put within bracket E. So, F is F has been introduced to ensure that these parentheses it has got the highest precedence over any other operator. So, this way this parser so, it could generate pars trees unambiguously, but the difficulty that we face is many a times what happens is that as you are increasing your number of non-terminals in your grammar so, the parsing table becomes larger, but in particular at least the GOTO part of the table that will become larger because GOTO part will have one entry for each of the non terminal in each of the states.

So, that GOTO part is going to have problem. Like if I say that this after constructing the item so, suppose if I see that if there are say  $m$  number of items and there are three non-terminal symbol. So, GOTO part will have at least  $M$  into three number of entries. Whereas, if somehow I can make this grammar that has got only say one non terminal then that table of table size will become  $M$  into 1. So, that way the number of entries in the table becomes small, so, that table becomes simple.

So, what we do we many times this parser the parser generators they say that you we purposefully make the grammar ambiguous. So, that the parsing table size becomes small and then naturally once we do that so, it can give rise to conflicts. So, I can so, I have to reduce I have to dissolve this conflicts and we dissolve the conflicts in a particular fashion. So, depending on the grammar so from the knowledge about the language for which we are designing the parser. So, we can try to resolve the we can try to resolve the ambiguity by putting additional rules.

Like, if I substitute if I write this particular grammar without using this extra non terminals  $T$  and  $F$  I always tell it is  $E$ , then we know that this addition has is of lower precedence than multiplication. So, if this I think is told to the parser so, when it is generating the pars tree so, if it has got a confusion that on getting the next input symbol, the input operator plus or star whether to shift or whether to reduce, so it can take a decision depending on what operator it has seen and what is the current situation, so, based on that, it can take a decision.

So, we will see that these ambiguous grammars it can lead to conflicts, but at the same time this ambiguous grammars so they can be used to make the parsing table small and that helps in the overall parsing process, so, makes the parsing process faster. So, that way, this sometimes we will see that we purposefully make it make the grammar ambiguous.

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

**Lecture – 31**  
**Parser (Contd.)**

So, these ambiguous grammars can be used for reducing the complexity of the parsing table. So, we can have less number of entries in the paring table, but it can give rise to problems. So, let us take an example and try to see how it is going to happen.

(Refer Slide Time: 00:31)

The table below is a portion of the parse table shown in the slide:

STATE	ACTION						GO TO
	id	+	*	(	)	\$	
0	S <sub>3</sub>			S <sub>2</sub>			1
1		S <sub>4</sub>	S <sub>5</sub>				Acc.
2	S <sub>3</sub>		S <sub>2</sub>				6
3		R <sub>4</sub>	R <sub>4</sub>		R <sub>4</sub>	R <sub>4</sub>	7
4	S <sub>3</sub>		S <sub>2</sub>				8
5	S <sub>3</sub>		S <sub>2</sub>				
6	S <sub>4</sub>	S <sub>5</sub>					
7		R <sub>1</sub>	R <sub>2</sub>		R <sub>1</sub>	R <sub>1</sub>	
8		R <sub>1</sub>	R <sub>2</sub>		R <sub>2</sub>	R <sub>2</sub>	
9	R <sub>3</sub>	R <sub>3</sub>		R <sub>3</sub>	R <sub>3</sub>		

So, let us say that this standard ETF grammar that we have seen previously. So, that is there this T and F. So, these two non-terminals are not used. So, we have got the grammar rewritten using only the non-terminal E. So, E producing E plus E, E producing E star E, E producing within bracket E and E producing id. So these are the grammar rules.

So, naturally if I have got an expression like say id plus id star id then I have got multiple number of parse trees now. So, if I say I will be doing decomposition like this. So, E plus E and then this E will be giving me id this E will give me E star E. So, if I mark them as id 1, id 2, id 3. So, this is basically id 1. So, and this is this E star E and this giving me id 2 and this giving me id 3, fine or somebody may do it like this E it

gives E star E where this E gives me id 3 and this E gives me E plus E and this E gives me id 1 and this gives me id 2.

So, we have got two different parse trees for the same string and using; so, in one case so, you have so, if you look into it carefully in one case I am doing the multiplication first between id 2 and id 3 and then I am adding to id 1. So, if you look in a bottom up fashion. So, we are doing it like this on the other case I am doing the addition first id 1 plus id 2 and then multiplying it by id 3. So, this is the other case.

Now, syntactically both are correct, but it is giving my giving me two different parse trees. So, it is giving rise to ambiguity. So, the grammar is ambiguous grammar. So, that is that is established. So, this grammar is an ambiguous grammar. So, what happens when we try to construct the corresponding parse tree.

So, let us see what happens. So, let us corresponding parsing table this SLR parsing table. So, this I0 so, this the as per our policy we know that we have to introduce another special non terminal E dash start symbol E dash producing E and then I0 will be E dash producing dot E and closure of that. So, this will give me E producing dot E plus E, E producing dot E star E, E producing dot within bracket E and E producing id. From I0 to on E it will come to I1 this E dash producing E dot then this E producing E plus E is there. So, E producing E dot plus E and similarly from this rule I will get E producing E dot star E.

From I0 on this open parenthesis so, it will be like say open parenthesis dot E close then E producing similarly now it has got dot E, so, all these rules will come now. Similarly, I0 on id it will be I3 which is E producing id dot then from this I from this I1 on plus so, it will come to this I4. So, E producing E plus dot E and again since dot E is there. So, all of them will come. So, I4 will come. So, this way it will be creating all these items.

So, once we have created all these items then we will be trying to construct the parsing table and before that since it is SLR parsing table construction. So, we need to have this follow set computed. So, the follow set in this case is very simple. So, by from this rule you see that E may be followed by plus, from this rule you see that E may be followed by star E may be followed by close parenthesis and dollar is the since E is the start symbol of the grammar. So, dollar is there in the follow set. So, the follow set of E is like this.

Now, let us try to see what happens to the parsing table. So, the parsing table will be something like this say from this state 0 on id it comes to a state 3. So, this is the thing is that we have explained previously from I0 on id it comes to state 3 then from I0 on open parenthesis it comes to S2 I2 then from I1 on from the state I1 on this if it finds a dollar then this E dash producing E dot is there. So, that is the accept state and this on this plus so, this will be a shifting. So, this I1 plus so, this is the state 4. So, that is S4 state and then we have got this S5 on I1 star. So, I1 star is S5. So, this way this table is made.

So, similarly from I3 from this state I3 on this particular rule we see that whatever is in follow of E for them I have to do the reduction follow of E has got this plus, star, close parenthesis and dollar. So, from I3 plus, star, close parenthesis and dollar so, I have got reduced by rule number 4. So, that is fine. Now, what about this state S 7? So, that I7; so, this is the thing. So, in I7 you see that first we will look into say this rule E producing E plus E dot. So, dot is at the end of the production. So, that means, whatever is in the follow of this E, so, I have to do a reduction by this rule and follow of E has got all of them plus, star, close parenthesis, dollar. So, by plus by from for plus, star, close parenthesis and dollar so, for all of them I do a reduction by rule number 1. So, they are doing a reduction by rule number 1.

However, there is also another situation like E E producing E plus E dot plus E. So, that means, so, this says that on plus it should do a shift and shift to what? It should shift to an item where E producing E plus dot E is coming; so, E plus dot E. So, that is basically state number 4. So, state number 4 is this one. So, E plus dot E so, it is coming to state 4. So, this is the other action. So, this is one action and this is the other action. Similarly, on star; so, it is coming to state 5. So, this S 5 is one action and R 1 is another action, fine.

Now, but for this close parenthesis and dollar there is no confusion. So, there is no shifting. So, only I have got some conflicts in this part. Similarly, from state 8 if you try to see then here I have got this E producing E star E and dot. So, whatever is in follow of E there I have to add do a add reduction by this particular rule number 2. So, this by this rule number 2 I am doing the; I am doing the reduction. And, now this rule this particular item is telling me that if you see a plus then you should do a shift and go to the state 4. So, this S 4 is added and similarly for this rule is telling me this item is telling me that if you see a star you should do a shift and go to the state I 5. So, this is the shift 5.

So, this way there will be ambiguities and whenever you have got unambiguous grammar. So the corresponding SLR parsing table it will have this type of conflicts. And, in this particular case so, even if you construct one LALR parser you will find that this conflicts will persist. So, they will do not get resolved by going to the canonical LR or LALR parser.

However, we know the actions that we have to do like see if we try to resolve this conflicts then you see that. So, at this state number 7 we have a conflict between on seeing a plus whether to do a reduction or do a shifting. So, you know that whenever at state 7, so, you have seen this thing E producing E dot and then you are saying this E producing dot plus E.

So, by associativity property, so, we know that this addition may be done later, so, I can shift this addition. So, this particular addition will be doing later. So, we will we will shift this addition into the stack. So, that way I may forego this reduction I can do a shift operation and similarly if you see a star; if you see a star then you should definitely do a shift. So, with a plus there was an option because I can do the previous addition first and then proceed with this addition, but if you see a star here then definitely you should a shift; you should not do the reduction by E producing E plus E.

Basically, so this E has given me E plus E and then you have got this star and this E. So, this naturally in that case I actually meant the multiplication between this E and this E. So, I should definitely do a shifting. So, this action should definitely be a shift, ok. So, this is not a reduction action, so, this should be a shift action.

However, somebody may say that I will be doing a reduction here I will not do the shift because this addition will be similar. So, I will do a shift I will do a reduction because this is like E plus E plus E something like this. So, you have seen this E plus E and then you have so, this E plus E if I reduce to a new expression E and with that if I add E so, that is good enough; by associativity property of this addition, so, you know that that is they that will not cause any harm. So, possibly this is also an option that I do a reduction and I do not do a shift and this is arbitrary. So, somebody may say I will do shift, somebody may say I will do a reduction because there is no precedence between this plus and this plus. So, they are similar.

However, when this thing becomes a multiplication; so, when it is E plus E star E then this multiplication has to be done first and for that purpose I need to do a shift. So, this action must be a shift action whereas, this action may or so, the I will not do this reduction I will do this shift whereas, this state I can do a reduction or a shift maybe I decide in favour of reduction.

Similarly, from state 8 you see you have already seen a product like say E star E and then this rule is telling me after that if you are seeing something like this plus E, then what are you going to do? Definitely you are doing this you should do this multiplication first and then rather than the shifting this plus. So, if you shift this then that will mean that I will do this addition first and then multiply with this E. But, since you have already seen one star multiplication of two expression so, you should take that as the valid sub expression. So, I should do a reduction by rule number 2, I should not do a shift, ok.

And, similarly if you see this one say second the second one; so, it says that you have seen E star E and then again another star and then E. So, that is the situation. So, if this E is basically one E star E and then the so, this part I have already seen and then I am going to see this thing. So, naturally I should so, here also same thing like previously. So, I can do a shift or I can do a reduce. So, both are valid, but for the sake of simplicity maybe we say that we will do this multiplication first and then with the result I will take this multiplication. So, as a result here also reduce by 2 rule number 2 may be the valid choice and this is not the valid choice.

So, this way you can tell the parser once the parsing table has been constructed and the parser tells you that we have got shift reduce conflicts at this, this, this places or reduce-reduce conflicts at this places so, you can tell the parser that parser generator tool that you can resolve it in this fashion. So, often by telling the precedence between this at this terminal symbols this ambiguities gets resolved. But, or maybe you can tell it explicitly or parsers they generate parser generators they have some default rule goby if there is a shift reduce conflict it will go by shift operation only, but that is not always valid because you have seen in many cases. So, going by shift operation only in case of shift reduce conflicts so, that is not a good choice because at least for this state 8 we have seen that the shift options are not correct, it should be the reduce action. So, so, you can tell the parser like that or you can tell explicitly that do a reduction operation at this point.

So, these way ambiguous grammars can be utilized and once you have used it. So, it will give rise to these conflicts and this conflicts if you can resolve cleverly then you can get a parsing table that will have less complexity then the other the original grammar unambiguous grammar.

(Refer Slide Time: 15:27)

## Error Recovery in LR Parsing

- Undefined entries in LR parsing table means error
- Proper error messages can be flashed to the user
- Error handling routines can be made to modify the parser stack by
  - popping out some entries
  - pushing some desirable entries into the stack
- Brings parser at a descent stage from which it can proceed further
- Enables detection of multiple errors and flashing them to the user for correction

The slide has a watermark of a man in a pink shirt speaking on the right side. At the bottom, there is a blue footer bar with the 'swayam' logo and other educational icons.

Another important issue with this parsing process is the error recovery. So, in other parsing techniques also we have seen that there is a concept of error recovery because in the recovery process so, we whenever the parser is having a stack so, it is putting those states into the stack. And, it may so happen that parser has gone to a state from where it cannot recover, it cannot come back to a valid state from where it can proceed with the parsing.

So, that is a difficulty. So may be the when the parser is progressing. So, we know that this if the parsing table entry is blank then all those entries are error entries. So, if there are error entries then we should be careful and the parser will just stop telling the syntax error. So, if you do not take care of it at the parsing algorithm itself then of course, this parser will be will not be able to give you any error message and we should be we should be able to give appropriate error message the why the parser has reached an erroneous state. So, that has to be told.

So, this undefined entries in the LR parsing table means error. So, proper error messages can be flashed to the user. So, if we can analyze a particular entry then we can try to we

can try to figure out like how we have arrived at that particular entry, what how that state and input combination has arrived and if we know that then appropriately we can flash some error message.

Now, error handling routines can be made to modify the parser stack by popping out some entries from the stack and pushing some desirable entries on into the stack. So, we know that this input string is erroneous. So, we take out some invalid entries from the stack and pushing some entries which the parser designer thinks that that is correct, that will be a correct sequence. And, that way most of the time what it does is that it tries to push some symbols which are going to be the end of string marker and things like that so that one sentence ends or one block ends. So, like that.

It will attempt to bring the parser to a descent stage from which it can proceed further. So, it can identify further errors in the in the input stream. So, that way it can it I needs to come to a descent stage and, enables detection of multiple errors and flashing them to the user for correction. So, in one go you can flash a number of error messages to the user and the user can correct all of them and then come back and resubmit the job of compilation. So, that way this will be useful.

Otherwise what will happen is that the parser will stop at the first entry itself and telling that there is an error and then how to where is the error at least the line number of the source file needs to be told, and possibly what is the error like say semicolon missing or some particular token is missing. So, that type of message, so if we can give then that will be helpful for the parsing further user to rectify the program.

(Refer Slide Time: 18:47)

## Error Recovery – Example

$E' \rightarrow E$   
 $E \rightarrow E + E \mid E * E \mid id$

Follow( $E$ ) = {+, \*, \$}

I0: {[ $E' \rightarrow E$ ], [ $E \rightarrow E + E$ ], [ $E \rightarrow E * E$ ], [ $E \rightarrow id$ ]}  
I1: {[ $E' \rightarrow E$ ], [ $E \rightarrow E + E$ ], [ $E \rightarrow E * E$ ]}  
I2: {[ $E \rightarrow id$ ]}  
I3: {[ $E \rightarrow E + E$ ], [ $E \rightarrow E + E$ ], [ $E \rightarrow E * E$ ], [ $E \rightarrow id$ ]}  
I4: {[ $E \rightarrow E * E$ ], [ $E \rightarrow E + E$ ], [ $E \rightarrow E * E$ ], [ $E \rightarrow id$ ]}  
I5: {[ $E \rightarrow E + E$ ], [ $E \rightarrow E + E$ ], [ $E \rightarrow E * E$ ]}  
I6: {[ $E \rightarrow E + E$ ], [ $E \rightarrow E + E$ ], [ $E \rightarrow E * E$ ]}  
I7: {[ $E \rightarrow E * E$ ], [ $E \rightarrow E + E$ ], [ $E \rightarrow E * E$ ]}

State	ACTION				GOTO
0	id	+	*	\$	E
1	e2	S3	S4	Acc	
2	e2	R3	R3	R3	
3	S2	e1	e1	e1	5
4	S2	e1	e1	e1	6
5	e2	R1	S4	R1	
6	e2	R2	R2	R2	

e1: Seen operator or end of string  
while expecting id  
e2: Seen id while expecting operator

So, we will take an example. So, suppose we have got a grammar like this  $E$  producing this is a simple expression grammar that has got this addition and multiplication  $E$  producing  $E$  plus  $E$  or  $E$  star  $E$  or  $id$ ; so, then an  $E$  dash. So, to make the corresponding SLR parser, so, we add that this extra production  $E$  dash producing  $E$  to get the augmented grammar and then we compute the follow set of  $E$  follow set of  $E$  from this grammar itself you can see that you have got plus, you have got star and since  $E$  is the star symbol of the grammar we have got dollar. So, this is the follow set plus star and dollar.

Next thing is that we have to construct the LR 0 items. So, this I0 is  $E$  dash producing dot  $E$  and then since dot  $E$  is there so, closer will take  $E$  producing dot  $E$  plus  $E$  producing dot  $E$  star  $E$ ,  $E$  producing dot  $id$  into the set I0, then in I1 I will have from go to I0  $E$ . So, this is this will give me  $E$  dash producing dot  $E$  producing dot  $E$  plus  $E$ ,  $E$  producing dot star  $E$ . So, like that it will give me.

Now, you see that so, in this way we try to construct all the LR 0 items and in this particular case we have got 8 such items I0 to I7. Now, now suppose so, suppose this table that is constructed so, it has got this the it has got this thing. So, this I0 from this state I0, so, this id it says I0 id is I2, so, it is shift 2. Similarly so, but these entries are undefined; these plus, star, dollar so, they are all undefined. So, in state I0, if you find

that the so, so what is this set when are you going to consult this particular entry? So, your state is S 0 and the input stream it has got a plus in it input pointer is here.

So, in state S 0 that is at the very beginning of your parsing process if you see a plus, so, that is the error because your expression can start with an identifier, it cannot start with an operator. So, we can flash this particular message E 1 seen operator end of string is not there seen operator while expecting id. So, it was expecting an identifier, but it has seen an operator. Similarly, if this first symbol itself is a dollar; that means, the end of string. So, there is a null string basically; for null string it is not there. So, we can say that so, that if it sees a dollar then it can say that it is a seen end of string character while expecting an identifier. So, that is E 1.

Similarly, E 2 so, say so, for all these cases so, if it sees a plus, star or dollar so, it can flash the message E 1. Similarly in state 1, if you are in state 1; that means, you are in this situation. So, this is you have seen an expression and then you can what you can expect is that you have seen some expression say id plus id and you are somewhere here, and then the next thing that you can expect is another; so, so, this does not give anything. So, this is a complete thing. So, if you are at this point; so, next you are expecting a plus or a star that is you are expecting an operator, but you have got an id. So, this entry will be consulted when in state 1 you have got an id. So, in state 1 if you get an id so, the so, you are expecting if you are not seen the dollar then you are you are you are expecting a plus or a star.

So, if you see id that is another error E 2, so, seen id while expecting operator; so, this is by analyzing the entry so, you can find out what may be the problem and accordingly you can populate this table. Similarly, say this state 2. So, state 2 id dollar and again if you are getting an id so; that means, that is an error because you have you have seen an identifier so, you cannot see one identifier following another identifier. So, you cannot the situation cannot be there. So, in between there must be some operator. So, there also I can flash this message E 2 that is you are expecting an operator not an identifier.

State 3; so, in state 3, so, if you find a plus when state 3 you what you are expecting to see is an identifier, ok. So, the so, naturally you can flash this message that E 1 that is E so, you have if you see this plus, star or dollar so, we can say that I was expecting an

identifier and I have got an id. So, that is that is the error. So, I was expecting an identifier and I have got an operator plus, star or end of string dollar. So, that is an error.

Similarly, state 4 so, here also you are expecting an identifier and we have got this one of these operators plus, star or dollar. So, that is also the error E 1. State 5; so, this is this is the situation that you are expecting a plus or a star that is you are expecting some operator and you have got an id. So, that is the error. So, you are expecting an operator there. Then, state 6 I6; so, here also you are the expecting an operator and there you have got an id. So, this is the rule. So, the state 7 is not shown here. So, accordingly you can add that column then it will be at their particular row in this table.

So, this way we can have this error messages. So, we can appropriately flash error message then what may be the action like that is about that is about the situation that this problem has occurred. So, it was expecting an identifier and it has got an operator say. So, say this E 1; say when this E 1 is seen that it was seen an operator while expecting id then one thing that the parser can do is that purposefully in the input stream so, it can push an identifier. So, it can. So, it has a got a plus so, what it can do purposefully it can introduce, it can take the pointer back by one position take it to here and introduce write here one id and then give it to the lexical analyzer as a so, it or the parser that the token is id.

So, that way for this E 1 when this error E 1 occurs apart from flashing the message to take the parser out of the erroneous state for this particular grammar so, it can take the it can push an id token into the input stream and then the parsing process will continue normally. Similarly, for the E 2 situation so, it can it was expecting an operator so, since it was expecting an operator so, it can do like this. So, it can push in one plus into the input stream because plus has the lowest precedence. So, it will not hamper with any other operators. So, I can put a plus.

So, though the parser will generate a will not generate a parse tree because these are erroneous thing, but the parser will be able to continue and it may it may be able to find out more errors in the expression. So, that way so if I have got lines like this then if it has got if it has found an error here. So, it will be in the normal case it will stop at this point, but if you can somehow modify the input stream by introducing the appropriate token, then possibly it can proceed further and it can again detect some error at some later line.

And, on the other thing is that you can also try to modify the stack, this stack that the parser maintains for more complex situation. So, we have to do that, but for this particular example it is fine. So, it is if we can just the there are these are very this is the very simple grammar, but this explains the how can we have this error messages introduced and how this error recovery parts can be introduced.

So, at the end of this process, so, you see that none of the actions in the table that they are undefined. So, there none of them are blank. So, the parser when in the parsing algorithm when it is proceeding so; it will always find some valid entry into this table in the in the action part. So, it may be a normal parsing action in terms of shift and reduce or it may be an error action in terms of these error routines E 1 and E 2. So, that way it will be able to take care of those errors and it can recover the parser will recover from the error.

And, the user will also give a given enough feedback about how this parsing algorithm is going to how this parsing is, what are the errors that has occurred during this parsing; so that the user will be able to correct all those errors and resubmit the program for compilation.

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

**Lecture – 32**  
**Parser (Contd.)**

Next, we will look into an example of how this erroneous thing can be detected, errors can be detected and the recovery can take place.

(Refer Slide Time: 00:26)

**Example (Contd.)**

String: "id + \$"

Stack	Input	Error message and action	State	ACTION				GOTO
0	id+\$	Shift	0	(S2)	e1	e1	e1	1
0id2,	*\$	Reduce by E->id	1	e2	S3	S4	Acc	
0E1,	*\$	Shift	2	e2	R2	(R3)	R3	
0E1+3	*\$	"id expected", pushed id and 2 to stack	3	(S2)	e1	(e1)	e1	5
0E1+3id2,	*\$	Reduce by E->id	4	S2	e1	e1	(1)	6
0E1+3E5	\$	Shift	5	e2	R1	S4	R1	
0E1+3E5*4	\$	"id expected", pushed id and 2 to stack	6	e2	R2	R2	R2	
0E1+3E5*4id2,	\$	Reduce by E->id-						
0E1+3E5*4E6	\$	Reduce by E->E*E						
0E1+3E5	\$	Reduce by E->E+E						
0E1	\$	Accept						

FREE ONLINE EDUCATION  
**swayam**

Suppose we have got an input string which is say this id plus star. So, definitely you can understand what the errors at this place we have got an error and after the star we have got an error. So at both the places some identifier is expected and let us see how this parsing process it can detect this particular situation.

So, initially the stack is containing 0 and this input is containing id plus star dollar. So, 0 and id, so 0 id says the shift 2. So, it will be shifting id into the stack and the new state is 2. So, this state is also pushed into the stack. Now, this 2 plus so 2 plus says reduce by rule number 3. So, it will be reducing by that rule E, producing id and then this and this the then new state will come. So, that will become this E 1. That will be the new state and then state 1, it is plus state 1 plus is shift 3. So it will be shifting plus into the stack and this 3.

Now, 3 star. So, 3 star is E 1. So, it was expecting id ok. And so, that is the problem now that it has it was expecting id and got an operator. So, it will be pushing it will assume that the input that it has seen is id only. So, if it assumes like that, so it will flash the message that id expected and then it will push and identify id and 2, the state 2. So, if it gets an id, then the action is shift 2. So it will push one id into the stack and the state new state 2 into the stack.

Now 2 star, so 2 star it says that it reduced by rule number 3. So, rule number 3 will be applied and accordingly this reduction will take place and this will be the new configuration of the stack. Now this 5 star, so 5 star is shift by shift 4. So, it will shift star into the stack and 4 into the stack. Now, this 4 dollar, so 4 dollar this is E 1. So, it is id expected. So, it will push an id into the stack and the state is the. So, if it gets an id, then this 4 id is S 2. So, it will push it into the stack and id into the id into the stack. So, that way so the id and 2 so, they are pushed into the stack.

Now, this 2 dollar, so 2 dollar is reduced by rule number 3. So, it will reduced by E producing id. So, that way it will be coming to this configuration then 6 dollar. So, 6 dollar is this one reduced by rule number 2. So, it will do that, then this will be the configuration then 5 dollar reduced by E producing E plus E. So, this 5 dollar is reduced by rule number 1, do that then you come to this configuration 1 dollar. So, 1 dollar is accepted.

So, what has happened is that, there were errors in my input string and accordingly, it could flash these messages id expected, id expected like that, but the parsing process could need not stop. So, what it did is that in the input stream, it has purposefully pushed in the ids. So, those input symbols are ultimately going to the stack. So, here if the way it is shown, so it has not shown that it is put into the input, but the input the symbols are also put into the stack. So, those symbols are pushed into the stack. So, that way the shifting is done that.

So this, so this way we can rectify the problems and the parser can continue and it can take care of this syntax errors and it can flash multiple errors. So, in this case it could flash both the errors. So, it could flash multiple errors and continue with the parsing process and come to a decent end of the parsing algorithm.

(Refer Slide Time: 04:37)

LALR Parser Generator - yacc

- yacc – Yet Another Compiler Compiler
- Automatically generates LALR parser for a grammar from its specification
- Input is divided into three sections
  - ...definitions... Consists of token declarations C code within %{ and %}
  - ...rules... Contains grammar rules  $E : E^+ T \{ \} \}$
  - ...subroutines... Contains user subroutines

So next, we will be looking into an LALR parser generator tool called yacc. So, the name comes from Yet Another Compiler. So, compiler means it is a compiler for compilers. So, it can like compiler it is supposed to generate the machine language program from the input source program. So, here as if I give the specification of the compiler and from there, the output is also a compiler. So, output is a so, it is a compiler but as an output it produces another compiler which for the given specification.

So, this is called yacc. So, there is many other tools have been developed, but this was the first tool that came up with the Unix operating system and later on many other tools have come, but the philosophy remains same ok. So, the specification style and also that we will see slowly. Of course, what I should say is this tool is quite powerful in the sense that it has got many interesting features which are beyond the scope of this normal compiler theory this LALR or CLR parsing; it has got many context sensitive parts also. So, that way this if you look into the complete manual of this yacc, then it is much more powerful but since it is a part of our, yes part of our course, so we will be looking into the basic features and how it can be used for developing compilers.

So, it is an LALR parser generator. So, it automatically generates LALR parser for a grammar from its specification. So, the first thing that you are that you should do like given a grammar. So, you see that the most important concern after learning about this

LR parsers, we see the most important concern that we have is regarding the conflicts, whether there is any shift reduce conflict or a reduce conflict or not.

So, the first thing that often done is that given whenever you come up with a grammar G, whenever we have got a grammar G we parse it through this yacc tool to see this yacc will tell us whether there is any shift reduce any conflicts or not to see whether there is any conflicts. So, if there is any conflict. So, I can modify this grammar and see and come up with a grammar which does not have this conflict. So, that is the first thing. So, by whatever be the technique by specifying the precedence or modifying the grammar rules or whatever we do, so the most of the compiler design processes.

So, they go by that so, first the bare minimum grammar is given to the yacc tool and it analyses the grammar and tells whether there is any shift reduce conflict or reduce conflict in the grammar or not. And once you are once you are true, then we try to put some extra actions into corresponding to the grammar rules so that we can do our desired function. So, the desired function maybe code generation, desired function maybe say one expression evaluation or maybe generating some other output in some different format. Basically, a format conversion from one format to another format so, that is the thing that the compilers are doing. So, we can do like that.

So, it generates LALR parser. So, the so, you should be very happy with that because it can give us LALR parser so easily. Now, just like Lex, this tool also has got the input specification file in some particular format. So, we have got the three portions in it the definitions part, the rules parts and the subroutines part and they are separated by this double percentage symbols ok. Just like this Lex specification file, we had portion separated by this double percentage. So, here also we have the same thing.

The definition part it will contain the token declarations, C code within this symbols that is percentage open parenthesis, percentage open brace and percentage close brace. So, you can have all the tokens that you have got in your language. So, you can define them in this portion in the definition section and certain part of code, you may want to be copied verbatim onto the output. So, that is also that code you can put within these symbols. So, they will be copied verbatim into the output file.

Then, we have got the rules so, then in the rules, so we can write in terms of terminals and non-terminals. So, if you have whatever, so if you have detail if you declared some token here and that token appears in these rules. So, they will be taken as terminals and whatever is not defined as token, so they will be taken as non-terminals. So, they will be taken as non-terminal symbols.

And then, the third part of this file or the specification file. So, this will have the subroutines some additional subroutines because this rules the grammar only. It not only the grammar rule, so we will we will have the corresponding actions also like you can have a rule like E producing E plus T. So, E plus T, then we can have the corresponding rule here and while writing this rule, corresponding action here. So, while writing this action, you may need some additional c routines and those additional c routines can be written in this subroutine section ok. So, these are the three parts that this yacc specification file can have.

(Refer Slide Time: 10:34)

**Example – Calculator to Add and Subtract Numbers**

- Definition section
  - %token INTEGER
- declares an INTEGER token
- Running yacc generates y.tab.c and y.tab.h files
- y.tab.h:
 

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define INTEGER 258
extern YYSTYPE yylval;
```
- Lex includes y.tab.h and utilizes definitions for token values
- To obtain tokens, yacc calls function yylex() that has a return type of int and returns the token value
- Lex variable yylval returns attributes associated with tokens

Now, this is a typical example of a calculator to add and subtract numbers. So, we have so, this calculator, so you can enter some integer and through that and you can have this addition and subtraction operation and then you can, so that calculator has to be defined. So, this is the yacc specification file structure.

First we have to talk about the definition section. So, in the definition section, so, we have got only one token since we are assuming that only integers can be there. Our

calculator is very simple it just does integer addition and subtraction. So, we have got a token whose name is integer. So, it declares this integer to be a token. So, on this specification file, so if you write this say grammar dot y as the input specification file for the yacc and pass it through this tool yacc, then it will generate two files; one file is y dot tab dot c and another file is y dot tab dot h.

So, this y dot tab dot c, this actually contains the final LALR parser. So, this entire parsing program will be contained in this y dot tab dot c. And this y dot tab dot h, so this is basically a header file which contains these token declarations and all. So, that it can be attached with the Lex specification file, Lex output file and Lex output file will include this y dot tab dot h. So, Lex you remember, that there was a there was a output which is Lex dot yy dot c. So, this lex dot yy dot c. So, this file it includes this y dot tab dot h has to include y dot tab dot h.

So, the idea is that in this lex specification file, you might have written like this says that say for this numbers integer numbers. So, d followed by a digit followed by digit star. So, you might have written like a digit followed by digit star and then, the corresponding action may be return a particular token integer. So, this token we want to return. Now, this integer declaration is available in this file y dot tab dot h. So, that way I have to have this, so that that they that they are going to act as the interface between the Lex tool and the yacc tool.

So, this one see, so y dot tab dot h; so, it contains many things not only this token declarations, but many other things also like it has got this. So, if not defined YYSTYPE, it will define YYSTYPE as int. So, YYSTYPE is the yacc stack type. So, this is basically the parser stack type. So, this parser stack type, so by default this parser stack will contain integers only because it has to contain the state number and the tokens and the tokens are has defined to be integers. So, ultimately, this stack contains nothing, but integers. So, by default it will take parser stack type as integer. But you can define your own stack type also.

So, for example, if you think that my tokens are not only integers, but it has got some other attributes in it which is very common. For example, if I am say having some integer suppose the number value is 25. So, this 25, I want to have as a attribute of the token integer, the corresponding token return is token is integer, but it has got an

attribute whose which is the which is the value attribute and the value is equal to 25. So, that way the attribute has to be told.

So, if we do that, so you can define this YYSTYPE that stack type to be consisting of this attributes also as a part of the token or as a part of the non terminals. So, if it is not defined in that case, it will define it to be integer. If it is defined if the user has defined YYSTYPE, then it will take that YYSTYPE. Then this hash define integer 258. So, this is the token definition. So, in this in our particular example there is only one token. So, 0 to 257, so they are they are kept for this ASCII characters and the next tokens. So, they are defined the next values are used for defining the tokens. So, about 255 onwards, it will start.

And this YYSTYPE also, this variable I have told previously basically the attribute part of the token. So, that is that that they are of that is of type YYSTYPE. So, these are there in the y dot tab dot h if you file if you open. So, you will find at least these lines for this particular case. So, Lex includes y dot tab dot h and utilize this definitions for token values that I have already said and to obtain tokens yacc calls the function yylex. So, while discussing on this Lex tool we have told that there is a function yylex.

So, whenever you need the next token, so, you can give a to give a call to it. So, if you have got a main routine, from the main routine you can give a call to yylex. But whenever you are having this parser also integrated with the lexical analyzer then from the parser it will give a call to yylex to get the next token. And it is yylex return type is integer and it returns the token value. So, it will be returning the value of the token.

And lex variable YYLVAL returns attributes associated with the token. So, yylex will return the token value and this YYLVAL, it will be simultaneously set to the attributes associated with the tokens. For example, this will be yylex will return the token value that is 258 ok, but this will be returned to this will be returned by this function yylex as the return value. But this YYLVAL, so this will be having the value 25 if the lexical analysis tool has taken enough care to initialize YYLVAL to the corresponding value 25 for this particular case ok; otherwise, that will be garbage.

(Refer Slide Time: 17:37)

The slide shows a hand-drawn diagram of a lexical analyzer state transition diagram. The states are represented by circles, and transitions are labeled with characters or character ranges. A state labeled 'yytext' is shown with a transition to another state labeled 'return INTEGER'. The diagram is annotated with handwritten text: 'lex.y.c' at the top, 'YYVAL' with an arrow pointing to the variable declaration, 'yytext' with an arrow pointing to the variable name, 'return INTEGER', 'return PLUS', and 'return MINUS'. Below the diagram is a snippet of C code from a Lex input file:

```
%{  
#include <stdio.h>  
void yyerror(char *);  
#include "y.tab.h"  
%}  
%%  
[0-9]+ {  
    yyval = atoi(yytext);  
    return INTEGER;  
}  
[-+\n] return *yytext;  
[\t]; /* skip whitespace */  
/*yyerror("Invalid character");  
%%  
int yywrap(void)  
{  
    return 1;  
}
```

The slide also features a watermark for 'swayam' and other educational icons.

Next, we will see, so the overall lex input file for this particular application will be like this. So, this part will be copied verbatim, so as I said within over this percentage brace start and percentage brace close. So, hash include stdio dot h, so yyerror function and hash include y dot tab dot h. So, all the token declaration that you have so that they will come in this portion although they will come as hash defines. If you open the file Lex dot yy dot c, after compiling through the lex tool, then you will find that all the token declaration, so, they have come in this part.

Now, this is the rules part for the lex file. So, for 0 to 9 plus that is at least one or more occurrences of the digits in the range 0 to 9. So, this YYLVAL, so this is set to be equal to a to I ACSII to integer of yytext. So yytext, you remember that this is a variable that contains the next matched portion of the input string. So, input string may be like this. So, maybe at this point, the input pointer over somewhere here when this yylex function was called and then it has the scan the input and it has seen that up to this much. So, this constitutes the next token.

So, this yytext will be equal to this substring ok. So, this y, so this is for example, if I have written like 25 plus and at the input pointer was somewhere here. So, it will take this 25. So, this part as the yytext and then see this is a text value or the character value. So, or a string value, so I want to convert it into an integer; so, I call this function atoi on this string variable yytext and it returns the corresponding integer value 25. So, the I

make this YYLVAL equal to that integer value, so that it is, so that it will be taking it will be keeping that one in the you can you can get this attribute in the parser. And the token returned is integer. So, it is returning the token integer, but the YYLVAL is set to be equal to the corresponding value.

Then, the minus, plus or new line, so it will return the corresponding character string directly. So, star yy t, it will return start yytext. So, this plus, minus, so we have not defined them as separate tokens. So, you can always do that. So, you can define tokens like plus and minus, then you can say that on say getting a minus. So, you can say return minus if you have defined those tokens or plus, you can say return plus. So, like that can be done, but this yacc tool it will allow you to write strings directly. So, if, so characters directly. So, you can put it like this plus. So, we will see that.

So, this yytext is returned, so it will match with this plus in case of plus. So, we will not need to make it separate. Then for whitespaces, so it will be it will be it will be removing the whitespaces any. So, then we have got a dot here, so that means, for any other symbol any anything else any other character appearing on the input stream. So, it will call the function yyerror and yyerror. So, yyerror is a standard error routine. So, if you parse it one string, so it will just print that string.

So, it will be printing the invalid character; so, any other character that is 0 to 9 plus, minus, blank, tab. So, apart plus, minus, new line, blank tab, so apart from that, if it sees any other character, so it will simply it will simply print that the invalid character. So, that is the lex specification file. Now, what is the corresponding yacc specification file? So, that, we will see.

(Refer Slide Time: 21:42)

So, this is the yacc input file. So, we have got this, so this is. So, this is the yylex function is defined, yyerror function is defined then we are we will. So, next part is the first part is this part will be copied verbatim to the output as we know. Then, we have got this token integer there is only one token integer that is defined.

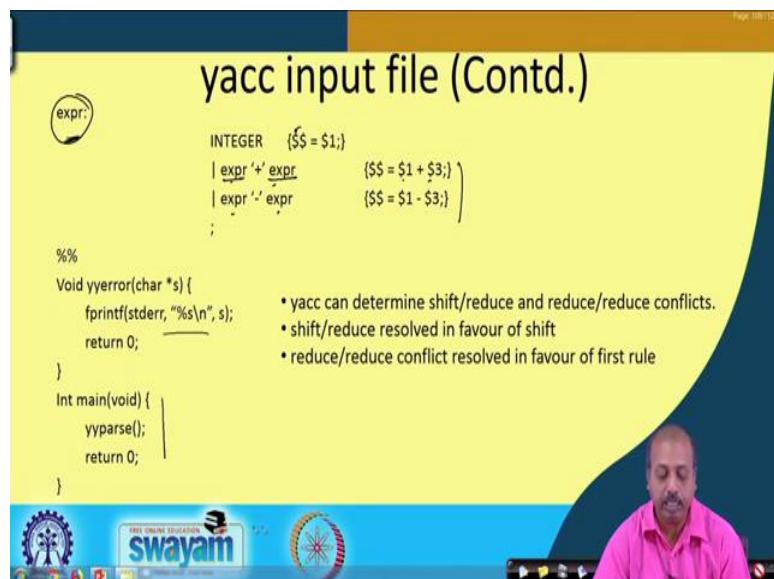
Now, program is defined as program expression newline. So, we can have multiple lines of expressions like. So, it is you can have like 2 plus 3, 5 plus 6, 6 minus 1, etcetera then it is expected that since it is the calculator. So, as you write 2 plus 3 and then, plus newline, so it should print the value 5. After that, if you can enter 5 plus 6 and plus newline. So, it will print 11. Then 6 minus 1, so it will evaluate it can put the print the value 5. So, it is the, so my grammar rule is program producing program, expression, newline. So, if I if I write in a short hand this program as p, so p producing p expression and the newline.

So, this p can again produce another p expression or epsilon. So, this is there or epsilon. So, that way I can produce this multiline input sequence because this p can be replaced by another p expression, newline, then expression newline ok. So, that way I can, so I can get these two lines. So, this is so, after that if this p is replaced by epsilon, so it turns out to be expression, newline, expression, newline. So, that way I can have two expressions, so any as the number as I have got multiple number of expressions. So, in each line there can be one expression. So, I can do that.

Now, when this newline is seen, so what we do? We print the value dollar 2. So, whenever you are trying to refer to this grammar symbols in your action part. So, this part this is the left hand side will be referred to as dollar , then this will be referred to as dollar 1, dollar 2, dollar 3. So, when I say dollar 2 actually it will be referring to this and it has got some attribute associated with it that is in YYLVAL. So, that there was that that stack type so that will have this value dollar 2. So, we will see how you are going to assign this.

So, it will print that value of dollar 2 and so this is one rule for that we have got this action or so. Nothing is written here means, this is basically that epsilon. So, epsilon need not write explicitly. So, if there is a blank; that means, that is epsilon and then this is the semicolon, that is the end of this rule. So, this particular grammar it has got only one production rule expression, program producing program expression newline and after that, for the expression we can have some rule.

(Refer Slide Time: 25:13)



```

expr: INTEGER { $$ = $1; }
      | expr '+' expr { $$ = $1 + $3; }
      | expr '-' expr { $$ = $1 - $3; }
      ;
%%

Void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
    return 0;
}

Int main(void) {
    yyparse();
    return 0;
}

```

• yacc can determine shift/reduce and reduce/reduce conflicts.  
 • shift/reduce resolved in favour of shift  
 • reduce/reduce conflict resolved in favour of first rule

Like say, expression it may be an integer. So, in that case, dollar dollar is made equal to dollar one. So, because dollar one the in that YYLVAL, the lex lexical analysis tool the Lex, it has given me the value. So, that value will be assigned to dollar dollar. So, that is this expression will get the value there.

Similarly, if it is expression plus expression, then it will be doing that that the value of this expression will be made equal to value of dollar 1 plus value of dollar 3, so dollar 1

plus dollar 3. Similarly, if it is expression minus expression, so it in that case it will do dollar dollar equal to dollar 1 minus dollar 3 this. So, the main routine, so this will give a call to yyparse and yyparse routine it will parse the entire string entire input file and it will produce all the outputs like this and if there is some error. So, it will call this yyerror function. So, it will print this it can print some messages and it can do that.

So, this yacc can determine shift, reduce and reduce, reduce conflict. So, this is one problem one thing then this shift reduce conflict is resolved in favour of shift and reduce reduce conflicts are resolved in favour of the first rule. So, that I have already said, but this may not be acceptable. So, in that case, you need to modify the grammar you can you need to tell the precedence and all so that this parser will be generated properly. So, this is a very simple example. So, there are many many other interesting feature. So, I would suggest that you look into the manual for this Lex and yacc to get a good hold of how to use these tools for writing good compilers.

(Refer Slide Time: 27:07)

## Syntax Directed Translation

- At the end of parsing, we know if a program is grammatically correct
- Many other things can be done towards code generation by defining a set of semantic actions for various grammar rules
- This is known as Syntax Directed Translation
- A set of attributes associated with grammar symbols
- Actions may be written corresponding to production rules to manipulate these attributes
- Parse tree with attributes is called an annotated parse tree

So, next so we have seen these tools, so next we will be looking into something called Syntax Directed Translation. So, at the end of the parsing process, we know if a program is grammatically correct or not and many other things can also be done towards the code generation by defining a set of semantic actions for various grammar rules. So, this is called syntax directed translation. So, while doing the translation, so it is guided by the syntax of the language.

We can have some set of attributes like this grammar symbols that we have taken. So, we can have some attributes associated with the grammar symbols and we can write actions in terms of those attributes. So, this parse tree with these attributes said. So, that is known as annotated parse tree. So, how are you going to use this?

(Refer Slide Time: 27:56)

## Example – generate postfix expression

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$$

Attribute val of E and T holds the string corresponding to the postfix expression

$$E \rightarrow E_1 + T \quad \{E.val = E_1.val \mid\mid T.val \mid\mid '+'\}$$

$$E \rightarrow E_1 - T \quad \{E.val = E_1.val \mid\mid T.val \mid\mid '-'\}$$

$$E \rightarrow T \quad \{E.val = T.val\}$$

$$T \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9 \quad \{T.val = \text{number}\}$$

$\mid\mid$  means string concatenation

Input string: "3 + 2 - 4"

```

graph TD
    E[E] --- E1[E]
    E --- T1[T]
    E1 --- T2[T]
    E1 --- T3[T]
    T1 --- D1[3]
    T2 --- D2[2]
    T3 --- D3[4]
    E --- E2[E]
    E --- T4[T]
    E2 --- T5[T]
    E2 --- T6[T]
    T4 --- D4[3]
    T5 --- D5[2]
    T6 --- D6[4]
    
```

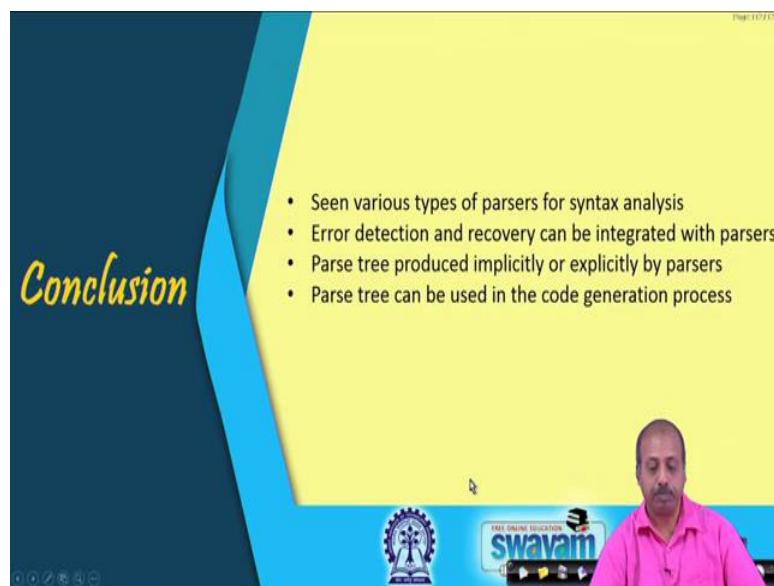
Is say this one, suppose we have got a grammar like this. So, E producing E plus T or E minus T or T and T producing all these digits 0, 1 to 9; so, we have got attribute val for E and T that will hold the string corresponding to the postfix expression ok. Now, so, we want to convert the a string to a postfix expression. So, postfix means the operator comes after the operands. So, for example, if I have got an expression like say 5 plus 6 into 7, then in the postfix expression, so this will be 5 sorry this will be 5 6 7 star plus. So, while doing the evaluation. So, once you see the an operator, so you take out the previous two operands, do the operation and then replace this part by the result that is 42 and then again you see an operator. So, you take the last two operands and do the operation and make the value 47.

The advantage of this postfix expression is you do not need to have parenthesis ok. But there are many applications where this postfix expression is used, but how to convert this an expression infix expression to postfix. So, this is a grammar based solution for that. So, let us see how are you writing these rules.

For example, let us look into this rule E producing E plus T. So, for the sake of understanding we are writing this second E as E 1. So, E e dot val is basically E 1 dot val. So, E 1 has got the corresponding value. This operator is called the concatenation string concatenation operation. So, with this val, so this T dot val will be concatenated and this plus will be concatenated. Similarly, it is E 1 minus T. So, with the E dot val E 1 dot val T dot val and minus will be concatenated. So, that way it will work and this T producing 0 to 9. So, solve them that T dot val will be equal to the number.

So, for example, if this is the string, so this is the parse tree produced. Now, when this reduction is did made, the val is made equal to 3. Similarly, when this reduction is made by E producing T, then this you look into this rule it say E dot val is T dot val. So, this is E dot val is made equal to 3 similarly here. So, E producing E plus T, say this rule. So, this is equal to E 1 dot val that is 3, then concatenated with E 2 dot val T dot val that is 2 and concatenated with plus. So, I get this string and similarly when I have got when I do this particular reduction, so this 3 star 3 2 plus then 4 and minus. So, they are concatenated. So, I get the whole postfix expression. So, this way I can have the syntax directed translation schemes for doing many activities.

(Refer Slide Time: 31:01)



So, to conclude we have seen various types of parsers for syntax analysis. We have seen error detection recovery integrated into the parsers. Parse tree can be produced implicitly or explicitly by the parsers and parse tree can be used for the code generation process as

we have seen in the syntax directed translation scheme. So, this is the code generation is actually done by the syntax directed translation policies and while going to the intermediate code generation phase, so we will be looking them in detail so.

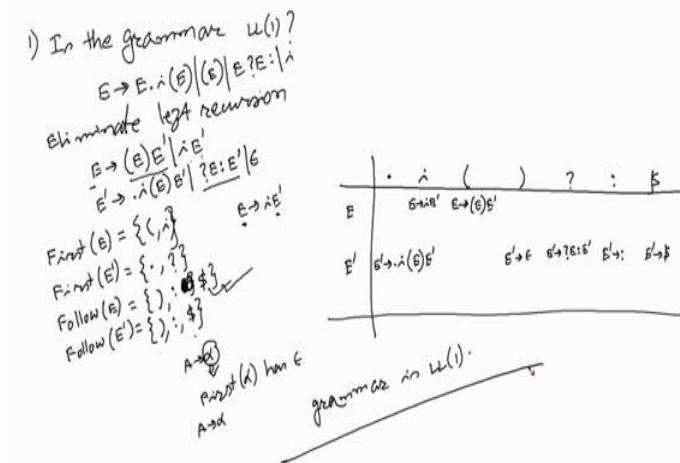
Thank you for this part of the lecture.

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

**Lecture - 33**  
**Parser (Contd.)**

So we have seen our discussions on Parser design. So, in the next few classes so, what we will do is that we will take a few examples and try to solve them, because this particular chapter it requires lot of practicing as I told you. And, you must have also noticed that calculation of this individual parameters, individual sets is often a bit cumbersome. So, I would like that you also do the practices so, that you can be you are quite confident about it. So to start with, we will solve if a few problems. So, first we solve one problem which is from predictive parsing.

(Refer Slide Time: 00:53)



So in this problem we will try to see whether a given grammar is whether a given grammar we can have this whether the is LL 1 or not. So the question is the grammar is LL 1. So, the way to answer this question definitely is to try to construct a LL 1 parsing table and see whether there is any duplicate entry in any of the cells of the table. So, the grammar given is like this E producing E dot i within bracket E where, this dot i then open parenthesis close parenthesis they are all a terminal symbols of the grammar. And, E is the non-terminal symbol with in bracket E or E question mark E colon or i. So, here

this dot this i question mark colon. So, these are all terminal symbols of the grammar. Now, you see that this grammar to check for LL 1 the first obstacle that we have here is that the grammar is left recursive.

So, E producing so you have you have got this E at the right hand side again. So, first job to do is to eliminate left recursion, the first step for solving this problem is to eliminate left recursion. So, how do you do this? So, there is a set of rules that I have discussed previously. So, if you apply those rules, then it will turn out to be a grammar like this E producing within bracket E and then E dash or i E dash. So, the rules for which immediate left recursion does not appear so, for they are to be taken and this E dash is a new known terminal that has been introduced. And, now E dash can be made to produce dot i within bracket E E dash or question mark E colon then E dash or epsilon.

So, this is the grammar that you get after eliminating the left recursion. So, the next step is to compute the first and follow sets because, we tried to get the predictive parsing table. So, the next step is to get the first and follow set. So, we have got to non-terminals. So, I should have two first sets first of E and first of E dash. So, first of E so, starting with E you see can understand that it will have the symbol open parenthesis and i ok.

So this is the first set, similarly first of E dash, it has got dot and then this question mark. So, this is there in the first of E dash. Now, the follow set follow of E and follow of E dash. So, this is to be computed. Now, follow of E so, if you look into the grammar. So, we have got this close parenthesis because, in this rule it is ending after you get a close parenthesis.

And, also if you look into this rule you see the colon can come after E and this so, whatever. So then whatever is in first of E dash is follow of E; so, by the because whatever is in. So, if you looking to say this rule so, E producing i E dash. So, whatever region follow of E will be in follow of E dash and then this one follow of E dash you can see that follow of E dash will be equal to follow of E. So, that way so, by this rule say E producing i E dash. So, E producing i E dash.

So, whatever is in follow of E is also in follow of E dash. So, that way follow of E already I have got this so, close parenthesis and colon. So, these two will be there and it has been the start symbol of the grammar. So, dollar will be there in the set and also if

so, this from this rule E producing E within bracket E and then E dash. So, this dollar will also come in the follow set of E.

So, then this if you try to construct the table the parsing table then it will be like this. So, then we have got the symbols dot i then open parenthesis, close parenthesis, then question mark colon and dollar ok. Now, this so, this is by the rules that we have for constructing the predictive parsing table. So, it says that if there is a rule like a producing alpha then and therefore, so you have to look into the first of alpha and for all those first of all those symbols I have to add A producing alpha into the set.

So, if you look into the first rule E producing within brackets E then etcetera then the first. So, this is the alpha part this part is the alpha part. So, the first of it contains the open parenthesis. So, this open parenthesis so, I will add the rule E producing within bracket E then E dash. So, this will be added similarly, when it is applied on the second rule E producing i E dash. So, this will be added here because, I have got i as the star symbol of the i as the i in the first of this i E dash. So, this will be there then for the third rule it says that E dot producing dot i etcetera. So, in the dot I should have this rule added to it.

So, E dash producing dot i within bracket E then E dash this will be there and then this now you see there is a rule like it has producing epsilon. So, there was another rule that if first of alpha if first of alpha contains epsilon. So, if there is a rule such that A producing alpha and first of alpha has got epsilon in it; then I have to add this A producing I have to add this A producing alpha to all the places in the follow of A. So, this E dash producing epsilon so, the in that case alpha becomes equal to epsilon. So, first of alpha definitely contains epsilon. So, this E dash producing epsilon has to be added to all the places where, we have got this where these follow symbols of E dash are coming.

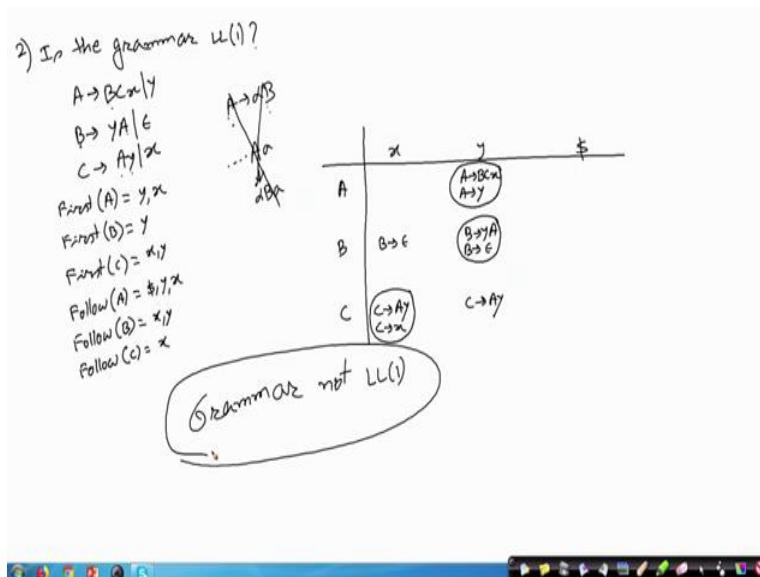
So follow of E dash contains open close parenthesis, colon and dollar. So, this close parenthesis will have so, I have to have E dash producing epsilon added here, then E dash producing so, for this colon. So, there also I have to have this E dash producing colon appearing there and I have to have these E dash producing epsilon added to this one; E dash producing dollar. Now, once now if you look into this rule E dash producing dot i so, this we have already discussed.

Now, this rule E dash producing question mark then E colon E dash. So, there the first of this are this is the alpha and first of alpha contains question this question mark symbol.

So, this will be added at this point that E dash producing question mark E colon E dash ok. So, this way we can construct this thing this predictive parsing table. So, in fact, in this follow set so, these dollars should not be there because, dollar since the E dash is the star symbol; dollar is there in the side because, dollar is the start symbol of the grammar. So, for star symbol of the grammar dollar is there and as a result from this rule E producing i E dash whatever is in follow of E is in follow of E dash so, dollar should also be here so, this was alright.

So, ultimately when we have constructed the parsing table, you see that we have got this in the in this table there is no entry which is multiply define. There is no cell in this particular table where, you have got more than one rules applied rules to be noted there. So, these blank entries are all error. So, that is fine, but there is no duplicate entry. So, this grammar is definitely LL 1 grammar; so, the grammar is LL 1 ok. So, this way you can solve this particular case that this grammar is LL 1. Now, let us look into another grammar and try to see whether that is also LL 1 or not; the grammar is like this that example number 2 question is same.

(Refer Slide Time: 11:41)



That is the grammar is the grammar LL 1 and the grammar given is like this A producing BC x or y B producing y A or epsilon, C producing A y or x ok. So, this grammar we

want to see whether it is LL 1 or not. So, first thing to check whether there is any left recursion. So, there is no left recursion because here I have got A producing BC. So, if I substitute this rule also here it will not lead to any left recursion, similarly here also if I substitute in the place of A B BC. So, it will not start with C. So, this is not a left recursive grammar so, that part is fine. So, we can come to the second step of this problem solving where we try to see what is the first set and follow set.

So, you to have to look into you have to compute this sets first of A first of B first of C, then follow of A follow of B and follow of C. So, these are the sets that we have to construct. Now, first of A is anything that you can start with A. So, it is you have got this y and then you can do it like this that for A can give me y definitely. And, then this if you replace by this BC and then B replaced by epsilon and then C replaced by x.

So, as a result x can also come in the first of A, similarly first of B you have got y ok. So, otherwise a other so, nothing else can come only y can come here and the first of C so, since in starts with A. So, whatever is in first of A is in first of C. So, I have got this symbols x and y both of them. And x from the second rule also this x is automatically coming to the first of C. Now, for the follow set.

Now the follow set since A is the start symbol of the grammar. So, whenever it is not mentioned what is the which one is the start symbol, the first non-terminal appearing on the left hand side or the first rule so, that will be taken as the start symbol. So, dollar is definitely one start symbol dollar, dollar is definitely there in the start symbol A's follow set and then we have to see all the what can follow. Like here if you apply if you look into this rule it says that whatever A is followed by y so, y you will definitely come in the follow of A. Then what is the follow of B?

So, follow of B is whatever is in first of C E can be in follow of B. So, first of C contains x and y. So, follow of B will have x and y in it and follow of C; so, C may be followed by x. So, this is there and from this rule B producing y A whatever is in follow of A will be in follow of B. So, that way so, whatever this B; so, whatever is in follow of whatever is follow of B will be in follow of A.

Like whatever I have got a rule like A producing alpha B, then whatever is in follow of A will be in follow of B. Because, if there is a string where, I have got A followed by

some terminal symbol say a that the next step I can replace it by alpha B a. So, whatever is in follow of A is in can be in follow of B. So, follow of A has got dollar and a.

So, follow of A has got here you see that follow of B whatever is in follow of B can be in follow of A if we go by this rule. So, follow of B has got x and y. So, that will come to follow of A also. So, x y will also be added here. So, this is so, this is the situation. So, I have computed the first and follow sets. Now, after completing the first and follow sets I have to try to construct the LL 1 parsing table.

So, for the LL 1 parsing table construction it go it do it like this. So, I have got the non-terminals A B and C and then I have got the terminal symbols x y and dollar. Now, we come to the first rule A producing BC x. So, I have to see what is the first of BC x. So, first of BC x is equal to first of B first of B is y. So, I will have this rule A producing BC x added here.

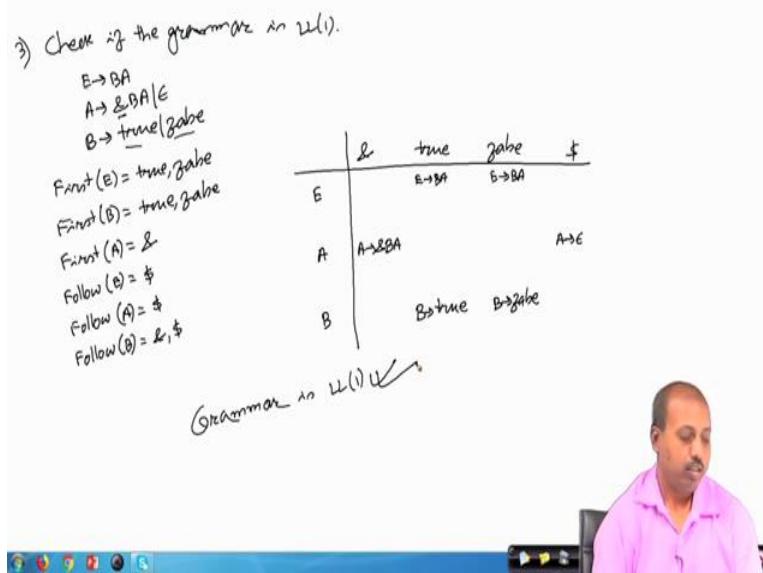
And then so, that will be added here then coming to the second rule A producing y. So, it says that in the first of y this rule should be added so, first of y contains y. So, this rule will be so, this rule will also be added here; A producing y fine then whatever from the second rule we see that B producing y A so, this first of y A contains y. So, B y I will have this rule B producing y A. So, this will be added here now, I have a rule B producing epsilon. So, a B producing epsilon is there so, whatever is in follow of B there this rule has to be added.

So, follow of B contains x and y. So, follow of B will have B producing epsilon. So, this rule will be added here as well as this rule should be added here B producing epsilon, then I will have this third case C producing A y. So, whatever is in first of A y so, first of A y is equal to first of A that is y and x. So, there I will be adding this rule. So, C producing A y and C producing A y so, these two are added and there is a rule C producing x.

So, it says that this rule should be added here also C producing x ok. So, you see that after constructing this table after constructing this table so, we see that several entries are multiply defined. So, this is a multi this there are two rules here, two rules here and two rules here. So, if you make a predictive parser so, parser cannot take a decision like which rule to follow for doing the for parsing the string which rule should be followed. So, naturally so, this is this grammar is not LL 1. So, this grammar is not LL 1. So, this is

the decision that we have for this, that way so, you can always try to do I can try to check whether a grammar is LL 1 or not by constructing the corresponding predictive parsing table. Next we take another example where we try to see whether a given grammar is LL 1 or not.

(Refer Slide Time: 20:05)



And this third example; so, this is the grammar is like this it has got a set of rule. So, again the question is same check if the grammar is LL 1, if the grammar is LL 1. So, the rule is given the production rules are like this E producing BA then A producing ampersand BA or epsilon then B producing true or false ok. So, here this is something like a Boolean grammar and the here all these are non all these are terminal symbols, this ampersand and then true false. So, these are all terminal symbols E B and A they are the non-terminal symbols. Again if you try to check whether this grammar is LL 1 first of all ah; so, the grammar does not have any left recursion.

So, I do not need to do a left recursion removal, but next step is to find the first and follow sets and try to construct the corresponding table. So, if I try to construct so, the first and follow sets. So, first of E then first of B, then first of A then follow of E follow of A and follow of B. So, these are to be computed fine. Now first of E, what is first of E? So, first of E is same as first of B and first of B contains true false.

So, this first of B has got true and false. So, first of E will also have true and false. And what is the first of A? First of A is ampersand so, first of A contains ampersand; so, this

is the thing about this first set. Now, what about the follow set? Follow of E since, is the start symbol of the grammar; so, dollar is definitely there in the follow set.

Now, I have got this E producing BA; that means, whatever is in first of A will be in follow of B. So, first of A has got ampersand so, that will come to follow of B. So, follow of B contains ampersand and then here in this rule E producing BA so, this A can be replaced by epsilon. So, whatever is in follow of E can be in follow of B. So, follow of E contains dollar so, this dollar will come in the follow of B and what about follow of A.

So, follow of A by the same rule whatever is in follow of E will be in follow of A. So, follow of E has got dollar so, this dollar is there in the follow of A ok. So, this way you can construct the first and follow sets for the grammar. So, for the symbols non-terminal symbols; so, you can check that there is no more thing one or more symbol that we can add to the first and follow sets.

So, next we try to construct the corresponding parsing table to see whether they are duplicate entries ok. So, it is like this so, we have got the symbols non-terminals E A and B and the terminal symbols ampersand, true, false and dollar. Now, this first rule tells me that whatever is in first of B there I have to add E producing BA. So, first of B contains true and false. So, there I should add E producing BA E producing BA so, they are added.

Now, coming to the second rule it says A producing ampersand BA. So, first of ampersand BA contains ampersand so, there I should add this rule that A producing ampersand BA. So, this is added. Now, the second part A producing epsilon. So, since it is epsilon I have to check the follow of A and whatever is in follow of A there I have to add A producing epsilon so, follow of A contains dollar.

So, this A producing epsilon should be added there. So, A producing epsilon is added. Now, coming to the third rule B B producing true or false. So, I will be adding B producing true at this point and B producing false at this point So, all the rules have been taken into consideration so, my table construction is over. So, whatever entries are not defined they are all error entries. So, and you see that there is no duplicate entry. So, this is also this is a grammar is LL 1. So, we can conclude that the grammar is LL 1 grammar

ok. So, next we will be looking into another example of a bigger grammar for which we try to construct the LL 1 parsing table.

(Refer Slide Time: 26:11)

1) Grammar for Boolean expression

$$\begin{aligned}
 R &\rightarrow R+T \mid T & \text{First}(R) = \{a, b\} \\
 T &\rightarrow TF \mid F & \text{First}(TF) = \{+\} \\
 F &\rightarrow F*P \mid P & \text{First}(F*) = \{\cdot, a, b\} \\
 P &\rightarrow (R) \mid a \mid b & \text{First}(R) = \{a, b\} \\
 &\Downarrow & \text{First}(+) = \{+\} \\
 R &\rightarrow TR' & \text{First}(T) = \{a, b\} \\
 R &\rightarrow +TR' \mid \epsilon & \text{First}(+) = \{\cdot\} \\
 T &\rightarrow FT' & \text{First}(F) = \{a, b\} \\
 T' &\rightarrow FT' \mid \epsilon & \text{First}(FT') = \{\cdot\} \\
 F &\rightarrow PF' & \text{First}(P) = \{a, b\} \\
 F' &\rightarrow *PF' \mid \epsilon & \text{First}(\cdot) = \{\cdot\} \\
 P &\rightarrow (R) \mid a \mid b & \text{First}(R) = \{a, b\}
 \end{aligned}$$


So, this is basically the grammar for Boolean expression, this is the grammar for Boolean expression and we have to construct the corresponding LL 1 parsing table. So, the grammar given is like this R producing R plus T or T, now T producing TF or F then, F producing F star P or P, P producing within bracket R or a or b. This is some sort of Boolean expression so, this is a modified version of that.

So, it is not truly a Boolean expression, but some sort of Boolean expression. So, if we eliminate this left so, first step for getting the um corresponding LL 1 parser is to remove the left recursion. So, if you remove the left recursion so, this grammar can be converted into something like this that R producing T R dash then R dash producing TR dash or epsilon.

Then so, there should be a plus here plus T R dash or epsilon then this T producing F T dash and then T dash producing FT dash or epsilon. Then this F giving PF dash and F dash giving star PF dash or epsilon and then this P giving within bracket R or a or b. So, this is the after eliminating left recursion so, this is the grammar. Now, next part is to compute the first and follow sets. So, the first set R can be computed first of R, first of T first of we go in a order; first of R first of R R dash, then first of T, then first of F dash, then first of F, then first of F dash and first of P ok.

So, first of R so, from the so, first of P is this open parenthesis a b. So, this is open parenthesis a b. So, this is the first of P, then first of P dash F dash is this star from this rule. So, from this one F producing PF dash so, whatever is in first of P is in first of F. So, this one contains open parenthesis a b.

Now, coming to this one whatever is in first of P is in first of T dash. So, first of T dash also contains open parenthesis a b and then from this rule so, so whatever is in first of a so, that is done. So, this is so, whatever is in first of F will be in the first of T. So, again open parenthesis a b. So, they will be in the first of T, then from these rule R dash producing this. So, first of R dash contains plus first of this plus T R dash contains plus. So, this is plus only and from R it says whatever is in first of T will be in first of R. So, first of T contains open parenthesis a b. So, they are included here.

So, we will continue this in the next class.

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

**Lecture – 34**  
**Parser (Contd.)**

So, next we have to construct the follow sets for the non terminals.

(Refer Slide Time: 00:19)

4) Grammar for Boolean expression																																																																	
$R \rightarrow R+T \mid T$	$FL(R) = \{+, \), \$\}$																																																																
$T \rightarrow TF \mid F$	$FL(T) = \{+, \), \$\}$																																																																
$F \rightarrow F*P \mid P$	$FL(F) = \{*, \), \$\}$																																																																
$P \rightarrow (R) \mid a \mid b$	$FL(P) = \{a, b, +, \), \$\}$																																																																
$\Downarrow$	$FL(\epsilon) = \{+, \), \$\}$																																																																
$R \rightarrow TR' \cdot$	$FL(R) = \{a, b, +, \), \$\}$																																																																
$R' \rightarrow +TR' \mid \epsilon$	$FL(R') = \{+, \), \$\}$																																																																
$T \rightarrow FT'$	$FL(T) = \{+, \), \$\}$																																																																
$T' \rightarrow FT' \mid \epsilon$	$FL(T') = \{+, \), \$\}$																																																																
$F \rightarrow PF'$	$FL(F) = \{*, \), \$\}$																																																																
$F' \rightarrow *PF' \mid \epsilon$	$FL(F') = \{*, \), \$\}$																																																																
$P \rightarrow (R) \mid a \mid b$	$FL(P) = \{a, b, +, \), \$\}$																																																																
<i>(Grammar in L(1))</i>																																																																	
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th></th> <th style="text-align: center;">+</th> <th style="text-align: center;">*</th> <th style="text-align: center;">(</th> <th style="text-align: center;">)</th> <th style="text-align: center;">a</th> <th style="text-align: center;">b</th> <th style="text-align: center;">\$</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">R</td> <td style="text-align: center;"><math>R \rightarrow R+T</math></td> <td></td> <td></td> <td></td> <td style="text-align: center;"><math>R \rightarrow TR'</math></td> <td></td> <td style="text-align: center;"><math>R \rightarrow TR' \cdot</math></td> </tr> <tr> <td style="text-align: center;">R'</td> <td style="text-align: center;"><math>R' \rightarrow +TR'</math></td> <td></td> <td></td> <td></td> <td style="text-align: center;"><math>R' \rightarrow \epsilon</math></td> <td></td> <td></td> </tr> <tr> <td style="text-align: center;">T</td> <td></td> <td style="text-align: center;"><math>T \rightarrow FT'</math></td> <td></td> <td></td> <td style="text-align: center;"><math>T \rightarrow FT'</math></td> <td style="text-align: center;"><math>T \rightarrow FT' \cdot</math></td> <td></td> </tr> <tr> <td style="text-align: center;">T'</td> <td style="text-align: center;"><math>T' \rightarrow \epsilon</math></td> <td style="text-align: center;"><math>T' \rightarrow FT'</math></td> <td style="text-align: center;"><math>T' \rightarrow \epsilon</math></td> <td></td> <td style="text-align: center;"><math>T' \rightarrow FT'</math></td> <td style="text-align: center;"><math>T' \rightarrow FT' \cdot</math></td> <td style="text-align: center;"><math>T \rightarrow \epsilon</math></td> </tr> <tr> <td style="text-align: center;">F</td> <td></td> <td></td> <td style="text-align: center;"><math>F \rightarrow PF'</math></td> <td></td> <td style="text-align: center;"><math>F \rightarrow \epsilon</math></td> <td style="text-align: center;"><math>F \rightarrow PF' \cdot</math></td> <td></td> </tr> <tr> <td style="text-align: center;">F'</td> <td style="text-align: center;"><math>F' \rightarrow *</math></td> <td style="text-align: center;"><math>F' \rightarrow *PF'</math></td> <td style="text-align: center;"><math>F' \rightarrow \epsilon</math></td> <td></td> </tr> <tr> <td style="text-align: center;">P</td> <td></td> <td></td> <td style="text-align: center;"><math>P \rightarrow (R)</math></td> <td></td> <td style="text-align: center;"><math>P \rightarrow a</math></td> <td style="text-align: center;"><math>P \rightarrow b</math></td> <td></td> </tr> </tbody> </table>		+	*	(	)	a	b	\$	R	$R \rightarrow R+T$				$R \rightarrow TR'$		$R \rightarrow TR' \cdot$	R'	$R' \rightarrow +TR'$				$R' \rightarrow \epsilon$			T		$T \rightarrow FT'$			$T \rightarrow FT'$	$T \rightarrow FT' \cdot$		T'	$T' \rightarrow \epsilon$	$T' \rightarrow FT'$	$T' \rightarrow \epsilon$		$T' \rightarrow FT'$	$T' \rightarrow FT' \cdot$	$T \rightarrow \epsilon$	F			$F \rightarrow PF'$		$F \rightarrow \epsilon$	$F \rightarrow PF' \cdot$		F'	$F' \rightarrow *$	$F' \rightarrow *PF'$	$F' \rightarrow \epsilon$	$F' \rightarrow \epsilon$	$F' \rightarrow \epsilon$	$F' \rightarrow \epsilon$		P			$P \rightarrow (R)$		$P \rightarrow a$	$P \rightarrow b$	
	+	*	(	)	a	b	\$																																																										
R	$R \rightarrow R+T$				$R \rightarrow TR'$		$R \rightarrow TR' \cdot$																																																										
R'	$R' \rightarrow +TR'$				$R' \rightarrow \epsilon$																																																												
T		$T \rightarrow FT'$			$T \rightarrow FT'$	$T \rightarrow FT' \cdot$																																																											
T'	$T' \rightarrow \epsilon$	$T' \rightarrow FT'$	$T' \rightarrow \epsilon$		$T' \rightarrow FT'$	$T' \rightarrow FT' \cdot$	$T \rightarrow \epsilon$																																																										
F			$F \rightarrow PF'$		$F \rightarrow \epsilon$	$F \rightarrow PF' \cdot$																																																											
F'	$F' \rightarrow *$	$F' \rightarrow *PF'$	$F' \rightarrow \epsilon$	$F' \rightarrow \epsilon$	$F' \rightarrow \epsilon$	$F' \rightarrow \epsilon$																																																											
P			$P \rightarrow (R)$		$P \rightarrow a$	$P \rightarrow b$																																																											



So the follow set will be like this. So for I am writing in short follow of R equal to, so these sets are to be computed, follow of R dash. Then follow of T, follow of T dash, then follow of F follow of F dash and follow of P. So these are the sets that I have to compute.

Now, follow of R, so if you look into the grammar you see that R may be followed by this close parenthesis and the dollar being the start symbol. So, close parenthesis and dollar, they will come to the follow of R. Now follow of R dash, so if you look into this rule, it says whatever is in follow of R is in follow of R dash.

So, this close parenthesis and dollar they will be coming here. Now, follow of P, so it says whatever is in first of R dash, so if you look into this rule it says whatever is in follow of R dash will be in the follow of T, so follow of R dash has got plus. So, this follow of T will have plus here. And then, first of R dash, so R dash can give me epsilon, so if this R dash is replaced by epsilon in that case, so whatever is in follow of R dash

will also be in follow of T. So, that way this follow of R dash has got closed parenthesis and dollar. So, they will also come in the follow set of T.

Now follow of T dash. So follow of T dash whatever is in follow of T is in follow of T dash by this rule, T producing F T dash. So, this plus, close parenthesis and dollar they come to the follow of T dash. Now, comes follow of F. So, this says that whatever is in first of P is in whatever is in first of F dash will be in follow of P. So, first of F dash has got star. So, this follow of P, follow of P will have star there. So, before that this follow of F, so how to get this follow of F? So follow of F, you can say it is given by this first of T dash. So, first of T dash has got this open parenthesis a, b. So, this first of follows whatever this first of T dash has got open parenthesis a, b. So, they can come to the follow of F.

So, this is basically first of T dash. So open parenthesis a, b, they can come to the follow of this set. Then, then follow of F dash, so follow of F dash tells whatever is in follow of F will be in follow of F dash. So, what is there in follow of [FL]? So, follow of F has got this open parenthesis a, b. So, they will be in the follow of F dash. And this T dash this T dash can give me epsilon. So, whatever is in follow of T is also in follow of F. So, whatever is in follow of T follow of T has got this plus, close parenthesis and dollar. So, they can also come in the follow of F. So this plus, close parenthesis and dollar, they can come in the follow of F.

Now, this one, so, so whatever is follow of F is equal to follow of F, that is also there in follow of F dash. So, this plus, close parenthesis and dollar so, they will also come here. And then this P, so P is appearing here. So, whatever is in first of F dash is in follow of P. So, first of F dash has got star, so that have been that I have written. Now, it says that this F dash can give me epsilon. So, whatever is in follow of F dash will be in the follow of P. So, follow of F dash has got all the symbols. So, they will all of them will come in the follow of P. So, all of them will come in the follow of P.

Now, we have got this set of rules now I can we can try to make the corresponding table. So, I have got this R, R dash, T, T dash, F, F dash and P and I have got the symbols here that is plus, star, open parenthesis, close parenthesis, then a, b and dollar. Now, we have to go rule by rule. So, first rule it says R producing T R dash. So, I have to look into the first of T. So, first of T contains open parenthesis a, b, there I have to add this rule. So,

open parentheses. So, R producing T R dash, here also R producing T R dash, here also R producing T R dash. So, they have been added. Now, come to the second rule R dash producing plus T R dash. So, that rule will be added here R dash producing plus T R dash.

Now, it says that R dash producing epsilon is there. So whatever is in follow of R dash there, I have to add this R dash producing epsilon. So, follow of R dash has got this close parenthesis and dollar there, I have to add R dash producing epsilon and here also I have to put R dash producing epsilon. So, these two are added. Now, come to the third rule T, T producing F T dash. So, whatever is in first of F T dash, that is in first of F. So, I have to add this rule. So, this T producing F T dash has to be added to first of F is open parenthesis. So, T producing F T dash, then a, b. So, T producing F T dash, they are all added there; T producing F T dash has been added.

Now, come to the fourth rule; T producing T dash producing F T dash. So, it says that whatever is in first of F T dash, there I have to add these rules. So first of F T dash has got first of F. So, there I have to add this rule. So, I have to add the rule T dash producing F T dash here, then here T dash producing F T dash, here also T dash producing F T dash. Now, there is a rule T dash producing epsilon. So, I have to look into the follow set of T dash. So, follow set of T dash has got plus, close parenthesis and dollar. There I have to add this rule T dash producing epsilon, T dash producing epsilon and here also, T dash producing epsilon. So, this is done.

Now, we come to this rule P F producing P F dash. So, whatever now I have to see the first of P. So, first of P has got open parenthesis a, b, there I have to add this rule. So, F producing P F dash, F producing P F dash, then this has got P has got open parenthesis a, b. So, F producing P F dash and here also, F producing P F dash. So, they are added now coming to this rule. So, first contains a star. So, F dash star I should have this rule, F dash producing star P F dash.

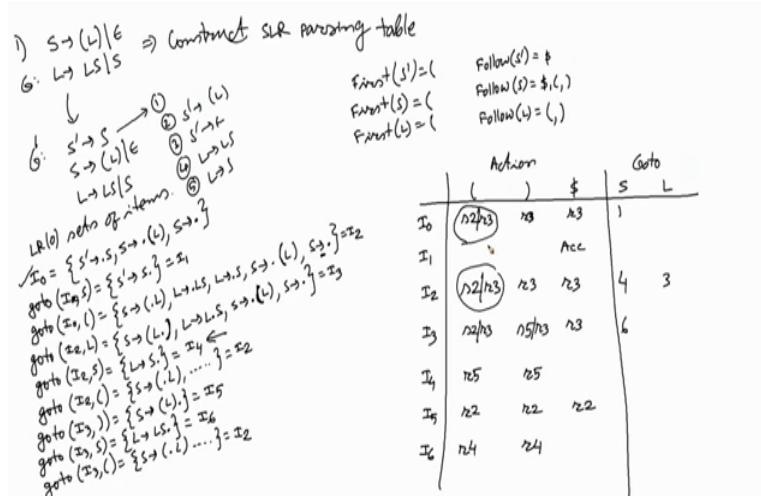
Now, this F dash producing epsilon, so, I have to look into the follow set of F dash. So, follow set of F dash has got all these entries. So, there I have to add this rule. So, open parenthesis. So, F dash produces epsilon, then a F dash produces epsilon; b there I add this rule, F dash producing epsilon, then plus F dash producing epsilon. Then, close parenthesis and dollar. So, dollar also I have to add F dash producing epsilon. So, this

has been added. Now, come to this rule P within bracket R. So, this open parenthesis, I have to add this rule. So, there I have to add the rule P producing within bracket R.

Then coming to the second rule, I have to add P producing a here and coming to the third rule I have to put P producing b at this point. So, this is the construction of the predictive parsing table level; one parsing table and here also you see that there is no duplicate definition. And since there is no duplicate definition, so we can say that the grammar is L L 1, so this grammar is L L 1 ok. So, this way, whenever it is, whenever we try to answer the question whether a grammar is L L 1 or not, so the safest way is to do this and if the directly we need to construct the grammar. So, there also we can do it like that.

Next, we will be doing a few exercises on working on this L R parser. So, we have seen the L L parser examples.

(Refer Slide Time: 11:12)



Next, we will see some L R parser examples. So, in that category the first example that we take is for the grammar a very simple grammar is producing within bracket L or epsilon and this L producing L S or S. The question is to construct the S L R parsing table. Question is to construct the S L R parsing table. Now, this is the grammar G. Now, the first step of this construction you know that we have to augment the grammar by introducing one more start symbol.

So, by augmentation, so this is the grammar G given to us we get the grammar G dash which is which has got an additional symbol is just producing S and then, we have got the rest of the rules are there S producing within bracket L or epsilon and L producing L S or S. Now, I have to construct the LR 0 sets of items, so this is the first step. Second step is to construct the LR 0 sets of items. This is the second thing to do and for that matter what we do we have to first the set is I 0. So, the set I 0 will have only one item in it which is S dash, initially you will have only one item in it S dash producing dot S

And then, by the closure rule we know that we have to construct the closure and for constructing closure. So, I have to look for productions where left hand side is S because there is an item here where S producing dot S. So, I have to look for productions which has got left side as S and then apply the closure rule. So this will give me this particular thing as another item and since S producing epsilon is there. So, that will give me S producing dot. So, this is the third grammar rule, this is third item. So, these are the three items that I have constructed.

Now, what are the other LR 0 items? So, for that I have to see the goto parts. So, wherever there is a dot before any grammar symbol of any item, so I have to do a goto on that. So, we do it like this. So, we construct the set goto I 0 S; goto I 0 S will give me S dash producing S dot. No other item has got dot before this before s. So, this is the thing and after in this in this items. So, if even if you take closure nothing is going to happen because dot is at the end. And this is a new item that we have got. So, let us name the item as I 1 ok. So, that is the item I 1.

Next, I have to construct other possible goto items. So, to goto I 0 open parenthesis. So, I 0 open parenthesis, so this will give me S producing open parenthesis dot L close parenthesis. And since there is a dot before L, so, by closure I will get all these items L producing from this rule, I will get L producing dot L S and I will get a L producing dot S.

And then, and then since I have got this L producing dot L S um, so this particular rule, so that can that will give me again the same items. But this S producing dot S, so this can give me two more items; one is S producing by this rule ok, S producing dot within bracket L and another item is S producing dot. So, this is another new item that I have got. Let us name the new item as I 2.

Now, from I 0, I cannot have anymore goto s. From I 1 also, there is there is no dot before any grammar symbol. So, I 1 also does not give any goto. From I 2, from I 2, I have to see whether I can get something new. So, goto I 2, L goto I 2 L gives me S producing within bracket L dot. So, this is one item and then another item that I get is L producing L dot S. So, this is another item. So, there is no other item where there is a dot before L. So, nothing else comes, only thing is that now I have got this dot S. So, I have to look for grammar rules and from there, I will get the new items S producing within bracket dot within bracket L, then S producing dot. So, these two items will come.

So, this is again a new item we have not yet generated. So, this is thus the item I 3 ok. Now, goto I 2, I 2 L, I have done. So, I 2 s, so goto I 2 S will give me L producing S dot. There is no other item where I have got a dot before L. So, that is new item I 4. Now, goto I 2 then this open parenthesis goto I 2, open parenthesis. So, this will give me S producing open parenthesis dot L. And then, since there is a dot before L, so, it will giving me all these rules again, all these items again. So, all those items will be coming here. So, this essentially gives me back I 2. So, goto I 2, open parenthesis is I 2 only.

Now, go to I 2. So, I 2, I do not have anything as I believe. So, this is open parenthesis is done. Now, I have to look for this item I 3 goto I 3, then this close parenthesis. So, that gives me S producing that gives me the item S producing within bracket L and that dot come here. So, that is a new item that I have got and that is I 5.

Now, I 3 S goto I 3 s. So, that will give me this L producing L S dot, L producing L S dot and that is a new item. So, this is item I 6. Now, goto I 3 S, I have done I 3 open parenthesis. So, I 3 open parenthesis, so this will give me the item S producing open parenthesis dot L ok. And this as soon as this is there, so I will get all these things, so etcetera. So, that is same as I 2 ok.

Now, for the remaining items I 4, I 5 and I 6, so there is no grammar symbol after the dot ok. So, it they cannot give any more items. So, I have constructed the set L R 0 sets of items, I 0 to I 6, there are seven things, seven cases. The next part of this parsing table construction is to get the follow sets for the grammar symbol but for getting follow sets, we also have to get the first sets. So, again we do the first and follow computations. So, first of S dash, first of and first of L, so this is to be computed, first of S dash is same as first of S and first of S is open parenthesis. So, this is there, open parenthesis is there.

And first of L by this rule L producing S, so, you see it has got the same as the first of S. So, that is the first computation.

Now, the follow computation follow of S dash, then follow of S and follow of L ok. Now, follow of S dash since S dash is the start symbol of the grammar. So, dollar is definitely there. Then, for the follow of S you see that whatever is in follow of, whatever is in follow of S dash is in follow of S. So whatever is in follow of S dash, so that is in follow of S.

So, dollar is there and you see from rule, you see whatever is in first of S is in follow of L. So, first of S has got this open parenthesis. So, follow of L has got this open parenthesis and follow of L from this rule, you see close parenthesis is also there. So, close parenthesis is 3 in the follow of L. And by this rule, L producing S whatever is in follow of L is in follow of S. So, this open parenthesis and close parenthesis, they also come in the follow of S.

So, this finishes the follow set computation. Next, I have to find out the parsing table. Now, for making the parsing table, we have to the parsing table. We will have two part; the action part and the goto part. So, this is the action part and this is the goto part. So, in the action part, I will have the grammar similar terminals, the we have got two terminal symbols open parenthesis close parenthesis and the dollar that is always there. Now, as the rows, I will have this items I 0, I 1, I 2 we can also write only the states 0, 1, 2, 3. So, also you can write this I 3, I 4 like that, so I 4, I 5 and I 6, so these are the items.

Now, in this I 0, so you see this rule L S producing dot within bracket L. So that means, I if I get this open parenthesis, I should do a shift operation and that is shift by and that will be a shift and the I have to consult the goto part. So, goto dot you know I 0 open parenthesis is I 2, I 0 open parenthesis is I 2. So this is shift 2. So, this is the rule that we get from this item I 0. Now, I 0 does I 0 also has got one rule which is S producing dot. So, I have to see the follow of S and then I have to add it, I have to do the reduction by S producing epsilon for them ok.

So, follow of S has got this dollar, open parenthesis and close parenthesis there I have to tell the reduction rules. So, this is reduction by rule number 3. So, this is our rule number 1, this is rule number 1. Then rule number 2 is S dash producing within bracket L. Rule number 3, S dash producing epsilon. Rule number 4 is L producing L S and rule number

5 is L producing S. So, always we will follow this convention. So, this is reduction by rule number 3, this is also reduction by rule number 3 and here also, there is another action which is reduction by rule number 3. And, so, you see already we have got a shift reduce conflict here that is I do not know getting an open parenthesis whether I should shift it or I should reduce by rule number 3. So, that is the problem.

And the goto part it will have the grammar non terminals S and L. So, one I 0 s, so I 0 S is I 1. So, I will write one here, there is nothing like I 0 L. So, nothing comes to the other one. So, that finishes the processing of I 0. So, I 0 is over. Now, this one I 2, so I 1, I mean the item I 1. So, in item I 1, so I have got this S dash producing dollar. So that means, so, this S dash producing dollar. So, S dash producing S dot and then I have to look into the follow of S dash and there I have to say accept. So, this is the accept.

Now, goto I, now the set I 2, so, the set I 2 has got here. There is a dot before open parenthesis. So, I should be doing a shift operation I 2 open parenthesis. So, this should be a shift. And I 2 I 2, this open parenthesis is I 2 only. So, this is shift 2 and again I have got this rule S producing dot. So, this reduce by 3 will also come, reduce by 3 and here I will get this reduce by 3 and here also there will be reduce by 3. Now, I 2 has got transition on L and S both; I 2 L is 3. So, I 2 L is 3 and I 2 S is 4. So, we have to add them here.

Now, comes item I 3. In item I 3, so, there is a dot before this close parenthesis. So, I have to do a shift on I 2, sorry I 3 close parenthesis. So, this will do a shift and I 2 I there close parenthesis is I 5. So, this is shift 5 ok. So, this will be shift 5. And then, this I 3 and then I have got this one I 3 has I 3 has got this rule also this item the dot before open parenthesis. So, I 3 open parenthesis, so this will be shift this is also be a shift and this I 3 open parenthesis is I 2. So, this is shift 2.

And then, I have got is S producing dot. So, all the reductions will come as we had earlier. So, this will be reduction by 3, this will be reduce by 3, this will be reduce by 3. And this, I 3 S, I 3 S is 6. So, this is 6 and I 3 L does not have any transition. So, it is not possible. Now, come to I 4, so this 1 I 4 L producing S. So, I have to look into follow set of L. So, follow set of L has got open parenthesis and close parenthesis. So, there I have to apply this rule, this L producing S, that is rule number 5. So, this will be reduced by rule number 5 and this will also be reduced by rule number 5.

Now come to I 5, it has got this particular rule that is rule number 2 and I have to look into the follow set of S. So, follow set of S has got dollar, open parenthesis, close parenthesis. So, there I have to add like reduce by rule number 2, reduce by rule number 2, reduce by rule number 2. Now, come to I 6. So, I 6 has got L S dot. That is rule number 4 and there I have got this follow set of L is open parenthesis and close parenthesis.

So I have to reduce by rule number 4 and reduce by rule number 4. So, this way you can construct the S L R parsing table for this particular grammar. So, it has got shift reduce conflicts in it. So, naturally we have got the grammar is not S L R ok, but we can try to resolve this conflict somehow by adding some domain knowledge on this grammar ok. So, we will continue with the examples in the next class.

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

**Lecture – 35**  
**Parser (Contd.)**

So, next we will be looking into another example of this SLR parsing table construction.

(Refer Slide Time: 00:17)

2)  $S \rightarrow B | S a b S$   
 $B \rightarrow b B | \epsilon$

1.  $S \rightarrow S$   
2.  $S \rightarrow B$   
3.  $S \rightarrow S a b S$   
4.  $B \rightarrow b B$   
5.  $B \rightarrow \epsilon$

LR(0) items of interest:  
 $I_0 = \{ S \rightarrow .S, S \rightarrow B, S \rightarrow S.a.b.S \} = I_1^-$   
 $goto(I_0, S) = \{ S \rightarrow S, S \rightarrow B \} = I_2^-$   
 $goto(I_0, B) = \{ S \rightarrow S.a.b.S \} = I_3^-$   
 $goto(I_0, \epsilon) = \{ S \rightarrow S.a.b.S \} = I_4^-$   
 $goto(I_1, S) = \{ S \rightarrow S.a.b.S \} = I_5^-$   
 $goto(I_1, B) = \{ S \rightarrow S.a.b.S \} = I_6^-$   
 $goto(I_1, \epsilon) = \{ S \rightarrow S.a.b.S \} = I_7^-$   
 $goto(I_2, S) = \{ S \rightarrow S.a.b.S \} = I_8^-$   
 $goto(I_2, B) = \{ S \rightarrow S.a.b.S \} = I_9^-$   
 $goto(I_2, \epsilon) = \{ S \rightarrow S.a.b.S \} = I_{10}^-$   
 $goto(I_3, S) = \{ S \rightarrow S.a.b.S \} = I_{11}^-$   
 $goto(I_3, B) = \{ S \rightarrow S.a.b.S \} = I_{12}^-$   
 $goto(I_3, \epsilon) = \{ S \rightarrow S.a.b.S \} = I_{13}^-$   
 $goto(I_4, S) = \{ S \rightarrow S.a.b.S \} = I_{14}^-$   
 $goto(I_4, B) = \{ S \rightarrow S.a.b.S \} = I_{15}^-$   
 $goto(I_4, \epsilon) = \{ S \rightarrow S.a.b.S \} = I_{16}^-$   
 $goto(I_5, S) = \{ S \rightarrow S.a.b.S \} = I_{17}^-$   
 $goto(I_5, B) = \{ S \rightarrow S.a.b.S \} = I_{18}^-$   
 $goto(I_5, \epsilon) = \{ S \rightarrow S.a.b.S \} = I_{19}^-$   
 $goto(I_6, S) = \{ S \rightarrow S.a.b.S \} = I_{20}^-$   
 $goto(I_6, B) = \{ S \rightarrow S.a.b.S \} = I_{21}^-$   
 $goto(I_6, \epsilon) = \{ S \rightarrow S.a.b.S \} = I_{22}^-$

$f_{NXT}(S) = a, b$        $f_{NXT}(S) = \epsilon$   
 $f_{NXT}(S) = a, b$        $f_{NXT}(S) = a, \epsilon$   
 $f_{NXT}(B) = b$        $f_{NXT}(B) = \epsilon$

	Action	S	Goto
$I_0$	<del>a</del> <del>b</del> <del><math>\epsilon</math></del>	$I_3$ $I_5$	$I_1$ $I_2$
$I_1$	$I_4$	accept	
$I_2$	$I_2$	$I_2$	
$I_3$	$I_5$	$I_3$	$I_5$
$I_4$	$I_6$		
$I_5$	$I_4$	$I_4$	
$I_6$	$I_4$	$I_4$	
$I_7$	$I_{15}$ $I_{19}$	$I_5$	$I_2$ $I_3$

So, the grammar given is like this; so, S producing B or SabS, and B producing small b capital B or epsilon. And, as you know the first step is to frame the augmented grammar. So, I have got this augmented grammar constructed S dash producing S, then S producing B, S producing SabS, B producing bB and B producing epsilon. So, this is rule number 1, 2, 3, 4 and 5 of my grammar.

So after you have constructed this rules now that the augmented grammar G dash now the next step is to frame the set of items sets of items. So, this LR 0 items; LR 0 sets of items so if you try to construct then we can do it like this I 0 is given by the set that S dash producing dot S and then I have to look for productions with left hand side S. So, I will get this S producing dot B then S producing dot SabS, ok, then B producing dot bB and B producing dot ok. So this will be the; so the so this will be the set of items that we have now.

Now so this is the so this S dash producing dot S and from there we have got the remaining items. So, as this S producing dot S is again coming so, that will give me again this S producing dot B and this rule. So, that way it will continue. Now, from this set so, if I try to construct this go to I 0 and S. So, go to I 0, S will have S dash producing S dot, and then this S producing S dot abS. So, this is a new set I 1 that we get ok. Now, go to I 0 B. So, that will give you S producing B dot sorry S producing B dot only this item. So, this is the set I 2.

Now, go to I 0 on S we have done. So, small b I 0 small b; so, this will give me B producing b dot B, and now since dot B is coming so, I will get all these rules like B producing dot small b capital B and B producing dot. So, they will come and that will be my set I 3. Now, go to I 0, b we have done now comes go to. So, there is nothing more capital B we have done, small b we have done, that is fine; go to I 1 from I 1 we can go on small a to S to a item Sa dot bS. So, that is I 4.

Now, go to I 1 I cannot go any more in the there is no other dot, so, that is I 2 also nothing now I 3; I 3, b. So, that will give me B producing bB dot, ok. So, that is only one set so, one that we have that is a new item I 5. Now, I 3 capital B we have done I 3 small b. So, go to I 3 small b. So, that will give us B producing b dot B and then as soon as I get this item. So, this rest of the items will follow. So this will give me I 3 only.

Now, I 3 is done now I 4 go to I 4 on b. So, that will give us S producing a sorry S producing Sab dot S and since this dot S is coming so, all these rules will come again that S producing dot B then S producing dot B, then I will get this one S producing dot SabS and since I have got this one S producing dot SabS so, this see dot B S producing dot B is there so, this B producing dot small b capital B will come and B B producing this dot so, that will also come. So, that will be our I 6 that will be the item I 6.

Now, go to so, I 4, b is done; I 5 there is nothing I then the I 6. So, I 6, S. So, this will have S producing SabS dot, then I will have this S producing S dot abS, ok. So, these two will be I 6, S. So, that is our I 7 and then go to I 6 from I 6 on capital B. So, there that will give us S producing B dot. So, it will give us S producing B dot that is equal to I 2. So, this gives us equal to I 2 and then S we have done capital B we have done small b. So, go to I 6, small b. So, that is equal to B producing b dot B. So, that is I 3 go to I 6, b

is also done. Now, I have this  $I_7$  go to  $I_7 a$ . So, that will give us  $S$  producing  $Sa$  dot  $bS$ . So,  $Sa$  dot  $bS$  that is equal to  $I_4$ . So, this is equal to  $I_4$ .

So, we so, all the items have been constructed  $I_0$  to  $I_7$ . Now, I have to go for the first and follow computations and then we can make the table. So, the first of  $S$  dash, then first of  $S$ , first of  $B$ . So, first of  $b$  contains small  $b$  first of  $S$ , ok. So, first of  $S$  can be so, if you start with  $S$  so, you can go on replacing this  $S$  we can replace this  $S$  by  $B$  and then this  $B$  by epsilon. So, as a result you can say that  $a$  can be there in the first of  $S$  and from this rule  $S$  producing  $B$  so, you can see whatever is in first of  $B$  is in first of  $S$ . So, this is the first of  $S$  is  $a, b$  and first of  $S$  dash is whatever is in first of  $S$  is in first of  $S$  dash. So, this  $a$  and  $b$  are there in the first of  $S$  dash.

Now the follow sets. So, follow of  $S$  dash then follow of  $S$  and follow of  $B$ , ok. So, follow of  $B$  we have to see like what can we do, but follow of  $S$  has got this small  $a$  in it so, this has got small  $a$  from this rule you can say that  $S$  may be followed by small  $a$ , so, it is there and follow of  $S$  dash will definitely have dollar. And, because  $S$  dash is the start symbol and by this rule so, whatever is in follow of  $S$  dash will be in follow of  $S$ , so, the dollar will also come here. And, follow of  $b$  is by this rule  $S$  producing  $B$  whatever is in follow of  $S$  is in follow of  $B$  by this rule. So, follow of  $S$  has got  $a$  and dollar; so,  $a$  and dollar will be there in the follow of  $B$  also.

Now, you can try that no more item can no more this terminal can be added to this follow set and then I have to construct the table and while constructing the table. So, there will be action part and go to part. So, this is the action part and this is the go to part. In the action part I have got terminals like  $a, b$  and dollar small  $a$  small  $b$  and dollar and in the go to part I will have the non terminals  $S$  and  $B$ . Now, the items  $I_0, I_1, I_2, I_3, I_4, I_5, I_6$ , so, and  $I_7$  so, there are eight items like that. Now, I have to look into  $I_0$ ;  $I_0$  is so, there is a rule like  $B$  producing dot  $B$  dot  $B$  so, this says that I have to go by shift and  $I_0$  small  $b$  is  $I_3$ .

So this is should be shift 3. And, this  $B$  producing dot is there so, whatever is in follow of dot so, they are whatever is in follow of  $B$  I have to add this is add this particular rule. So, follow of  $B$  is  $a$  and dollar. So, there I should add this  $B$  producing epsilon  $B$  producing sorry. So, this is written as reduce by rule number  $B$  producing dollar is rule number 5. So, reduce by 5 and here it is written by reduce by 5.

And, go to I 0 S is I 1. So, this is 1, go to I 0 b is I 2. So, this is 2. Now, come to so, I 0 is done now come to I 1. So, I 1 has got this rule S dash producing S dot. So, there I have to add the accept for the follow set, so, follow set of S dash is dollar. So, there I have to say accept. So, this is accept and this rule says that I have to see I 1, a. I 1 a is the I 4. So, this should be shift and 4. So, this is shift and 4. Now, come to I 2. So, come to I 2. So, this is the set S producing B dot. So, I have to look into the follow of S and this is rule number 2. So, follow of S is a dollar. So, there I have to reduce by rule number 2, fine.

Then comes this I 3. In I 3 I have got this one B producing dot b so, I have to do a shift for b. So, I 3 B should be a shift and this I 3 go to I 3 is small b is I 3. So, this is shift 3 and this one B producing dot is there so, follow of B is a and dollar there I have to do reduce by rule number 5 and it is reduced by rule number 5. So, they are done. Now, I 3 I 3, B is I 5. So, this is 5 and I 3 there is no other I 3 on the non terminal symbols. So, the I 3 that is done.

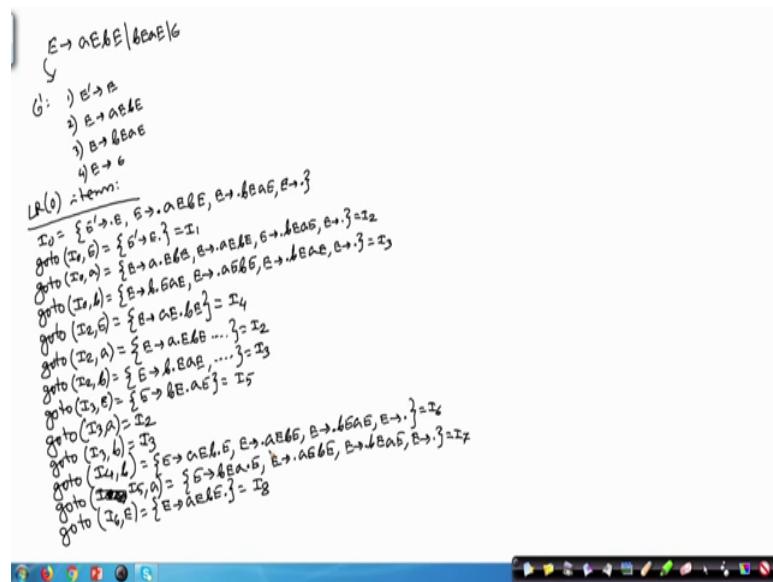
Now, come to I 4. In I 4, so, this should be a shift operation and I 4, b is a shift operation and the new state is I 4, b is I 6. So, this is shift 6. Now, come to this one I 5, so, b B dot. So, b B dot is rule number 4. So, I have to look into the follow set of capital B so, a and dollar. So there I should do a reduce by rule number 4 and here I should do a reduce by rule number 4, fine. Now, come to this I now, come to I 6, ok. So, in I 6 I have got this rule say before non-terminal so, this terminal so, this one bB. So, this B is. So, this so, this should be a shift I 6, B is a shift and I 6 small b is 3. So, this is shift 3 and by this B producing dot so, B producing dot I have to go for this follow of b which is a and dollar that should be reduced by rule number 5. So, this is reduced by rule number 5. This is also reduce by rule number 5.

Now, come to I 7. So, I 7 says that sorry before that I 6 there will be some go to part. So, I 6, B is I 6, S is 7. So, this is I 6, S is 7 and I 6, B is 2. So, this is 2 this I 6, capital B is 2. So, that is I 2. Now, I 7; in I 7 I have this one S producing S ab S dash. So, I have to that will be a reduction rule for follow of S. So, follow of S is a and dollar and this is rule number 3. So, this is rule number 3 and this is also rule number 3 reduce by rule number 3.

And, then I 7, a says that I should do a shift and I 7, a is again telling me a shift. So, and the new state, so, this I 7, a should be a shift operation and S a is I 4. So, this is shift 4 or

reduce by rule number 3. So, at this point so, that that ends the formation of this particular table and you see at the end at this point there is shift reduce conflict. So, that is coming here. So, that is so, that that give rise to shift reduce conflict. So, this particular grammar is not a not an SLR grammar. So, this is maybe it is LR 1 or so, but it is not and it is not SLR. So, this way you can construct the SLR parsing table for this particular grammar. So, next we will be looking into another example.

(Refer Slide Time: 18:58)



Next, we will be looking into another example which is like this. So, we have a grammar given like this. So  $E$  producing  $aEbE$  or  $bEaE$  or epsilon; so, this is the grammar given and we have to construct the corresponding SLR parsing table. So, for doing that first we make the augmented grammar  $G^*$  dash and forming the augmented grammar  $G^*$  dash. So, we have to add one extra production  $E$  dash producing  $E$  and from there I have to the rest of the rules will be there  $E$  producing  $aEbE$  then  $E$  producing  $bEaE$  and  $E$  producing epsilon, fine.

Now, next we have to make the LR 0 items we have to make the LR 0 items. Now, how do we do this? like first of all the  $I_0$  set  $I_0$  has got this  $E$  dash producing dot  $E$  and then  $E$  producing dot  $aEbE$ , then  $E$  producing dot  $bEaE$  and  $E$  producing dot; this is the item  $I_0$ . Now, go to  $I_0$ ,  $E$  go to  $I_0$ ,  $E$ , so, that will give us  $E$  dash giving  $E$  dot that will give us  $E$  dot producing  $E$  dot and that is the set  $I_1$ .

Now, go to I 0, a; go to I 0, a is E producing a dot EbE, and then since this dot E is coming so, again all these rules will come E producing dot aEbE, E producing dot bEaE and E producing dot, ok. So, that is the set I 2. This set we call I 2. Now, go to from I 0, b has to be considered I 0, b. So, that will give us E producing b dot EaE, then since dot is coming before E. So, all those rules will come again E producing dot aEbE, E producing dot bEaE and E producing dot. So, that will be the set I 3.

Now, goto; so, I 0 we have finished now go to I 2, E go to I 2, E. So, that will give us E producing aE dot bE, it will give us aE dot bE. So, that is the set that is the item I 4 ok, I 2 E is done. Now, I 2, a go to I 2, a is E producing this rule from this rule I will get a dot EbE and as soon as this is there. So, all the items will be coming. So, they will come and this will give me the set I 2 only, this will give me the set I 2 only. Now, go to I 2, b go to I 2, b so, that will give us this E producing b dot EaE and as soon as this dot is there then again I will have this same set of rules coming. So, that will again give rise give me the set I 3 that will give me I 3.

Now, I 2 I have done a and b both are done. So, I 2 E is also done. So, I 2 is over now come to I 3. go to I 3, E; go to I 3, E will give me this I 3 go to I 3 I 3 E will give me bE dot aE. So, E producing bE dot aE that is that will be a new set E producing bE dot aE that is a new set I 5 now go to I 3, E we have done I 3 I 3 we have got I 3, a. So, go to I 3, a I 3, a is a dot EbE. So, that is equal to I 2 and go to I 3, a is I 2. Similarly go to I 3, b will be go to I 3, b will be be sorry I 3, b will give me b dot etcetera. So, that is equal to I 3. So, this will be equal to I 3.

Now, go to, I 4 b; go to I 4, b will give us E producing aEb dot E and as soon as it gives that. So, this will give us the remaining rules like dot E so, is coming. So, this E producing dot aEbE E producing dot bEaE and E producing dot. So, that is a new set I 6 and go to I 4 go to I 4, b we have done I 4 does not have anything else, then I 5 sorry go to I 5, a go to I 5, a that will give us E producing bEa dot E. And, then since this dot E is coming then all these rules will again come aEbE E producing dot bEaE and this E producing dot. So, that will be a new item I 7.

Now, go to I 6. So, I 5 is done. Now, I 6 go to I 6, E is E producing aEbE dot aEbE dot. So, that will be a new set E producing this I 6, E. So, this will give us aEbE dot. So, that

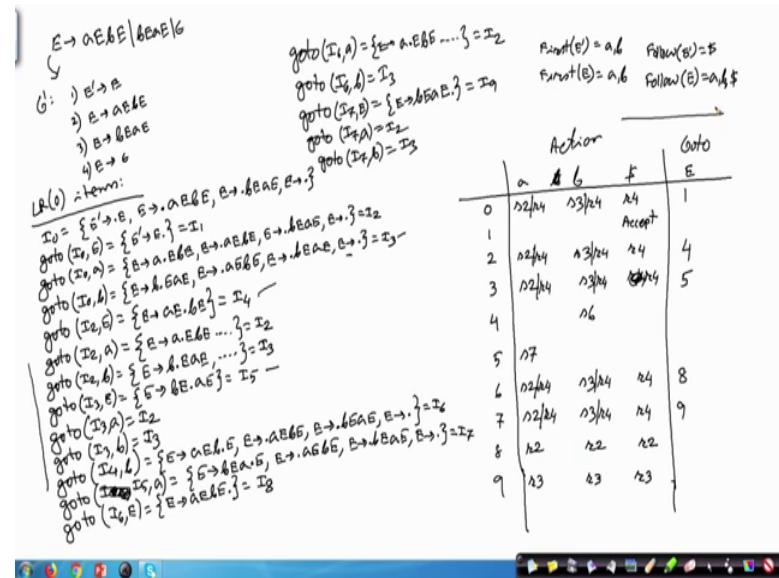
is a new set I 8 and then I can construct this I 7 from this from this I 7 from this I 6 I can make I 6, a, ok. So, those sets are to be constructed.

So, we will continue in the next class.

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

**Lecture – 36**  
**Parser (Contd.)**

(Refer Slide Time: 00:14)



So, next, we have to make the remaining goto s. So this goto I 6, a we have done so, I 6, a goto I 6, a, so that will be giving me the item E producing a dot E b E. Now, a dot b E a dot EbE so, that is same as this item I 2, so, etcetera. So, that is same as I 2. So, I 6, a is I 2 then I 6, b I 6, b. I 6, b will give us this b dot etcetera b dot E etcetera and that is nothing, but I 3. So that is the set I 3. So, this will give me I 3 only. Now, goto I 6 something more is there dot a dot b is done. So, E is also done. So, I 6 is over.

I 7, E; I 7, E will give me this b E b E. So, I 7 E goto I 7, E will give us E producing bE aE dot and that is a new set. So, this is I 9. Now, I 7, E I have done I 7 will give me a dot something. So, goto I 7, a is a dot something that is I 2 and goto I 7, b will be b dot something. So, b dot something is I 3. So, this is I 3. Now, I 7 then what about I 8? So, I 8 there is no go because dot is at the end I 9 also dot is at the end. So no new set can come from there. So, that completes the construction of this LR 0 items.

After that we have to look for the first and follow of this non-terminals. So, first of E dash first of E. So, let us look into these two sets. So, first of E is definitely a and b, ok.

So, this is first of E is a and b and first of E dash is equal to first of E, so, that is also equal to a and b. And, now follow of E dash and follow of E, ok. So, follow of E you can see here a and b is followed by a. So, this is a and b and since E dash is the start symbol of the grammar. So, this has got dollar in it and by the first rule. So, whatever is in follow of E dash is in follow of E. So, dollar is also there in this set. So, this is a b dollar and that is dollar.

After this we try to construct the parsing table for this particular grammar ok. So, SLR parsing table; this will have this action part and the goto part. So, I have got the symbols a, b and dollar. These are the three symbols you know terminal symbols and non-terminal I have got only E and I have got the states ranging from 0 to 9, ok. So, this is 0, 1, 2, 3, 4, 5, 6, 7, 8, 9; I have got nine such cases. Now, I have to look into individual items and see what can be done with that.

Now, looking at I 0; so, I 0, a is a shift operation and I 0, a is 2. So, this is shift 2. I 0, b; I 0, b is 3. So, this is shift 3 and then in I 0 I have got E producing dot so, I have to look into the follow of E and there I have to add the reduce rule E producing; so, E producing epsilon. So, this is reduce by rule number 4. So, this is this has got another entry now reduce by rule number 4, this has got another entry now reduce by rule number 4. And, this 0, E is 1, so, I have to make it 1.

Now, I 0 is over now I 1. So, E dash producing E dot. So, then this is E dash follow has got dollar. So, this should be accept this should be accept. Now comes this I 2 and in I 2 I have got this one a dot a and then I 2, a is I 2 only. So, so this is shift 2 then I 2, b; I 2, b is I 3. So, this is shift 3 and then I have got this rule E dot E dot. So, the follow set I have to see. The follow set of E is a, b and dollar there I will have to do reduction by rule number 4. So, all these are to be added reduce by rule number 4, reduce by rule number 4 reduce by rule number 4. So, they are to be added and I 2, E; I 2, E is 4. So, this should be 4.

Now, comes to now we come to this set I 3 and I 3 has got E a here. So, this I 3, a; I 3, a is I 2. So, this should be shift 2 from this item I can say it is shift I 3, b; I 3, b is 3 only. So, this is shift 3. Now, this item tells me that I have to add the reduce rules. So, there these reduce rules are to be added. So, all these reduced rules are added sorry this is reduce by rule number 4 and this I 3, E is I 5. So, this should be 5. Now, for item I 4 there is

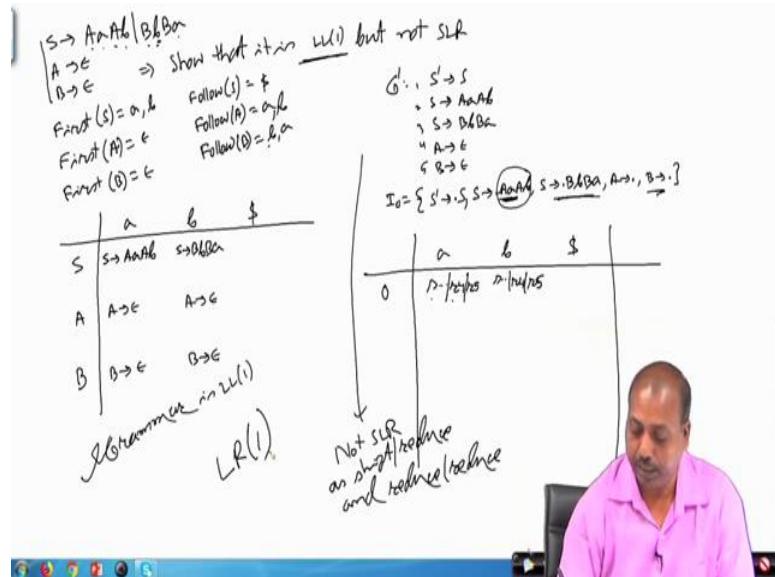
only shift action on b and I 4, b is I 6. So, I 4, b is shift 6. So, I 4 is done now comes to I 5. So, I 5 is this a. So, I 5 a is I 7. So, this is shift 7. So, there is no conflict at this point. So, I 5 is done.

Now, come to I 6. So, I 6, a is a shift operation and I 6, a is I 2. So, this is shift 2. I 6, b is I 3, so, that is shift 3 and then by this E producing dot. So, all this reduction by rule number 4 will come, reduce by rule number 4, reduce by rule number 4. And, your I 6, E I 6, E is I 8; so, this goto part it will have an entry 8. Now, come to item I 7 in I 7 I have got this one a dot a. So, this should be shift and I 7, a is 2. So, this is shift 2 then I 7, b; I 7, b is 3. So, this is shift 3 and this E producing dot. So, this follow of E a, b dollar I have to add the reduce rule reduce by rule number 4. So, they are all reduce by rule number 4 and I 7, E I 7, E is what I 7, E is I 9. So, this is 9.

Now, come to item 8, ok. So, the set 8 I 8. So, there I have got a rule this thing that is rule number 2. So, I have to look into the follow of E that is a, b dollar and there I have to reduce by rule number 2. So, it says it is reduce by rule number 2, reduce by rule number 2, reduce by rule number 2. And, then for I 9; for I 9 I will have this rule. So, that is rule number 3 and so, all of them should be reduce by rule number 3 fine.

So, this way we can construct the SLR parsing table for this grammar. So, essentially you have to patiently construct this LR 0 items, then you have to patiently construct this follow sets and rest of the thing is a bit mechanical. So, you have to look into this individual item sets and see whether you can get a you can get a either shift or reduce for different actions. So, that can be done, ok.

(Refer Slide Time: 11:01)



Next, we will be looking into a grammar which is given by the expression which is ok. So, next we will be looking into a grammar which is very simple, but we will see that it is not always mandatory that a grammar has to be SLR and there so, if we look into the corresponding LL grammar. So, maybe the normally it is seen that if a grammar is LL so, it is also a SLR, but there are some exceptions also. So, next we will be looking into a grammar where we have got that type of exception.

So, the grammar is given by this; so,  $s$  producing  $S$  a sorry  $AaAb$  or  $BbBa$  and then  $A$  producing epsilon and  $B$  producing epsilon. So, this is a grammar and that is given to us and the question is to show that it is SLR, it is LL 1, but not SLR. So for solving this problem, so, first we try to show that this is LL 1. For showing LL 1, so, we have to construct the first and follow sets.

So, we first construct the first of  $S$  first of  $A$  and first of  $B$  ok. So, first of  $S$  is equal to first of  $A$  and first of  $B$  and first of  $A$  has only epsilon in it. So, this has got only epsilon in it and then since  $A$  can be reduced to epsilon. So, this set this first contains  $A$  and from this  $B$  can be reduced to epsilon, so, this can contain  $B$ . So, that is the first set.

And, the follow set; so, follow of  $S$  then follow of  $A$  and follow of  $B$  ok. So, follow of  $S$  since  $S$  is the start symbol dollar is there, then it says that the follow of  $A$ , so,  $A$  can be followed by small  $a$ . So, this is small  $a$  and  $A$  can be followed by  $b$ . So, this is also there

a and b. Similarly follow of B, so, B may be followed by B small b and B may be followed by small a. So, that is also same b or a.

So, after constructing this follow sets so, we first and follow sets we try to construct the predictive parsing table. So, if we can successfully construct this table then the grammar is definitely LL 1. So, we have got S, A and B and we have got the terminals a, b and dollar. Now, coming to the first rule S producing AaAb; so, this so, you have to looking to the first of this and first of this whole thing contains this small a because A can be reduced to epsilon. So, this rule is added here S producing AaAb. And, for the second rule it says that I can since this can the first set can contain the small b so, this will be added Bb Ba.,

Then, this A producing epsilon; so I have to look into the follow set of A and add this rule here. So, A producing epsilon; A producing epsilon and for the third rule B producing epsilon, so, I have to add the rules there B producing epsilon, B producing epsilon. So, for this grammar you see that there is no entry in this parsing table which is multiply defined. So, this grammar is LL 1. So, the grammar is LL 1.

Now, what about the SLR case? So, for constructing the SLR parsing table, so, we first augment the grammar. So, we get the augmented grammar G dash as S dash producing S, S producing AaAb, S producing BbBa and A producing epsilon, B producing epsilon. After this we have to set the LR 0 item. So, first so, I 0 is constructed as S dash producing dot S and then S produces dot AaAb, S produces dot BbBa, A produces dot B produces dot. So, this is the I 0, ok.

Now, we can construct the full set of items, but looking at this one itself we can try to understand like what will be the action part. So this is my set I 0; then these are the terminal symbols a, b and dollar in the action part. Now, this rule this item tells me that whatever is in first of A, so there I have to whatever is in first of this right hand side so, there I have to do a shift operation. So, since this first of this right hand side A can be reduced to epsilon, so, first of this contain this small a. So, this says that shift by something a shift I am going to a new state. Similarly, this item will tell me that you do a shift and go to a new state.

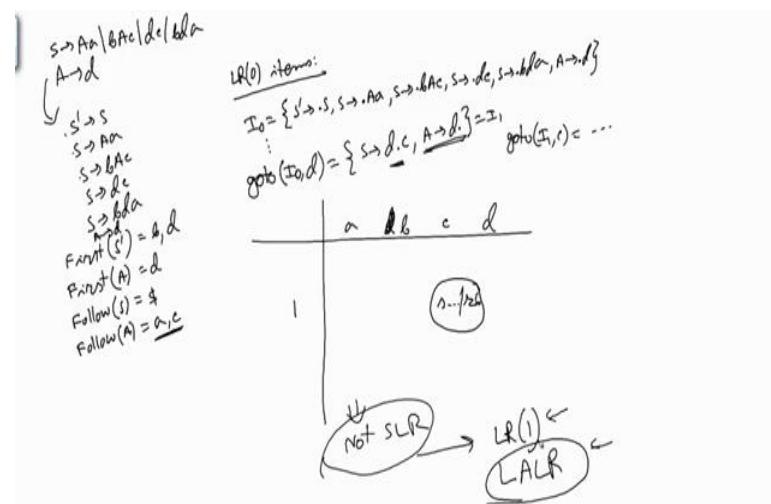
Now, this A producing dot is there, so, that means, that it tells me to do a reduction; reduction for all the symbols which are in the follow of A. So, follow of A contains a and

b. So, this will also suggest to do a reduction by rule number say 1, 2, 3, 4 5 reduction by rule number 4. This will also tell that you do a reduction by rule number 4. Now, when we come to this item B produces dot, so, that says that whatever is in follow of B there I have to do a reduction by this particular rule. Follow of B contains b and a; so, this will have reduce by rule number 5 this will also have reduce by rule number 5.

So, in this item itself you see that we have got shift-reduce conflict as well as reduce-reduce conflict. So, this grammar that we are trying, so, this is not SLR. So, this is not SLR as both shift-reduce and reduce-reduce conflicts are coming shift-reduce and reduce-reduce conflicts are coming. So, that way we can find some cases where the grammar is not SLR, but the grammar is LL 1. Of course, you can you may see that this grammar can be we for this grammar we can make say LR LR 1 parser. So, if we are doing this canonical LR it may be possible that we do not get this type of conflicts. So, that is that we will see later.

So, that way whatever the statement that wherever you can get an LL parser so, we can also get an LR parser. So, that statement is not violated by this example. This example just shows that there may be the case where the grammar is LL 1, but not SLR. So, it may be the grammar is LR, but it is not SLR. So, only that part ok. So, this way you can you have to see like what is happening to the grammar by looking into the corresponding table and then we have to look into the corresponding sets.

(Refer Slide Time: 20:05)



Next, we consider another example which is where we try to show that the grammar is not SLR. Say we have got a grammar like this say S producing Aa or bAc or dc or this bda where the S and A they are the non terminals of this grammar and rest are all terminal symbols and A produces d. So, this is the grammar that we have.

So, to see that whether it is an SLR or not first of all we do this augmentation S dash producing S S producing Aa, S producing bAc, S producing dc, S producing bda, fine. The next part of the job is to find out the first and follow sets. So, the first of S dash; so, since we have to show that whether it is SLR or not so, we first do we do this first follow computation before going to a LR 0 item because that may help us in identifying the negative cases much earlier.

So, first of A is equal to d definitely because that can only give me this thing and first of S is b and d, now that is the first. Now, the follow set follow of S and this follow of A; so, follow of a you can see a small a can come, small c can come ok. So, this is follow of A and follow of S since it is the start symbol so, this is dollar ok. So, this is the follow set.

Now, if you try to construct this SLR the LR 0 items. So, now, we try to construct the LR 0 items. Now, I 0 is S dash producing dot S comma S producing dot Aa, S producing dot bAc, S producing dot dc and S producing dot bda. Oh, that other rule is there that A producing d, so, that is not written here and a producing dot d. So, this is the LR 0 item I 0.

Now, I have to construct this goto s. So you can construct all those gotos, but let us see what happens to this goto I 0, d? goto I 0, d will tell me that this S producing d dot c will be one item and A producing d dot will be another item fine. So, if this item I call I 1, then when I am trying to construct the SLR parsing table so, for this cd etcetera. So, we have got the terminals a, b, c, d.

Now, I have got this say this set is I 1. Now, for I 1, when I am trying to look into this individual items so, this rule will tell me that you should do a shift operation. So, this 1, c should be a shift to some state fine. So, whatever be the goto I 1, c. So, whatever be the goto I 1, c equal to so, to that state it will go.

Now, what about this rule this item, so A producing d dot? So, it says that you have to look into the follow of A and there you have to do the deduction. So, follow of A is a and c. So, you can understand that this will give me a reduce it will give me a reduce by rule number whatever be the rule number. So, a so, it is 1, 2, 3, 4, 5, 6 it will give me a reduce by rule number 6. So, this way it gives rise to a shift reduce conflict here. So, this will give rise to a shift reduce conflict here and the grammar will not be an SLR grammar. So, this is not SLR because it is giving a conflict like this.

Now, it can be shown that this can successfully give me LR 1 parser it can successfully give me LR 1 parser. In fact, for this particular grammar you can see that this can also give me LALR parser, ok. So, we will see the, we will do that exercise later we will see that this particular grammar though it is not SLR. So, it is LR 1 as well as it is LALR 1, but it is not SLR, ok.

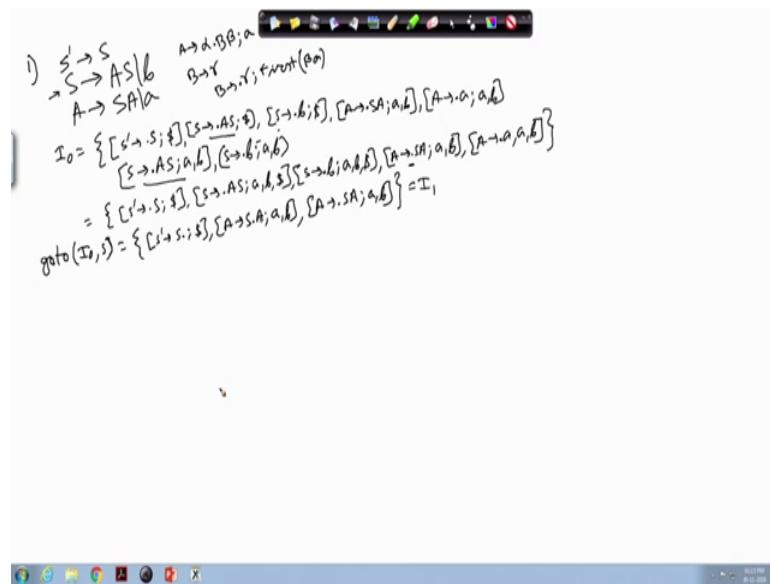
So, we will continue in the next class.

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

**Lecture - 37**  
**Parser (Contd.)**

So, today we will look into some more examples of these parsing techniques particularly the LR parsers and LALR parsers. In the previous classes we have seen examples of SLR parsing. So, today we will first look into some LR parsing at a table construction method.

(Refer Slide Time: 00:33)



So the grammar that we will consider is it will be a simple one and so that we have got less number of states, but you will see that even with that less number of states. So, that is going to take quite some number of states compared to this SLR even if the grammar is simple. So this LR parsers it will have large number of states.

So, the grammar that we consider is like this we have got this S producing AS or b and then we have got A producing SA or a. So, this is the grammar. So you have got the where this S capital A so these are non-terminal symbols and small a and small b these are terminal symbols. So, the first step is to augment the grammar by putting this S dash just producing S as one production and then we have to construct the set I 0. So, I 0 is the closer of the item this first production S dash producing dot S with look ahead dollar and

then we have to consider the closer of this and closer of this will have all these things like S producing dot AS dollar then S producing dot b and dollar.

So, like that will have two rules and after that, since we have got this S producing AS and this S dot a is there. So, I have to construct or to see what is A producing things. So in that way I will get the items like A producing dot SA and then the rule says that when you are doing this S producing A producing dot SA then, I have to say what is what will be the terminal symbols after this a look ahead symbols. And, the look ahead symbols rule was like this that; if you have got a rule like A producing alpha dot B beta then and if there is a rule like B producing gamma then I will add this item B producing dot gamma to this set and the look ahead will be the first of so, there is a.

So, this was the item alpha dot B beta semicolon a. So, a was the look ahead. So, I have to look into the look ahead of first of beta a. So, whatever comes in the first of beta a so, all those symbols will come here. Now, you see that when I am looking into this particular production this particular item S dash S producing dot AS then, I have got these things. So, dot is before a so, the beta part is capital S. So, I have to see what is the first of capital S and first of capital S you see b will come and a will come. So, while adding this item the look ahead symbols will have both a and b. So, that way the look ahead symbols will be added and then there is another rule a producing dot a.

So that will also come A producing dot a and the look ahead symbols a and b. So, they will come. So, this way this item will get added and now you have got these things this A producing dot SA. So, A producing dot SA so, again I have to apply this rule. So, this dot S. So this from this I will get the item S producing dot AS and then I have to see whatever is in the first of A. So first of A can have this small a as well as this small b both of them can be there in the first of A. So, that that way the look ahead it will be a and b ok. So, this is another item added.

Now, this thing we can simplify now like now you see that this item and this item they are core part is same only the look ahead part is different. So, this is a b and this is dollar. So, we will write it together. So, ultimately we will get it the items like this. So, S dash producing dot S semi colon dollar this will be there then S producing dot AS and the items the look aheads will be a b and dollar ok.

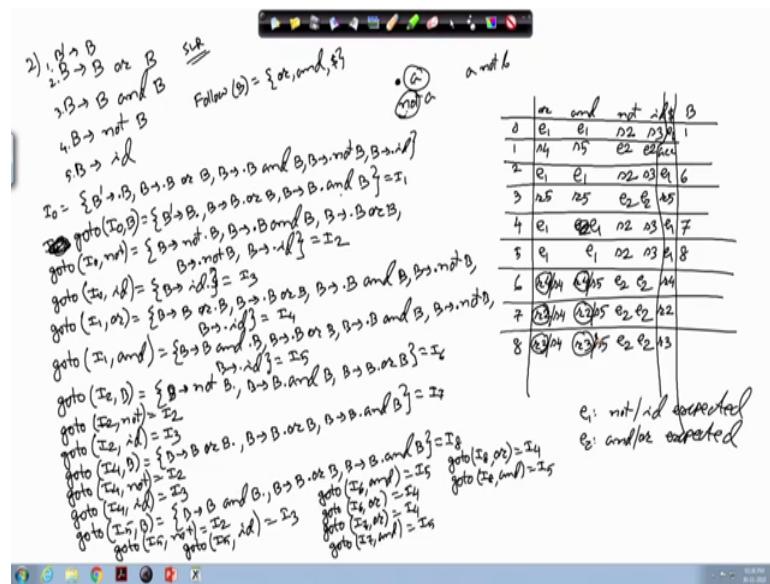
Now, I will have this S producing dot b. So, S producing dot b and here also I will get the look ahead a and b. So, like just is in this case we had this. So, for this one also I will have this a, b and dollar so, a b. So, this item will also be there. So, it is not I have not written it. So, this S producing dot b will come with the look aheads a and b. So, this item is also there now I can club these two because their core part is same. So, this is a b dollar and then I have got this a producing dot SA a, b then A producing dot a a b.

So, this is the item setup item I 0. Now for I 1 so, I have to do go to I 0, S; go to I 0 S will give me from the first one I will get S dash producing S dot semi colon dollar. So, this is giving me then dot S. So, this rule is there so, this will give me A producing S dot A semi colon a, b then since this S producing dot A is there. So, I have to take this A producing things also. So, I will get the items A producing dot SA with the look aheads of first of A. So, first of A has got both a and b. So, a and b is already there so, I will get this A producing dot SA with a, b ok.

So, then there is this A producing dot SA is there. So this will give us A producing dot A producing S dot a. So, that is there and this will be the thing like I will have this production then A pro A this there is A dot before a. So, this somewhere I 0 S. So, this S producing this one S producing S dot A and then I will have this from this rule I will get S producing S dot A ok. So, this will be the set I 1 ok. So, this way we have to go on constructing the different items and there will be a large number of items created. So, it is very difficult to do it by hand.

So, but in this way if you continue then you will get the set of items that are coming and then you can combine them using some; if some of them are same then you can combine some of the items are same you can combine them. And, then ultimately you can make the parsing table and from the parsing table you can go into the parsing part of it. So, this way we can construct the table. So, next we will be looking into another example where we try to we will be doing SLR parsing only, but we will try to see like how this error functions are taken care of. How this so, for that purpose so we take the grammar.

(Refer Slide Time: 19:29)



We take the grammar which is for the Boolean expressions. So, Boolean expression grammar is like this. So, B producing B or B, then B producing B and B, then B producing not of B and B producing id. So, if this is the grammar then we have to add that initial production B dash producing B. So, these are the rules that we have. So, these rules are number like 1, 2, this is 3, this is 4, this is 5. Now, if we try to so, we are trying to construct an SLR parsing table and we will embed the error functions into it. So, what are the errors that will see and accordingly we will put the error productions also.

So, this LR 0 items so, the I 0 is given by this B dash producing dot B and then whatever it is so, all these rules will come B producing dot B or B, then B producing dot B and B, then B producing dot not of B and B producing dot id. So, this is the I 0 part. Now for the I 1 we will so, for so, we will see like go to I 0 B go to I 0 B so, this will give us B dash producing B dot, then B producing so, from here B producing B dot or B B producing B dot and B. So, these are the three items. So, this is the set I 1.

Now, I 0 the I 0 not is possible. So, go to I 0 not so, this will give us B producing not dot B and then all these rules will come that B producing dot B and B, B producing dot B or B, B producing dot not of B and B producing dot id. So, all these rules will come. So, this is my set I 2. Now go to I 0 id; go to I 0 id is B producing id dot. So, this is the set I 3. Now for from I 1, I can get go to I 1 or this will give me B producing B or dot B and

since dot B is there. So, all these items will come B producing dot B or B, B producing dot B and B, B producing dot not of B and B producing dot id. So, that will be the set I 4.

Now, go to I 1 and go to I 1 and so, this will give us B producing B and dot B and then all these rules will come. So, B producing dot B or B, B producing dot B and B, B producing dot not of B and B producing dot id. So, this is my I 5. Now, with I 1 everything is done, now have to do with I 2 ok. So, then this one is there B. So, go to I 2 B. So, this is B producing not of B dot and then this one B producing B dot and B then, B producing B dot or B ok. So, that will give us set I 6. Then I 2 B I have done, then I 2 not go to I 2 not; go to I 2 not so, this will give me B producing dot not dot B. So, that is same as that item I 2. So, go to I 2 not is I 2 only.

Then go to I 2 id, this is B producing id dot that is equal to I 3 fine. Now I 4 go to I as I from I 3 I cannot get anything from go to I 4 B. So, this will give me B producing B or B dot, then this one B producing B dot or B, then B producing B dot and B ok. So, these are the three production. So, that is my I 7 that is I 7. So, I 4 B is done. Now I 4; I 4 not go to I 4 not this will give me same as B dot B not dot B. So, B not dot B. So, that is the item I 2; so, this is I 2 only then go to I 4 id I 4 id is B producing id dot. So, B producing id dot is I 3. So, this is I 3 only. So, I 4 is over now I 5. So, go to I 5; go to I 5 B is B producing B and B dot and then this B producing B dot or B then, B producing B dot and B ok.

So, this is the item I 8. Now I 5 B is done. Now other possibilities from I 5 is I 5 not. So, go to I 5 not is not dot B. So, that is equal to I 2. So, that is equal to I 2 then go to I 5 id is B producing id dot. So, that is I 3 ok. So, I 5 is over. Now I 6 ok so, go to I 6 on and I 6 and will give me this item this one this I 5 only. So, this is B producing B and dot b. So, that is the from here I will get B and dot b. So, it will give me I 5. So, go to I 6 and this I 5 go to I 7. So, I 6 is go to I 6 or go to I 6 or is B or dot. So, that is I 4 ok. So, I 6 is over.

Now, from I 7 I will get go to I 7 or so, this will give me B or dot B so, that is I 4 and I 7 and I 7 and will give me B and dot b. So, that is I 5 and from this set I 7 so, I 7, I have done from I 8. So, go to I 8 or will give me B or dot B. So, that is I 4 and this one go to I 8 and will give me B and dot B. So, B and dot B that is I 5 ok. So, this finishes construction of these items sets of items now we can construct the parsing table for so, for doing the for our purpose the parsing table construction. So, I have got 9 states. So,

the states are the items are like 0, 1, 2, 3, 4, 5, 6, 7 and 8 and that terminal symbols that I have or, and, not and id.

And then the go to part I have since I have got a single non-terminal B. So, that is B and also I need to constitute the follow set; the follow set of B; follow set of B it has got or, it has got and ok. So, follow set of B has got or and. And of course, since we have got B as the start symbol of the grammar. So, dollar will also be there B or dollar or and dollar. Now I have to look into the set I 0. Now, it says that this B producing dot not B is there. So, I have to I have to on not I 0 not is I 2. So, this will B shift 2 ok.

Then I 0 id I 0 id is I 3 I 0 id is I 3. So, this will be shift 3. Now these two entries or and. So, they are blank. So, they are actually error entries and I 0 B; I 0 B is I 0 B is I 1. So, this is 1 now from I 1. So, I 1 there is a B dash producing B dot. So, I will have this one as this dollar is I have not written that dollar. So, dollar is accept here; the dollar is accept and then you see that we have got this or dot before or. So, I 1 or is I 4. So, this is shift 4 and then, and is shift 5 sorry I am writing at a wrong place. So, this is I 1 so, shift 4 and shift 5 and this dollar is this one then the part before that is id.

Now, this from I 2 from and there is from I 2 you have to see like there is a dot before a terminal symbol at not. So, I 2 not I 2 not is I 2. So, this is shift 2 and I 2 id; I 2 id is I 3. So, this is shift 3. So, these two are done there is no reduction possible in I 2. Now I 3 it is a B producing id dot. So, this is telling me in the follow of B or, and and dollars. So, I have to make it this one. So, this rule number 5. So, or and in I 3 so, this is reduced by rule number 5; this is also reduced by rule number 5 and this dollar is also reduced by rule number 5 ok.

So, this is I 3 is done. Now come to I 4; in I 4 I have got dot before this not and I 4 not is I 2. So, this is shift 2 and I 4 id is; I 4 id is I 3. So, this is shift 3 and in I 4 I do not have any reduction because there is no dot at the end. So, that is done. Now come to I 5 then I 5 then there is 2, I 2 there was go to B or something I 2 B is I 6. So, this will be 6 that is there. Now come to the set I 4 I have done. Now I 5, I 5 again there is a dot before not and dot before id and I 5 not is I 2; I 5 not is shift 2 and I 5 id is I 5 id is shift 3. So, this two remain as it is and then I have this I 5 B; I 5 B is I 8. So, I 5 B is I 8. So, that will be there.

And what is I 4 B; I 4 B is I 7. So, this will be 7 fine. So, I 5 I do not have any dot at the end so, nothing more to do. Now I 6, I 6 I have got a dot at the end. So, for the follow set I have to do reduce by rule number 4. So, or and dollar. So, there will be reduced by rule number 4, reduced by rule number 4, reduced by rule number 4 fine I 6. Now that is for the first one. Now this one dot, and. So, this is telling me B dot B and B. Now this is I 6 and I 6 and is I 5. So, this also says that you shift and go to state 5 and then this I 6 or is I 4. So, this is also telling me to shift and go to 4 state 4.

Now, I 7 I 7 is here. So, I 7 in I 7 we see that this there is a B or dot B or B dot. So, this is the reduction will come. So, or and dollar. So, they will be reduced by rule number 2 or, and dollar they will reduce by rule number 2 and then I have got this thing this dot B or dot B dot or B. So, I 7 or is I 4. So, this will also say shift by go to state 4 and I 7 and is I 5. So, this will tell me shift by go to state 5. So, that will be finishing then this one I 7 I 7 I will have this situation. So, this is I 6 I 6, I 7 then I 8.

So, I 8 we have got this B producing and B dot. So, that is reduced by rule number 3 and here also I will get reduced by rule number 3, here also I will get reduced by rule number 3 and then in I 8, I have got this dot or. So, this will be this will tell me I 8 or is I 4. So, this is shift 4 and this is shift 5. So, that will be constituting the whole parsing table. Now after constituting the parsing table so, there are many entries which are blank fine. Now, I have to frame the error routines and accordingly I can put some error function there ok.

So, let us see, what is the expectation like at this point if you are so, you are at state 0 and then you are expecting that at this point you are got an or. So, your expression started with an or now how can a Boolean expression start? It can start with some identifier say a or it can start with the not symbol not a like that. So, either it has to start with not or an id. So, these two are the valid one. So, otherwise they are invalid. So, at this point I was expecting either a not or an id. So, accordingly I write here 1 error function e 1. So, for both of them this is e 1. So, here also this is e 1.

In state 1, when I am in state 1 then I have seen something with B and I am expecting or and. So these are the two expectations. So, accordingly I can say as if I was expecting some operand ok. So, this instead of that I have got a not ok. So it has come like this some id a and then not B. So, that is that cannot be correct. So, this is also an error, but here I am expecting an operator the and or or and. So, this is either it is and or it is or. So

these are another error function. So we have got 2 error functions e 1 which is expecting an operand or in or the not and e 2 is we are expecting the operand or or and so, either of those two.

Similarly, in state 2 also so, you can see the I 2. So, it was expecting some to some operand or not. So, here also this is e 1 these two are e 1 similarly I 3 so, these two are e 2. So, this is also e 1 ok. So, this way you can fill it up like I 4, I 4 I was expecting again not or this thing id sorry this is also e 1 not or e 1, then in I 5 I was expecting in I 5 I was expecting to see some not or id. So, this is also e 1, in I 6 I was expecting and or or. So, these are e 2. This is also e 1, this is also e 1 and this is in I 7 I was expecting or and and. So, this is also operand expected in I 8 also I was expecting or an end. So, this is e 2.

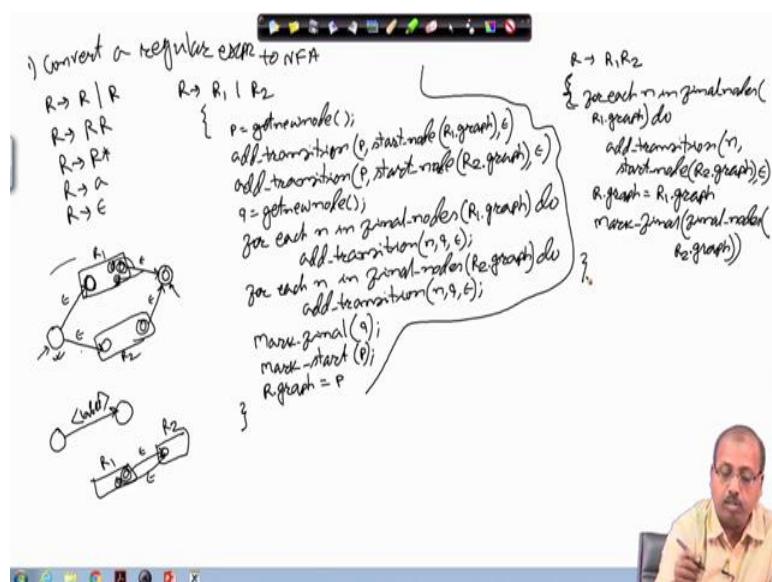
So, where e 1 is not or id expected. So, this is the error function that we have and in e 2 we have got and or expected. And of course, that is a shift reduce conflict at this point. So, here you are expecting the to end the expression, but you have got an or operator. So, instead of shifting so, I should do the reduction because that is a not id they have higher precedence than or. So, I will this so, this should be the rule. So, here also this should be the rule, here also I should do the reduction not shifting and here I will be doing this reduction only ok. So, that way you can incorporate this error functions into this parsing table and make the parsing table full so that you can do error detection and recovery.

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

**Lecture - 38**  
**Parser (Contd.)**

Next, we will be taking some examples of syntax directed translation. So we will be taking an exercise which will be like this.

(Refer Slide Time: 00:23)



So, we want to know the grammar for regular expressions. So, from the regular expression we need to convert to NFA. So can we have a compiler assisted technique for doing that? So, the job is to convert a regular expression to NFA ok. So, this if we can do so, we can have an automated tool for that purpose of course, in R lexical analysis chapter we have seen the things to be done. So you can formalize it and put it into 1 compiler constructions step. So, that you can do this we can have this conversion part automated.

So, for the regular expression the grammar that we consider is like this  $R$  producing  $R$  or  $R$  then  $R$  producing  $RR$  then  $R$  producing  $R^*$  and also we have got the terminal symbols like say  $a$ . So  $R$  producing  $a$  and it can give me  $\epsilon$ . So, these are the rules that we have for the regular expression. Now, for syntax directed translation so, what we will assume is that we have got some attributes. So each regular expression for that we

have a corresponding NFA and the NFA has got a start state and a number of final states; so, that these are there. So if we assume that for every regular expression; so, we will maintain its corresponding NFA as an attribute of it.

Then we can combine those attributes as say syntax directed translation actions and so, that I can get the NFA for the complete expression, for the complete regular expression. So, let us take the first rule like say  $R$  producing  $R_1$  or  $R_2$ . So, I am purposefully writing it as  $R_1$  and  $R_2$  so that in my action part; so, I can very clearly mean like which attribute whose attribute I am talking about ok. So, otherwise they are all regular expressions  $R_1 R_2$  so, they are all regular expressions. So, on the right-hand side I have got to regular expressions they are concatenated, but they are connected by the OR operator and then that gives me the overall regular expression. So and further rule for this if I have got a regular expression for  $R_1$  and there is a regular expression for  $R_2$ .

So, it has got a start state and the final state. So, it has got a starts to the final state and so, this is for  $R_1$  and this is for  $R_2$ . Now, if you look into this construction example then it says that you have to have a new state and add some epsilon transitions to them ok. And, then you should have another new state which will be called the call the final node then you will be adding epsilon transitions from here. So, this is the action to be done ok. So, how are you going to write? So, we write it like this. So,  $p$  is a pointer which is get new node, we assumed that we have a function get new nodes. So, if you call this function it will return you a new node of the graph ok.

So we will call that function. Then I have to show this is the I have got this new node now, I have now had to add this epsilon transitions. So, I do add the transition add transition what transition? So, a transition when I say so, I have to tell the start state, I have to tell the final state and I have to tell what is the label. So, these are the three things to be saved ok. So, start status this new state that I have taken then this that destination state is the state I am talking about. So, this  $R_1$ 's start node. So, if I assume that there is a function called start node. So, such that if you pass the graph of  $R_1$  to it so, it will return its start node. So, this is start node of  $R_1$  dot graph and I have to tell what is the label. So label is epsilon.

So, I assume that there is a function called at transition. So, if you call this function so, it will add a transition between once node 1 to node 2. So, node 1 is  $p$  in our case node 2 is

the start node of R 1 and then the transition that is added is labeled with epsilon. So, that is add transition then I can do, this is done. Now, for the second one I have to do another transition has to be added for R 2. So, for that I can do add transition P start node of R 2 dot graph and label is epsilon that add transition can be done. Now so, this part I have made. So, once I have this R 1 and R 2 these graphs available. So, I have made this left part of the diagram.

Now for the right side, for the right side what I have to do is, I have to get another node. So, I say q, q equal to get new node and that will be a terminal node. So, we will come to come later how to make it a terminal node so, that we will do later. Now, this node this graph R 1 may have several final states and for each of these final states are to add this epsilon transitions. So, I write like this that for each n in final nodes of; so, I assume again that there is a function called final nodes that returns the set of final nodes of a graph. So, R 1 dot graph ok. So, for each node n in final nodes of R 1 dot graph do we add transition, add transition from this final this n to q with label as epsilon fine.

And similarly, for R 2 also I need to do the same thing that for each n in final nodes of R 2 dot graph do add transition; add transition n q epsilon. So, once we have done that so, these transitions have been added. Now, what we have not done so, far is, marking this node as the start node marking this node as the final nodes. So, marking p as start node and marking q as the final node of the graph.

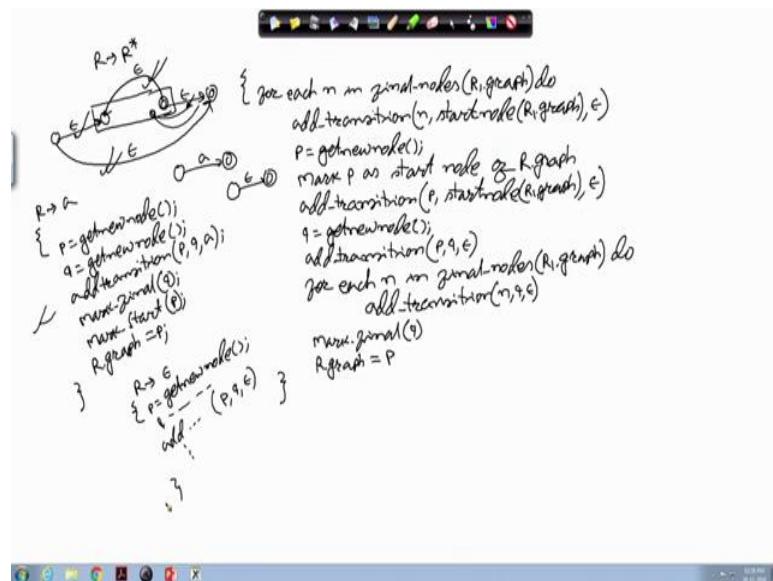
So, we assume that there is a function called mark final which will mark the q node q as a final node and then we say that we are to mark p as they start node. So, I say that as if as if I have got another function that mark start p. So this is made the start node of p. And, then the attribute R dot graph is made equal to p. So this way I can write the syntax directed translation scheme.

So, that for R 1 or R 2 I can get the corresponding NFA who by combining the NFA's of R 1 and R 2. So, next we will be looking into the NFA for this R 1 R 2. So, the second rule. So, R producing concatenation R 1 R 2. So, here so, we can say that the rules that will the code that we will have are the actions that we have is that. So, here I have to add transitions like if this is R 1 and this is R 2 then what we do? Is from the final state of R 1 from all final states of R 1; so, we add epsilon transitions to this initial state of R 2. So,

that is the action. So, we write it like this that for each n in final states final nodes of R 1 dot graph final nodes of R 1 dot graph do.

So what we do? We add transition. So we add transition n from n to the start node of R 2 dot graph start node of R 2 dot graph and the label being epsilon. So, this is the transition that we have added and then we have to say that as now R dot graph can be simply made equal to R 1 dot graph and then I have to mark the final states. So, mark final for this particular graph R. So, I have to from the final states of R will be the final states of R 2. So, this is mark final. So, from the by applying the function final nodes I get the final nodes of R 2. So, that is R 1 R 2. So, this way we can do it for this R , R concatenation of 2 states.

(Refer Slide Time: 11:39)



Now, I need to do it for R star. So how do we do it for R star? So R producing R star and this is a bit tricky because, you know that what we need to do is, if this is the situation, then I have to have one epsilon transition from here to here. I have to have an epsilon transition from here to here, I should have an epsilon transition like this and I should have an epsilon transition like this. These are the things to be done for converting R 2 from R2 get the NFA for R star ok. So, what we do? We do it like this. So, I have to add this epsilon transitions. So, first we do the epsilon transition edition from a final states of R to its initial state.

So, that part we do we write it like this for each  $n$  in final nodes of  $R$  1 dot graph ok. So, what do we do? We add the epsilon transitions. So, that is add transition  $n$  start node of  $R$  1 and the label is epsilon. So, these transitions have been added. So, these transitions are now done. Next I have to add this start state. So for that I have to get a new node. So,  $p$  as get new node and then this is the start node of the of the graph  $g$ . So, mark  $p$  as start node of  $R$  dot graph. Now, I have to add the epsilon transition from this node  $p$  to the start of  $R$  1. So, we say add transition  $p$  comma start node of  $R$  1 dot graph and the label being epsilon.

So, that this transition has been added now, now I have to add this transition. So, for that I have to get another node so, that is called  $q$ . So, that is another get new node. So, after I have got the node so, I do an add transition; add transition  $p$   $q$  epsilon. So, this transition is added from  $p$  to  $q$  epsilon label being epsilon.

Now, I have to add this transition. So, for every final state that I have here in  $R$  1 so, this epsilon transitions are to be added. So I do it like this for each  $n$  in final nodes of  $R$  1 dot graph do add transition  $n$   $q$  epsilon ok. So, these epsilon transitions have been added and then I have to mark this  $q$  as the final state. So, mark final  $q$  and then we have to say that this  $R$  dot graph equal  $p$  ok.

So, this way we can write down the syntax directed translation scheme for the converting this regular expression  $R$  2  $R$  star. Now, there are two more simple rules that I have got. So,  $R$  producing  $a$ ; so, for  $R$  producing  $a$ , so what I have to do is, have to have something like this and this is  $a$ , this was the thing.

So, how to do this? So, I right like  $p$  equal to get new node, you have first of all create two nodes: one for one will act as the start node other will act as the end node, the terminal road and  $q$ . So, the they are the two nodes that I have got  $p$  and  $q$  and then I have to add transition from  $p$  to  $q$ ; add transition from  $p$  to  $q$  and the label of the transition is  $a$  ok. Then I have to say that  $q$  is a final state so, that that can be done by calling the function mark final then to mark  $p$  as the start node.

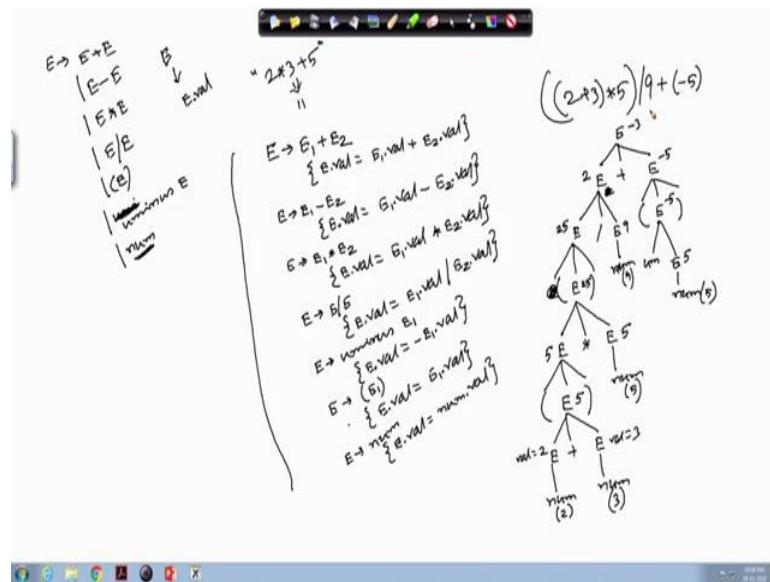
So, that is by calling the function say mark start  $p$  and then I have to say that  $R$  dot graph equal to  $p$ . So, this is for  $R$  producing  $a$ . So, only the single terminal symbol and what remains is the epsilon transition  $R$  producing epsilon. So,  $R$  producing epsilon so, here

also it is similar only thing is that this label is epsilon. So, the code for this is the same as this one, but only instead of we will be adding the epsilon.

So, we will have like p equal to get new node q equal to get new nodes everything will be same only in the add transition part so, this should be p q the label is epsilon. So, rest of the thing will be as it is. So, if you have this set of rules then center so, by syntax directed translation mechanism; so, you can easily generate the corresponding NFA from some regular expression.

So, next we will be using another example for this syntax directed translation which is for evaluating the arithmetic expressions. Suppose, we have got the arithmetic expression grammar which is given by this.

(Refer Slide Time: 19:33)



So, E producing E plus E or E minus E or E star E then E division within bracket E and then unary minus unary minus, we write it as a separate a separate terminal symbol u minus. So, so that to differentiate it from the binary minus and it can be a simple number. Suppose this is the grammar that is given so, we know how to general generate the corresponding parsers. So, that we have already learned. Now, for syntax directed translation like we want to evaluate an expression. So, how to do this? So, we assume that with this with a non-terminal E so, we have got an attribute which is E dot val. So, which will hold the value of the expression.

So, if the value of the expression is a number as you note that this is not an id so, this is a number. So, at the compilation stage itself we will be able to compute the value of the expression.

So, it does not need to go to the runtime thing. So, that is why so, it is useful like you maybe while writing some expression we get we have written like 2 into 3 plus 5. So, for the programmers is so, it is written like this, but the compiler can easily convert it into the number that is 11. So, how that conversion from this; so, this is coming in the source text and how it is getting converted to 11. So, that you can do by means of this syntax directed translation mechanism and we assume that we have got an attribute val. So, which will be holding the value of the expression.

So, you can very easily guess like what can be the syntax director translation schemes for this thing like say E producing E 1 or E 2 ok. So, in this case the action is nothing, but so, I have to compute this E dot val equal to E 1 dot val plus E 2 dot val. So, this simply maybe the action. So, because those two values are to be added so, E 1 dot val that attribute contains the value of the expression E 1 and E 2 dot val contains the value of the expression E 2. So, I can just add them by making this E 1 dot val plus E 2 val plus.

Similarly, this E 1 minus E 2 so, E producing E 1 minus E 2. So, there also you can write the sentence directed translation scheme like E dot val equal to E 1 dot val minus E 2 dot val fine. Then E producing say E 1 star E 2; so, there also you can write like E dot val is E 1 dot val multiplied by E 2 dot val. Then E slash E division E slash E so, this E dot val equal to E 1 dot val divided by E 2 dot val; then unary minus E 1. So, this I can say that E dot val equal to minus of E 1 dot val.

And E producing number so, within bracket E, E producing within bracket E E 1. So, this is very simple that E dot val is nothing, but E 1 dot val and then I should have this E producing number. So, here this E dot val will be equal to the lexicon analyzer is, if we assume that it has written the value of this token number on to the into one attribute of it which is num dot val. So, this will be equal to num dot val ok. So, you can check like how it will work, like if I have got an expression like say 2 plus 3 into 5 slash 9. So, if we have got something like this then say plus minus 5 ok.

So, how will it look like? The expression the parse tree will be like this. So, this is E plus E, this E gives me within bracket E, this E gives me unary minus then E, then this E

gives me number which is equal to 5. Then this part will be giving me E slash E and this slash E. So, this E gives me an gives me a number whose value is 9. Now, this E will be giving me E sorry within bracket E like that and that will give me E star E. Now, this E will give me a number which is equal to 5; now this E will give me within bracket E. So, that will give me E plus E. So, that will give me number that is 2, this will give me number that is 3. So, this is the parse tree that will be produced by the parser.

Now, this syntax directed translation rules so, they are applied when we are doing the reductions. So, when the shift reduce parser so, it will do the reduction. So, it will first do this reduction. So, this as a result the corresponding val attribute will be equal to 2 by this rule. So, this E dot val equal to num dot val. So, these val equal to 2. So, these val will become equal to 3.

Now this will be doing the addition E 1 dot val plus E 2 dot val. So, this value will become equal to 5, now this is within bracket E so, this will also become equal to 5. Now this will make it 5 so, now this is 5 into 5. So, this will make it this value will be made equal to 25. So, this will also be made equal to 25 because, E within bracket E is E dot val equal to E 1 dot val by this rule this is also 25.

Now, this when this reduction is done so, this E dot val will be equal to 9. Now, when this reduction will happen so, 25 by 9; so, if I assume it is an integer division so, this will give me 2. So, this E is E dot val attribute will be equal to 2. Now this side so, this E dot val equal to number dot val that is 5 then this unary minus so, this will become minus 5.

So, this will give us minus 5 and finally, this E dot val so, it is 2 plus minus 5 that is equal to minus 3. So, it will get that E dot val computed as minus 3. So in this way you see at the syntax directed translation mechanism so, we can write down a set of actions. So, that if those actions are taken during the parser reduction process; so it will be able to compute it will be able to do much useful computation.

So, in this particular example so, it is doing it is for finding the value of integer expressions and previously we have seen a technique by which we can convert this regular expression to NFA. So, that conversion so, we can I have appropriate functions written so, that I can easily get the corresponding graphs. So this way this syntax directed translation can be useful for in the parsing process in in integrated fashion with the parsing process so, that you can do many useful jobs done by the parser itself.

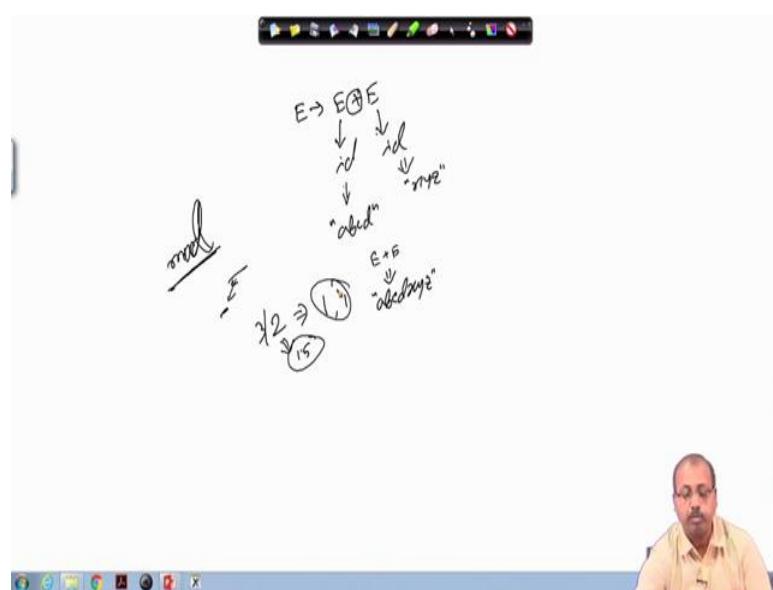
**Compiler Design**  
**Prof. Santanu Chattopadhyay**  
**Department of E & EC Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture - 39**  
**Type Checking**

The next part of our course, it will be talking about Type Checking. So type checking is very important because what happens is that in most of the programming languages, they use variables and expressions they have got some types. And most of the time this with this depending on that type so certain word lengths are allocated to individual types. For example in many computers you can find that integer is given 2 bytes or floating point numbers are given 6 bytes. So, like that. So, if the type is not correct, then this computation becomes difficult. So, that is one thing.

Second important point is that see whenever I have got an identifier. So, we can the grammar rule says that, if I have got say E producing id. So E producing id plus id something like that then

(Refer Slide Time: 01:10)



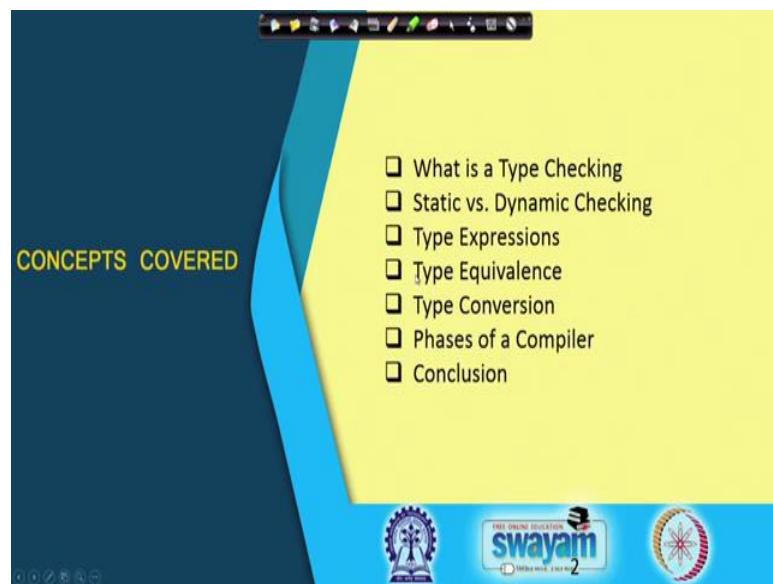
So, grammar rule has told me that we have got this type of rule. So, E producing E plus E and we have seen that this E can give me id this E can finally, give me id. Now, but this plus operation; so it is not defined on different types of identifiers. For example, if it is integer integers, then this id plus id has got a meaning. But if it is say for example, say

character variable. So, what do you mean by this plus. Similarly sometimes it may be meaningful, but in a different context. For example, if this is so, they are corresponding to some string. So, maybe the first this is one string. So, abcd and this is another string say xyz.

And in that case this E plus E. So, this E plus E may mean that this is the concatenated string abcd xyz whereas, the it really has a totally different meaning when you have got this either expression or these ids as integers. So, this way we need to check like what which operation that will be that is we are that the programmer is willing to say whether it is a string operation or whether it is an integer operation and integer operation. So, like that is one thing and many things are not allowed also on different types for example, if we say one many of the computer programming languages they will support the modulus operator. So, modulus operator is say it will do an integer division and as a result. So, this will give me the remainder part.

So, this after division the result and the remainder so, they are often true when we have got the integer arithmetic. So, when you are dividing 3 by 2. So, result may be 1 and with a remainder of 1. So, if it is an integer division, but if it is a if these are taken as real numbers, then the answer should be 1.5. So, that way it is whether I am going to follow a real or real number division or an integer division. So, that is determined by the type of the operands. So, it is very important that we should know that types not only for determining the size of individual of identifiers, but also to know whether certain types of operations are allowed in that on that particular structure or not. So, that is that these are the two things for which we will see this compiler designers. So, they will be using this type checking.

(Refer Slide Time: 04:04)

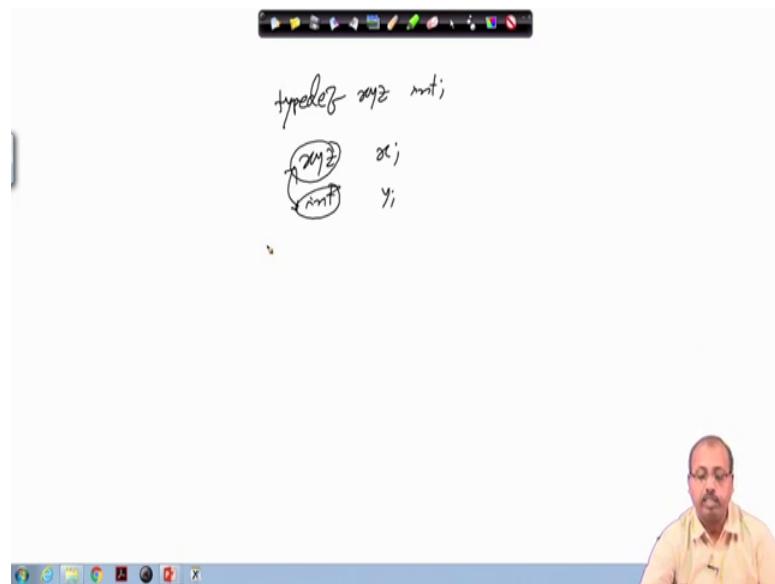


So topics that we are going to cover are like this. So we will see what is type checking that is the first thing, then this type checking may be static or may be dynamic. So static means so, it is done at the compile time itself and dynamic means while the program is in execution, we do the type checking. So, both have got their implication. So, step most of the programming languages they will go for static type checking, but many a times we will see that some programming languages they have allowed dynamic type checking. So dynamic type checking so that is the help full because the same variable x defined in two different contexts. So they may have different bearing. So, in one case if x may be considered as an integer, in another case x may be considered as string variable. So, like that it can happen. So we can have this static versus dynamic checking.

Then there are for type checking so there is something called type expression. So, if you are given a complex structure like, if you look into the C programming language then you see they we have got this structure and union type of declarations where a number of data items are combined together to make the overall structure. So, what is the type of the overall structure or when I have got an array of fundamental types like the array of integers and array of real numbers like that. So, this array of the certain types of that also creates a new type so, that is an array type. So that way is also some new types. So, how to formalize this whole form whole type construction phase; so that will be the type expression.

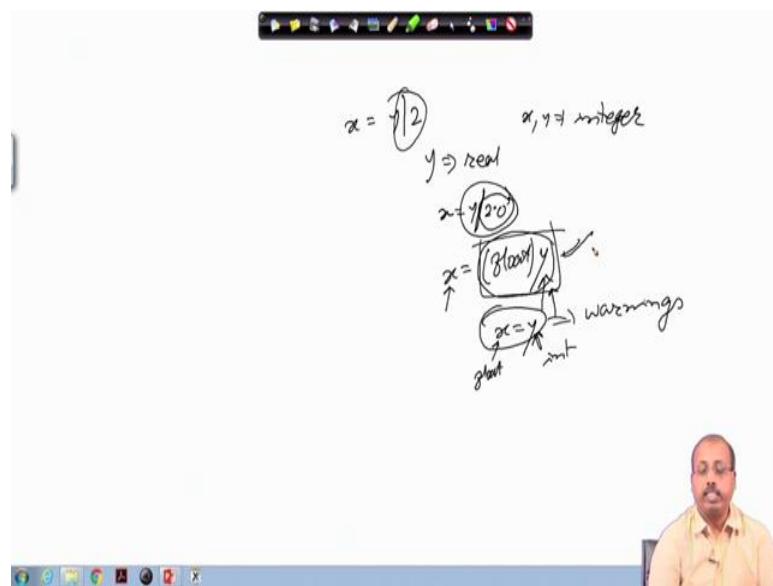
And we need to check type equivalency very often we need to answer this question whether this x and y their types are equivalent or not, they might not have been defined to be of same basic type, but it may so happen that the user might have redefined as the integer as another type like we can always say in C language you know that there is a typedef type of thing.

(Refer Slide Time: 06:15)



So, we can write like this typedef say if say xyz integer. So, later on if I have a variable x y z so, a variable x of type xyz and another variable y of type integer. So, are they equivalent or not? So, this is a very difficult question to answer like if you just look at this xyz so and this int so, they are of different types, but if we go further. So, we can see that they are derived from the same basic types. So, they are going to be equivalent in that case. So, whether we go to that level of equivalency check or not that is a different question. So, that is often dictated by the programming language, but if it has to be supported then, it becomes difficult because we have to do it more rigorously. So, this type equivalency we have to do. Then many times you will need type conversion.

(Refer Slide Time: 07:22)



So, type conversion is like this that suppose, I have got an operation in suppose I am writing a program in C language and where I am writing say  $x$  equal to  $y$  divided by 2 and here if this  $x$  and  $y$  they happen to be integer, then this operation is an integer division operation. So this is taken as an integer division whereas if this  $y$  happens to be a real number then this is taken as a this division is taken as a real division real number division.

And many a times what will happen is that you even if  $y$  is of type integer so, you can write like  $x$  equal to  $y$  divided by 2.0 and since this 2.0 is of real is a is of type real then, this  $y$  divided by 2.0 so this whole division is done in real number division not an integer division. So, this way so, locally we may need to change the type. So, convert the type of an expression from one type to another type. And we have also have got casting of types. So, for example, I can write  $x$  equal to float of  $y$  in say C language where this  $x$  may be a floating point number, but  $y$  may be an integer. So, when I say that float  $y$  so, this  $y$  is converted locally into a floating point number and then it is assigned to  $x$ .

So, naturally the type of this right hand side expression is becoming floating point. So, if I write simply  $x$  equal to  $y$  then, if  $x$  is the  $x$  is a floating point number and  $y$  is an integer. So, this is clearly a type mismatch because that is outcome of this is often dependent on the compiler and the underlying system like how this integer and floating point numbers are represented in the underlying machine, so underlying computers, so

based on that this result will be determined and particular in particular the sizes of x and y will not be same. So, which part of x will get the value of y? So, that becomes a concern. So, if x is say 6 byte and y is 2 bytes. So, where does these 2 bytes of y go? So, that is that may not be very clearly defined ok.

But as soon as you say that this is as soon as you write it like this x equal to float of y. So, then you are very clear in your mind that I first want to convert y into a floating point number and then use it for this assignment. So, many cases if you write like this. So, compiler designer, they should give a warning. So, you might have seen this type of warnings that there is a type mismatch in this assignment whereas, if you do it like this then the compiler will know that you are really trying to do a local conversion of type. So, as a result no warning is generated. So, this comes as a responsibility of the compiler designer to check that type of individual identifiers and accordingly the type of expressions.

So, we will see how this can be done. Next so phases of a compiler. So this will be where we will see this in the particular this type conversion coming into picture and then we will draw the conclusion.

(Refer Slide Time: 10:47)

What is Type Checking

- One of the most important semantic aspects of compilation
- Allows the programmer to limit what types may be used in certain circumstances
- Assigns types to values
- Determines whether these values are used in an appropriate manner
- Simplest situation: check types of objects and report a type-error in case of a violation
- More complex: incorrect types<sup>3</sup> may be corrected (type coercing)

SWAYAM

3

So, try to answer this fundamental question like what is type checking. It is one of the most important semantic aspects of compilation. So, please note the term that this is the semantic aspect, it is not the syntactical aspect. So, you cannot use the standard automata

theory to answer whether a program is correct from there from its type point of view because of the reason that automata theory can say that this is an identifier the token types and all the tokens and all what are the different tokens that are present in the program accordingly whether the combination of those tokens make follow some grammar rule or not. So, those things can be done by your lexical analyzer and parser, but it cannot determine type of expression automatically unlike the parsing which can determine the syntactical errors very automatically. So, there is no automated way to determine this type types and all. So, this that is why it is a semantic aspect it is not a syntactical aspect.

And it allows the programmer to limit what types may be used in certain circumstances. As I was telling, that certain operators so they are applicable on certain operand types only. So, if you look into the programming language manual so, it will tell you like for different operators; so, what are the expected operand types. And then, so accordingly the programmer has to follow that, but who will check this thing? So, this has to be checked by the compiler not as part of this syntax analysis, but as part of the semantic analysis. So, this type checking process it will assign types to values. So as you are writing expressions so, it will be assigning some types to those expressions and determines whether these values are used in an appropriate manner. So if we represent types by values then whether this values are going to be used like say when I am having this an addition operation, I expect that the operands are to be of type say integer then whether we have got the integer operand or not for both the operands of this addition operation. So, that has to be checked. So, that is the in appropriate manner whether the operands have been used or not.

The simplest situation is like this, the check types of objects and report a type error in case of a violation. So, as I was telling so you check the types of individual identifiers as they are occurring and if you are finding some mistake some problem as I was telling that if I have if you write like  $x = y$  and the types of  $x$  and  $y$  are not same then this is a type mismatch. So if we have got a very strongly typed language. So, that will not allow you to have this type mismatches. So, that is an error. So, if the compiler should flash an error that this is there is a type mismatch at such and such line.

However, the more complex situation comes when you have to correct the type like see you find that there is a type mismatch, but you locally change the type of  $y$  to so, that it

matches with the type of x. So, if you do this thing then that is known as this type coercing. So, you locally change the type of some of the some part of the expression and so, that this operands so, they are having their correct types. So, this is more complex because the compiler designer need to understand like what is the expected type and all and how to do the conversion etcetera. So, do we doing it correctly is may also be a challenge like how to how to get a good type checking type coercion done.

(Refer Slide Time: 14:48)

## Static vs. Dynamic Checking

- Static Checking
  - Type checking done at compile time
  - Properties can be verified before program run
  - Can catch many common errors
  - Desirable when faster execution is important
- Dynamic Checking
  - Performed during program execution
  - Permits programmers to be less concerned with types
  - Mandatory in some situations, such as, array bounds check
  - More robust and clearer code

So, once that is next we will be looking into this static versus dynamic checking. So, type checking is important, but when do you do the type checking? So, one type of approach is to do static checking. So, type checking is done at compile time itself. So, at compile time itself all the variables and identifiers. So, they will have their types known and we can check their type at the compile time only. And in case of dynamic checking, so this is performed during program execution. So as you can understand so, this is more complex the dynamic type checking.

So, this whenever you are going for program execution, so at that time so you need to do the checking. So, the compiler designers job is more critical in case of dynamic checking because in case of static checking what the compiler designer will do is that if this is the description of some this is some program. So, it will check line by line and say at this point it has got say x equal to y. So, it will immediately know what is the type of x, what is the type of y accordingly it will take a decision; whether it will say that it is or whether

it will say that there is a type mismatch or it will do a coercion or not. So, those are secondary issues, but it is the compiler designer can do that thing at this point.

But the same thing, when you are doing in a dynamic type checking the situation is more difficult because here for  $x$  equal to  $y$  so, the this compiler for under static type checking. So, compiler designer might have generated the target code where say this part of the code was doing that assignment  $x$  equal to  $y$ . And since it is static type checking so, the compiler designer knows that if the code has been generated properly. So, this  $x$  and  $y$  they are of a similar type. So, there is no type problem. But in case of dynamic type checking, what is happening is that so here, at the when you are generating the code. So, suppose this is the code in that code so apart from that assignment of  $x$  to  $y$ . So, you have to also have the code where the type checking will be done. So, you should have some code here which will be doing the type checking.

So, that way it becomes costly at the in the dynamic checking philosophy. So, this additional code has to be there in my along with this assignment. So, that it will; here it will do the check for  $x$  and  $y$  and as defined in the programming language so, it may the program may generate an error if these types are not correct or the program may do something that the execution should do something so that it this it is taken care of. So, that particular check is done by this piece of code. So, if it finds so, after analyzing the types dynamically at the runtime so, if it finds that the  $x$  and  $y$  are of different types. So, as it is defined in the programming language it will take a it will take an action.

So, if the programming language says that this is an error so, the execution will abort. And if the if the programming language says no it is correct it can you can continue so, they may be it will locally do some conversion. So, those things will be done in this piece of code. So, you see their dynamic checking. So, it will necessitate to have additional code in the target code that is produced.

Now, then why should we go for this? So, dynamic checking; so, if it is so complex then why you why are you going for this? So, this is important because it is it may be the case that statically if we bound it then a particular function if it is called from different contexts the its variables may have different meanings. So, that will not be possible with static checking because then because the actual call is happening dynamically ok. So, that has to be taken care of by the dynamic checking only. So, this so, that is why the

dynamic checking is done in several programming languages, but of course, it makes it a bit difficult to generate code and a life of the compiler designer becomes a bit tough. Anyway so, this static checking: so, this is type checking will be done at compile time and in case of dynamic checking this checking will be performed during program execution.

So, in static checking, properties can be verified before program run. So, whatever be the type checking property so, all those checks can be incorporated. As I was telling that if some operation is being carried out whether the operands are of correct type or not then so, all those checks can be done before the program is run. However, in case of dynamic checking, so, this will not do these things. So, it will permit programmers to be less concerned with types. So, programmers can be made free like if you compare between say the languages like say C and Pascal. So if Pascal is a language where it is strongly typed. So, all variables they are having their types and all and the types cannot be changed and all, but in case of there are certain operations that that can be done on a certain types of variables only. But in cases c there are much more flexibility. So, though it is also first typed language, but it is not as strongly typed as Pascal. So, but so, that way the programmers are a bit free to have to be less concerned about the types.

So, whether it is good or bad that is a question like it may be good because programmer has got more flexibility. So, like whether I have defined all the variables there types or not so, that may not be a concern, but at the same time it may be a problem because the program may generate error and basically the unforeseen type of error. So, which the programmer might not have thought about like; it may give some error due to this type problem and it is very difficult to catch this type of errors in program run. So, static checking it can catch many common errors. So, what are the common errors like we have got some variable types, they are type there is a type mismatch and all so that way though they are very common error. So, this compiler can very easily catch those thing.

Now so, this that way the static checking can do this thing. It can do this check and all. Then dynamic checking, so this is maybe because this type of common errors like whether this x equal to y the x and y is type are matching or not. So, that is not checked that is checking it dynamically is difficult, but many times what happens is that we need to have this dynamic checking also. So, one such example is the array bounds check. So,

if we look into this for example, suppose I am writing in a in a program a line like a i equal to some expression.

Now, I have defined a to be an integer array a 1000. So, naturally if the value of the i and the I mean that this a array a is the valid indices at 0 to nine 999. So, these are the valid indices for a. Now this if this value of i is less than 0 or it is greater than 999, then what fine? Now so, in that case what is going to happen? So, statically you cannot check this because until and unless the program has run, you do not know what is the value of i. So, there is no question of statically checking whether this index is within the limit or not; that is not possible. So, dynamically we can do that because when the program is executing. So, this is my program. So, when the program is executing at this point I know what is the value of i. So, if I know the bound of this array a that its minimum index is 0 and maximum is 999. So, I can have a check like this.

So, before doing this assignment so if I for this doing this assignment if this is the code which will be doing the assignment on top of that I can add another piece of code which will be checking that if i is less than 0 or i is greater than 999 then, it is an error. So, I can call some system error function and that will flag some error message and maybe the program will get aborted. So, like that. So, that way it is done. So, this array bounds check is a very important thing because it will it is really helpful because the programmer while writing the program might not have been very careful to ensure that this array index i does not go beyond its limit.

Now if we have this type of check in the code that is generated so, the some programming language will tell you that you should have this array bounds check, some programming language will tell you that no there is no array bound check. For example, if you look into language like Pascal, you will see that this array bounds check is there whereas, if you look into language like C. So, C will not ask for this array bound check. Why it is not asking for array bound check? Of course, there is a reason because this C language so, it will allow to access a i by pointers also like you can always say like p equal to star a p equal to says suppose I have got an integer array a 1000 and there is an there is an integer pointer p. So, I can always write like this that p equal to a and then this star p plus i equal to something.

Now, this  $i$  when even if you know the value of  $i$  so, what do you do by that? So,  $\star p$  plus  $I$  means if this is the location pointed to by  $p$ . So, it will go  $I$  locations ahead and try to modify this. So, whether this modification you are doing as part of a array or independent of a array so, that is not known. So, the C language tells that  $I$  will not do any such dynamic checking because they just support this type of pointer arithmetic so they have removed this. So, whether this. So, you see that it if sometimes becomes risky because if this value of  $i$  is say, 1010 then what will happen? If this is the array  $a$  and it ranges from your 0 to 999. Now when you are saying like so you have set the  $p$  pointer here and then you are saying this  $p$  plus 1010 and then put a star over that so; that means, it will be modifying some location here whatever it may mean whatever that address may mean. So, it may be within the address space of the program, it may be outside the address space of the program. So, it will try to modify this particular memory location.

Whereas this is not possible in languages a strongly typed language like Pascal where you do not know do you do not have this type of pointer based access. So, if you even if you write say a 1 0 1 0. So, in case of C language this will be correct whereas, if you are trying language like Pascal so this will be this will not be allowed ok. So, it depends on the programming language. So, whether it is this array bounds check will be there or not so, it is dependent on the programming language.

Now, so this static checking it is desirable when faster execution is important because you see if you are doing dynamic checking as I have said that for every assignment and this operation so, you need to check the types of operands and that will make the program inherently slow. Like if I have to do say  $x$  equal to  $y$  plus  $z$ , then you need to check first whether this  $y$  and  $z$  they are of some type which supports this addition operation. So, that is the first thing to check and after you have done this addition so, you have got the expression  $E$ . Now this whether this  $x$  and  $E$  are of similar type or not so, that you can do this assignment. So, that also has to be checked and in the code that I have for this  $x$  equal to  $y$  plus  $z$ . So, if this part of the code is doing this  $x$  equal to  $y$  plus  $z$ . So, a part of it is dedicated for doing these checks.

That whereas in this for the static checking I will not have that. So, I will only have the code which will be doing this  $x$  equal to  $y$  plus  $z$ . So that part knowing the fully well that the compiler has statically checked the types of  $x$   $y$  and  $z$  and found that they are ok. So,

this way you see that this static checking. So, this it may be it is desirable because the faster it can lead to faster execution whereas, dynamic checking.

So, this is the execution will be slow, but it will be more robust and the code is clear like in a C language as I was telling that array bounds check we cannot have because it does not have this dynamic checking. So, since dynamic checking is not there so, the code becomes a bit clumsy at some point sometimes; it may behave in a different fashion which is unexpected by the programmer whereas, if the dynamic checking is there then all those problems can be resolved. So you can get a more robust and clearer code with dynamic checking.

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

**Lecture - 40**  
**Type Checking (Contd.)**

Next we will be talking about type expressions. So, type expression as the name suggests. So this is an expression of types ok.

(Refer Slide Time: 00:23)

**Type Expressions**

- Used to represent types of language constructs
- A type expression can be
  - Basic type: integer, real, char, Boolean and other atomic types that do not have internal structure. A special type, type-error is used to indicate type violations
  - Type name
  - Type constructor applied to a list of type expressions

*id, operators  
↓  
basic types*

*atomic?  
abc?*

FREE ONLINE EDUCATION  
**swayam**

So, instead of it being the type of expressions, so I will read it as expression is the involving types. So it is used to represent types of language construct. So, we will make an expression whose individual elements will be some types and they will be utilized for representing types of different programming language constructs. So, type expression , it can have basic type like any expression can have the basic ids identifiers and some operands and all. So like here also for type expression, so there can be some basic types which are similar to which are synonymous to identifiers in a normal expression.

So, if I have got a normal expression. So there are some identifiers are there and some operators are there. So in case of type expressions so these id's are basically the types themselves. So the basic types the basic types. So, they will be the id. So basic type like integer, real, character, Boolean and other atomic types that do not have internal structure. So like a programming language. So, that can define like what are the basic

types that are supported by it and they so maybe some programming language will support beat as a type. So that the basic types of those types which cannot be broken down further they are atomic you can say. So they are you cannot break an integer into further types ok. So, that is the thing. So, then if it is the case then that will be called a basic type.

So, now, what is the basic type? So, for example, what about say string. So, if I say a string. So, is it a basic type now the answer depends on the programming language like somebody may say yes string is not a basic type because I can break this string into characters. So, this a, b, c, d if it is a string. So, I have got this a, b, c, d as individual characters in it. So, I can see that there are characters inside it, but seeing that there are characters inside it does not say that this is a this is not a basic type. Why am I saying? So, like given a string. So, if there is an operator by which I can extract the individual characters and assign it to the I can access this individual characters as characters only. So, if I can do that in that case I will say that string is not a basic type. But if it is such that on string you can only do operations like screen concatenation, string copy, then string reversal and all that. So only those type of operates are allowed, but you cannot extract the individual characters for from the string. So if you cannot do that. So, in that case string also happens to be basic type.

So, that is what I want to mean is that the definition of basic type is more on this atomicity behavior. So in some if you cannot break the type further then I will call it as on as a basic type. So this integer most of the programming languages they will be using this basic type. And there is another special type which we will call type error. So, which is used to indicate type violation. So, if I have got an expression of type. So, what is its type?

(Refer Slide Time: 04:05)

The slide has a yellow header bar with the title "Type Expressions". Below the title is a bulleted list:

- Used to represent types of language constructs
- A type expression can be
  - Basic type: integer, real, char, Boolean and other atomic types that do not have internal structure. A special type, type-error is used to indicate type violations
  - Type name
  - Type constructor applied to a list of type expressions

Hand-drawn annotations on the slide include:

- A circled "int(x)" with an arrow pointing to the word "integer".
- A circled "y" with an arrow pointing to the word "unknown".
- A circled "x = y + z" with an arrow pointing to the word "type-error".

The bottom of the slide features a blue footer bar with the "swayam" logo and other icons.

So, suppose I have got say an integer. So, I say that `int x`. So, when I am trying to derive the type of this `x`. So, I am telling its type is integer fine. Now if it at some place I suppose there is a variable `y` whose type is not known, then its type is unknown and if I am trying to do an operation like `x = y + z` then what is the type of this expression `y + z`. So, `y + z` I cannot do because I do not know the type of `y` maybe `z` is of type integer, but for type `y` I do not know what is the type. So, in that case type of this whole expression will be marked as type error.

So that you can see it is not confused to be either integer or something like that, but my process can continue, my compilation process can continue. So when I am trying to compare this type of `x` with the type of this expression I see that this expression is a type error and this `x` is of type integer. So naturally this assignment cannot be done. So, there is a type mismatch. So the compiler designer can detect this situation. So, this time period is another basic type; so apart from the basic types like integer, real, character, Boolean etcetera. So we will also have type error as another basic type.

So, it can be the name of a type. So maybe we do define a new type and then that is a type name. So I define that say for many programming languages they will allow this type def.

(Refer Slide Time: 05:45)

- Used to represent types of language constructs
- A type expression can be
  - Basic type: integer, real, char, Boolean and other atomic types that do not have internal structure. A special type, type-error is used to indicate type violations
  - Type name
  - Type constructor applied to a list of type expressions

So, type def abc integer. So this means that I am defining a new type whose name is abc and whose values contained will be the value that values will be same as the type integer. So, this so, this is not this is a type name. So, I am giving in the type name abc whose type is actually which is actually integer, but I can define variable x to be of type abc and I can have another variable y of type integer. Now when I am writing x equal to y whether it is an error or not whether this is a type error or not that is a question.

So, some programming language may be very strictly typed. So, it will say yes it is a type mismatch because abc x is a type abc and y is a type integer. So, the strongly typed languages so, they will say this is a type error whereas, for the languages which are more flexible. So, it will try to look through for look further and see that they are derived from the same basic type integer. So, as a result this assignment maybe through. So, it is programming language dependent. So, that is the type name.

And type constructor applied to a list of type expressions. So, list of type expressions like I can say many programming languages like say you can have this structure definition. So, structure abc and there you can have different types so, integer some field maybe they are integer a, integer b, characters c. So, like that we are defining as types. So, this so, this is a constructor. So, a list of types. So, it is collection of one integer, two integer and one character. So, integer, integer and characters they are combined together to it is a list of type expressions are taken together to get this constructor structure.

Similarly, I can have say integer a 100. So, when I am saying this then I am telling that a 100 is a new type where which is our array of integers. So, array of integers this whole thing is a new type like. So this actually you should read it like this, this expression though when most of the programming languages we read it like this, but in reality it is like this integer 100 and that is a, as if this integer 100 is the type. So, array of 100 integers taken together and I am defining a variable a of this time. So, you should read it in this fashion, though for the sake of our understanding we write in this form, but it is like this.

So, similarly if I have got something like this. So say integer a 100 and b 50. Now you see that can I write that a equal to b. I cannot because there is a type mismatch here what is the type mismatch. So, actually this a is of type integer 100. So, this is the type of a and the type of b is actually integer 50. So, it is b is a variable which is of type which is collection of 50 integers and as a variable of type which is a collection of 100 integers. So, this is also a constructor. So, this is an array constructor we can say. So, this is constructing an array of 100 integers this is constructing a type of array of 50 integers and this a and b so, they are variables of these individual types. So, we cannot do the assignment. So, the it will be stopping as at type error.

Now, some pro so, if they are same type like if this is also say 100 instead of that. So, this is also 100 then in that case a and b both are of same type. So, this is also 100 of both of them are of type of integer 100. So, in that case a equal to b is a feasible operation. So, many programming languages do allow this array assignment directly. So, it will be doing element to element allocation. So, element to element copy, but it is allowed in the some programming languages so, but this type has to be same. So, this way this type becomes very critical. So, it is the just by looking into the program, so, we cannot say that this is they are there this types are correct or not. So, it is very much defined by the underlying programming language that semantics like whatever is given there.

(Refer Slide Time: 10:51)

Type Expressions

- Arrays are specified as  $\text{array}(I, T)$ , where  $T$  is a type and  $I$  is an integer or a range of integers. For example, C declaration "int a[100]" identifies type of  $a$  to be array(100, integer)
- If  $T_1$  and  $T_2$  are type expressions,  $T_1 \times T_2$  represents "anonymous records". For example, an argument list passed to a function with first argument integer and second real, has type integer  $\times$  real

$a = \text{array}(100)$   
 $T = \text{array}$

$\text{int func}(\text{int } m, \text{float } n)$   
 $T = \text{int } \times \text{float}$

$T' = T_1 \times T_2$   
 $\text{int float}$

So, next we will be looking into this thing that. So, arrays are specified as array I, T where T is the type and I is an integer or a range of integer. For example, so, this is basically said that is what I was telling that integer a 100 is nothing, but integer a 100 is nothing, but is a type of a is array 100 integer. So, it is written like this array it is a constructor array and hundreds of integers are taken together to give this thing.

Now, we have got this thing that if  $T_1$  and  $T_2$  they are two type expressions then this  $T_1$  cross  $T_2$  it represents anonymous records. So, for example, when you are passing an argument list to be a function the first argument integer a second real has type integer cross real. So, what I mean is suppose I have got a program and at this point I am calling the function say function 1 and there I am passing two values a and b and here in the function the function is like this. So, here int m and say float n and there I have written the function. Now I need to check that when I am calling this function func 1 from this point. So, the parameters that I am passing is really matching with the argument, the arguments that I am passing here is really matching with the parameters that I have here in the function. Now how do I do this thing?

So, actually the way we are doing is we are taking these a matching if this a with the first argument first parameter matching the type of b with the second parameter like that, but formally speaking. So, what are we doing? So, the, if I say that the for this is a type expression. So, then this type expression is given by of  $T_1$  cross  $T_2$  where  $T_1$  is the type

of the first parameter and the first argument and T2 is the type of the second argument as a result this gives me a new type which is the cross product of these two types T1 and T2 and then and then what is the type of this. So, this is nothing, but a type T dash which is again T1 cross T2 and then this T1 and T2. So, this is this is integer and this is float.

So, T dash that we have here is a type integer cross float. So, here also this T should must be integer cross float then only I can say that these two types are matching otherwise not. So, this way formerly this parameter passing technique that we have that is also we have to check for the arguments and we do this thing. So, some programming languages where it is where you have to do this type checking. So, they will do it like this some programming languages will say no I am free. So, you can pass any arbitrary arguments. So, that is that is possible. So, then this checking is not done. So, this type construction is also not done, but anyway. So, for most of the programming languages, so it will do this type checking and this will be required.

(Refer Slide Time: 14:31)

**Type Expressions**

- Named records are products with named elements. For a record structure with two named fields – length (an integer) and word (of type array(10, char)), the record is of type  
 $\underline{\text{record}((\text{length} \times \text{integer}) \times (\text{word} \times \text{array}(10, \text{character})))}$
- If T is a type expression, pointer(T) is also a type expression, representing objects that are pointers to objects of type T
- Function maps a collection of types to another, represented by  $D \rightarrow R$ , where D is the domain and R is the range of the function

*Handwritten notes:*

Diagram illustrating a named record:

```

graph LR
    Record["record { length: int, word: char[10] }"]
    Record --> Length["length: int"]
    Record --> Word["word: char[10]"]

```

The notes show two boxes labeled "record" with curly braces. The left box contains "length: int" and "char[10]". The right box contains "word: char[10]". There is a double-headed arrow between the two boxes, indicating a relationship or mapping between them.

Next we will see like the named record. So, how does it record will look like. So, named record so, they are products with named elements, for a record structure with two for named fields length and word the record is of type length into integer length cross integer cross word cross array of 10 character. So, this is the this actually telling me that I have got a record say for example, if you look into the c programming language. So, this is define a structure where the first field is first field is it has got a name length and this is

an integer. So, integer length and the second field is word which is of array 10. So, it is like character word 10. So, this is the structure.

Now what is the type of this structure. So, that is what we try to figure out. So, its type is depicted by this particular type expression. So, it is length cross integer. So, length is of type length is of type integer. So, length into integer because length is a name of the field and integer is its corresponding type and this is word is the length name of the field and it is an array of 10 characters so this product. Now what it means is that this if I have got another structures say structure so, integer instead of length so, I call it say len and instead of word I call it say wd.

Now, whether these two structures are these two types that same or not. So, this is also a type def, this is also a type def in from c language. So, they both of them are defining types. So, are these two types same or not. Now as far as this definition is concerned this types are not same because here it says that it should be length cross some integer value, but here it is len cross some integer value. So, they are not same. Similarly this is word cross array 10 character, but here it is wd cross array 10 cross characters. So, 10 character. So, naturally so, they are not equivalent if I go by this thing. But of course, if you go further so, you can see they can say that this len and length they are actually equivalent because they are both of them are integers and then this wd and word so, they are also equivalent because both of them are character is obtained entries.

So, that way I can say that they are equivalent, but it requires more work ok. So, the either type checker has to do this type of checking then the work involved is more? So in most of the cases what happens is that we stop at this level itself telling that these two types are not equivalent ok. So, we will come to this when you go to the type checking in more detail. So, this is the named record how are they how they are types are derived so, that is mentioned here. And also you can have a pointer. So, pointer the if T is a type expression, then pointer T is also a type expression that represents objects that are pointers to objects of type T.

(Refer Slide Time: 18:25)

Type Expressions

- Named records are products with named elements. For a record structure with two named fields – length (an integer) and word (of type array(10, char)), the record is of type  
 $\text{record}((\text{length} \times \text{integer}) \times (\text{word} \times \text{array}(10, \text{character})))$
- If T is a type expression,  $\text{pointer}(T)$  is also a type expression, representing objects that are pointers to objects of type T
- Function maps a collection of types to another, represented by  $D \rightarrow R$ , where D is the domain and R is the range of the function.  
Handwritten note:  $(\text{integer} \times \text{character} \times \text{float}) \rightarrow \text{integer}$  (with arrows from integer, character, float to integer)  
 $f(\text{int } x, \text{char } y, \text{float } z)$   
return m;

So, this pointer T is also this is also a type expression whose typing. So, this will represent objects that are pointer to objects or type T and a function maps a collection of types to another representing D to R where D is the domain and R is the range of the function.

So, for example, if I have got a c function like integer say f 1 is the name of the function and it takes parameters like say integer x, character y, float z then it. So, finally, returns some into your value as the type of the f 1 is integer. So, it returns some integer say m fine then D is the domain. So domain says that from which set it can pick up the values. So, it can pick up the values. So, from that domain which is integer, integer cross character cross integer cross float. So, this is the domain because it can take up values from this domain. So, in the how is this domain? So, these domain is collection of three entries where the first entry is an integer, second entry is a character and a third entry is a float ok. So, this is D and this range is the output of the function the output type is integer. So, you can say that given three values of this type one is the one integer one character and one float the or the rather the first one integers, second one character, third one float is ordering really matters. So, it will be outputting an integer.

So, this is the type of the function. So, if I tell what is the type of the function so, type of the function is this thing. So, it takes parameters integer will first parameter integers, second parameter character, third parameter float and it outputs one integer value. So,

that way so, function maps a collection of types to another represented by D to R where D is the domain and R is the range of the function. So, we can we can represent these functions like that.

(Refer Slide Time: 21:07)

- Type expression “integer  $\times$  integer  $\rightarrow$  character” represents a function that takes two integers as arguments and returns a character value
- Type expression “integer  $\rightarrow$  (real  $\rightarrow$  character)” represents a function that takes an integer as an argument and returns another function which maps a real number to a character

$r = \lambda x. f$

$r = \lambda x. f2$

$f2(x) = \lambda y. g2$

return  $f2(\text{real } x)$

Now, once we do this then the next one. So, for; so, this is another example like the integer cross integer to character. So, this type expression it will represents a function that takes two integers as arguments and returns a character value. So, that is one possibility then the type expression integer to real to character. So, this represents a function that takes an integer as an argument and returns another function which maps a real number two a character. So, this is like this that so, either if I look into this one more carefully. So, this is basically that function pointer type of concept that you have in many programming languages like c.

So, what the function does is the function 1 so, it takes an integer value integer x and depending on the value of x it will return some other function return some other function f 2 so, where which will be taking a real number and it will be returning the characters. So, this returns the so, it will be; so, it will return the function f2 such that it takes some real number say it will take some real number and this f 2 will be. So, this is often written like this to tell that f2 is a function pointer. So, like that. So, this is a point function pointer. So, it depends the syntax depends on the programming language that we are looking into, but essentially what I mean is that given the value of x it will return

some function maybe if  $x$  equal to 1 it will return the function  $f_1$ , if  $x$  equal to 2 it will return the function  $f_2$ . So, like that as a return value it does not return a single value, but it returns another function and that function that it is returning is of type like this. So, this function  $f_1$  given a real value it will produce a character or the function  $f_2$  given a real value it will produce a character.

So, this way this function pointers can also be taken care of in the type expression. So, this of course, makes it bit complex, but it can be taken care of there.

(Refer Slide Time: 23:32)

The slide has a yellow header with the title 'Type Systems'. Below the title is a bulleted list:

- Type system of a language is a collection of rules depicting the type expression assignments to program objects
- Usually done with syntax directed definition
- 'Type checker' is an implementation of a type system

At the bottom of the slide, there is a blue footer bar featuring the Indian Space Research Organisation logo, the text 'SWAYAM' with 'SWAYAM' in large letters and 'India's Online Education Platform' below it, and a circular emblem. On the right side of the footer, there is a video player interface showing a man speaking, with a progress bar and a timestamp '9'.

Now, what is a type system? So, type system of a language is a collection of rules depicting the type expression assignments to program objects. So, so, if you look into the programming language manual then apart from giving the grammar that has got how this individual constructs of the grammar will be will be there how are they organized, what is the grammar for that. So, will also tell you like how what are the types which types are allowed and all so, it will give you all those details.

So, that gives rise to a type system. So, type system it is a collection of rules that will depict the type expression assignment to assignments to program objects and usually done with syntax directed definition. So, this type checking and type conversion. So, they are often done in the syntax directed translation phase itself then we will see how this can be done and type checker is an implementation of the type system. So, as a compiler designer so, we also have to include a type checker. So, it is not only that we

generate code or do some syntax directed translation to do something else. So, that is there, but apart from that.

So, you also need to have some type checking rules and those type checking rules are also to be made part of this syntax directed translation phase. So, that will be known as type checker.

So, this is more difficult if the language is strongly typed them this type checker will be more complex the language is relatively simple then the type checker will also be a bit simple.

(Refer Slide Time: 25:12)

Strongly Typed Language

- Compiler can verify that the program will execute without any type errors
- All checks are made statically
- Also called a sound type system
- Completely eliminates necessity of dynamic type checking
- Most programming languages are weakly typed
- Strongly typed languages put lot of restrictions
- There are cases in which a type error can be caught dynamically only
- Many languages also allow the user to override the system

So, what is a strongly typed language? So, strongly typed language, so, compiler can verify that the problem will execute without any time errors. So, this is the most important like the compiler. So, if the strongly typed language. So, if the compiler says yes the program is correct syntactically and semantically correct then, so, it will be it is telling that the program will not give those type of bugs, those type of problems while executing it will not come up with some type error.

So, it is not that there is an assignment like  $x = y$  and while executing so, it will be coming up with a statement that there is a type mismatch and that is why there is a there is an error in the program of course, the logical bugs maybe there in the program that cannot be captured by any compiler, but so, type related issues are there so that can be

detected by the compiler itself and once the compiler certifies that the program is correct from the type point of view. So, there cannot be any problem during execution.

All checks are made statically. So, they are before the program is run and before the code is generated all the checks are done and this is also called a sound type system because, because of the strongly typed, so everything is known everything is fixed. So, this is called a sound type system. So, it completely eliminates the necessity of dynamic type checking. So, of course, so, that dynamic type checking will not be necessary because everything will be checked statically of course, at array bounds check that we have talked about.

So, it cannot be done at the compilation time or the statically, but if that array bounds check is not there then of course, I do not need to do it dynamically. So, if the language does not require this array bounds check then h I can say that it will be it does not require an dynamic type checking.

Most programming languages are weekly typed, they are they are not strongly typed because strongly typed means for every small things. So, we have to you have to have this type things taken care of. So, the programmers will also find that many things are too trivial and this too trivial things are also not taken care of like that may be the situation; so, like so, that creates a difficulty. So, that is why most programming languages they are weekly type and this is strong strongly typed languages put a lot of restrictions on the programmer.

So, we will see that if there are a lot of restrictions so the strongly typed language so it will it will not allow you to do arbitrary assignment and operations. So, as a result it is like that for example, so, if I have got say this  $x = y + z$  then this  $y$  is of type float and  $z$  is of type integer. So, strongly typed language you will not allow you to do this addition, but you see that we can do this addition because we can always take it as a floating point addition and do the addition.

So, if it is a strongly typed language then everything has to be everything has to be very crucial. Another very subtle example maybe like this, suppose I want to write like  $x = y / 2$  where  $x$  and  $y$  both of them are of type float, but while I am writing like this  $y / 2$  this  $2$  is it  $2$  is an integer and this why is it floating point number. So, you see now there is a type mismatch because the two operands involved in the division

operation they are not have same type. So, it requires that the programmer writes it as y by 2.0 not as 2. So, that these type of funny things can come up. So, that is why if you make it very strongly typed. So, it can give rise to many problems for the programmer also. So, this strongly typed languages they are; they are not this type checking does not is not made so strong. So, there are cases where this type error can be caught dynamically only.

So this is a typical situation is like that array bounds check that I was talking about and many languages also allow the users to override the system. So like at the type casting type caution. So those are there. So we will see that in subsequent classes. So there we can override this type system and say the programmer says that no these type assignment is correct. For example, in c language so if I have got this thing so, y equal to say this x and x and y are of two different types then says maybe x is say integer and y is say float.

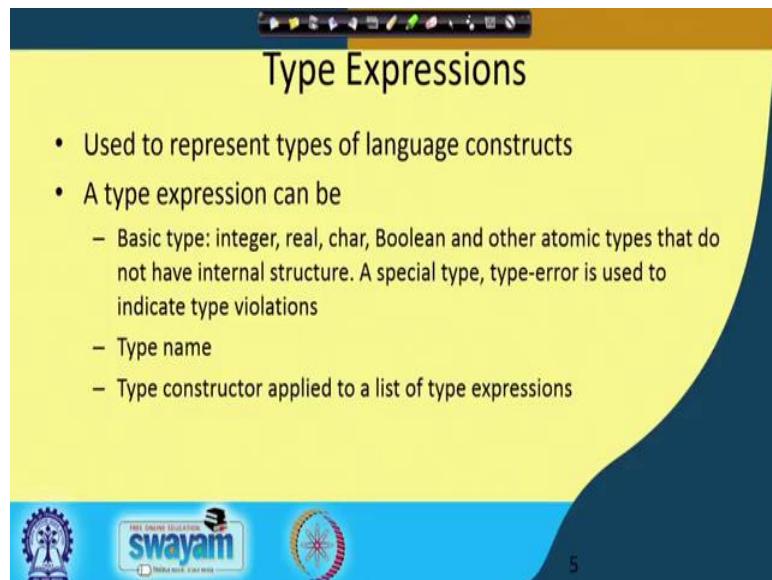
So we often write like y equal to int x ok. So that will be overriding the rule that y the integer cannot get the value of a float. So if you do this; that means, is x is converted to integer and then only it is assign. So this type of overriding may be possible in the strongly typed language whether this many programming languages they will bypass this strongly typed concepts by having these extra things in the language itself.

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

**Lecture – 41**  
**Type Checking (Contd.)**

Type expressions: So, for all programming language constructs and the programming language programming elements so, will have some types. So, we can understand that when we have got some variables so, they have got a type.

(Refer Slide Time: 00:28)



**Type Expressions**

- Used to represent types of language constructs
- A type expression can be
  - Basic type: integer, real, char, Boolean and other atomic types that do not have internal structure. A special type, type-error is used to indicate type violations
  - Type name
  - Type constructor applied to a list of type expressions

When we are talking about some say expression, the expressions have got some type, but what can happen is that these constructs programming language constructs so, they will also have some type. For example, some statement also can have a type a typical example maybe the statement may be taken as correct or the syntactically correct or syntactically incorrect or maybe syntactically it is correct, but in the expression in the when you go to this components of it there is some type error. So, for the whole pas construct so, we can say that there is a type error. So, that when we are pointing out some type mismatch so, we can tell that at such and such statement, so, there is a type error.

So, they are used for type expressions they are used to represent types of language constructs. A type expression can be basic type; basic type like integer, real, character, Boolean and other atomic types. Atomic type means they cannot be divided further. So,

like an integer type so, if you cannot divide it into further, until and unless now say if we say they say that there is a bit type and based on that bit the integer is framed based on the bit. So, if that is the case then that so, in that case integer also can become a derived type. Particularly, if you are looking into the hardware description languages, so, there we have got this bits as the basic type and from there we can construct integers of different bits.

So, similarly this real character Boolean so, they are all atomic types and they do not have further internal structure. So, they are the atomic type. And, there is a special type called type error so, for indicating type violation. So, whenever there is a type violation so, associated with the color correspond with the construct. So, we will put that it is type is of type error so that will can point out the type errors. Then, we can have a type name that is we can define a new type and give it a name and we can have some type constructor. So, now we will see that like array is a constructor, then your record is a constructor, so, how can we construct types from the basic types. So, that will be the constructed thing.

(Refer Slide Time: 02:46)

**Type Expressions**

- Arrays are specified as  $\text{array}(I, T)$ , where  $T$  is a type and  $I$  is an integer or a range of integers. For example, C declaration "int a[100]" identifies type of  $a$  to be  $\text{array}(100, \text{integer})$
- If  $T_1$  and  $T_2$  are type expressions,  $T_1 \times T_2$  represents "anonymous records". For example, an argument list passed to a function with first argument integer and second real, has type  $\text{integer} \times \text{real}$

Diagram illustrating the construction of a type expression:

$$I \rightarrow T_1 \times T_2$$

Annotations:

- $T_1$  and  $T_2$  are shown with arrows pointing to the  $T_1$  and  $T_2$  in the expression  $T_1 \times T_2$ .
- A handwritten note  $\text{array}(100, \text{integer})$  is written above the first bullet point.
- A handwritten note  $\text{int}[1..20] \quad \text{float}[-5..10]$  is written below the first bullet point.

**Swayam**

So, we look into some examples of this constructed types; first of all the arrays. So, array is a constructed type, like say array  $I, T$ ; where  $T$  is a type. It may be a basic type or may be a constructed type or derived type and  $I$  is an integer or a range of integers. Like say in C declaration we have got say this int a 100. So, the in this int a 100 so, it identifies

the an array a to be of size has to be of type 100 integer. So, in as I said that enough it can be a range of integers also for this is particularly true in some other programming languages like C language you will not find it particularly hardware description languages.

So, we have got this type of constructs like you can define an array a and you can tell the range. So, one was say 1 dot 20 so, it is saying that the lines the indices will run from 1 to 20. So, that way it is not uncommon it is not impossible to have say minus 5 dot say 10. So, this is also an array. So, we have got that into the indices will run from minus 5 to 10. So, these are all these are all possible like this range that is why it is said to be a range. So, in this case the range is range is obvious. So, it is 0 to 99, ok. It is 0 to 99, but in some cases so, it is not so obvious. So, like here it is you have to be more explicit like minus 5 to 10 so, whereas, here the range is implicit in nature. So, that is one type of constructor that is collection of basic collection of some already defined type so, elements of that type.

Then, we can have some records like say T 1 and T 2, if they are 2 type expression then T 1 cross T 2 represents anonymous records. So, if you have got any variable or symbol of that particular type T 1 dot T 2. So, like say u is a variable whose type is say T 1 cross T 2, that will mean that I will in you there will be 2 components; one is u 1 and another is u 2, ok. So, there will be u 1 and u 2 where this u 1 will be of type T 1 and u 2 will be of type T 2. So, this is of type T 1 and u 2 is of type T 2.

So, that way we can have some record type. So, is there so, this is very close like if I have got say one argument as integer and another argument as real when you are calling a function, when we are passing parameters then this argument list passed to a function with first argument integer and second real it will have the type integer cross real. So, that way we can have this type expression. So, this is unnamed because or anonymous record because it does not have a name. So, sometimes we associate some name also with this type of type this.

So, in this case so, if we have a name lab for example, the structure construct in C language so, where you have got the structures they have got some name. So, that name will get associated with the type.

(Refer Slide Time: 06:30)

- Named records are products with named elements. For a record structure with two named fields – length (an integer) and word (of type array(10, char)), the record is of type $\text{record}((\text{length} \times \text{integer}) \times (\text{word} \times \text{array}(10, \text{character})))$
- If T is a type expression, pointer(T) is also a type expression, representing objects that are pointers to objects of type T
- Function maps a collection of types to another, represented by  $D \rightarrow R$ , where D is the domain and R is the range of the function.

So, so, apart from that so, we have got say at named records. So, that our previously we have seen unnamed records so, we can have named records. So, these are these are products of products with named elements. So, for a record structure with 2 name fields, length and word; while length is an integer and word is an array of 10 characters. Then the record type is it is a it is type is record and it is length cross integer, so, that gives the names of the individual fields of the record and then they are corresponding types are mentioned the word and array10 character. So, that we the length is of type word and integer is an array of 10 characters.

So, it has got so, these are go so, this length is length is of type integer and what is an array of character 10. So, here this length has got that the that the first field has got the name length, second field has the got a name word and also their corresponding types are mentioned. So, this is one type of named record that can for which you can have type expression. Then, if T is a type expression, then pointer T so, is also a type expression. So, representing objects that are pointers to the objects of type T. So, this is quite common like in almost all the programming languages that supports pointers will have some way to specify this. So, here also as a compiler designer so, we have here to support this pointer type and this point at T is a type expression that will represent objects of pointers to objects of type T.

And, in case of function it maps a collection of types to one collection of types to another. So, it is a D to R where D is the domain and R is the range of the function. So, we can have something like this like if I have got a function say integer say integer f 1 and it has got arguments like say integer x and real y integer x and real y. So, here the domain that we have is basically this arguments they are belonging to this cross type integer and real. So, this is integer cross real and then that is that is the D actually. So, the integer cross real. So, this represents the D and the result is of type integer. So, this int, so, this is that R. So, R is the range of the function.

So, domain of the function means the sets from which the function can get inputs and range of the function means the set to which the function can produce output. So, we have we can have this type of function mapping, ok. So, that also so, this function f 1 is of type this will map collection of types to another and represented by D to R.

(Refer Slide Time: 10:02)

- Type expression “integer × integer → character” represents a function that takes two integers as arguments and returns a character value
- Type expression “integer → (real → character)” represents a function that takes an integer as an argument and returns another function which maps a real number to a character

So, next we will see how this say how this integer cross integer to character. So, this is a function this is an example of a function that takes 2 integers as arguments and returns a character value. So, as I was telling in the previous example that f 1 so, it was integer cross integer cross real to integer. So, here the integer cross integer to character. So, it takes 2 integers as arguments and returns a character value.

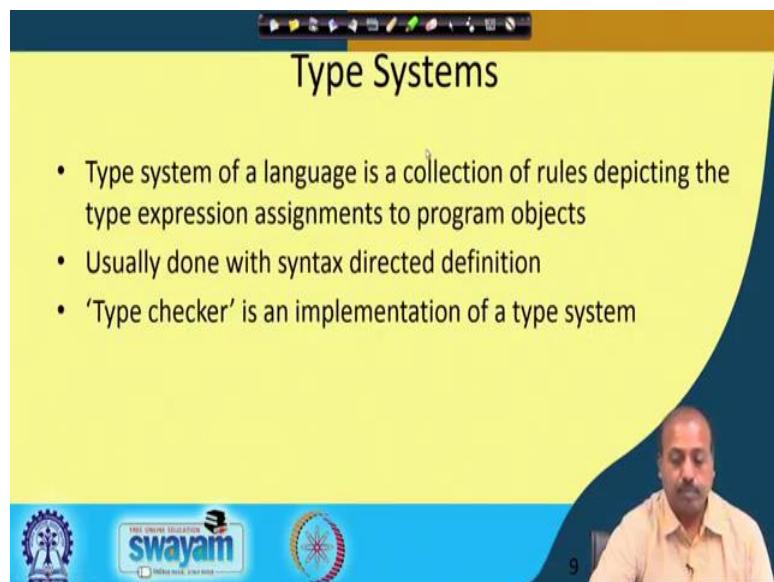
Then we can have type expression integer to real to character. So, this is basically some sort of function pointer. So, that returns a function. So, many programming languages

support this type of constructs like you can a function it can return not only some basic type basic value or these derived types, but it can also return another function as the return value. So, that function will be called when that fact that particular function will be called based on that.

So, this integer to real to characters; so, this so, this function takes integer as an integer value as an argument it takes an integer value as an argument and as a return type it returns this real to character type. So, as this thing as you can understand that if this is that domain so, it takes values from D and this whole thing is the range so, it maps this D to R. But, what is this R? This R is again real to character. So, this is again a function. So, this is also for this function the domain is real and range is character. So, it is mapping from D to R. So, that way you can say that this return type is also a function. So, from domain D integer, so, it is returning a function which can take a real argument and return a character value as the answer.

So, this way we can have these function pointers also associate with that certain types.

(Refer Slide Time: 12:10)



So, type system: so, this is it is a collection of rules for any programming language manual if you look into it will tell you like what are the rules of that particular language type rules of that particular language. So, this type system of a language is the collection of rules that depicts a type expression assignment to program objects like what are the

how are these objects are assigned for what will be the type rules. So, they will be collected in the type system.

And, it is usually done with a syntax directed definition. So, it is done with syntax directed definition means that you can have this type check and all and they will be implemented by means of this syntax directed transmission scheme because syntax directed translation scheme, so, it will be the parse tree that is generated by the parser it will have these individual symbols. So, you can get for the individual symbols they are basic types and those types are being converted into some derived type again by some grammar rules so that while those grammar rules are applied so, you can check whether the corresponding type is constructed properly or not. And, if it is not then it will be a type error otherwise we can derive the type of the newly constructed portions.

And, type checker this is an implementation of a type system. So, we have got type system that is specified by the language and type checker is implemented by the compiler designer. So, type checker it will be using the syntax directed translation mechanism and then it will be accordingly it will be producing some actions while doing the reductions in the programming language in the in the grammar rules while doing in the parser while it is doing the reductions. So, it can check for those rules the language.

(Refer Slide Time: 14:05)

## Strongly Typed Language

- Compiler can verify that the program will execute without any type errors
- All checks are made statically
- Also called a sound type system
- Completely eliminates necessity of dynamic type checking
- Most programming languages are weakly typed
- Strongly typed languages put lot of restrictions
- There are cases in which a type error can be caught dynamically only
- Many languages also allow the user to override the system

Now, if you look into the languages they can broadly be divided into 2 classes. One class is called strongly typed language where all the variables that you have all the

functions all the identifiers that you have in the in the program so, their types are well defined. So, either they are they are explicitly defined or they are implicitly defined. So, why do I say so, like in most of the languages there like say C, so, there for every variable you have to have a corresponding definition whereas, certain programming language like for example, Fortran where the first character if it is a vowel in that case it will mean that this is that this is going to be an integer. So, that way there are differences, but whatever it is there are some implicit rules and there may be some explicit definitions.

So, this compiler can verify that the program will execute without any type errors. So, while doing the parsing the entire program is known. So, it is looking into the entire program and if the compiler does not give any type error then it is guaranteed that during execution of the program there will not be any type error. So, this checks are made statically. So, they are static checks and they are also called a sound type system because this type system the type checker that we have that is very rigorous in nature. So, they are called a sound type system.

Completely eliminates the necessity of dynamic type checking because there is no nothing like no scope is left for this dynamic checking. So, it will eliminate the necessity of this dynamic type checking and most programming languages are weakly typed. So, why it is so because many times it is not possible to have everything done statically. For example, this array bounds check ok, then so, we cannot have those things that they are there need to be done statically.

So, most programming languages they are weakly type and strong because this strongly typed languages they will put a lot of restrictions on the programmers. Like whenever you are defining a function, so, you should have its type specified. Even for the language say C, so, if you define a function and do not if the function definition comes later so, if you have a program where at this point I have defined the function and say f 1; so, if I have defined the function f 1 at this point and then so, here we have got the function if 1 and somewhere earlier I am writing that code for you know I am calling the function f 1 say x equal to f 1 something like this.

Then at this point of time the compiler has not seen the type of f 1. So, it becomes difficult because what is the type of f 1. So, whether this statement is correct from the

type point of view that will depend on the interpretation of f 1, but in C language you know that it implicitly assumes that f 1 is of type integer. So, that way so, this does not put this put this requirement that f 1 has to be defined previously and things like that. So, there is this type of this type of flexibilities may be there as a result we can have this weakly typed constructs. So, weekly type things so, it will it will make the life of the programmer easy, but at the same time it also gives rise to the problem that sometimes the program may generate some error while executing actually in actual.

So, strongly typed languages they have got put a lot of restrictions and there are cases in which a type error can be caught dynamically only. As I was telling that say some array bound increasing beyond the limit; so, there those type of checks cannot be done statically. So, they are to be done dynamically only and similarly when you are say calling a function pointer. So, when it is returns returning a function. So, whether that function is of proper type or not, so, that can be checked only dynamically because it will be decided at that point only whether the function that has been returned is returning a proper value, ok, it is returning a proper value of correct type for assignment to some other variable. So, that way it can it can be checked dynamically only.

And, many languages will allow users to override the system like; this type overriding, like type coercion and all where you can have a typecasting ok. So, for example, C language, so, that allows to this typecasting type of facility. So, it can allow you to override the system the type system.

So, these are the some of the concerns for the strongly typed language which we not only it may or it may not be followed very rigorously because of these problems.

(Refer Slide Time: 19:30)

Type Checking of Expressions

- Use synthesized attribute 'type' for the nonterminal E representing an expression

Expression	Action
$E \rightarrow id$	$E.type \leftarrow lookup(id.entry)$
$E \rightarrow E_1 op E_2$	$E.type \leftarrow$ if $E_1.type = E_2.type$ then $E_1.type$ else type-error
$E \rightarrow E_1 rlop E_2$	$E.type \leftarrow$ if $E_1.type = E_2.type$ then boolean else type-error
$E \rightarrow E_1[E_2]$	$E.type \leftarrow$ if $E_2.type = integer$ and $E_1.type = array(s,t)$ then t else type-error
$E \rightarrow E_1 \uparrow$	$E.type \leftarrow$ if $E_1.type = pointer(t)$ then t else type-error

Now, how do we do type checking for expressions as I was telling that it will be using some syntax directed translation mechanism for doing this type check. So, this so, it is so, we assume that with the non-terminals and terminals so, we will have some type, ok. So, with there is an I attribute type which will be used for representing the type of a of an expression E. So, this E producing id. So, this E dot type the corresponding action so, when this particular reduction will be done, so, what we do in the simple in the symbol table will look up for this id dot entry. So, it will be finding the entry in the symbol table and once it finds it, so, the corresponding type is also known. So, this expressions type is made equal to type of that particular entry.

Then, we have we have got a rule reduction like this E producing E 1 operator E 2. So, this operator may be any arithmetic or logic operator, ok. So, this E 1 of E 2, so, this E dot type is if E 1 dot type and E 2 dot type are same then it is E 1 dot type else it is type error. Now, this gives rise to many issues like we are so, it is expecting that the type of these 2 operands of this particular operator they are same. Of course, we have not checked whether this particular operator is applicable on these operands or not. So, that check can also be introduced.

So, if we do all those things then; so, if for example, if I am talking about say addition and this E 1, E 2 they appear to be say string variable, then this addition does not have any meaning in them. So, though E 1 dot type and E 2 type both are string so, we so, in

that case also the type of E should be type error. Of course, it is not captured in the rule that we have specified here. So, here it is simply checking that types of E 1 and E 2 and if there are types are same then it is said to be equal to E is type is said to be equal to E 1 dot type otherwise it is said to be a type error.

Similarly E 1 a relational operator E 2. So, say some x greater than y this type of expressions. So, here also this E 1 dot type and E 2 are type they must be same. However, if they are same after this relational operation the return type will be a Boolean variable. So, then in that case it returns a Boolean otherwise it is a type error. So, this way I can have this E 1 dot type equal to E 2 dot type then it will then it will have this Boolean type a Boolean type otherwise it is a type error.

Then this is this is for the array. So, E producing E 1 within bracket E 2. So, this E 1 and E 1 is supposed to be some name it is supposed to be of type array and this E 2 is some index, ok. So, then it is the rule that we have is if E 2 dot type is integer, so, if this is this is supposed to be an index. So, this in this must be the it is type must be an integer. So, if E 2 type is integer and E 1's type is array S T, ok. So, if E 1's type is array S T, so, this is an array and then S is the size and T is the type of individual elements in the array S T declaration. Then this in that case T will be in that case this x type of this E will be T otherwise this is a type error. So, this way we can derive the types of array elements and in an expression.

Then this is E 1 pointer. So, if I if I equal to E 1 pointer, so, in that case E dot type is so, it is assigned to be T if E 1's type is a point at to T. So, T is the basic element and if it is E 1's type is pointer to T then E's type will be T otherwise this is a type error. So, this is just a simple example of this type rules, but you can think about more rigorous rules as I was talking about. So, whether these operators are applicable, whether this arrays that we have, so, they are giving rise to further indices or not, so, multi dimensional arrays and all. So, they are not taken care of in this particular expression.

(Refer Slide Time: 24:15)

## Type Checking of Statements

- Statements normally do not have any value, hence of type void
- For propagating type error occurring in some statement nested deep inside a block, a set of rules needed

$S \rightarrow id = E$	$S.type \leftarrow$ if $id.type = E.type$ then void else type-error
$S \rightarrow \text{if } E \text{ then } S_1$	$S.type \leftarrow$ if $E.type = \text{boolean}$ then $S_1.type$ else type-error
$S \rightarrow \text{while } E \text{ do } S_1$	$S.type \leftarrow$ if $E.type = \text{boolean}$ then $S_1.type$ else type-error
$S \rightarrow S_1; S_2$	$S.type \leftarrow$ if $S_1.type = \text{void}$ and $S_2.type = \text{void}$ then void else type-error

Then for the statements: so, I said in some previously that statements also can have some type, ok. So, statements normally do not have any value hence their type is void. However, so, if there is a type error in statement which is nested deep inside a block a set of rules are needed because if there is a type error in inside a block then there is no point trying to generate code for the remaining part of the program. So, that entire block can be marked that there is a type error.

So, this is what is done like say id is producing id equal to E, so, this id type if ids type is equal to E S type in that case this is type of S is void otherwise it is a type error. So, if there is a type mismatch between id and E then there is a type error and that type error is assigned to S, so that at a higher level we can understand that there is there was a type error that has occurred with this statement. So, we can take some appropriate action. So, it may be aborting the aborting the code generation or whatever, but we can do that.

Similarly, this one so, if E then S 1. So, this S dot type so, if it is type is equal to if a dot type is Boolean then it is S 1 type else type error. So, in this so, the more many programming languages they will require that this is E's type be Boolean some other language. So, it may not require so, like for example, in C language it is E dot type need not be Boolean it has to be some it may be any integer or real any other type, but. So, here it is taken as E dot type is Boolean. Some other programming language they will require that a the expression in the if statement to be Boolean type. So, it will check that

if a dot type is Boolean then S type is equal to S 1 type. So, if the S 1 type is void then S type will also be void; if S 1 type is a type error then S type will also be a type error, but if it is not Boolean in that case so, this S type is said to be equal to type error.

Then this while statement this is also similar if E dot type equal to Boolean then S 1 dot type else type error. So, here also it is same. And, then this concatenation of statements S 1 semicolon S 2 so, occurrence of successive occurrence of 2 statements S 1 and S 2. So, if S 1 type is void and S 2 type is void then type of S will also be void otherwise type of S is type error. So, if there is an error in S 1 or S 2 then there types are not void. So, in that case it will be a type error.

(Refer Slide Time: 27:22)

Type Checking of Functions

- A function call is equivalent to the application of one expression to another

$$E \rightarrow E_1(E_2) \mid E.\text{type} \leftarrow \text{if } E_2.\text{type} = s \text{ and } E_1.\text{type} = s \rightarrow t \text{ then } t \text{ else type-error}$$

So, if type taking of functions sso, if we have got something like this like E producing E 1 within bracket E 2. So, this is E 1 is the it is a function call. So, E 1 is expected to be some id or so. And, then this E 2; so, this E 2 is expected to be the parameters that we are passing.

So, so, this is like E dot type is if E 2 dot type equal to s, and E dot type is E 1 dot type is s 2 t. So, E 2 dot types is s and E 1 dot type should be from that domain like if E 2 is an integer and then this function for E 1 it should its domain should be integer and if its domain is same as u E 2's type and the then we have to look into the range of E 1. So, range of E 1 is the return value type of E 1. So, that is t since the function given is s 2 t, so, that is the return type of it and then it will be in that case the E's type will be t

otherwise it will be a type error. So, in this way we can have type checking for functions noted.

(Refer Slide Time: 28:37)

## Type Equivalence

- It is often needed to check whether two type expressions 's' and 't' are same or not
- Can be answered by deciding equivalence between the two types
- Two categories of equivalence
  - Name equivalence
  - Structural equivalence

FREE ONLINE EDUCATION  
swayam  
India Niye, Jee Niye

So, we can talk about type equivalency. Now, it is sometimes it is needed to check that whether 2 types s and t they are same or not. So, we can answer this by deciding equivalency between the 2 types and this equivalency there can be 2 categories; one is called name equivalency, another is called structural equivalency. So, equivalency check is necessary because many times we need to derive for an expression like say if x equal to say y; now this x and y they may be apparently of different types, but we want to see whether they are they are of some equivalent type or not.

So that if it is one type of equivalency like name equivalency so it will it will find that if the names of the 2 types are different then it is they are not named equivalent, so, in that case it is a type error. And, in some cases so, we will look for further. So we look into the structural part of it and then we will see that whether they are structurally equivalent or not and if they are structurally equivalent then we can say that the 2 types are equivalent.

So this type equivalency is very important because we can so, many times for the programmers ease, so they define different types and they use different constructs for defining the types and then while combining them while checking them in the later part of the program we are we do not make the programmer constrained by the type

definitions. So if they are equivalent type then it should be taken into consideration. So that gives us to this equivalency.

So we will see this equivalency rules, how to do all these checks in the next class.

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

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

(Refer Slide Time: 00:17)

**Name Equivalence**

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

```
typedef int Value
typedef int Total
...
Value var1, var2
Total var3, var4
```
- Variables var1, var2 are name equivalent, so are var3 and var4
- Variables var1 and var4 are not name equivalent, as their type names are different

15

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

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

(Refer Slide Time: 01:32)

**Structural Equivalence**

- Checks the structure of the type
- Determines equivalence by checking whether they have same constructor applied to structurally equivalent types
- Checked recursively.
- Types array( $I_1, T_1$ ) and array( $I_2, T_2$ ) are structurally equivalent if  $I_1$  and  $I_2$  are equal and  $T_1$  and  $T_2$  are structurally equivalent

*types def integer value or number*

*x: array[50 int]; y: array[100 int]; y = x*

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

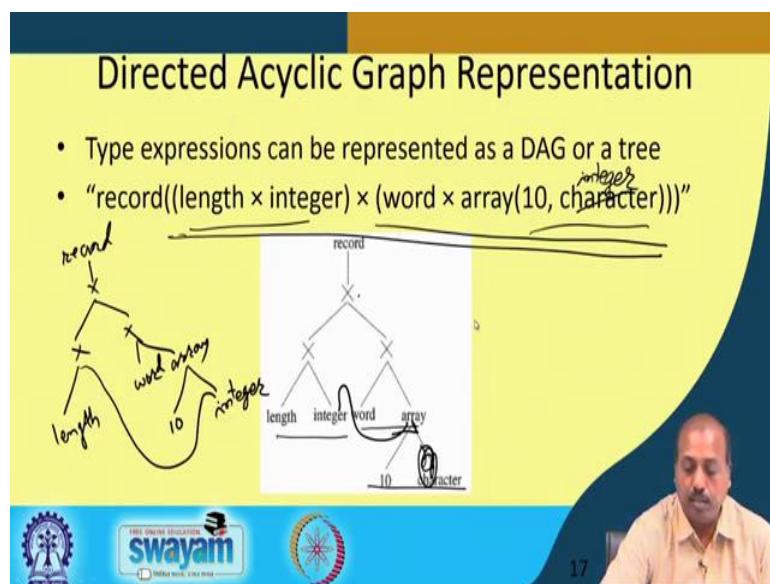
So, this check is done recursively like if it is 1 step further so till you reach a basic types. So, the check will continue. So, it will bring it down to the basic level and that basic level it will check whether they are going to be of same type or not. So, they will be checked recursively. So, if I have got types  $I_1, T_1$  an array of whose type is side size is  $I_1$  and type of individual elements  $T_1$  and you have got another array whose size is  $I_2$  and type is  $T_2$ , they will said to be structurally equivalent if  $I_1$  and  $I_2$  they are equal and  $T_1$  and  $T_2$  they are structurally equivalent.

So, this is like if I have got say 1 integer array. So, this  $I_1$  equal to say 50 and  $I_2$  equal to say 100; though both  $T_1$  and  $T_2$  are integers. So, this is also integer; this is also integer. So, in that case; so in that case, so  $I_1$  and  $I_2$  they are not of same type. So, naturally I will not have this the overall same size  $I_1$  and  $I_2$  are of this not same size. So, if I have got

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

So, this is 100 and this is 50 though the basic types are integer, but it will give rise to an error. So, we can, so, this is the structural equivalences because this I1 and I2 these values are not matching. So, they are not structurally equivalent.

(Refer Slide Time: 04:41)



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

So, that is there and we have got this word cross array and that array is again this 10 characters, so, this 10 character is here. So, that is a product and then we have got the cross product of this word and array. So, that gives rise to this part and then, we have to go at this point. So, we have got the cross product of the whole thing, so, that is the record. So, in this way we can have some tree type of representation for this particular declaration. So, there can be directed acyclic graph also like if at the lowest level it may so happen that this one instead of being a characters, so this may be an integer.

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

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

(Refer Slide Time: 07:06)

## Function dag\_equivalence

```

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

```

So, this is a function that checks for this equivalency of this directed acyclic graphs like if s and t represents the same basic type then return type is true; so, this is the first rule. So, if s and t they are of basic types if both are integer or real or character or Boolean like that. So, that in that case it will return that s and t are equivalent, if s is an array I1 T1 and t is another array I2, T2. So, in that case we have to if I1 and I2, these two values are same; then you need to check whether this at T1 and T2. They are of same equivalent type or not. So, accordingly it is calling this function dag-equivalency recursively with the parameters T1 and T2 otherwise it will return false.

So, if I1, I2 are not same in that case it will say that it is they are not equivalent; otherwise it will depend on the types of this T1 and T2. Now, if s is some record type of

structure s1 cross s2 and t is another record t1 cross t2. Then, we have to go for the equivalency check of s1 and t1 and further s2 and t2. So, the return value will depend on the equivalency of s1, t1 and equivalency of s2, t2. So, it checks the dag equivalency of s1, t1 and dag equivalency of s2, t2. So, after this equivalency check; so if both of them written true, then only it will return true; otherwise it will written false.

Now, if s is a pointer s is a pointer s1 and t is a pointer t1, so, they are 2 pointers. So, then it will be returning the dag-equivalency of s1 and t1 because if there is s1 and t1, they are equivalent; then, this s is a s and t. So, they are also pointers so similar types. So, as a result they are declared to be equivalent. Now, if s and t they happens to be function call. So, if that is s goes from s1 to s2 its domain is s1 and range is s2 and t grows from t goes from t1 to t2 that is to had t1 is the domain and t2 is the range. Then, it will be checking the dag equivalency of s1, t1 to see that their domains are same and it will check the dag equivalency of s2, t2 to check that their ranges are same.

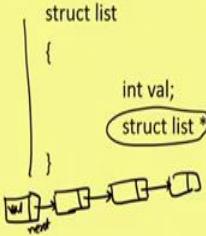
So, if they are; if they are correct then it is fine and so, otherwise if s and t do not satisfy they do not come into any of these 5 rules that we have in that case it is returning false. So, that way it is telling that this the s and t are not equivalent. So, this function dag equivalency, so it can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

(Refer Slide Time: 10:25).

## Cycles in Type Representation

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

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



list = record

```
graph TD; list --> val1[val]; list --> integer1[integer]; list --> pointer1[pointer]; val1 --> list
```

(a)

list = record\*

```
graph TD; list --> val2[val]; list --> integer2[integer]; list --> pointer2[pointer]; val2 --> list
```

(b)

- (a) Acyclic representation (b) Cyclic representation



19

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

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

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

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

(Refer Slide Time: 12:43)

## Cycles in Type Representation

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

Value → var1  
typedef int Value  
var1, var2

20

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

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

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

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

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

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

(Refer Slide Time: 15:42)

## Type Conversion

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

```

int x -> real / x
real y;
z = x + y
  int      real
        +-----+
        |       |
        x       y
    
```

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

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

21

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

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

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

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

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

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

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

(Refer Slide Time: 19:20)



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

(Refer Slide Time: 20:09)

$E \rightarrow E_1 + E_2$   
{ check E<sub>1</sub>.type and E<sub>2</sub>.type to be one of char, int, float, double;  
If other types E.type ← type.error  
if E<sub>1</sub>.type = float and E<sub>2</sub>.type = float  
    E.type = float  
else if E<sub>1</sub>.type = double and E<sub>2</sub>.type = double  
    E.type = double  
else if E<sub>1</sub>.type = int and E<sub>2</sub>.type = int  
    E.type = int  
else E.type = char  
    E.type = char  
}

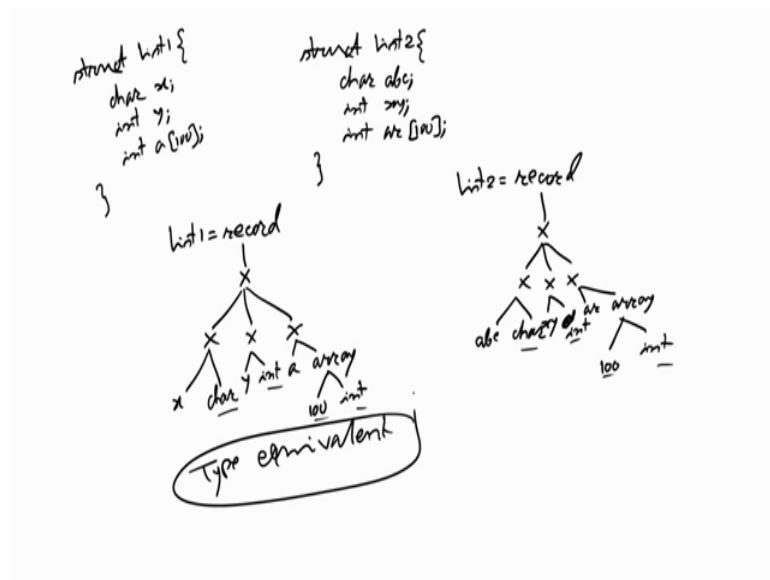
So, the first one that we will look into is for the type coercion. So, we have previously seen that type coercion this type check of this expression like say E producing E<sub>1</sub> plus E<sub>2</sub>. So, previously we said that if these type of E<sub>1</sub> and type of E<sub>2</sub> are same then it is fine otherwise there is a problem. So, we can modify that check and we can do it like this.

So, first we can check E<sub>1</sub> dot type and E<sub>2</sub> dot type check E<sub>1</sub> dot type and E<sub>2</sub> type to be one of the basic types like character integer; then float doubles. So, if we are talking in terms of C language. So, these are the basic types that we have. So, it has to be one of those types. So, if it happens to be like that, then if other types if it is not one of those basic types, then we can say that E dot type is type error.

So, this is a type error, otherwise for if E<sub>1</sub> dot type is. So, if E<sub>1</sub> dot type is float and E<sub>2</sub> dot type is also float if both of them are float, then we can say that this e dot type is equal to float. Similarly, if both of them are double else, if E<sub>1</sub> dot type equal to double and E<sub>2</sub> dot type equal to double, then you can say that E dot type equal to double. Else if E<sub>1</sub> dot type equal to integer and E<sub>2</sub> dot type is also equal to integer. So, in that case E dot type is equal to integer, else E dot type equal to character.

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

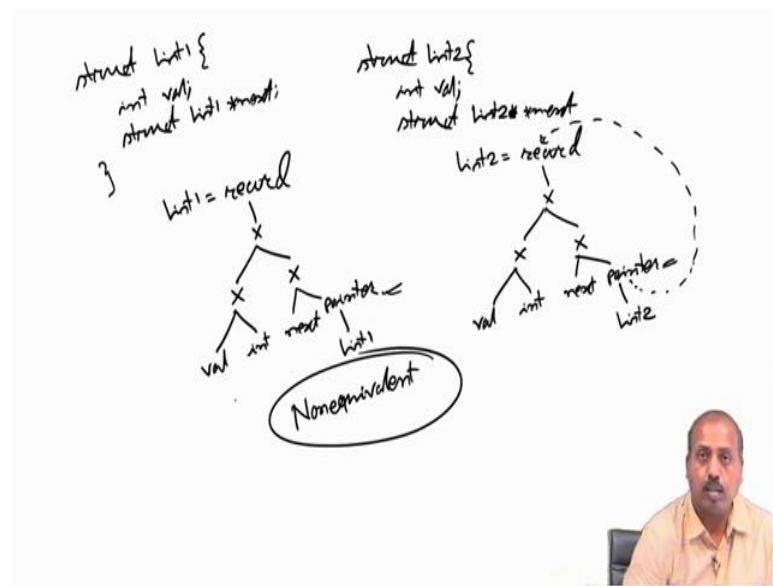
(Refer Slide Time: 23:42)



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

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

(Refer Slide Time: 26:29)



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

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

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

Thank you.

**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 is 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)

CONCEPTS COVERED

- ❑ Information in Symbol Table
- ❑ Features of Symbol Table
- ❑ Simple Symbol Table
- ❑ Scoped Symbol Table
- ❑ Conclusion

FREE ONLINE EDUCATION  
swayam  
India@70

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)

## Introduction

- Essential data structure used by compilers to remember information about identifiers in the source program
- Usually lexical analyzer and parser fill up the entries in the table, later phases like code generator and optimizer make use of table information
- Types of symbols stored in the symbol table include variables, procedures, functions, defined constants, labels, structures etc.
- Symbol tables may vary widely from implementation to implementation, even for the same language

FREE ONLINE EDUCATION  
swayam  
India@70

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 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.

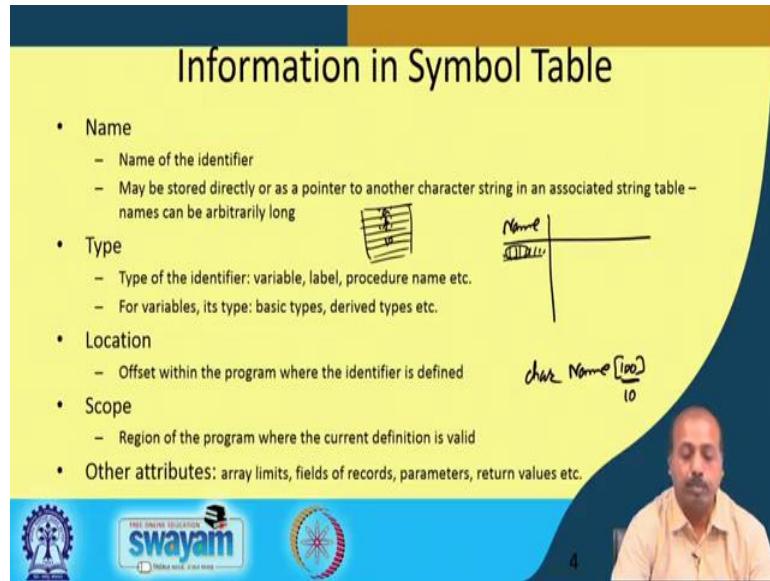
So, looking into the symbol table we understand that either type of y is an array or type of y is a function. So, this procedures functions so, they are also put into the symbol table; so, that we can easily search them out. Some defined constants like we can have apart from these variables I can have 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 x1 character y1. 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 x1, y1.

So, this x1, y1 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)



The slide has a yellow header with the title "Information in Symbol Table". Below the title is a bulleted list of information stored in a 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.

On the right side of the slide, there is a hand-drawn diagram of a symbol table entry. It shows a vertical line with a box labeled "Name" at the top and a box labeled "char Name [100]" with the value "10" below it. To the left of the slide, there is a logo for "FREE ONLINE EDUCATION SWAYAM" with the tagline "India's First Online Education Platform".

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

The slide features a yellow header with the title "Usage of Symbol Table Information". Below the title is a bulleted list of four points about the usage of symbol tables. To the right of the list is a handwritten diagram showing the evaluation of an expression and the mapping of variables to object code. The handwritten notes include:  
- Evaluation of  $x = 7 + 2 * 6$  resulting in  $x = 15$ .  
- A diagram showing two boxes: one for "integer x" (labeled "4 bytes") and one for "character y" (labeled "1 byte").  
- An arrow pointing from these boxes to a stack of horizontal bars labeled "object code".  
- A handwritten note next to the "object code" stack says "integer x occupies the first 4 bytes".  
The bottom of the slide has a blue footer with logos for IIT Kharagpur, Swayam, and the Indian Space Research Organisation (ISRO). There is also a small photo of a person on the right side.

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 when you are going to check types of some expression; say some assignment statements or some or some say operation like  $x$  plus  $y$  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 to 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 t1 equal to z into w, then t2 equal to y plus t1 and then x equal to t2. So, it is converted into these three statements. So, this t1, t2 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 t1 and t2 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 t1 and the space for t2.

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 t1 and t2 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 t1 and t2 so, they are not required simultaneously.

So, the same space t1, t2 can be given to the temporaries t3 and t4 also ok. Now, while doing this merging we have to know that these types of t1 and t3 are same. So, that the same amount of space can be allocated for them they can share the same amount of space. Similarly this t2 and t4 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 f  
{ int x,y;  
}*

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

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.

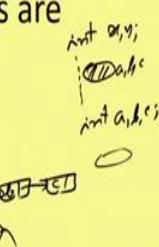
So, that we can just so, the symbol table is very large so, it cannot be accommodated in the primary storage. So, we will do we will do put a part of it in the secondary storage. Then there are scope rules like in block structured language a variable defined in upper blocks must be visible to inner blocks 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 ✓ 

8



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 its index based on some function and then those symbols will be stored at the corresponding place only.

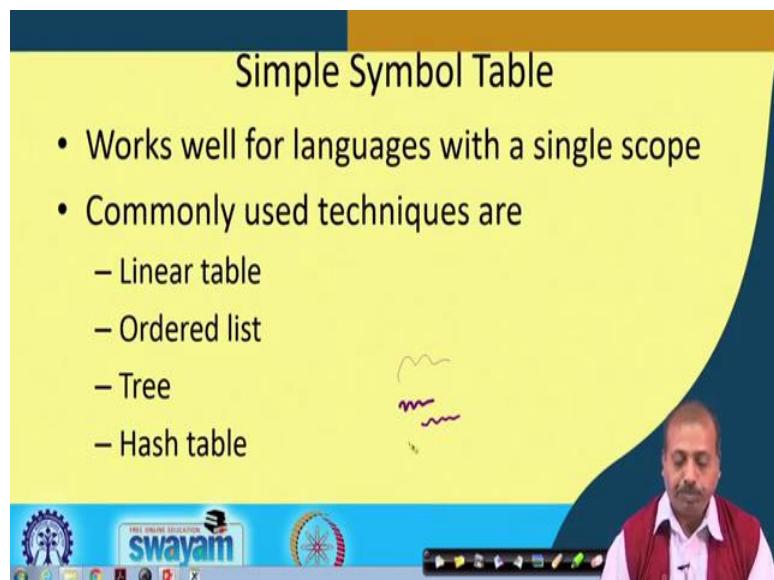
So, this way different types of organizations of this simple symbol table can be taken up. And, then when we are going to complex symbol table so, basically the scope rules they will define this complexity of the structure. And, accordingly they will be organized, but following these basic principles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

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

**Lecture - 44**  
**Symbol Table (Contd.)**

In our last class we have been discussing on simple Symbol Table and we have seen that this simple symbol tables they work well for languages with a single scope.

(Refer Slide Time: 00:25)



The slide has a yellow background with a blue header bar. The title 'Simple Symbol Table' is centered in the header. Below the title is a bulleted list of four items:

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

On the right side of the slide, there is a small video frame showing a man with a beard and mustache, wearing a red vest over a pink shirt, looking down at something. The video frame is set against a blue background with some decorative elements like gears and a Ferris wheel.

And single scope means all variables and identifiers that we have in the program they are visible to the entire program. So, there is nothing like there are like where they are, like if we have got local variables in the visible only within the function or procedure. So, that type of considerations is not there. So all the variables and identifiers so, they are visible to the entire program. So for this type of programming languages, it is we can have a very simple structure and these are the alternatives; like we can have a linear table, we can have ordered list tree or hash table. These are the commonly found of symbol table organisations.

So one thing I would like to tell you that it is not mandatory that you should follow only one of these 4 alternatives for symbol table organisation. So depending on the programming language and the type of information, that you need to store so, you may prefer some other organisation as well. So, so or some complex data structures can be

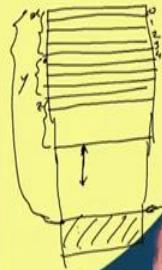
thought about and then the symbol table maybe organised like that. So these are actually the commonly used options that we are exploring.

(Refer Slide Time: 01:37)

## Linear Table

- Simple array of records with each record corresponding to an identifier in the program
- Example:

	Name	Type	Location
int x, y	x	integer	Offset of x = 0
real z	y	integer	Offset of y = 4
procedure abc	z	real	Offset of z = 8
...	abc	procedure	Offset of abc = 12
L1...	L1	label	Offset of L1 = 16
...			



So next day if we look into these different table organisations the first one is the linear table. So, it is a simple array of records with each record corresponding to an identifier in the program. So, in the program, like here is an example, we have got a program where we have got these variables integer variables x and y, a real variable z, the procedure abc and we have got a label L1. So, so far as per as the symbol table contents are concerned so, we are storing the name of the symbol or name of the identifier, its type and the location; location is the offset from the beginning of the program.

So, what do you mean offset? It is like this that suppose we have got a program say, this program it is now if you say that at the if you consider the corresponding object file the object code that is produced. So, in that object code, I should have space for this x and y. So, x is the first location that we have in the program. So, you can say that the offset of x is equal to 0 fine, then the after that we have got integer y.

So, assuming that these integers are 4 byte long so these 4 bytes will be reserved for x. So, these 4 bytes are reserved for x and after that the next 4 bytes it will be storing for y it will have the y values. So, these 4 values so, they are corresponding to y. So, y's offset is this one so, that is equal to 4. So, if you look in terms of byte offset so, z is 0 1 2 3 and then y starts at 4. So, that way the offset of y is 4 and after that offset of z. So, offset of z is 4

plus size of y the size of y is 4. So, z will start from this point and it will be spanning over next 6 bytes.

So if you assume that the real numbers are required in 6 byte so it will have 6 bytes. So this offset is equal to 4 this offset equal to 8 and then we have got some statements here. So it is taken as if we do not have any more definitions here. So, they are some pro some statements are coming. So we can say that from this point onward the actual code for the program we will start. So, from this point onward the code we will start. So, accordingly we can calculate the offsets like, as we are car converting this programming language statements into machine language.

So accordingly it will take few lines of the few bytes of the machine language program to store the corresponding source language statements. So that way the offsets will be proceed processing. Now, when we come to this procedure a b c. So, with so, this a b c name is installed into the symbol table, its type is set as procedure and the offset is from the beginning of the program after how much time, after how much byte this a b c is coming. So if I assume that this so this translation goes up to this point and then from this point onwards I am storing the procedure the code for procedure a b c. So this is the code for procedure a b c.

So what is the offset that is starting from the beginning of the program; if I assume that this location is 0 so what is the corresponding byte index. So that is the offset of a b c. So the concept comes like this because ultimately this program will be loaded into memory and when it is loaded so a similar such image we will get created onto the memory. So they will be copied byte by byte from the secondary storage to the memory main memory.

So, we they are the offsets will be maintaining their relative positions, variables that I diff defining the program the program statements that we are defining. So they will be at relative offsets as it is there in this translated program. So this procedure a b c , we will have the offset of a b c is told a here, then L1.

So L so after that again some statements are coming so, they are coming here and at some point of time this level L1 occurs. Though in languages like C we are not having these levels much, but in some programming languages a we have that. So, in C also this labels are allowed, but it is not encouraged, but anyway so, since it is there as an

identifier in many programming languages. So, it is also kept in the symbol table that L1 is of type level and as similarly we can compute the offset of L1 from the beginning of the program.

So, that way all the offsets are calculated and those those offset values are stored in the location part of the symbol table. So, this is the linear table organisation. So, this is simply an area of records and each record has got 3 fields in it: name of the symbol, the type of the symbol and the offset. So, so that way we can have this structure.

(Refer Slide Time: 06:43)

**Linear Table**

- If there is no restriction in the length of the string for the name of an identifier, string table may be used, with name field holding pointers
- Lookup, insert, modify take  $O(n)$  time
- Insertion can be made  $O(1)$  by remembering the pointer to the next free index
- Scanning most recent entries first may probably speed up the access – due to program locality – a variable defined just inside a block is expected to be referred to more often than some earlier variables

*Handwritten note: A diagram shows a rectangle with a cross inside, labeled 'x'. To its right is a circle with a cross inside, labeled 'y'. Above the rectangle is a brace spanning both, labeled 'x = y;'. Below the circle is another brace spanning both, labeled 'x = y;'. There is also a small circle with a cross inside, labeled 'z'.*

swayam

Now apart from this linear table so, other organisations are also possible, but linear table if there is no restriction on the length of the string for the name of an identifier string table may be used. So, what I want to so, mean is that this name part so, name part ultimately; so, we are storing the name of the symbol. Now, if there is if the programming language we will tell whether these identifiers should have a finite length or not. So, if it is not finite so, in that case it is better that this name field instead of being a character array so, it can be a pointer and it points to the string table.

(Refer Slide Time: 07:17)

**Linear Table**

- Simple array of records with each record corresponding to an identifier in the program
- Example:

Name	Type	Location
x	integer	Offset of x
y	integer	Offset of y
z	real	Offset of z
abc	procedure	Offset of abc
L1	label	Offset of L1

So, as I was discussing in the last class so this string table; so, it will have all the string variables all the strings that are occurring in the program. So here this so, suppose for this x. So, this it will be stored it will be storing this x followed by the null character and this pointer will be pointing to this particular entry.

Similarly after that we have got the symbol y followed by null character. So, this will be a pointer to this y. So we can have the situation like this; similarly at some point we have the name level procedure name a b c. So, it is stored like this a b c and then the null character and then this will be a pointer to this entry in the string table.

So this way using string table so we can have we can store the variable names which are whose length is not restricted. So if the programming language says the length of the symbols not restricted then we can do that. However, mean in most of the cases we have got this length the restricted. In that case so, we can have this simple character array as the name part. So in that case when it is when string table is used in the name part will be holding the pointers. The operations that we do on symbol table as we say previously that lookup insert modify and delete.

So if you want to search in the table then it will take order n time because, assuming that there are a number of entries in the table. So it will use a simple linear search so, it will take order n time. Similarly, if you want to insert then you need to go to the end of the table. So, assuming that you go to the end so, it so, that can be taken like that. So, then

modification; modification also it needs to search. So, because so, that will take order n time, so, insertion can be made order 1 by remembering the pointer to the next free index. So, if you remember what is the next free index then it is order 1 because, we do not have to search for a free slot in the table ok.

So if you are remembering the say one point one index like how many entries in the table or fool in that case the next insertion is at the next free slot. So that way it can be done in constant time. Scanning most recent entries first may probably speed up the access due to program locality? So what we mean is that what we mean is that; so, the so, program locality means that at any point of time if I am if I having a program. So, suppose I am at this particular line in the program. So in this line suppose it is accessing the variable x, then it is very much likely that in near future, it will be accessing the variables which are close to x.

Close to x in the sense that they may be defined within the same block, like if this is a C program and I have got a block like this where this x and y variables have been defined. Then in this part of the code it is very much likely that it will be referring to the variables x and y. So, if I can somehow make this x and y easily accessible then my compilation speed will be better because, I do not have to search the symbol to the entire symbol table need not be search. So, or searching the first few entries itself so, we can get the x and y variables. So, this is due to program locality because due to the locality so, it is very much likely that; so, the variables which are defined in the large in the most nested block.

So, they will be referred to more often compare to some other variable; like there may be a statement like say x equal to y plus z where z is a non-local variable. But, this is the possible the chances are less because it will be accessing the local variables more. So, scanning the most recent entries first it will be probably speed up the access process; as a result the compilation will be easier will be faster.

And, a variable defined just inside a block is expected to be referred to more often than some variable some then some earlier variable. So, this is what I was talking about that in that x equal to y plus z; so, in that access you see two local variables are accessed and one global variable is access one non-local variable is accessed. So, as a result most of

the time it is accessing local variables. So, we should do something so, that this most recent entries they can be searched very easily.

(Refer Slide Time: 12:17)

**Ordered List**

- Variation of linear tables in which list organization is used
- List is sorted in some fashion , then binary search can be used with  $O(\log n)$  time
- Insertion needs more time
- A variant – self-organizing list: neighbourhood of entries changed dynamically

Handwritten notes on the slide:

- A diagram showing a list of symbols:  $\overbrace{ab}^{aa}$ ,  $\overbrace{ac}^{aa}$ .
- A diagram showing a list of symbols:  $\overbrace{aa}^{ab}$ ,  $\overbrace{ab}^{ac}$ .

So, so this is the variation of linear list which is an ordered list. So, it is a variation of linear tables in which least organisation are used. So, this is the least organisation and least is sorted in some fashion and then a binary search is used for  $\log n$  time. So, sorted so, if you are so, sorted in some fashion means it may be based on the name of the symbol you solve them on the ascending order. Then if it is so, then as you know that once the array is sorted so, we can use binary search and that binary search takes order  $\log n$  time. So, that way it is easy; however, the problem comes when we are trying to do insertion.

Because, insertion it has to be made in ordered fashion; like if I have got say in the symbol table the mm symbols if we have in the symbol table the mm symbols like say a b and a c ok. And, then suppose I want I want to insert a new symbol a a then the difficulty is that you can so, you cannot insert a a at the end because, then this sorting order will be will be lost. So, what you have to do is that you have to have a a at the beginning followed by a b followed by a c. Now, for doing that this a c and a b they need to be copied down by one position making the first slot free and there you can insert this a. So, that it becomes like this. So as a result this insertion becomes order  $n$  time whereas, search remains search becomes  $\log n$  order  $\log n$  time.

So, this is the, this is a trade off that we have to that we have to do. So, it is a insertion is made more costly so, that this search becomes easier. A variant which is known as self organising list so, neighbourhood of entries are change dynamically. So, we will see how is it taking place. So, we change the so, the so, so that as and when these variables are accessed so, local variables so, they are neighbour to each other. So, that way we can have some better access method. So, we will see how this can be done.

(Refer Slide Time: 14:35)

## Self-Organizing List

- In Fig (a), Identifier4 is the most recently used symbol, followed by Identifier2, Identifier3 and so on
- In Fig (b), Identifier5 is accessed next, accordingly the order changes
- Due to program locality, it is expected that during compilation, entries near the beginning of the ordered list will be accessed more frequently
- This improves lookup time

**Before:**

Identifier 1  
Identifier 2  
Identifier 3  
Identifier 4  
Identifier 5  
free

first → Identifier 4 → Identifier 5

(a)

**After:**

Identifier 1  
Identifier 2  
Identifier 3  
Identifier 4  
Identifier 5  
free

first → Identifier 5 → Identifier 4

(b)

Identifier 5  
Identifier 4  
Identifier 3  
Identifier 2  
Identifier 1

→ Identifier 5

So, this is an organisation of self or this is, as this is an example of self organisationally self organizing list. So you see suppose in so, first we consider this particular figure; this figure a in this figure a identifier 4 is the most recently used symbol. So, either so, we keep track of the sequence in which the symbols are being accessed. So, for so, the there is a pointer first or if that that points to the last referred a symbol ok. So, in this case identifier 4 is the most recently used symbol and before that it was identifier 2. And before that it was identifier 3, before that it was identifier 1 and before that it was identifier 5; that means, in my program there was a sequence of access like in the some in some lines I accessed identifier 5 identifier 5.

After that I have accessed before this identifier 5, identifier 1, then after some lines I have got this identifier 1. Then after some line we have got this identifier 3, identifier 3, after that I have accessed a identifier 2 and then identifier 4. So, that this identifier 4 this is the most recent one before that we have accessed identifier 2 in this statement.

Before that we have accessed identifier 3 in this statement then identifier 1 in this statement and then identifier 5. So, so, this is the current sequence ok. Now, suppose after this so, the is the advantage like it by program locality it is expected that in near future will be accessing in this order only. So, it is more likely that identifier 4 will be accessed compared to identifier 2.

And, identifier 3 will be just chances of accessing identifier 3 is even less, identifier 1 is even lesser and identifier 5 access probability is least. So, that is the expectation and that happens due to program locality. Now, after sometime so, after sometimes so, if identifier 5 is accessed. So, here suppose so, identifier 5 is accessed then this least will change the this first pointer comes to identifier 5 and this points to identifier 4 from 4; so, remaining part remains unaltered ok. So, that they are remaining unaltered and we are just remembering last 5 accesses. So, this identifier 5 access so, this particular link is lost because we are just remembering last 5 accesses.

So, that way we can have this mod these list modifiers. At any point of time when you are doing compilation so, this first pointer points to the least recently referred symbol. And then successive pointers so, they will be access they will be referring to the previously accessed symbols in that order. So, program locality so, it will be say that it will have the expectation that during compilation entries near the beginning of the order list will be accessed more frequently. So, this ID 4 ID 5 this ID 4 ID 3 ID 4 ID 2 ID 3; so, they are supposed to be accessed more compared to ID 5. But, here the example that we have taken so, it is slightly non-intuitive and all on a sudden instead of identifier 4 3 to identifier 5 has been accessed.

So, that can happen because it is not mandatory the program will always follow the locality of reference; sometimes it act does non local reference also. And here it is done in identifier 5 and as a result this list gates reorganized, but after that so, it will be now it is now it is having in the figure b. So, we are having the situation where the least recently least recently access symbol is pointed to by the first pointer. So, this way it can improve the lookup time because, I do not search this local identifiers much more faster compared to non-local identifiers.

(Refer Slide Time: 19:19)

The slide has a yellow header with the word 'Tree'. Below it is a list of bullet points:

- Each entry represented by a node of the tree
- Based on string comparison of names, entries lesser than a reference node are kept in its left subtree, otherwise in the right subtree
- Average lookup time  $O(\log n)$
- Proper height balancing techniques need to be utilized

On the left, there is handwritten text 'Binary Search Tree' next to a diagram of a binary search tree. The tree has root 'x'. 'x' has children 'abc' and 'y'. 'abc' has child 'L1'. 'L1' has child 'P'. 'y' has child 'z'. To the right of the tree is a small diagram showing a path through a binary search tree with nodes labeled 'P', 'Q', 'R', and 'S'.

At the bottom of the slide, there is a blue footer with the 'swayam' logo and other icons.

So, next will be looking into another type of organisation, which is known as tree organisation. So, each entry represented by a node of the tree. So, here the entries are same like we have got the name of the symbol, then we have got its type, its offset etcetera. But instead of putting them on a linear table so, we are putting them in the form of a tree. And, based on string comparison of names entries lesser than a reference node are kept in the left sub tree otherwise it is on the right sub tree. So, if you look into the tree at this point you see that root is x and then the symbol a b c.

So, symbol a b c should come to the left side of x because, when you are doing a comparison the first characters are compared. So, x is compared with a and since x is if you look into the corresponding ASCII codes then x ASCII code is more than a ASCII code. So, as a result if x is taken to be greater than a b c or a b c is taken to be less than a. So, as a result this b c is put on the left sub tree and similarly y is greater than x.

So, y goes to the right sub tree of x and then when it comes to L1 so, you see that x L1 L so, L is less than x. So, it comes to the left sub tree. Now, this node has got the symbol a b c and this L is more than small a in terms of ASCII code. So, as a result L goes to the L1 goes to the right sub tree of a b c.

Similarly, here when z comes so, z is greater than x. So, it is in the right sub tree of x then when compared with y it is greater than y. So, z is ultimately put in the left sub right sub tree of y. So now, if we have got a new symbol coming for example, suppose there is

a symbol say P; now where will it go? So, P so, it is less than x. So, it will come to the left sub tree of x then it is a b c. So, a is greater than so, P is greater than a so, it will come to the right sub tree; right sub tree has got this L and this P is less than L. So, P will come a P is sorry P is greater than L. So, P will come at this point K L M N O P so, P is greater. So, P will come to the right sub tree so, that way these new symbols will be put into the symbol tree into this tree type symbol table.

So, what you are doing is we are maintaining some structure which is known as binary search tree. So, your maintaining this structure of binary search tree such that the nodes which are less than the root node; so, they will go to the left sub tree and nodes which are larger than the root so, they will go to the right sub tree. And, this process we will go on recursively.

Now, look up top L average look up time will be order  $\log n$  because, there are if there are  $n$  symbols and it is pa perfectly balanced. If this tree is perfectly balanced then there will be  $\log n$  such levels of course, that may be a situation where this symbols are such that they give rise to a tree like this. Suppose I have got the symbols coming in this sequence a b c d like that such that successive symbols are all greater than the previous symbol.

So, this will create a tree like this and in that case complexity instead of being  $\log n$ ; so, this will become order  $n$  such complexity will become order  $n$ . But for all practical purposes so, this will not happen because we will be getting symbol such that we get the left and right branches both and it becomes more balanced. And, if it is not balanced then there are some techniques by which we can do this balancing which is known as height balancing. I will suggest that you look into some data structure book that will be talking about this height balanced tree and all.

So, proper height balancing techniques can be utilised; so, that this complexity of this lookup can be made to be equal to  $\log n$ . So, this way we can take help of this tree type of structures.

(Refer Slide Time: 23:41)

## Hash Table

- Useful to minimize access time
- Most common method for implementing symbol tables in compilers
- Mapping done using Hash function that results in unique location in the table organized as array
- Access time  $O(1)$
- Imperfection of hash function results in several symbols mapped to the same location – collision resolution strategy needed
- To keep collisions reasonable, hash table is chosen to be of size between  $n$  and  $2n$  for  $n$  keys



Next we will be looking into another structure which is known as hash table. So, hash table this is useful to minimise access time. So, when it is the most common method for implementing symbol tables in compilers. So, what is a hash table?

(Refer Slide Time: 24:03)



45, 61, 95

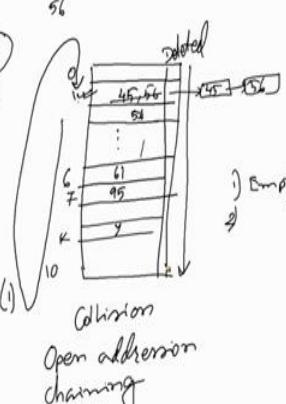
$$h(x) = x \bmod 11$$
$$h(45) = 1$$
$$h(61) = 6$$
$$h(95) = 7$$
$$h(y) = y \bmod 11$$
$$= x$$

56

0(1)

Collision  
Open address  
Chaining

Empty slots



So, hash table is something like this like suppose, I have got some symbols some numbers. Let us work with same numbers say 45 61 and say 95. So, these are the 3 numbers and what we do we take so, so if this is equal. So, you take a function  $h$  of  $x$  which is  $x \bmod 11$ . So, 45 mod 11 so,  $h$  45 will give me 1 ok; then  $h$  61 will give me 6

and  $h(95)$  will give me 7 ok. Now, if it is like this then we can do what we do in the table we have got entries running from 0 to 10. If it is mod 11 so, that entries will be running from 0 to 10. So, in the 0 slot there is nothing, in slot number 1 we have got 45, in slot number 6; so, these are all entry a empty at slot number 6 we have got 61 and at slot number 7 we put 95.

So advantages that now, suppose I give you a number and ask whether it is there in the table or not search for a particular number. So, for any particular number is given say  $y$  is a number given as a question. So, what we do we apply the  $h$  function on  $y$  to get  $y \bmod 11$  and we need to come to the corresponding index. So, if this value is equal to say  $k$  then we look at the  $k$  th entry into this table. So, if this number is there if the number  $y$  is there as per our policy the number  $y$  will be in this slot only.

So, as a result for searching a number I do not have to go for a sequential search through this entire table, I can do it in constant time. So, this access can be made constant time. However, the difficulty with this type of organisation is that there will be many slots which are empty. So, in between there may be many empty slots; so, that is one problem.

So, first problem is empty slots because depending upon the, this mode value that we have got. So, I will get in day array indices in that range table Indies in that arranged. So, many of the entries may not have the corresponding values there and second thing is that there can be several numbers which fall on to the same index like say; so, if I have got say 56, suppose the next number given is 56. So, when I apply this function so,  $h(56)$  is also equal to 1. So, far so, I have I need to store 2 numbers here now 45 and 56, but that is not possible because every slot has got a single number. So, this is known as the problem of collision.

So, if you look into the data structures and there the there are some techniques by which we can resolve this collisions. So, so, what you can do maybe we store 56 at the next free slot ok; the next free slot that is available. Or, what we can do is that we can we do not put the numbers here directly rather we make a chain where we put this numbers on a chain 45 56 like that. So, these are these are known as collision resolution techniques.

So, we resolve collision like that and then if we are depend you doing like this. So, when we are putting this number of the next available free slot. So, this is called open addressing, this collision resolution technique is called open addressing and the other one

where we are putting them on a change. So, they are that is that technique is known as chaining.

So both the methods though they have got their overheads like you have got we have got this thing; we need to somehow search through this chain or search through this entries ok. So till for example, if the question is say 56 and we have use a open addressing so, 56 is here. So applying the hash function will come to slot number 1 and then we have to go on searching from this point and round come back to that entry. So if the number 56 is not there, then will be searching for the entire table and we will not be getting it so that, that at that point we can declare that the number is not present in the table.

But it is a bit costly definitely whereas, so but it is a very much likely that if the number is there then you can get it the then you can get it within a very short amount of time and chaining is slightly better than that. The difficulty that we have is with the deletion because, if you want to delete an entry then how do you delete it; how do you say that this particular slot does not have the number. So we need to have another flag here there which we which we call say deleted flag and that flag has to be turned on if a number is deleted. If a number is not called if this particular table entry is not containing a valid data in the deleted flag should be on.

Now so for symbol tables of this is not a big problem because for symbol table we can just have this deleted we can have the deletions are very less. So we have got only in mostly insertion and search and the search is the most important operation in symbol table. And if we have got a hash table type of organisation then this search can be done very fast.

So that way that is why this hash table based say symbol table hash table based symbol table organisation is very popular and it is used in many compiler design or compiler designs. So that when there are large number of symbols and a symbol table is very large; so instead of searching through the table in linear or binary search fashion. So it is better that we organise them as hash table and in constant time, we can access the individual entries from the table.

**Compiler Design**  
**Prof. Santau Chattopadhyay**  
**Department Electronics and Electrical Communication Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture - 45**  
**Symbol Table (Contd.)**

So hash tables. So there you used to minimize the access time as we discussed in the last class.

(Refer Slide Time: 00:21)

**Hash Table**

- Useful to minimize access time
- Most common method for implementing symbol tables in compilers
- Mapping done using Hash function that results in unique location in the table organized as array
- Access time  $O(1)$  ✓
- Imperfection of hash function results in several symbols mapped to the same location – collision resolution strategy needed
- To keep collisions reasonable, hash table is chosen to be of size between  $n$  and  $2n$  for  $n$  keys

$$h(m) \rightarrow \text{ASCII}(a) + \text{ASCII}(b) + \dots + \text{ASCII}(z)$$
$$h(m) = \dots$$

So, this access is a constant time access. So, this is basically access time is order 1. So, as we have noted here this one. So, most common method for implementing symbol tables in compilers. So, that is one we have already seen why is it like that. And mapping done using hash function that results in unique location in that table organized as array. So, there you can think about different hash functions. The if the number if the entries that you are going to put are integers then we can have you can have say hash function which is say like the mod function can be used.

However, if the that where the items that you are going to store in the hash table or on the basis of which will be doing the search, so if it not a number. For example, for symbol table particularly what happens is that we are not going to store them as numbers, but rather we are going to store the name of the symbols as character string and we will be searching based on character string only? Now how to convert this character

string to a number? So there can be several strategies for that a very simple strategy may be like this that, if I have got the got a string like say a b x y ok. Then what we do? We take the corresponding ASCII values of this a b x and y.

So, we take the ASCII value of a add them. So, ASCII value of b plus ASCII of x plus ASCII of y. So, if we add them then it will give me a number and based on that number I apply a hash function. So, h of number, so it can give me some index of the array index. So, this is just one example. So, is not mandatory that it has to be done like this, but this is just an example because the hash functions that we will consider, so most of the time they will be working with integer arithmetic. So, it is one way to convert one string variable into string name into an integer into an integer value.

So, mapping done using hash function that results in unique location in this, in the table organized as array and access time will order 1. Now imperfection of hash function result in several symbols mapped to the same location. So, this we have discussed in the last class. We have seen that if depending upon the hash function, so several keys may be mapped onto the same location. So, that is called a collision and we have to have some collision resolution strategy.

Now, it has been seen that for normally for this colo to keep the collision reasonable hash table is chosen to be your size between n and 2 n for n keys. So, it is we keep twice the number of keys that we are going to store. So, if you if you expect that the programs that a compiler will compile, will have about say 100 symbols, then you keep the hash table of size about 200.

So, you take a prime number which is just less than to a just less than 200 and then you can just you can apply the hash function.

(Refer Slide Time: 03:51)

## Hash Table

- Useful to minimize access time
- Most common method for implementing symbol tables in compilers
- Mapping done using Hash function that results in unique location in the table organized as array
- Access time O(1)
- Imperfection of hash function results in several symbols mapped to the same location – collision resolution strategy needed
- To keep collisions reasonable, hash table is chosen to be of size between n and 2n for n keys

$$p < 200 \quad h(x) = x \bmod p$$

Suppose so, I have got a prime number P which is we have suppose I have got a prime number P which is less than 200 and then I can make this h of x to be x mod P. So, in that case though they are only at most 100 symbols, but I am keeping it 200 because then even if there is a collision, so I will be able to so the collision will be less. Because, this P is a large number ok. So, it is expected that the collisions will be less.

So, there are several studies on how to design this hash suppose hash functions and also that these collisions are less and all. So, but that is beyond the scope of our discussion. So, I would suggest that you refer to some data structure classes for that hash table organization. So, for as a compiler designer. So, we are just going to use it for this symbol table organization.

(Refer Slide Time: 04:50)

The slide has a yellow header bar with the title "Desirable Properties of Hash Functions". Below the title is a bulleted list of four properties:

- Should depend on the name of the symbol. Equal emphasis be given to each part
- Should be quickly computable
- Should be uniform in mapping names to different parts of the table. Similar names (such as, data<sub>1</sub> and data<sub>2</sub>) should not cluster to the same address
- Computed value must be within the range of table index

Handwritten notes are present on the slide:

- A circle around the first bullet point with the text "abcd" above it and "xyz" below it.
- A circle around the third bullet point with the text "h" below it.
- A handwritten formula  $h(x) = x \bmod p$  with an arrow pointing to the "p-1" term.

At the bottom of the slide, there is a decorative footer with the "swayam" logo and other icons.

So, next we will be looking into some desirable properties of this hash function. So, first thing is that it should depend on the name of the symbol and equal emphasis be given to each part.

So it is like this that I have got the symbol name as say a b c d, I have got a symbol name say a b c d. Now the function that we have the hash function h that I am com computing, it should have equal weight age on all these parts a b c and d. Otherwise what will happen is that, these if 2 names if there is another variable like say a x y z. And if the maximum emphasis is on the symbol a on the first on the first character only of this string, then there is a chance that it will be they will be mapped to the same location by this hash function h.

So, that function h should be such that it gives equal emphasis to each part of the name. Should be quickly computable, so it should not take lot of time to compute the function, because our objective is to reduce the access time, and for all the accesses that we are doing to the table. So, it will be computing this function h.

So, it should be suppose quickly computable, so that is another desirable property should be uniform in mapping names to different parts of the table. Similar names should not cluster to the same address. So, it is like this that say this data 1 and data 2 this is an example. We have got 2 variable 2's identifiers data 1 and data 2. Now in these two identifiers only the last say last character is different. So, if I have got an h function the

hash function  $h$  should be such that this data 1 and data 2, they should be the match to they should be mapped to so different locations. There should not be matched to the same location.

So, this is known as clustering. So, clustering means if the names are similar. So, there the hash function it generates address which are also similar. So, that should not happen. So, we should we should if the hash function should avoid this clustering. And then computed value must be within the range of table index. So, as I said that we can have this modulo function, so  $h$  of  $x$  equal to  $x \bmod P$ . So, in this case the range is from 0 to  $P$  minus 1.

So, if you are using some other hash function instead of this modulus function which is pretty common. Then what can happen is that, the range of indices that is generated from this  $h$  function, so they do not match with the actual table index range ok. So, that way it can create difficulty. So, this is also another desirable property.

So, these are the desirable properties like as a compiler designer, we have to choose the hash function. So, we have to keep in mind all these points to come up with a good hash function.

(Refer Slide Time: 07:53)

## Scoped Symbol Table

- Scope of a symbol defines the region of the program in which a particular definition of the symbol is valid – definition is *visible*
- Block structured languages permit different types of scopes for the identifiers – *scope rules* for the language
  - Global scope: visibility throughout the program, global variables
  - File-wide scope: visible only within the file
  - Local scope within a procedure: visible only to the points inside the procedure, local variables
  - Local scope within a block: visible only within the block in which it is defined

So, next we will be looking into; so far we are looking into linear symbol table. So, where there is a single scope and then all the variables and the symbols and identifiers

they are visible to the entire program. But it is not the very suppose it is not a very common situation.

So, again most of the programming languages they are hierarchical in nature and at least they have got these block boundaries. So, we have got procedures and functions such that they are independent of each other and whatever variables you are defined within a procedural function. So, this is visible only within that procedural function. It is not visible outside. And similarly, there are some identifiers or some variables which are global in nature. So, that they are visible to the entire program. So, this way we can have different types of visibilities of identifiers in the program.

So, this they for these type of situation for this type of programming languages, so we you should have symbol table which is called as which are known as Scoped Symbol Table. So, this scope of a symbol it defines the region of the program in which a particular definition of the symbol is valid. So, if the definition is or so it is also known as the definition is visible.

So, so if I have got a definition say, in this in this program. So, suppose somewhere here I have got a definition like `i n t x`. Now, how far in the program this `x` is visible. So, if I say that this `x` is visible to the entire program. So it is visi it is seen, it is visible here. Like, if there is a statement like `y equal to x plus 5`. So, this `x` refers to this `x`. Or it may be so happen that , after this `x` has been defined from this point onwards only this `x` is important suppose `x` definition is visible.

So, if I have got a statement like `z equal to x plus 10`, then this `x` will refer to this, but this `x` will not in that case because, I said that it is visible from the point from where which it is defined. So, this `x` will actually correspond to some previously defined `x` here and this `x` will correspond to that, so as soon as you are coming to a new definition, so previous definition will be lost. So, that way we can have some scope rules. So, this is a very trivial scope rule that I have talked about.

So, normally programming languages they will define their own scope rule and accordingly we have to we have to come up with different symbol table organizations. So, that we can check those suppose check up the variable and identify definitions following the scope rules.

So, this block structured languages, they permit different types of scopes for the identifiers or this is known as scope rules for the language. One is called global scope, and the visibility is throughout the program, so that is the global variables. But almost all the programming languages they will allow these global variables, so they are visible throughout the program. Then there is File-wide scope, so visible only within the file.

So, maybe the same program is distributed over a number of files. May be for may be that there are different suppose people in a group who are developing the software for the same software parts of the software developed by different people in the group and they are defining their own files. So, if I define a variable in my file, so that is local to my procedures.

Similarly, somebody else defines another variable in their in that file so that it is visible only to procedures in that file. So, that is known as File-wide scope. So, this is not much meaningful because ultimately all these files are going to be combined. So, what happens to those combinations, but definitely the code generation and also they will be with respect to this the identifiers is defined within the file at which it is being used.

So, we have the File-wide scope is there. Then there is local scope within a procedure, so in that case it is visible only to the points inside the procedure that is the local variables. Particularly the local variables the scope is the local scope. And local scope within a block it is visible only within the block in which it is defined.

(Refer Slide Time: 12:27)

## Scoped Symbol Table

- Scope of a symbol defines the region of the program in which a particular definition of the symbol is valid – definition is *visible*
- Block structured languages permit different types of scopes for the identifiers – *scope rules* for the language
  - Global scope: visibility throughout the program, global variables
  - File-wide scope: visible only within the file
  - Local scope within a procedure: visible only to the points inside the procedure, local variables
  - Local scope within a block: visible only within the block in which it is defined

The slide has a yellow header bar with the title 'Scoped Symbol Table'. Below the title is a bulleted list of scope rules. A callout bubble points from the text 'Local scope within a procedure' to a diagram of a procedure block. The diagram shows a box labeled 'Procedure P' containing code with local variables 'x' and 'y'. The SWAYAM logo is visible at the bottom of the slide.

So, I can have local scope within a procedure, like I can have a situation like this. So, I have i can have a procedure P and there I can have some variable defines integer x y like that.

So, then this x y, so they are visible to the entire if this is the body of the procedure for this entire body this x and y are visible ; however, it may so happen that in this, there is a block here and within this block another definition of x comes. And so here if I refer to x, then this x will talk about this x. So, this is not that local x for the procedure. So, this is local for the block. So, this is the local scope within a block and. So, this x is not visible outside this block; however, the x that I have here. So, it may be visible everywhere within the procedures.

So, that way we can have certain set of rules for local scope within a procedure and local scope within a block. So, you can have different types of scope rules.

(Refer Slide Time: 13:39)

## Scoping Rules

- Two categories depending on the time at which the scope gets defined
- Static or Lexical Scoping
  - Scope defined by syntactic nesting
  - Can be used efficiently by the compiler to generate correct references
- Dynamic or Runtime Scoping
  - Scoping depends on execution sequence of the program
  - Lot of extra code needed to dynamically decide the definition to be used

So, we will be looking into these scope rules and how are they going to affect this programming language converter suppose code generation for different programming languages. So, there are 2 categories of scoping rules on that on based on that time at which the scope gets defined.

So, one is called static or lexical scoping another is called dynamic or runtime scoping suppose. So, this static or lexical scoping scope is defined by syntactic nesting. So, if I

have got something like this, suppose there is a block and , suppose there is a block and in that block I have got this suppose variables a a and b. And then, suppose so there is another block here and there I have got the variable integer x y here.

So, this syntax static or lexical scoping tells that as you are going by into this. So, whatever is defined in the outer block, so it is also visible to the inner block. But whatever is defined in the inner block, so that is not visible to the outer block. So, this x and y variables are not available at this point.

So, if I write something like x plus b something like that, so that that in that case compiler should give the error that x is undefined, because x is not defined in the current scope or the local scope of the this at this point. However, if I write say a equal to x plus b at this point, so that will be absolutely fine because this a and b so they will be coming from the outer block. So, from the outer block these variables are identifiers are defined and they are used they can be used in the inner block.

So, this is the scope defined by syntactic nesting. So, by using this nesting, we can define this scope. Now they can be used efficiently by the compiler to generate correct references. So, this is a static or lexical scoping, so this is much more easier for the compiler to handle, because it can check the situation and accordingly it can generate the references. Then, we have got dynamic or runtime scoping. So, here the scoping depends on the execution of sequence of the program.

So, naturally there are lot of extra code needed to be dynamically decide the definition to be used. So, basically in this case it depends on say, how we have reached a particular point. So, it is so previously. If this is a program and at this point of time, so if I have got a reference to a variable x ok. So, if it is a static or lexical scoping, so I just need to check the blocks. So, this blocks.

So, if I do not find x defined here I need to look into the surrounding block for x ok. So, that way, it can find the x value variable. However, in case of dynamic scoping, so it depends on the sequence by which we have come to this point. So, it, so may be that we have come through this path. So, accordingly this x that we are talking about will be we will have to search for x onto this path. Similar in some other execution if we have come to x where this path in the program then we have to search for into these procedures for a definition of x.

So, that way it is statically we cannot determine it. So, it has to be done dynamically. Only at runtime it will be say for as it will be may looking into this h differences, the different paths by which we have come to x this particular point and accordingly it will be finding out the correct reference. So, this dynamic of runtime scoping so that is there which is in some programming languages, you will find it, but of course, it is a very complex concept to implement ok.

So, this if we have to support this in the compiler generator it has to generate lot of extra code so that it can dynamically find out the reference during execution.

(Refer Slide Time: 17:46)

## Nested Lexical Scoping

- To reach the definition of a symbol, apart from the current block, the blocks that contain this innermost one, also have to be considered
- Current scope is the innermost one
- There exists a number of open scopes – one corresponding to the current scope and others to each of the blocks surrounding it

The diagram illustrates nested lexical scoping. It shows a code structure with three nested procedures: P1, P2, and P3. Procedure P1 contains P2, and P2 contains P3. Procedure P3 contains a reference to a variable x. A callout box points to this reference with the text: "Current scope of x is P3, it has another open scope P1". This visualizes how the scope of x is determined by the innermost block where it is used, but it also has an outer scope (P1) that is still active.

So, nested lexical scoping. So to reach the definition of a symbol apart from the current block, the blocks that contain this innermost one also have to be considered. So, suppose this is the situation. So, let us take this example where we have got a procedure P 1 and within this procedure P 1 another procedure P 2 has been defined and this procedure ends at this point. Then, we have got a procedure P 3 and in P 3, we have got a reference to x.

Now, where to look for this x? So, this procedure P 3, you see that it is nested within procedure P 1. Similar procedure P 2 was also nested within P 1, but procedure P 2 is over. So, when we are at this point, we have got 2 possible scopes for x. One possibility is that here itself in the procedure P 3 itself x is defined, so that is one possibility. And

other possibility is that, it is defined in the procedure P 1, so here somewhere here it is defined.

So, first it will look into this procedure P 3 definitions. So, if it can find this x within that, so it will be using that reference. And if it is not, then it will be looking into this P 1 scope because P 1 is the next higher level of nesting that we have. So, if we look into P 1's scope and accordingly it will try to find the reference.

So, to reach the definition of a to reach the definition of a symbol apart from the current block , the blocks that contain this innermost one also have to be considered. So, it is not that we look only in procedure P 3, but we also look into the other surrounding blocks.

So, current scope is the innermost one and there exists a number of open scopes one corresponding to the current scope and others to each of the blocks surrounding it.

(Refer Slide Time: 19:48)

Nested Lexical Scoping

- To reach the definition of a symbol, apart from the current block, the blocks that contain this innermost one, also have to be considered
- Current scope is the innermost one
- There exists a number of open scopes – one corresponding to the current scope and others to each of the blocks surrounding it

Procedure P1  
...  
Procedure P2  
...  
end procedure  
Procedure P3  
x =  
...  
end

Current scope of x is P3, it has another open scope P1

So, if there is another block surrounding it, so if this procedure P 1 was inside another procedure says P, and the procedure P ends here. Then this x so when you are talking about this x it has got an open scope x, a open so open scope P 3 P 1 as well as another open scope P. So, it will have 3 scopes in that case P 3, P 1 and P.

So, that way hierarchically it has to go for from one nesting level to the previous nesting level. So, till it comes to the outermost level, so all these pros all these nesting levels are available for probable scope.

(Refer Slide Time: 20:30)

## Visibility Rules

- Used to resolve conflicts arising out of same variable being defined more than once
- If a name is defined in more than one scope, the innermost declaration closest to the reference is used to interpret
- When a scope is exited all declared variables in that scope are deleted and the scope is thus *closed*
- Two methods to implement symbol tables with nested scope
  - One table for each scope
  - A single global table

So, in this way the nested lexical scoping is defined. And there are some visibility rules. It says that, it is that these are used to resolve conflicts arising out of same variable being defined more than once. And if a name is defined in more than one scope, the innermost scope the innermost declaration closest to the reference is used to interpret. So, this is actually the tiebreaking rule like it may so happen that we have got this thing. We have got say, we have got say 1 the x equal to something here and so this x is defined in this block as well as it is defined in some outer block.

So, this is also defined here, in some outer block. So, in that case, this its the rule says that for getting reference so it will refer to the local one. So, to the innermost one and if it is not available innermost one then only this outer one will be used.

So, that is the innermost declaration closest to the reference it will be used for the interpretation. And when a scope is exited, all declared variables in that scope will be deleted and the scope will be closed. So, this scope is exiting like say one procedure compilation is over. So, there, so all the local variables for that procedure so they are no more visible, so as a result they can be closed and we can have also this type of situation that one. So, we have compiling and one block is over. So, this is the we have we have we come to the end of a block.

So, at that point also. So, all the symbols that are defined in this block , so they can be wiped off and. So, that is the situation. So, that is called the closing of the scope. The scope is thus closed.

There are 2 methods to implement symbol tables within a state scope. 1 is 1 table for each scope, another is a single global table. So, both the approaches are used in compiler design with they have got their relative merits and demerits.

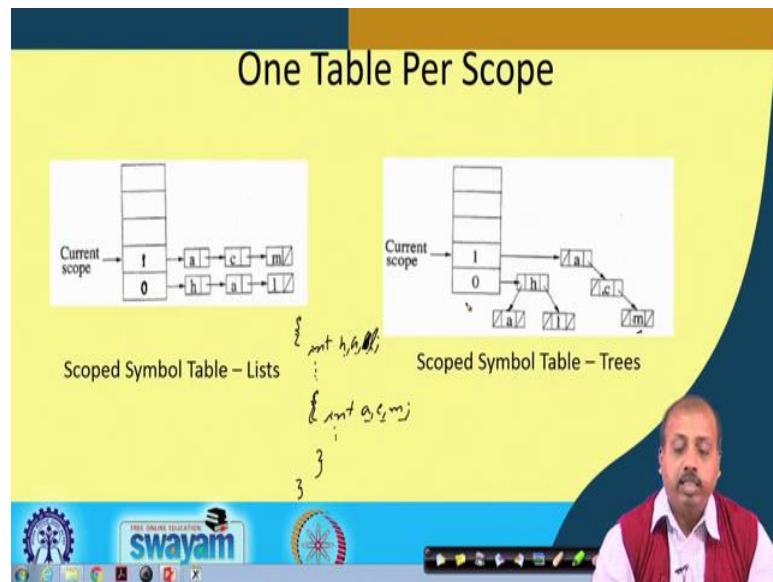
(Refer Slide Time: 22:35)

## One Table Per Scope

- Maintain a different table for each scope
- A stack is used to remember the scopes of the symbol tables
- Drawbacks:
  - For a single-pass compiler, table can be popped out and destroyed when a scope is closed, not for a multi-pass compiler
  - Search may be expensive if variable is defined much above in the hierarchy
  - Table size allotted to each block is another issue
- Lists, Trees, Hash Tables can be used

So, 1 table per scope it says that, you maintain a different table for each scope and a stack will be used to remember the scopes of the symbol tables.

(Refer Slide Time: 22:50)



So.. So, naturally, so we have got its got an example like say we have got a stack where this is the symbol table. Symbol tables are organized in the form of a list, so this is h and a. So, these are two symbols at h a and l. So, these are three symbols in the scope 0. So, this example that we are talking about it is something like this say, we have got this say integer h a l.

And sometime later, we have got another brace where we have got integer a c m. Now for this, so when I am, when I am in this particular block. So, I had to create a symbol table where this h a and l, so they are put on a list. And we use a stack, so we will note down the level of the nesting and the nesting level is 0. So, this h a and l are there. Then, after that we find this another brace. So, the nesting level becomes equal to 1.

So, a nesting level is implemented, it becomes 1. And then we find this symbols a c and m, so they are put on a list so making the symbol. ma making the symbol table for this one. And there, this nesting level is the nesting level is equal to 1. So, that is noted here. So, this is the situation one table per scope under linear table organiser under list type of organization.

So, you can have 3 type of organization also like, so individual tables that we have got. So, they are organized as trees like this h a l. So, they are organized as a tree. So, h is at the root, so a is less than h. So, it is on the left side and l is more than h, so that is on the right side.

Similarly, this a c m, so is at the root, c is greater than that and m is further greater than that. So, that forms a table, so accordingly this stack that we have so they contain the scope definitions. The current scopes, so they it is the scopes 0 and 1. And there is a current scope pointer that points to the current scope that we have. So, this way we can think about this scope symbol table.

So, , so we can have, we can maintain a different table for each scope and a stack will be used to remember the scopes of the current say of the scopes of the symbol table. So, when the current procedure or current function is over. So, we can just get rid of these thing like if this current scope is over, then this table can be deleted and this stack pointer can be decremented. This current scope pointer can be decremented. So, that it points to the next innermost scope.

So that way, we can do this thing. So, the there are some drawbacks of this particularly for single pass compiler. So, table can be popped out and destroyed when a scope is closed, but not for a multi pass compiler. So, for a single pass compiler means that creates the symbol table and does code generation at the same phase so or single in a single pass it does both.

But there are some compilers which now which do it in 2 phases. In the first phase, it makes the symbol tables calculates all offsets and all for all the symbols. So, that code generation part becomes simple. In particular, what happens is that in if in a programming language, we have got the we have got this type of facilities that I can refer to some variable x here and this x will be defined sometime later in the program.

So, that in that case, if I have got a single pass compiler then the difficulty is when you are here. So, you do not know the size of x ok. So, that way you cannot generate a proper code. So, in a two pass compiler what will happen is that, in the initial in the first stage, so it will ignore all these statements. So, it will just look for these type of definitions in the program. And accordingly, it will make the symbol table. It will definitely calculate the offsets like getting these type of statement, so it will calculate the size of those statements and it will accordingly come it will compute the offsets of all the statements. So, that it when it comes to this line. So, it knows it knows the offset of x.

So, that way at the end of phase one, so we have got only the symbol table constructed and in phase two the actual code generation will be done. So, that is a multi-pass

compiler. So in case of single-pass compiler, as soon as we are through with a procedures, so we can forgive forget the corresponding symbol table. So, the it can be popped out and destroyed. However, multi-pass compiler that is not the case, because now we have to remember all the individual compiler individual symbol tables that we had made.

So as a result, search may be expensive if variable is defined much above in the hierarchy. So, because now we have to search number of tables are to be searched. Like you see, that in this table, in this table say in this table. So, if you are looking for say the symbol h ok. Then, first if you look into the current scope so, it will scan through all this one and then it will come it will not find h here, so it will come to this level 0 and then it will be searching for this h. So it will get it will be getting h here.

So in the worst case when the symbol is not present in that, in any of these tables then what will happen? The all the table entries are to be searched to tell what is or where is it. So that wayit is difficult because the overhead becomes more. So, it has to search through a number of a number of places ok. The number of tables for getting the offset.

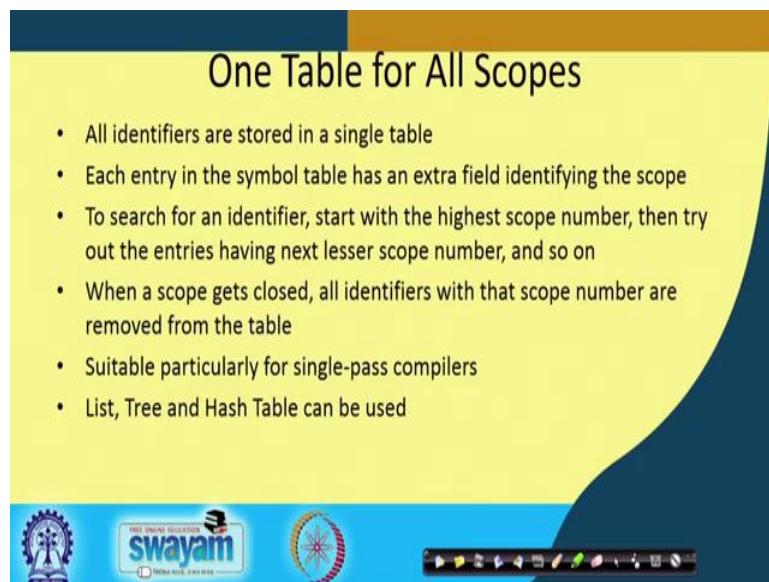
So search is search may be expensive. Table size allotted to each block is another issue, because how big a table you will be assigned to individual blocks. So if you are if the size is not is not estimated properly, then we may either can give it a small number or a large number of cells as a result the table may be underutilized or it may be that table is not sufficient. So all these data structures like list trees and hash tables can be used for this one table per scope type of organization.

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

**Lecture - 46**  
**Symbol Table Runtime Environment**

This one table per all scope type of organization; so, we can have different table organizations as we have seen previously.

(Refer Slide Time: 00:23)



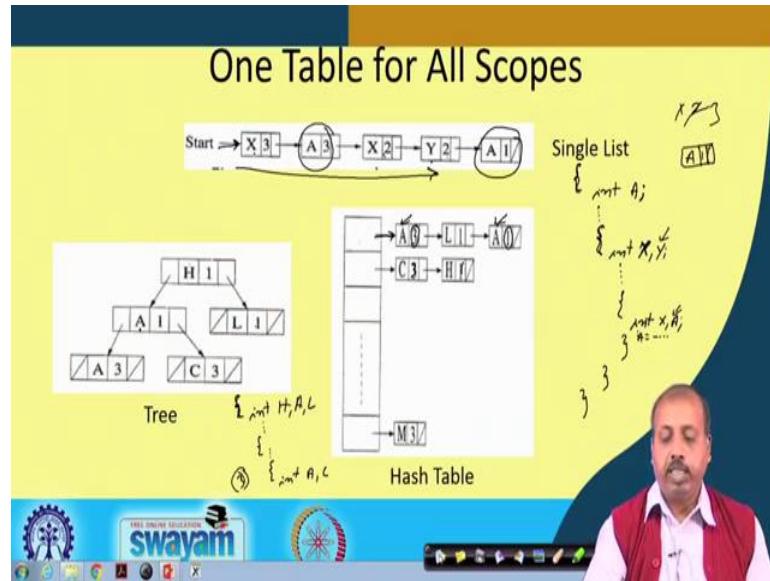
**One Table for All Scopes**

- All identifiers are stored in a single table
- Each entry in the symbol table has an extra field identifying the scope
- To search for an identifier, start with the highest scope number, then try out the entries having next lesser scope number, and so on
- When a scope gets closed, all identifiers with that scope number are removed from the table
- Suitable particularly for single-pass compilers
- List, Tree and Hash Table can be used

The footer features the 'swayam' logo with the text 'FREE ONLINE EDUCATION' and 'SWAYAM' in a stylized font, along with other small icons.

Like it can be an array, it can be a list, it can be a tree, or hash table, so any of them can be used. So, we will see that how they can be used for that purpose. So, all identifiers are, so if I have got one possibility is that we have got one table for all scopes. So, we have got a single table and that suffices, so that is the made for all scopes together. So, all identifiers will be stored in a single table ok. And each entry in the single table has an extra field identifying the scope.

(Refer Slide Time: 00:57)



So, let us take an example and then try to see. Like here, if I are having a list type of organization of the symbol table; so this is X, A, X, Y, A. So, these are some symbols defined in the program and we are storing the corresponding levels. So, the scope number, so in this pa so this particular example, so the situation is like, this that we have got one outer block where I have got this integer A. And then within that, so we have got another block where it is this X and Y are defined, integer X, X and y. And that after that, there is another block within this where this A and X they are defined; integer X and A, so they are defined; so, this is the nesting.

So, what happens is that, when you have got this one. So, at this point, so this is, so previously my level was 0. Now this, I have found this integer A and, so as soon as we find this brace; so level is incremented by 1. So, this scope level the current scope is made equal to 1. With that it comes to this point and it gets a symbol A. So, this A is put into the symbol table A and it is the corresponding scope number is also noted. So, that is why it is A and 1.

So, this block is this box is created you can say in the symbol 2. So, it has got only 1 entry now. Then it goes forward. Then after some time it finds that there is another brace; so, this current scope is incremented to 2. And then it finds there is 2 new definitions X and Y, so they are put into the symbol table like this and then, their corresponding types are corresponding scope levels are 2. So, the these 2 this a scope

number is stored there. Then after some time it comes to another brace. So, this scope is incremented to 3 and this X and A. So, it that there will be referring to this X and A; so they will be stored here this X and A.

Now in this code, so if there is a reference to say A equal to something. In that case, it will start looking from this point and it will try to see where this A occurs first. So, the A occurs first at this point, so it will be, it is referring to this particular A.

Similarly, if it is referring to Y, then it starts from this point searches for Y and it gets the Y at this point; so it is taken as Y. So, that the so it is taken as this Y at level 2. So, this way we can have one table for all scope. Similarly for a tree type of organization, so it may so happen that that at this H, A and L. So, they are, so we have got these symbols H A and L in the top level. So, at this level we have got say integer H, A and L. Then after sometime we have a brace here there is no declaration, but after some time again there is a brace and there I have got this integer A, C.

So it may be like this and then we can see that. So, this, so as it is coming to the third brace. So, this at this point the scope number is equal to 3. So, in tree organization also well apart, apart from the name of the symbol so we are also storing the corresponding scope number. So, these are 1 for H, A and L and 3 for A and C.

So, you can also have a hash table organization; so, otherwise it is remain same. So, this key values A, L, C so they are used for doing the hash for applying on the hash function. So, the hash function is so since these two, since these two identifiers both are A their names are A. So, they are matched to the same location. So, we are using a chaining type of approach for this collision dissolution and you also store the corresponding levels ok; so corresponding scope level that is 3 and 1.

L is also mapped here because my it is assumed a it may be the situation the hash function is such that that A and L they map to the same location. Similarly C and H they map to the same location. So, this C, so we also store the corresponding level, the scope level 3 and 1 with C and H and similarly at this point for M stored.

So, in this way we can have a single table for all scopes and that way we can organize the symbol table ok. So, the advantage is that we do not have to have many many symbol tables and it is at a common place you can say. Now whether it is convenient or not, so

that is dependent on the compiler designer, so there is not much choice in fact. So it is up to the compiler designer to decide like which one is more comfortable with him.

(Refer Slide Time: 06:24)

The slide has a dark blue header with the word 'Conclusion' in yellow. The main content area is divided into two sections: a yellow section on the left containing a bulleted list, and a video player on the right showing a man speaking. The video player includes a logo for 'SWAYAM' and other interface elements.

- Symbol table, though not part of code generated by the compiler, helps in the compilation process
- Phases like Lexical Analysis and Syntax Analysis produce the symbol table, while other phases use its content
- Depending upon the scope rules of the language, symbol table needs to be organized in various different manners
- Data structures commonly used for symbol table are linear table, ordered list, tree, hash table, etc.

So, next we will be looking into the come to the conclusion of this step this discussion.

So, symbol table, so though not part of code generated by the compiler, it helps in the compilation process. And phases like lexical analysis and syntax analysis produce the symbol table; while other phases use its content. And depending upon the scope rules of the language, symbol table needs to be organized in various different form different manners.

Data structures commonly used a linear table, ordered list, tree, hash table etcetera. So, and as I said that there can be other data structures also and we can go for those data structures. So, depending upon the programming language, so the compiler designer may use some innovative data structures. And this is entirely up to the compiler designer. So, with just by consulting the lexical rule or scoping rules of the language, can come up with different type of organization for the tables.

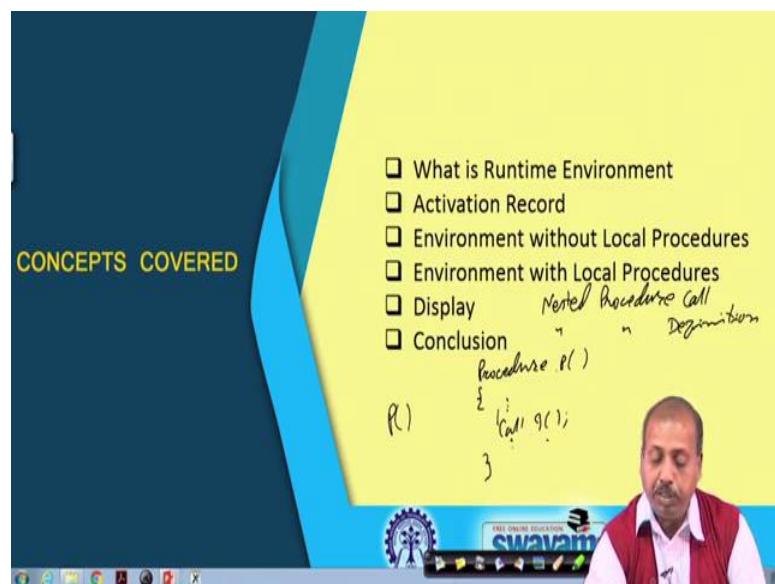
So, we complete this part on this discussion.

So, next we will be looking into the; this how can we organize the runtime environment for this compiler; for in the code generated. So, runtime environment what it means is that, while running a program we need to maintain certain information. For example, you

are calling a procedure. So, when a procedure is being called we need to pass a number of parameters. Similarly when we are returning from the procedure then the return address and the return value, so those are important things, so they are to be taken into consideration and they should be put into the proper places.

So, these are some important points and then this, in this runtime environment management part. So, it will do something, so that the compiler puts sufficient amount of code for this runtime management of the system.

(Refer Slide Time: 08:25)



So, the topics that we are going to cover are like this. That, what is runtime environment? Then there is a concept called activation record. So, activation record is a collection of information which are important for this procedure call and return.

And we will see that there is well defined set of rules between the procedure which is calling a sub procedure. And similarly, there is also some part of some responsibilities on the called procedure, both at the time of going to the going to the procedure and coming back from the procedure. So, there are well defined actions to be taken by these individual modules.

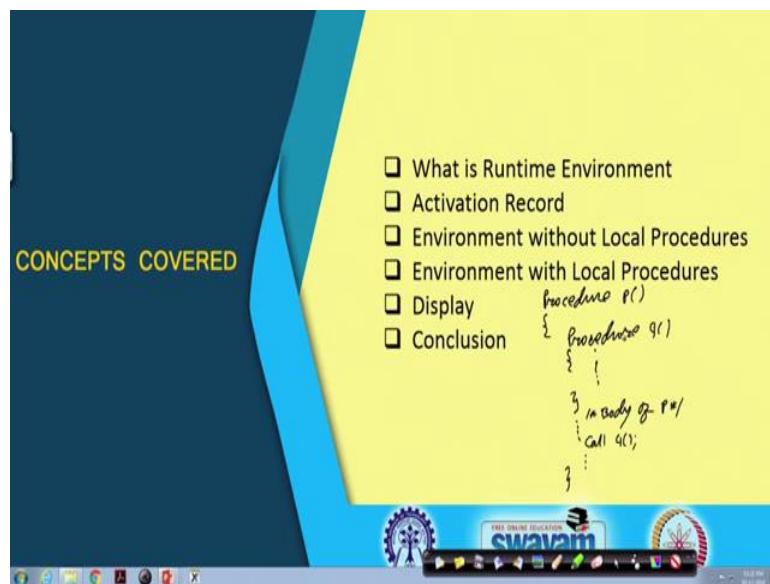
So, we can have environment without local procedures and there can be environment with local procedures. Some programming languages are such that we apply it is

allowing us to have nested definition of procedures. So, it is there are two situation, one is called nested procedure call another is nested procedure definition.

So, one is nested procedure call another is nested procedure definition. So, they so, when I say nested procedure call; so, this is very common. So, almost all programming languages they will allow you to do something like this that. Suppose this is a procedure P, so this is a procedure P and then from this body of the procedure you call another procedure q, so that is the procedure that is the nesting of procedure. So, or nest a nested call of procedure. So, from P, so from the main program you have given a call to P and while executing P it is giving a call to q. So, the q is called within the procedure P. So, that is the nesting of the procedure. Nesting nested call of the procedure.

Now, it may so happen that in the procedure P I define another procedure.

(Refer Slide Time: 10:38)



So, I can have say I have got a procedure P and within this we define another procedure q. So, we define another procedure q. So, this is the procedure q that we have. And after that, the body of P starts. So, from this point I can say this is the body of P. Now here you can have a call to q. You can have a call to q, but can I have a call to q from outside the procedure P?

So, some programming languages still that you can do that. Some programming languages will tell you, no you cannot do that. Some programming languages will even

say that you have to call procedure P once before you are going to call procedure q; so, that is also there. So, different programming languages they have different types of semantics for these local procedures.

So, this q is a procedure which is local to the procedure P, so this runtime environment for procedures with for situation with local procedure is different from situation without local procedure. So, naturally, with local procedure the situation is going to be difficult, because all these scope rules of the language, so they will come into play. And accordingly they will be giving the types of identifiers and these types of procedures and all that is there.

Then there is a concept called display. So, displays for making this runtime execution faster ok, so how to do it fast? So, that will be having some special data structure called display. So, that is put into the runtime environment, so that it will be doing it first. So, we will see all these concepts and finally, we will draw conclusion on this thing.

(Refer Slide Time: 12:43)

The slide has a yellow background with a blue header bar. The title 'What is Runtime Environment' is at the top. Below the title is a bulleted list of definitions and components of the runtime environment. To the right of the list is a hand-drawn diagram showing three vertical columns representing memory segments: 'Code' (with a small icon), 'global' (with a box icon), and 'Local & arguments' (with a stack icon). At the bottom of the slide, there is a watermark for 'swayam' and a video player interface showing a speaker icon and other controls.

- Refers to the program snap-shot during execution
- Three main segments of a program
  - Code
  - Static and global variables
  - Local variables and arguments
- Memory needed for each of these entities
  - Generated code: Text for procedures and programs. Size known at compile time. Space can be allotted statically before execution
  - Data objects:
    - Global variables/constants – space known at compile time
    - Local variables – space known at compile time
    - Dynamically created variables – space (heap) in response to memory allocation requests
  - Stack: To keep track of procedure activations

So, how are you going to go through the discussion? So, first we try to answer this question that, what is runtime environment? So, runtime environment it refers to the program snap-shot during execution.

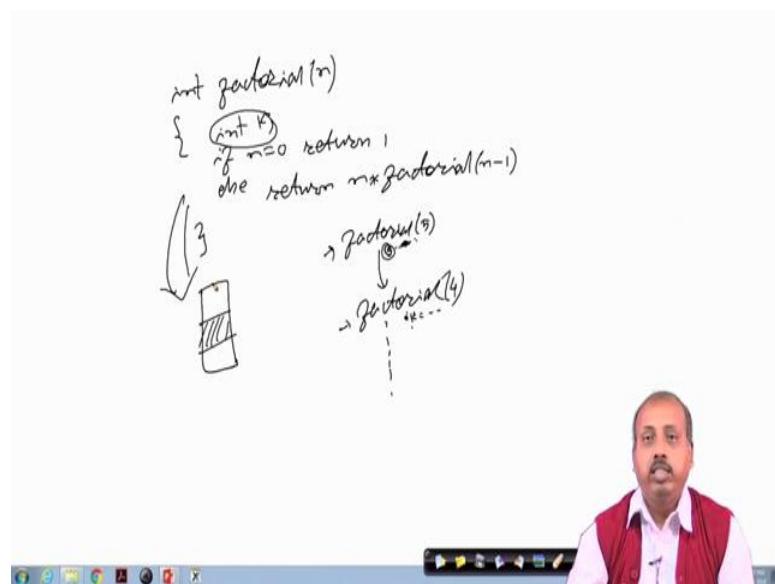
So, program snap-shot means, the program is running and as if I take a snap-shot of the program as it is running. Now you see that a program may be very small, when it is

residing in disk; so, it may be the object code that is produced is it may be very small. So, it may have only a few lines. But when it is executing in the memory in the computer, so then this; the snap-shot may be very large. Why? Because any program that you have, so it has got three portions in it.

So, this body if you look into so it can be divided into three parts, one part of it is called code, another part is called the static or global variables, and another is local variables and argument. So, these are the three parts. If you look into this piece of program, so you can find that there are there so the there so it is the code. But when you take the snapshot then this snap-shot can be divided into 3 portions. Some part of it is code, some part is corresponding to the global variables, and some part of it is corresponding to the locals and arguments.

So, the static variables, so they are also similar to global variables only, because they will be giving they are treated in a global fashion only, so they are treated; so, static variables and global variables, so they are taken together. Now what can happen is that, suppose I am writing a function which is say suppose I am writing a function which is say factorial function.

(Refer Slide Time: 14:42)



So, this factorial function, so I can write it like this. Int factorial in and then I can say if n equal to 0 then return 1 else return n into factorial n minus 1.

So, this is the piece of program that I have. Now suppose, so when this is there in that disk as the object file. So, I have got the translated version of this program here. So, this is the translated version. And here, so now, what will happen is that when this program is in execution, then what happens is that; suppose I have given a call to factorial 5. So, this factorial 5 has in turn given rise to factorial 4 ok. So, it has given rise factor then factorial 3 like that.

So, what happens is that at a runtime you can understand there can be several instances of this function that are available. Where when it was in the disk you see that I have got only a part of the code which corresponds to this factorial function. But while I am executing the program, so there are several copies of the factorial program that are running. So, naturally, so they I they are not same because the variables that this function is computing are not same as the variables that this function is computing.

The situation will more become more difficult if I have some local variable say K here. Then the K that I am referring to in this function factorial 5 call, so that K and the K that I have in factorial 4, so this K, so these two K's are not same. So, they must be two different K's. Otherwise if this factorial 4 functions modifies this K; so then this K will get affected, so that is not desirable. So, this all the local variables that I have, so there has to be separate copies for them, so, this is actually the challenge.

So, whenever you have got this execution going on. So, you get a snap-shot of the program and this program snap-shot is not the same as the original say object file copy that you have. And then there are some, so there are more things that we have then how are you going to manage this situation. Like how are you going to have so many invocations of this same function. Or if there is a nested call, then how are you going to take care of the nested call. So, that is that is the challenge of this particular part of discussion.

So, this refers to memory snap-shot during execution there are 3 main segments of a program, the code segment, static and global variables, and local variables and arguments.

(Refer Slide Time: 17:53)

The slide has a yellow header with the title 'What is Runtime Environment'. Below the title is a bulleted list of points:

- Refers to the program snap-shot during execution
- Three main segments of a program
  - Code
  - Static and global variables
  - Local variables and arguments
- Memory needed for each of these entities
  - Generated code: Text for procedures and programs. Size known at compile time. Space can be allotted statically before execution
  - Data objects:
    - Global variables/constants – space known at compile time
    - Local variables – space known at compile time
    - Dynamically created variables – space (heap) in response to memory allocation requests
  - Stack: To keep track of procedure activations

On the right side of the slide, there is a hand-drawn diagram illustrating memory segments. It shows a stack at the bottom with 'int x;' written above it. Above the stack is a section labeled 'main()' containing local variables 'a', 'b', and 'm'. At the top of the diagram is a section labeled 'f1()' containing local variables 'x1' and 'y1'. A bracket on the left groups 'x' and 'main()' under the heading 'int x;'. Another bracket on the right groups 'f1()' under the heading 'int x;'. There is also a handwritten note 'param' next to the 'f1()' section.

A man in a red vest is visible on the right side of the slide, likely the lecturer.

So, if I have got a C program, so suppose I have got something like this. That, I have got an integer variable  $x$  and then, so this is the C program only, then there is a main function. And in the main function, I have got some local variables  $a, b$  etcetera. And it is some some point of time it is giving a call to function  $f1$  and here I have got the function  $f1$ . And there I have got the local variables and also  $int m, n$  like that.

So, if you take the snap-shot of this program; suppose at when the program is executing at present  $a$ , if the main routine has given a call to  $f1$  and we are at this point of execution of this function  $f1$ . Then, I can say that the program that I have in the memory now, in the main memory that the snap-shot of the program that I have now.

So, it can be divided into 3 parts. One is the code part, code part is actually the translated version of all these statements that I have. All these high level statements, so if you translate into machine code. So, what is it? And the code part is not going to change; so, it is going to be static in nature. So, it is not going to change. Like then at this point I have got a variable  $x$ , so that comes to the global variables section. So, we have got this code, then we have got the global variables. So, this variable  $x$  will be assigned some space on to this section. And all these variables like  $a, b$  then  $m, n$ , so they will come as local variables and that local variable are to be handled and variable and arguments are to be handled.

So, this way any may any program while it is executing it can be thought about consisting of 3 segments, code, static, and global variables, and local variables, and arguments. And memory needed for each of these entities because code will need space, global variables will need space, and this local variables will also need space; so all of them will need space.

Now for the code, so text for a code will consist of text for procedures and programs and the size will be known at compile time. Because when it doing the compilation, so every high level instruction is converted into a set of machine level instructions, so the compiler generate the compiler will know like what is the size of this individual instruction. So, accordingly it can compute what is the size of the code. And as a result, I can know I can allocate this space allow statically before the execution starts. So, this memory can be allocated for this code part.

Now, for the data object parts. So, global variables are constants. So, here also space is known at compile time, because there is only one copy of those global variables. So, they can be done at compile time; however, local variables again local variables also the space is known at compile time. So, how much space is needed that is known. And there can be some dynamically created variables because, so most of the programming languages, so they will allow some dynamic memory allocation, dynamic memory policy for example, this C programs they have got this function malloc.

So, you can write like, so if I have got an integer pointer P. So, I can write like malloc something like this. Now this P, so the this space that you get is it was not there with the program at the beginning right. This variable P was there with the program at the beginning, so the compiler could allocate space for that, but for this dynamically allocated memory the space was not available.

So, this space is in response to memory allocation request. So, that is called for that is therefore, the dynamically created variable. So, that is the space is called heap space that is there. And there will be a stack to keep track of the procedure your activation, so which procedure is called and all where from we are returning. So, this return address, return value they are to be stored; so, they will be kept in the stack.

So, this memory needed for this entries like, we need some code space, we need some space for this global variables, local variables and these dynamically created variables.

And also to keep track of the procedure activation, so we need to keep some stack. Now, how are you going to do that? So, that will be the challenge.

(Refer Slide Time: 22:38)



So, we do it like this. So, this code is so, if you take this as the memory space that we have allocated for a particular program. The lower end of the space is allocated to the code part and the code size is known; so it will go up to some point. Then for the static variable or global variable, so the size is again known. So, they will be coming under the static part; so this they will come.

And then the remaining part of the memory, so they are allocated simultaneously to heap and stack, so this heap will grow from the low side and the stack will grow from the high side. So, heap will go from low to high, stack will grow from high to low because if a program has got more of dynamic memory allocations, so it will need more of heap space. On the other hand, if a program needs more of procedure calls, if there are a lot of nested procedure calls, then this stack will be a stack requirement will be high. Because for every such call you will see that some space is needed in the stack. So, since we cannot predict like how the behaviour of a program.

So, whether it is going to take more of dynamic space or more of procedure calls. So, we divide it we allocate the same space for the heap and stack. So, code occupies the lowest portion then the global variables are allocated in the static portion and the remaining portion of the address space that is in the stack and heap they will be allocated from the

opposite ends to have the maximum flexibility. So, it can very well be done that you allocate some space for heap and the remaining space for stack, but the flexibility will be less. So, to maximize the flexibility, so we do it like this.

(Refer Slide Time: 24:24)

**Activation Record**

- Storage space needed for variables associated with each activation of a procedure – *activation record or frame*
- Typical activation record contains
  - Parameters passed to the procedure
  - Bookkeeping information, including return values
  - Space for local variables
  - Space for compiler generated local variables to hold sub-expression values

Now, what is an activation record? So, activation record is the storage space needed for variables associated with each activation of a procedure. So, there that will be called an activation record or frame; so this is some storage space.

For every call that we make to a procedure, so some of the some space will be needed for creating the parameters that you are passing, creating the local variables and all, so all these information. So, this is actually some extra that you that are coming into picture; so that will be required. So, that will be done that will be made in the activation record. And this activation record itself may be created at different places. So, it may be created statically, it may be created dynamically, it may be on heap, may be on stack like that. But they will be they will be required for each such call they will be required.

So, typical activation record contain are like this. So, parameters passed to the procedure then the bookkeeping information keep including return values, space for local variables, and space for compiler generated local variables to hold sub expression values etcetera.

So, it is like this, that suppose I have got a procedure call at this point. So, this I write like  $x$  equal to some procedure  $proc$  1 is called, it has got some parameters  $a, b, c$  passed

here. And this procedure proc 1, so it has got say it has gotten this corresponding arguments m1, m2, m3 and it has got some local variables. So, p, q, r are the local variables.

Then what happens is that, when you are making this call somehow I need to remember that what are the values I am passing this a, b, c what are the values I am passing, so that has to be remembered. Then this return address has to be remembered. Like after finishing this procedure 1 where are you going to come back, that return address has to be remembered. And within once you are in the inside the procedure then you need to have some space for this p, q, r this local variables and it may there may be some return value of from the procedure.

It may be returning the value of p plus q from the procedure, so the return value has to be there should be some space for that. So, this will happen at runtime, but the compiler needs to allocate space, so for this thing to occur ok. So, it needs to have space, so that while the code will be executing it will be keeping all those information in those spaces, so that the program will execute properly.

And not only this, so like when it is doing the translation of these individual statements that we have in this procedure of proc 1, so it may generate some local variables. Some, some expression can generate some local variables. As I was telling, like if I if there is a statement like y, x equal to y plus z into w, so it is for this the code that is generated has got something like this t1 equal to z into w, t2 equal to y plus t1 and then x equal to t2.

So, this t1 and t2, so these two are local variables, so that are the call that they are generated by the compilers; they are not there in the original statement, but they are generated by the compiler in the code translation.

So, they are this compile this temporaries they are also to be stored in the activation record. So, all these are the extra information that we need to store while a procedure call is made and they will be made part of the activation record.

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

**Lecture – 47**  
**Runtime Environment (Contd.)**

(Refer Slide Time: 00:17)

**Location for Activation Record**

- Depending upon language, activation record can be created in the static, stack or heap area
- Creation in Static Area:
  - Early languages, like FORTRAN
  - Address of all arguments, local variables etc. are preset at compile time itself
  - To pass parameters, values are copied into these locations at the time of invoking the procedure and copied back on return
  - There can be a single activation of a procedure at a time
  - Recursive procedures cannot be implemented

*Diagram illustrating parameter passing by value:*  
A hand-drawn diagram shows a procedure call `p1(x,y)`. An arrow labeled `proc p1(x,y)` points to a stack frame. Another arrow labeled `Call by value-read` points from the stack frame back to the call site.

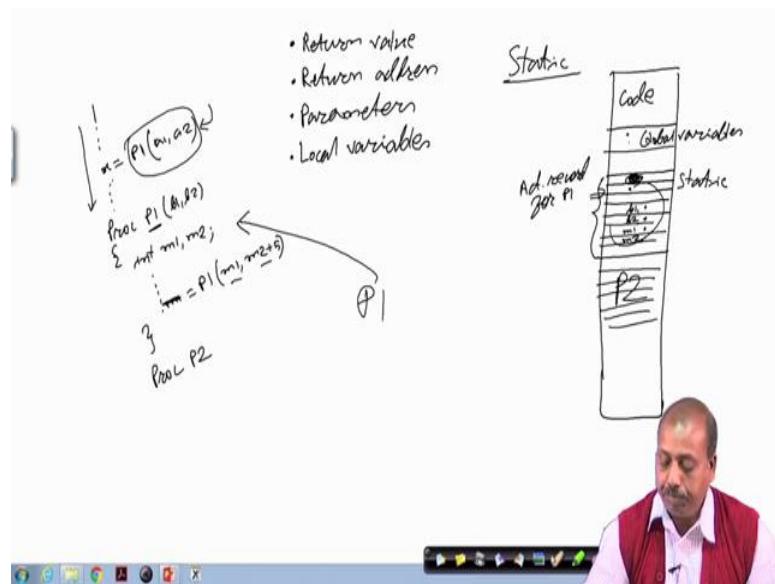
*Video frame:*  
A video frame shows a man with a mustache, wearing a red vest over a white shirt, speaking. The background includes a blue banner with the text "FREE ONLINE EDUCATION" and "swayam".

So, location for activation record. So, where do you create this activation record? So, that is a question. Now, depending upon the language that you have, because the activation record can be created in the static stack or heap area. So, these are the three options that we have, where are you going to create this activation record. So, if it is created in the static area that is the global area that we have so, I like you we have seen that a program can have the global variable area.

So, where this, that is called the static area. So, that can apart from the static the global variable. So, it can also have this activation records the area for activation records. Now, this early programming languages like FORTRAN, they used to have this activation record created in the static area. So, so this address of all arguments local variables etcetera or p set at compile time itself and two pass the parameters values are copied into this locations at the time of invoking, the procedure and copied back on return.

So, there can be single activation of a procedure at a time and recursive procedures are cannot be implemented. So, let us try to understand what do you mean by that.

(Refer Slide Time: 01:31)



So, I said that activation record it has got so, it has got the entries like this return value, then return address, then the parameters that you are passing and the local variables local variables.

Now, what this static allocation we will do. So, we are looking to the static allocation what the static allocation we will do. So, it will fix up some some entries ok, like if in a if I got a program where at some point of time I am giving a call two procedure  $P1$ . So, this is say  $x$  equal to say  $P1$  and then I am passing the values like say  $a1$  and  $a2$  has two arguments for this. And then this is the procedure  $P1$  and there I have got the local variable say  $m1$  and  $m2$ .

Now, so if this is the code that is generated, there is this is the translated version of the object code that is generated. So, with the some part of it is corresponding to the code of the program and say this part of this file the object file is corresponding to the static. So, this is called the static location that is the global variables are allocated here.

So, some part of it will be occupied by the global variables and some part may be allocated for this activation records, like by analyzing this program you know that at this point this procedure  $P1$  is going to be called so, we your mark a portion within this static area to act as the activation record for  $P1$ .

So, this part is your marked to be the activation record for P1 that is whenever in my program it is any from any point in the program, if I am going to make a call to P1 procedure then these entries will be filled up here.

So, the first slot maybe for the return value second slot may be for the return address , then there are two parameters that I am passing so, this is for a so, this has got two values say a1 a2 let us call them b1 b2. So, this is the space for b1 this is the space for b2 and this is the space for m1 and m2. So, like that it has got so this part of this memory so, it is your marked two act as the activation record for P1.

Now, what the compiler we will do at this point when it is generating the code for this procedure call so, it will have the code such that this e1 is copied into this location a2 value is copied into this location fine, then the return address is saved on so, the return address is return this space is reserve for return value this space is saved for the return address. So, this way we can do this thing then this m1 and m2 so, they are for the local variable. So, when it comes to the local when it comes to this code. So, when it is doing a translation of this m 1 some modification of this m1 m2 so, they will be effecting these locations m1 m2 like that.

Now on return so, what will happen it knows the compiler knows that the return value is available at this at this particular address. So, it will be coping it to x director. So, for this the for this call the return value will be available at this location for. So, for this x equal to this P1 etcetera so, it will be copying the content from this location and putting it into x. So, that is how this static things we will work ok.

Now, for every procedure that you have in your program. So, is there maybe another procedure P2 in the program. So, there will be another such frame created in the static area which will correspond to P2. So, for every procedure called they have a for every procedure that we have in your system so, there will be a there, there will be an activation record created. And whenever a call is made to that particular procedure this record will be filled up by the calling procedure, and at the manipulations will be done on this variables on this on this activation record only.

And after coming back, this return value is again available in the activation record. So, from there it will be taken. So, this is how the static thing will work, now the problem that we have is that see we cannot have more than one activation of P1 simultaneously,

because in that case if there is an somebody else from some other place is also allowed to call P1, when this call is active. Then what will happen is that this P1 will get activation record will get overwritten, as a result the execution will become erroneous.

So, apparently it seems that why they should happen because ultimately we have got a single processor so, as a result. So, there can be only one call at a time, but there can be situation like this P1 maybe a recursive procedure. So, within this their this P1 maybe giving a call two P1 itself with some parameter say m1 and m2 plus 5. Suppose it does something like this then what will happen so, then this m1 and m2 plus 5 these two values will be copied on to b1 and b2. So, previous b1 b2 contains will be lost. So, it cannot return to the proper place so, this all calculations will be lost.

So, that is why this static type of organization. So, of this activation record though it is very simple in the sense that, we know the size of this static section ok. So, it is very simple, but it cannot handle complex situations by like say recursion. So, this is say a recursive is a P1 being a recursive procedure. So, it cannot handle that situation. So, for the static allocation of activation record. So, it only one active only one procedure call should be active at a time.

So, coming back to the point that we were discussing. So, it was there in the early programming languages like FORTRAN ok. So, this is being FORTRAN you do not have recursion. So, there it was possible so, address of all arguments local variables etcetera are they are computed at the compile time itself and their we fix some locations for them and the code is generated such that, whenever a procedure call is made the code is generated such that the parameters that you are passing. So, they will be copied onto this activation record space.

And after it is over so, they will be copied back from this activation record to the actual variables. So, that way if there is a change in the values of those variables within the execution of the procedure. So, that will get reflected. So, what I mean is that, suppose I have got this as the activation record for procedure P1, now I am giving a call to this procedure P2 at this point in the main program so, I am giving a call to the procedure P1 with the values like a and b.

So, whatever arguments we have so, this values will be copied there. So, this value of a will be copied into if this P1 has the procedure P1 has got the arguments x and y suppose this is for x and this is for y.

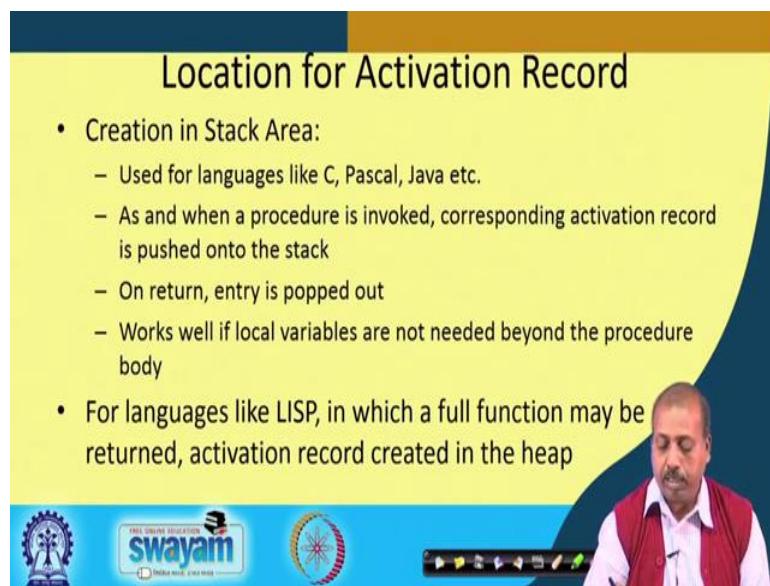
So, at the beginning the value of a will be copied on to x and value of b will be copied into y. And once the procedure is over then this x and y values are now modified due to execution of this P1 and then this x value will be copied back to a and this y value will be copied back to b so, this is a type of parameter passing which is known as call by value result. This is call by value results at the time of calling the procedure the values are copy to the parameters and at the time of returning the parameters are copied the final values of the parameter so, they are copied back to their original variables.

So, this way so, they be copied back on detail. So, this is call by value result type of parameter passing. And as I discussed previously so, there can be a single activation of a procedure at a time, because multiple activation we will destroy the activation of this activation record content of the previous call and so, and recursive procedures definitely cannot be implemented ok.

(Refer Slide Time: 10:57)

### Location for Activation Record

- Creation in Stack Area:
  - Used for languages like C, Pascal, Java etc.
  - As and when a procedure is invoked, corresponding activation record is pushed onto the stack
  - On return, entry is popped out
  - Works well if local variables are not needed beyond the procedure body
- For languages like LISP, in which a full function may be returned, activation record created in the heap



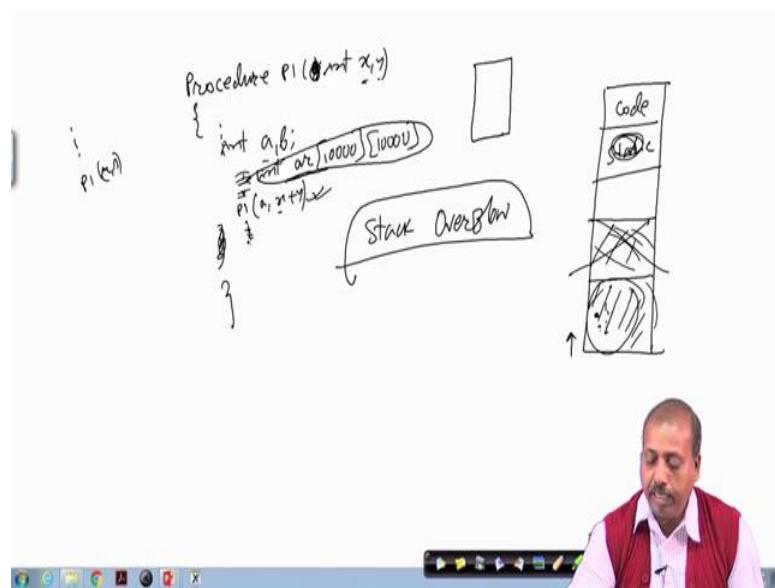
So, next we will be looking into other possible location for this activation record. So, on the stack area so, can we create this activation record on the stack area.

So, it is used in most of the programming languages like C Pascal java etcetera say they do create this activation record in the static area in the stack area. So, as and when a procedure is invoked corresponding activation record is pushed in to the stacks the activation records structure remain same as we have seen previously.

So, it has got the return value return address then parameters and local variables, but they are not created in the static area, they are not created in the along with the global variables. Rather the when this procedure call will be made one such frame will be pushed into the stack. And then this the code that we will have within the procedure when it is referring to those parameters and local variable so, they will be doing it from the stack.

And when it is returning this entry will be popped out. So, that will go out of the stack. So, this works well if local variables are not needed beyond the procedure body.

(Refer Slide Time: 12:15)



So, it is like this what I mean is what I mean is that so, in the previous example. So, we have got this procedure P1 and it has got say arguments that is say int x and y and then within that I have got some local variables a b etcetera.

So, this activation record structure is fixed as we had previously the return address then x, y, a, b etcetera, but they are not created on the global are or the static area rather. So, as you remember that this program snapshot that we talked about so, it has got the code

part, it has got this static part and it has got this heap and stack simultaneously. So, this heap the stack starts from the bottom. So, when a call will be made to the procedure so, this if this is the main routine at some point of time you have given a call to P1 with say value say r and s, then this activation record will be created onto the stack.

So, this activation record is pushed into the stack and here I have got this r and s copy that x and y then this space for ab etcetera. And this code whatever we have. So, they will be referring to the locations that we have in the stack so, the that way the code will be generated. Now, after sometime when this P1 is over then what we will do we will just pop out this block pop out this frame from the stack.

So, after this procedure is over this local variables a b or this parameters arguments xy so, they are not visible because they are already deleted, they do not have any existence after this. So, if you do not need them then it is fine. So, you can do it on the then you can do it in this fashion but of course, the what is the biggest advantage of this the biggest advantage is that you can support recursion now. Because now if there is a call to P1 at this point with the parameter say a and x plus y then you can so, this frame is there in the stack.

So, you can push another frame here which will be corresponding to the activation for this one. And when this call will be over so, this record will be popped out, but this record will remains in the stack. So, that this previous call. So, whatever statements you have after this whatever the statement you have after this. So, they will be executing properly referring to this locations that we have in this part ok. So, that way this recursive procedure calls can be handled very efficiently, if we have got this stack type of realization.

However there are problems we had some places, because one very serious problem that you have is that this local variables that we have so, they are created in the stack. And you really do not know how many times this procedure P1 will be called recursively.

Now, as I said that is local variable so, they are also created in the stack. Now, suppose I have got this local we have got an array here say that is also an integer array say, but its dimension is very high say 10000 by 10000, then what will happen. So, this array we will get created onto the stack itself. So, this your activation record will now be very large, because it has to accommodate such a big array in it, such a large number of

integers in it. So, that will be a very big array. So, this activation record will be very large and the situation will be even worlds. So, if this P1 happens to be recursive function because now recursive procedure, because now every activation we will have one such large set of integers created as local variables.

So, as a result what can happen is that this program while executing so, it can be running out of memory ok. So, this there are some errors called stack overflow type of errors. So, that can occur and the program may misbehave ok. So, to what is the solution, solution is that you do not make this arrays local ok. So, you make this array global variable so, that this the array will come to the static area and they will not be created with every activation of this procedure.

So, though it may or may not solve your purpose like, if you need that for every activation of the procedure I should have a new array, then this is this does not provide any solution, but may a time we will just do it carelessly so, we just make large local arrays so, it is advisable that we do not do that because if you do this large local array. So, they will be created on to the stack and as a result they will if there is particularly if it is a recursive function or recursive procedure, then it will be invoked again and again so, there will be large space occupied by this local arrays. So, there may there is a chance of stack over flow.

So, we can we can just try to solve this problem by having this local arrays made global. So, putting them outside and putting them to the static part of the code. So, so, this is good because we can have this recursion part implemented very easily, you can have this recursion part implemented very easily. So, for languages like less when which is a full function may be return so, activation record is created on the heap.

So, this is another situation where. So, the here I was talking about creation in the stack some programming language like list. So, it will create it will return day full function as a return value of one function like that. So, then in the those cases activation record will be created in the heap, because then how many parameters they it has a dynamically allocate variables and all so, that can be done only from the heap. So, the in that case activation record will be created in the heap ok.

(Refer Slide Time: 18:33)

The slide has a yellow header with the title 'Processor Registers'. Below the title is a bulleted list of six items. At the bottom of the slide is a blue footer bar featuring the Indian Space Research Organisation (ISRO) logo, the 'swayam' logo, and other small icons.

- Also a part of the runtime environment
- Used to store temporaries, local variables, global variables and some special information
- Program counter points to the statement to be executed next
- Stack pointer points to the top of the stack
- Frame pointer points to the current activation record
- Argument pointer points to the area of the activation record reserved for arguments

Now, processor registers so, they are also part of the runtime environment. So, proceed because when a program is executing so, it has got the, it is a processor registers are holding some of the variable values and all. So, that is also part of the runtime environment.

So, they are used to store temporary is local variables global variables and some the special information. So, these are the purpose of this processor registers program counter is another very important register in the processor that is that points to the statement to be executed next. So, that is also a part of the runtime environment then stack pointer.

So, that is also that is also a processor register. So, that is also used to point to the top of the stack. So, that is also part of the runtime environment there is a frame pointer which points to the current activation record. So, some of the processes they will allow this frame pointer to be available. So, you can as a separate register. So, you can use it for pointing to the current activation record.

Some processes will not allow this thing ok. So, in that case you have to dedicate some general purpose register to work as the frame pointer register. And there is a argument pointer that points the area of the activation record reserve for arguments. So, these are the threes supporting pointers that are needed from for supporting this is a dynamic activation records one is the stack pointer, one is frame pointer another is argument

pointer. And as I said that we may or may not be possible that we may or may not have all this pointer registers available.

And many a times so, what we have to do is that we have to use some general purpose register to act as the as this registers but; however, they also form part of the runtime environment. So, they are also to be taken care of.

(Refer Slide Time: 20:33)

## Environment Types

- Stack based environment without local procedures – common for languages like C
- Stack based environment with local procedures – followed for block structured languages like Pascal

Now, if you look into the environment types there can be two type of situation that you can find out, one is tag based environment without local procedures. So, where this local procedures are not supported and we can have stack based environment with local procedures. So, without local procedure as I was telling previously. So, with so we do not have nested procedure definition.

But with local procedure means we have got nested procedure definition. So, they have got so, they have followed in some block structured language like Pascal. So, we will find this nested procedure definition whereas, if we look into languages like C nested calls are allowed, but nested definitions are not allowed. So, we will have to see like how can we handle this situation both of them in grand time environment.

(Refer Slide Time: 21:29)

The slide has a yellow header with the title "Environment without Local Procedures". Below the title is a bulleted list:

- For languages where all procedures are global
- Stack based environment needs two things about activation records
  - Frame pointer: Pointer to the current activation record to allow access to local variables and parameters
  - Control link / Dynamic link: Kept in current activation record to record position of the immediately preceding activation record

Below the list is a hand-drawn diagram of a stack-based environment. It shows a vertical stack of activation records. The top record is labeled "P1()", with a bracket underneath it. To its left is another record labeled "P1()". Below "P1()" is "P2()", which is also bracketed. To its left is another record labeled "P2()", also bracketed. A horizontal line extends from the bottom of "P2()" to the right, with three dots "...". Handwritten arrows point from the labels "P1()", "P2()", and "...".

The footer of the slide features the "swayam" logo and other navigation icons.

First we look into the simpler version that is where you do not have the local procedures. So, all so all the procedures that we have so, they are globally in nature. So, there is you cannot have a one procedure definition nested within another procedure. So, all procedures are global and stack based environment will need two things one is frame pointer another is control link or dynamic link.

So, in this case what happens is that we have got so, this. So, we have got two things one is the frame point and that point to the current activation record to allow access to the local variables and parameters. And there is a control link or dynamic link that is kept in current activation record to the position of immediately preceding activation record.

So, you will see some example basically what happens is that say you have got say one procedure say from the main program, I have given a call to procedure P1. Now, within procedure P1 there is a call to procedure P2 fine, now while we are executing procedure P2 now which definitions are we going to use ok.

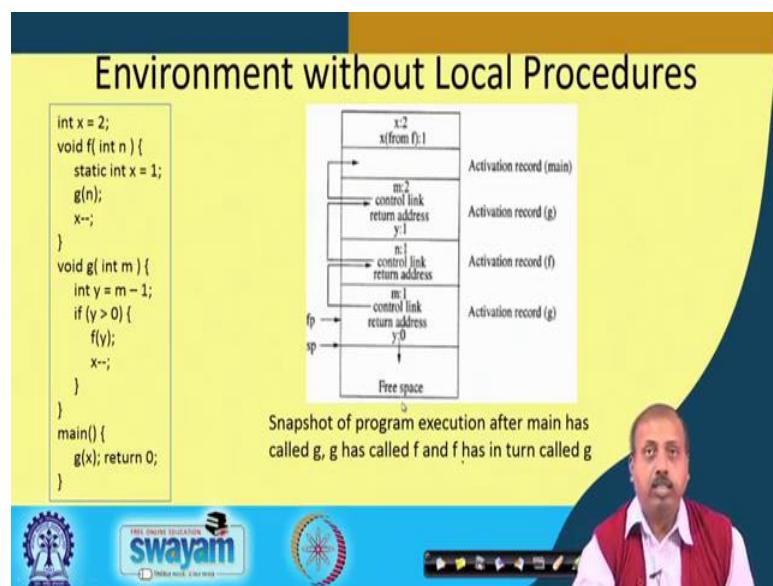
So, if we are if you so if this is the body of P2, then what are the definitions that we are going to use. Now it so, depending upon the scope rules of the language. So, it may be that it is a static scoping so, that is we are so, P2 where from which from which portion of the procedure we should take the definition that they are fixed.

So, that is defined by the static nesting nest nesting rules or it may be its a dynamic nesting rule. So, since at this point of time P2 has come through P1. So, if there is a variable x so, it may so happened that here it is referring to an x equal to something and I have got a global x and I have got another x which is here in P in P1.

The when I am referring to this x so, is it referring to this one or this one so, that is the question. So, now if so if it is dynamic situation dynamic scoping rule, then it can tell me which x to follow. So, there I can have in the activation record one control link. So, that can tell what is the preceding activation record so, so that way I will first search in the activation record of P1 and getting x there so, when I am writing x equal to so, it will refer to this x.

On the other hand if I say that P2's dynamic link so, it is pointing to the global activation so, this the global the table in that case. So, it will be referring to this x. So, based on this activation record. So, we can with this control link we can find out like which variable definitions are going to be used. So, we will take some example and then try to explain.

(Refer Slide Time: 24:49)



Say this is the situation we have got this gives the definition like we have got the functions f and g and from the main program there is a call to g. And so, this is so this is the situation so, this is the activation record for main.

So, main does not have any local variable. So, there is nothing here and that the local part so, if nothing we cannot understand much here, but this x is a global variable so this x is coming as the static part. So, this is in the static this is in the static area ok.

Now, this activation record and then there is another x here so, in within function f which is called static int x1. So, this is this is another x. So, this x and this x they are not same. So, this is a global variable and this is another x coming from the function f so, this x from f. So, that is equal to 1 so that is the situation. And then so this is the snapshot of the program after main has called g, g has called f and f has intern called g.

So, main has called g, g has called f and f has called g from there. So, from here so, we have got this one this m so, we have got from f to g. So, the first call gx y equal to m minus 1 so, this m is getting the value so 2 has been passed. So, y is equal to 2 minus 1 that is 1 so, m is equal to 2 so, that is the parameter that I have passed, then we have got the control link return address etcetera and this is the local variable y so, y equal to 1. So, that is the situation and then there is a call to f of y. So, this function has been called so, this activation record is pushed into the stack now ok.

So, previous to we had add up to this up at the main routine, then it at called g. So, up to this much is created in the activation record. And then now this f has been called so, this is the activation record for f that has been pushed. And in the activation record of f say again we have got this parameter n. So, n value of n is equal to 1. And then because 1 we have passed 1 there so, 1 is passed there and then the this is the static part static int x is the static. So, it has gone to the global part global section and this control link and details I will come to control link later.

Then from here it has given a call to the 0 routine. So, from after this is the activation record for g is created and pushed into the stack. And then this y equal to m minus 1. So, y, y becomes equal to 0 so, m was equal to 1 so y becomes equal to 0 so, this is the situation. Now, this control links they are extremely pointing to the previous activation record that we have so, so from main when I was calling to call to g.

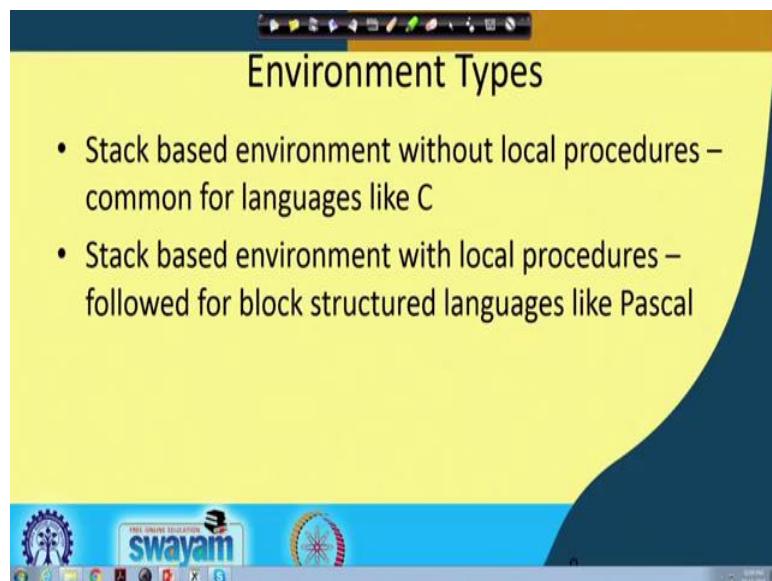
So, control link was pointing to main similarly from g when I was considering so, g even from g I given a call to f so, from f the control link is pointing to the 0 g function. And from here from g the control routine is pointing to the previous 1f.

So, this is the situation now the situation may change so, as we go into further calls and that may complicate the situation. So, here it is very simple. So, here you can say that the control links are not necessary, because I can just pop out the entries from the stack and that will be going absolutely fine this happens here, but in more complex situation that may not be the case ok. So, we will see that in successive examples.

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

**Lecture – 48**  
**Runtime Environment (Contd.)**

(Refer Slide Time: 00:18).



So, in our last class we have been discussing on Runtime Environments. And we have seen that there can be two types of environments that are possible; one is without local procedure and another is with local procedure. So, without local procedure means; within a procedure we are not going to define any new procedures. However, and with local procedure mean within a procedure; so we can define your own procedures further.

So, that the procedure is local for so, inner procedure is local for the outer procedure only and they are not visible outside the main procedure; so we will come to that. So, this the first type of case, where we do not allow this local procedure. So, that is C type of languages they have this facility. On the other hand so, the impel languages like Pascal; so you can have local procedure so that within a procedure, we can define another procedure.

(Refer Slide Time: 01:13).

Environment without Local Procedures

- For languages where all procedures are global
- Stack based environment needs two things about activation records
  - Frame pointer: Pointer to the current activation record to allow access to local variables and parameters
  - Control link / Dynamic link: Kept in current activation record to record position of the immediately preceding activation record

So, we will see both of them and then how to go for their implementation in runtime, how are they going to be handled. So, with local procedures we say, without local procedures; so for languages where all procedures are global. So, that is the situation where we have got local non-local procedures. So, here if you look into the type of this language.

So, if you are writing a program. So, we can have a main program and after that you can have a number of procedures and these procedures are visible to, all these procedures are visible for in the entire program; so that is the situation. So, we do not have difficulty as per as this definition of procedures are concerned. However, there maybe problems with that situation, that or the every all the procedures are visible at all or at all places.

So, that may be a concern and some sort of hiding that you have otherwise so that is not visible. So, if you have got some design style, where you have got a main operation to do and under that only certain operations are valid. So, that type of situation; so you may like to hide the other procedures from this the internal procedures from outside, that is not possible.

So, here all procedures are global; so any procedure defined in the language; so for the entire program it is visible. So, in this type of situation, this stack based environment that we have that we create for this local procedures, they need two things for the activation record; one is known as frame pointer. So, in frame pointer so pointer to the current

activation record is there; so that is a there is a that is called a frame pointer. So, this allows access to the local variables and parameters. So, this actually points to the current frame, that is the frame corresponding to the current procedure that has been called. And there is another link called control link or dynamic link. So, that will be that will keep the information about from which procedure this was invoke.

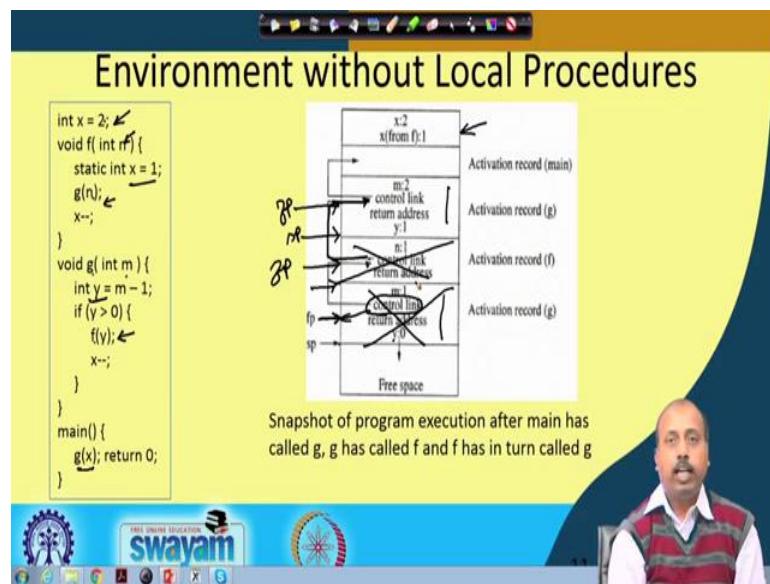
Accordingly, from which activation record so this activation record got created. So, this is basically the parents parent relationship like if there is a say procedure P1 and that procedure P1 is calling , if that procedure P1 is calling say procedure P2.

So, if P1 is calling P2 then, if this P1 is calling P2, there in the P2s activation record. So, there will be a pointer, which will 0.2 this P1s activation record; so that will happen. So that is that will be called control link or dynamic link. So, the idea is that when P2 will be over so, we need to say that P1s activation record is the current 1.

So, accordingly that can be active that can be made the current, current procedure. So, basically this frame pointer that we have so, when we come back from P2 to P1, this frame pointer can be made to 0.2 the activation record for P1 and we keep the information about activation record of P1 in the control link of P2.

So, we will see some examples that we will make it more clear. So, as we, as we go into more and more, some example; so this will be more clear. So, let us see how this things are going to work.

(Refer Slide Time: 04:43).



So, this one is an example like; here I have got a function say, this is the main, this is a function f and there is another function g and we have got a main routine. So, as there is so, this is f and g, you see they are global procedures or global functions, whatever you call it and that is available throughout the program.

We have got a global variable integer  $x_2$  here and the within this functions f and g we have got this local variables. So, this is static i integer and this is a normal integer. So, this is this is a local integer. So, this is a local variable  $y$  is a local variable and  $x$  since, it is defined as a static integers so that is in that sense it becomes global.

So, it gets allocated to the static area or the global area. So, you see that in this area this  $x$  are the main program the global variable  $x$  and this static variable  $x$  so they have been allotted. Now, when this function is called so from the main routine; so this is the activation record for main and from main, we have given a call to g.

So, in the call for g we have got this variable this, this activation record got created, when this when this function g is called so, this part of the activation record is created and there; we have got this, this is the parameter  $m$  that is passed. So, it is getting the value  $g(x)$  and these  $x$  refers to this global variable  $x$ . So,  $m$  is  $m$  is  $m$  is getting the value 2. So, this parameter is passed, after that there is a control link and that control link points to the previous frame ok.

So, from where this g has been invoked. So, this g has been invoked from this main. So, as a result this control link points to this. So, then we have got there is this return address so that that will be stored and a local variable y has been created. So, y is allocated space in the activation record for g and y equal to m minus 1. So, y evaluates to 1 at this point of time.

So, after sometime when this, this point comes, it is calling the function f and as a result, there will be another activation record created for this call f and in that we have got this, this parameters got a parameter n

So, this n is given allocated space on to this activation record and this value of n is equal to the value of y and value of y was 1 so this gets the value 1. And the control link in the activation record for f, it points to the for the; it points to the frame corresponding to the function g from where the f is called like here the function f is called from function g.

So, this activation record this control link points to the, previous frame from where this particular call was invoked. And then at that at some point later when is the in the call f so, here again g is called so, f and g are they are two mutually recursive procedure.

So, at some point of time suppose this g has been called within f. So, as a result so, next activation record for g will be getting created and here I will have this, this m. So, there, here we have got this call to g and g has got parameter m. So, this m is coming and this m is assigned the value 1, because this g of, it is calling g of n. So, whatever value was passed here; so with that it is called; so, it is calling with n1.

So, that way, this is this will be this g is called with the value of m being equal to 1 and then the control link points to the previous activation record of f and this return address and the another copy of this local variable is created y, that is made equal to 0, because that is equal to m minus 1, it evaluates to 0. And then you see that this control link points to the previous activation record and the current frame pointer points to this current frame, the currently active procedure and the stack pointer points to the up to which this is full, this stack is full activation record stack is full; so it points to this.

So, this is the snapshot of the program when main has called g, g has called f and f has called g recursively; so this is the situation. Now, after say this g gets over so, in the execution of g so, suppose y is such that. So, y is equal to 0. So, it will not call f any

more. So, it will be over after sometime then this part of the activation record will get deleted and this frame pointer, it will get the value from this control link.

So, what this control link we are storing the frame pointer for the previous activation record. So, that value will be copied to the frame pointer as a result the frame pointer will be coming to this point. It will point to the activation record for f and so once this is d, deleted. So, stack pointer will automatically come to this point; after sometime when this one also gets over so, this will get deleted and now, the frame pointer will get the value of previous control link that we had so, this control link was pointing this.

So, this frame pointer will be coming to this point and the stack pointer will come to this point. So, this way the so this one also get delayed in destroyed. So, this way this activation records they grow dynamically on to stack. So, as and when we are calling a new procedure the corresponding activation record is getting created and it is getting stored into the stack.

(Refer Slide Time: 10:54).

## Accessing Variables

- Parameters and local variables found by offset from the current frame pointer
- Offsets can be calculated statically by the compiler.
- Consider procedure g with parameter m and local variable y

The diagram shows a memory stack layout. At the bottom, a frame pointer (fp) is at address 1000. Above it is a portion of the activation record. A control link points to the previous frame's fp, which is at address 994. The current frame's fp is at address 996. The activation address is at address 998. A local variable y is at address 999. An offset yOffset is shown relative to the fp. The calculation for offsets is as follows:

$$\begin{aligned}mOffset &= \text{size of control link} = +4 \text{ bytes} \\yOffset &= -(\text{size of } y + \text{size of return address}) = -6\end{aligned}$$

Hence, m and y can be accessed by  $4(fp)$  and  $-6(fp)$ .

FREE ONLINE EDUCATION  
swayam

Now, so, this is fine. So, this is fine as long as we do not have local procedures. Now, how do you access variable? So, this is a typical situation where we have got this frame ok. So, this is this is a this is a frame in the activation record and in that frame. So, there is a frame pointer. So, this frame pointer; so this actually points to the, a points to a position where before so, there is a portion of the activation record above the frame pointer and there is a portion of the activation record below the frame pointer.

So, above frame pointer, so, we have got the parameters that are passed ok. So, parameters and local variables so, these are the two things that the compiler we will need to access; need to get the addresses so that during execution, those locations can be accessed by the computer. So, this is parameter and local variables, they are found by offsets from the current frame pointer.

So, current frame pointer is here and above that there will be, there is a, there control link is told. So, this is again and a address so, certain number of bytes will be needed. So, above that we have got all the parameters. So, this portion is for the parameters; this portion is for the parameters. So, in this case we have got only one parameter m.

So, that is told there and then so, starting from the frame pointer. So, if you go by a negative offset so, so, go by a positive offset. So, you can find you can reach this particular point; so this offset for the parameters. On the other hand this local variables; so, they are stored after the frame pointer.

So, that is after this frame pointer, we have got this return address and after that we have got all the offsets all the local variables created. So, the way local variable y gets created here. So, we can understand that the offset of this frame pointer is, offset of this parameters is equal to the size of control link ok. So, that is the beginning of the parameter. So, in this particular case there is only one parameter.

So, the offset of m is equal to size of control link equal to plus 4 bytes and in this function g that we had previously considered there is only one local variable that is named y. So, offset of y is size of y plus size of return address, because we have got this field size of so the size of return address will come and after that we have got this space for y.

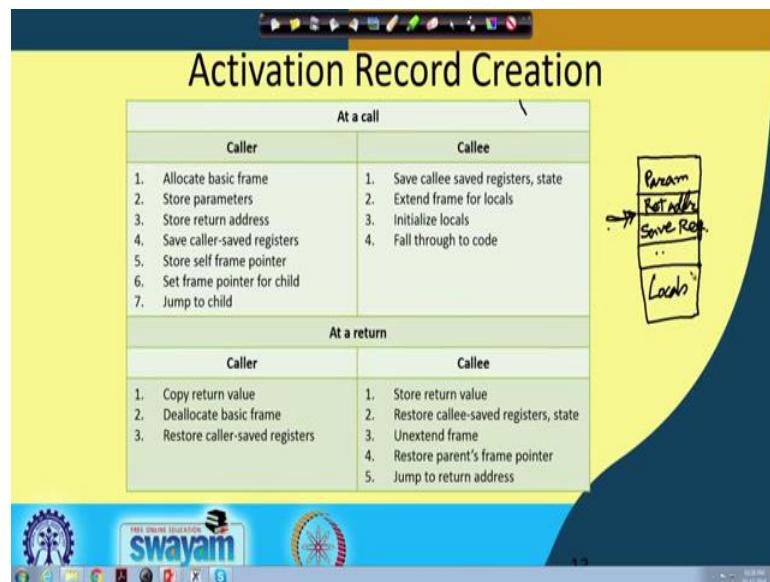
So, they so negative of that, so minus of size of y plus size of return address. So, if this frame pointer is say 1000, then this offset of y becomes 994 ok. So, 2 bytes are used for storing the return address and 2 bytes are used for the storing this y. So, you can say that at an offset of so, return address is 2 bytes and say integer is 4 bytes.

So, as a result total is 6. So, from 994 so this, so this is 4 bytes. So, if I take integer to be 4 bytes. So, this will be 2 plus 4 6. So, it will be by minus 6 offset from the frame pointer. So, this m and y so, they can be accessed by 4 fp and minus 6 fp. So, 4 fp means;

it is fp plus 4 and then this a minus 6 fp means; fp minus 6; so these are the two values that we have.

So, accordingly it can access the parameters and this local variables. So, this accessing of these parameters and local variables are with respect to the frame pointer. Next, so if I have got multiple number of parameters passed or multiple number of local variables. So, they are offsets will be calculated accordingly the compiler can find out the actual byte at which this is value will be stored it can do that.

(Refer Slide Time: 15:11).



So, activation record creation. So, the so, there is some part of responsibility on, on behalf of the caller, the function which calls this new function and the procedure which calls the inner procedure and then there is and there is responsibility of the callee also.

So, this caller calls the callee. So, this activation record creation it is partial responsibilities is with the caller and partial responsibility with the callee; so let us see what are the things that are going to happen at a call at the caller end first. At the caller end; so it will be allocating the basic frame, because the first a frame has to be created. So, that frame gets created by the caller, store parameters so, parameter values will be stored there.

So, it you can understand that. So, it is like this. So, like activation record suppose this is the activation record that has been created by the caller ok. So, in that caller so, it will be

first it will store the parameter; so in the first part, it will be storing all the parameters. All parameters are stored, then it will store the return address; so this is the return address that is stored.

So, these are known by the caller. So, they are filled by the caller and then there may be some registers CPU registers, which are saved by the caller, because there maybe some registers, which are very useful for the caller and the caller does not want that the callee routine should override them ok.

So, that way it can save some registers. So, some save registers which are important for the caller. So, that can happen and after that it will store self frame pointer. So, it will store the frame pointer in the actually, this is that. So, it will the self so, in the frame pointer that I had previously so, that frame pointer will be stored here and then it will jump to the child routine. So, this set frame pointer of our child so, frame pointer so, for child will be pointing to this new frame pointer and then it will be jumping to the child routine.

At the callee end we have got the, their maybe some registers that callee wants to save. So, they will be saved and then it may, it like to save, some state of the some variables also so, that can also be saved then extend the frame for locals after that it will extend this frame so that it can create all the local variables there.

So, this is the all this is the space for the local variables and then it will fall through the code; so this is the responsibility on part of the callee. Similarly when you are returning from the procedure then there are certain responsibilities for the caller and certain responsibilities for the callee. On the caller side, it will copy the return value deallocate. So, let us first talk about the callee, because while returning callees for portion comes first.

So, it will store the return address a return value in the slot in the activation record restore callee saved registers and state. So, whatever registers callee had saved so, they will be retrieved and they will be restored. Then unextend frame; so unextend frame means; this locals will get deleted, the stack pointer will be implemented; as a result all the locals will get deleted. Then restored parents frame pointer; so frame pointer of parent was stored. So, that is restored now and it will jump to the return address available in the activation record.

So, this way so, this is the portion of the job that is done by the callee. On the other hand, the caller it will copy return value. So, the return value may be copied into some array some variable or so, so, that will be done by the caller. So, then the frame will get deallocated and so, the caller saved registers whatever registers the caller had saved. So, it may like to restore those registers; so that way this will be done.

Now, one thing you should understand that these saving of registers and also. So, it is very much dependent on the system ok. So, it may in some cases; so it may be that this saving registers is necessary some cases it is not that important. So, based on that so, this compiler designer may take a decision that whether to save this registers into the frame or not.

(Refer Slide Time: 19:57).

## Environment with Local Procedures

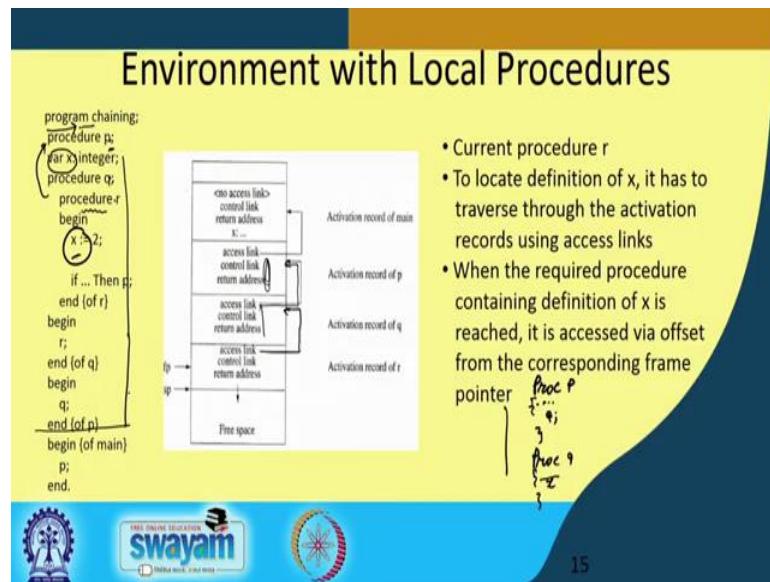
- For supporting local procedures, variables may have various scopes
- To determine the definition to be used for a reference to a variable, it is needed to access non-local, non-global variables
- These definitions are local to one of the procedures nesting the current one – need to look into the activation records of nesting procedures
- Solution is to keep extra bookkeeping information, called *access link*, pointing to the activation record for the defining environment of a procedure



14

So, next we will be looking into this environment with local procedures like if there are local procedures in my description, in my program then how are they going to be handled?

(Refer Slide Time: 20:13).



So, the basic problem that we have is something like this like this is a program whose name is changing. So, here I got a procedure *p*, it has got a variable *x* of type integer and a procedure *q*. So, procedure *q* is defined inside procedure *p*. So, within procedure *q* another procedure *r* is defined. So, this begin end is the portion for *r* after that we have got. So, this begin is corresponding to the procedure *q*.

So, procedure of *q* up to this much; so this is actually the definition parts. So, the variables, local procedures, etcetera; so they can be defined here. So, of course, in this case we do not have any local variable in procedure *q*. So, the nothing comes there; however, if it is there then they can be defined in this portion, then it this begin is for the body of procedure *q*. So, procedure *q* starts at this point; so procedure *q* starts at this point and then it gives a call to procedure *r* and then it is the end of *q* then this is the begin of procedure *p*.

So, procedure *p* is starts at this point, it gives a call to procedure *q* and that is the end of *p* and you have got this main program, which is calling the procedure *p*. So, now this is the situation like. So, environment with local procedure so, this is the situation like so, current so, we have got this. So, this is we will need a concept of access link. So, let us explain that first, then will be coming back.

So, for supporting local procedures variables may have various scopes. So, like say suppose, I have got a variable reference here. So, this suppose I have got a variable

reference within a procedure, that gives us like this, that say here, I have got a reference to some variable like here, I have got a reference to x.

Now, this x assigned as 2 now how are you going to handle this? Like when you are searching for this procedure for this variable x. So, you do not find the definition here. So, you do not find the procedure of the x definition here. So, you need to go up and look into the nested, next higher level procedure that is procedure r. Now, here after coming to this sorry, in procedure p.

So, after coming to procedure p, you find that the variable x is defined. It may so happen that this x is not defined even here; so x is defined globally. So, that way there is a hierarchy of say hierarchy of this frames into which you need to look for and then and that is defined by the scope of the language.

So, it here it is said that if some in this particular language. So, it is assumed that if something is defined here so, that will be visible to entire program, if something is defined within p. So, that will be defined only within p so, it is visible to this up to this much this x will be visible ok, because that is the begin of the end the p ends are this point. So, up to this much this x is visible; so how to take care of this situation?

So, for that matter I need to know in which sequence, I should search the variables. So, this is so, this x in this particular case, you see that x is not local to the procedure, at the same type x is not a global variable. So, this is a non local, non global type of situation.

So, it is defined in some higher level procedure, but it is not defined here ok. So, how to handle this situation so, that is actually told here; so this is for supporting a local procedure. So, variables may have various scopes, so that may be there and to determine the definition determine the definition to be used for a reference to a variable, it is needed to access non local non global variable.

So, that is that is important, that is at a typically a type of thing that we are coming across and these definitions are local to one of the procedures, nesting the current one and need to look into the activation records of nesting procedure. So, we need to go up by one or several levels for getting the corresponding definition.

So, now how to do this traversal? So, that is important. And for doing this we keep one extra bookkeeping information called access link. So, this access link comes important here. So, another so, we have got a frame pointer and control link previously, which was storing the frame pointers. Now, we will see that one access link will be introduced so, which are going to keep the note of this next variable, next procedure, or next nesting procedure.

So, how is access link works? See you so, this is the situation. So, here at currently at procedure r, we are currently at procedure r so, how did you come to procedure r? So, this is the main this is the activation record for the main. So, there is no access link because it is at the highest level only then at procedure p, we have got one access link that points to the frame of main ok. So, this, because if I do not find any definition in procedure p from the nesting part. I know that if this definition has to be searched for a global variable.

So, that way it is pointing to the activation record of main. Now, when you are at procedure q so, from p we have given a call to q and when did when we called this q so this next frame got created. So, this frame got created and in this frame we have got so, in this frame we have got this thing, your, this frame pointer frame pointer is there, but its access link points to the this activation record for p. So, and at the next level from p your, from q we are giving a call to r.

So, if we are within r then this access link points to the frame corresponding to q. So, that if something is not found in r. So, it will be referring to procedure q, q's definition and if something is it is not found here, also then it will find it here. Now, you see here what has happened is that so this procedure p, within procedure p we have got procedure q and within q we have got procedure r, if that was not the situation then of course, this will not happen like this.

Like say, if the situation was something like this, so you have got this procedure p ok. So, we have got some definitions and it is giving a call to procedure q, but it is not defined inside this. In that case so, if this is procedure q, in that case that the access link of procedure q will not point to the frame for p, but it will point to the main routines, the access link for main routine, because q is not nested within p.

So, in this case what was happening is the q is nested within p. So, if in q we do not find some definition, we can look into the procedure p is variables ok, but if in this particular example. So, p and q they are two different procedure; there is no nesting of p and q.

So, even if p calls q so, so, q cannot use the variables that are local to p as its variable ok. So, what you what will be the happening is that if some in some variable is referred here, x is referred here so, it will not look into the activation record for p for getting and by getting the location for x, but it will look into the as into the global variable. So, this will be more clear as we see more and more examples.

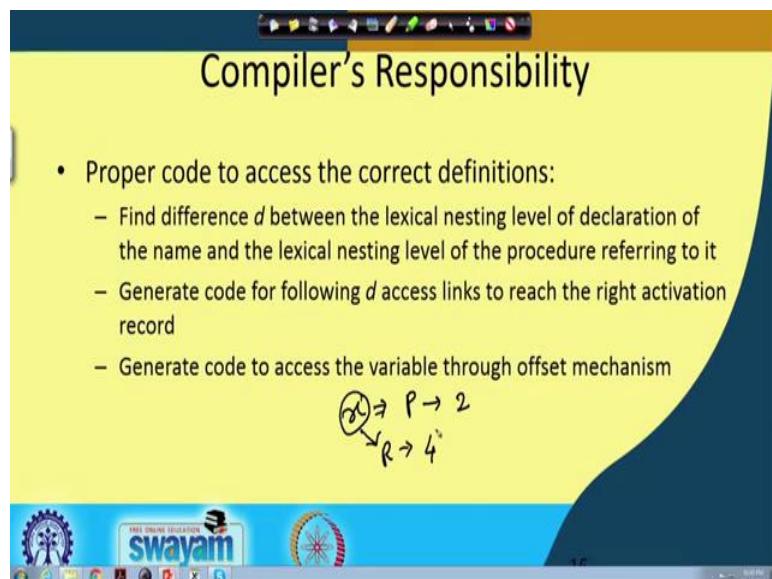
Now, so this so this is the situation, when the current procedure is r and to locate the definition of x, it has to traverse through the activation records, using access links and when the required procedure containing definition of x is reached. It is accessed by a offset from the corresponding frame pointer.

So, in this case from for r, it will go to q and so, at this point so, in the, in procedure r, there is a reference to x. So, when trying to generate code, for that it will not find the reference x. So, with this access link tells me that you have to go to the activation record for q. So, it will come here, it will search for x here and again it will not find it, then it will go up by one more level and it will come to this point and then it will search into this local variables of this p and then it will get x. So, it has it will traverse by the access links to get the corresponding variable.

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

**Lecture – 49**  
**Runtime Environment (Contd.)**

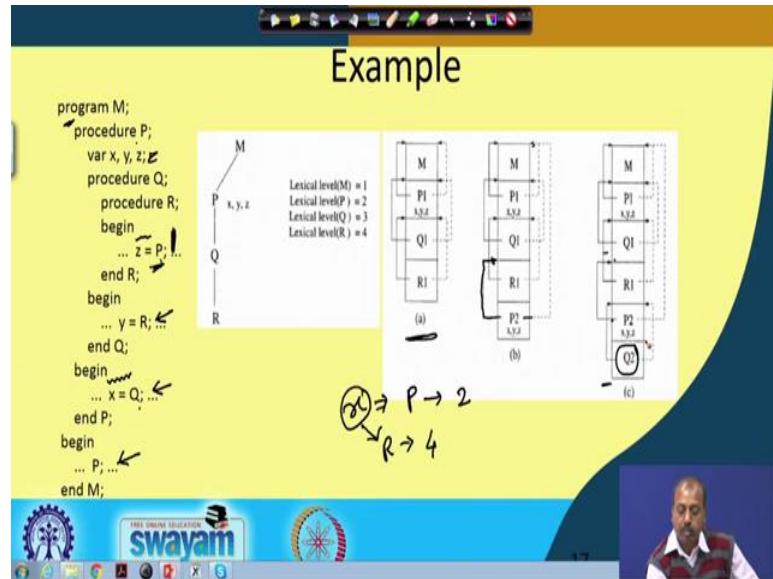
(Refer Slide Time: 00:18)



So, what is the compiler's responsibility? Like, so this access link is fine. So, we have to go by the access link to come to the corresponding definition of your name of an identifier. But, how the compiler will generate code so that this access will be correct. So, first thing that it has to do is to find difference  $d$  between the lexical nesting level of declaration of the name and the lexical nesting level of the procedure referring to it. So, this is basically in our previous case previous example that we had.

So, we were referring to  $x$  and then the, so it is defined it is, so this is at this is this was at procedure  $P$  and procedure  $P$  is nesting level was some say 2 because main was 1 and  $P$  was 2. And then this it is now different at the current procedure that we are referring is a procedure  $R$  whose nesting level is 4, so it has to go by 4 minus 2. So, it has to go by two access links further to come to the proper definition of  $x$ . So, this will generate code for following  $d$  access links to reach the right activation record and then it will generate code to access the variable through offset mechanism.

(Refer Slide Time: 01:37)



So, we will take an example and explain it. So, like in the previous case what we had. So, this was the program that we are talking about. So, here we have got a number of variable, so  $x$ ,  $y$ ,  $z$  etcetera. Now, suppose we are at a situation where from the main routine the  $P_1$  has been called ok. So, when the  $P_1$  has been called, so in  $P_1$  it has got reference for this  $x$ ,  $y$  and  $z$  and this  $P$  ones. So, this access link points to the made as it corresponds to the parent frame, so that is  $M$ . So, it is point  $P$  is the access link is pointing to  $M$ .

Similarly, so from  $P_1$  when this after from this procedure  $P$  we are going to call  $Q$ . So, this is the begin of  $P$ , so from here we are going to call  $Q$ . So, when this call  $Q$  is made so this  $Q$  is invoked and then it has got its access link is pointing to  $P$  because  $Q$  is defined within  $P$  ok. So, if you call if you have to look for some definitions. So, we have to look into the local variables of  $P$  first ok.

So, if it is, like this  $x$ ; so,  $x$  is not defined in  $Q$  in the procedure  $Q$  ok. So, it is so this is not defined in the procedure because, so this variable  $y$  is refer to. So,  $y$  is not defined in the procedure  $Q$ . So, when it is trying to get the corresponding declaration for a  $y$  so it has to look into the activation record the, it has to look into the activation record for the nesting procedure and the nesting procedure happens to be procedure  $Q$ . So, this  $R$  it points to procedure  $Q$  and  $Q$  a points to procedure  $P$ .

So, that is the situation where this M has given a call to P, P has given a call to Q, so this is M has given a call to P at this point P has given a call. So, it has got the main has call given a call to P ok, then P has given a call to Q. So, this is the P, so this has given a call to Q and Q has given a call to R. So, this call has been made and we are in this procedure. So, we are in this procedure. Then that is the situation that is depicted by the first diagram ok. So, it has not yet made this call for procedure P.

So, you can say that as if we are somewhere here, ok. So, at this point the this is the situation of the activation record that in the stack. So, this solid lines are the control links and this dotted lines are the access links. So, this access links, so, they are pointing to the corresponding procedure into which we have to look into.

Now, the situation changes when we have got this thing see this, see this one, see this when from this P when from this R when this P is called again. So, another frame gets created and this P2 is this frame P2 is pushed into the stack. And then this frame pointer so this control link points to the parent frame from where it was called.

And the access link for P, so see now it points to this points to M. Why? This procedure P is statically nested within M. So, it is now, so though it has been called from R. So, this call is from R, but R is not the place where it should look for some variable which are which may be undefined in P. So, for all the variables that this P2 wants to access. So, it has got this local variables fines, but if it has to access in the code for P if there are some access to the some variables which are defined in the form in the main then in that case it will be going into this. So, you see that the control link points to R1, but the access link points to M ok. So, this is the situation at this point.

Now, after from this if I think that this P call has been made and within from this P call, so it has given a call to Q and Q has given a call to R. So, then there this is the situation at that point. So, this M, P1, Q1, R1, so that was remaining from R1 a call to P has been made, so this is P 2.

So, access link points to the main routine, control link points to R1, the access link points to the frame corresponding to main and this control link points to the frame corresponding to R1. And from P to another call to Q has been made, so this Q frame has been. So, this Q frame has been the activation record for Q has been created and then this

points to the this access link points to the activation record for P because this procedure Q is defined within procedure P.

So, that is statically nested thing. So, Q is within statically nested group of P procedure P. So, that way this access link pints to P. And then this control link also points to P because from P this called to Q has been made ok. So, both of them are done like this.

And you note that this Q, it is not pointing to M, the access link is not pointing to M because Q is defined within procedure P. So, for definition it has to look into procedure P only. So, this way this access link and control link, so they can be useful for getting the getting access to local proper variables that we need for a particular program for a particular procedure ok. And for a define depending upon the language the this nesting rules we will come and based on that the compiler generator can design this portion.

(Refer Slide Time: 07:56)

## Compiler's Responsibility

- Proper code to access the correct definitions:
  - Find difference  $d$  between the lexical nesting level of declaration of the name and the lexical nesting level of the procedure referring to it
  - Generate code for following  $d$  access links to reach the right activation record
  - Generate code to access the variable through offset mechanism

Now, so that is the situation, so what the compiler designer has to do, is to find the difference  $d$  between the lexical nesting level of the declaration of the name and the lexical nesting level of the current procedure. And after knowing that so it will be it will generate code, so that at runtime the system we will traverse  $d$  such links and it will come to the proper frame. And after coming to the proper; after coming to the proper frame, so it can use the offset mechanism that we have discussed previously to come to the exact location where that particular variable occurs ok.

Now, you see that this is the very combustion thing because what is happening is that as this nesting level increases, so there is a they get tree like this. So, this will be, this will be growing like this and then for if something is defined at level M that for example, then it has to go through all this levels, so it has to traverse through a large number of links to come to M. So, can we do something so that it may be made faster, ok. And for doing it faster so, there is a concept called display. So, this display has got nothing to do with say computer display and all. So, this is a data structure and in the display data structure. So, we store the relevant pointers to the to this activation record stack.

(Refer Slide Time: 09:20)

The slide has a yellow header bar with the word 'DISPLAY' in bold. Below the header is a bulleted list:

- Difficulty in non-local definitions is to search by following access links
- Particularly for virtual paging environment, certain portion of the stack containing activation records may be swapped out, access may be very slow
- To access variables without search, *display* is used

At the bottom of the slide, there is a blue footer bar with the 'SWAYAM' logo and other icons. On the right side of the footer, there is a small video window showing a person speaking. The slide is numbered '18' in the bottom right corner.

So, we will see how this thing works. So, it is the difficulty in non-local definitions is to search by the by following access links. So, that we have understood because if there are nonlocal definition, so you have to go by the access links only to explore the previous procedures and come to the proper definition.

And particularly for virtual paging environment certain portion of the stack containing activation records may be swapped out and access may be very slow. So, what it may. So, virtual paging is a memory management policy where we keep only a few pages of an executing program into the main memory. So, the idea is that we can we do not load the entire program into main memory at any point of time. So, only the relevant pages are loaded. And then after the after sometime if a reference is made to a page which is not present in the main memory then a page fault occurs and the system we will system

will tell the disk save controller to transfer the proper pages from the disk to the main memory and then the execution of that process resumes.

So, it becomes slow because in between the disk controller has to come into play. So, if it happens like this that this stack becomes very large then the system may decide that, I will be swapping out some portions of the stack for some are more important jobs and then the activation. So, as a result some of some part of the activation record maybe swapped out from the main memory. So, this will make the access very slow.

And how to solve this problem? So, we are trying to look for some solutions which is similar to the cash memory type of organization, where the most important and most relevant portion, so they are kept in some fast accessible memory. So, here also we do something like this. So, we use a data structure called display which will be used to access variables without search ok.

(Refer Slide Time: 11:23)

The slide has a yellow header with the word "Display". Below the header is a bulleted list:

- Display  $d$  is a global array of pointers to activation records, indexed by the lexical nesting depth
- Element  $d[i]$  points to the most recent activation of the block at nesting depth  $i$
- A nonlocal X is found as follows:
  - If the most closely nested declaration of X is at nesting depth  $i$ , then  $d[i]$  points to activation record containing the location for X
  - Use relative address within the activation record to access X

The footer of the slide includes the Swayam logo and other educational institution logos, along with the number 19 and a photo of a speaker.

So, how are you going to do this? So, this is the situation.

(Refer Slide Time: 11:25)

**Example**

- Maximum nesting depth 4, so 4 entries in the display
- In Fig (a), M has called P, P has called Q and Q has in turn called R
- Compiler knows that x is in procedure P at lexical level 2
- Code is generated to access second entry of the display to reach the activation record of P directly
- Same is in Fig (b)

(a)

(b)

20

So, we have got this M. So, the this suppose this was the this were the procedures M, P, Q and R. So, at this point of time M has given a call to P, P has given a call to Q and Q has given a call to R. So, that is the situation. So, there are 4 procedures which are active at this point of time. So, 4 activation records are important at this point. So, accordingly this display point display has got 4 levels 1, 2, 3 and 4, level 1 points to M, level 2 points to P1, 3 point to Q1 and R, 4 points to R1.

Now, once the compiler knows that, the difference between this current position and this definition is by certain number of access links. So, it can go up or down by so many access links from here and directly come to this. For example, to go two step backward, so is the current level is 4 it will do 4 minus 2, so it will come to this one and accordingly it will be accessing Q1. So, it does not need to go through this pointers through this Q 1 to reach P1. So, from R1 it can immediately reach P1.

Similarly, after sometime when P R1 has given a call to P, so, another activation record P2 is created. Now, you know that P2's variable, so they are all defined in M. So, if something is not available within P2 that becomes non-local for P2 then it needs to look for M. So, the nesting level of P2 is 2 and nesting level of M is 1. So, if the compiler finds that it has to go by access links, so the difference will be 2 minus 1 equal to 1. So, d will be equal to 1. So, from this it will be just going by one level and it will be coming to M and get it done, get the things.

So, let us try to explain what are writing here. So, what is display? Display is a global array of pointers to activation records indexed by lexical nesting depth. So, this is a for this is an array of pointers to activation record and index is lexical nesting depth. So, this M the main routine will have depth i, then the procedures that the based on this static nesting, so they will be having 1, 2, 3, 4 like that.

Element  $d[i]$  points to the most recent activation of the block at nesting depth i. So, this is the  $d[i]$ . So, a non-local x will be found like this that if the most closely nested declaration of x is at nesting depth i, then  $d[i]$  points to the activation record containing the location for X. So, if the most closely nested declaration is at, so is at X is at nesting depth i like here. So, here the for getting the x, y etcetera. So, we know that it is at level 2. So, that way it can, so, the X is at nesting depth 2.

Then  $d_2$  will point to the activation record for the location for X and then we can use relative address within activation record to access X. So, this is basically that frame pointer plus minus for local variables and parameters; so, that is there. So, this says that a to go to a particular depth so you do not need to go traverse by the access links. So, just look into the index of the display and get it come to the access link. So, in this particular example say maximum nesting depth is 4 because there are 4 procedures, so maximum nesting depth is 4. So, we have got 4 entries in the display.

So, in figure a, the first part of the figure M has given a call to P, P has called Q and Q has intern called R. And now if the compiler has if there is if R is referring to x in the code of R if it is referring to x and it has to found out then the compiler knows that the that identifier x is in procedure P at lexical level 2. So, by searching symbol table it knows that x is at level 2. So, it has to go by 2 levels as I was telling. So, this code to generate code is generated to access control entry of the display to reach the activation record of P directly. So, since this is 4 and this is 2, so it will do 4 minus 2, 2, so, it will come here directly.

And here also as I was explaining that this x, y, z. So, if P is referring to some variable which is not defined in P2, then this compiler we will know that whether it is defined in M or not if it is so, then it will know that what is the difference between these two levels. So, this is at level 2 and this is at level 1. So, 2 minus 1, 1, so, it will immediately find the index 1. So, that way it can this it can utilize this display, so that we do not need to

traverse by the links we can directly come to the corresponding, we can be directly come to the corresponding activation record.

(Refer Slide Time: 16:52)

## Maintaining Display

- When a procedure P at nesting depth i is called, following actions are taken:
  - Save value of  $d[i]$  in the activation record for P
  - Set  $d[i]$  to point to new activation record
- When a procedure P finishes:
  - $d[i]$  is reset to the value stored in the activation record of P

21

But maintaining display is a problem because this compiler has to do something extra for maintaining the displays. At runtime this has to be done so compiler has to generate appropriate code, so that these displays are maintained.

When a procedure P at nesting depth i is called, so, we are doing these actions. Say value of  $d[i]$  in the activation record for P and said  $d[i]$  to point to new activation record. So, this  $d[i]$  will be saved in the activation record. So, previous display value will be same. So, that will be because when coming back when we will need to restore the this displaying index value, so that way it is saved there and then  $d[i]$  is  $d[i]$  points the new activation record. And similarly, when a procedure P finishes  $d[i]$  will be reset to the value stored in the activation record for P, so that the display gets restored to its proper value. So, you can just try out these things and try to see like it is really happening like that.

(Refer Slide Time: 17:55)

The screenshot shows a slide titled "Example" with a yellow background. On the left, there is a program code listing:

```
1. Program X
2. var x, y, z;
3. Procedure P
4.   var a;
5. begin (of P)
6.   a = Q; ✓
7. end (of P)
8. Procedure Q
9. Procedure R
10. begin (of R)
11. P ↙
12. end (of R)
13. begin (of Q)
14. R ↙
15. end (of Q)
16. begin (of X)
17. P ↗
18. Q ↗
19. end (of X)
```

On the right, there are several hand-drawn activation record diagrams. One diagram shows a stack of records with arrows indicating control links:  $x \rightarrow Q \rightarrow R \rightarrow P$ . Another diagram shows a stack with  $x \rightarrow a \rightarrow R$ . Other diagrams show the stack at different points in the execution, such as  $x \rightarrow P$  or  $x \rightarrow d$ . There are also some annotations like "6, 11, 14, 17, 18" and checkmarks next to certain lines of code.

Next, we will be looking into an example where we have got a set of procedures nested like this, and then we are going to use we are going to see like how this activation records will look like ok. In the first part of the problem, so we will see how this activation records we will look like in this part and then we will be is a going for some display based solution.

So, it says that show the continent the stack of activation records at line number 6, 11, 14, 17 and 18. So, at line number 6 what has happened is, so this P has started and so from the main routine, so from the main routine that is the name of the program is X it has given a call to P and after that it has given a call to Q. So, at this point has given a call to Q, so, what happens at that point ok.

So, a X has given a call to P and P has given a call to Q. So, the activation record it will be like this that I will have this X, P and Q will be the activation records. So, this is the; this is the activation record for X, this will be the activation record for P and this will be the activation record for Q. Now, as per as control links are concerned, so Q will be pointing to P because P has given a called to Q and P will be pointing to X. So, this will be the activation record.

And as far as your access link is concerned this is the control links and the access links you see P and Q. So, they are not, so Q is not nested within P. So, for both of them if you are looking for some variable, so you have to look into X, so, for both of them the access

link will be this X. So, this is the thing and here also this is the situation. So, this is the situation at line number 6.

Now, what about line number 11. So, at line 11, so this one, so what has happened? P has given a call to Q. So, sorry X has given a call to Q, the situation is like this, X has given a call to Q and this Q has given a call to R and R has given a call to P, so, that is the situation. So, X has given a call to Q or. So, this situation may be made more difficult also.

So, let us say suppose this is a situation. So, suppose this P call is somehow over and we are concerned about this Q call, then what will happen, the activation record it will be looking something like this. So, there I will have the portions for X, I will have for Q, I will have for R and P. And for this control link, so they will be like this. So, P has, R has called P. So, it is like this then this R Q has called R and then Q has this X has called Q. So, this is the situation as per as access links are concerned.

And as far as control links are concerned. So, this is the R is nested within Q. So, if you do not find some definition then for R you have to look into Q; so, it is like this. As far as Q is R is pointing to this. As far as Q is concerned, so Q is X, so, Q will be pointing to X and this P will also be pointing to X. P will be also be pointing to X. Only Qs, Rs access link will be pointing to Q ok. So, this is the situation.

Of course, you can make it more complex. So, you can try to create the record for this situation X calling P, P calling Q, Q calling R, R calling P. So, that is also possible. Like if I assume that it has it is from this line it has called P and P has given a call to Q and from Q it has come to R and from R it has come to P. So, the situation will be in that case the situation is something like this.

So, we have got X, then P1, then Q, then R, then P2 ok. Now, this the control links are simple control link, so P2 came from R, R came from Q, Q came from P1 and P1 came from X that is the control link. As far as access links are concerned, so this P1, Q and this P2 they will point to the X and then R will point to Q. So, that will be the situation for access link.

Now, at line 14, so this point. So, if I assume that the call is like this X to Q to R. X has given a call to Q and Q has given a call to R, then the situation is X, Q and R and now X

is. So, now, the control links will be like this from R to Q and from Q to X, because the call came from X to Q to R, at the access links will be like this because access link of R will be pointing to Q and access link of Q will be pointing to X. So, that is the situation.

At line number 17; at line number 17 so, it has just given a call to P. So, line number 17 the situation is from X, P has been called, so the situation will be simply like this. So, this is for X and this is for P. Now, the control link is like this and access link is also like this ok.

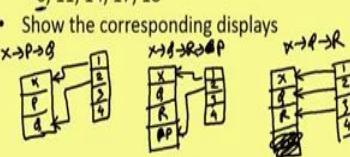
And X to Q, so the line number 18, so it is X to Q. So, if I assume there is X to Q. So, what will happen is that I will have two portions X and Q. So, Q will point to the X for activation for the for the control link and the access link will also be; access link will also be pointing to a X because whatever is defined in X. So, they can be available in Q, so, this way we can do it using control link and this is this access links.

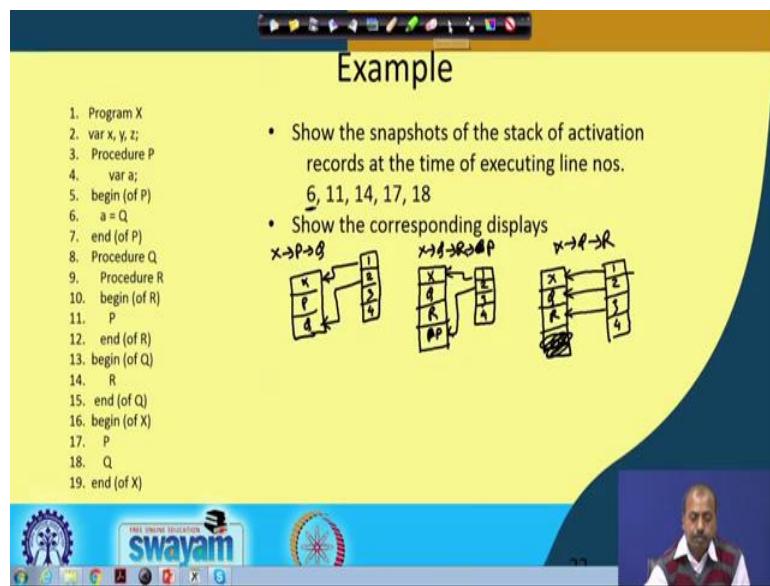
Now, next we will see next we will solve the next part of the problem where it is done using displays. So, how will it look like if we do it using displays?

(Refer Slide Time: 26:06)

**Example**

1. Program X  
 2. var x, y, z;  
 3. Procedure P  
 4. var a;  
 5. begin (of P)  
 6. a = Q  
 7. end (of P)  
 8. Procedure Q  
 9. Procedure R  
 10. begin (of R)  
 11. P  
 12. end (of R)  
 13. begin (of Q)  
 14. R  
 15. end (of Q)  
 16. begin (of X)  
 17. P  
 18. Q  
 19. end (of X)

- Show the snapshots of the stack of activation records at the time of executing line nos.  
 $\underline{6}, \underline{11}, \underline{14}, \underline{17}, \underline{18}$
- Show the corresponding displays  




The first situation is at line number 6; so, that is X to P to Q. So, X to P to Q, so we have got this activation record X, P, Q. And now since there are 4 procedure, so the display will have 4 entries in it. So, display we will have 4 entries, in it 1, 2, 3, 4 and then this, so the currently valid once are this X and Q, these two are only valid once. So, it will be

because if it if I do not get something in Q have to look in X. So, rest of the entries are not valid so, it will be like this.

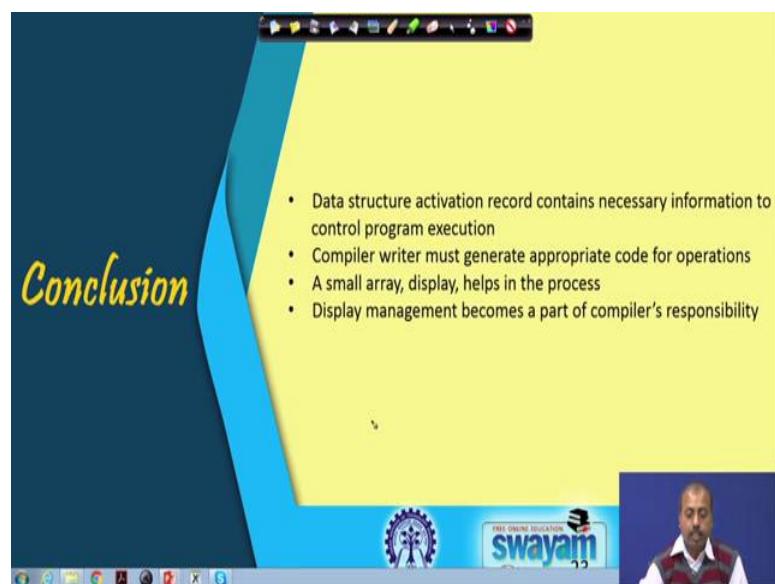
Similarly, at line 11, so if I say that the call is like this that X to Q to P to R. So, if the situation is like this then I will have the situation. The activation record will be like this X.

Student: (Refer Time: 27:14).

Q, P R I am not drawing the control link, so that you can at line 11 you know, so this is sorry this is R and this is P. So, this is R this is P and then I have got this display 1 2 3 4. Here, this 1 points to X and then this 2 it will point to P, because if I am looking if P is the currently valid 1 and if I do not find something in P I have to look in X only.

Then this one, so X to Q to R so if you looking for this situation then the display will be sorry the activation record will have 3 entries X, Q and R, my display has got 4 entries in it 1, 2, 3, 4 then X is there and R is defined within Q. So, this 2 will be pointing to this Q and 3 will be pointing to R ok. So, this way we can draw the you can make the displays, and that shows the how this displays are going to be updated and all.

(Refer Slide Time: 28:52)



Now, so the next we will come to the conclusion part. So, we have seen that this data structure activation record it contains necessary information to control program execution and compiler writer has to generate appropriate code for operations and a

small array called display it helps in this runtime environment management process, and display management becomes a part of compilers responsibility. So, the if the display is introduced then this management code also has to be put into the code that is generated.

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

**Lecture – 50**  
**Intermediate Code Generation**

So, in next part of our course will be looking into Intermediate Code Generation. So, after so, far we have seen techniques by which we can just tell whether a program is syntactically correct or not and we can also do some amount of semantic checks like if the if all the symbols are defined properly; whether all the types have been defined properly; whether this program has got some type error ok; so all those things we have seen. Like then, we have also in the last chapter we have seen like how to what are the responsibilities for this management of runtime environment. So, what the compiler should do to do that.

But the major task that a compiler does apart from this checking of the syntax and all is to say is to generate the code which will be finally executed by the processor. And I have said that like a there is a phase before this called intermediate code generation which actually is not a mandatory one but it may very often so this forms a part of this compilation process because what can happen is that today maybe you are generating code, you are general writing a compiler that will generate code for say processor 1. Tomorrow you want to write a compiler for the same language, but it is targeted to a different processor; processor 2.

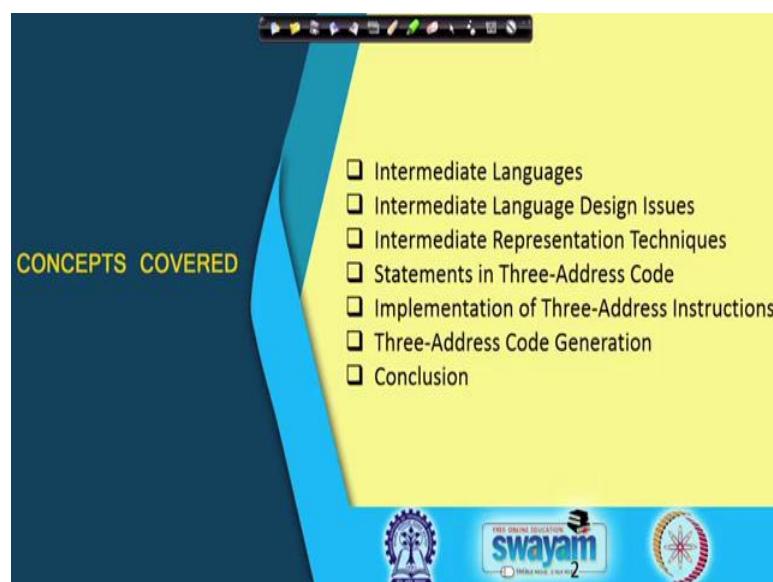
Now since the machine language of this processor 1 and processor 2 are different from each other. So, the code that you generated for processor 1 is not sufficient is not correct for processor 2 and there may be certain architectural features that are available in this processors such that if you can exploit those features the program will be more optimized.

So, that way we should try to generate we should try to have those things exploited. Now, if we concentrate on this entire compilation process for this machine 2 also from the very beginning, then the total amount of effort needed is just doubled ok. So, we so to make the situation simple; so, what is done is that normally we produce this code of this of a source language program up in some intermediary language and later on,

from this intermediary language the statements are converted into the target machine; so that is called the target code.

So, we have got the source program which we call source code that will be converted into an intermediate code and from the intermediate code it will be converted into this machine code or the target code. So, that is why this intermediary code generation often forms a very important part in this compilation process. And most of this language complexities are taken care of by this intermediary code generation phase such that when I go to the optimization phase so you do not need to be bothered much about the complicacies of the source language, syntax and semantics ok. So, they are already translated in some form and from that form we try to optimize.

(Refer Slide Time: 00:32)



So, this is what we are going to do in this particular chapter. So, as I said that this is an intermediary code or intermediate code. So, that has to have some language in which you write it. So, apparently it is a bit counter into it because he started with some source language program and then, we are converting it into some intermediary language program ok. But that will help us because this is intermediary language is going to be much simpler in their syntax compared to this source language that you are considering, so for which we are writing the compiler.

So, that way intermediary language is there going to be helpful. So, whenever I say so, so that is a language design issues, we will turn up like what are the types of statements

that I should have there; what we will be the complicity of expressions that we have; do you have a loop there; do you have a this case statement or so, which case statement or do you have if then else or do you have function calls do you have pointers? So, all these issues we will come up.

So, we have to address these things like if you say if you say that we will be generating target code in some language, then we have to some intermediate language. Then, what is the structure of that intermediate language that has to be told. So, this intermediate ah language representation techniques that also has to be thought about. Because ultimately so it may not be a source language program the text file this intermediate code may not be a text file. So, it may be that we if represent it in some other format, so that this representation becomes more efficient.

So, what are the statements that we are going to allow in Three-Address Code; Implementation of Three-Address Instruction like how are you going to implement it. So, that is the actually a part of the target code generation and this Three-Address Code generation process, how do you really generate code corresponding to this machinery this source language program statement. So, how do you really convert it into three address code statements so that is very important and then, we will be end up with a conclusion.

So, you see they this chapter forms the major part in the compilation process. So, far we were just trying to tell whether the program is syntactically correct or not. In some sense we were just passing the query of how to generate code. So, that we were just by passing. So, this chapter we will answer all your queries on that particular issue like how to generate the code.

(Refer Slide Time: 06:07)

The slide has a yellow header bar with the title "Intermediate Code". Below the title is a bulleted list of advantages:

- Compilers are designed to produce a representation of input program in some hypothetical language or data structure
- Representations between the source language and the target machine language programs
- Offers several advantages
  - Closer to target machine, hence easier to generate code from
  - More or less machine independent, makes it easier to re-target the compiler to various different target processors
  - Allows variety of machine-independent optimizations
  - Can be implemented via syntax-directed translation, can be folded into parsing by augmenting the parser

Handwritten annotations on the slide show the following diagram:

```
graph LR; A((ADD x,y,z)) --> B((y+2*w^5)); B --> C((t1=x+y)); C --> D((t2=x*y^2)); D --> E((ADD x,t1)); E --> F((MUL t1,t2));
```

The annotations illustrate the translation of a complex expression  $x + y + z$  into simpler intermediate code. It shows how the expression is broken down into temporary variables  $t1 = x + y$  and  $t2 = x * y^2$ , which are then used in subsequent additions and multiplications.

So, to start with what is an Intermediate Code? So, compilers are designed to produce representation of input program in some hypothetical language or data structures. So, this is the hypothetical language. So, this is intermediary code; so this is written in some hypothetical language is that you can thought about that you can think about. So, then how to do this thing; so, how to generate this hypothetical language program; so that will be seen.

So, then the representations between the source language and target machine language program so, so how do you represent them? Then, there are several advantages. So, what are the advantages? So, it is closer to target machine and hence easier to generate code from. For example, it very simple case may be like this that in your source language, you may write a complex expression like  $x$  equal to  $y$  plus  $z$  into  $w$  to the power 5 like that. Now it may so happen that while in where the target machine on to which you are doing this implementation, it has got add instruction, it has got multiply instruction it has got so like that.

So, it has got only these two instructions and then, this add instruction it has got only say ah it can accept only say two operands  $x$  and  $y$  such that after doing this  $x$ , we will get  $x$  plus  $y$ . Similarly, this multiply instruction it has only 2 operands  $x$  and  $y$  and after executing this, it will be getting something like this;  $x$  equal to  $x$  into  $y$ . Now, see this

instruction is going to be very complex as far as this basic add and multiplication instruction is concerned; but in three in this intermediary language code what we will do? We will not allow such complex expressions. So, we will allow expressions like say t1 equal to x plus y, t2 equal to x into z; so like that we will allow such that is.

So, the see here the operands that we have they are very close to the target language operands and this the number of operators that we have they are close to target language operators and this number of operands are almost same as the target language statement. So, whatever operands are how many as many operands are there. So, here also for the operators we will have similar number of operands. So, that way it is made most or most they it is 2 operand; sometimes, it is 3 operand etcetera. So, it is closer to the target machine and hence, it is easier to generate code from. Second important point that we have it is more or less machine independent.

So, this statements like say t1 equal to x plus y or t2 equal to x into z. So, it does not assume any underlying architecture for the machine. So, when I say add mul this sort of statement. So, it immediately assumes that the underlying processor, it has got instructions like add, mul etcetera. So, but here I am not assuming anything like this. What is the exact syntax what is what is the exact machine language statement for doing this addition; so that is not important here. So, you are representing this operation in some hypothetical language and writing it has only with the restriction that arithmetic operation, it will have only 2 operands ok; so that way we are writing it.

So, it is more or less machine-independent and thus, makes it easier to retarget the compiler to various different target processes. Otherwise what will happen is that there are so, if you look into this addition instruction itself, some processors you will find that it will so it will allow you to have 3 operand of format like add x, y, z such that the meaning is that z will get x plus y and some processors, they will allow you in this format add x, y telling that the operation is x equal to x plus y. So, in some cases it is a 3 operand instructions; some cases it is 2 operand instructions.

So, if you have generated code for a machine that supports 2 operand ah instruction and from there and now you want to modify the compiler so that it will be doing 3 operand. So, at it will support 3 operand instructions like this. So, they lot of changes are to be made, but you see that if you are doing it like this, then from here you can for the two

address you can generate like this and from three address you can generate like this. So, both are both can be done very easily; so that is why it makes it easier to retarget. So, this is called retargeting the compiler.

So, first the compiler was generating code for this machine and now it has been retargeted to generate code for this type of machines. So, that way there are many advantages. Then, it allows variety of machine-independent optimizations; machine independent optimizations are like this.

(Refer Slide Time: 11:18)

**Intermediate Code**

- Compilers are designed to produce a representation of input program in some hypothetical language or data structure
- Representations between the source language and the target machine language programs
- Offers several advantages
  - Closer to target machine, hence easier to generate code from
  - More or less machine independent, makes it easier to retarget the compiler to various different target processors
  - Allows variety of machine-independent optimizations
  - Can be implemented via syntax-directed translation, can be folded into parsing by augmenting the parser

Handwritten notes on the slide:

- $t_1 = x * 5$
- $t_1 = x + x + x + x + x$
- $\text{for } i = 1 \text{ to } 100$
- $x = y$
- $a[i] = x + z$
- $a[i] = t_1$
- $x = y$
- $a[i] = t_1$
- $a[i] = t_1$

Suppose, I have got say ah say one operation say  $t_1$  equal to  $x$  into 5. Now, it means that I will be multiplying  $x$  by 5 here. So, another possibility or doing it is  $t_1$  equal to  $x$  plus  $x$  plus  $x$  plus  $x$  plus  $x$ . So, in most in many processors what happens is that this  $x$  this multiplication takes lot of time compare to addition. So, if I break it up like this, then it will be easier to handle. Then, there are many other optimizations. So, that way like it may so happen that suppose, I am writing a loop like this for  $i$  equal to 1 to 100;  $x$  equal to 5 and then so, say  $x$  equal to  $y$  and then, I am writing  $a[i]$  equal to  $x$  plus  $z$ .

So, if I am writing like this, then you see this computation of this  $x$  plus  $x$  equal to  $y$ . So, assignment of  $y$  to  $x$  and then, this computation  $x$  plus  $z$ ; so, this is also going to repeat 100 times. A better code can be write this like I write like  $x$  equal to  $y$  and then, I take a temporary variable  $t_1$  and make it equal to  $x$  plus  $z$  and then, I put this loop like for  $i$  equal to 1 to 100,  $a[i]$  equal to  $t_1$ . Why is it good? Because this computations, I am not

doing this 100 times. So, if I can do this thing in some intermediary language so that is independent of the target machine. So, this is the machine-independent optimization that I am talking about. So, this type of optimizations can be carried out easily if we in the three-address code it is a in the intermediate code itself.

And it can be implemented via syntax directed translation mechanism and thus, it can be folded into the parsing by augmenting the parser. So, this is the other advantage like we have seen that this type checking can be done in the using syntax directed translation mechanism. So, here this intermediate code generation also can be done by using the syntax directed translation mechanism and as we are doing this. So, this code generation becomes simpler. This intermediate code generation process become simpler and we do not need a separate phase further; so it is integrated with the parser.

When the parser we will particularly the shift reduce parser. So, when they do this reduction step. So, at the time or doing the reduction, so we can give it some set of rules. So, that it will follow those rules and as a result, the intermediate code we will get generated. So, these are the advantages that we have with this intermediary this intermediary language programs; so we are going to use them.

(Refer Slide Time: 14:11)

The slide has a yellow header with the title "Intermediate Languages". Below the title, there is a bulleted list:

- Can be classified into – High-level representation and Low-level representation

Below the list, there are two columns of bullet points:

<b>High-level Representation</b> <ul style="list-style-type: none"><li>• Closer to source language program</li><li>• Easy to generate from input program</li><li>• Code optimization difficult, since input program is not broken down sufficiently</li></ul>	<b>Low-level Representation</b> <ul style="list-style-type: none"><li>• Closer to target machine</li><li>• Easy to generate final code from</li><li>• Good amount of effort in generation from the source code</li></ul>
---	--

At the bottom of the slide, there is a hand-drawn diagram. It shows three ovals: "Source Language prog", "Int.", and "M/c language". Arrows indicate a flow from "Source Language prog" to "Int.", and from "Int." to "M/c language".

The slide footer features the "swayam" logo and navigation icons.

So, what can be the type of Intermediate Languages? So, there can be different class different types of intermediate languages that have been reported in the literature.

So, they can be broadly classified into High-level representation and Low-level representation. High-level representations; so, they are closer to the source language. So, what we are trying to do is you can understand that we have got this source language; we have got this source language program and that we are translating to machine language program. So, this we are taking to machine language and we have said that intermediary language in program is somewhere here; so this is the intermediary. Now, you can make this intermediary language close to this source language or you can make it close to the target language.

So, if you make to close to the source language, so we call it a high-level representation and similarly, if you make it close to the machine-language. Then, we will call it a low-level representation. Now, which one is better? So, it is difficult to answer. So, if it is close to this source language program, then it may be very easy that to generate this intermediary code because the language constructs may be more or less similar. So, this is the amount of job needed for doing the translation maybe small. However, so this part will become difficult then. So, if this intermediary language is here ok, it is closer to the source language; then, this translation is going to take more time.

Similarly, if you take this intermediary language somewhere here, then this translation becomes may become very easy. There maybe one to one correspondence between this intermediary language statements and this machine language statements. But doing this part becomes difficult ok; it may become difficult. So, it is a very judicious choice like how much complex I should make my intermediary language representation; so that is a very judicious choice.

And high-level representations, so they are closer to source language program; they are easy to generate from input program, but the code optimization is difficult because the we are we are not very close to the machine language. So, many of the machine dependent optimizations; so they will they cannot be tried at this level and the input program is not broken down sufficiently. So, it we may not be able to apply many optimizations on that.

On the other hand this low level representation. So, they are closer to the target machine ok. They are easy to generate final code from. So, we can where as I was telling. So, there may be one to one correspondence at this point. So, that way this target code

generation may be very easy. But good amount of effort will be needed in this process from source program to this intermediary program. So, it may take good amount of effort. So, this way we can have different types of intermediate languages and they may have different advantages and disadvantages also. Now, how are you going to handle this situation like what are the different intermediary languages that have been proposed.

(Refer Slide Time: 17:28)

The slide has a yellow header bar with the title 'Intermediate Language Design Issues'. Below the title is a bulleted list of five items. At the bottom of the slide is a blue footer bar featuring the 'SWAYAM' logo, which includes a gear icon, the text 'FREE ONLINE EDUCATION', and 'SWAYAM' in large letters, along with the tagline 'LEARN MORE, LEARN BETTER'.

- Set of operators in intermediate language must be rich enough to allow the source language to be implemented
- A small set of operations in the intermediate language makes it easy to retarget
- Intermediate code operations that are closely tied to a particular machine or architecture can make it harder to port
- A small set of intermediate code operations may lead to long instruction sequences for some source language constructs. Implies more work during optimization

And before that we try to see what are the issues in the in such language design. First of all this set of operators in intermediate language must be rich enough to allow the source language to be implemented; so this is the first thing. So, maybe in my source language there is an exponentiation operator. Now, I should keep an exponentiation operator in the intermediate language also.

Otherwise, what will happen is that I will try to implement that exponentiation operation by means of multiplication in the intermediate language only to find that in the target processor already an exponentiation operator is available. So, again there will be reconversion from this chain of multiplications to exponentiation or many cases what happens is that this subtraction operation may not be supported or so if the subtraction is not supported, then instead of that maybe I can multiply with minus 1.

So, I would multiply with minus 1 and then add. Now, in the intermediary language; so if subtraction is not directly supported, then I may do like that only to find that in the target processor subtraction instruction is available. So, it is desirable that the set of operators

of intermediate language must be reach enough to allow the source language to be implemented; otherwise it becomes so difficult like if it is not possible to do something that is possible in the source language, but not in intermediate language; then, we cannot carry forward it. For example, maybe my source language is allowing me to or do of real number arithmetic, but my intermediary intermediate language does not allow.

So, naturally this real arithmetic cannot be taken forward to the next level; so that this type of problems can be there. So, this language the intermediate language that we are suggesting will not be applicable for this particular case. At similarly at this a small set of operations in the intermediate language makes it easy to retarget. So, this is the other issue like just being afraid that if I do not include source language operators in to my intermediate language. So, it may make it you less useful. So, we may incorporate many of this operators onto this intermediate language. But there then, then for each of them, we have need to have the corresponding translation into machine code the target for the target processor.

So, the retargeting becomes difficult in that situation. So, a small set of operations in the intermediate language makes it easy to retarget. So, if there are large number of operations; so that we will also make a difficult. Intermediate code operations that are closely tied to a particular machine or architecture can make it harder to port. Like there may be certain operations which are specific for a particular processor.

For example, in case of say hm say this X-86 architecture there is an instruction by which you can copy a block of memory locations from one region of memory to another region. So, that is the very useful instruction, very useful feature; but if we do if we do use that in case of say array copy or something like that in the intermediate language, then a processor which does not support it; so there it will become difficult.

So, we should not be very tightly guided by some target processor architecture and instruction set to decide upon the three this intermediate language instructions. On the other hand, a small set of intermediate code operations may lead to long instruction sequence for some source language constructs. So, that can also happen. Like if the if my instructions are very simple in case of intermediate language, then some source language program when we are trying to write in terms of intermediate language; then it may

require very large code. So, that can happen; so that has to be taken care of. So, it implies more work during optimization. So, more work will be necessary during optimization.

Because the code at this point itself become sub optimal and then, we are code optimization phase which is anywhere very costly; so, that is going to take more amount of time. So, these are the issues that we have with the so with this intermediate language design problem and we have to do something, we have to select a language for this intermediate representation; so that the issues are taken care off.

(Refer Slide Time: 22:07)

The slide has a yellow header bar with the title 'Intermediate Representation Techniques'. Below the title is a bulleted list of representation types:

- High-level Representation
  - Abstract Syntax Trees
  - Directed Acyclic Graphs
  - P-code
- Low-level Representation

At the bottom of the slide, there is a blue footer bar featuring the 'swayam' logo and other educational icons. A photo of a man speaking is visible on the right side of the slide.

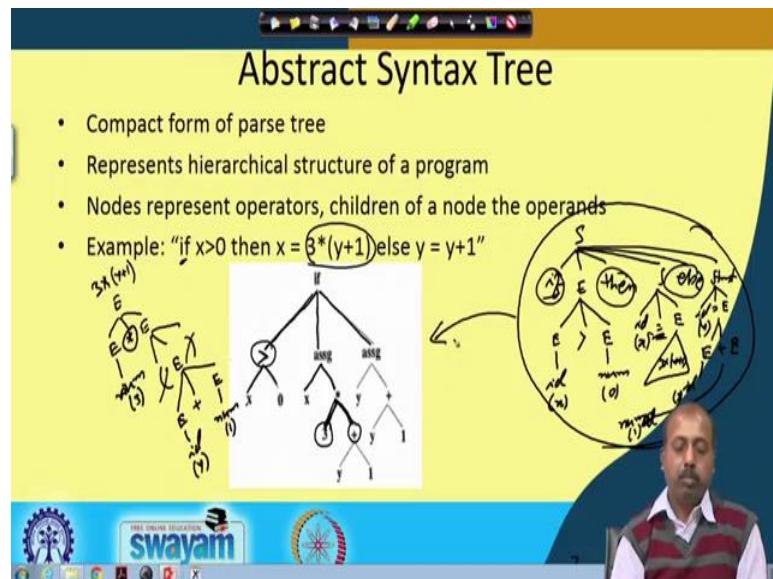
So, Intermediate Representation Techniques. So, we have got High-level Representations like Abstract Syntax Tree. Then, Directed Acyclic Graphs and something called P-code and there are some Low-level Representation techniques. So, we will see slowly like what are these methods.

So, this Abstract Syntax Trees; so, this is in some sense this is a representation of the parse tree only. So, this is the syntax tree. So, parse tree in some sense, so it represents the it is it is some sense it is represents the program. So, if you just take this parse tree say this terminals that are coming and convert them into some other form which may be bit different from the source language; so that may give rise to a syntax tree.

Then Directed Acyclic Graph; so this is for between the operators, we can between the operands we have got certain operators. So, we can represent them in the form of a DAG

and we can happy code so that we will see. And the Low-level representation, so they are ah very close to the machine language; so, we will see what are those things.

(Refer Slide Time: 23:23)



So, this is the Abstract Syntax Tree type of representation. The compact form of parse tree as I was telling that parse tree in some sense, so it represents the program. So, you can exploit that particular structure for representing some representing the program in a different form. So, this is a compact form of parse tree and represents hierarchical structure of a program. Nodes represent operators children of a node the operands like if I have got something like this that if x greater than 0; then x equal to 3 into y plus 1 else y equal to y plus 1.

So, we can say that so the parse tree for this depending on the programming language, so it will be some statement giving rise to if expression, then statement and this else statement and then, this E giving rise to so relational operator E greater than another relational operator and then these E giving raise to id and that is your x and this E giving rise to number that is the 0.

Similarly, this S giving raise to id equal to some expression and this expression somehow giving this 3 into y plus 1. This statement giving rise to again the statement that id equal to E; This id is y. This id was x and then, this is giving E plus E etcetera. So, you giving E plus E and then ultimately this E giving id; this E also giving id. So, this id this giving number which is 1 and this is y.

Now, you see. So, this is the syntax tree. So, the parser has generated this thing. So, you can just use a slightly different format from here, where we say that this if statement. So, we take the if statement and we represent in the abstract syntax tree since it has got 3 portions; one is the condition, one is the then part assignment, one is the else part assignment. So, you do not; we do not store this tokens explicitly like if, then, else then etcetera. So, those are not there not relevant because here it means that this is the this is the link for the condition. This is the link for the then and this is the link for else; so knowing that I do not need to do anything.

Now, for the condition part, so this is a greater than. So, this has to be stored because there can be different conditions. So, I need to store the condition that we have. So, this is greater than and after greater than we have got this x and 0. So, they are necessarily. So, similarly this assignment; so assignment means if they are there will be 2 sides; 1 is the left side, 1 is the right side. So, I do not need to store this equality in the syntax tree. So, this is equality is not stored. So, rather this x is stored on the left side and the right side, I store this expression 3 into y plus 1; so this node is star.

So, since this is a binary operator, there will be 2 operands. So, I do not need any other thing. So, I just store three here and then this is plus. So, it is y plus 1. In fact, if you look into this parse tree for this 3 into y plus 1. So, it is not very simple because the way it will be generated is something like this; so E star E. Then, this E giving number which is 3 and then this E giving bracket start E bracket close and then, this will give us E plus E. This E giving us id which is y and this E giving us number which is 1. So, this is such a complex thing, but we do we really need to remember all this things. Because as soon as we know that this is star, so I should have this number this left and right side of this left and right operands of this multiplication operation.

So, that is done here; so, this 3. So, this is the left side of the operand the left operand these are these are right operand and similarly, for plus also; so, this bracket and all, so they are not necessary just removed. So, that way from the parse tree, you can do some modifications so that the non essential portions of the parse tree, they are just trimmed out and what you have got is the bare minimum skeleton which will be able to represent the instructions or the statements of the programming language. So, in this so this is a representation which is very close to the source program. At the same time it removes

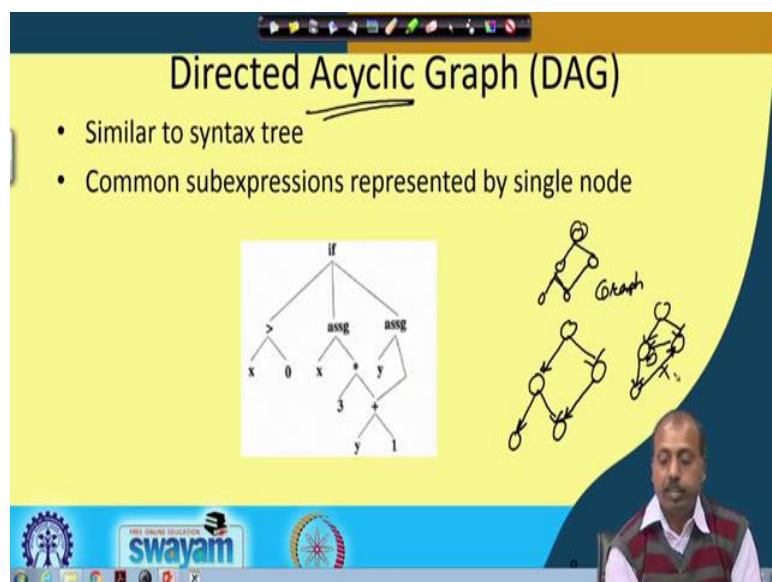
many of the details of the source program and still it captures the essence of the program for which we are trying to generate the target code.

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

**Lecture – 51**  
**Intermediate Code Generation (Contd.)**

The next high level representation that we will be looking into is known as directed acyclic graph.

(Refer Slide Time: 00:21)

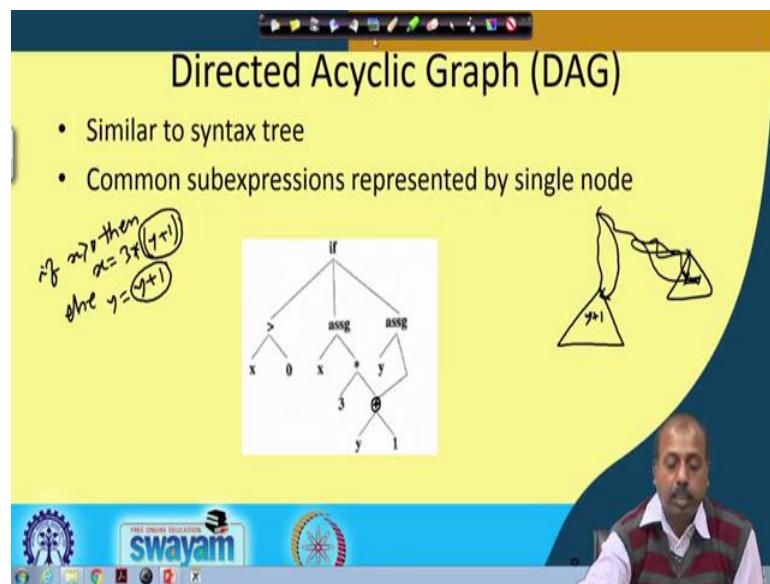


So, the previous representation that we had that was a tree. So, syntax tree and now this is a graph and you know that as a data structure the difference between tree and graph is that, in case of tree every node has got a unique parent node. And one so, it so that parent is only one every node has got at most one parent the root node does not have any parent, but otherwise like if this is a tree. So, the tree is always like this.

So, every node will have one parent accepting the root node which does not have any parent, but it is never the situation that one node has got two parents. So, that if that thing occurs then this is a graph and we call it a directed acyclic graph where this ages are directed so, from root node. So, it has got children so left and right children. Similarly from here so this is there. So, and then at some time so, if it is a graph then this type of ages will also be coming and it is not cyclic.

So, this is an acyclic graph. So, we do not have a situation where say like this. So, you do not have a situation like this so, where it creates a cyclic so, this is not there. So, this type of representation directed acyclic graph representation so, that is also used in many such intermediate language. So, it is similar to syntax tree in which common sub expressions they are represented by a single node.

(Refer Slide Time: 02:05)



So, like in the previous example that we had looked into so, it was like this that if  $x$  greater than 0, then  $x$  equal to 3 into  $y$  plus 1 else  $y$  equal to  $y$  plus 1. Now in this case so, this  $y$  plus 1 part so this expression part so this is becoming common ok. So, if you look into the parts tree so, in different portions of the parts tree I was  $x$  I was evaluating this  $y$  plus 1. So, one in the then part and another in the else part, so, in both the parts so we were evaluating that. So, in case of a directed acyclic graph; so, what is done is this common express common sub expression. So, they will be represented by a single node.

So, we do not do I do not keep a separate point a like this. So, this also points to this one only. So, that way it is helping like say here  $y$  plus 1. So, this node computes  $y$  plus 1 so, it was used in this expression also that 3 into  $y$  plus 1 and it is used in this statement also like  $y$  equal to  $y$  plus 1. So, in both assignment and both the assignment statements, so, this  $y$  plus 1 expression was common and that is taken out.

So, that gives rise to a graph representation and we will be we can go for a directed acyclic graph representation for that. So, that is one type of representation.

(Refer Slide Time: 03:33)

P-code

- Used for stack based virtual machines
- Operands are always found on the top of the stack
- May need to push operands to the stack first
- Syntax tree to P-code:

Computes architecture

Accumulator based  
ADD B  $\Rightarrow$  AC + B

stack based

$x = y * z^2$

Code to evaluate  $E_1$  OR Code to evaluate  $E_2$

Mult

Code to evaluate  $E_1$   
Code to evaluate  $E_2$   
 $r_0 = \text{pop}$   
 $r_1 = \text{pop}$   
 $r_2 = r_0 * r_1$   
 $\text{push } r_2$

push  
push  
mult  
push

The slide also features the Swayam logo at the bottom.

Then there is another high level representation which is known as P code. So, this is used for stack based virtual machines. So, what is a stack based virtual machines? So, virtual machine means the machine does not exist in actual ok. So, this is a conceptual machine you can say may be there are some machines which are built around this concept, but it is, but the basic idea is that this is a stack based machine.

So, like if you look into this computer architecture, then there are different types of architectures that we can think about or we can come across one is known as accumulator based architecture. So, in accumulator based architecture what happens is that all the arithmetic logic operations that you are doing one of the operand and the result.

So, that is always the accumulator like we have got instructions like add B which in mean that the accumulator A, we will get A plus B ok. So, this accumulator becomes a very important register and that way if all the operations arithmetic logic operations the accumulated is a part of it.

On the other hand this stack based machines, so, it will assume that we do not have any such registers or things like that. Rather there is a stack and whenever any operation is needed to be done. So, it will assume that the operands are always available in the stack. So, any operation to be done, so, it will be taking out 2 top most entries do the operation

and then it will be pushing the result back onto the stack. So, it is like this that suppose I have got say there this one say  $x$  equal to  $y$  into  $z$ .

Then what it will do it will have this it will be push putting this  $y$  and  $z$  into the stack and then when this operation has to be done so, it will take out this  $z$  and  $y$  from the stack. So, you can say that it is as if it will be doing 2 pop operations to get this  $y$  and  $z$ , then after that it will do a multiplication operation or in some sense in some cases you can say that the multiplication operator automatically it will pop out 2 topmost entries from the stack and do the multiplication on them. And then it will put the result back on to the stack. So, it may be like this that is suppose I am doing this operation.

So, multiplying two expressions  $E_1$  and  $E_2$  sub expressions  $E_1$  and  $E_2$ . So, somehow we have already have got the code for evaluating  $E_1$  code for evaluating  $E_2$  and then I say multiply. So, when it is there so that will mean that the operands after finishing this code to evaluate  $E_1$  the final result is already put into the stack. So, the final result of  $E_1$  is already available in the stack, then we have got the code to evaluate  $E_2$ . So, after executing this is the final result will be available in the stack.

Then it comes to this MULT for a MULT instruction so, it will take out these two values to the multiplication and put the result back onto the stack or if you are looking into a more detailed version ok. So, that then it will look like this there is a code to evaluate  $E_1$  code to evaluate  $E_2$ , then  $r_{naught}$  equal to pop. So, that is why they are the top most entries popped out and it is kept in  $r_{naught}$  then  $r_1$  equal to pop. So, that is also taken out of the stack. And then in  $r_2$  we do this multiplication  $r_0s$  into  $r_1$  and then we do push  $r_2$ . So, result is pushed into the stack so, this is called P code type implementation.

So, this is a high level representation so, where it is close to the stack based representation, but the difficulty that we have is definitely if the target is a accumulator based architecture, then converting it into target code becomes difficult so, that is there, but in still this is one of the some this is a technique for doing this code generation ok.

(Refer Slide Time: 07:53)

Low-level Representation – Three Address Code

- Sequence of instructions of the form " $\underline{x} = \underline{y} \text{ op } \underline{z}$ "
- Only one operator permitted in the right hand side
- Due to its simplicity, offers better flexibility in terms of target code generation and code optimization

$x = y * z + w * a$        $t_1 = y * z$   
                                 $t_2 = w * a$   
                                 $x = t_1 + t_2$

$x = y * z + w * a$   
                                 $t_1 = y * z$   
                                 $t_2 = w * a$   
                                 $x = t_1 + t_2$

**swayam**

So, next will be looking into the low level representations which are very common in this intermediate code; intermediary code generation which is known as three address code. In three address code the name came the name three address came from the fact that almost all the instructions.

So, they will have three addresses three address components in it ok, it will have three address components in it like most of the instructions are of the form  $x$  equal to  $y$  of  $z$ . So, you need to tell now what is the operand  $y$  and what is so, you need to tell the this operands  $x$   $y$  and  $z$ . And since they are identifiers or variables or numbers so, I will have three addresses for  $x$   $y$  and  $z$ . So, that is why the name is for three address code and only one operated is permitted on the right side.

So, you cannot have an instruction like  $x$  equal to  $y$  star  $z$  plus  $P$ . So, that is not possible because on the right hand side we can have only one operator. So, if you have to write like this then first of all we have to write like  $x$  equal to say  $t_1$  equal to  $y$  in to  $z$  and then write  $x$  equal to  $t_1$  plus  $P$ . So, like here so you see that we have got an example  $y$  into  $z$  plus  $w$  into  $a$ , first  $t_1$  equal to  $y$  into  $z$   $t_2$  equal to  $w$  into  $a$ , then  $x$  equal to  $t_1$  plus  $t_2$ . So, it is a very refined form of the complex expressions that you may have in the source language program, but at the same time which is not losing the meaning of the source language program ok.

So, everything is maintained but it is more elaborate and this individual instructions that we are having or the individual statements that we are having here so they are very simple. Due to its simplicity it offers better flexibility in terms of target code generation and code optimization. So, you see that most of the underlying processors they will support this three address format ok. So, in some machines what may happen is that you have got only two operand instructions some machines, you can have three operand instructions, but whatever it is so, it is very easy to convert this statements into that form.

So, that is why this is one of the most common formats in which these three address code is generated by the compilers. So, will be also discussing with this three address code based in policy.

(Refer Slide Time: 10:29)

## Statements in Three-Address Code

- Intermediate languages usually have the following types of statements
  - Assignment
  - Jumps
  - Address and Pointer Assignments
  - Procedure Call/Return
  - Miscellaneous

Now, statements in three address code. So, they are having this intermediate language that uses three address code, usually have these types of statements assignment jump address and pointer assignments procedure call return. And there is another category called miscellaneous. Now you see that almost all the major programming language constructs so, they are taken into consideration here.

So, we have got the assignment statement jump for go to. So, address pointer arithmetic is there and there is a the if then statement is also there. So, we will see that will come under the miscellaneous category.

(Refer Slide Time: 11:15)

Assignment Statement

- Three types of assignment statements
  - $x = y \text{ op } z$ , op being a binary operator
  - $x = \text{op } y$ , op being a unary operator
  - $x = y$
- For all operators in the source language, there should be a counterpart in the intermediate language

Assignment statement which forms the hard core part of this three address instructions. So, there are three types of assignment statements; one is like  $x$  equal to  $y$  or  $z$ . So, operator being a binary operator sometimes, we can have an unary operator say like there can be an operator like if  $x$  and  $y$  are Boolean variables.

Then I can have  $x$  equal to not of  $y$  ok. So, in that case so this not is an unary operator so it takes only operand. Whereas, if I say  $x$  equal to  $y$  and  $z$  then this and is a binary operator like this or so, in arithmetic domain, so, we have got this unary minus  $x$  equal to minus of  $y$ , where this minus is a unary minus. Whereas, if I write like  $x$  equal to  $y$  minus  $z$  then these minus is a binary minus though so, they are meanings are totally different.

So, though both of them appear to be minus symbol, but they are different. So, in if you looking into any real compiler design task, then you will find that it is given to the lexica and analyzer to differentiate between these two situation in some one case it returns unary minus as a token in the second case it returns minus as a token.

So, that way sometimes it is a bit confusing and the third type of assignment is  $x$  equal to  $y$ . So, there is no operation in the value of  $y$  is assign to  $x$ , for all operators in the source language they are should be a counter part in the intermediate language. So, that is mast because otherwise you will not be able to generate all you will not be able to have easy code generation into intermediate code. So, you have to think about converting complex

source language operators into simpler operators of this three address code language three address code. And that is that often brings laws of optimization.

So, that way it is not advisable that we should have a three address code three, this intermediate language such that the less number of operator less operator said they are then the source language operators.

(Refer Slide Time: 13:45)

Jump Statement

- Both conditional and unconditional jumps are required
  - goto L, L being a label
  - if x relop y goto L

if  $x > y$  then L1  
 $a = b + c$   
 $x = y - z$

L2  
 $a = b - c$   
 $x = y + z$

goto L1  
goto L2

So, once that is said so, will be looking into this operators some more statements of this three address code one is known as the jump statement. So, we have got this goto L where L is a level so, you can have so this L is again a hypothetical one because, this is for this is for just an intermediate representation ok.

So, in the so I can just use any symbol here as L like in a program sorry so, any program so you can have at this point say go to L so, you go to say L1 and this L1 may be a level here some statement may have L1 before that so, that will mean that this level is L1. So, when it is assumed that when this statement will be executed so, it will be the control will be coming to L1. Similarly there can be a go to L2 and L2 may be somewhere here. So, this way so this goto L is a generic go to statement of to the level L. And we have got this conditional go to, so, if x relational operator y go to L so ok.

So, this is the so, if then else type of statement so, if x relop y go to L. So, if this statement is there so, I do not need any separate if then else statement for example, if I

tell you like this that if  $x$  greater than  $y$ , then  $a$  equal to  $b$  plus  $c$ , then say  $e$  equal to  $f$  minus  $g$  else  $a$  equal to  $b$  minus  $c$ ,  $e$  equal to  $f$  plus  $g$  say something like this. Then I can do it like this that I can I do it so, I can write like this. So, if  $x$  greater than  $y$  go to L say L1. And then here I write like  $a$  equal to  $b$  plus  $c$   $e$  equal to  $f$  minus  $g$ , then  $a$  go to L2 and this is my L1.

So, where I write like  $a$  equal to  $b$  minus  $c$   $e$  equal to  $f$  plus  $g$  and this is my L2. So, you see that in this language I do not need the then and else part of the if statement. So, if then else type of constructs they can be converted into this a if condition goto some level type of statements. So, that is done very often. So, we will see that this jump statements can be useful for doing this.

(Refer Slide Time: 16:39)

**Indexed Assignment**

- Only one-dimensional arrays need to be supported
- Arrays of higher dimensions are converted to one-dimensional arrays
- Statements to be supported
  - $x = y[i]$
  - $x[i] = y$

The slide also features a hand-drawn diagram of a 3D array structure labeled  $a[100][100][100]$ . The diagram shows a cube divided into smaller cubes, with indices  $(x, y, z)$  assigned to each corner. A person's face is visible in the bottom right corner of the slide.

So, it also supports some array type of structure because array is now so, common in most in many of the almost all the programming languages. So, if you do not support array as a basic operation in the three address code, then it becomes very difficult to generate the corresponding code in the target machine. So, or so it becomes very difficult for converting the source language array indices to the intermediate code, so, only one dimensional array is supported.

So, in your source language they are may not be any restriction on the number of dimensions, but ultimately what is happening is that see your memory on to which you will be storing the will be storing this array so, that is one dimensional in nature. For

example, if I have got a 2D array a 100 by 100, then also I will be storing them in this fashion say a[1,1], a[1,2] up to say a[1,100], then a[2,1] so, I will be storing in them in this fashion ok.

So, I do not need to really have a 2D representation, because while I am talking about the target code. So, target code it will be accessing memory only in a one dimensional fashion. So, even if my target machine supports this array access. So, there is no point having multidimensional array in the target code, so, that the target code will not support multidimensional array.

So, this conversion from this multidimensional array access to single dimensional array access, it is better done at the intermediate code generation phase itself. So, that at the target code generation stage so, we do not be bothered about this error dimensions and also. So, there we can concentrate more on other optimizations. So, arrays of higher dimension so, they are converted to one two one dimensional arrays so, that makes it the so, this conversion code has to be generated by the compiler.

(Refer Slide Time: 18:53)

The slide has a yellow header bar with the title "Indexed Assignment". Below the title is a bulleted list:

- Only one-dimensional arrays need to be supported
- Arrays of higher dimensions are converted to one-dimensional arrays
- Statements to be supported
  - $x = y[i]$
  - $x[i] = y$

To the right of the list, there is a hand-drawn diagram. It shows a variable  $t$  assigned to a value involving a multidimensional array  $x[i][j]$ . The diagram illustrates the conversion of a 2D index  $(i, j)$  into a 1D index  $i \times N + j$ . A separate part of the diagram shows a 2D array  $x[m][n]$  with indices  $i$  and  $j$ .

The footer of the slide features the "swayam" logo and various navigation icons.

So, it will do that conversion and so, I like I can have definitely have a statement in my source language program like say  $t$  equal to say  $x[i,j]$ . And then in the three address code I should have the appropriate code so, that this  $x[i,j]$  so this is converted into some one dimensional array.

And you know that based on the size of these array the dimensions of this array. So, you can very easily compute the corresponding index at which you have to refer to so, that can be that is done in fact, it is if the psi, if the array is of dimension say capital M by capital N, then each x axis is it is having say n such entries.

So, this i into N plus j. So, if you go to that so this if you are storing this x ray, in this fashion 1 1,  $x[1,1]$ , then  $x[1,2]$  in this fashion, then you can do it in this way. So, that way it will be so, if this is this is not i this is i minus 1, i minus 1 into N plus j.

So, for example if it is 1 1 then i this will be this will come to this first lot. So, if it is 1 2 that will come to the second slot like that. So, that way it will be doing the conversion. So, any data structure book on this array so, that will discuss on that. So, this arrays of higher dimension they can be converted to one dimensional array and the compiler has to embed this code into the three address code that is generated ok.

So, this we will see that it is exactly what is done in the code generation phase ok. So, this will be done so, this statements to be supported are only one dimensional arrays x equal to yi or xi equal to y. So, these are the two star type of statement that we need to support.

(Refer Slide Time: 20:49)

- Statements required are of following types
  - $x = \&y$ , address of y assigned to x
  - $x = *y$ , content of location pointed to by y is assigned to x
  - $x = y$ , simple pointer assignment, where x and y are pointer variables

Now, address and pointer assignment. So, this is another very important issue. So, there is may be a question like whether we should support this pointers or not this address

arithmetic or not, but this is required because many programming languages they support this address pointers and all.

So, if it is not supported then it will become very difficult to convert those statements like suppose for example, in c language so you have got this integer pointer int star P and at some point we have got a statement like a equal to start P. Now, how are you going to handle the how are you going to write it in terms of some lower level language program, in which this pointers are not supported it pointers are not supported, it is really it is almost impossible to write it in the proper with proper flavor.

So, we can always do something so, that it will give you some other representation which will not be truly a pointer, but something close to there, but you will never get a true pointer there. And the underlying machine so, most of the underlying processors the process, they support this type of pointers and all so, they have they support indirect access. So, at the architectural level pointers are beings supported.

So, the machine code that we are; that we are going to use they are going to support pointers. So, if the intermediate could does not support pointers, then of course, that is of no use. So, that is why in intermediate code also we need to support pointers. So, statements required for pointers are like this that x equal to ampersand y. So, address of y is assigned to x, then x equal to start y the content of location pointed to by y is assigned to x and x equal to y so, simpler pointer assignment so, x and y both are pointer variables so, you write like x equal to y. So, it does not tell anything about this implementing pointers and all.

So, it may be in some if you are high language supports that. If it supports that pointer arithmetic in terms of say making x equal to x plus 1 and things like that. So, if this sort of things are supported then you have to have those features also incorporated into the three address code.

But for our understanding we assume that our pointers are simple so, we do not have any pointer arithmetic supported only pointer assignment can be there. So, in that case this representation is good enough ok. So, if you are in c type of language you have this pointer arithmetic which is mostly used for accessing arrays using pointers, but that may not be necessary in some other programming languages.

So, that way we so. For our discussion, so, we will not take this pointer arithmetic into consideration; we will take them simply as pointer assignment.

(Refer Slide Time: 24:03)

Procedure Call/Return

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

param  $x_1$   
param  $x_2$   
...  
param  $x_n$

$P(x_1, x_2, \dots, x_n)$   
 $P(\dots)$

param  $x_1$   
param  $x_2$   
param  $x_3$   
enter  $f$   
return  
return  $x$

retrieve  $x$ , Save returned value in  $x$

Procedure call or return so, this is of course, important. So, a call to procedure  $P$  so, there are two part in it. So, we have to pass all these parameters so this way the this is the call for handling for doing this parameter passing. So, this  $x_1, x_2$  up to  $x_n$  so, we call say that they are all parameters ok, so, this has to be done.

Now how these parameters will be implemented in the target language, so, that is a question. So, you do not answer that at this level, but we keep a note that this  $x_1, x_2, x_n$  so, they are parameter. So, if we have already seen that in runtime environment management so, we need to create the activation record.

So, once this parameter statements are found so, this compiler we will know that the when it is generating the target code that there will be activation record created and these are the parameters going to be used in the activation record. So, when the activation record we will get created in that this  $x_1, x_2, x_n$  so, they will be given space. So, you will know how big and activation record you need to make, once you know this parameter. So, three address code is not for execution. So, we just keep a we it is just for keeping a note that what will be required at the target level.

So, that is what is kept here so, this is  $x_1$  to  $x_n$ . So, those parameters are kept and to proceed to a procedure is implemented using the statements like enter f level f return and return x retrieve x etcetera. So, enter f is so if I have got a procedure call. So, if this is the main routine and somewhere here I have given a call to procedure P with parameters like  $x_1, x_2, x_3$ .

Then what it will do. So, it will put this param statements param  $x_1$ , param  $x_2$ , param  $x_3$  like that. And then it will tell enter P. So, enter P so this will be; this will be of so, when if this is enter P statement comes so, by this time the activation record has already been made so, this activation record will be pushed into the stack and then it will be branching to the subroutine P.

Similarly, this leave f so, this is for the cleanup actions. So, that will be called by this callee routine. So, there so, that will be done by the callee routine, so, it will be cleaning up the this activation record area and all. Similarly this return x retrieve x so, this so this return means there is no return value. So, it will just return from the procedure. So, this return values are not there, but in return x there is a written value. So, as a result if you have return like something like say equal a equal to P so, P  $x_1, x_2, x_3$ .

Then after returning so, this whole procedure should have some return value x that should that will be assigned to a. So, this return value is needed so, that is obtained by return x statement. So, the return value of P is taken into return x. So, if we if this is the procedure P. So, at the end of it you can have a return x statement, that will do return. And this retrieve x so, this is will also save return value in x so, that is also there.

So, with these are actually synonymous so many so you can use any of these statements. So, thus we see that the procedure call return. So, basic procedure call return is also implemented in a three address code. So, that we can also implement this procedure calls sub subprogram calls and returns. So, this three address code that we is not very simpler very low level at the same time it is not very high level also. So, it is exactly we can say it is exactly at the middle of that two end the source language and the target language.

So, it is almost at the middle of the thing so, that it is equidistant from both the sides. So, that makes it interesting because if you want to target two different target processors. So, you can put half of the effort you can say in some sense half of the effort of total

compiler writing for generating the generating a new compiler for the second target machine.

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

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

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

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

(Refer Slide Time: 01:01)

**Procedure Call/Return**

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

param ( $x_1, x_2, \dots$ )  
 $x = f_1(a_1, a_2)$   $B_1$   
param  $a_1$   
param  $a_2$   
enter  $f_1$   
retrieve  $t_1$   
 $x = t_1$   
return...  
local vars

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

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

So, it has been converted into if there are n parameters, then it will be converted into n parameters statements, parameter x<sub>1</sub> to parameter x<sub>n</sub>. And then for processing for a procedure call so, it is done as enter f. So, enter f at this point several actions may be necessary depending upon the language for which we are generating the code and the target machine.

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

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

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

So, that will be at the procedure you can have at the function end. So, you can convert into return x and retrieve x. So, this save return value in x. So, this will be return value will be saved in the variable x. So, that way it will be converted like. If, I have got some say functions say f1 being called with some parameter a 1 a 2. So, at the time of call I should have these three addressed statements param a 1, then param a2, like that and then I should have enter f1 ok.

Then, at the body of when I am translating the f1 part, I should have these local set up part. So, this is a local part so, this so, this enter f1. So, if the local part may be there they may be created so, at the end. So, I can have these return or return x etcetera and then this leave f will be coming at the bottom.

So, leave f1 will come at the bottom. So, that will be leaving thing and this return statements should come before that and this in the main program. So, I can have this retrieve thing, like it may be that I have got a call like x equal to f1 something. So, retrieve the return value in to some temporary variable t1. And then I can have x equal to t1. So, this may be the final code that is generated the 3 address code that is generated. So, there may be some variant of this, but this is one possible strategy by which you can do this translation.

(Refer Slide Time: 05:07)

Miscellaneous Statements

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

label L

lsp( ) goto L

L B1

FREE ONLINE EDUCATION SWAYAM

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

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

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

So, it is like this for jumping to a procedure we said that it is like goto L and then what is this L? So, somehow if this L is a procedure ok. So, maybe I have in the main in the source language program I have got a call for a procedure. So, I have got a x equal to f1 something ok. Then, they are their the goto L is there. So, it is expected that L, I should start the procedure f1. Here, I should have the procedure f1.

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

(Refer Slide Time: 07:13)

The slide has a yellow header bar with the title "Three-Address Instruction Implementation". Below the title is a bulleted list:

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

Below the list are two diagrams showing quadruples:

- Left Diagram:** An example of a quadruple for addition. The operation field (Op) contains "PLUS". The first source field (Src1) contains "y" with a small circle indicating it is a pointer. The second source field (Src2) contains "z". The destination field (Dest) contains "x". The text above the diagram is "x = y + z".
- Right Diagram:** An example of a quadruple for a jump. The operation field (Op) contains "JMP\_GE". The first source field (Src1) contains "t1". The second source field (Src2) contains "t2". The destination field (Dest) contains "instruction labelled L". The text above the diagram is "if t1 >= t2 goto L".

At the bottom of the slide, there is a logo for "swayam" and various navigation icons.

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

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

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

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

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

(Refer Slide Time: 08:55)

The slide is titled "Example". It shows a snippet of "Intermediate code" enclosed in a box. The code is as follows:

```
if x+2 > 3*(y-1)+4 then z = 0
t1 = x + 2
t2 = y - 1
t3 = 3 * t2
t4 = t3 + 4
if t1 <= t4 goto L
z = 0
Label L
```

A handwritten note "Intermediate code" is written next to the box. The slide has a yellow background and a blue footer with the "swayam" logo.

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

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

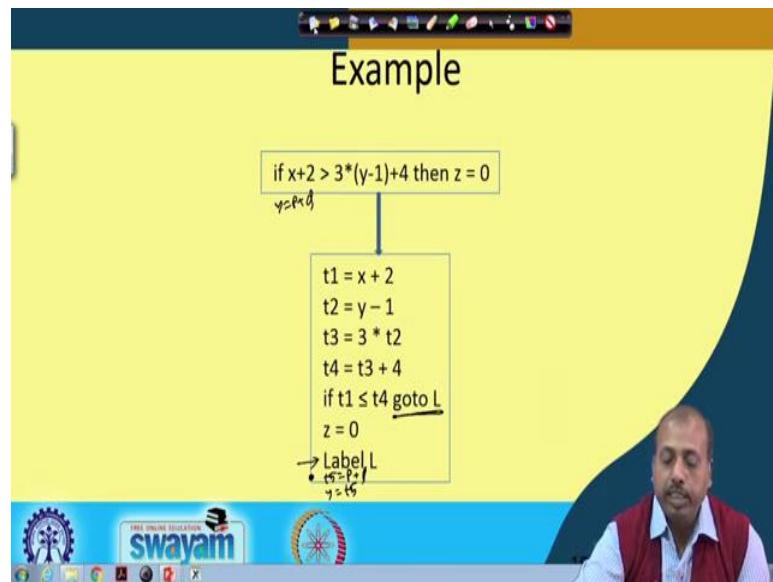
So, here is a single line source single source language statement corresponding to that. So, many three address code instructions have been generated. So, let us try to check it. So, first this  $x + t_1$  equals to  $x + 2$ . So,  $t_1, t_2, t_3, t_4$  so, they are all temporary variables. And you will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

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

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

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

(Refer Slide Time: 11:23)



So, if there was a statement here, if there was a statement like say  $y$  equal to  $P$  plus  $Q$  after doing this, then we will have this code for  $y$  equal to  $P$  plus  $Q$ . So, another  $t5$  equal to  $P$  plus  $Q$ , then  $y$  equal to  $t5$ . So, that would have been done, but here it is so, so when it says when it says goto  $L$ , label  $L$ . So, it will start it will come to this label  $L$ .

So; that means, my execution should start from this point, if  $t1$  is less or equal  $t4$ . So, this way three address codes are will be generated. So, will see how using different translation rules or syntax directed translated mechanism, how can we generate such three address code.

(Refer Slide Time: 12:03)

## Three Address Code Generation - Assignment

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

**Attributes for non-terminal E:**

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

**Attributes for non-terminal S:**

- S.place – sequence of three-address statements

**Attributes for terminal symbol id:**

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



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





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

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

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

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

(Refer Slide Time: 14:11)

## Three Address Code Generation - Assignment

<b>Grammar:</b> $S \rightarrow id := E$ $E \rightarrow E_1 + E_2$ $E \rightarrow E_1 * E_2$ $E \rightarrow -E_1$ $E \rightarrow (E_1)$ $E \rightarrow id$	<b>Grammar Rule</b> <b>Semantic Actions</b> $S \rightarrow id := E$ $S.code := E.code    gen(id.place := 'E.place)$ $E.place := newtemp();$ $E.code := E_1.code    E_2.code    gen(E.place := 'E_1.place +' E_2.place)$ $E.place := newtemp();$ $E.code := E_1.code    E_2.code    gen(E.place := 'E_1.place *' E_2.place)$ $E.place := newtemp();$ $E.code := E_1.code    gen(E.place := 'uminus' E_1.place)$ $E.place := E_1.place;$ $E.code := E_1.code$ $E.place := id.place;$ $E.code := ::$
<b>Attributes for non-terminal E:</b> <ul style="list-style-type: none"> <li>• E.place – name that will hold value of E</li> <li>• E.code – sequence of three address statements corresponding to evaluation of E</li> </ul> <b>Attributes for non-terminal S:</b> <ul style="list-style-type: none"> <li>• S.code – sequence of three-address statements</li> </ul> <b>Attributes for terminal symbol id:</b> <ul style="list-style-type: none"> <li>• id.place – contains the name of the variable to be assigned</li> </ul>	
 <ul style="list-style-type: none"> <li>• Function <code>newtemp</code> returns a unique new temporary variable.</li> <li>• Function <code>gen</code> accepts a string and produces it as a three-address quadruple.</li> <li>• '  ' concatenates two three-address code segments</li> </ul>	



So, if I tell you that E dot code has got so, this E, it has got 2 portion; one is the place and another is the point at code. So, this is the code that is required for evaluating E. And E dot place is the name of the place or name of the temporary that finally, holds the value of E. So, if the last assignment here last computation here is say t5 equal to something and that holds the value of this E dot place then these value of the expression E, then this E dot place will be equal to t5 ok. Now, given this thing so, if I just look up by one more step, this id assigned as E. Then, what is the thing to be done for generating the code for S.

So, in the code for S, I should have the first the code for E code for evaluating E. So, this should come as it is. And after that I have to do the assignment that is this id dot place id dot place holds the place name of the id, like actual statement may be a equal to b plus c. So, this id, so, id dot place it will hold the value a. So, then this a equal to so, this I will have this code followed by this extra code, that id dot place assigned as E dot place. I will have this extra code id dot place assigned as t5.

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

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

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

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

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

Similarly, if you look into say this rule say E producing within bracket E1, then E dot place equal to E1 dot place. So, because the here whatever is the so, variable that hold the, whatever may be the symbol that holds the value of E, that is E1 dot place. So, that is that itself will hold the value of E ok. So, E dot place equal to E1 dot place. And then these E dot code is also equal to E1 dot code; so, that much is fine.

Now, let us look into this rule E producing unary minus E1. At this point so, I have to have something like this. So, I will so, for generating the code of E first I have to include the code for E1. So, I have to include the code for E1 followed by I have to generate a

piece of code, where this I need a new E dot place. Because, E dot place whatever temporary has been used in the computation of E1.

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

So, this way this E dot place equal to newtemp. So, this newtemp function will return a new temporary variable. And that will be assigned to E dot place. So, as if that temp new temporary variable. So, it will hold the value of this expression E. And what is the code for E? Code for E is the code for E1 so, this particular symbol. So, this concatenates two, three-address code segments.

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

So, E dot place assigned as unary minus E1 dot place. So, this particular code will be generated. So, this unary minus E1 dot place. So, this unary minus thing will introduce a E1 dot place was holding the value of E1 is the name of variable that holds the value of E1. So, that will be negated by this unary minus symbol and that will be assigned to E dot place.

Similarly, if you look into say this particular statement E producing E1 or star E2, then also in the code for E I should have the code to evaluate E1, I should have the code to evaluate E2 and then I should generate an additional piece of code. So, that will multiply E1 dot place and E2 dot place.

So, it is done in this fashion. So, E1 dot code concatenated with E2 dot code concatenated with this gen function. So, gen function it accepts a string and produces it as a three address quadruple. So, this is the whole string that is accepted that is taken as

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

So, that temporary assigned as E1 dot place so, a temporary variable which holds the E1's final value and then multiplied by E2 dot place. So, E2 dot place holds the final value of E2. So, that way they will be so, this particular piece of code will be concatenated with the code for E1 code for E2 and then this one. And then the similarly this E producing E1 plus E2. So, this is also similar as for as code generation is concerned.

So, E dot place equal to new temp and then E dot code equal to E1 dot code concatenated with E2 dot code concatenated with generate E dot place assigned as E1 dot place plus E2 dot place. And then off course I will need this particular statement E dot place equal to newtemp to hold the to get new temporary variable. So, this way for arithmetic assignments statement so, we can draw the corresponding we can draw the corresponding parse tree and then follow this particular semantic actions ok.

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

(Refer Slide Time: 23:17)

The slide shows the derivation of the expression  $x := (y+z)*(-w+v)$ . The parse tree on the right illustrates the structure of the expression, with nodes representing terminals (y, z, \*, -) and non-terminals (E, t<sub>1</sub>, t<sub>2</sub>, t<sub>3</sub>, t<sub>4</sub>, t<sub>5</sub>, t<sub>6</sub>, t<sub>7</sub>, t<sub>8</sub>, t<sub>9</sub>, t<sub>10</sub>, t<sub>11</sub>). The reduction table on the left details the steps:

Reduction No.	Action
1	E.place = y
2	E.place = z
3	E.place = t <sub>1</sub> E.code = {t <sub>1</sub> := y + z}
4	E.place = t <sub>1</sub> E.code = {t <sub>1</sub> := y + z}
5	E.place = w
6	E.place = t <sub>2</sub> E.code = {t <sub>2</sub> := uminus w}
7	E.place = v
8	E.place = t <sub>3</sub> E.code = {t <sub>3</sub> := uminus w, t <sub>3</sub> := t <sub>2</sub> + v}
9	E.place = t <sub>3</sub> E.code = {t <sub>2</sub> := uminus w, t <sub>3</sub> := t <sub>2</sub> + v}
10	E.place = t <sub>4</sub> E.code = {t <sub>1</sub> := y + z, t <sub>2</sub> := uminus w, t <sub>3</sub> := t <sub>2</sub> + v, t <sub>4</sub> := t <sub>1</sub> * t <sub>3</sub> }
11	S.code = {t <sub>1</sub> := y + z, t <sub>2</sub> := uminus w, t <sub>3</sub> := t <sub>2</sub> + v, t <sub>4</sub> := t <sub>1</sub> * t <sub>3</sub> , x := t <sub>4</sub> }

The watermark 'FREE ONLINE EDUCATION swayam' is visible at the bottom of the slide.

So, here is an example like a using that previous grammar. So, we have got an example string that is x produces. So, this x assigned as y plus z into minus of w plus v. So, this is

if you look into the grammar. So, the route is s producing id assigned as E. So, this is the parse tree. So, s id assigned as E, so, id this id is S. So, that will be known by the lexic analyzer. So, lexic analyzer will identify that id dot place has x. And, then this E, so, this will be so, this star will be generated so, E star E.

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

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

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

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

So, when I start with a bottom or left sides. So, I can see that I can do this thing, but I cannot do this. So, I after doing this I have to stop and go towards right a bit and then see what is the next available reduction, so, that is the this one. And then once this two are done so, this reduction is now ready. After, once this is done this reduction is now ready.

So, but after that I find that I cannot proceed further, then I have to come back and see what is the next operand at the next reduction at the lowest level that is possible. So, I find that this is the next reduction.

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

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

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

So, it will first generate a new temporary variable, it will first generate a new temporary variable in E dot place and then it will generate the code E1 dot code E2 dot code and this particular line being generated separately. So, how is it happening you seen that this E dot place equal to t1. So, here I have got a new temporary variable t1. And then E dot code E1 dot code and E2 dot code so, they are null E1 dot code E2 dot code are null. So, nothing done and then this new temporary it says the E dot place that is t1 equal to E1 dot place plus E2 dot place so, y plus z. So, this is the code that is generated.

(Refer Slide Time: 28:41)

The screenshot shows a slide titled "Example" with the expression  $x := (y+z)^*(-w+v)$ . Below the title is a table of reductions:

Reduction No.	Action
1	$E.place = y$
2	$E.place = z$
3	$E.place = t_1$
4	$E.code = \{t_1 := y + z\}$
5	$E.place = w$
6	$E.place = t_2$
7	$E.place = v$
8	$E.place = t_3$
9	$E.place = t_4$
10	$E.place = t_5$
11	$S.code = \{t_1 := y + z, t_2 := uminus w, t_3 := t_2 + v, t_4 := t_1 * t_3, x := t_5\}$

To the right of the table is a parse tree diagram:

```

graph TD
    S((S)) --> id1[id]
    S --> E1[E]
    S --> E2[E]
    E1 --> id2[id]
    E1 --> E3[E]
    E1 --> E4[E]
    E3 --> id3[id]
    E3 --> E5[E]
    E3 --> E6[E]
    E5 --> id4[id]
    E5 --> E7[E]
    E5 --> E8[E]
    E7 --> id5[id]
    E7 --> E9[E]
    E7 --> E10[E]
    E9 --> id6[id]
    E9 --> E11[E]
    E9 --> E12[E]
    E11 --> id7[id]
    E11 --> E13[E]
    E11 --> E14[E]
    E13 --> id8[id]
    E13 --> E15[E]
    E13 --> E16[E]
    E15 --> id9[id]
    E15 --> E17[E]
    E15 --> E18[E]
    E17 --> id10[id]
  
```

A handwritten note  $t_1 = y + z$  is written next to the reduction table. A man is visible on the right side of the slide.

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

So, it is unary minus so, this so, if you look into the action for this unary minus you see that it is so, it is written like  $E$  producing minus of  $E_1$ . So, that is a newtemp will be generated and  $E_1$  dot code followed by this temp generation of this extra thing ok. So, that will be done.

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

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

**Lecture - 53**  
**Intermediate Code Generation (Contd.)**

(Refer Slide Time: 00:19)

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

So, in our last class we are discussing with this particular example like  $x = y + z$  into  $-w + v$  with these example and we have seen upto reduction number 6. Now, what about reduction number 7? So, reduction number 7 is again  $E$  producing  $id$  and then the action is this  $E$  dot place equal to  $id$  dot place that is  $v$  and  $E$  dot code is null.

Now, coming to this reduction number 8, so  $E + E$ . So,  $E + E$  it says that you should get a new temporary variable first. So, this  $E$  dot place equal to  $t_3$  that is a new temporary and then  $E$  dot code it will have  $E_1$  dot code,  $E_2$  dot code and then this generation of a new code that  $t_3$  equal to something.

So,  $t_1$  equal so,  $t_2$  equal to unary minus  $w$  that is  $E_1$  dot code. So, this one this is our  $E_1$ . So, that is  $E_1$  dot code, then for this thing for  $E_2$  dot code it was null. So, there is nothing, then it generates this particular line that new  $E$  dot place that is  $t_3$  equal to this  $E_1$  dot place that is  $t_2$  plus  $E_2$  dot place that is  $v$ . So, that way this particular line this code is generated. So, now, I have got a code that  $t_2$  equal to unary minus  $w$ . So, the

situation I have got is  $t_1$  equal to  $y$  plus  $z$ . So, that is there and  $t_2$  equal to unary minus  $w$  and then  $t_3$  equal to  $t_2$  plus  $v$ . So, that much has been done.

Now, we come to reduction number 9. So, reduction number 9 is nothing, but within bracket E. So, this so, nothing is generated so, this place we remembered right as  $t_3$  only ok, so, this place is  $t_3$ . So, you can write down the corresponding places. So, this is this place is  $y$ , this place is written as  $z$ , this place is  $w$ , this place is  $v$ , this is also this is now this E plus E. So, reduction number 3. So, it was generated a new temporary. So, this place is  $t_1$ , then this place is  $t_2$  so, this minus of E so, this place is  $t_2$ . So, this is within bracket so, that is equal to  $t_2$ .

Now so, this is this place is  $t_3$  and then when I am doing this so, this place is also  $t_1$  ok. Now, it will be doing  $t_2$  into  $t_3$ . So,  $t_1$  into  $t_3$  that has to be done. So, then a line number 10, it has got E dot place equal to  $t_4$  line number reduction number 10. So, reduction number 10 is this one. So, E plus E, so, that way it will get a new temporary variable. So, that is what  $t_4$  and then code that is generated is E1 code E2 code followed by this code for this multiplication.

So, E1 code is  $t_1$  plus E1 plus  $z$  E2 code is  $t$  equal to unary minus  $w$  and then  $t_3$  this extra code that is generated. So, this line is also there. So, this is the code of E1. So, the upto this much is the code of E2 upto this much is the code of E2. And then this is the extra thing that is added for doing the multiplication that  $t_4$  equal to  $t_1$  into  $t_3$ .

And finally, this S dot code it says that you have to have this first this E1 dot. So, this E dot code. So, E dot code is this whole thing upto this much is E dot code as it is pointed out here. So, it will be E dot code and then it will have this id dot place assigned as E dot place. So, id dot place is  $x$  and then this E dot place is  $t_4$ . So, the so, this line number reduction number 10 has got E dot place in  $t_4$ . So, this  $x$  assigned as  $t_4$ .

So, ultimately we have got these as the final piece of code. So, S dot code points out. So, in this style you can generate code for this arithmetic expressions, but the only problem is that we are keeping the code as a pointer. So, that for every non terminal that, I have in my reduction tree. So, there is a pointer to a corresponding code.

And so, their may be the same code that repeated at several place like, see this particular line say this  $t_2$  equal to unary minus  $w$ . So, we have writing it here as well as here. So,

all these places this particular line is going to be repeated. So, the type of code generation function that we have used is that gen. So, that gen function has got this issue that is it will generate the code and while generating codes it will try to concatenate and all these things.

So, a better strategy may be that we just write the code on to a file as an when it is generated ok. So, as an when a line is generated by this the in this syntax directed translation mechanism. So, that is immediately written on to the file. And then so, that has got some difficulty also because we need to rectify some part of the code later many time what happens is that the jump targets are not known.

So, we cannot fill up those issue those points previously, but if we have got this type of organization where you have got this gen based codes. So, the code entire code is available as pointer to these individual non terminals. So, in that case that correction may be easy, but in case of while you are writing to file that correction may be slightly difficult. So, will see how are you going to do all those things?

(Refer Slide Time: 06:37)

- Consider array element  $A[i]$
- Assume lowest and highest indices of  $A$  are  $low$  and  $high$ , width of each element  $w$  and start address of  $A$ ,  $base$
- Element  $A[i]$  starts at location  $(base + (i - low) * w) = ((base - low * w) + i * w)$
- First part of the expression can be precomputed into a constant and added to the offset  $i * w$

Now, so, this is next we look into the code generation for arrays. Like, array this is a special thing, because array is a special data structure that we have. And almost all the programming language is they will have an array type of structure in them. So, generating code for array is very important.

So, let us consider an array element  $A[i]$ . And we will try to see how this  $A[i]$  will be translated into this 3 address code. So, if we assume that the lowest and highest indices of  $A$  are low and high. So, in so, in general so, this array when you are declaring. So, say for example, in language c we are writing a 100 with the implicit assumption that the index will run from 0 to 99. So, this 0 is the low index and 99 is the high index.

But, it did not be. So, like if you look into different programming languages you will find different standards like, somebody may think that this array this 100 in that case low is equal to 1 and high is equal to 100. Somebody may say that I can I should be able to define some array and I should be able to tell its index range very specifically. For example, somebody may say it is minus 10 to plus 10.

So, the array indices are  $A[-10]$ ,  $A[-9]$  in this so, it goes up to  $A[0]$  and then  $A[1]$ , going upto  $A[10]$ . So, that can happen. So, the so, all these styles are available. So, to make it very generic what has been done is that we assume that there are low and high indices are available, that will tell us what is the lowest and what is the highest possible index for the array.

And there are width of each element is  $w$  and the start the array you starts at the base address  $A$ . Now, because when this array will be loaded into the main memory as part of the program then array may not be starting at address 0. So, depending on that so, there will be a base address, but that base address is known when the program is going into execution, because at that time I have got; I have got the array has already been loaded into memory, so, its start address is known. So, where will this element a I will start ok. So, this will be starting at.

(Refer Slide Time: 09:27)

**Code Generation for Arrays**

- Consider array element  $A[i]$
- Assume lowest and highest indices of  $A$  are low and high, width of each element  $w$  and start address of  $A$ ,  $base$
- Element  $A[i]$  starts at location  $(base + (i - low) * w) = ((base - low * w) + i * w)$
- First part of the expression can be precomputed into a constant and added to the offset  $i * w$

Handwritten notes on the slide:

- $\text{int } a[100] \Rightarrow 1 \text{ to } 100$
- $w = 4 \text{ bytes}$
- $base = 1000$
- $1000 + 0 * 4 = 1000$
- $i = 1$
- $1000 + (1 - 1) * 4$
- $1000 + 0 * 4 = 1008$
- $i = 2$

So, let us take an example suppose I am defining an integer a 100 as the array and we are assuming that the indices they are running from 1 to 100 ok. So, this is not a c style of declaration. So, it is a hypothetical language where E I into a 100 if we give. So, the indices will run from 1 to 100. And if we say that the size of every element in this array is say 4 bytes; that means, in the array organization the locations so, 1, 2, 3 and 4, so, they are deserved for  $A[i]$ .

Similarly, 5 from 5 onwards a 2 will start 5, 6, 7 and 8, 5, 6, 7 and 8 they will have the element a 2; they will have the element a 2. So, it will go like this. Now, how to get a corresponding address for i? So, the address for i is given by this base plus i minus low into w.

So, because this base starts at some point suppose it start at address 1000. So, if I give i equal to say; i equal to say 1, then what happens. So, 100, 1000 plus i minus low so, low is also equal to 1. So, this is 0 into size of this integer is 4. So, that is equal to 1000 and you see that indeed the index 1 starts at location 1000.

So if we try out some other value of i, say i equal to say 3, then it is 1000 plus 3 minus 1 into 4. So, that is 1000 plus 8 so, 1008. So, you see that from 1008 only so, 1, 2, 3, 4 is so, 1, 2, 3, 4 is the first element, then so, then this 5, 6, 7, 8. So, they are the second element so, but third element start at 1008.

So, 1008 if it starts. So, that is going to be in fact here it is assumed that this  $i$  minus  $low$ , so, this  $low$  starts at 0. So, that is why it is the array index starts with 0. So, that is why it is happening like this. So, if it is some other values. So, I have to do a plus here to come to the correct 1 ok. So, this  $i$  minus  $low$  so, that is 3 plus 1, 3 minus 1, 2 plus 3, so, that is equal to 3. So, 3 into 4 it is starting at location 12. So, this is a element this is the third element. So, this is the element a 0, then I have got a1 here then from 8, 9, 10, 11 I will have A[2].

(Refer Slide Time: 13:01)

**Code Generation for Arrays**

- Consider array element  $A[i]$
- Assume lowest and highest indices of  $A$  are  $low$  and  $high$ , width of each element  $w$  and start address of  $A$ ,  $base$
- Element  $A[i]$  starts at location  $(base + (i - low) * w) = \underbrace{(base - low * w)}_{\uparrow} + \underbrace{i * w}_{\rightarrow}$
- First part of the expression can be precomputed into a constant and added to the offset  $i * w$

So, a 0 holds the addresses 0 to 0, 0, 1, 2, 3, then a1 it will occupy the addresses from 4 to 4, 5, 6, 7. And then a 2 will occupy the address 8, 9, 10, 11 and then a 3 will occupy the address from this 12 to 15.

So, in this particular case it is assumed that the array index starts with 0. So, if you take that the array index starts with 1, then you have to; you have to at 1, at another 1 here ok, in this expression then it will be correct. So, first part of the expression so, it can be rewritten like this  $base + i - low$  into  $w$ , so if you just recomputed this expression. Then, what happens is that this  $base - low$  into  $w$  so, that you can club in 1 group and this  $i$  into  $w$  in the other group.

So, this  $i$  minus so, first part. So,  $base - low$  into  $w$ . So, this part is constant, because  $base$  is known and  $low$  all those values are known. So,  $base - low$  into  $w$ . So, that becomes a constant part and that  $i$  into  $w$  is the variable part.

So, that is that will be called the offset part. So, this first part so, this is p computed and it is stored it is remembered with the array. So, whenever this array is defined. So, you see you can compute this base low w all these fields are known. So, you can immediately compute this constant part and store it as a component in the symbol table. So, that later on when this array is referred to. So, from the symbol table you can just retrieve that this part the value of this base minus low into w.

So, that way you get the base address and then this i into w so, that you can compute getting the value of i. So, that way it can be done. So, first part of the expression can be precomputed into a constant and added to the offset i into w.

(Refer Slide Time: 15:23)

**Code Generation for Arrays**

- For two-dimensional array with row-major storage,  $A[i_1, i_2]$  starts at location
 
$$\begin{aligned} & \text{base} + ((i_1 - \text{low}_1) * n_2 + i_2 - \text{low}_2) * w \\ &= \text{base} - ((\text{low}_1 * n_2) + \text{low}_2) * w + ((i_1 * n_2) + i_2) * w \end{aligned}$$
 where  $n_2$  is the size of the second dimension
- This can be extended to higher dimensions

So, if that is so, then how can we do this translation like? Say two dimensional array two dimensional array so, it has got array with row major storage. So, as you know that two dimensional array. So, there may be row major storage or column major storage. So, in a row major storage, so, we store the elements row by row. So, if say this is a 2 dash two dimensional array ok, where this individual rows and columns are running, these are the rows and these are the columns.

Now, in a row major organization means that ultimately memory is going to be single dimensional. So, it is not multi-dimensional. So, this first element so, we can store in these order. So, you can start store it like  $a(1,1)$  then  $a(1,2)$  then  $a(1,n)$  then  $a(2,1)$ ,  $a(2,2)$ , so, like that. That is what I am doing is I am storing the elements in this order. First this,

then this, then this, like this, so, I am doing it like this. So, you can do, that organization is called row major organization, I can also store it in a column wise fashion like first I store  $a(1,1)$ , then I store  $a(2,1)$ . So,  $a(m,1)$ , then I store  $a(1,2)$ , that way also I can do, so, if I do that. So, that will give us a column major organization.

So, here formula for calculating this  $a[i1][i2]$  in a row major organization is given by base plus  $i1$  minus low1 into  $n2$  plus  $i2$  into low2 into  $w$ . So, this is this you can also verify from any book on data structures. Now, after simplification so, you may like to take the constant parts together. So, that they can be precomputed and stored with the symbol table and this dynamic part may be computed separately. So, this base minus low1 into  $n2$  plus low2 whole thing multiplied by  $w$  so, this whole thing is the constant part ok. So, this is the constant part for the array and then this part is the variable part this part is the offset part.

So, while  $n2$  is the size of the second dimension. And so you can go for higher dimension also like, you can accordingly this formula will have one more stage. So, again you will get a constant part and a offset part, for this code generation for this array a array element access. Now, how to generate the actual code? So, that we will try to see in the successive slides.

(Refer Slide Time: 18:15)

**Array Translation Scheme**

**Grammar:**

$$\begin{aligned} S &\rightarrow L := E \\ E &\rightarrow E + E \mid (E) \mid L \\ L &\rightarrow Elist ] \mid id \\ Elist &\rightarrow Elist, E \mid id[E] \end{aligned}$$

**Attributes:**

- $L.place$ : holds name of the variable (may be array name also)
- $L.offset$ : null for simple variable, offset of the element for array
- $E.place$ : name of the variable holding value of expression  $E$
- $Elist.array$ : holds the name of the array referred to
- $Elist.place$ : name of the variable holding value for index expression
- $Elist.dim$ : holds current dimension under consideration for array

The slide is part of a presentation on Swayam, a free online education platform.

So, what is the grammar? So, that is the first thing. So, our grammar is like this. So, it is slightly twisted to make this code generation process simpler. So, we do it like this that  $S$

producing, L assigned as E. Where, E can produce E plus E within bracket E or L and this L can be Elist bracket close id, where Elist can be Elist comma E or id within bracket E.

So, you see that a very simple type of assignment array assignment may be like this say x equal to a say x equal to a sorry, I should follow this particular format say 1 say a[10] very simple one dimensional array access x equal to a[10].

(Refer Slide Time: 19:07)

**Array Translation Scheme**

**Grammar:**

$$S \rightarrow L := E$$

$$E \rightarrow E + E \mid (E) \mid id [Elist]$$

$$L \rightarrow Elist \mid id$$

$$Elist \rightarrow Elist, E \mid id[E]$$

**Attributes:**

- L.place: holds name of the variable (may be array name also)
- L.offset: null for simple variable, offset of the element for array
- E.place: name of the variable holding value of expression E
- Elist.array: holds the name of the array referred to
- Elist.place: name of the variable holding value for index expression
- Elist.dim: holds current dimension under consideration for array

The slide also features a parse tree diagram on the right side, showing the hierarchical structure of the expression  $a[10]$ .

Now, when I am doing this? So, what is done? So, it is broke up into statement like this S this an assignment statement. So, that this is the assignment L assigned as E. And then this one from the left hand side I do not have any array. And to come to that part I have to use this rule L producing id so, L producing id, which is x. Now, for the E part so, I have got an array ok. So, that to come to array I have to come to L actually, from L I can goto this Elist etcetera so, I have to come to L and from this L I have to generate this a 10 this access.

So, I will do it like this, I will make it like Elist and bracket close. And then Elist so, since they have the first part of this right hand side. So, that is for multidimensional array ok. So, there is a comma here, if this is multidimensional array it will be used. So, in our case in this particular case I have got a single dimensional array ok. So, I will follow this particular structure.

So, I will be doing like id open square bracket and E. This id is a now this E is an expression. So, expression can give me say number or id. So, this is a number and the number is 10. So, here I have not written explicitly the rule for that this one can be this E producing number can be a separately one or this I have not written it explicitly if I want to do it in a proper fashion as per this grammar then I have to do it like this.

This E producing L, L producing id and id is nothing, but this 10 do it like this ok. So, this grammar is slightly twisted otherwise I should have like this array name should be. So, this expression should be another rule should be id within bracket E some Elist should be something like this, but for the sake of code generation, so, it has been twisted a bit ok.

Now, this particular grammar for the different non terminals we assume the presence of different attributes. Now, the first attribute the L dot place ok. So, L dot place it holds the name of the variable may be an array name also ok. For this for the holding the name of the variable. So, this is L dot place will hold the name of the variable corresponding to L.

And L dot offset so, L dot offset identifies whether this is an array access or a variable access. So, if it is a simple variable then L dot offset is null. And if it is a array access then this L dot offset will hold the offset of the element for the array. So, previously we have seen that there can be this E dot w. So, we have seen previously that this type of offsets can be calculated ok. So, those offsets are calculated in this those offset will be calculated and that will come to this L dot offset, then E dot place. So, this will hold the name of the variable holding the value of expression E. So, that is E dot place.

Now, E list dot array is that is so, these are about the attributes of the L and E, for the Elist non terminal. So, we will have a number of attributes. So, Elist dot array it holds the name of the array that we are referring to, then Elist dot place it holds the name of the variable that holds the value for the index expression. So, if it a multidimensional array, then this I will get a complex expression, if it is single dimensional array then will this expression will be simple.

So, corresponding to the index whatever expression comes. So, that will be kept in Elist dot place and Elist dot dim. So, this holds the current dimension under consideration for

the array, because for each dimension the maximum size is different. So, like if I have got an array say a if I have got an array a say 100, 10.

(Refer Slide Time: 24:13)

**Grammar:**

$$\begin{aligned}S &\rightarrow L := E \\E &\rightarrow E + E \mid (E) \mid L \\L &\rightarrow Elist \] \mid id \\Elist &\rightarrow Elist, E \mid id[E\end{aligned}$$

**Attributes:**

- L.place: holds name of the variable (may be array name also)
- L.offset: null for simple variable, offset of the element for array
- E.place: name of the variable holding value of expression E
- Elist.array: holds the name of the array referred to
- Elist.place: name of the variable holding value for index expression
- Elist.dim: holds current dimension under consideration for array

Annotations on the right side of the slide:

- A handwritten note above the grammar section shows the expression  $a[100, 10]$  with arrows pointing to the first dimension '100' and the second dimension '10'.
- Below the attributes, there is another handwritten note showing the expression  $a[x+y, z+w]$  with arrows pointing to the first dimension 'x+y' and the second dimension 'z+w'.

So, for the first dimension it is the dimension is 100. So, for the second1 it is 10. So, that way that way this Elist dot dimension. So, it will hold the current dimension under consideration. So, when we are doing something like it may be I am so, I am doing it like this. So, this is a x plus y comma z into w. So, this may be the array element that I am trying to access. So, this x plus y so, this is for the first dimensions.

So, this is so, this x plus y 1. So, it corresponds to the first dimension and this z into w. So, this corresponds to the second dimension. So, we will see this array translation scheme. So, it will be using this attributes very cleverly to generate the corresponding code.

(Refer Slide Time: 25:05)

$S \rightarrow L := E$

{ if  $L.\text{offset} = \text{null}$  then  
  emit( $L.\text{place} := E.\text{place}$ );  
else  
  emit( $L.\text{place} '[' L.\text{offset '}] := E.\text{place}$ )  
}

$E \rightarrow E_1 + E_2$

{  $E.\text{place} := \text{newtemp}()$ ;  
  emit( $E.\text{place} := E_1.\text{place} + E_2.\text{place}$ )  
}

$E \rightarrow (E_1)$

{  $E.\text{place} := E_1.\text{place}$  }

Let us see, what are you going to do? So, initially I will look into this assignment statement, which is L assigned as E. So, this L assigned as E. So, if L dot offset is null; if L dot offset is null then I know that I do not have got so, there is no array reference here. So, I have to generate a code which was previously the simple assignment statement where it was S producing id assigned as E.

So, this was the situation and there we are said that the code that is generated is id dot place equal to E dot place. So, this was the code that is generated. So, here also it is similar to that. So, only thing is that if this L dot offset is null; that means, it is a simple identifier like this. So, in that case I can generate this code that L dot place assigned as E dot place. Otherwise, this is an array access, but by this time this L dot place holds the name of the array and this L dot offset. So, this holds the temporary into which the index expression has been calculated.

So, you can just; you can just look into the previous slide. So, this L dot offset is offset of the element of the array. So, this so, L dot place so, this is the holds the name of the variable, and L dot offset is that contains the offset of the element for the array. So, this will be done. So, this L dot offset has got the offset part calculated into a temporary. So, that temporary variable will be used as the index. And that will be assigned the value of E dot place.

So, E dot place already have the temporary that has got the value of E that will be done. So, this is so, this is the new rule that we have and that is taken care of in this fashion, then this rule is the old one so, E1 plus E2. So, as we know that here I have to get a new temporary variable for this E.

So, we get a new temp here and then we emit this particular code, that E dot place equal to E1 dot place plus E2 dot place, so, they are added. And so, E 1 dot place and E 2 dot place there are in some temporaries. So, they will be so the code says that those values are to be added and that code should be this E dot place should be assigned to that particular temporary. Similarly, E producing within the kit E 1 so, E dot place should be equal to E 1 dot place. So, there is no change at this point. So, this way for this simple rules so, we can do this assignment for more complex ones, so, the situation will be a bit different like say this one.

(Refer Slide Time: 28:03)

```

Semantic Actions for Arrays

E → L
{
    if L.offset = null then
        E.place = L.place
    else
        E.place = newtemp()
        emit(E.place ':=' L.place '[' L.offset ']')
}

L → id
{
    L.place = id.place
    L.offset := null
}

L → Elist
{
    L.place = newtemp()
    L.offset = newtemp()
    emit(L.place ':=' c(Elist.array) /* c returns constant part of the array */
    emit(L.offset ':=' Elist.place * width(Elist.array))
}

```

Say this E producing L. Now, again the same thing that is this expression that I am talking about so, this is an expression now this L it may be a simple variable or it may be an array expression.

So, what is done is that, if it is a simple variable then this L dot offset will be equal to null. In, that case E dot place equal to L dot place. Otherwise, this I so, are to get a new temporary variable for E dot place. So, that is obtained by this newtemp. And then it generates this piece of code that E dot place assigned as L dot place, then this square

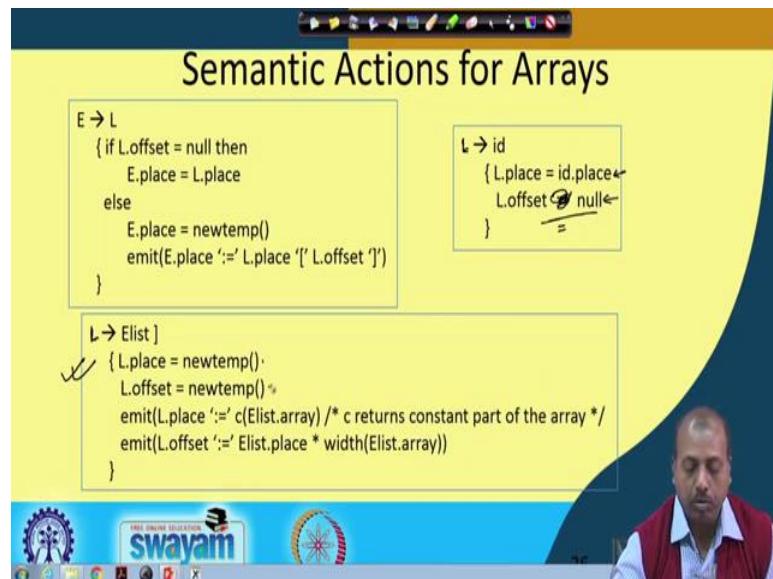
bracket L dot offset then this bracket close. So, this extra code will be generated ok. So, that way it will be generating the code for this E producing L.

So, this way, now, at the end of this so, this E will have this place attribute. So, E dot place which will be holding; which will be holding the value of this computed expression of L. And this L so and it will also have a piece of line of code in the code that is generated that E dot place is assigned as the array element access. So, we continue with this in the next class for this array access.

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

**Lecture - 54**  
**Intermediate Code Generation (Contd.)**

(Refer Slide Time: 00:18)



So, the next rule that we have for the array reference is this L producing id. So, for this L producing id the situation is very simple. So, this L dot place. So, this means that that L that I am talking about it is a single variable it is not an array.

So, this L dot place equal to id dot place and L dot offset is null. So, this is this is made null. So, that it is it will mean that I will have so, I will have I will be having simple infact this quote is not necessary. So, it should be like this that L dot offset equal to null. So, that means, it is a simple variable.

Now, let us look into this particular rule. So, this is array specific. So, this is L producing Elist bracket close. So, how are you going to handle this? L dot place is new temporary; L dot offset is new temporary. So, for this L I need to calculate the I need to generate the place and offset. So, they are generated in 2 new temporaries. And, then L dot place so, L dot place that holds the constant part of the array. So, if you look into the rule that we have seen previously.

(Refer Slide Time: 01:36)

**Grammar:**

$$\begin{aligned} S &\rightarrow L := E \\ E &\rightarrow E + E \quad | \quad (E) \quad | \quad L \\ L &\rightarrow Elist \quad | \quad id \\ Elist &\rightarrow Elist, E \quad | \quad id[E] \end{aligned}$$

**Attributes:**

- L.place: holds name of the variable (may be array name also)
- L.offset: null for simple variable, offset of the element for array
- E.place: name of the variable holding value of expression E
- Elist.array: holds the name of the array referred to
- Elist.place: name of the variable holding value for index expression
- Elist.dim: holds current dimension under consideration for array

24

So, this L dot place it holds the sorry Elist dot place is the variable holding the value for the index expression, at any point this whenever it is done so, L producing Elist. So, L should so, L dot place should hold the name of the variable, which may be an array and L dot offset will be the offset within the within the array.

(Refer Slide Time: 02:06)

**E → L**

```
{ if L.offset = null then  
    E.place = L.place  
else  
    E.place = newtemp()  
emit(E.place ':=' L.place '[' L.offset ']')
```

**L → id**

```
{ L.place = id.place  
L.offset = null }
```

**L → Elist [ ]**

```
{ L.place = newtemp()  
L.offset = newtemp()  
emit(L.place ':=' c(Elist.array) /* c returns constant part of the array */)  
emit(L.offset ':=' Elist.place * width(Elist.array)) }
```

26

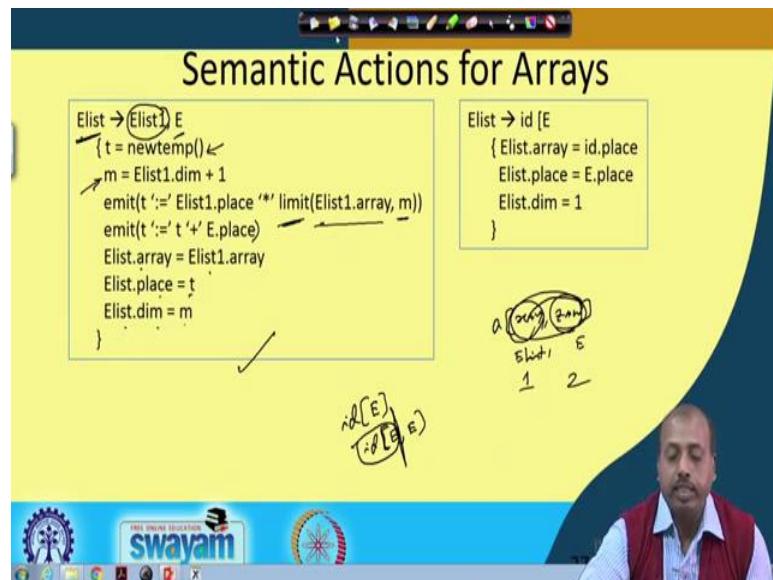
So, you see that this L dot place is newtemp L dot offset is newtemp. Now, after that so, this I have to generate this piece of code that is this newtemp equal to the constant part of the array. So, that is so, that will have this base plus etcetera etcetera so, that constant

part of the array that is available in the symbol table. So, Elist dot array holds the Elist dot array holds the name of the array.

And, then this so, for that the constant part will be retrieved from the symbol table and that will be assigned to L dot place. And, L dot offset will be equal to this Elist dot place multiplied by width of Elist dot array. So, this whatever be the Elist dot place. So, it has got the offset part. So, that multiplied by width of this individual array elements for this particular dimension.

So, they will be multiplied and that will be given to L dot offset. So, this way it will over now.

(Refer Slide Time: 03:12)



So, now if I multidimensional array. So, I will have to do it like this that Elist producing Elist1, comma E. So; however, so, then this t equal to newtemp. So, this gives me a new temporary variable. And, m so, this holds that this is the new dimension that I that I have got. So, Elist1 dot dimension. So, that was the dimension that we are considering previously so, which was this part.

So, for example, if you have got an array access like say x plus y comma z into w as I was talking about, now at this point this x plus y. So, this is available in Elist1 part and this is in E ok. Now, from that I have to get this Elist. So, Elist is for this entire thing. So, how to get it?

So, t equal to newtemp so, m. So, dimension that I am talking about is previously whatever was the dimension of this Elist1 dot dim so, this is Elist1 dimension was 1. Now, this is the second dimension has been added. So, this becomes equal to 2.

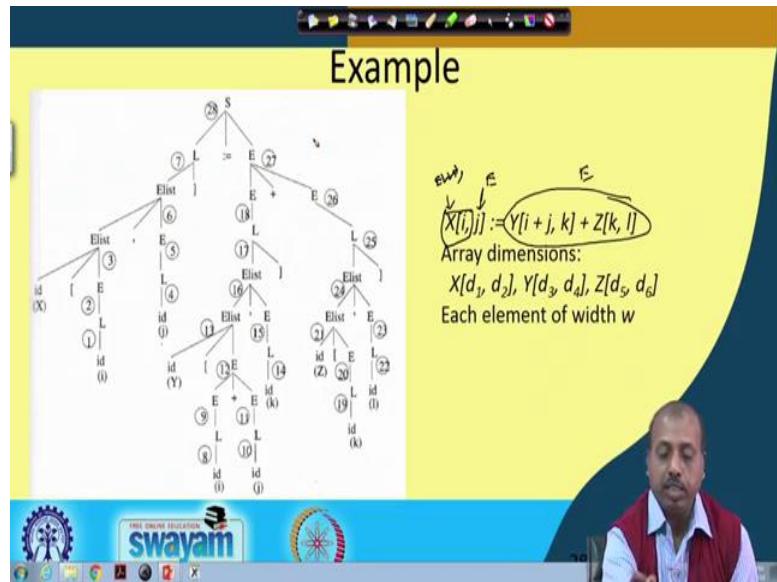
And, then we emit this particular code the t assigned as Elist1. So, t is a temporary here. So, Elist1 dot place multiplied by limit of Elist1 dot array for dimension m. So, we assume that there is a function limit that will search the symbol table and tell us what is the size of the array, what is the dimension of the array in the mth order. So, if it is a two dimensional array. So, initially m is equal to 1 after that m equal to 2. So, this limit will tell in each dimension what is the maximum value or maximum index that the array can take. So, that will give us this thing.

Then, we emit this particular code. So, this t assigned as t plus E dot place. So, that is the offset that has been calculated into this E. So, that will be added and then Elist dot array so, for this Elist. So, this is the array of the array name is same as the Elist1 dot array. So, that is taken and this Elist dot place equal to t and Elist dot dimension equal to m ok.

So, this is done this way, now the next rule that I have to consider is Elist producing i d within bracket E. So, this is basically I have got an array i d within bracket E and either it is bracket close or other situation is id within bracket E comma the next dimension say is another E. So, like that. So, here it is so what we do we break at this point, we break at this point and we call it Elist.

So, this Elist producing id open parenthesis the open square bracket E. So, like that. So, in that case Elist dot array is equal to id dot place, Elist dot place equal to E dot place and Elist dot dimension equal to 1. So, that is the first dimension. So, Elist dot dimension equal to 1. So, this way I can have this semantic action for the arrays. So, bit combustion, but once you get habituated to this then you will be able to understand that it is very simple it can be the translation can be done very efficiently.

(Refer Slide Time: 06:39)



So, here is an example. So, suppose I have got this in this example. So, there are 3 arrays X, Y and Z. And, their dimensions are X is of dimension  $d_1$  by  $d_2$ , Y is of  $d_3$  by  $d_4$  and Z is  $d_5$  by  $d_6$  and each element is of width w.

So, width of each element is taken to be same and that has to be because I am doing some arithmetic operation assignment etcetera on this that has to be done. Now, suppose that statement I have to for which I need to generate the code is this one, that  $X_{ij}$  equal to  $Y_{ijk}$  plus  $Z_{kl}$  ok. So, on the left hand side as well as on the right hand side we have got arrays ok. And, the arrays are of two dimension each of them is a two dimensional array.

So, the first assigned so, for the first state that I have is S producing L assigned as E, where the L should generate the left hand side, that is this one and E should generate this right hand side this whole thing. So, let us see how L generates this left hand side. So, we have to start with this production E list bracket close. So, in fact, the parser so, if you if you give to it the parser so, it will generate the parse tree like that.

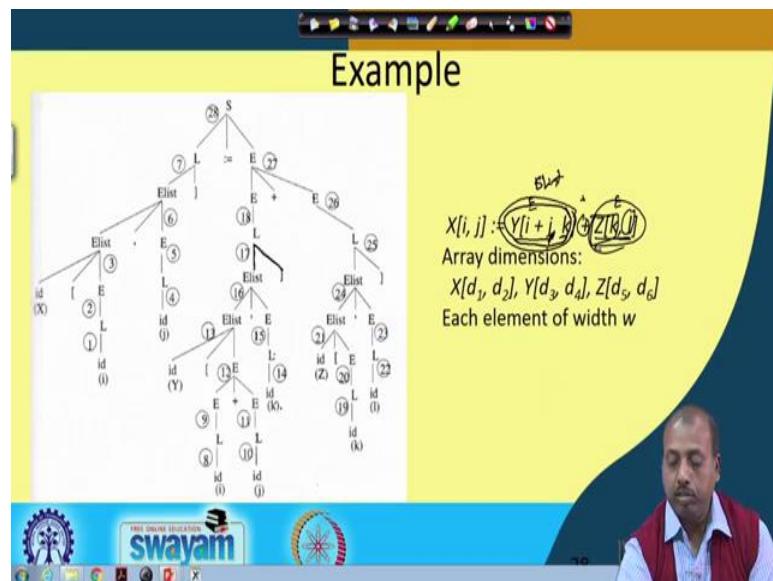
So, for the sake of our practice. So, we have to see that we will do like this, that L produces Elist bracket close. So, that is taken and then this from this so, this Elist is the last part. So, since it is Elist bracket close so, this is basically this part j bracket close part this is the j bracket close part.

So, I have to generate this part the first part and for doing the first part so, I have to have this Elist comma E. So, that from E I can generate this i sorry so, from this E I can generate this j part from E I can generate this j part, and from this Elist comma E. So,

from Elist comma part so, I can generate this part. So, that is that will be the Elist. So, that should be Elist comma E. So, like that.

Now, from this Elist I have to generate this X bracket start i. So, comma is done. So, I have to generate from this Elist X bracket start i. So, that is id bracket start E ok. Now, from E i come to L and from L i come to id. So, that is i fine. Now, for this j part so, that is so, I was stuck at this point Elist comma E and then from E I have to generate L from L to id that is j. So, that completes the parse tree for the left hand side, for the right hand side. So, this is an this is an E this whole thing is E and this is the sum of 2 expressions. So, the first rule that I should have is E plus E. So, that is done here.

(Refer Slide Time: 10:05)



So, this is first E, this is plus this is second E. Now, how this first E is going to be generated? So, again this is an array reference. So, I have to go by this E producing L and L as I said that whenever there is an array. So, the first rule that I have is by this one Elist bracket close. So, it is Elist bracket close.

So, this whole thing is Elist, this whole thing will be Elist and then the bracket close. And, after that from this Elist I have to generate this part comma and k. So, that is done here Elist comma E ok. Now, from this Elist I have to generate y bracket start i plus j. So, that is id bracket start E ok. And, this id will give me Y, now this E it is giving me E plus E.

So, this as this  $i + j$  I have to get  $i + j$ . So, it gives  $E + E$  then this  $E$  gives me  $L$  this  $L$  gives me  $id$ , which is  $I$  and this  $E$  gives me  $L$  which is  $id$  which is  $j$ . So, this part is done.

So,  $i + j$  comma so,  $y i + j$  comma up to that much we have seen how the parse tree has been made for the  $k$  part. So, this  $E$  producing  $L$ ,  $L$  producing  $id$  and that gives the  $k$   $k$  part. So, that finishes this one and plus is already done. Now, I have to do for this  $Z k L$ . So,  $Z k L$  again the same thing from this  $E$  if we want to come to an array the first rule is this  $E$  producing  $L$ , and then  $L$  producing  $Elist$  bracket close.

So,  $Elist$  this whole thing is  $Elist$ , then the then the bracket close is separate. Now, this  $Elist$  have to break it into  $Elist$  comma  $E$ , where  $E$  will give me this  $L$  part and  $Elist$  will give me this part and comma will be there. So, it is  $Elist$  comma  $E$ . Now, this  $Elist$  will give me  $i d$ . So, this should give me this  $id$ , that is this  $Z$  this is the  $id$  part, then this is bracket start and  $E$  and  $E$  giving me  $L$ ,  $L$  giving me  $id$  which  $id$  is  $k$  and this  $id$  is  $Z$ .

So, this way I can generate this part and then this  $L$  part. So, this can be generated by following this  $E$  producing  $L$  producing  $i d$  and this  $i d$  gives me  $L$ . So, that aims the understanding of the construction of the parse tree. So, one thing is that you see you see this is automatically done by the parser. So, you do not have to do it by hand, but just for our code generation practice and understanding the process of code generation.

So, we are doing it manually by hand ok. Now, how the code is actually generated. So, after that what I have done is that I have marked the reductions in the reverse order. The first reduction that will be done is this one. So, this  $L$  producing  $i d$ . So, that is the first reduction that the parser can do. So, this is number does 1 then this 1 number 2. So, once 1 and 2 are over. So, I can do this reduction. So, that is 3.

After, that I can do this reduction this is 4, then this one 5, this one 6, this one 7 I cannot proceed further. So, I have to come down to see where is 8. So, this is my 8, once this is done this is 9 so, this is 10, this is 11. Once 9 and 11 are done so, I can do this 12, once 12 is done I can do this 13, once 13 is done I cannot proceed further. So, I have to come down and see at this level that I can do this reduction,  $L$  producing  $id$  14, then after that. So, do this reduction so, 15 and after 15 so, I can do this reduction. So, this is so, this is this reduction is number 16, this reduction is number 17, this is 18.

So, up to that is done now again I have to fall back and see what is the next possible reduction. So, this is this one L producing id. So, number 19, then E producing L 20, then this reduction 21, then L producing id 22, E producing L 23, then this is 24, this is 25, 26, 27, 28. So, this way we number them so, that we can understand like how this code will be generated for this individual statements ok.

(Refer Slide Time: 14:53)

The screenshot shows a table titled "Example" with two columns: "Step No." and "Attribute assignment". The table lists 28 steps, each with an attribute assignment and the corresponding generated code. To the right of the table, handwritten annotations show the derivation of various temporaries (t<sub>1</sub> through t<sub>18</sub>) from the generated code. The annotations include assignments like t<sub>1</sub> = i \* d<sub>2</sub>, t<sub>2</sub> = t<sub>1</sub> + j, and t<sub>18</sub> = t<sub>17</sub> \* w. In the background, a man wearing a red vest is visible, and the bottom of the screen features the "swayam" logo.

Step No.	Attribute assignment	Code generated
1	L.place = i, L.offset = null	
2	E.place = i	
3	Elist.array = X, Elist.place = i, Elist.dim = 1	
4	L.place = j, L.offset = null	
5	E.place = j	
6	Elist.array = X, Elist.place = t <sub>1</sub> , Elist.dim = 2	t <sub>1</sub> = i * d <sub>2</sub> , t <sub>1</sub> := t <sub>1</sub> + j
7	L.place = t <sub>1</sub> , L.offset = t <sub>2</sub>	t <sub>2</sub> := t <sub>1</sub> * w
8	L.place = i, L.offset = null	
9	E.place = i	
10	L.place = j, L.offset = null	
11	E.place = j	
12	E.place = t <sub>4</sub>	t <sub>4</sub> := i + j
13	Elist.array = Y, Elist.place = t <sub>4</sub> , Elist.dim = 1	
14	L.place = k, L.offset = null	
15	E.place = k	
16	Elist.array = Y, Elist.place = t <sub>5</sub> , Elist.dim = 2	t <sub>5</sub> := t <sub>4</sub> * d <sub>3</sub> , t <sub>5</sub> := t <sub>5</sub> + k
17	L.place = t <sub>5</sub> , L.offset = t <sub>7</sub>	t <sub>6</sub> := C(Y), t <sub>7</sub> := t <sub>5</sub> * w
18	E.place = t <sub>6</sub>	t <sub>8</sub> := t <sub>6</sub> [t <sub>7</sub> ]
19	L.place = k, L.offset = null	
20	E.place = l	
21	Elist.array = Z, Elist.place = t <sub>9</sub> , Elist.dim = 1	t <sub>9</sub> := k * d <sub>4</sub> , t <sub>9</sub> := t <sub>9</sub> + l
22	L.place = t <sub>9</sub> , L.offset = null	t <sub>10</sub> := C(Z), t <sub>11</sub> := t <sub>9</sub> * w
23	E.place = l	
24	Elist.array = Z, Elist.place = t <sub>10</sub> , Elist.dim = 2	t <sub>12</sub> := t <sub>9</sub> * d <sub>5</sub> , t <sub>12</sub> := t <sub>12</sub> + l
25	L.place = t <sub>10</sub> , L.offset = t <sub>11</sub>	t <sub>13</sub> := C(Z), t <sub>11</sub> := t <sub>10</sub> * w
26	E.place = t <sub>12</sub>	t <sub>14</sub> := t <sub>13</sub> [t <sub>11</sub> ]
27	E.place = t <sub>13</sub>	t <sub>15</sub> := t <sub>12</sub> + t <sub>14</sub>
28		t <sub>16</sub> [t <sub>15</sub> ] := t <sub>13</sub>

So, this is the action that goes on as we proceed through this parse tree and try to generate the code. So, at step number 1 so, at step number 1 is this one. So, this L producing id and the rule for that is L producing id. So, L dot place is id dot place L dot offset is null.

So, that is done here. So, L dot offset is id dot place that is i and L dot offset is null. So, no code is generated. Now, step number 2 E producing L. So, E producing L tells that E dot place will be L dot place.

(Refer Slide Time: 15:34)

The slide is titled "Semantic Actions for Arrays". It contains three code snippets:

- E → L**:  
{ if L.offset = null then  
    E.place = L.place  
else  
    E.place = newtemp()  
    emit(E.place ':=' L.place '[' L.offset ']')  
}
- L → id**:  
{ L.place = id.place  
L.offset ':=' null  
}
- L → Elist**:  
{ L.place = newtemp()  
L.offset = newtemp()  
emit(L.place ':=' c(Elist.array) /\* c returns constant part of the array \*/)  
emit(L.offset ':=' Elist.place \* width(Elist.array))  
}

The slide footer includes the Swayam logo and the number 26.

So, L dot offset is null so, in that so, I have got I will have E dot place equal to L dot place. So, this E dot place equal to i. Now, step number 3. So, step number 3 is this one. So, this is Elist producing id within bracket E. So, let us see what is the corresponding action. So, this is this is the rule. So, it says that Elist dot array will be id dot plus Elist dot place will be E dot place and Elist dot dimension will be 1. So, let us see whether that has happened here or not. So, Elist dot array equal to X ok, that id Elist dot place equal to E dot place that is i and Elist dot dimension equal to 1.

So, that has been done here. Now, comes to step number 4 this L producing id. So, L producing id again the same thing that L dot place equal to id dot place. So, L dot place equal to j L dot offset equal to null. Now, at step 5. So, E dot place equal to L dot place. So, E dot place equal to j. Now, step number 6 is again so, this one Elist comma E. So, that reduction is done.

(Refer Slide Time: 16:53)

```
Elist → Elist1, E
{ t = newtemp()
m = Elist1.dim + 1
emit(t ':=' Elist1.place '*' limit(Elist1.array, m))
emit(t ':=' t '4' E.place
Elist.array = Elist1.array
Elist.place = t
Elist.dim = m
}

Elist → id [E
{ Elist.array = id.place
Elist.place = E.place
Elist.dim = 1
}
```

27

FREE ONLINE EDUCATION  
swayam

So, for that Elist comma E, so, this is the rule. So, I have to get a new temporary variable number or dimensions are to be incremented, then it will generate this type of code. So, what is done here you see at step number 6 is Elist dot array is the name of the array coming from Elist 1 dot array. Now, Elist dot place equal to t1. So, new temporary and this dimension increases to one more than is 2 and then t1 equal to i into d2 so, that is the size of the second dimension.

So, i into d2, then t1 equal to t1 plus j so, that is t1 plus this offset part E dot place. So, that will be added. So, this code will be generated and, then at reduction number 7 at reduction number 7. So, L producing Elist bracket close this open close parenthesis. So, Elist bracket close. So, this one, this L dot place newtemp L dot offset newtemp, then it will emit this piece of code.

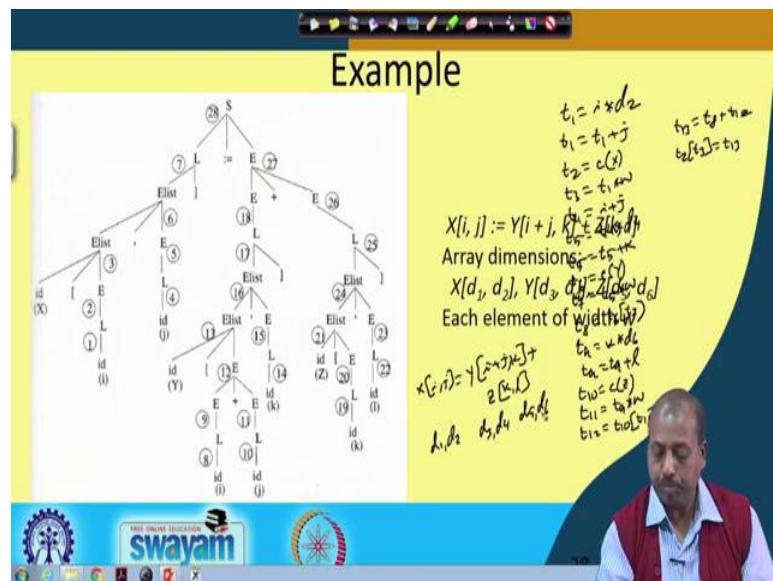
So, this L dot offset L dot place is t2 L dot offset equal to t3. Now, it generates the code that t2 equal to C of X and t3 equal to t1 into w. So, t1 it has calculated the offset. So, that multiplied by the size in the dimension that is tw. So, it is t1 into w, then, at step number 8 reduction number 8. So, it will be at reduction number 8. So, it is L producing i d. So, it will be doing it this way that L1 dot place equal to i and L dot offset equal to null no code is generated.

So, next code is generated somewhere at this point at this reduction, reduction number 12 ok. Now, so, you can trace through this code generation process. So, will do some bigger

exercise later once we come to the end of this chapter, but ultimately we have got this piece of code generated. So, we have got this  $t_1$  equal to  $i$  into  $d_2$ ,  $t_1$  equal to  $t_1$  plus  $j$ , then  $t_2$  equal to constant part of  $x$   $t_3$  equal to  $t_1$  into  $w$ , then  $t_4$  equal to  $i$  plus  $j$ ,  $t_5$  equal to  $t_4$  into  $d_4$   $t_5$  equal to  $t_5$  plus  $k$ , then  $t_6$  equal to constant part of  $Y$ , then  $t_7$  equal to  $t_5$  into  $w$ .

Then,  $t_8$  equal to  $t_6$   $t_7$ , then  $t_9$  equal to  $k$  into  $d_6$ ,  $t_9$  equal to  $t_9$  plus  $L$ , then  $t_{10}$  equal to constant part of  $Z$ ,  $t_{11}$  equal to  $t_9$  into  $w$ ,  $t_{12}$  equal to  $t_{10}$   $t_{11}$ ,  $t_{13}$  equal to  $t_8$  plus  $t_{12}$ , and  $t_2$   $t_3$  equal to  $t_{13}$ . So, let us try to correlate this code with the example that I had that we had taken. So, it is  $X_{ij} + Y_{ij}$ .

(Refer Slide Time: 21:24)



So, we have got this  $x$  the example that we have taken is  $X_{ij}$  equal to  $Y_{ij} + Z_{ik}$ . And, the dimensions are  $x$  is of dimension  $d_1 d_2$ ,  $y$  is of dimension  $d_3 d_4$ , and  $z$  is of dimension  $d_5 d_6$  ok. So, let us try to see whether this particular code is being generated  $X_{ij} + Y_{ij} + Z_{ik}$  that is generated or not.

So, this is  $i$  into  $d_2$ . So,  $i$  into  $d_2$  is for the first dimension. So, it is doing like say  $i$  into  $d_2$  and then it is doing  $t_1$  equal to  $t_1$  plus  $j$ . So,  $t_1$  equal to so,  $i$   $t_1$  was  $I$  into  $d_2$  plus  $j$ . So, that is giving me the offset of this  $ij$ . And,  $t_3$   $t_2$  now equal to constant part of  $X$  and  $t_3$  equal to  $t_1$  into  $w$ . So, that gives me the actual offset. So, this  $t_2$  contains the base part and  $t_3$  contains the constant part of the array reference and  $t_3$  contains the base the offset part. And, at the end you will notice that I am doing like  $t_2$

$t_3$  assigned as  $t_{13}$ . So,  $t_2$   $t_3$  so, it is  $X_{ij}$  ultimately ok. So, if you look into this so, this  $X_{ij}$  equal to something. So, this  $X_{ij}$  is turning out to be this  $t_2$   $t_3$ .

Now, for the next part so,  $Y_i + j$  so,  $y_i + j$  for that so, this in  $t_4$  it is computing  $i + j$ . Then, that is multiplied by the dimension of the second array so, that is  $d_4$ , in the second in the second index. So, it is  $d_4$ , then  $t_5$  equal to  $t_5 + k$ . So, this is the second index that I have so, the it was  $X Y_i + j k$ . So,  $i + j$  you have computed the offset. So, with that the offset for  $k$  has to be added. So, that is done here. And,  $t_6$  is the constant part of  $y$   $t_7$  is the variable part of this offset part of  $y$  axis. So, this is  $t_7$  equal to  $t_5$  into  $w$ . So, ultimately I have got  $t_8$  equal to  $t_6 t_7$ . So, this  $t = 8$  contains the this part so,  $y_i + j k$  part.

Now, this  $Z_{kl}$  part so, for that  $t_9$  equal to  $k$  into  $d_6$  so,  $d_6$  is the dimension of the third array in the in the second index,  $t_9$  is  $t_9 + L$  and then  $t_{10}$  is the constant part of  $z$   $t_{11}$  is  $t_9$  into  $w$ , then  $t_{12}$  is  $t_{10} t_{11}$ . So, that way I have got this in the variable  $t_2$ . So, I have got the array array element copied, then  $t_{13}$  is  $t_8 + t_{12}$ . So,  $t_8$  was having this  $t_6 t_7$  and  $t_{13}$  is having  $t_8 + t_{12}$  is having  $t_{10} t_{11}$ . So, they are added. So, we get  $t_{13}$  equal to  $t_8 + t_{12}$ . Ultimately,  $t_2$   $t_3$  assigned as  $t_{13}$ . So, this is holding the value of  $t_{13}$ .

So, so,  $t_{13}$  is having the  $t_8 + t_{12}$ . So, this right hand side of the expression. So, this whole thing is available in  $t_{12}$ . So, that is added with  $t_8$  and  $t_8$  was holding the first array access  $t_{67}$ . So, they are added and then that is assigned to the array on the left hand side  $t_{12} t_{13}$  ok.

So, this way we can generate the code for the array elements. So, it is a bit combustion, but so, if with a little bit of practice. So, you will be able to do it by hand so; will also try to do a few some exercises in the class towards the end of the chapter.

(Refer Slide Time: 25:35)

**Attributes of Boolean expression  $B$ :**

1.  $B.\text{true}$ : defines place, control should reach if  $B$  is true
2.  $B.\text{false}$ : defines place, control should reach if  $B$  is false

**Grammar:**

$$B \rightarrow B \text{ or } B$$
$$\quad | \quad B \text{ and } B$$
$$\quad | \quad \text{not } B$$
$$\quad | \quad (B)$$
$$\quad | \quad \text{id} \text{ relop id}$$
$$\quad | \quad \text{true}$$
$$\quad | \quad \text{false}$$

**Assumed true and false transfer points for entire expression**

- If  $B$  is known
- If  $B_1$  is true,  $B$  is true  $\rightarrow$  need not evaluate  $B_2 \rightarrow$  called short-circuit evaluation
- If  $B_1$  is false,  $B_2$  needs to be evaluated
- Thus,  $B_1.\text{false}$  assigned a new label marking beginning of evaluation of  $B_2$
- Function  $\text{newlabel}()$  generates new label

$B \rightarrow B_1 \text{ or } B_2$

(  $B_1.\text{true} = B.\text{true}$   
 $B_1.\text{false} = \text{newlabel}()$   
 $B_2.\text{true} = B.\text{true}$   
 $B_2.\text{false} = B.\text{false}$   
 $B.\text{code} = B_1.\text{code} ||$   
 $\quad \text{gen}(B_1.\text{false}, ':') ||$   
 $B_2.\text{code}$

Next, will be looking into translation of Boolean expressions so, Boolean expressions are something very important, because many a time we have got this Boolean variables and this Boolean controls are there like say if then else type of control, while loop type of control. So, like that so, we have got these Boolean expressions. So, they form part of this control statements in a programming language.

So, for doing this translation of the Boolean expression so, will be doing it like this. So, will assume that there is an attribute  $B$  dot true for any Boolean expression  $B$ , it will have an attribute  $B$  dot true. So, that defines the place and control that control should reach if  $B$  is true. So, in the program execution, if this is a portion of the program and at this point suppose I am evaluating, this is here I have got some code which is evaluating the Boolean expression  $B$ .

Now, as I said that these are these are normally part of if then else or while go to statements. So, if this  $b$  condition is true then where to go ok. So, may be if it is if then else statement. So, if some  $x$  is greater than  $y$ , then  $S_1$  else  $S_2$ . So; that means, if this is my  $B$  ok. So, if  $B$  happens to be true then I should jump to the portion of code where  $s_1$  is being executed.

So,  $S_1$  has been coded. On the other hand if  $B$  is false in that case I need to jump to another piece of code which  $S_2$ . So, that way with  $b$  the Boolean variable  $B$  so, we assume that we have got for the Boolean non terminal  $B$ . So, we assume that there are

two attributes B dot true and B dot false, while B dot true defines place the control should reach if B is true, and B dot false will define place while control should reach if B is false.

So, the grammar that will be considering is this and or grammar so, and or not these are 3 operators. So, B or B B and B not of B within bracket B, then we also have got this relation relational operators id relop id. So, you can have this relop. So, this is the relational operator. So, all the relational operators like greater than less than so, greater or less or equal greater or equal equality. So, all of them we club together and call them the relational operator relop.

Now, this relop operator so, this will so, this. So, we are only allowing this id relop id. So, you can only write like x greater than y like that. So, we can we cannot have expression like x plus y greater than z into w. So, that type of expression. So, we are not allowing in this particular grammar ok. So, that can be done, but for the sake of our understanding, because we do not want to bring this arithmetic operators into consideration. So, they are not taken here.

And, then the since it is a Boolean expression so, we have got 2 constant identifiers true and false. So, they are also part of the grammar. So, this is the grammar that we have and then we will see in the next class, how to have this code generation for the Boolean expressions.

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

## Lecture - 55

So, for example, if you look into the first rule of this of this grammar B or B.

(Refer Slide Time: 00:19)

# Translation of Boolean Expressions

**Attributes of Boolean expression B:**

1. *B.true*: defines place, control should reach if B is true
2. *B.false*: defines place, control should reach if B is false

**Grammar:**

$$B \rightarrow B \text{ or } B$$

- | *B and B*
- | *not B*
- | *(B)*
- | *id relop id*
- | *true*
- | *false*

- Assumed true and false transfer points for entire expression
- B is known
- If B1 is true, B is true → need not evaluate B2 → called short-circuit evaluation
- If B1 is false, B2 needs to be evaluated
- Thus, B1.false assigned a new label marking beginning of evaluation of B2
- Function newlabel() generates new label

$$B \rightarrow B_1 \text{ or } B_2$$

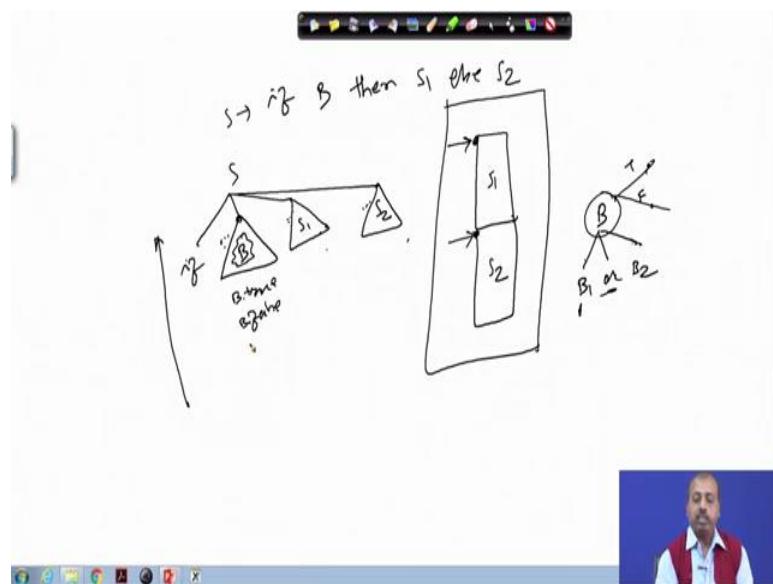
```

{ B1.true = B.true,
  B1.false = newlabel()
  B2.true = B.true
  B2.false = B.false
  B.code = B1.code || gen(B1.false, ~) || B2.code
}

```

So, for the sake of our understanding so, we are writing it as B, so B producing B1 or B2. So, like this now we will assume that the true and false transfer points for the entire expression B is known. So, it is part of so, B is part of some bigger expression like say. So, let us do it like this.

(Refer Slide Time: 00:49)



Say if some Boolean expression then I have got a block of statements S1 else S2 where S1 is a block of statements and S2 is another block of statements. Now if you look into the parse tree for this so, it will be something like this S producing. If then a portion of the parse tree which will correspond to B then there will be a portion of the parse tree which corresponds to S1 and there will be a portion of the parse tree where I will have this S2, fine.

Now, it is assumed that when I am doing this reduction. So, if B is true it will so, so since all these reductions have already been done. So, this S1 and S2 position in the three address code file. So, they are known is not by that time the codes have already been generated. So, I can very easily tell what is the offset of the code S1, what is the offset of the code S2 in the full three address code file, the file that contains the three address code. What are the offsets of these positions? So, naturally I can assume that for this B this B dot true and B dot false. So, these two pointers where they should point to. So, that is known when I am going to do this reduction.

Similarly, if I, so, just to extra polite that fact. So, if I have got a rule like this. So, B1 or B2 so, we know that when I am doing this. So, for this B so, we already know the true and false pointers for that.

So, where the true and false should go, but then for now if now what the situation is that if B1 happens to be true then since it is an or operation. So, if this is the true branch and

this is the false branch then B1 being true, it should follow that true branch of B and B1, so B2 B so, similarly B2 being true. So, it should follow the true branch of B and similarly if B1 is false and B2 is false, if both are false then only it should go to the false branch.

Now, it is a bit confusing because I said that when I am doing this reduction. So, in this particular case when I was doing this reduction, this targets were known where S1 and S2 are in this file that are known, but here my B expression is not yet over. So, I do not know where the where so, I am in the, I am actually parsing somewhere here bring the parsing at this point. So, at that time I have not reduced the S1, I have not reduced S2. So, as we are looking into the numbering policy of this reduction. So, you see that these numbers that are appearing for this. So, they are much less compared to this numbers and this numbers. So, they have not yet been reduced. So, they are that they will come later. So, naturally for a single parse type of operations. So, it is difficult, but if we assume that my compiler the code generation phase will be in a two pass fashion.

So, in the first pass, so you can identify all these jump targets, ok. For B dot true B dot false. So, that the second pass while generating the actual code so, you can just use those say, you can just use those addresses. Of course, you can do something else also that we will see for single pass compilation. So, we need to do some sort of back patching type of operation, so that we will see later. So, for the time being, it is assumed that we have got this whenever we are having this particular rule B producing B1 or B2. So, it is assumed that the true and false transfer points for the entire expression B are known. Now if B1 is true B is also true. So, we need not evaluate B2. So, B1 appears to be happens to be true then B2 did not be evaluated without evaluating B2. Also, we can say that the whole expression B is going to be true.

So, this is called a short circuit evaluation, because we are not evaluate B2, if B1 already true. Similarly, if B1 is false then off course we cannot tell anything, because now I need to evaluate B2 also. And if B2 is also false then it will be then we have to follow the false label.

So, the way the code is that the semantic actions the, the syntax directed actions are written is like this that beyond the true is made equal to B dot true. So, since, you assume that we already know B dot true. So, B1 dot true will be like that, but for B1 dot false, I

do not know where to go, ok. So, that way I generate a new label. So, this new label is say L1 so, B1 dot false equal to L1. Now B2 dot true equal to B dot true B2 dot false equal to b dot false, because I will first have the code for B1, then I will have the code for B2.

So, if B 1 is not true then it has run like this. So, it has run like this and then I will be so, this B2. So, at the end B2 dot false will be equal to B dot false and this B dot code, it will have the code for B1, it will have the code for B2. And then, but in between there will be a label that will get inserted.

So, here you see that the B 1 dot false. So, you have got the label. So, that will be inserted here. So, B 1 dot false colon. So, that will be inserted then I will have got the code for B 2 dot I will have the portion for B 2 dot code. So, what I was the situation final situation for B dot code will be something like this.

(Refer Slide Time: 06:57)

**Attributes of Boolean expression B:**

1. *B.true*: defines place, control should reach if B is true
2. *B.false*: defines place, control should reach if B is false

**Grammar:**

$$B \rightarrow B \text{ or } B$$

- |  $B \text{ and } B$
- |  $\text{not } B$
- |  $(B)$
- |  $\text{id relop id}$
- |  $\text{true}$
- |  $\text{false}$

\*Assumed true and false transfer points for entire expression  
B is known  
•If B1 is true, B is true → need not evaluate B2 → called short-circuit evaluation  
•If B1 is false, B2 needs to be evaluated  
•Thus, B1.false assigned a new label marking beginning of evaluation of B2  
•Function newlabel() generates new label

**Pseudocode:**

```

 $B \rightarrow B1 \text{ or } B2$ 
{ B1.true = B.true
  B1.false = newlabel()
  B2.true = B.true
  B2.false = B.false
  B.code = B1.code || gen(B1.false, ':') || B2.code
}
  
```

The slide also features the 'swayam' logo and other navigation icons at the bottom.

So, here I will have got the code for I will have the portion for B1 dot code, this is the B1 dot code and after that I will have a, this newlabel that is there. So, this newlabel L1 has been generated. So, this is so, B1 dot false contains the label.

So, this is L1 suppose the newlabel that is generated is L1 and after that it will be the B2 dot code will come. So, after B1 dot code, this L1 will be generated this L1 label will be generated then this B2 dot code will be generated.

So, it will be going like this. So, this after this B1 dot code is over see this label will get inserted and then this B2 dot code will be put at the end. So, so this way I can generate the code for this B1 or 2. So, we can we look into the other rules other grammar rules for generating the code for other Boolean Expression or other rules like B equal to B1 and B2.

(Refer Slide Time: 08:11)

**Translation of Boolean Expressions**

$B \rightarrow B1 \text{ and } B2$ $\{ B1.\text{true} = \text{newlabel}();$ $B1.\text{false} = B.\text{false};$ $B2.\text{true} = B.\text{true};$ $B2.\text{false} = B.\text{false};$ $B.\text{code} = B1.\text{code}   $ $\quad \text{gen}(B1.\text{true}, ':')   $ $\quad B2.\text{code}$ $\}$	$B \rightarrow \text{not } B1$ $\{ B1.\text{true} = B.\text{false};$ $B1.\text{false} = B.\text{true};$ $B.\text{code} = B1.\text{code};$ $\}$	$B \rightarrow (B1)$ $\{ B1.\text{true} = B.\text{true};$ $B1.\text{false} = B.\text{false};$ $B.\text{code} = B1.\text{code};$ $\}$
$B \rightarrow id1 \text{ relop } id2$ $\{ B.\text{code} = \text{gen}('if' id1.place \text{ relop } id2.place 'goto' B.\text{true})    \text{gen}('goto' B.\text{false})$		
$B \rightarrow \text{true}$ $\{ B.\text{code} = \text{gen}('goto' B.\text{true})$	$B \rightarrow \text{false}$ $\{ B.\text{code} = \text{gen}('goto' B.\text{false})$	

FREE ONLINE EDUCATION  
**swayam**

So, in that case if B1 is true then I have to evaluate B2 also, but if B1 is false then it is a short circuit. So, if B1 is false. So, B1 dot false is made equal to B dot false then B2 dot true is. So, B1 dot true is newlabel then this B1 dot false is B dot false so; that means, it is a short circuit. And then B2 dot true is B dot true B2 dot false is B dot false then B dot code is the concatenation of B1 dot code then this particular label B1 dot true colon and then B2 dot code.

So, this that way I can have this B1 and B2 then not of B1. So, B1 dot true is B dot false B1 dot false is B dot true and B dot code is equal to B1 dot code. So, no new code has to be generated, but this true and false pointers.

So, they have to be reversed. So, wherever the B1. So, B is B dot false wherever it was going. So, they, they should be we should go there if B1 is becoming true. Now, and wherever it was going on B becoming true. So, on B1 becoming false we should go there, ok. So, then B within bracket B1. So, this is also very simple. So, B1 dot true equal

to B dot true B1 dot false equal to B dot false B dot code equal to B1 dot code. So, like that now B producing true. So, this is B dot code, it will generate a goto B dot true.

So, then because we know it is definitely true. So, that way I do not have to evaluate anything. So, this is B dot, B dot code will be. So, this particular line will be generated goto B dot true. So, B dot true is already known the target is known. So, it will go there similarly, B dot false is like this. Now, id1 relop id2. So, this will generate the two statements. So, if id1 dot place relational operator id2 dot place goto B dot true otherwise it will goto B dot false. So, that way this id1 relop id2 will be done.

(Refer Slide Time: 10:41)

- Makes the scheme inherently two-pass procedure
- All jump targets are computed in the first pass
- Actual code generation done in second pass
- A single-pass approach can be developed by
  - Modifying the grammar a bit, and
  - Introducing a few more attributes
  - A few new procedures
  - Generated code can be visualized as an array of quadruples

So, how does it help us in generating the generating code? So, this helps because I can have the short circuit of Boolean operators and all. So, I do not evaluate the complete expression and so, that helps, ok.

In many many programming languages you see this is helpful like see suppose I have got this shape. So, I have got an array in my programming integer a 100 and at some point of time I need to check whether if  $a[i]$  is greater than 50 then something something. Now one problem with this type of situation is that. So, if this index  $i$  itself has become more than 100 the value of the expression  $i$  itself is more than 100 then whenever, whenever I am trying to do this  $a[i]$  access. So, this can generate some violation of condition, ok. So, the some run time error may occur. So, it is a safe practice that we write like this. If  $i$  less than 100 less or equal 100 and  $a[i]$  greater than 50 then I have got the piece of code.

Now, here you see that if I happen to be more than 100 then this part itself will evaluate to false. So, I will not go into this evaluation and it will. So, it will not try to access some arbitrary memory location to get the value of a i. So, that way my code execution is much safer, ok. So, this is too good like for this short circuit of this Boolean expression evaluation. So, this is really helpful; however, the way that we are generating the code it has got problems, because the first thing is that it makes the scheme inherently two pass proceeding, because in the first pass all the jump targets will be computed and in the second pass the actual code generation will be done. So, that is that is required, because I need to have this B dot true B dot false. So, those pointers their destinations are calculated before we generate the code.

However, so that makes it a bit cumbersome also. So, there is a modification to the grammar, so that we can make it a single pass approach. So, single pass approach, it can be developed by modifying the grammar a bit and introducing a few more attributes, ok. So, few new procedures and the generated code can be visualized as an array of quadruples. So, that way we can do, that is correcting some portions some corrections can be carried out in the code that is generated. Basically when the jump targets become known then only we can tell the address like.

(Refer Slide Time: 13:43)

- Makes the scheme inherently two-pass procedure
- All jump targets are computed in the first pass
- Actual code generation done in second pass
- A single-pass approach can be developed by
  - Modifying the grammar a bit, and
  - Introducing a few more attributes
  - A few new procedures
  - Generated code can be visualized as an array of quadruples

For example if I have got a statement like if some Boolean expression B then S1 else S2. Now while parsing this B so, I have got at several points the B becomes true or B

becomes false like that. So, at those points are need to transfer the control to either S1 or S2 but until unless the code for B is over.

So, if this is the file where I am writing the code for this three address code. So, until unless the code for B is over I cannot start the code for S1 and until unless the code for S1 is over I cannot start the code for S2. So, when I am this intermediary points, I really do not know this particular offsets the offset of S1 and offset of S2, because it is all dependent on size of B1 size of S1, etcetera. So, I need to do something so, that in the first pass I can take it as I can know that the offset values, ok. The second pass I do that; however, in the single pass procedures. So, you have to modify something so, that we can we can generate the code.

(Refer Slide Time: 15:03)

The slide is titled "Attributes". It contains two bullet points:

- B.trueList
  - List of locations within the generated code for B, at which B definitely true
  - Once defined, all these points should transfer control to B.true
- B.falseList
  - List of locations within the generated code for B, at which B definitely false
  - Once defined, all these points should transfer control to B.false

The footer of the slide includes the "SWAYAM" logo and other navigation icons.

So, how is it being done? So, the attributes that we are talking about new attributes apart from that B dot true and B dot false. So, we have got a truelist and a falselist for every Boolean expression. So, B dot truelist, it is the list of locations within the generated code for B at which B is definitely true. So, as I was telling that if this is the portion of this is the parse tree for B part. So, this is the parse tree for B. Suppose at this points in the parse tree the B expression is definitely true.

So, we put all these locations in a list called B dot truelist. Similarly there may be a few locations where B is definitely false. So, they are put on to the list B dot falselist. So, in B dot truelist, it is the list of locations within the generated code for B at which B is

definitely true. And once defined all these points should transfer control to B dot true. So, when I am generating the parse tree for B. So, at that time I do not know what is B dot true and B dot false targets but after some time they will become defined.

So, for example, that if then else statement that, then part and else part the statements they will come after some time of parsing. So, at that point so, this jump targets will known and for this for the nonterminal B. So, we have note down the true list where all this points of all these points at which the B is true. So, that that values have been kept.

So, at all those places, we can replace the address the jump address that can that was left as blank at that point. So, they can be filled up with this jump this new address which is this B dot true or the address of S1 in if then else statement similarly B dot falselist. So, the it contains the list of locations within the generated code for B at with B is definitely false and once defined all these points should transfer control to B dot false.

So, they should I should have the all controls transferred to B dot false. So, all these points I can do some correction in the target code that is generated. So, that they are rectified and now they contain the address of B dot false as their jump target.

(Refer Slide Time: 17:25)

Extra Functions

- *makelist(*i*):* creates a new list with a single entry *i* – an index into the array of quadruples
- *mergelist(*list1*, *list2*):* returns a new list containing *list1* followed by *list2*
- *backpatch(*list*, *target*):* inserts the *target* as the target label into each quadruple pointed to by entries in the *list*
- *nextquad():* returns the index of the next quadruple to be generated

34

Some more functions are also necessary it makelist i. So, it creates a new list with a single entry i. So, and so, this is an index into the array of quadruples. So, some slide

earlier we are said that generated code is visualized as an array of quadruples, ok. So, here also we say that this makelist i.

So, if this is an index into that array of quadruples then mergelist of list 1 and list 2 it returns a new list containing list one followed by list 2. So, the margin of two list then backpatch list target. So, this will insert the target as the target label into each quadruple pointed to by entries in the list.

So, this will be. So, this list is containing all the places where this jump targets are not yet filled up, ok. Now for them so, if we have if we come across the correct target with which they to which these all these control should jump to. So, they that is say that that is called target. So, all those locations they will be filled up with the value of target and there is a function called nextquad that will return the index of next quadruple to be generated. So, many a time will need to know what is the address of the next piece of code that will be generated so, the at the next quadruple of code that will be generated. So, that is by this nextquad function.

(Refer Slide Time: 18:53)

So, the grammar that will have now is a modified version B producing B or MB, B producing B and MB not of B within bracket B id relop id true false and epsilon M producing epsilon. So, this is a. So, in this particular case, you see what has been done we have introduced a new dummy variable M, ok. So, we have introduced a new dummy variable M.

So, which we call and the this is this is dummy, because ultimately I am replacing it by M producing epsilon. If it does not have any effect on the on the M, ok, it does not have any effect on the grammar, but this will be useful for knowing some addresses and modifying the proper values inserting proper jump targets and all.

So, this M it has got an attribute called M dot quad that can hold index of the of a quadruple. So, that so, you can say M dot quad equal to some quadruple value. So, that will tell us this that will hold the in that can be assigned some index of the of a quadruple.

Now, how is it going to be used? So, it can be cleared by looking into say this particular rule B producing B1 or MB2, ok. So, before this execution of B2 starts reduction M producing epsilon has already taken place. So, if you look into the parse tree. So, this will be like this. So, B producing this B1 then M, sorry B1 or then M and this B2 and this M produces epsilon. So, this is the parse tree part. Now as we have seen previously by that number the policy of the numbering of those reductions. So, first this reductions will be numbered then this reduction will be numbered and then only this reductions will be numbered. So, before this B2 reductions are made B2 reductions are numbered. B2 reduction takes place just before that this M producing epsilon. So, this reduction will take place.

So, since this M producing epsilon is not going to give me any new code. So, this M dot quad which is the address of the. So, I can make that it can hold the index of a quadruple. So, if there is a function called nextquad. So, that gives me the index of the next quadruple to be generated. So, far we are generated till say up to this point, we are generated say 105 quadruples. So, this next quadruple is 106. So, what we can do is that we can make this M dot quad equal to 106. So, that will mean that that will be exploited later for correcting some addresses in the table. So, how is it being done? So, let us see in the successive slides.

(Refer Slide Time: 22:13)

**Translation Rules**

- $B \rightarrow B1 \text{ or } MB2$ 
  - { backpatch(B1.falselist, M.quad)
  - B.truelist = mergelist(B1.truelist, B2.truelist)
  - B.falselist = B2.falselist
- $B \rightarrow B1 \text{ and } MB2$ 
  - { backpatch(B1.truelist, M.quad)
  - B.truelist = B2.falselist
  - B.falselist = mergelist(B1.falselist, B2.falselist)
- $B \rightarrow \text{not } B1$ 
  - { B.truelist = B1.falselist
  - B.falselist = B1.truelist
- $B \rightarrow \{B1\}$ 
  - { B.truelist = B1.truelist
  - B.falselist = B1.falselist
- $B \rightarrow \text{true}$ 
  - { B.truelist = makelist(nextquad())
  - emit('goto' ...)
- $B \rightarrow \text{id1 relop id2}$ 
  - { B.truelist = nextquad()
  - B.falselist = nextquad()
  - emit('if' id1.place relop id2.place 'goto' ...)
  - emit('goto' ...)
- $B \rightarrow \text{false}$ 
  - { B.falselist = makelist(nextquad())
  - emit('goto' ...)
- $M \rightarrow \epsilon$ 
  - { M.quad = nextquad() }

So, this is the full set of rule that we have. So, we have got this first rule is  $B1$   $B$  producing  $B1$  or  $MB2$ . So, then what we do? So, at this point so,  $B1$  and  $B2$  they are truelist and falselists, they are known and this  $M$  it has got the quad as I was telling that at this point. So, I have got see this  $B1$  or  $MB2$ . So, this was the situation and  $B1$  true list has got all these points  $B1$  falselist has got this point  $B2$  true list has got this point and  $B2$  falselist has got this first point say. So, this true list and false list have been done.

Now, with so, what will happen if  $B1$  is false? So, if  $B1$  is false. So, I have to come to the  $B2$  part. So,  $B2$  evaluations starts at portion when  $B1$  evaluation ends and  $B2$  evaluations starts at this point and that is that will be. So, before that this  $M$  producing epsilon. So, this will be done and for this  $M$  producing epsilon the corresponding action that we take is  $M$  dot quad equal to nextquad. So, this  $M$  has got a special attribute quad, ok.

So, which holds the index of the next quadruple to be generated in the code generation process. So, this  $M$  dot quad is next quad. So, in this particular case. So, it will hold the start address of computation of  $B2$ . So, this way so, this so, this backpatch  $B1$  falselist with  $M$  dot quad. So, all these falselist, all these positions they were they are target was targets were not known now. So, once this  $M$  dot quad is known. So, all those values should be backpatched with  $M$  dot quad.

So, all those blank locations they can be filled up with the value of M dot quad then B dot truelist. So it is a. So, for B dot for B truelist is the collection of all true points of B1 and all true points of B2. So, this is the truelist of B1 of B say mergelist B1 truelist and B2 truelist. So, mergelist means, it will merge the truelists and get the overall thing and B falselist is B2 dot falselist, because B ones falselist have already been backpatched with start address of B2. So, it will branch from here to here in the execution and then B2s falselist. So, that will be that will be. So, Bs falselist will be same as B2s falselist similar type of rule can be made for this and so, B1 and MB2.

So, with backpatch B1 truelist with M dot quad B truelist is made equal to B2 dot falselist and B falselist is made mergelist of B1 falselist and B2 falselist. So, this is just the complementary version of this or rule the and rule now not of B1, B producing not of B1. So, B dot truelist equal to B1 dot falselist and B dot falselist equal to B1 dot truelist similarly B producing within bracket B1. So, this is truelist and falselist remain unchanged B producing true. So, this will be this will have this will this is generate a code. And this at that point the expression is definitely true; because the expression itself is being reduced by the rule B producing true, true being the constant token for this is the constant value for the Boolean expression true.

So, that is this emit goto. So, this has to be filled up, ok. So, where to go we do not know, because once this target will be known then only this can be filled up. So, this way it is left like this, but before that this nextquad function has been called. So, this is a point where the expression is definitely true so, that so, sorry this, whenever this goto is generated.

(Refer Slide Time: 26:37)

**Translation Rules**

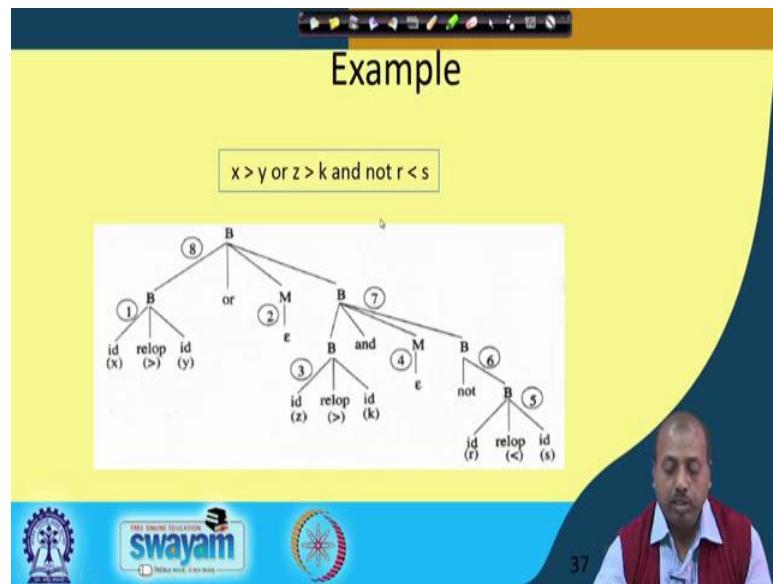
$B \rightarrow B1 \text{ or } MB2$ { backpatch(B1.falselist, M.quad) B.truelist = mergelist(B1.truelist, B2.truelist) B.falselist = B2.falselist }	$B \rightarrow B1 \text{ and } MB2$ { backpatch(B1.truelist, M.quad) B.truelist = B2.falselist B.falselist = mergelist(B1.falselist, B2.falselist) }
$B \rightarrow \text{not } B1$ { B.truelist = B1.falselist B.falselist = B1.truelist }	$B \rightarrow \{B1\}$ { B.truelist = B1.truelist B.falselist = B1.falselist }
$B \rightarrow id1 \text{ relop } id2$ { B.truelist = nextquad() B.falselist = nextquad() emit('if' id1.place relop id2.place 'goto' ...) emit('goto' ...) }	$B \rightarrow \text{true}$ { B.truelist = makelist(nextquad()) emit('goto' ...) <span style="border: 1px solid black; border-radius: 50%; padding: 2px;">100</span> }  $B \rightarrow \text{false}$ { B.falselist = makelist(nextquad()) emit('goto' ...) <span style="border: 1px solid black; border-radius: 50%; padding: 2px;">200</span> }  $M \rightarrow \epsilon$ { M.quad = nextquad() }

So, this particular quad, if it is quad number say 100. So, this is a place where I have to do the correction, ok. So, this makelist nextquad. So, it will make a list with 100 as one of the entry. Now there may be several other points that will come where B is definitely true. So, all these will be put into this chain, ok. And there will be that will be called B dot truelist and there will be backpatched with the when this goto target will be known.

So, all these places will be filled up since its target becomes a 200. So, I will be writing 200 here in that case. So, the code will be corrected then id1 relop id2. So, here also I have got the situation like B dot truelist equal to nextquad B dot falselist equal to nextquad then emit if id1 dot place relational operation id2 dot place goto and then again this part is not known.

So, this will filled up later when this B dot truelist this B dot true will become available. So, they will be backpatched with that, ok. So, this way we can have this Boolean grammar rules for them we can have the corresponding actions.

(Refer Slide Time: 28:05)



Now, so we can will look into this example. So, this  $x$  greater than  $y$  or  $z$  greater than  $k$  and not of  $r$  less than  $s$ , and this is the corresponding grammar that we have corresponding parse tree that we have. So, we initially have this  $B$  and since or is of list precedence. So, it is divided by this rule  $B$  producing  $B$  or  $B$   $M$   $B$   $M$   $B$  then after that. So, this or is  $id$   $relop$   $id$ . So, this first  $B$  is  $id$   $relop$   $id$ . So, this is  $x$  greater than  $y$  then the second one is now this  $M$ ,  $M$  is epsilon. Now, so, this now this so so, now, this so, this since or is of the list precedence. So, this is the second part of this or this whole thing is the second part this or and it has again had got two things like and not and etcetera.

So, first this before the end so, this  $z$  greater than  $k$ . So, that part is generated. So,  $B$  and  $M$   $B$  that is generated and this from this  $B$ , we get  $id$   $relop$   $id$  and that is  $z$  and  $k$   $M$  giving epsilon and then this  $B$  is gives me not of  $B$ . So, not of  $B$  and  $B$  giving me  $id$   $relop$   $id$ . So,  $r$  less than  $s$  so, this way we can generate the, we can write down the parse tree for this particular example string using the Boolean grammar. So, we look into the code generation for this in the next class.

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

**Lecture - 56**  
**Intermediate Code Generation (Contd.)**

So, in our last class we were discussing on translation of Boolean expressions.

(Refer Slide Time: 00:21)

**Translation of Boolean Expressions**

**B → B1 and B2**

```
{ B1.true = newlabel()  
B1.false = B.false  
B2.true = B.true  
B2.false = B.false  
B.code = B1.code ||  
gen{B1.true, ':'} ||  
B2.code  
}
```

**B → not B1**

```
{ B1.true = B.false  
B1.false = B.true  
B.code = B1.code  
}
```

**B → (B1)**

```
{ B1.true = B.true  
B1.false = B.false  
B.code = B1.code  
}
```

**B → id1 relop id2**

```
{ B.code = gen('if' id1.place relop id2.place 'goto' B.true) || gen('goto' B.false))
```

**B → true**

```
{ B.code = gen( 'goto' B.true ) }
```

**B → false**

```
{ B.code = gen( 'goto' B.false ) }
```

*if w7 then s1 else s2*

And in that process we have seen a first one grammar where it is that the Boolean expression grammar that we know from our previous discussions on the parser chapter. So, rule B may non terminal B can produce B1 and B2 or B1 or B2 or say B1 not of B1 or within bracket B1 id1 relop id2 like that.

(Refer Slide Time: 00:53)

Disadvantages

- Makes the scheme inherently two-pass procedure
- All jump targets are computed in the first pass
- Actual code generation done in second pass
- A single-pass approach can be developed by
  - Modifying the grammar a bit, and
  - Introducing a few more attributes
  - A few new procedures
  - Generated code can be visualized as an array of quadruples

So, the main difficulty with this type of grammar was that it is difficult to have a single pass through the source code and generate the intermediary code. Why do I say so is because of this reason that we have to have this goto's like say. So, at places we have written that, so, this particularly this rule where it is so, id1 relop id2.

So, if id1 dot place is say a relational operators say it is greater than. So, if it is greater than id2 dot place then goto B dot true. So, this B dot true may not be available when we are looking doing this particular reduction a typical example is done if then else statement. So, if some Boolean expression say x greater than y then we have got the statement block S1 else the statement block S2.

So, in the parse tree, so, it will be like this the if statement. So, the if statement it will have the keyword if then the Boolean condition B and then I will have the then part then S1 block and then else S2 block. Now this B has got this x greater than y. So, this has got that this particular parse tree so, x greater than y.

Now, the point is when this particular reduction is being made like when I am doing this reduction x greater than y to B. So, at that time so, this S1 code has not been generated the code for S1 has not been generated, code for S2 has also not been generated. So, we really do not know what is the where the control should jump if this condition is true or if this condition is false. So, so, that way we need a 2 pass procedure; in the first pass we

can just note down the, that entity start addresses of all this code blocks and in the second pass we fill up all these gotos like this goto this B dot true.

So, B dot true will get defined only when S1 code has been generated. So, if this is the target 5 this is the intermediary code file where you have generating the code may be in this part I have got the code corresponding to S1 and in this part I have got the code corresponding to S2.

So, B dot true should be filled up with this particular offset and B1 dot true sorry B dot true and B dot false should be filled up with this particular. So, these two addresses are offsets within the intermediary code file is not known, when I am doing this reduction. So, I have to make it a 2 pass procedure. So, to get to resolve this issue, so, we can make the intermediary code generation in a single pass by having the by modifying the grammar a bit and that modification is given by this grammar.

(Refer Slide Time: 04:01)

**Modified Grammar**

$B \rightarrow B \text{ or } MB$   
 |  $B$  and  $MB$   
 |  $\text{not } B$   
 |  $(B)$   
 |  $\text{id relop id}$   
 |  $\text{true}$   
 |  $\text{false}$   
 $M \rightarrow \epsilon$

$M$  is a dummy nonterminal with attribute  $M.\text{quad}$ , that can hold index of a quadruple

Consider the rule  $B \rightarrow B1 \text{ or } MB2$ :  
 Before the reduction of  $B2$  starts, reduction  $M \rightarrow \epsilon$  has already taken place. Hence,  $M.\text{quad}$  points to the index of the first quadruple of  $B2$

So, where we have introduced in another non terminal B another nonterminal M; so, this M produces epsilon. So, if you replace this particular grammar in this particular grammar. So, if you replace this M by epsilon, so, you see that this M does not have any effect ok. So, there will it is the standard Boolean grammar that we have.

However, this M producing epsilon, so, this is this can be very much useful, because when I am doing this particular reduction reproducing B1 or MB 2. So, the reduction is

like this B producing. So, here I have got the parts tree for B1 then or then M and then the parts tree for B2 and then this M produces epsilon.

Now when this; when this particular reduction will be done, so, when this when this B2 code will be generated. So, before that this M producing epsilon, so, this reduction will be done and M producing epsilon, so, it does not have any anything more. So, only thing.

So, at this time of reduction, so, if you look into the code the code that is generated, so, in that code file I will already have the code for B1. So, code for B1 is already there, now the code for B2 should come after this the code for B2 should come. So, in between, so, I need to know, what is this particular address this particular offset. And that can be found by the M producing epsilon production because once this is done. So, B1 code has been generated. So, if I have something like say next quadruple number or next quadruple address that next quadruple address if you look if you query for that. So, would be getting this particular address from where the code for B2 will be generated.

So, at this time of reduction, so, we will know where the next code will be generated. So, in that way we will know what is the code what is the start address for this the code B2. So, that will be utilised for generating the three address code in a single pass fashion. So, this is the thing that before reduction of B to starts the reduction of M producing epsilon has already taken place.

So, we have got an attribute M dot quad, so that can hold the index of a quadruple. So, that index is known; so, M dot quad points to the index of the first quadruple of B2. So, because I am so, if I ask you for the, if I somehow make the system. So, that it can return me the next quadruple index then that can be assigned to this M dot quad attribute and accordingly we can do something, so, that this code generation is proper. So, we will see how is it done.

(Refer Slide Time: 07:01)

The slide is titled "Translation Rules". It contains several boxes defining grammar rules:

- $B \rightarrow B_1 \text{ or } MB_2$   
{ backpatch(B1.falselist, M.quad)  
B.truelist = mergelist(B1.truelist, B2.truelist)  
B.falselist = B2.falselist  
}
- $B \rightarrow B_1 \text{ and } MB_2$   
{ backpatch(B1.truelist, M.quad)  
B.truelist = B2.falselist  
B.falselist = mergelist(B1.falselist, B2.falselist)  
}
- $B \rightarrow \text{not } B_1$   
{ B.truelist = B1.falselist  
B.falselist = B1.truelist  
}
- $B \rightarrow (B_1)$   
{ B.truelist = B1.truelist  
B.falselist = B1.falselist  
}
- $B \rightarrow \text{true}$   
{ B.truelist = makelist(nextquad())  
emit('goto' ...)  
}
- $B \rightarrow \text{id1 relop id2}$   
{ B.truelist = nextquad()  
B.falselist = nextquad()  
emit('if' id1.place relop id2.place 'goto' ...)  
emit('goto' ...)  
}
- $B \rightarrow \text{false}$   
{ B.falselist = makelist(nextquad())  
emit('goto' ...)  
}
- $M \rightarrow \epsilon$   
{ M.quad = nextquad() }

A handwritten note "100 goto . ." is written near the "true" rule, with an arrow pointing to a circled dot. Another arrow points from the "true" rule to a circled "out".

So, this is the modification of the grammar and the associated rule that we have got. So, first rule is for the or part so, this B producing B1 or MB2. So, as you know that by the short circuit because of you know that if B1 is true I do not have to evaluate B2, because this is an or but if B1 is false, so, I have to evaluate B2. So, what we do in B1's false list. So, if you remember, so, we said that this false list it has got this false list it has got the list of locations within the code of the B at which B is definitely false.

(Refer Slide Time: 07:41)

The slide is titled "Attributes". It lists two attributes:

- $B.\text{trueList}$ 
  - List of locations within the generated code for B, at which B definitely true
  - Once defined, all these points should transfer control to B.true
- $B.\text{falseList}$ 
  - List of locations within the generated code for B, at which B definitely false
  - Once defined, all these points should transfer control to B.false

So, so, this way it is basically, so, this B dot false list is the points at which B1 is definitely false. So, that that place so, we will be back patching with this M dot quad. So, M dot quad has got the start address of B2 so, they will be filled up with the start address of B2. So, that will be the backpack and for getting the true list that is true list is the attribute that has got all the points for the code for B at which B is definitely true.

So, B dot true list is it is created by merging B1 true list and B2 true list because these are all the points at which the expression B is definitely true and B dot false list is equal to B2 dot false list because these are the positions at which the overall expression B becomes false.

So, that we can modify the three address code generation statements the syntax directed translation mechanism so, that the locations are corrected. Similarly, if you look into this B producing B1 and MB2, so, there by the short circuit principal we know that if B1 is false we do not have to do anything, but if B1 is true we need to evaluate B2 also and B2 is start address or the start quadruple index is available in M dot quad.

So, that way it is doing this thing that back patch B1 true list with M dot quad. So, that way so, all that points at which B1 is true, so, there this M dot quad is put. So, that we can we can jump to the evaluation of B2. Bs true list is equal to be true false list. So, so, if sorry this is a mistake. So, this should be true list not false list so, should be true list. So, B true list equal to B2 true list because for the overall expression B.

So, this is true this true list will be wherever B2 is true. So, that that way it is going to be; going to be true that because at this point both B1 and B2 are truth. And Bs false list is either B1 is false or B2 is false. So, it is a merge merger of these two list B1s false list and B2s false list. So, by merging them we get the false list for B.

Now, not of B1 so, B2 list equal to B1 false list and B false list equal to B1 true list. So, it is just swap the locations at which B1 was true with the locations at which B1 was false to get the false and true locations for B. B within bracket B1 so, this is nothing, but the true list will be copied to B true list and false list B1s false list should be copied to Bs false list.

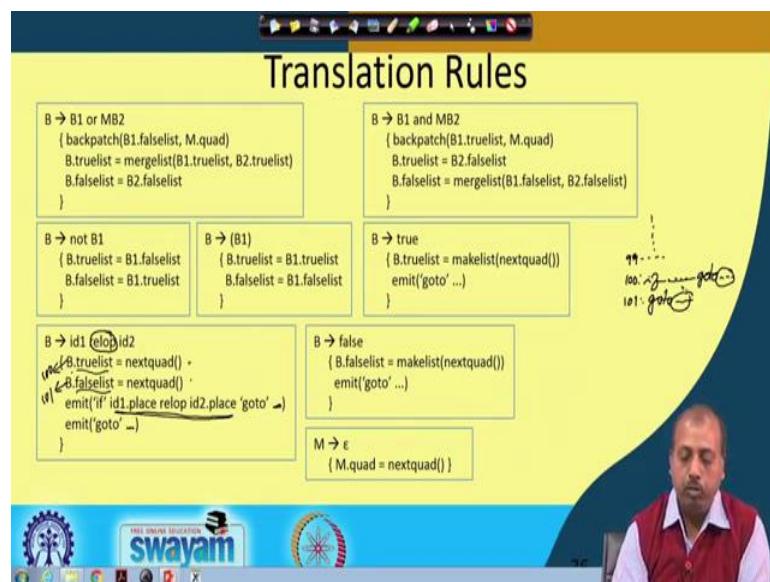
Now B producing true, so, this is I need to generate a code because there I have to generate a goto, this goto has to be generated because at this point this definition is this

Boolean expression B is definitely true so it generates a goto at some point ok. So, this value is unknown at present so, it will be filled up later. But this index, so, this three address code are being generated, so, what is the index of this particular line?

So, that is available by calling this function next quad ok. So, before generating this codes. So, before emitting this code or generating this code we are calling the function next quad. So, naturally next quad will return me the index of this particular quadruple and that is passed to the function make list. So, if this number is say 100, then it will make a list with 100 as 1 as an one entry and B dot true list is made to point to this. So, that will identify that if I am looking for B dot true list. So, this is the location where B is definitely true.

Similarly, so, this id1 relop id2; so, this is interesting. So, this is we B dot true list is next quad B dot false list is next quad then we generate the, if then else for the true and false part. So, it is relational operator may be any of the greater than less than those symbols.

(Refer Slide Time: 12:03)



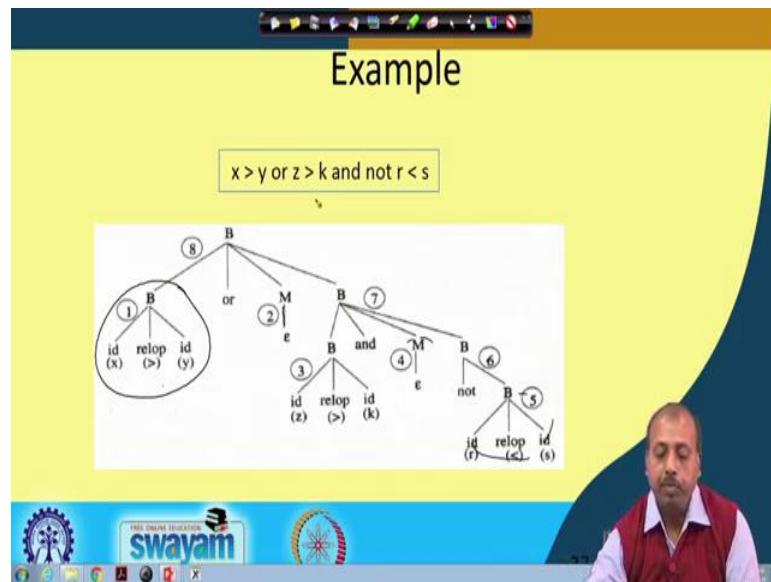
So, if id1 dot place is of relational operator id2 dot place. So, if this expression goes to be happens to be true ok, if id1 dot place is say relop is greater is greater than id dot place then go to. So, this part is not known, so, this part is there and then it generous this. So, before doing this so, you see that we have called is next quad function twice once to get the quadruple index for true list and ones to get the quadruple list for false list.

So, at present suppose I am at suppose I have generated code up to index say 99 up to this much has been generated. So, at this point if you call next quad, so, this B dot true list is becoming equal to 100 and B dot false list is become equal to 101 fine. Now it generates the code at 100 it generates the code that if etcetera etcetera that goto that part is generated and then at 101 it generates goto. So, this is nothing, but the if then else execution.

So, if id1 dot place is relationally operator having the relational operator id2 dot place then it will generate then it will goto this thing. So, this is the true list. So, this 100 happens to be the true list of B and 101 happens to be the false list of B. Later on, when this Boolean expressions target will be known like if it is a part of this if then else statement or while statement like that so, target will be known at that time we will be filling up this goto this positions with some backpatching procedure. So, this will be clear when we goto some statement like if then else or while loop like that.

Similarly, this B dot false list is just the replica of B dot B producing false is a replica of B producing true and so only thing is that instead of true list. So, we do a false list here and B dot false list is make list because this B is always false ok, so, there is no true parts. So, there is no true list true list is null and false list is equal to this, then M producing epsilon so, M dot quad equal to next quad. So, these are the functions that we have in the translation rules. Now how are you going to use this translation rules for generating some code so, that we will see.

(Refer Slide Time: 14:35)



So, suppose this is the; this is the Boolean expression that we have a  $x$  greater than  $y$  or  $z$  greater than  $k$  and not of  $r$  less than  $s$ . So, this is the parse tree, that is produced and if we number the reduction. So, first this reduction will be made then this  $M$  producing epsilon will be done, then it can do this reduction. So, that is number 3, then this can be done that is number 4, then this can be done number 5, then the 6, then this 7. So, you see that before this 7th reduction is made by this time is  $M$  producing epsilon has already taken place.

So, and for this part for this part of the Boolean expression the code has already been generated. So, after this the code for evaluating this second part will be there and this  $M$  dot quad will definitely have the corresponding quadruple index for the second part.

So, let us see how this code is being generated ok. So, initially so,  $B$  dot true list so at reduction number 1 so, id relop id.

(Refer Slide Time: 15:35)

Reduction	Action	Code generated	Full Code:
1	B.trueList = {1} B.falseList = {2}	1: if x > y goto ... 2: goto ...	1: if x > y goto ... 2: goto 3
2	M.quad = 3		3: if z > k goto 5
3	B.trueList = {3}, B.falseList = {4}	3: if z > k goto ... 4: goto ...	4: goto ...
4	M.quad = 5		5: if r < s goto ...
5	B.trueList = {5} B.falseList = {6}	5: if r < s goto ... 6: goto ...	6: goto ...
6	B.trueList = {6}, B.falseList = {5}		1, 6 true exit, 4, 5 false exit
7	Backpatches list {3} with 5 {3}, {5}	3: if z > k goto 5	
8	Backpatches list {2} with 3 B.trueList = {1,6}, B.falseList = {4,5}	2: goto 3	

So, if you look into this group. So, first it gets to next quad and that that are assigned to B dot true list and B dot false list. So, that is what is exactly done here. So, B dot true list is it is by calling the function next quad it will get 1 and B dot false list by getting the calling the function next quad we will get 2 and it will generate 2 lines of code as it is said by this emit statement. So, if id1 dot place etcetera. So, it generates this code that if x greater than y goto this is not known now and this is goto not known now and true list and false list for the B is kept as 1 and 2.

Now, for reduction number 2, so, this M producing epsilon. So, M dot quad equal to next quad that is the semantic action and M dot quad is equal to next quad. So, already we have generated 2 quadruple, so, next quad is at 3 fine. Now comes the reduction number 4. So, reduction number 4 is sorry reduction number 3. So, that is again another if there is id relop id. So, it will generate true list and false list next quad.

So, B dot true list equal to 3 dot false list equal to 4 and it will generate these two three address code like if z greater than k goto and 4 at 4 goto. So, this part is generated then this M dot quad equal to then this redetection number 4. So, M producing epsilon, so, M dot quad gets 5, because that is the next quadruple index. Then it will do reduction number 5. So, at this reduction number 5, so, again this is an if then else. So, 2 true list and false list will be created by calling the function next quad twice.

So, that is done here. So, you get the true list as 5 and the false list as 6. So, at it generates the code at offset 5 and 6 like if r greater than S goto and at 6 goto. And then at reduction number 6 not of B so, you know that true list and false list they will get interchange. So, no code will be generated, but this true list becomes equal to B1's false list so, that is 6 and these false list is equal to B1's true list that is equal to 5. Now I know that line now at reduction number 7 ok, reduction number 7 is an and operations. So, for this and you see first it says backpatched B1 true list with M dot quad ok.

So, B1 true list so, this at whatever at production number 3 whatever was the true list. So, that will be backpatched with the reduction number force M dot quad. So, so, backpatches list 3 with 5, so, this is the reduction number 4. So, this is the true list is having 3. So, at location 3, it will back patch with the value 5. So, it will be correcting this.

So, this value was previously unknown and now this value will be corrected and you will get a 5 here ok, so, that will happen. And when it says that so, that is so, in B1 and so, that so, that was a reduction number 7 you are looking into.

So, and so, and there is a. So, B dot true list is merger of B1 true list and B2 true list. So, that way this B dot true list is made B1 true list and B2 true list. So, those true list will be merged and that will be the true list of B. So, then it will then it comes to reduction number 7 so, that B1, so, with this B and MB2 and after that reduction number 8. So, in reduction number 8 so, it says backpatch list 2 with 3 and B dot true list is 1 6 and B dot false list is 4 5. So, this is created by merging the thing like at 1 and 5 whatever were the false list.

So, whatever that 1 and 6 this true list. So, they are merged getting 1 6 and the false list is equal to this 4 5 actually this merger is not shown here, this B and MB2 that at that at reduction number 7 there is a merger. So, so, this B dot true list. So, this and B, B dot true list is B2 for true list and B false list is merger list of 2 false list.

So, for this it will be 5 3 and 3 6, so, these 2 true lists are to be merged 3 and 6. So, these 3 is 3 is this 1 and 6 is this one. So, 3 and 6 so, they will be merged. So, that a result this true list will be 3 6 and false list will be B2 is false list that is 5 ok. So, in the next step, it will be doing the final reduction and it will be generating this code.

So, at the end this one and so, this is the code that is generated by backpatching say this location and these two backpatches have taken place. And then it the final true list will be 1 6 will be the true exit for the whole expression B and 4 5 false exit for the whole expression.

So, we will be taking more examples and it will be clear further. So, you just also practice something. So, we will also do some more exercises that will make it more clear.

(Refer Slide Time: 21:53)

The slide has a yellow header bar with a toolbar icon. The main title 'Control Flow Statements' is in bold black font. Below the title is a bulleted list of statements:

- Most programming languages have a common set of statements
  - Assignment: assigns some expression to a variable
  - If-then-else: control flows to either then-part or else-part
  - While-do: control remains within loop until a specified condition becomes false
  - Block of statements: group of statements put within a *begin-end* block marker

At the bottom, there are three logos: the Indian National Emblem, the 'swayam' logo (Free Online Education), and the Ashoka Chakra. The number '39' is visible in the bottom right corner of the slide area.

So, next we will be looking into the control flow statements like how do you have this control flows like if then else when while block then assignments. So, these statements how for them how are you going to generate the three address code. So, most programming languages they have a common set of statements like assignment, if then else while do and block of statements if they many a time some time will have got the begin and marker block marker sometime we have open brace close brace markers.

So, like that so, we have got different block markers. So, based on that the blocks are made. So, how to generate code for this type of statement?

(Refer Slide Time: 22:33)

The slide is titled "Grammar". It contains the following grammar rules:

$$S \rightarrow \text{if } B \text{ then } M S$$
$$\quad | \text{ if } B \text{ then } M S N \text{ else } M S$$
$$\quad | \text{ while } M B \text{ do } M S$$
$$\quad | \text{ begin } L \text{ end}$$
$$\quad | A \text{ /* for assignment */ }$$
$$L \rightarrow L M S /$$
$$\quad | S$$
$$M \rightarrow \epsilon /$$
$$N \rightarrow \epsilon$$

Below the grammar, there is a diagram showing a goto statement. It shows two quadruples, S1 and S2, with an arrow labeled "goto" from S1 to S2.

**Attributes:**

- $S.\text{nextlist}$ : list of quadruples containing jumps to the quadruple following  $S$
- $L.\text{nextlist}$ : Same as  $S.\text{nextlist}$  for a group of statements

Nonterminal  $N$  enables to generate a jump after the then-part of if-then-else statement.  $N.\text{nextlist}$  holds the quadruple number for this statement.

The bottom of the slide features the "swayam" logo and other navigation icons.

So, the grammar that we will be considering is something like this that the statement it can be an if then statement if then else statement, while statement, a block of statement or it can be an assignment ok. So, it is like this that statement is if Boolean expression then M. So, this M is introduced for the same purpose that we have seen previously for as a placeholder so, that we can generate the next quadruple index.

So, here in this, when so, when this reduction is being made so, by that time. So, we will be knowing the address for the starting quadruple address of S as results in B dot true list, so, we can put that particular quadruple number. So, it will be clear as we look into some example.

Similarly, if B then M S N else M S. So, this is so, if some Boolean expression B is true then it will come to this statement execution and that quadruple can be found by the M dot quad. And this so, this is introduced to have a goto after the then part. So, basically if this is the code for the then part that is the S1 and after this I am putting the code for S2 so, in between there should be a goto statement.

So, that should be goto otherwise what will happen if the condition is true. So, it will start executed at this point it will execute S1 then it will go into execution of S2, but you do not want that. So, if S1 is executed after that the control should come out and it should come to this point how is it should jump over the S2. So, for ensuring that so, this

N non terminal is introduced and we will see that it will generate a next it will generator a goto statement accordingly.

So, if B then M S N else M S similarly the while loop is modified while M B do M S and then for the block of statements so, this is begin and end. So, these are the block markers and this L is the list of statements and A is a simple assignment statement. So, and L for L that block of statement, so, it can be a single statement or it can be a null sequence of statements. So, this particular rule this two rules. So, they will be capturing that situation.

So, L producing L M S or S and the standard reduction like M producing epsilon, an N producing epsilon. So, they are there. So, we have got the attributes like S dot next list. So, that is that will have the list of quadruples containing jump to the quadruple following S ok. So, it will have the. So, jumps to the, so, it will have the list of quadruples where I have to put the next value of S after executing S what next where the control should go next.

So, at all the places so, in the block of S so, you may need to (Refer Time: 25:37) there may be a various points at which you need to tell like where what to do next. So, what will be the next quadruple entries? So, that so, those are all these locations they will be kept in S dot next list as and when that target will get define what we will do we will backpacks this value at this places so, that the generated code will be corrected.

And this L dot next list. So, this is again same as S dot next list expecting that this is for a group of statements so, we want to do that. So, that is L dot list and this nonterminal N as I discussed. So, it enables to generate a jump after the then part of the if then else statement and N dot next list it holds the quadruple number for this particular statement ok.

So, for this statement that is the goto statement. So, we will see the rules for generating the code.

(Refer Slide Time: 26:35)

$S \rightarrow \text{if } B \text{ then } M_1 S_1$   
{ backpatch(B.trueList, M1.quad)  
S.nextList = mergelist(B.falseList, S1.nextList)  
}

$S \rightarrow \text{if } B \text{ then } M_1 S_1 \text{ N else } M_2 S_2$   
{ backpatch(B.trueList, M1.quad)  
backpatch(B.falseList, M2.quad)  
S.nextList = mergelist(S1.nextList,  
mergelist(N.nextList, S2.nextList))  
}

$S \rightarrow \text{while } M_1 B \text{ do } M_2 S_1$   
{ backpatch(S1.nextList, M1.quad)  
backpatch(B.trueList, M2.quad)  
S.nextList = B.falseList  
emit('goto' M1.quad)  
}

So, first one; so, if the if then statement, so, if  $B$  then  $M_1 S_1$ . So, what we will have to do is that. So, so, in this particular reduction is being made you see that I have got this situation if  $B$  then  $M$  and  $S_1$ . So, by this time this  $B$  has already been reduced and we have got with associated with this  $B$  we have got  $B$  dot true list and  $B$  dot false list.

As we have seen in the Boolean expression conversion. So, whenever we have got have done this reduction, so,  $B$  dot true list is having all those places where the expression  $B$  is definitely true and  $B$  dot false list are all those places where the expression  $B$  is definitely false. So, this is already available. Now  $M$  producing epsilon so, that was there. So, we have only already we already know what is the next address of this after this where the code of  $S_1$  will be there.

So, and then this  $S_1$  has already been the code for  $S_1$  has already been generated and that start of set is available in  $M$  dot quad ok. So, in the file I have got the code for  $B$  I have got the code for  $S_1$  these are available and then and  $M$  dot quad actually having this particular index  $M$  dot quad is having this particular index.

So, what I can do is that at all the places where  $B$  is true. So, they can be connected with they can be backpatched with his  $M$  dot quad. So, there we had the goto part and goto left it the target I will leave the target as blank, now all those targets can be filled up with this  $M$  dot quad which is the start address of this  $S_1$ . So, we have goto somewhere.

So all these goto that were there in the code of the B where B is definitely true by consulting this B dot true list so, we can backpatch everything of this B dot true list due to M dot quad. Similarly now for false of course, I want to skip over this S1 fine.

So, for that for this statement S I do not know what is the next statement. So, so, when that next statement will get defined then I have to rectify all the places. So, what are the places I need to rectify? So, wherever B is false then I will be skipping over this S1. So, if somehow this address becomes known this offset becomes known then wherever B is false. So, there should be filled up with this particular quadruple index.

And similarly at within S1 also there maybe places where I need to generate that I want to the next statement. So, there also I will be filling up with this star. So, this is S dot next list is merge list of B false list and S1 next list. So, we will continue with this discussion in the next class.

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

**Lecture - 57**  
**Intermediate Code Generation (Contd.)**

(Refer Slide Time: 00:20)

```

S → if B then M1 S1
{ backpatch(B.truelist, M1.quad)
  S.nextlist = mergelist(B.falselist, S1.nextlist)
}

S → if B then M1 S1 N else M2 S2
{ backpatch(B.truelist, M1.quad)
  backpatch(B.falselist, M2.quad),
  S.nextlist = mergelist(S1.nextlist,
    mergelist(N.nextlist, S2.nextlist))
}

S → while M1 B do M2 S1
{ backpatch(S1.nextlist, M1.quad)
  backpatch(B.truelist, M2.quad)
  S.nextlist = B.falselist
  emit('goto' M1.quad)
}

```

So, similar to the if then statement so we can now look into if then else statement. Only thing is that now the else part is there so, it becomes slightly more complex. So, if I look into the parse tree, then the parse tree is something like this S producing if then this Boolean expression B, then the key word then M1 S1 N M2 N S2.

So, by this time the B has already been reduced this S1 has already been reduced M1 M and N so, they have already been reduced to epsilon and this is also done. And by the process of this reduction we know that M1 dot quad will have the start quadruple index for S1 and M2 dot quad will have the start index or start of set of S2. So, they are known with that we are going to generate code for this one.

And now you can understand that is you know similar fashion so wherever this B was true so, all those places the go to target. So, I can backpatch them with this M1 dot quad, because M1 quad will have the start address of S1. So, we backpatch B true list with M1 quad then B false list can be backpatch with M2 quad. So, in the previous if then part so, you see that we could not backpatch B false list because B false list target was not known

at that point. So, after this statement if I have the next statement then only I will be able to backpatch that. So, it was kept as the in as a part of the next list.

But in this if then else statement so, if this condition is false then the target is known. So, it has to go to S2 and task S2 start offset is available in M2 dot quad. So, this B dot false list can be backpatch with M2 dot quad. And what is the next list that way at which points I have to have to go to the next statement of S? So, that is defined by S1s next list, S2 next list and N next list.

So, all the three next list so, they need to be combined together and then only I will be getting the overall next list for this S. So, that is the code for this if then else type of statement the generating code for if then else type of statements so, we can have this type of actions. What about while loop? So, while loop is slightly because now I have to; I have to go back like once if the condition is true it comes into execution of S1 and after S1 is over; so, I have to go back to the evaluation of B to see whether the condition is still true or not.

(Refer Slide Time: 03:03)

```

Translation Rules

S → if B then M1 S1
{ backpatch(B.trueelist, M1.quad)
  S.nextlist = mergelist(B.falselist, S1.nextlist)
}

S → if B then M1 S1 N else M2 S2
{ backpatch(B.trueelist, M1.quad)
  backpatch(B.falselist, M2.quad)
  S.nextlist = mergelist(S1.nextlist,
                        mergelist(N.nextlist, S2.nextlist))
}

S → while M1 B do M2 S1
{ backpatch(S1.nextlist, M1.quad)
  backpatch(B.trueelist, M2.quad)
  S.nextlist = B.falselist
  emit('goto' M1.quad)
}

```

So, how do you do that? So, you see that again the if we draw the tree the parse tree then it will be something like this while M1 B M2 S1 and M1 and M2 they produce epsilon. So, this B has already been done and this S1 has already been done. So, this M dot quad actually points to the starting quadruple of B and this M 2 dot quad points to the starting

quadruple of S1 ok. So, that is the situation with which I am going to generate code for of while.

So, what do we do as the first statement with backpatch S1's next list with M1 dot quad because as soon as S1 is over so, I have to recalculate B and B is started address start offset is available in M1 quad so, that is what is done. So, this S1 next list is made to S1 next list is backpatch. So, wherever I had to I had to the next wherever this next was not defined. So, they are now backpatch with the start index of a start starting quadruple of set of S B; so, that is done.

Second thing that you have to do is wherever this B expression is true. So, all those places they should at that places I should go to execution of S1. So, this B's true list I have to backpatch with M2 dot quad, because M2 dot quad is having the start address of S1, so, all those places can be corrected and S next list. So, for the S next place it so, it comes out of this S only when this B becomes false ok. So, all the points where B is false ok; so, all the points where B is false so, they are the points to be corrected once I know the next statement of this while.

So, this S next list will have B false list and then it will generate another code. So, go to M1 dot quad. So, because the so far the code that is generated it has got the code for B and it has got the code for S1. So, when this S1 was looked into so, we did not know whether it is going to be part of the while statement. So, at the end I need to generate another goto statement to goto to where goto to this M1 dot quad. So, M1 dot quad is having the start starting quadruple index of B.

So, it will be the start of B goto the start of B, so, that is M1 dot quad. So, it will be done that way. So, the while loop will be implemented in this fashion. So, this major statement that we have seen that this if then else statement if then statement then this while statement; so, they can be done in this fashion.

(Refer Slide Time: 06:16)

Translation Rules (Contd.)

$S \rightarrow \text{begin } L \text{ end}$   
{  $S.\text{nextlist} = L.\text{nextlist}$  }

$S \rightarrow A$   
{  $S.\text{nextlist} = \text{nil}$  }

$L \rightarrow L_1 M S$   
{  $\text{backpatch}(L_1.\text{nextlist}, M.\text{quad})$   
 $L.\text{nextlist} = S.\text{nextlist}$  }

$L \rightarrow S$   
{  $L.\text{nextlist} = S.\text{nextlist}$  }

$M \rightarrow \epsilon$   
{  $M.\text{quad} = \text{nextquad}()$  }

$N \rightarrow \epsilon$   
{  $N.\text{nextlist} = \text{nextquad}()$   
 $\text{emit('goto')} \dots$  }

So, what about the block of statements like S producing begin L end ok. So, how are you going to do that? So, here this S dot next list is equal to L dot next list; so wherever whenever this next statement will come. So, I have got something like this begin some block of statements and then end and after that some statement will come in my program.

So, wherever this nexts are there this wherever this have to be defined; so they are to be there for this whole begin end block also. So, that is why S dot next is equal to L dot next list. So, if we have got a simple assignment statement like S producing A then there is no next as such. So, initialize this S dot next list to nil and then this one this one is interesting.

So, when we have got one statement followed by another. So, we have got a block so, we are breaking down a block of statements into a structure as if we have got a block of statement followed by a single statement that makes a bigger block.

So, this is bigger block is made using a structure like this that we have got a smaller block containing one statement less than the bigger block and the last statement is coming as a new one. So, in that case so, this rule will be something like this L produces this L1 M and S, where is L1 is a block of statements and this S is a single statement and this M producing epsilon is there.

So, will be used to hold the starting index of this quadruple of S. Now this L1's next lists; so after L1 is over after L1 is over so, it has to go to execution of S. So, all the places where this L1 next was necessary, so, those places can be backpatched with the start index of S that is M dot quad. And L's next list will be S next list because only the next part will come only from after this S has been executed. So, all this next so, they are pointing to this and for this big block overall block that next list is equal to S next list.

Then L producing S, so, L next list is S next list so, that is very simple. M producing epsilon, so that is M dot quad is next quad and this N producing epsilon so, this is a special case. So, N next list is next quad, but it also generate a goto statement, so, it will be using go to statements. So, we will see that this will be utilised for generating code for different types of instruction like this if then else statement we have seen.

So, this N dot quad so, that will be used for generating the go to after the then part; so, that is shown here. So, that part is shown here that N producing epsilon so, it is generating a goto statement at this point.

(Refer Slide Time: 09:40)

**Example**

**begin**  
  ~~while a > b do~~  
  ~~begin~~  
    ~~x = y + z~~  
    ~~a = a - b~~  
  ~~end~~  
  ~~x = y - z~~  
~~end~~

**Final Code:**

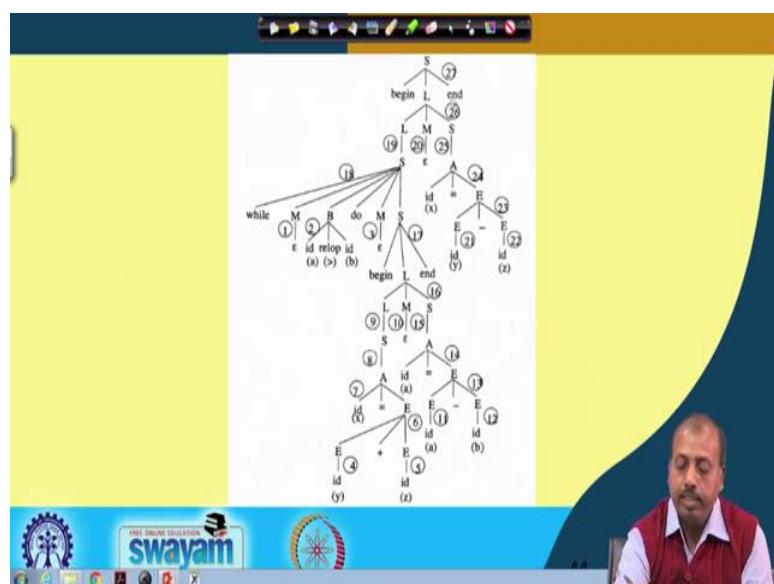
- 1: if a > b goto 3
- 2: goto 8
- 3: t1 = y + z
- 4: x = t1
- 5: t2 = a - b
- 6: a = t2
- 7: goto 1
- 8: x = t3

So, next we will be looking into some example. So, we have got a code fragment like this that begin while a greater than b do begin x equal to y plus z a equal to a minus b and x equal to y minus z and end.

So, that is the code fragment. So, the final code, so, we will see how this code is been generated, but before that you just try to visualise like how this reduction can how this code will finally, look like. So, this while a greater than b. So, it will be generated like if a greater than v goto 3, at 3 it will be doing this x equal to y plus z. So, t1 equal to y plus z then x equal to t1 then for this a equal to a minus b; it will be t2 equal to a minus b and a equal to t2. And then it will be doing this statement like a equal to t2 now it generates goto 1. So, from this point it is going back to this, so, that is the while loop.

So, this is a while loop being executed and if a greater than b is not true. So, it will not come to 3, but rather it will come to statement number 2 it will be doing goto 8. So, goto 8 is outside the loop so, this x equal to t3. So, this, y minus z has to be there. So, there should be another statement here there should be 8 where it is t3 equal to y minus z and this should be 9; this should be 9. So, we will see how this code is being generated by means of these three address code statements.

(Refer Slide Time: 11:30)



So, this is the parse tree ok, so, it is a bit complex. So, S producing begin L end then L producing L M S; so first here we have got a while statement.

So, that is the you have got first you have got a while statement here. So, that we are done begin end parts of this while statement in the block. So, that is while M B; while M B do M S that is a while statement M producing epsilon, B producing id relop id and then this S is another block of statement the body of the while loop.

So, body of the while loop has got two statements x equal to y plus z and a equal to a minus b; accordingly I have to generate a block of statements ok. So, this is begin end L so, that is generated and now it is generating L M S. So, L will generate the first statement; so, this will generate the first statement this S will generate the second statement.

Now, how this first statement is generated? So, L produces S, S produces the assignment and that is id equal to E. So, you can see that the expression was something like this that x equal to y plus z. So, this x will be id and this y plus z will be an expression, so, this will be done. So, this assignment id equal to E and this id is x and then we have got this E producing E plus E. So, E plus E giving me ideas id plus id, so, E and E so, that is id producing id producing id.

So, that gives us x equal to y plus z. Now for this part the this other word a equal to a minus b for that part it generates this S producers assignment statement it produces id equal to E, this id is a and this is E minus E then this it id E gives id that is a and this E gives id that is b that is a minus b.

And then after the while loop, so, we have got another statement id equal to x equal to y minus z. So, for that it will be doing like this then it will be generating this parse tree S producing a, a producing id equal to E and E producing E minus E and then why that is E giving id is here so, y and z. So, this is the whole parse tree that has been generated; now how to generate the code for that?

(Refer Slide Time: 13:58)

The screenshot shows a software interface with a table of reductions on the left and generated code on the right.

Red. no.	Action
1	$M.\text{quad} = 1$
2	$B.\text{truelist} = \{1\}, B.\text{falselist} = \{2\}$ Code generated: 1: if $a > b$ goto ... 2: goto ...
3	$M.\text{quad} = 3$
4	$E.\text{place} = y$
5	$E.\text{place} = z$
6	$E.\text{place} = t_1$ Code generated: 3: $t_1 = y + z$
7	Code generated: 4: $x = t_1$
8	$S.\text{nextlist} = \{\}$
9	$L.\text{nextlist} = \{\}$
10	$M.\text{quad} = 5$
11	$E.\text{place} = a$
12	$E.\text{place} = b$
13	$E.\text{place} = t_2$ Code generated: 5: $t_2 = a - b$

Generated code (lines 14-27):

```

14 Code generated:
15   6:  $a = t_2$ 
16    $S.\text{nextlist} = \{\}$ 
17   Backpatch(\{1\}, 5)
18    $L.\text{nextlist} = \{2\}$ 
19    $S.\text{nextlist} = \{\}$ 
20   backpatch(\{1\}, 3) => Code modified as:
21   1: if  $a > b$  goto 3
22    $S.\text{nextlist} = \{2\}$ 
23   Code generated:
24   7: goto ...
25    $L.\text{nextlist} = \{2\}$ 
26    $M.\text{quad} = 8$ 
27    $E.\text{place} = y$ 
28    $E.\text{place} = z$ 
29    $E.\text{place} = t_3$   
Code generated:  
8:  $t_3 = y - z$ 
30   Code generated:  
9:  $x = t_3$ 
31    $S.\text{nextlist} = \{\}$ 
32   Backpatch(\{2\}, 8) => Code modified as:
33   2: goto 8
34    $L.\text{nextlist} = \{\}$ 
35    $S.\text{nextlist} = \{\}$ 

```

At the bottom, there is a logo for 'swayam' and the number 45.

So, this three address code is produced is like this ok. So, the three address code produced is like this that this M dot quad so first reduction, so, you have to go by the reduction and then try to understand.

So, the first reduction that is made is this the reduction numbers if you see first reduction is this one, so, M M producing epsilon. So, M dot quad will be assigned as the next quad, so, that is 1. Then this is 2; so id relop id so, you remember that it is it will generate the code that if a greater than etcetera etcetera the true list false list etcetera.

So, how is it done? So, it will be generating a, so, it will generate these two lines of code if a greater than b goto and then line number quadruple 2 goto and it will have this b dot true list equal to 1 and b dot false list equal to 2. So, this is the code for the b part. Now deduction number 3 deduction; deduction number 3 is M producing epsilon. So, M dot quad will be equal to the next quad, so, M dot quad equal to 3. Now reduction number 4; so, reduction number 4 is your E producing id and then we have got this E producing id, so, E dot place equal to id dot place so, that was the action so for arithmetic expression.

So, E dot place E dot place is equal to y. Now at now at this point so, the reduction number 5. So, E producing id again E dot place equal to id dot place that is z. So, E dot place equal to z then we have got this reduction number 6.

So, reduction number 6 is this one E producing E plus E and when this E producing E plus E is coming then I have to get a new temporary variable at this point. So, if you remember the the code generation process for arithmetic expression. So, it was first generating E dot place equal to new temp is t1 and then it will generate a code that t1 equal to y plus z.

So, E1 dot place plus E2 dot place so, that is generated. Now reduction number 7, so, reduction number 7 gives us this one this assignment A producing id, A producing id equal to E. So, for that it has to generate the code that id dot place equal to E dot place. So, id dot place is x that is equal to t1 so, this code is generated. Now at reduction number 8, I have got S producing A, so, S producing A so, S dot next list equal to null. So, that was the action here, so, S dot next list equal to null. So, that was the action here so, S dot next list is equal to null; so that is done so, S dot next list is null.

Now, reduction number 9, so, L dot next list is S dot next list; so, that was the action. So, L dot next list equal to S dot next list. So, by that so, this is having L. So, this will also have L dot next list equal to that is also equal to; that is also equal to null. Now reduction number 10, so, reduction number 10 is M producing epsilon. So, M dot quad will get the next quad value that is 5, upto 4 we have generated code so, the next squad index is 5.

So, that is M dot quad equal to 5. Now reduction number 11, E producing id. So, this is E dot place is equal to id dot place so, that is done. Now E dot place equal to a similarly reduction 12 E dot place equal to b, reduction number 13 so, it is E minus so, it has to get a new temp and then generate a code.

So, this new temp is obtained like this that E dot place is equal to t2 the new temporary and now it will generate a code at quadruple index 5 that t2 equal to E1 dot place minus E2 dot place that is a minus b; so, that is reduction number 13. Now, reduction number 14; so, reduction number 14 is this one A assignment statement id equal to E.

So, that will be id dot place equal to it; it will generate this code that id dot place equal to E dot place, so, this a equal to t2. So, this code is generated at quadruple index 6. Now then reduction number 15, S producing A so, S producing assignment. So, this so, that is a S dot next list equal to null.

So, that will be done, so, S next list is null. Now at reduction number 16; so, at reduction number 16 we have got L producing L M S. So, for that L producing L M S it will backpatch L1 next list with M dot quad. So, it will backpatch L1 next list that is reduction 9s next list with reduction 10s M dot quad. So, reduction 9s next list is null and reduction that is 5 so, that is the backpatch null with 5, so, nothing happens.

So, that is no backpatching is done and this L dot next list will be equal to null so, that is there. And then reduction number 17, so, at reduction number 17 so, it is begin L end. So, this particular block and then for the begin L end box. So, S dot next list equal to L dot next list. So, as a result this S dot next list becomes equal to null. Now reduction number 18, so this is the while loop ok. So, first so, while loop actions were like this that so backpatch S1 next list M1 quad.

So, see if we do that. So, S1s next list so, this is 17th next list with backpatched with threes sorry so, reduction 1s M dot quad. So, 17th next list 7ths next list is null so, that has to be backpatched with 1 so, nothing happens ok. So, this is null nothing happens, but there is another backpatch at this point B true list with M2 dot quad.

So, B true list is that is reduction number 2 with 3 quad ok. So, B true list is 1, so, that will be backpatched with these three, so, backpatch this list 1 with 3. And naturally the line number 1, so, this code will be modified if a greater than b, go to 2. So, this this part which was unspecified previously now gets defined so, that is done.

Now after that this S next S next list equal to B false list. So, B false list was equal to 2 so, this S next list becomes equal to 2 and then it will generate a code, sorry it will generate a code go to M1 dot quad. So, it will generate this particular line of code this goto will be generated goto.

So, that is so, this S dot next list is null and then it will generate a goto part. So, this goto will be generated and then we can; so it should be goto 1 I think go to M1 quad. So, M1 quad is equal to 1. So, that way this goto 1 this statement that was here so, this is generated at this point. So, this 1 should be goto 1 not blank, this should be goto 1.

So, that while loop is done. Now, we have got in the statement. So, this while loop has been done, now we have got say this part. So, that is 17 is so, 18 is done now this 19.

So, L producing S; so, L producing S so, that is L dot next list is S dot next list. So, that way L dot next list is; L dot next list is equal to S dot next list that is equal to 2 that is done. Now reduction number 20, so, reduction number 20 is M producing epsilon. So, this M dot quad will be next quad that is equal to 8. Now reduction number 21, so, this E dot place will be equal to id dot place, so, E dot place will be equal to 21.

So, E dot place equal to y then at 22, E dot place will be made equal to z, now at 23 at 23 so, this reduction will be done. So, it will generate we will take a new temporary, it will take a new temporary and then it will be generating the code of E1 dot place minus E2 dot place.

So, this will be the thig that is new temporary t3 is generated that is E dot place and then this t3 equal to y minus z that is done. Now at 24, at 24 so, it will have this id equal to E. So, this id dot place equal to E dot place so, that code has to be generated. So, it generates that t3 equal to y minus z. Now come to reduction number 25, this S producing A; so, this S producing A so, this will have this sorry sorry sorry this 24.

So, that is the id equal to; id equal to E. So, id id dot place will get E dot place that code has to be generated. So, this S equal to t3 that is generated at quadruple index 9 and then at 25 so, S producing A so, dot next list equal to nil, so, that is done. Now at 26; so, this doing this L M S so, it will be back patching something this L M S the rule was like this backpatch L1 next list with M quad.

So, L1s next list, so, 19s next list will be backpatched with M quad. So, that way it will be back patching 2 with 8. So, the code will be modified as go to 8 ok, so, this code will be generated. And now if this L dot next list is null and finally, at 27 so, this S S S dot next list will be L dot next list and that is equal to null. So, we get the code in this fashion.

So, that matches with this one and then this go to 8 is generated at the if this if as part of this if then else this to 8 is generated. So, this is the final code that is generated and it is generated in this fashion.

So, this is the modified code of line number 2. So, it was go to blank here and that is backpatched with the value 8 at this point. So, this way it is a bit cumbersome but you see this is mechanical. So, you can just if you have the set of rules. So, you can always

do that and you can generate the corresponding set of code. Now only thing is that you have to remember the attributes properly and you have to apply those rules with patience so, but all these corrections that we are doing. So, ultimately for the grammar you see that the grammar modification and introduction of these rules. So, they are making it easy for generating the code.

So, we will be doing a good number of example. So, then it will be very clear like how this translation is going to go to be done automatically. So, first thing is that you have to draw a proper parse tree then you have to identify the sequence in which the reductions will be done.

So, the first challenge is drawing this parse tree and once you have drawn the parse tree the next challenge is to identify the reduction order like looking into this parse tree. So, you have to start looking from the leaf nodes and then you see which reduction from the left side can be done at the beginning.

So, that way you will get these reductions, like this M producing epsilon happens to be the first reduction then this happens to be the second deduction. So, if there is a mistake in this numbering then of course, you will not be able to do it by hand so, but this as far as the (Refer Time: 27:47) is concerned. So, it will not do any mistake because this S L are this LR parser (Refer Time: 27:51) so, they will always do the reductions in this way only.

But when you are doing an exercise by hand you have to be very careful that we do not; we do not get a mistake in the numbering these reductions. And once the reductions have been numbered so, we have to look into the corresponding; you have to look into the corresponding rules and the translation rules and then you have to apply those rules.

And you see that apart so, there are two parts one is at a different rule there may be some code generated portion and their maybe some portion which is corresponding to the actual some attributes. So, all these are so all these non terminals; so for this code generation purpose so, we have associated several attributes like true list, false list, next list, true false.

So, all those pointers then place of sets so, like that quad. So, like that there are many such attributes that we have associated with the non terminal. So, how to calculate those

attributes; so that is very critical like if again if there is a mistake in calculation of those offsets then also it will not give you the correct code generated.

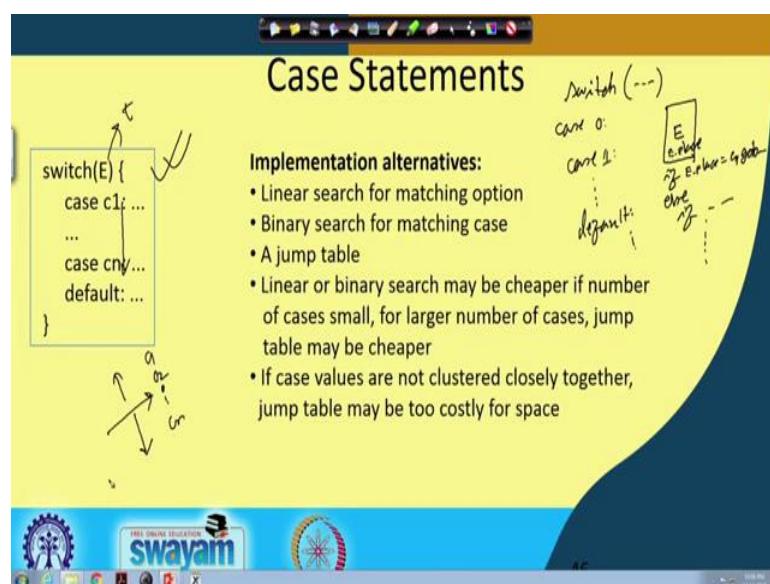
So, that way you have to be a bit careful, but it is a mechanical process as far as automation is concerned there is no problem; as far as doing it manually is concerned, so, it is a combustion. Anyway so, next we will be looking into some other issues like function calls etcetera and how to generate the corresponding three address code in the next class.

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

**Lecture - 58**  
**Intermediate Code Generation (Contd.)**

So, another very important statement that we have in many programming languages is the Case Statement or Switch Statement.

(Refer Slide Time: 00:21)



So, in typical case statement we will look something like this. We have got a for example, in the C language we have got this switch statement switch, some Boolean some expression and then we have got different cases, so case 0, case 1; so like that there can be different cases and there is a default case also.

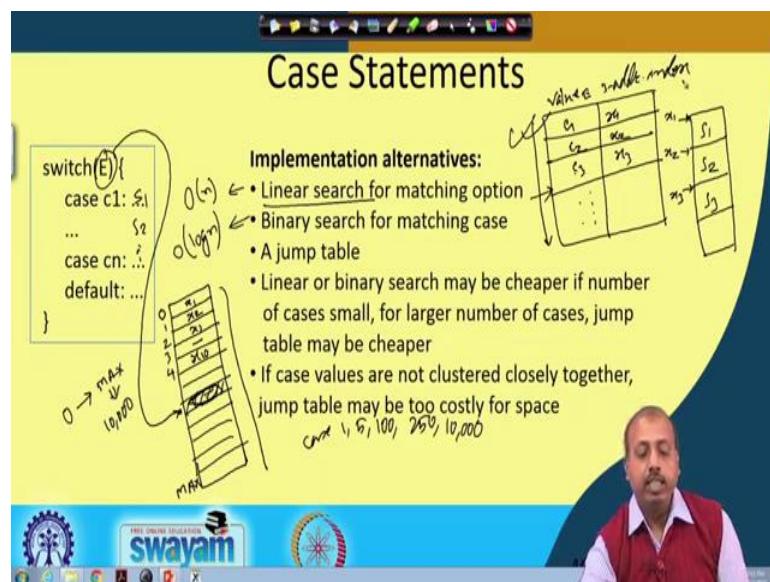
Now, as we know that this case statement can be modified into some if-then-else a branch a bunch of if-then-else statement. Like first we have the code for evaluating. So, if we if we take this as the format first you have the code for evaluating E and then you have this type of code; so if the final value is available in E dot place, so if E dot place equal to c1. So, if whatever be that E dot place equal to c1, then go to some code else again another if etcetera. So, this way you can generate a set of nested if-then-else statements and you can generate the corresponding case statement code.

So, so you can have different implementation alternatives. Like you can have a linear search for the matching option; like for after you have evaluated E you can have a you can have a search and you can say that if this particular number is matching, if the particular case value is matching with the value of the expression they will go to that option, so that is the linear search type of matching.

You can have a binary search for matching case because if we assume that this cases, so they are all going in ascending order then I can do a binary search after evaluating this into some temporary t. So, you can search for t into this alternate values alternative values c1, c2 up to cn. So, try at the middle and then that is a binary search. So, if it matches with t that is fine, if it is less than t, if t is less than this the middle value then you search in this portion of the area otherwise you search in the other part of this array. So, that was the binary search principles. So, here also you can have a similar such principle.

So, so after that you have some sort of a jump table. So, what is a jump table? So, jump table structure is like this; so I have got a table, ok.

(Refer Slide Time: 03:00)



So, in this table there are two columns. So, one column is the value of this switch value of E and this is the three-address code index, three-address index. Now, in the code part I have got. So, here if I say that this block is s1 this block is say s2 like that. So, in the code file I have got this codes likes. So, here I have got the code for s1, here I have got

the code for s2, s3 like that. Now, if this start address is say x1, so this start address is x2, these start of set is x3 like that. So, you can have a table where c1 is x1, c2 is x2, c3 is x3. So, you can have this type of table.

Now, so, this we call a jump table because, so by consulting this table you can figure out where to jump, ok. Now, we have to search through this table to get the jump target that is x1, x2, x3 etcetera. Now that is what I can search I can search using a linear search method I can try match with all these values one by one or we can do a binary search as I was telling. So, you can (Refer Time: 04:19) back the middle of the table, see whether the value is equal or less or greater and accordingly you can do that search. And as you know that linear search this has got a complexity of order n if there are n alternative cases, whereas the binary search we will have a complexity of order log of n for n number of cases.

Now, this jump table structure is good and this linear or binary search is cheaper if number of cases is small, ok. However, for this for this larger by larger number of cases, so jump table may be cheaper because; so if this, so other implementation that you can do is that you can just instead of doing the jump table instead of keeping all these values here directly, so you can use another type of jump table where it is like this. I do not keep the value of this c1, c2. So, they are implicit the values of c1, c2 are implicit. So, suppose this value of this expression it can go in the range from say 0 to say some maximum max.

So, I have got a table where this indices they run from 0 to max, fine. And then at so assuming that there is a case for case 0 and that is x1, so I write x1 here. So, for 1, if the value is 1 suppose the jump target is x2, so I write x2 there. So, for 2 it may be x3, for 3 it may not be there, case 3 may be absent, case 4 maybe something say x10. So, like that. So, there are, so now what happens is that you do not need to do a linear search or binary search at all.

So, once you have evaluated the expression E, so you can use it directly to index into this table. And say this particular E t s; so, suppose this E evaluates to this particular index, so you take the jump target from here and go to that particular address. So, that way we can have a very efficient jump table implementation rather than searching through the searching through the index values.

Now, this is good if the number of cases are number of cases are large, so that way the jump table may be cheaper. However, it may so happen that numbers are more, but this the that the values are all the cases and not present, in the sense that this max may be a very large number. So, this may be say 10,000, ok, but the actual cases that you have are having say case 1, 5, 100, and say 250, and 10,000. So, these are the only few cases. But if you are thinking about this type of implementation then you have to keep a table with 10,000 indices whereas, if you are doing a if-then-else, so this type of realization where you will be comparing with each and every location and then come to a decision, so that way the table maybe compact.

So, this is a tradeoff between this normal implementation with linear search and a pure jump table based implementation. So, depending upon the choice of the compiler designer, so they can go for either of these two options. So, depend on, so it depends on the type of programs that are being written in the system, ok.

(Refer Slide Time: 07:46)

**Jump Table Implementation**

Let the maximum and the minimum case values be  $c_{max}$  and  $c_{min}$  respectively

```

Code to evaluate E into t
if  $t < c_{min}$  goto Default_Case
if  $t > c_{max}$  goto Default_Case
goto JumpTable[t]
Default_Case:
    ...

```

$JumpTable[i]$  is the address of the code to execute, if  $E$  evaluates to  $i$

So, this is the jump table implementation procedure. So, if maximum and minimum case values  $c_{max}$  and  $c_{min}$  respectively. So, first you have a code to evaluate the expression  $E$  into some  $t$ . So, first we have the code in this part evaluate  $t$  and ultimately this local variable  $t$  has got this  $E$  dot place. So,  $t$  corresponds to  $E$  dot place; so we have this thing.

Now, there a default cases. So, if this  $t$  happens to be less than  $c_{min}$   $t$  happens to be more than  $c_{max}$ ; that means, that that they will not match with any of the entries in the

jump table, so they will go to the default case. Otherwise it will as I was telling, so it will be consulting one particular entry in the jump table. So, if this is the corresponding jump table.

So, it will be consulting this particular jump table, and this  $t$  value will be used to index into this table and suppose it fix up, so this particular entry and goes there. So, that way I left this jump table  $t$  go to jump table  $t$ , otherwise this default cases (Refer Time: 08:52) this level will be there, and this go to statements are also there that will take the default statement, default part execution like this table.

So, jump table  $i$ . So, it will contain the address of the code to execute if  $E$  evaluates to  $i$ , so that is the jump table implementation. So, we can, so these are the alternatives that are done in realizing this case statements in programming languages for the three-addressed code translation.

(Refer Slide Time: 09:22)

## Function Calls

- Can be divided into two subsequences
  - Calling sequence: set of actions executed at the time of calling a function
  - Return sequence: set of actions at the time of returning from the function call
- For both, some actions performed by Caller of the function and the other by the callee

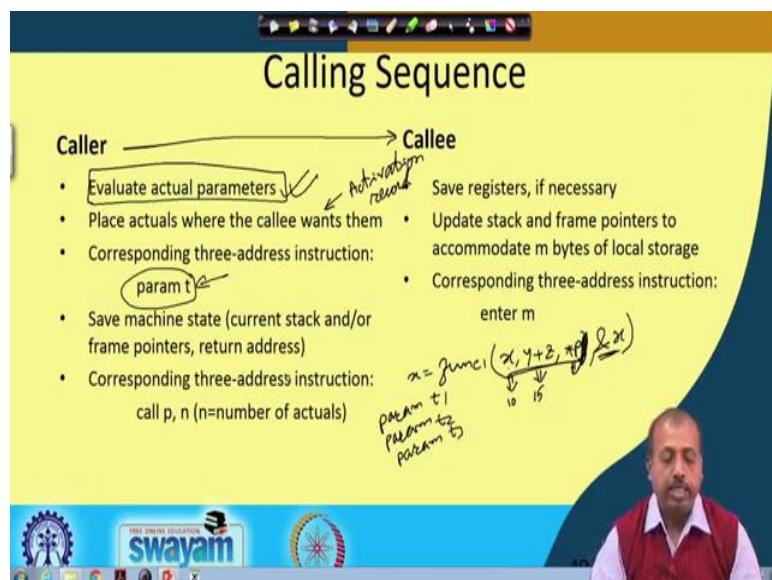
48

Next, we will be looking into the function calls. So, function calls they can be divided into two subsequences, so calling sequence and the return sequence. So, calling sequence is a set of actions that are to be executed at the time of calling a function and return sequence is the set of actions to be executed at the time of returning from the function call.

Though it appears to be pretty simple that you just note down the return address keep it in the stack and go back, go to the function and while coming back just take the return address from the stack, but that is not all. As we have seen that this runtime environment management, so it tells that you have to do many things for supporting recursion and all, so that way you need to do many many actions.

There is, there will be a set of actions at the time of returning from the function also at the time of going to the function also. And these actions are also distributed, in the sense that some part of the action it should be taken by the caller function and the other by the callee function. So, both of them have got definite role to play for generating this function calls, implementing the function calls.

(Refer Slide Time: 10:35)



So, what are the responsibilities? For the calling sequence, so caller so for the calling sequence the responsibilities are like this, the caller it is going to call the callee routine, ok. So, in the process before going to the callee routine, so it will be first evaluate the actual parameters, then place actuals where the callee wants them, then corresponding three-address instruction is the param t, parameter t. Then save machine status state that is current stack under frame pointer return address etcetera and the corresponding three-address instruction is called p n, where n is the number of actuals.

So, what do you mean by this? So, maybe I have given a call to a procedure or a function say x is equal to save function 1 and there I am parsing the parameters like x y

plus z star p like that. So, these are suppose these are the 3 parameters that I have passed. Now, this (Refer Time: 11:41) this x y plus z and star p they need to be evaluated first, ok. So, they are evaluated. So, this x maybe current value is 10, y plus z current value maybe say after evaluating this expression, so this may be 15, so that way I should have some code for evaluating this expressions, ok. And then the star p it has to be this a pointer axis, so it has to go to the corresponding location and get it. Sometimes we need to change send the address of a address of a particular variable. So, is send it like am percent x. So, then the this has to be evaluated.

So, address of a x has to be found out, so that evaluation code has to be there. So, this evaluate actual parameters though it is very simply written so, but it involves some proper amount of code and in fact, one parameter parsing the parameter that I am parsing that itself maybe another function call. So that way it becomes more complex, fine.

So, we place the, so after we have evaluated them then we have to place the actuals where the callee wants them. So, where will the callee want them? So, normally for this runtime environment management we have seen that the activation records are there, so the activation record will be created and all the all the parameters. So, they will be put into the activation record.

From the three-address code point of view, so we will call it as parameter t. So, if there are 3 parameters. So, there will be 3 parameter statement param say t1, param t2 and param t3. So, but the idea is that this param t1, t2, t3 when they are executed. So, they will actually be copying this values of this parameters on to the activation record slots. And after that it will save the machine state, so current state and of a frame pointer return address. So, whatever is required by the operating system, whatever is required to return from one procedure call and also it will be saving all those information into the into some proper place. So, it may be stacked, it maybe some other memory location etcetera, but it will do that.

And then, the accordingly the corresponding three-address code level, so just to model this phenomena at three-address code level we will have the statement call p n. So, it is calling the procedure p with n number of actuals n number of parameters, so that is the thing that will be done at the caller end.

At the callee end, so it will save registers if necessary. So, if the caller has not saved registers and it is necessary that this whatever register this caller had, so that should not be disturbed. So, it will first save the CPU registers into some (Refer Time: 14:33), some memory locations then update stack and frame pointers to accommodate n bytes or local storage, because at this point thus the callee routine knows like how many bytes of local storage it has all the variables that are defined in this.

So, it was not possible at the caller point because caller will not know like how many parameters, how many local variables the callee routine will have. But once we are in the callee routine, so we know the when the callee routine is parsed. So, by that time we know what is the number of parameters that are (Refer Time: 15:08) the number of local variables that are going to be there in the procedure in the function.

So, accordingly it will accommodate m m number of bytes of local storage and corresponding three-address code that will be using is enter m. So, a three-address code level keep it simple because this is this is a very machine dependent feature, so we do not want to go into much detail of it. So, I just keep a note that enough actions have to be taken, so that they are created in this in this fashion. So, this all the local variables are created there. So, that is about the calling sequence.

(Refer Slide Time: 15:49)

Callee	Caller
<ul style="list-style-type: none"><li>Place return value, if any, where the caller wants it</li><li>Adjust stack/frame pointers</li><li>Jump to return address</li><li>Corresponding three-address instruction: return x or return</li></ul>	<ul style="list-style-type: none"><li>Save the value returned by the callee</li><li>Corresponding three-address instruction: retrieve x</li></ul>

Now, for the return sequence, so it is just the other way. So, the callee routines, so it will place the return value if any where the caller wants it. So, normally as we know that in

the activation record, so there is a slot for this return value. So, depending upon that setting; so it will be putting this return value on to that particular slot. It will adjust the stack and frame pointers. So, maybe the pointer are to be reduced or increase whatever it is.

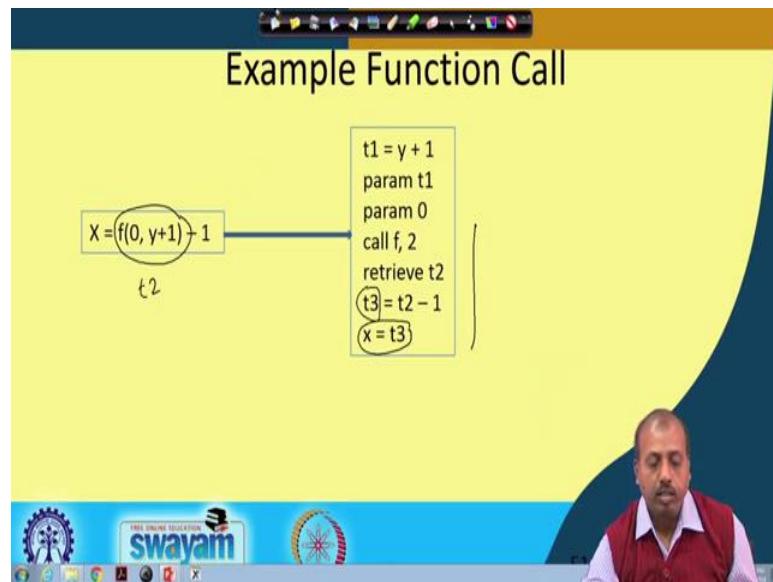
So, to it has to it maybe it has to get rid of that frame and all. So, all those things are done by adjusting the stack and frame pointer. Then it will jump to the return address. So, they say it will take the return address from the activation record and it will jump to the return address. So, the callee, that that way the callee will come back from the from it to the caller routine. And the corresponding three-address instruction is return x or return. So, this return x it will be returning the return value x and the simple returns. So, this is a void type of return. So, there is nothing there is no value that is return; so it is a void procedure call.

On the other hand if you look into the caller. So, it has got the responsibility to save the value returned by the callee. So, callee has return some value into some variable into some into some slot in the activation record, so that has to be saved into some register into some variable. For example, it may be like say x equal to sum function f1 etcetera.

So, at the end, so this function once activation record. So, it will have this return value somewhere, so that value has to be copied into x, ok. So, for that matter, so it has to generate three-address code where some temporary will be equal to this return value and then this x will be equal to that temporary. So, something like that has to be the done. So, the corresponding three-address instruction is retrieve x.

Since, you do not know from which how this the return value will come exactly in the target machine target machine language. So, we just we just keep it this retrieve statement. So, the caller it will be retrieving the return value from the activation record. Now, how will it retrieve; so those things are not known at this point because that is very much machine specific.

(Refer Slide Time: 18:17)



So, next we look into an example how this function call will take place. So, this  $x$  equal to  $f(0, y+1) - 1$ , so these are the two parameters passed and minus 1. So, this first it will be evaluating the parameters. So, you see that the first statement was evaluate the actual parameters. So, first that evaluation is being made  $t1$  equal to  $y + 1$  param  $t1$ . Then for the next one next parameter is 0, so that is also evaluated, so that its param 0.

Then it is call  $f$  and the number of actuals like how many parameters are passed etcetera, so how many local variables are there. So, all those are taken care, so call  $f2$ . Then after calling after returning; so here I do not have the code for  $f$  in this particular piece of statement. So, I assume that code will be coming somewhere else. But after returning from that code, so the caller routine, so it has to retrieve the value from the return value from the activation record, so that is retrieve  $t2$  it is expected to do that.

So, it will be retrieving the value from the activation record and then  $t3$  equal to  $t2$  minus 1, so it will be retrieving it will be doing the computations after  $t2$  has come. So, this  $f(0, y+1) - 1$ , so this is available in  $t2$ , so this  $t2$  minus 1, so that is made equal to  $t3$  and this  $x$  is finally, equal to  $t3$ . So, this is the three-address code generated for the parameter part, for the function call sort of thing.

(Refer Slide Time: 20:00)

The slide has a yellow header bar with the title "Storage Allocation for Functions". Below the title is a bulleted list:

- Creates problem as the first instruction in a function is:  
enter n /\* n = space for locals, temporaries \*/
- Value of n not known until the whole function has been processed.
- There can be two possible solutions
  - Generating final code in a list
  - Using pair of goto statements

To the right of the list is a handwritten note:

int f(a, b)  
{  
int x, y;  
x = y + a \* b;  
y = y + 15;  
x = x / (y + 15);  
}

A man in a red vest is visible on the right side of the slide.

So, that makes it and the caller side. Now, for the callee sides there we have to do the storage allocation part, ok. So, because this is the local variables are to be created and all. So, but here is a problem because as the first instruction in a function is enter n, so enter n so it will create space for the locals temporaries etcetera and value of the value of n is not known until the whole function has been processed. So, how much local space has to be allocated, so that is not known at this point.

So, the typical situation that I am I can think about is maybe this is it this is a function the integer f1 it has got some parameters, and into this I have got the local variables like say integer x and y and I write some expression involving x y etcetera, x equal to y plus some parameters passed a b c x plus a into b then y equal to y plus 15. Then again x equal to say x divided by say y plus 19. So, like that there maybe number of statements.

Now, one thing that is clear that for this x and y you need space; so for then we can calculate the space. But the difficulty comes because for this expression x equal to y plus a in to b. So, this is not a single. So, the for this, so this whole competition cannot be done at ones, because three-address code we have assume that there are only two operands per instruction, for competition.

So, it has to be done like t1 equal to a star b, then t2 equal to y plus t1 and then x equal to t2, so that way this extra two temporaries t1 and t2 have got created. Similarly, for this statement on this statement also, so other temporaries will get created. Now, when this

temporary is are getting created, so this is not known at this point, ok. So, until unless I have seen this entire function, I do not know how many temporaries will be coming. So, it is not known the value of n is not known until the whole function has been processed.

So, how to handle this situation? So, there can be two possibilities. So, either we can generate final code in a list. So, we can say that instead of writing quadruples on to the file the code quadruples on to the file we just keep them in a list.

(Refer Slide Time: 22:42)

Storage Allocation for Functions

- Creates problem as the first instruction in a function is:  
enter n /\* n = space for locals, temporaries \*/
- Value of n not known until the whole function has been processed.
- There can be two possible solutions
  - Generating final code in a list.
  - Using pair of goto statements

So, so this we have got the first statement as enter statement and this it will have another part which is the n part which is not known. And then I point to the next statement block, so next statement block that are generated. So, they are kept in a link list. So, at the end of this entire generation I will know the what is the value of n, so how many bytes of storage is necessary, then I will back patch, so this location with that particular value, ok. And then so then after that this whole code will be written on to the code file the intermediate code file in this order, so that is one possibilities.

So, generating the final code in a list, so that I can do all this corrections and then I can just dump on to a file. Other possibilities to use a pair of go to statements. So, use a pair of go to statements, so you can solve this problem. So, you will see how this can be done.

(Refer Slide Time: 23:43)

The slide has a yellow header bar with the title 'Generating Final Code in List'. Below the title is a bulleted list of six items:

- Generate final code in a list
- Backpatch the appropriate instructions after processing the function body
- Approach is similar to single-phase code generation for Boolean expressions and control flow statements
- Advantage: Possibility of machine dependent optimizations
- May be slow and may require more memory during code generation

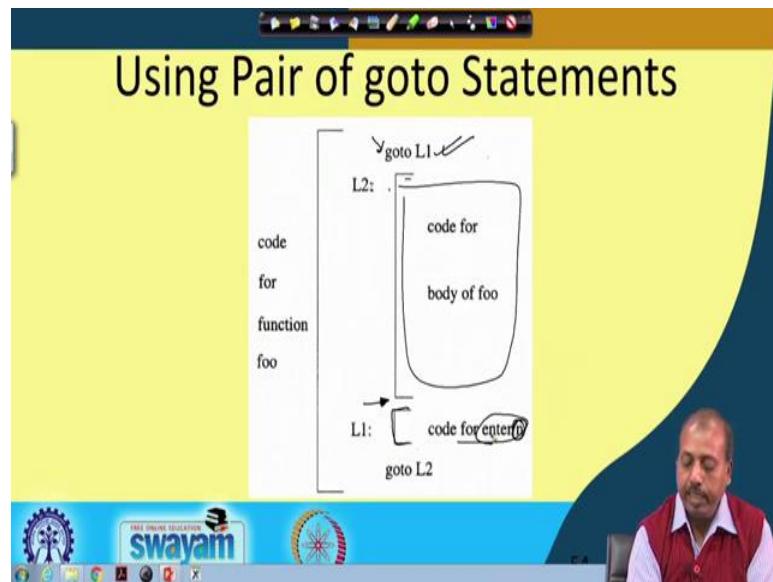
At the bottom of the slide, there is a blue footer bar featuring three icons: a logo of a temple gopuram, the 'swayam' logo with the text 'FREE ONLINE EDUCATION SWAYAM', and a circular emblem. The number '53' is also visible in the bottom right corner of the slide area.

So, this is the part which is generating final code in a list. So, generate final code in a list and back patch the appropriate instructions after processing the function body. This approach is similar to single phase code generation for Boolean expressions and control flow statements. So, in single phase generation also we have seen that we have producing the codes and after that we are doing some back patching.

So, here also I am telling the same thing you generate the code in a list and after that you do a back patching of them back patching of the code, so that is one possibility. So, it is possibility of machine independent optimization, so that is one advantage. So, we can do many optimization, so which are dependent on the machine. So, you can modify the statements accordingly, but it may be slow and may require more memory during code generation.

So, this linked list type of organization as you know that as per as accesses is concerned. So, it is going to be slow because it has to travel through this dynamic memory and all, so it takes time. And the second thing is that the memory requirement is also more because the (Refer Time: 24:49) because the amount of space necessary for holding the pointer, so they will come into the picture. So, this final code generation using this approach is difficult.

(Refer Slide Time: 25:00)



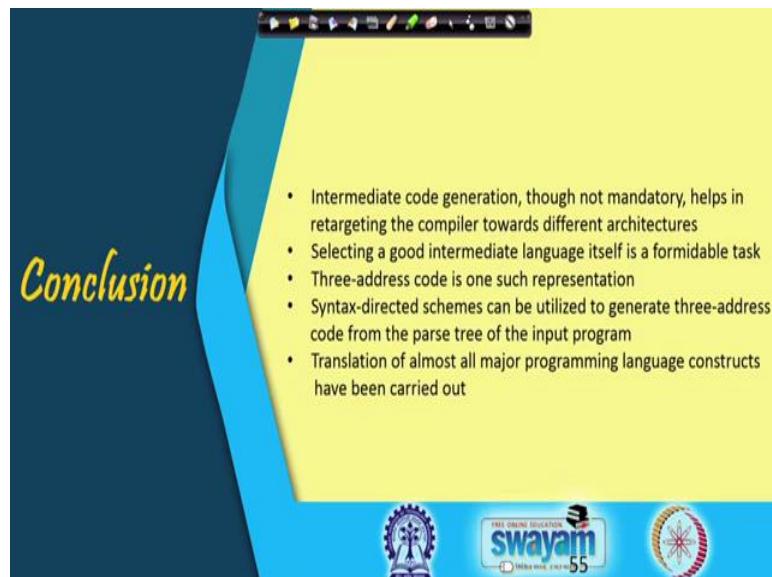
So, at the using a pair of go to. So, the code, so this is an this is an example suppose we have got a function called foo and in this function we have got. So, at the beginning of the function we generate a go to L1. So, in the here I have put the code for enter n, so that will be put into this L1 part. So, while generating the code. So, we just do it like this. So, we generate a level, so at this point we generate a new level and we put it, we generate a go to statement at this point. So, this is the.

Now, we generate another level L2 and we call it like this. So, they are the code for the body of foo is presented. So, here I have got the body of the procedure. Then at L1, so once, we are once we are at this point once the entire procedure and entire function has been generated the code has been generated we know how many temporary were used; so what is the space requirement and all. So, I can have this code for enter n. So, this n value is now known.

So, I can have the enter n thing. And after that we generate a go to for L2, so it comes in. So, how the whole function works? As soon as it is called, so it will be starting with this go to statement it will come here. So, it will it will have the code for enter n, so that there it will be allocating enough space for the temporaries and all. And then it will come to this statement go to L2. As a result, it will come here and start executing the body of the function.

So, this way using a pair of go to statements we can solve this local param in, local variable issue.

(Refer Slide Time: 26:48)



So, next will be coming to the conclusion; so we have seen a number of intermediate code generation mechanism. So, it is though intermediate code generation is not mandatory, it helps in retargeting the compiler towards different architectures. Now, selecting a good intermediate language itself is a formidable task because they have to be equally distance from the source language and the target language.

Three-address code is one such representation that we have seen it is very versatile and it is powerful that way. And syntax directed schemes are best used for generating the three-address code from the parse tree of the input program. And translation of almost all major programming language constructs we have seen and we can do this translation using them.

So, after this intermediate code has been generated. So, you can go to the code optimization phase, and the target code generation phase followed by code optimization. Now, the target code generation is nothing, but a template substitution. So, for is intermediary code templates, so you can think about a best way to implement the target code for that and accordingly you can generate code for that. And also, if that the optimization part, so you can do some optimization at the intermediary code level itself like you can find that the same sub expression is being computed again and again.

So, they can be completed only once, put into some temporaries used there. Or maybe some competition you can find out that it is not necessary because it is may be carried out in a loop, but it is not dependent on the loop index. So, they can be taken out of the loop. So, this type of optimizations can be done at the intermediary code itself.

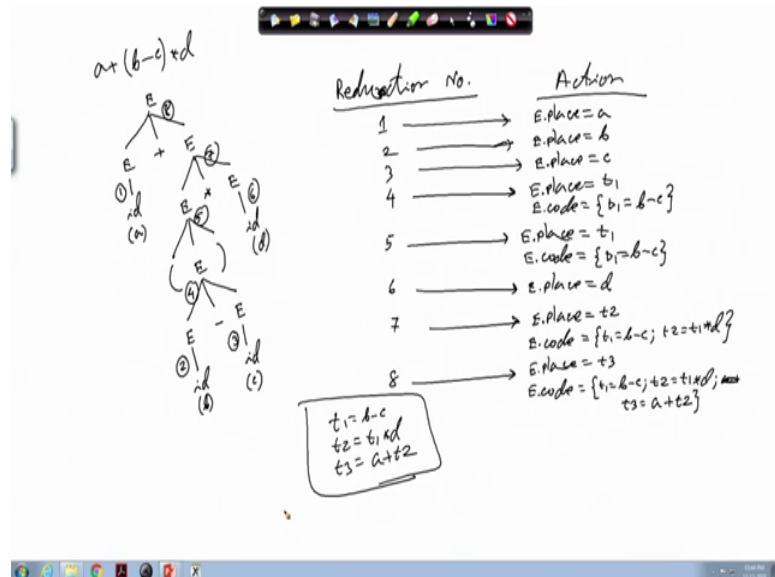
However, more important type of optimizations like say by putting the variables into registers and all, so that will be difficult because that will require the exact machine architecture and that can be done only at the target machine level, you cannot do it an intermediary code because we do not know how many registers we will have. So, this way, so the later part of the (Refer Time: 28:57) advanced concepts on this compiler. So, they will be discussing about this optimization issues, this target code generation issues and all. So, we will end our theory class with this. So, we will be doing some more exercises in the next classes.

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

**Lecture - 59**  
**Intermediate Code Generation (Contd.)**

So, next we will do some exercises on this 3 address code generation procedure. And as I told you just like doing the parsing; so this is also a very cumbersome job and you have to have patience for solving problems on this particular topic. So, first we take an example where we are having this arithmetic expression say a plus b minus c into d.

(Refer Slide Time: 00:33)



So, simple arithmetic expression like this. So, first we try to draw the arithmetic parse tree for this. So, this is E producing E plus E; then this E can give me id which is a and then this E from this E I have to generate this E star E. So, this E can give me within bracket E; then this E will give me E minus E, then this E will give me id which is b and this E will give me id which is c fine and then this E will give me id which is d.

So, this is the parse tree that is produced. So, next job is to number the reductions; so this reduction will be done at the beginning. So, this is the this is a 1st reduction then this is the 2nd reduction; this is the 3rd reduction, after that this is the 4th reduction, this is the 5th reduction, this is the 6th one, this is 7th one, this is 8th one. So, how am I doing this?

So, I am just looking from the leaf nodes from leaf reductions from left to right and whichever reduction is possible I am marking them.

So, this is possible, so I marked it; after that you see the only thing that is possible is this one. So, I marked it as 2, after that only thing that is possible is this one. So, though this one is also possible, but you see that or to if you have looking from the left then this comes first. So, the lr parsing, so it will be it will be doing this reduction first because it does a left to right scan. So, it will do this thing then, so that way so if you just do it; so will you will be getting the reductions in this fashion.

Now I have to the corresponding action if you try to look into. So, you can mark the reduction number; the reduction number and the corresponding action fine.

(Refer Slide Time: 03:18)

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

**Attributes for non-terminal E:**

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

**Attributes for non-terminal S:**

- S.code – sequence of three-address statements

**Attributes for terminal symbol id:**

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

**Annotations:**

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

So, at reduction number 1; so, E producing id; so, if you look into the rule. So, it says that E producing id; so this is E dot place equal to id dot plus and E dot code is null. So, we can do that; so E dot plus equal to id dot plus. So, E dot place equal to a and code is null. Similarly at reduction number 2, I will have this E dot place equal to b fine, at then reduction number 3, I will have E dot plus equal to c.

Now this one; so, this now reduction number 4 E minus E. Now, I have to get a new temporary variable. So, if you look into the rule again; so E plus E and E minus E they are same. So, this E dot plus is equal to new temp and then E dot code is concatenation

of E1 code, E2 code and generate E dot place equal to E1 dot plus E2 dot place. So, that way; so it will generate this new code E1 dot; E1 dot place E2 dot place. So, at 4 at reduction number 4, I will have this E dot place equal to t1 and I will generate a code that E dot code; it will have t1 equal to E1 dot place minus E2 dot place that is b minus c.

So, this code will be there; now reduction number 5, at reduction number 5 I will have this E reproducing within bracket E. So, this E dot place equal to t1; E dot plus equal to E1 dot plus. So, that is equal to t1 and E dot code equal to E1 dot code; that is t1 equal to b minus c, so that will be there. Now at reduction number 6, so I will have E producing id. So, this E dot place equal to id dot plus; so that is d.

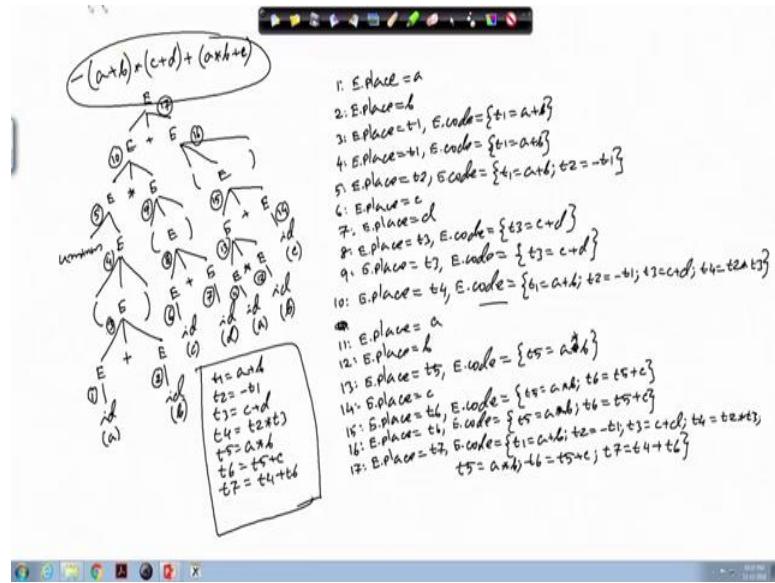
Now reduction number 7, E star E; so it has to get a new temp. So, E dot plus equal to new temp that is t2 and then it will be having some code generator; E dot code equal to E1 code E2 code and this multiplication. So, 5 code; 5 code is t1 equal to b minus c then 6 code, 6 code is null. So, there is nothing and then it will be generating another code that t2 equal to E1 dot plus multiplied by E2 dot plus.

So, E1 dot plus is E1 dot plus from 5; that is t1. So, that is t1 multiplied by E2 place that is d; so, this will have this is reduction number 7. Now reduction number 8; in reduction number 8 it is E plus E; so a new temporary has to be generated. So, E dot plus equal to t3 and then it will have the code like E dot code equal to E1 dot code; E1 dot code is null for reduction 1 it is null, E2 dot code that is 7; 7's code is t 1 equal to b minus c, t2 equal to t1 into d.

And with that we will add another code which is this addition E1 dot plus E2 dot plus. So, E1 dot plus is a E1 dot plus is so sorry, this E this E dot plus that is t3 equal to a plus t2 because this 7 plus 7 plus is t2. So, it will be doing like this. So, this is the final code that is generated; so we have got t 1 equal to b minus c then t2 equal to t1 star d and then t3 equal to a plus t2. So, this is the final piece of code that is generated.

So, you can; so you can generate three address code in this fashion. So, next we will be looking into another example which is again for arithmetic expression, but it is slightly more complex ok.

(Refer Slide Time: 08:27)



So, this is for the expression minus a plus b into c plus d plus a star b plus c; this is the expression now how to do this? So, the first of all I have to draw the parse tree. So, E producing E plus E and then this E should give me this one.

So, that is E star E; now this E should give me this unary minus E, unary minus and this is E, this E should give me within bracket E; this E should give me E plus E that is id plus id. So, this id is a, this id is b fine. Now, this E should give me this c plus d; so, how to get it? So, this is within bracket E bracket close. So, this should give me E plus E; E producing id E producing id; so that is c, this is d ok.

Now, this one; so this will give us within bracket E bracket close and then again this plus has to be generated. So, this is E plus E; so this E should give me E star E, this E should give me id that is a; this E should give me id that is b and then this E should give me id which is c; this is the whole parse stream. Now if I number the reductions, then this is the 1st reduction, this is 2nd, this is 3rd, this is 4th, this is 5th ok.

Now this is 6, this is 7, this is 8, this is 9, this is 10, this is 11, this is 12, then this is 13, this is 14, this is 15, this is 16, this is 17. So, we have got 17 reductions ok; so we can we have marked them like that. Now I have to I have the code generated and the action and the code generation. So, at reduction number 1 ok; so I will have E dot place equal to a; no code is generated and E dot place equal to a.

At reduction number 2; reduction number 2 I will have E dot place equal to b. At reduction number 3, so I need a new temp. So, this E dot place I need a new temporary variable, E dot place equal to t1. And then I have to generate the code that E dot code is t1 equal to a plus b. Then reduction number 4 is same E dot place and E dot code; so they will be copied. So, this E dot place equal to t1 and this E dot code it will be equal to t1 equal to a plus b.

Now, comes reduction number 5. So, reduction number 5 this unary minus; so I to get a new temporary variable for place. So, this E dot place equal to a new temporary t2 and E dot code will be this E1 dot code and then after that the minus of that. So, t1 equal to a plus b followed by t2 equal to minus of t1; so that is a unary minus t1; so that is reduction number 5.

Now reduction number 6; so this will have E producing id. So, E dot place equal to c then reduction number 7 E dot place equal to d. Then comes reduction number 8; so this is are to get a new temporary E dot place equal to t3 and after that E dot code will be this E1 dot code E2 dot code and then this addition. So, E1 dot code is nothing, E2 dot code is; so this for the 6th and 7th; their codes are nothing. So, they only thing; so I have to have this code t3 equal to c plus d. So, t 3 equal to c plus d code will be there.

Now, reduction number 9; at reduction number 9 so, this is within bracket E, so this E dot place equal to t3 and E dot code is equal to t3 equal to c plus d. Now, reduction number 10; at reduction number 10 what we do? It is E star E. So, this I have to get a new temporary. So, this E dot place equal to t4; then this E dot code E dot plus equal to t4.

And then E dot code will be this reduction 5 scored that is t 1 equal to a plus b, t2 equal to minus of t1; after that and then that that is E1. And for E2 it is 9; so 9's code is t3 equal to c plus d; these are put and then I have to have the code of this multiplication that is t2 multiplied by t3. So, t4 equal t2 multiplied by t3; so that is the code for this line number 10.

Now, line number 11, reduction number 11; so that is what is reduction number, reduction number 11 is this one. So, E producing id; so this E dot place equal to a. Now reduction number 12 E dot place equal to b reduction number 13 is E star E; so I have to get a new temporary. E dot place is equal to t5, E dot place equal to t 5.

Then reduction number 13; now I have to have the code part. So, this E dot code will be E ones code. So, E ones code is 13 sorry 13; so this is code for 11; code for 11 is null, code for 12 is also null. So, this is the code that I have is  $t_5$  equal to a plus b. So, that is reduction number sorry a star b ok. Now, reduction number 14; so E producing id. So, this E dot plus equal to id dot plus that is equal to c.

And 15 reduction number 15 is this E plus E. So, I have to get a new temporary E dot plus equal to  $t_6$  and then at I have to have this code generated; so, E dot code. So, code for 13 is  $t_5$  equal to a star b and then I have got code for 12, sorry, code for 14, code for 14 is nothing that is null. So, I have to now generate code for  $t_6$  equal to 13 and 14 they are plus; so, that is  $t_5$  plus c.

Now, reduction number 16. So this E dot place equal to  $t_6$  and E dot code equal to this thing that  $t_5$  equal to a star b,  $t_6$  equal to  $t_5$  plus c ok. Now, come 17; so 17 it will have a new temporary. So, this E dot place equal to  $t_7$  and this E dot code; E dot code it will have this E1 code that is 10 code parts. So, that is this one with  $t_1$  equal to a plus b,  $t_2$  equal to minus of  $t_1$ ,  $t_3$  equal to c plus d,  $t_4$  equal to  $t_2$  star  $t_3$ .

So, that is 10's code part; now 16, 16's code part is  $t_5$  equal to a star b,  $t_5$  equal to a star b,  $t_6$  equal to  $t_5$  plus c and after that I will have this 17 for that E plus E; so I will get this code generated that another line  $t_7$  equal to this reduction 10's E dot place; that is  $t_4$  plus reduction 16's E dot plus that is  $t_6$ .

So, this is the code that is finally, generated;  $t_1$  equal to a plus b and  $t_2$  equal to this thing. So, like that so you can generate the corresponding codes. So the final code that I have is this one that  $t_1$  equal to a plus b,  $t_2$  equal to minus of  $t_1$ ,  $t_3$  equal to c plus d,  $t_4$  equal to  $t_2$  into  $t_3$ , then  $t_5$  equal to a star b,  $t_6$  equal to  $t_5$  plus c and  $t_7$  equal to  $t_4$  plus  $t_6$ ; so this is the code that is generated. And you see that this is perfectly fine with the high level statement that we have here. So, it is perfectly fine with that and it is generating a proper code for that.

Now, the way we are writing the code writing thus this action; so here the rule that we have is making us to write like this, that I have to have this gen part and this code so, it was it was using this gen part. So, they are actually kept the codes are kept in a list sort of thing. So, as I was telling that at many places, we can keep the code in a list or we can use a you can write it on to a file directly. So, if you look into some later rules that we

have discussed in this; in this course like say array and all; so there we will be using this function emit.

(Refer Slide Time: 22:48)

```
S → L := E
{ if L.Offset = null then
    emit(L.place ':=' E.place);
else
    emit(L.place '[' L.Offset ']' ':=' E.place)
}

E → E1 + E2
{ E.place := newtemp();
emit(E.place ':=' E1.place '+' E2.place)
}

E → (E1)
{ E.place := E1.place }
```

So instead of gen, so we will use the function emit. The difference being that when it is gen; so it is generating the code and when you say emit as if it is writing on to some output file. And that way when it is emit, so it is just appending the code that is generated at this point with the already generated code. Whereas with the gen part, so with every at every non terminal the code part. So, that is holding the full code for that particular non terminal at that point. Like the example that we had taken, so example that we had taken here there we have seen that these gen parts. So, it was used for this E dot place and this thing.

So, this generating for code for arithmetic expression; so you are doing it like this. But here it is a bit combustion sometimes like as we have seen that the same thing we are writing again and again, but that is because of the fact that it is on a list. So, you cannot output half of the list for a particular folder. So, it will not have the other parts with which will it join ok. So, if it can be; so you can always write it in terms of emit functions, but we have to be a bit more careful.

Because as we have seen that in case of Boolean expression; so Boolean expression you can write you using this emit code and you can correct it putting them on a linked list and all, but for this gen code sort of things. So, this gen code sort of things it is in a list,

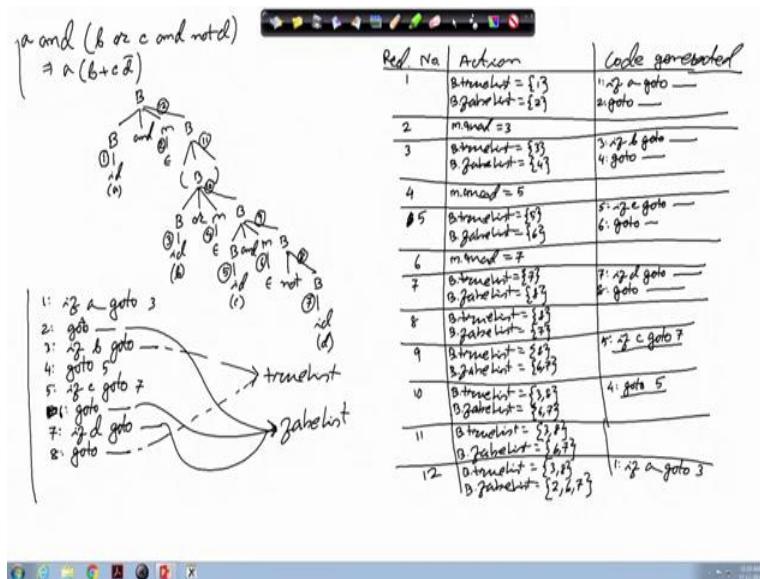
but in case of emit sort of things. So, it is you can put it on to a file directly. So, you do not have that much of overhead in doing the realization of those codes ok. So, in our next class we will be taking some more examples of this arrays and this complex programming language constructs, Boolean expressions etcetera and we will do some exercise to see how the code can be generated for those complex cases.

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

**Lecture - 60**  
**Intermediate Code Generation (Contd.)**

Today we shall look into conversion of Boolean expression into 3 address code and apart from that we will take some more exercises for practice.

(Refer Slide Time: 00:27)



So, the example that we take for Boolean expression is like this. So,  $a$  and  $b$  or  $c$  and not of  $d$ , that is basically  $a, b$  or  $c, d$  bar. So, corresponding to this Boolean expression, how can we generate the code that we will see. So, the grammar if we first thing is to draw the parse tree. So, the since we have got and at the top.

So, this will be  $b$  and  $m B$  then this  $B$  will give me  $id$  which is  $a$  then this  $m$  will give me epsilon and then this  $B$  should give me within bracket  $b$ . Now the next this  $B$  will give me the or. So,  $B$  or  $m, B$ . Now this  $B$  will give  $id$  which is  $B$  this  $m$  will give epsilon now this  $B$  will give me the and function. So, it is  $b$  and  $m, B$ . This  $B$  will give  $id$ , which is  $c$ . This  $m$  will give epsilon, this  $B$  will give not of  $B$ . This  $B$  will give not of  $B$  and this  $B$  will give me  $id$  which is  $d$ .

So, this is the parse tree. So, if I number the reduction. So, this reduction will be done first then this one. Then it will do this one. Then this one, then this, then this. Then it will do this reduction, then it will do this one. Then it will do this reduction and reductio0,n then it will do the or reduction. So, this is number 10 and this is number 11.

Finally, this is number 12. Now you have to consult the semantic action for this Boolean expression and accordingly we have to generate the corresponding code. So, if I make some sort a table like reduction number in the corresponding action, and the code that is generated, code generated. Do it like this then at reduction number 1. So, if we look into the Boolean expression rule grammar then it says B first one is B1 and MB2. So, sorry, reduction number, one is a reduction number, one is B producing id.

So, for b producing id the corresponding rule is this one. So, here it is not written, but in some other slide it will be there , but for b producing id. So, it generates a code which is if a, if id goto this goto will be filled up at number 2 it will generate goto which will be filled up. And here it will produce this true list and false list. So, B dot true list is equal to 1. And B dot false list equal to 2, ok.

So, this will be done at reduction number 1. At reduction number 2, at reduction number 2 that is m producing epsilon the only action is m dot quad, m dot quad is equal to next quad which is 3. That is done at reduction number 2. At reduction number 3 again we have got this b producing id. So, 2 lines of code will be generated, if we b goto to be filled up later and this goto to be filled up later. And action is that b dot true list is equal to 3 and b dot false list is equal to 4.

Now, reduction number 4 is m producing epsilon. So, it will do m dot quad equal to next quad that is equal to 5. Now reduction number 6, reduction number say, sorry, reduction number 5. So, reduction number 5 is again m producing id. So, it will generate 2 lines of code if c goto unknown, this part will be filled up later and this is the else part. So, this is an accordingly it will generate true list and false list. B dot true list equal to 5. And b dot false list is equal to 6. So, this is at reduction number 5. At reduction number 6 is m producing epsilon.

So, m dot quad m dot quad equal to next quad that is 7. So, this is done at reduction number 7 again B producing id. So, it will generate lines of code if d goto and goto. Now at reduction number 7, this true list and false list will be generated. B dot true list equal

to 7 and B dot false list equal to 8. Now reduction number 8 not of b. So, here the false list and true list they will get interchanged. So, B dot true list will be equal to 8, B dot true list equal to 8 and B dot false list equal to 7. At reduction 9, so, B and MB2.

So, it will first of all. So, the and rule if we see B. And so, this is the B and. So, it first says back patch b 1 true list with m dot quad. So, that back patching will be done if m dot b dot true list. So, B dot true list is 5 f b dot true list is a this 5. So, 5 true list. So, that is a truly you have got 5 here. So, this location will get corrected now. So, that will be filled up with this m dot quad. So, m dot quad for this m producing epsilon 9 line number 6. So, it is 7.

So, this fifth location will get corrected. So, it will have a code correction, where this fifth line will be modified like if c goto 7. So, this correction is done. This a correction and then it will be having this B dot true list, B dot true list will be made equal to what ever true list we have with a from the second B. So, this 8 true list that is equal to 8. And B dot false list, this will be merger of this B1 false list and B2 false list that is 5 and 8 false list.

So, 5 false list is 6 and this is 7. So, 6 and 7 will get merged. So, you will get false list as this one. So, this is reduction number 9. At reduction number 10. So, this B or and for B or again there is a back patching. So, for B or. So, it is back patch B 1 false list with m quad. So, B 1 false list. So, this is reduction 3 a is reduction 3 reduction 3 false list is 4. So, that will be corrected. So, the location 4 will now get corrected with go 2 that code will be modified. So, it will become goto 5 because this 4 if this m producing epsilon is a 4. So, corresponding m dot quad is 5. So, this 5 is filled up. So, this is again another back patching that is done and then this true list and false list.

So, B dot true list will be merger of the true list. So, B dot true list will be merger of this of 3 and this 9. So, this 2 true list will be merged. So, 3 true list is 3 and 9 true list is 8. So, 3 and 8 will get merged. So, create the true list. And B dot false list will be B 2 dot false list that is 6 and 7. So, this is done at reduction number 10. At reduction number 11 is within bracket B.

So, this true list and false list will get copied. So, B dot true list equal to 3, 8 and B dot false list equal to 6, 7. So, they are copied and at reduction number 12. So, this is again. And so, first of all there will be a back patching and then it will do something. So, and

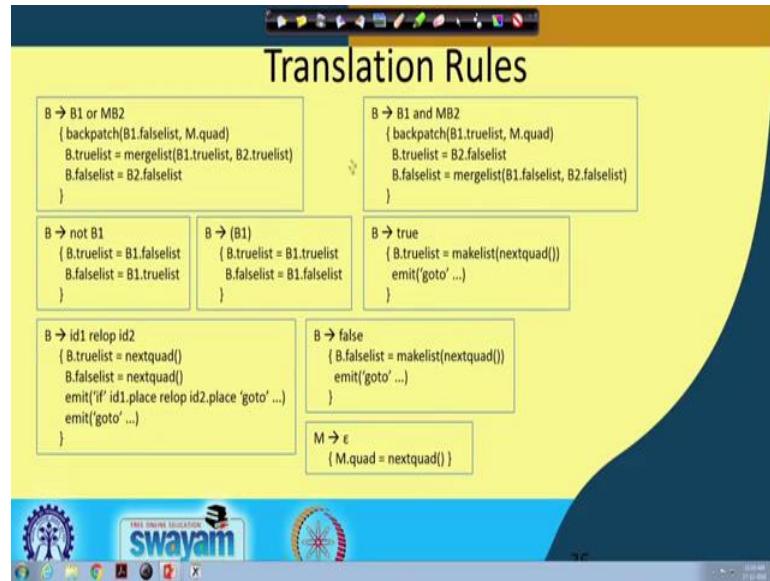
so, it will back patch B 1 true list with m quad. So, B 1 true list. So, B 1 true list it is 1 true list is 1. So, it will back patched with this 3, m quad is 3. So, this back patching will be done that is line number one will get corrected.

So, it will be collected like if a goto 3. And then B dot truly stand B dot false list will be created. B dot true list will be B2 dot true list that is 3 and 8 and B dot false list will be merger of 2 false list. So, this is the merger of this falls list of one and falls list of 11. So, false list of one is 2 and false list of 11 is 6 and 7. So, they are merged. So, this is the this is the this completes the code generation part. So, you can just write down the final code that is generated. So, the overall code that is generated is like this. At a offset, at index 1.

So, we have got if a goto 3 then 2 we have goto and this is not yet filled up , at 3 we have if b goto not yet filled up, at 4 it is goto 5, ok, 4 it is goto 5, at 5 it is if c goto 7. At 8 we have, sorry at 6 we have goto at 7 will have if d goto. At 8 we have goto. And then if you consult this final true list and false list so, this one and 3 and 8. So, 3 and 8. So, they will constitute the final true final true list. So, this is 3 and 8 will give me the true list. And this 2, 6, 7 they give me the false list is 2, 6 and 7 they together will give the false list for the overall Boolean expression fine.

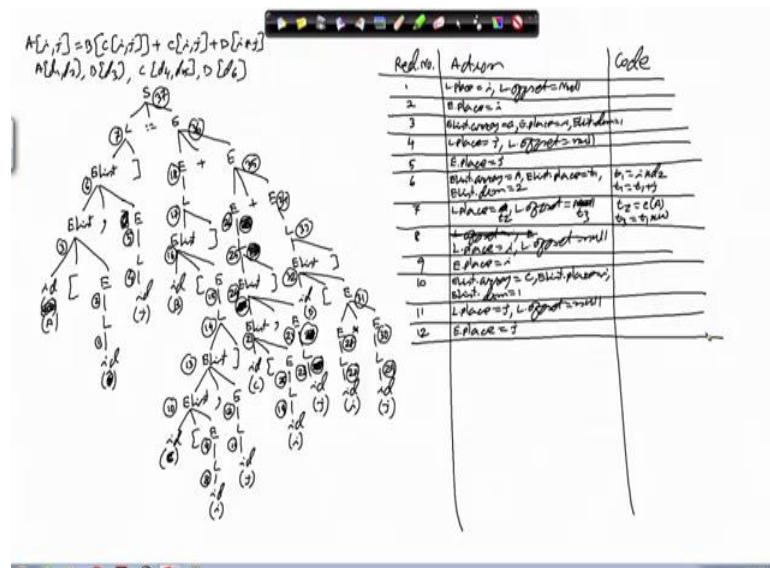
So, you see that. So, you know if you try to match. So, if a then you need to evaluate the second part. So, if a goto 3 now it is taking here if b. So, a and b both are true. So, it goes to the true list. Otherwise if a is false then it definitely false. So, it goes to the false list fine. Similarly, if b is false it comes to 5, and now it check c, if c is true then it goes to 7 here it checks if d then it goes to false list. So, if not of d, it will come to line number 8. So, that will goto the true list. So, in this way this piece of code is correct for the original Boolean expression that we had and this 3 address code generation process. So, it has successfully been able to generate the corresponding code. So, next we will look into an array conversion.

(Refer Slide Time: 15:23)



So, this for the array conversion it is a slightly tricky in the sense that. So, it is very conversion is slightly tricky because the number of rules that we have is more. So, will try to do it. So, let us see how this can be done.

(Refer Slide Time: 15:48)



So, let us consider an array difference like this. Say  $A[i,j]$  equal to  $B$  of  $C[i,j]$  plus  $C[i,j]$  plus  $D$  into  $j$ . Suppose this is the overall Boolean expression and this array has,  $A$  has got dimension  $d_1, d_2$ .  $B$  is of  $B$  is a single dimensional array. So,  $B$  is of dimension  $d_3$ .  $C$  is a 2 dimensional array. So, that is  $d_4, d_5$  and again  $D$  is a one dimensional array. So,

that is say d6. Now the step if you try to generate the corresponding parse tree. So, this is be s producing this is an assignment statement. So, we have got this left side assigned as some expression. Now this left side I have to generate this  $A[i,j]$  for generating this  $A[i,j]$ . So, this is E list bracket close. Then this E list generates E list , E , E list , e. So, this E list generates id bracket start E. This E generates L, L generates id. So, this id is your i this id is j sorry, sorry, this id is a this name of the array A. So, this id is I this id is i. Now this E produces L produces id. So, this is the j part.

Now, for the right hand side, I have got an expression. So, it is E plus E. So, from the first you have to generate this of. So,, So, then this E can give me thus left hand side. So, E producing L,L producing E list bracket close. Then this E list produces id bracket start and E this id is B. Now this E gives L. And this L will give that E list bracket close. So, this E E L will give me E list bracket close. This E list will give E list comma E. And then this E list will give id bracket start E this id is your c.

This id is that C array C. Now this E it will give me L which will intern give id that is your i. And this E will give L that will give id which is j, fine. So, this plus part we have done. Now thus that C i j plus d star i j. So, therefore, that part again I have to have this a E plus E. And from the first you have to generate this C i j. So, for this I have to go like L, L producing E list bracket close E list giving E list comma E. Then this E list giving id bracket start E, this id is c. Now this E gives me L, gives me id which is your i and this E will give L will give id which is j, fine.

Now, this plan E. So, this will again E, E producing L, L producing E list bracket close. Then this E list will give id bracket start E. So, this id is d now this E gives, E star E, this will give E star E. So, this will give E producing L producing id. So, that is your I and this E will give L producing id that is your j. So, this is the full parts tree that will be produced. Now if I want to number it.

So, first reduction is this one. Then this, 1, 2, 3, ok. Then it will do a reduction of this one, then this one, 4, 5 then this reduction 6, then this reduction 7. So, up to this much will be done then. It will be coming to this L producing id. So, that will be 8, now this is 9. So, this is 10. Now this is a 11, this is 12, this is 13, this is 14, this is 15, this is 16, this is 17, 18, now it will be doing this one, 19 then 20, 21, 22, 23, 24. Then it will come down here 25, ok. So, it will be. So, 25, 26 oh sorry there is a. So, this is not 22. So, this

will be a 22, this will be 22. So, this will be 23. Ok after that this will be 24 yes. So, this is 24, this is 25, this is 26.

Now, this L producing id. So, this will be 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37. So, this will be the whole parse tree with a with reduction numbers written. Now if we try to write down the code generated and the reduction numbers corresponding to that. So, it will be something like this. So, this is the reduction number. So, this is action. And the code that is generated. ok code generated. Now at reduction 1 we have this one, L producing id. So, accordingly it will have this L dot place equal to i. And L dot offset equal to null, this will be null.

And no code is generated. At reduction 2, at reduction 2 it will have this E producing L. So, E dot place equal to L dot place. So, E dot place equal to i. Still no code generated at reduction number 3 it will. So, it reduction number 3. So, there is an array. So, with the E list. So, it will associate E list dot array as A, then E list dot place as i and E list dot dimension equal to 1, dimension equal to 1. So, this is reduction number 3. At reduction number 4 it is doing this L producing id. So, again same thing L dot plus equal to j and L dot offset equal to null, at reduction number 5 it will place that E dot place.

So, at reduction number 5 we have this one E producing l. So, E dot place will be equal to L dot place, that is j that is reduction number 5. At reduction number 6. So, it will have E list dot array at reduction number 6. It will have E list dot array equal to A, then E list dot place, E list dot place will be a new temporary t1 and E list dot d dimension will be equal to 2. And it will generate a code that t1 equal to i into d2 and t1 equal to t1 plus j. So, this 2 lines of code if you consult the corresponding array rules then these 2 lines of code will be generated.

Now, reduction number 7 it will do L dot place equal to i. And L dot offset equal to null. And it will have this t2 sorry a at reduction number 7. So, L dot place will be equal to a new temporary t2 and L dot offset will be equal to another new temporary t3. And then t2 is equal to the constant part of a and t3 equal to t1 multiplied by width. So, we assume that width of all elements are same all arrays. So, then at 8 it will have my a reduction number 8. So, L producing id.

So, L dot offset equal to i and L dot. So, L dot place equal to i and L dot offset equal to null. At reduction number 9, will have this E dot place equal to L dot place. So, this E

dot place is equal to I, ok. So, that way it will do up to even read reduction number 9. At reduction number 10, at reduction number 10, it will have this that is this one. So, E list etcetera. So, this E list dot array will be equal to c because is the c array.

Then E list dot place equal to i, E list dot place equal to i and E list dot dimension will be equal to 1. So, no code is generated here reduction number 11. So, L dot place equal to j. And L dot offset equal to null. At reduction number 12, it will do E dot place. So, at reduction number 12 is this one E dot place equal to L dot place. So, that is equal to j. So, this way it will continue. So, will continue with this example in the next class.

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

**Lecture - 61**  
**Intermediate Code Generation (Contd.)**

(Refer Slide Time: 00:17)



So, at reduction number 13, we will have the next action. So, reduction number 13. So, it will have this E list dot array E list dot array equal to C then E list dot place equal to t3 and E list dot place equal to t3. And this E list dot dimension equal to 2. And then it also generate some code, the t3 equal to i into d5, then t3 equal to t3 plus j ok. So, that is 13 at 14 a reaction 14. So, it will do that a at reduction 14 is this one E list bracket close.

So, it will do L dot place equal to a new temporary t4. And L dot offset equal to another new temporary t5. So, t4 is equal to the constant part of c and t5 is equal to t3 into w. So, this two lines of code will be generated at reduction 15. So, it will have this E dot place equal to t6, ok. And it will generate the code t6 equal to t4 t5. This is a one dimensional array axis t6 equal t4 t5. Then at reduction 16 so, it will have the action that E list dot array; E list dot array equal to B and then E list dot place equal to t6, E list dot place equal to t6 and E list dot dimension equal to 1, ok.

Then at reduction 17, it will have L dot place equal to t7 another new temporary and L dot offset equal to t8. So, another new temporary, where t7 is the constant part of B and

$t_8$  is equal to  $t_6$  into  $w$ . And so, that is 17 at 18 at reduction 18 so, E producing L. So, it will generate a new, it will take a new temporary and E dot place equal to  $t_9$ , where  $t_9$  is equal to  $t_7 t_8 t_9$  equal to  $t_7 t_8$ .

So, that is 18. Then 19 at 19 so, you will have L producing i d. So, this L dot place equal to id dot place that is j and L dot offset equal to null; L dot offset equal to null. So, this is the action at 19, at 20 so, it will have this E list dot array equal to C, E list dot place equal to  $t_{10}$ , a new temporary. And E list dot dimension, sorry, at 20, sorry, at 20 it is a E producing L sorry, so, this is not correct. So, E producing L so, E dot place equal to i; so, E dot place equal to i.

Then at 21, at reduction number 21 we have this one this E list bracket close E list comma E. So, that part 21 we have E list dot array equal to C, E list dot place equal to I, E list dot place equal to i, because this id is C and this E dot a dot place is already i. So, that is E list dot place is i and E list dot dimension equal to 1; E list dot dimension equal to 1. Then at reduction number 22; at reduction number 22 so, it will do L reduction number 22 is this one, L producing id. So, will have L dot place equal to id dot place that is j and L dot offset equal to null.

Then comes reduction number 23 at reduction number 23. So, we have got this E producing L. So, this E dot place equal to L dot place so, L dot place is j. So, E dot place is equal to j. And no code is generated, at reduction number 24, at reduction number 24 we have this one, E list, E, so, E list, E. So, for that E list dot array has to be copied. So, E list dot array is same as 21's E list dot array.

So, E list dot array equal to C. Now E list dot place is a new temp. E list dot place is a new temporary  $t_{10}$  and E list dot dimension, E list, E list dot dimension will be equal to 2. And the code that will be generated is  $t_{10}$  equal to i into d5 and then  $t_{10}$  equal to  $t_{10}$  plus j. So, this is the code generated at 24.

At 25 so, E list L producing E list bracket close. So, it will do L dot place, L dot place equal to E list L dot place equal to  $t_{11}$  a new temporary. And L dot offset is another new temporary  $t_{12}$ . And it will generate some code. So, it will generate code like  $t_{11}$  equal to the constant part of the array C of c and  $t_{12}$  equal to  $t_{10}$  into w. It will generate this code at 25. Now at 26 I have E producing L. So, this E dot place; so, this E dot place will be equal to L dot place. So, L dot place is  $t_{11}$ . So, E dot place equal to, sorry, E it will be at,

E dot place will be a new temporary t13, this an array. So, t13 is equal to the array axis t11 t12 the array axis.

Now, reduction number 27, at reduction, 27 we have this one L producing id. So, this L dot place equal to I, L dot offset equal to null. Now comes reduction number 28. At reduction 28, we have this E producing l. So, this E dot place equal to i. So, that is reduction number 28. At reduction number 29 we have this, at reduction number 29 we have L producing id. So, L dot place equal to id dot place that is j and L dot offset equal to null. Now at reduction number 30, E producing L.

So, this E dot place equal to L dot place. So, E dot place equal to j. Now reduction number 31; at reduction number 31 we have, so, this E producing E star E. So, for that it has to generate some code that is a new temporary has to be generated and E dot place will be a new temporary t14, where the code generated will be t14 equal to i into j, ok. Multiplication of this E, E1 dot place and E2 dot place.

So, 28 and 30 so, 28 is i and 30 is j. So, their product is taken as 30. Now reduction number 32. So, in reduction number 32 we have got this rule E list producing id open parenthesis E. So, for that we have this L dot place; L dot place equal to, sorry, a 32 is E list something. So, this E list dot array has to be created. So, E list dot array equal to this new array d an E list dot place equal to t14 and this E list dot dimension equal to 1. So, that is 32.

Now, at 33 so, it will 33 is this one, L producing E list bracket close. So, E list dot array equal to, E list dot array equal to d, then E list dot place equal to t14. And E list dot dimension equal to; E list dot dimension equal to 1, sorry, this is 32 is reduction, 33 is this one L producing E list bracket close. So, it will be L dot something. So, 33 L dot place is equal to a new temporary t15 and this L dot offset is equal to another new temporary t16.

Where t15 code is equal to the constant part of the d array and t16 code is equal to t14 into w. Now comes reduction number 34 at reduction number 34 we will do that E list dot place, sorry, 34 is E producing L. So, E dot place equal to a new temporary t17. Where this new temporary t17 is equal to t 15 t 16 is an array axis, so, t15 t16.

Now, reduction number 35; now reduction number 35. So, this is E plus E. So, 35 it will be doing E, E dot place equal to a new temporary t18. And then it will generate code the t18 equal to t13 plus t17. That is 35 at 36, so, it is again E plus E. So now, it will generate another new temporary E dot place equal to a new temporary, t19, where t19 is equal to t9 plus t18. That is 36 and 37, so, it is L, this assignment statement. So, there is no action only that id dot place. So, that is L dot place of reduction 7. That is t2 t3. L dot place, L dot offset that is assigned as t19.

So, in this way the final code that is generated will be final code that will, that is generated. So, you can just club all these parts to get the final code. So, next we will be looking into another example, where we will try to generate the code for some statement of some language.

(Refer Slide Time: 16:50)



So, we will take this example- while a greater than b, do begin, if x equal to y, then c equal to d, c equal to a plus b, else d equal to a minus b, b equal to q plus r, end, then x equal to y plus z. So, we will draw the parse tree and then you will be you should be able to follow the code generation portion that we have seen previously and then we will see the final code. So, that will be like this. So, it is L M S because their multiple statements. So, we have to do it like this- then the L gives me S and this L give this S gives me that while loop. So, while M the Boolean condition B, then another M and S, then do, then another M then S.

Now, this M gives me epsilon. So, this B should give me the Boolean condition, that id bellow id. So, this id is A this (Refer Time: 18:30) is greater than, so, this is your B. Then this b gives us epsilon and then this S should give me this begin end part, so, this is begin L end. And then this L again there are 2 statements the if statement. And then this p equal to cube plus L statement. So, it should be L M S like that this L should give me S. And this S should give me the if then else statement. So, it is if B then M S N else S, sorry, L M then S. So, this is if then else statement. Now this B will give me that id (Refer Time: 19:32) id that is your x equal to y.

This M should give me epsilon, ok. Then this S should give me this c equal to a plus b, this assignment statement. So, this S give me A which is id equal to E. This id is c and this E is E plus E this E gives me id which is a, this E gives me id which is b. Now this n will give me epsilon this, m will give me epsilon now this S. So, this s should give me the else this else part. So, that is d equal to a minus b. So, this S should give me A which is id equal to E.

So, this id is d and this should give me E minus E, this E is id, this E is also id, so, this is a and this is b. So, this is the d equal to a minus b portion. Now this m gives me epsilon now this S will give me the next statement p equal to q plus, r ok. So, this p equal to q plus r, so, it will have S producing A, S produce A producing id equal to E and this E giving me E producing E plus E. So, this is id, this is id, so, this id is your c. Sorry, this id is your p, this is q and this is r. This is p equal to q plus r.

So, we are done with this begin end part. So, n gives me epsilon, then this S gives me, sorry, this S gives me the assignment statement A and this A gives me id equal to E. So, this id is x and then this is E plus E, this E gives id which is y and this E gives id which is z. So, this is the whole parse tree.

Now, if we think in terms of reductions then this is the reduction number 1, then this is reduction number 2, this is reduction number 3. Now after that we have got reduction number 4 as this one. Then this is reduction number 5, then we have got this one as reduction number 6, reduction number 7, 8, 9, 10, 11, 12. Now this one 13, 14, 15, 16, 17, 18, 19, 20 21, 22, then this is 23, this is 24, ok. So, this is 25. Oh, in between some numbers we have missed.

So, this is 12; so, once this if the end part is over that is 17. So, 18 onwards we did mistake 18, 19,s, these are wrong. 18 17 is this reduction. So, then this reduction should be 18. This should be 19, this should be 20. After that this should be 21, 22, 23, 24, 25, 26, 27, then this will be 28, this is 29, 30. Now this is 31, this E producing id, this should be 31, this is 32, this is 33, this is 34, this is 35 and this is 36, fine.

So, if you use this parse tree and follow the grammar rules that we have for this for this high level language statements. So, it will generate a code and the final code that will be generated will look something like this. So, at offset 1 so, it will generate code if A greater than b goto 3. At 2, it will generate goto 13, at 3 it will generate if x equal to y, goto 5. At 4 it will generate goto 8. At 5 it will generate t1 equal to a plus b, 6 it will generate c equal to t1, 7 goto 10.

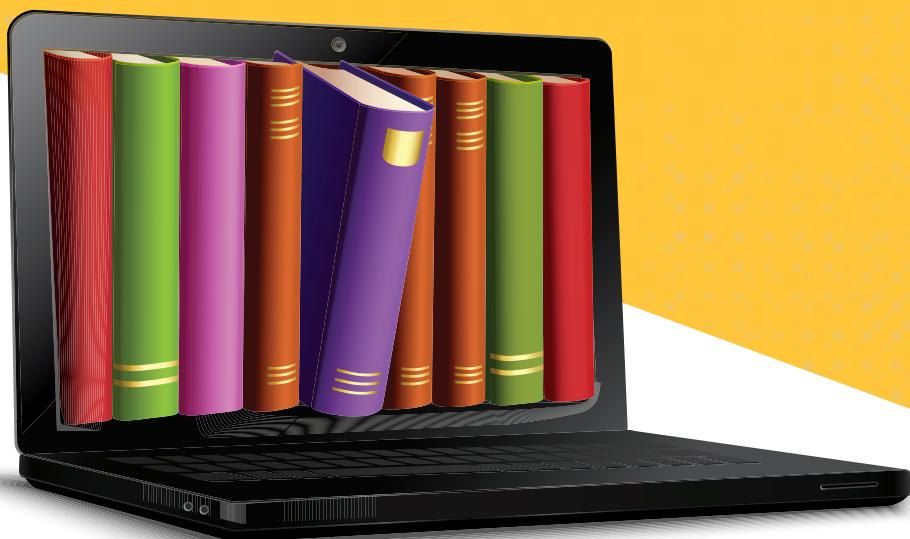
At 8, t2 equal to a minus b, at 9, d equal to t2, at 1,0 t3 equal to q plus r, at 11, p equal to t3. At 12, generate goto 1, then at 13 it will generator t4 equal to y plus z, 14. It will generate, x equal to t 4. So, you have to follow the rules that we have this for different production rules that we have for the grammar. And in fact, all this rules that we have, so, all these rules for assignment statement sementic action. So, this is for array, so, for other statement. So, we can do that. So, that way if we follow those rules then you will get this piece of code and if you check this piece of code c.

So, the if a greater than b, goto 3. So, it comes to c. So, this is the body part and other if a is not greater than 3, then it will come to goto 13. At 13 it is outside the loop. So, it is computing y plus z and then x equal to t4. So, x equal to y plus z, so, this computation is being done. So, if comes to 3. So, it is a body. So, it checks if x equal to y goto 5. So, it move 5 it computes at a plus b into t1. And then it is a, t1 is assigned to c then it goes it says goto 10 because it is keeping over the else part.

So, it comes to the goto 10. So, t equal to t3 equal to q plus r then p equal to t3 then it is goto 1. So, it is restarting the while loop. And if x is not get a not equal to y then it will goto 8. So, here it will do this a minus b then d equal to t2. So, d equal to a minus b and then it comes to the then comes out of if then else statement. So, p equal to q plus r. So, that is done and then again goto 1. So, it will be branching to this, so, this way. So, this piece of code faithfully replicates the code that we have for the while loop. And in this

way we can translate any set of statements following the rules that we have discussed in our class for generating the 3 address code.

**THIS BOOK IS  
NOT FOR SALE  
NOR COMMERCIAL USE**



(044) 2257 5905/08



nptel.ac.in



swayam.gov.in