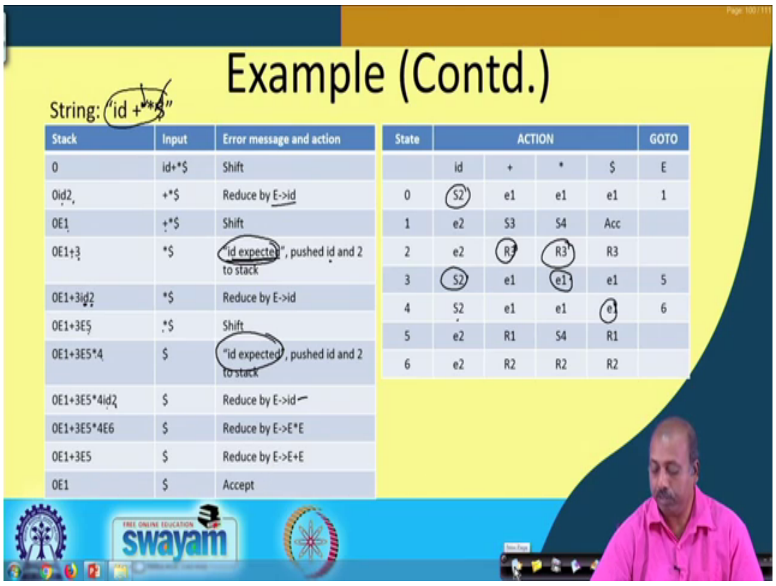
Compiler Design
Prof. Santanu Chattopadhyay
Department of E &EC Engineering
Indian Institute of Technology, Kharagpur

Lecture – 32
Parser (Contd.)

Next, we will look into in 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
0	id+*$	Shift
0id2	+*$	Reduce by E->id
0E1	+*$	Shift
0E1+3	*$	"id expected", pushed id and 2 to stack
0E1+3id2	*$	Reduce by E->id
0E1+3E5	*$	Shift
0E1+3E5*4	$	"id expected", pushed id and 2 to stack
0E1+3E5*4id2	$	Reduce by E->id
0E1+3E5*4E6	$	Reduce by E->E*E
0E1+3E5	$	Reduce by E->E+E
0E1	$	Accept

State	ACTION				GOTO
	id	+	*	$	E
0	S2	e1	e1	e1	1
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	

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.

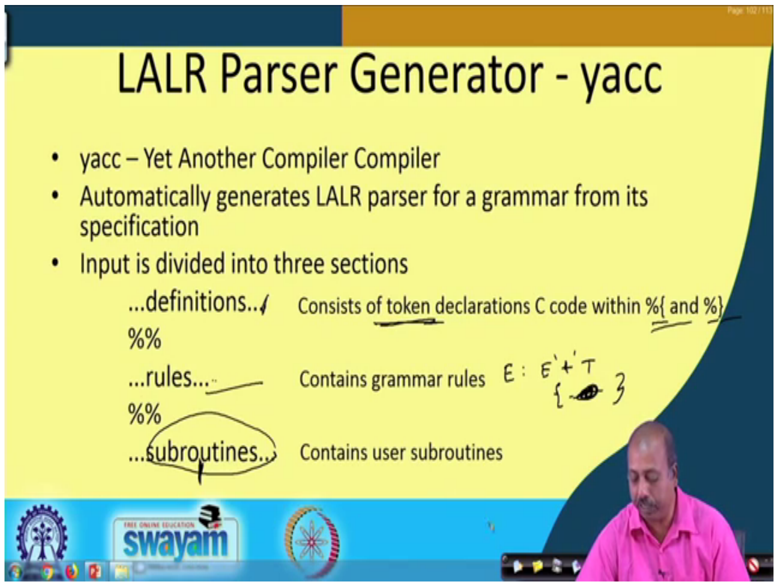
Now 2 star, so 2 star it say it says that it reduced by rule number 3. So, rule number 3 will be applied and accordingly this reduction will take place and this will be the new configuration of the stack. Now this 5 star, so 5 star is shift by shift 4. So, it will shift star into the stack and 4 into 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 accept.

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 Compiler. So, compiler 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 it is 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 reduce conflict or not.

So, the first thing that of a that is often done is that given when 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 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.

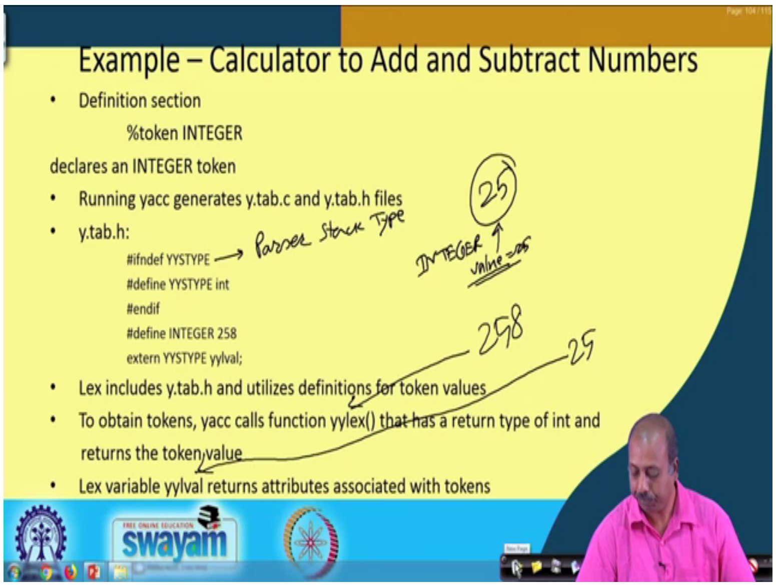
So, it generates LALR parser. So, the so, you should be very happy with that because it can give us LALR parser so easily. Now, just like Lex, this tool also has got the input specification file in some particular format. So, we have got the three portions in it the definitions part, the rules parts 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 this 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 say 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.

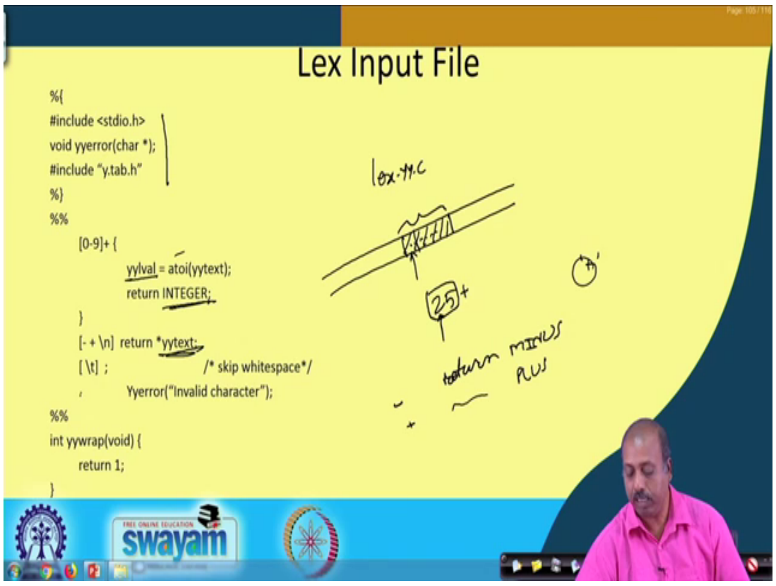
So, if we do that, so you can define this YYSTYPE that stack type to be consisting of this attributes also as a part of the token or as a part of the non terminals. So, if it is not defined in that case, it will define it to be integer. If it is defined if the user has defined YYSTYPE, then it will take 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)



Lex Input File

```
%{
#include <stdio.h>
void yyerror(char *);
#include "y.tab.h"
%}
%%
    [0-9]+ {
            yylval = atoi(yytext);
            return INTEGER;
    }
    [- + \n] return *yytext;
    [ \t] ;                 /* skip whitespace*/
    .       Yyerror("Invalid character");
%%
int yywrap(void) {
    return 1;
}
```

Next, we will see, so, 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 is constituting the next token.

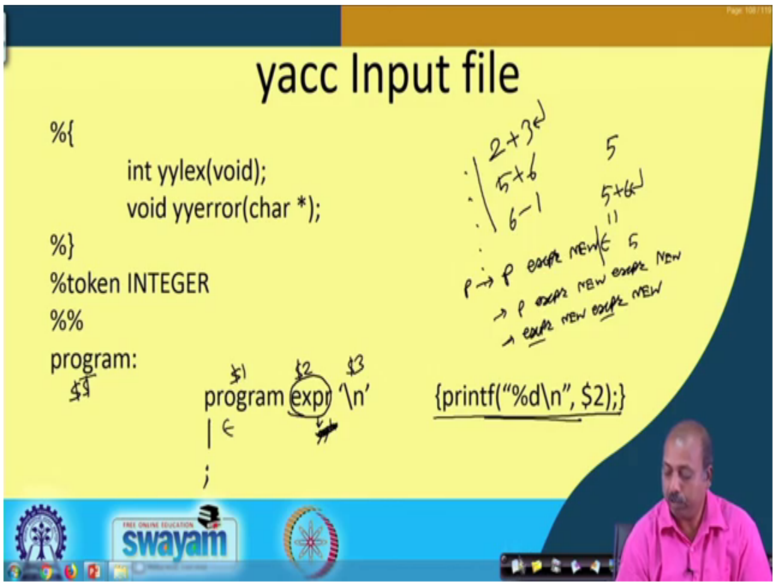
So, this yytext will be equal to this substring ok. So, this y, so this is for example, if I have written like 25 plus and at the input pointer was somewhere here. So, it will take this 25. So, this part as the yytext and then see this is a text value or the character value. So, or a string value, so 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)



yacc Input file

```
%{
        int yylex(void);
        void yyerror(char *);
%}
%token INTEGER
%%
program:
                program expr '\n'          {printf("%d\n", $2);}
                | ϵ
                ;
```

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.

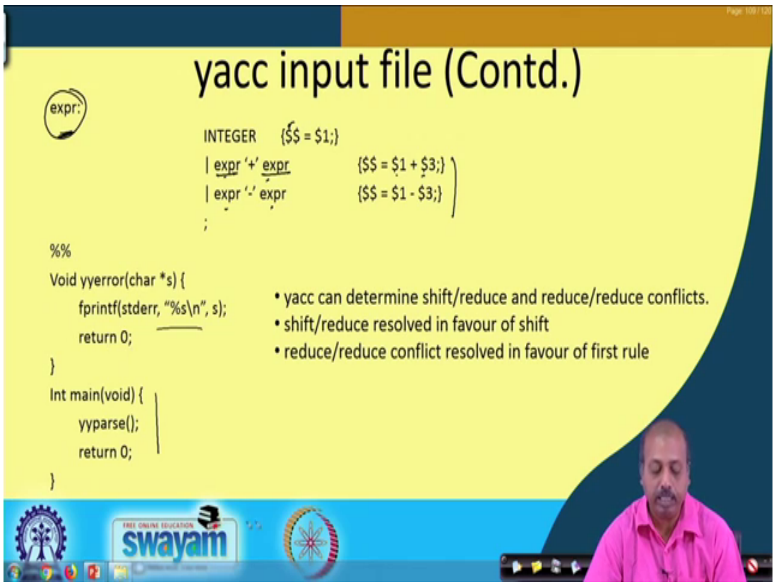
Now, program is defined as program expression newline. So, we can have multiple lines of expressions like. So, it is you can have like 2 plus 3, 5 plus 6, 6 minus 1, etcetera etcetera then it is expected that since it is the calculator. So, as you write 2 plus 3 and then, plus newline, so it should print 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 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 typ so that will have this value dollar 2. So, we will see how are you 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)



yacc input file (Contd.)

```
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;
}
```

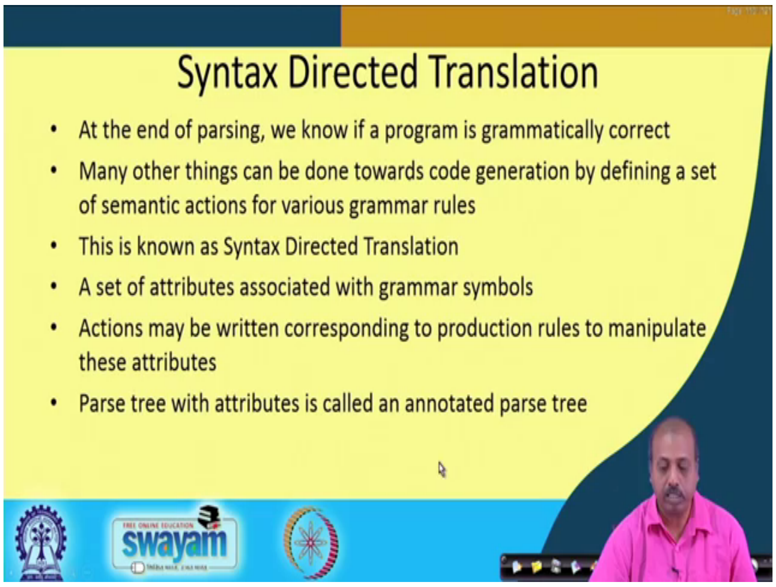
- yacc can determine shift/reduce and reduce/reduce conflicts.
- shift/reduce resolved in favour of shift
- reduce/reduce conflict resolved in favour of first rule

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 this tools for writing good compilers.

(Refer Slide Time: 27:07)



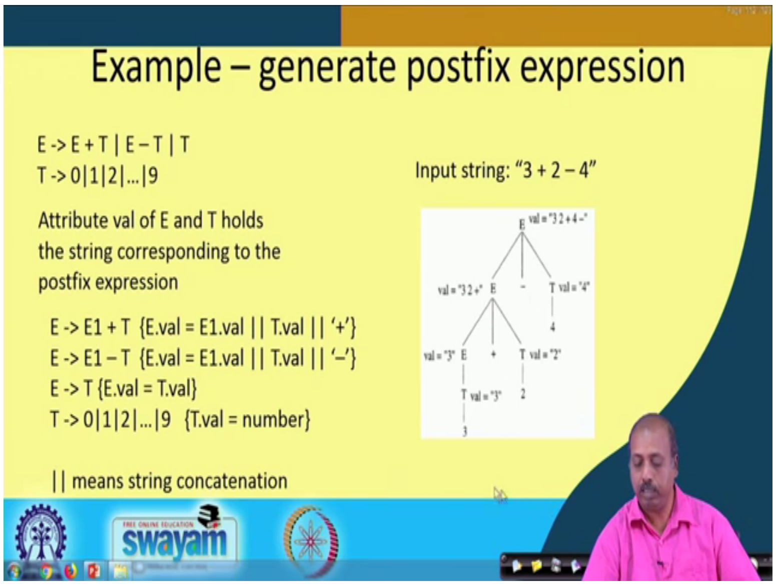
Syntax Directed Translation

- 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 -> E + T | E – T | T
T -> 0|1|2|...|9

Attribute val of E and T holds the string corresponding to the postfix expression

E -> E1 + T {E.val = E1.val || T.val || '+'}
E -> E1 – T {E.val = E1.val || T.val || '–'}
E -> T {E.val = T.val}
T -> 0|1|2|...|9 {T.val = number}

|| means string concatenation

Input string: "3 + 2 – 4"

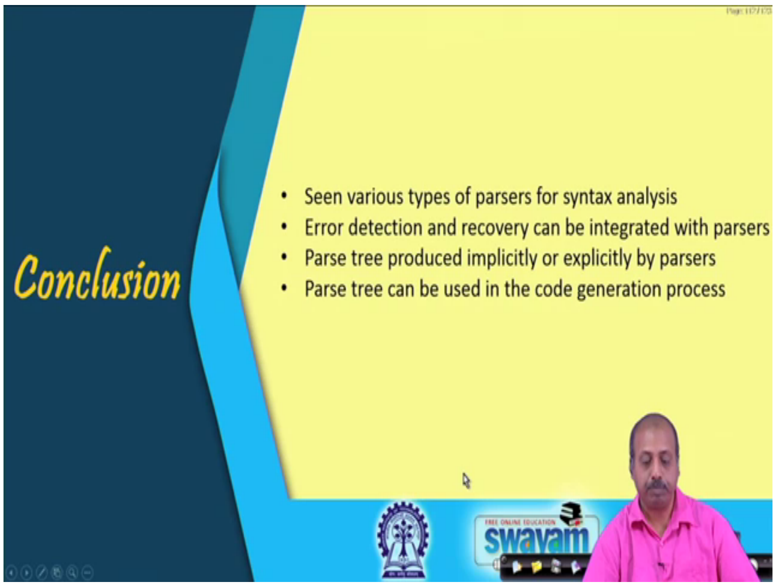
Is say this one, suppose we have got a grammar like this. So, E producing E plus T or E minus T or T and T producing all these digits 0, 1 to 9; so, we have got attribute val for E and T that will hold the string corresponding to the postfix expression ok. Now, so, we want to convert the a string to 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 may 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)


Conclusion

- Seen various types of parsers for syntax analysis
- Error detection and recovery can be integrated with parsers
- Parse tree produced implicitly or explicitly by parsers
- Parse tree can be used in the code generation process

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.