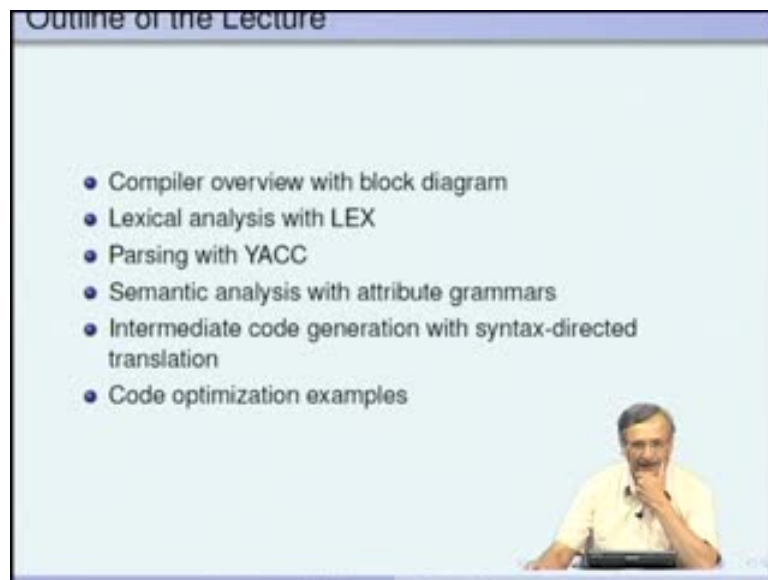


Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Module No. # 01
Lecture No. # 01
An Overview of a Compiler

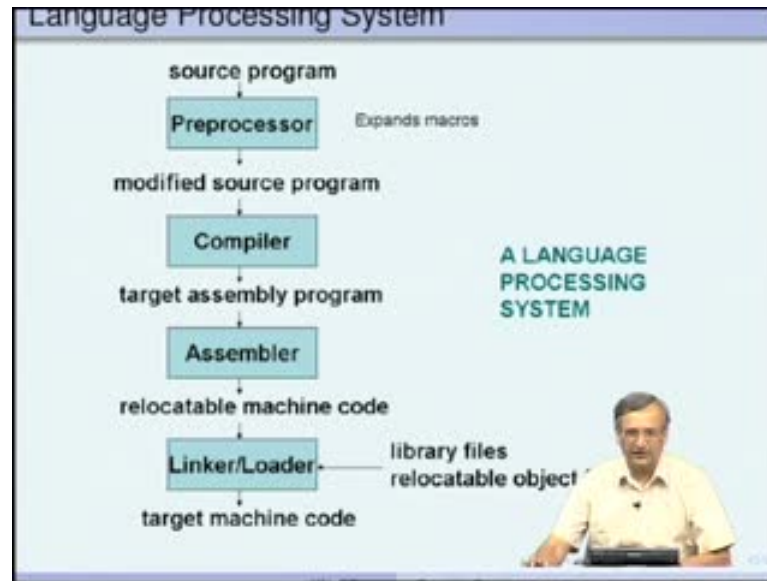
This is a lecture about an Overview of a Compiler.

(Refer Slide Time: 00:30)



In this lecture, we will go through a block diagram of a compiler and look at the various phases of a compiler. For example, there is a lexical analysis phase, there is a parsing phase, there is semantic analysis, there is intermediate code generation, code optimization, and also machine code generation. We look at each phase with some example, minor description of what is happening, and so on and so forth.

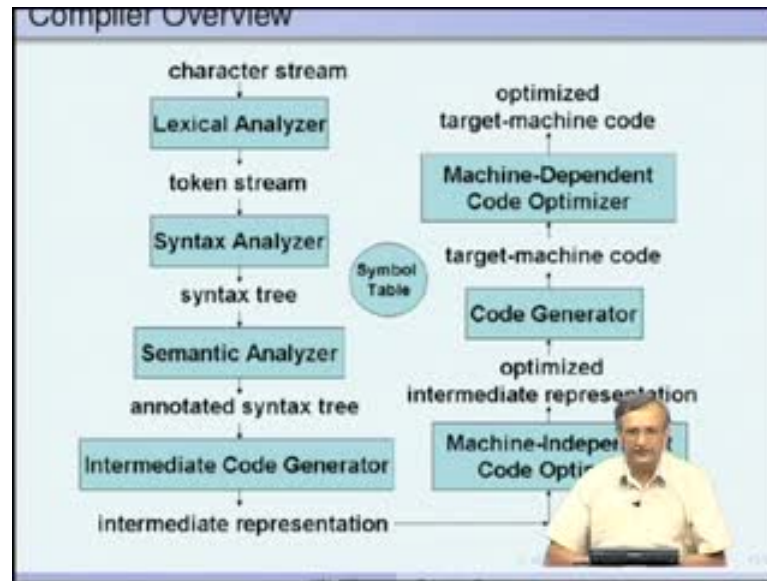
(Refer Slide Time: 00:54)



Let us begin with a block diagram of what is known as a language processing system. In a language processing system, the source program, for example, goes through a preprocessor. For example, in a C program, you have a number of macros such as hash define, hash include and so on and so forth. These are passed through the preprocessor. The preprocessor expands these macros and takes appropriate action at the compiler level itself and there is no code generated for such macros. Such modified source program is then fed to a compiler. The compiler generates machine code. The machine code could be in the form of an assembly code or it could be directly binary of the machine, etcetera.

In the case **that** it is an assembly language program, it is fed to an assembler, which converts it into the machine language of target machine, and then such modules are produced probably individually. Therefore, a linker or loader is needed in order to combine such modules. Finally, the loader gives out a complete relocatable **...** All these combined together will be the task image of the machine. This can be run on a machine.

(Refer Slide Time: 02:28)



For example, if you look at the compiler itself, which is a block in the previous diagram, it has these phases. This is our main stay for the entire lecture. There is a lexical analyzer, there is a syntax analyzer, semantic analyzer, intermediate code generator, machine independent code optimizer, code generator and then machine dependent code optimizer. These are the phases of a complete compiler. You can say –in some way, these are parts of the machine; the machine itself being the compiler.

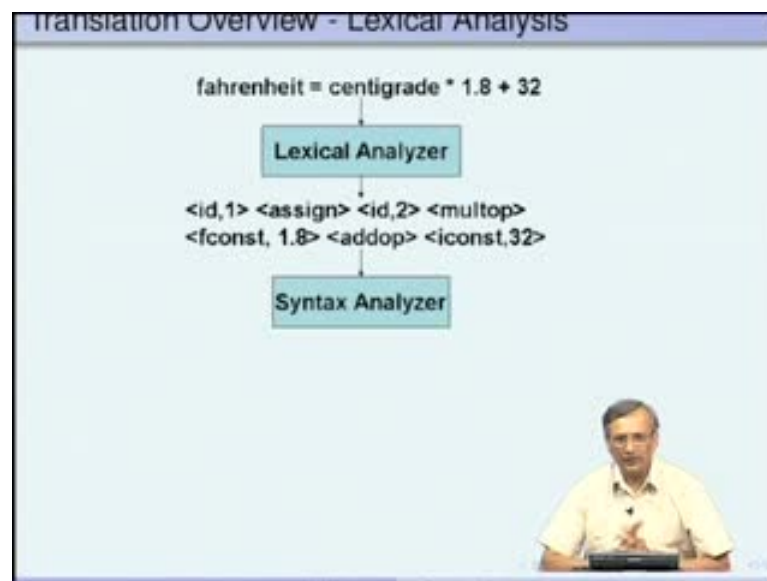
What does each one of these phases in a compiler **rule**? That is what we are going to be looking at in the rest of the lecture. For example, the lexical analyzer – it takes the program, which is given in the form of characters. The characters are all read from a file and then the lexical analyzer divides it into what is known as a token stream. Why are these needed? We will see very shortly. The token stream is then fed to a syntax analyzer, which checks whether the syntax of the program according to the programming language rules are all satisfied.

If they are satisfied, it produces a syntax tree; otherwise, it gives number of error saying that look there is no **...** Then, corresponding to the 'if then' statement, there is no assignment symbol in an assignment statement, the plus is missing in an expression, and so on and so forth. The syntax tree itself is not the end. The syntax tree for example, does not tell us that the program is valid.

To give you an instance, if we are trying to assign some value to an entire array; this is not permitted in C. With such **construct**, the syntax analyzer will not be able to point out the error and says this is wrong. So, what does the syntax analyzer do in such a case? The syntax analyzer in such a case says – sorry I cannot help, I will pass on this information to the semantic analyzer and that takes care of it. The semantic analyzer checks such mistakes, and then if everything is right, it produces what is known as annotated syntax tree.

The annotations are nothing but the semantic information of the program such as what are the symbol names, what are the various constant values, and so on and so forth. These are all fed to the intermediate code generator, which produces an intermediate representation. Why this is required, etcetera will be seen in the rest of the lecture. Then, the intermediate representation itself is improved by the machine independent code optimizer. There are many chances for such improvement as we shall see. Then, the optimized intermediate representation is fed to the code generator. The code generator actually is specific to every machine and then it converts the intermediate representation into machine code. Finally, there is some more improvement possible on the machine code itself; that is done by the machine dependent code optimizer. Finally, we get a very compact optimized target machine code.

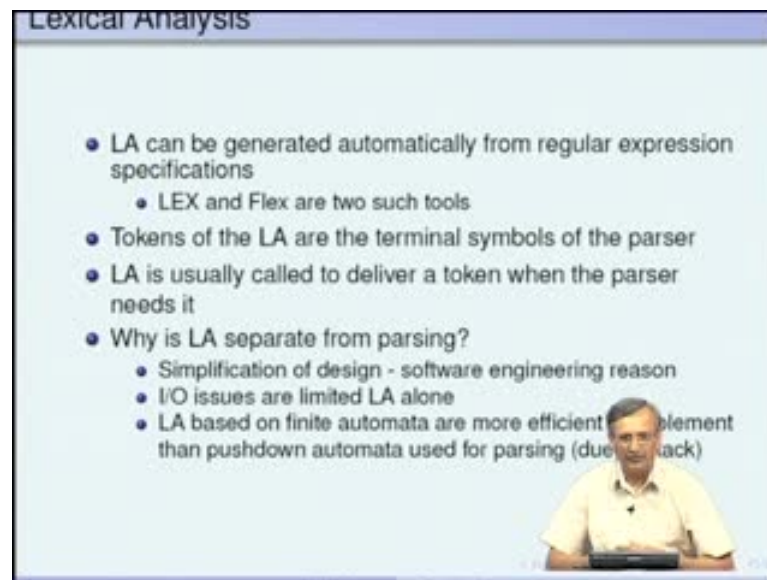
(Refer Slide Time: 06:10)



Now, let us look at each one these phases in some detail. Here is an example of what lexical analysis **task** is. Let us take a very simple assignment statement – fahrenheit equal to centigrade into 1.8 plus 32. There are characters in this particular assignment statement F, A, H, R, E, N, etcetera. There are symbols such as equal to, star, plus and so on. There are numbers such as **1.2** and 32.

The lexical analyzer converts such stream of characters into slightly more meaningful; what are known as tokens. For example, fahrenheit and centigrade are names. They are traditionally called as identifiers. What exactly is the identifier? That value will be stored in a symbol table. id,1 and id,2 will denote the two identifiers: fahrenheit and centigrade; 1 and 2 being the indices of the symbol table in which the names are stored. Similarly, the equal to itself is called as an assignment operator and it given a token as assign. Similarly, the multop and the addop; the constants are given the tokens fconst and iconst with the appropriate values. What exactly are these tokens? Inside the compiler, the token itself is represented very compactly as an integer. Because of this, the space required for storing the entire assignment statement will be very small compared to the storage, which is required by the character stream.

(Refer Slide Time: 08:09)



Lexical Analysis

- LA can be generated automatically from regular expression specifications
 - LEX and Flex are two such tools
- Tokens of the LA are the terminal symbols of the parser
- LA is usually called to deliver a token when the parser needs it
- Why is LA separate from parsing?
 - Simplification of design - software engineering reason
 - I/O issues are limited LA alone
 - LA based on finite automata are more efficient than pushdown automata used for parsing (due to stack)

© 2011 Pearson Education, Inc. All rights reserved.

Lexical analysis is very cumbersome if you try to write it by hand. Lexical analyzers are typically generated automatically from regular expression specifications. So, there are two tools, which are very well known for this purpose: one of them is called LEX, which

is available on every unix machines and Flex is the counter part of LEX, which is available from **GNU**. The tokens of the lexical analyzer will actually become the terminal symbols of the context-free grammar, which is passed by the parser. We will see this a little later. Lexical analysis is usually called as a function to deliver a token whenever the parser needs. It is not as if the entire stream of characters is converted to a stream of tokens and then the parser starts its work. It is actually called only when it is required to deliver a token.

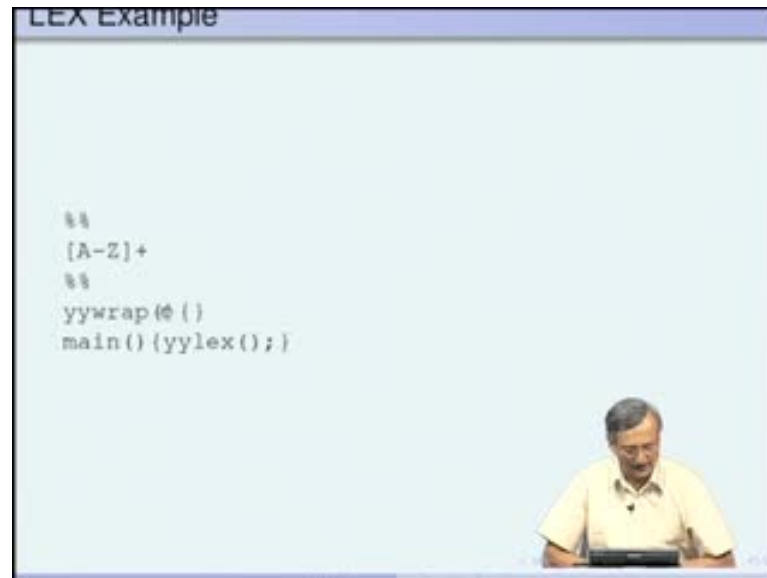
Why is the lexical analysis separate from parsing? Theoretically speaking, there is no reason to separate lexical analysis from parsing. In fact, as we shall see very soon, the lexical analyzers are specified using regular expressions. So, regular expressions can in fact be written as regular grammars, which are nothing but a special form of context-free grammars. Therefore, a parser, typically its specification for example, can be written inclusive of the lexical analyzer itself, but there are reasons why we make the lexical analyzer separate.

First of the reasons is the simplification of design. This is a software engineering decision. The compiler is a huge piece of software. Therefore, making the entire software into modules, actually enables good software engineering practices. One of them is to make the lexical analyzer separate. Similarly, the parser and so on and so forth. Then, the input output issues are all limited to lexical analysis alone. For example, (Refer Slide Time: 10:31) this is one of the modules of a compiler, which does intensive I/O. The program is in a file and it is in the form of characters. So, each character has to be read and then fed to the parser. So, the lexical analyzer might as well do this entire part and it is possible to design the software very efficiently for such I/O. There is no need to bother the rest of the compiler once the input output is taken care of by the lexical analyzer. Lexical analysis is based on finite state machines; finite automata as they are called. Such finite automata are far more efficient to implement than pushdown automata, which is used for parsing. Why? It is well known that to parse a context free language sentence, it is necessary to have a pushdown automaton and the pushdown automaton uses a stack.

If we actually convert the entire lexical analyzer specification into a context-free grammar, then there would be a huge number of pushes and pops corresponding to the character stream of the source program. So, this is very inefficient. We might as well do the pushes and pops on larger pieces of the source program, which are logical entities

called tokens. So, this makes the parser much more efficient. These are some of the reasons why lexical analysis is separated from **parse**.

(Refer Slide Time: 12:15)

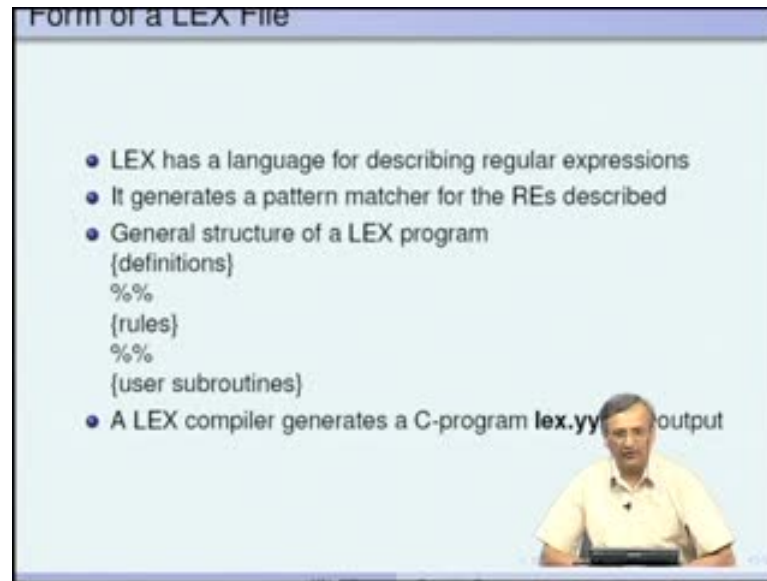


```
LEX Example

%%
[A-Z]+
%%
yywrap@()
main(){yylex();}
```

Let us now look at LEX in order to understand how it does lexical analysis. This is a very trivial simple example of a LEX program. What it does is – it simple recognizes capital letters A to Z in the whole program. Now, let us look at some of the details of such LEX programs. Whatever we have written here is called **...** A to Z plus is called a rule and what we have written below it, the yywrap, the main, etcetera are the program segments.

(Refer Slide Time: 13:02)



The slide, titled "Form of a LEX File", contains the following content:

- LEX has a language for describing regular expressions
- It generates a pattern matcher for the REs described
- General structure of a LEX program
 - {definitions}
 - %%
 - {rules}
 - %%
 - {user subroutines}
- A LEX compiler generates a C-program `lex.yy.c` as its output

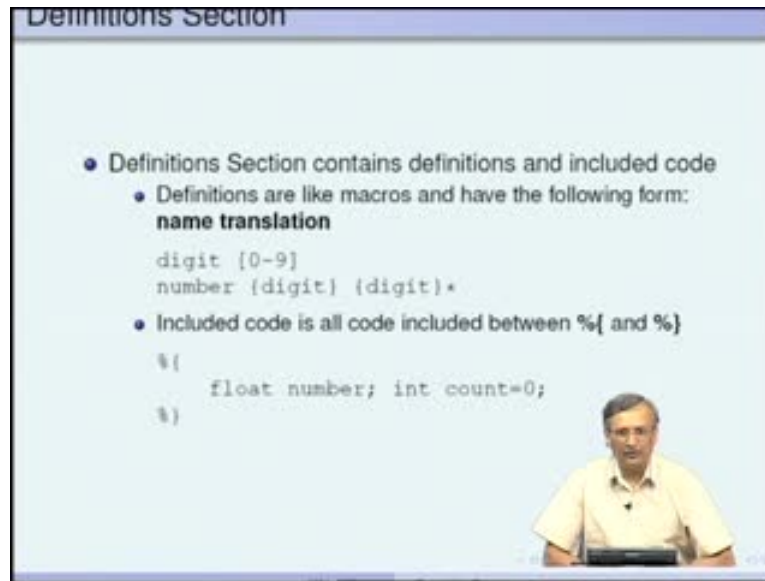
A small inset image of a man in a light-colored shirt is visible in the bottom right corner of the slide.

Let us look at some details of LEX. What is the form of a LEX file? a LEX specification; LEX has a language for describing regular expressions. I mentioned this already. It generates a pattern matcher for the regular expressions described. How does it do it? It actually converts each of these regular expressions to finite state machines and then arranges this in the form of a pattern matcher. We will look at some of these details very soon.

The general structure of a LEX program is simple. There are some definitions, which are nothing but short hand and then we have a marker in the form of two percent symbols. Then, we have the rules, the patterns, or the regular expressions. Finally, there is another marker in the form of two percent symbols, and then we have some of the user subroutines, which are supplied by the user. A LEX compiler generates a C program called a LEX dot y y dot c as its output. This can be either used by a parser or used on its own.

We will look at two examples, where it is used as a function by the parser and another example, where it is used as a standalone program.

(Refer Slide Time: 14:26)



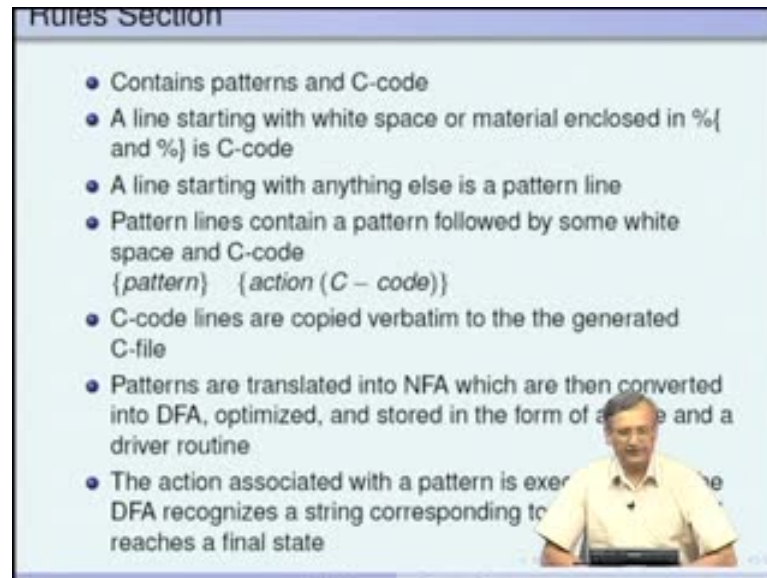
Definitions Section

- Definitions Section contains definitions and included code
 - Definitions are like macros and have the following form:
name translation
`digit [0-9]`
`number {digit} {digit}*`
 - Included code is all code included between `%{` and `%}`
`%{`
`float number; int count=0;`
`%}`

Now, the Definitions Section. The Definitions Section contains definitions and it also includes code that is supplied by the user. For example, the definitions are like macros. They are not regular expression on their own, but they are used in writing regular expression specifications. For example, `digit`; 0 to 9 and what is a number? Number is a single digit followed by 0 or more digits. Digit star implies 0 or more digits as it is usual in the regular expression notation.

Included code is all code, which is included between the two markers: percent flower bracket and percent flower bracket. For example, `float number int count 0` is a piece of code, which is supplied by the user for some initialization purposes.

(Refer Slide Time: 15:28)



Rules Section

- Contains patterns and C-code
- A line starting with white space or material enclosed in %{ and %} is C-code
- A line starting with anything else is a pattern line
- Pattern lines contain a pattern followed by some white space and C-code
`{pattern} {action (C – code)}`
- C-code lines are copied verbatim to the the generated C-file
- Patterns are translated into NFA which are then converted into DFA, optimized, and stored in the form of a table and a driver routine
- The action associated with a pattern is executed when the DFA recognizes a string corresponding to that pattern and reaches a final state

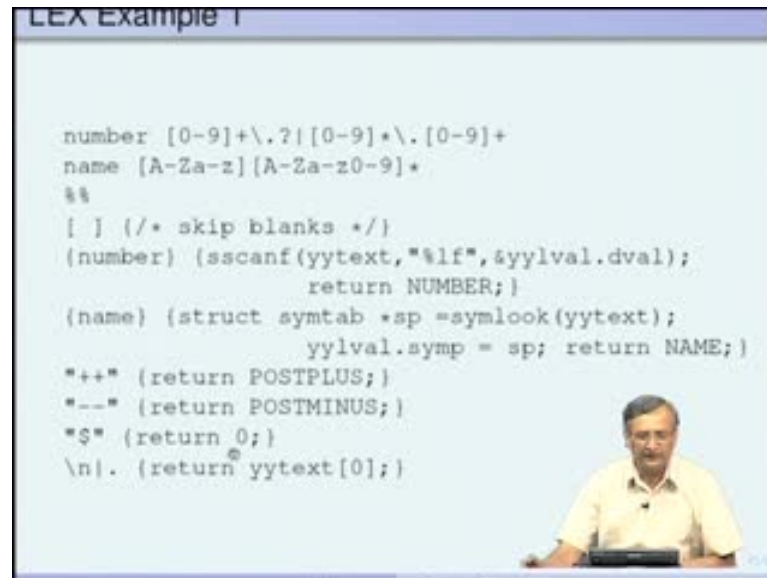
What does the Rules Section have? This is the heart of a LEX specification. It contains patterns and it contains C-code. A line starting with white space or material, which is enclosed in the markers: percent flower bracket and percent flower bracket; is all C-code. This is very similar to the C-code, which is included in the Definitions Section as well. Then, anything else; if a line begins with anything else, it is a pattern.

Pattern lines contain a pattern and then some C-code. Pattern, action in C code. We will see examples of this; that is how it is written. The C code lines are not processed by the LEX compiler at all. There are just taken and copied to the output. The patterns are regular expressions. So, they are translated into Nondeterministic Finite Automata, NFA. These are then converted to Deterministic Finite Automata because it is difficult to implement NFA's. These DFA's can be optimized. Why? There is a very famous theorem, which says – all forms of the same DFA can be converted to the minimal form. The state minimization theorem enables this and these DFA's are all stored in the form of a table and a driver routine.

The action associated with the pattern is executed when the DFA recognizes a string corresponding to that pattern and then reaches final state. This is very similar to what happens in a finite state machine – you start off from the initial state, then you feed characters through the various states, then the transition happens, each transition leads

you to a new state, and finally, you reach a final state. When you reach a final state, you have really recognized that particular pattern and then you can actually do some action.

(Refer Slide Time: 17:52)



```
LEX Example 1

number [0-9]+\.\?[0-9]*\.[0-9]+
name [A-Za-z][A-Za-z0-9]*
%%
[ ] /* skip blanks */
(number) {sscanf(yytext, "%lf", &yyval.dval);
          return NUMBER;}
(name) {struct symtab *sp = symlook(yytext);
        yyival.symp = sp; return NAME;}
"++" (return POSTPLUS;)
"--" (return POSTMINUS;)
"$" (return 0;)
\\n|. (return yytext[0];)
```

Here is a nice simple example of a LEX program, which is used as part of an expression parser later on.

There are actually many of these tokens, which are recognized by this particular LEX specification. There is a Definition Section, which shows two definitions: There is number definition and then there is a name definition. The number definition simply defines a number; the first part is 0 dash 9 plus back slash dot question mark; looks difficult, but it trivially says – the number is any number of digits followed by a dot which is an option. Then, there is a bar. The bar says – either this or that. So, number could also be 0 to 9 star followed by dot and then followed by 0 to 9 plus. In other words, the meaning of that is any number of digits followed by a dot followed by any number of digits again. Only thing is, it makes compulsory to have a digit after the dot, if the dot is present. Similarly, the name itself could be A to Z capitals a to z small followed by any of the letter A to Z small a to z, or any of the letter and numeral 0 to 9 any number of times; that is the star. So, we are really looking at letter followed by letter or the digit star.

Once the regular expression number is recognized, it has a small action following it, which is the scanf. What does an scanf do? The scanf really takes the text of that

particular token; that is, the numerals, which make that number, but it is still not in binary form. So, it reads it into a variable called `yylval`. `yylval` is a variable, which is generated by LEX. It is already known to us and it is understood by LEX as well.

This `yylval` is principally used to transfer the values from the lexical analyzer to the parser. For example, here we are reading a number and later, in the parser, for expressions, we will use the value of this particular number. How do we communicate the value from the lexical analyzer to the parser? That is through the variable `yylval`. The return NUMBER says – the token, which is generated is number and that is returned by this particular function; piece of action code. Whatever is written in capitals, for example, number, name, postplus and postminus; these are actually tokens. As we will see very soon, these tokens are really defined in the parser and the lexical analyzer is supposed to recognize and pass them to the parser.

The second one, the name; there is little more processing, which is done here. Once name is recognized, the symbol table is looked up; `symlook` actually with `yytext`. `yytext` is the text of the name itself; the characters corresponding to the name. `symlook` is the symbol table routine. It looks up the symbol table routine and then if it is already present, it actually gives you a pointer to it. If it is not present, it will insert the name into the symbol table and spread on its pointer. What is `yylval`? In this case, (Refer Slide Time: 22:12) `yylval` is actually the pointer value itself. `yylval dot symp` is nothing but the pointer value, which is a pointer into the symbol table for that particular name. What is the token? Token is the integer code name, which is returned by this action code. Similarly, for double plus, it returns `POSTPLUS`; for double minus, it returns `POSTMINUS`, for the end of file dollar, it returns a 0, and any other character including new line, it simply returns the character itself.

(Refer Slide Time: 22:47)

```
LEX Example 2

%{
FILE =declfile;
%}

blanks [ \t]*
letter [a-z]
digit [0-9]
id ((letter)|_)((letter)|(digit)|_)*
number (d|digit)+
arraydeclpart (id)*["{number}"]
declpart ((arraydeclpart)|{id})
decllist ((declpart){blanks}*, "{blanks
        (blanks){declpart}{bl
declaration (("int")|("float"))(blar
        {decllist}{blanks)
```

Here is a second example, which is slightly more complicated. The previous example was called as a function by the parser, whereas this is a program on its own. What does this particular program do? I will skip to the next slide and then get back to this in a minute.

(Refer Slide Time: 23:07)

```
LEX Example (contd.)

%%
(declaration) fprintf(declfile,"%s\n",yytext);
%%

yywrap(){
fclose(declfile);
}

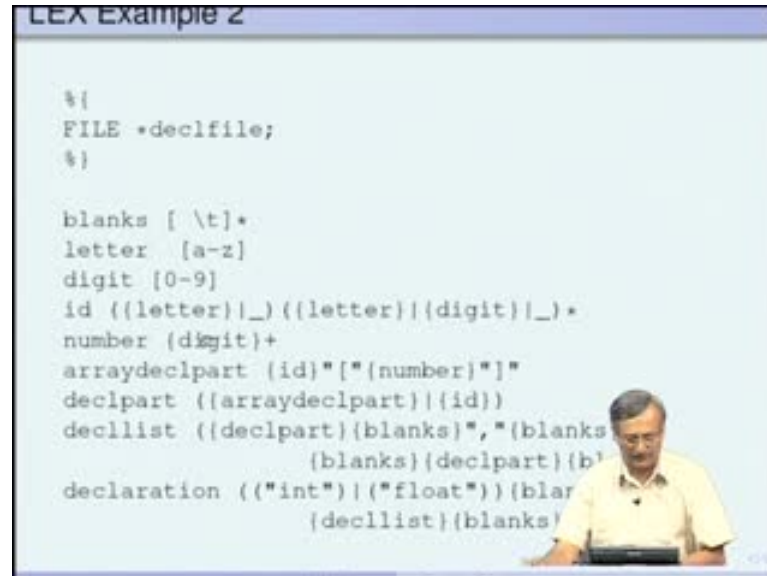
main(){
e
declfile = fopen("declfile","w");
yylex();
}

Examples of declarations:
int a, b[10], c, d[25];
float k[20], l[10], m, n;
```

Look at the bottom of this slide. It gives you examples of C declarations: int a comma b 10 c comma d 25; float k 20 l 10 m comma n. Now, it may be clear. It actually

recognizes such declarations. So, you must understand that not everything in a LEX specification needs to be **...** Let me take that back.

(Refer Slide Time: 23:31)



```
LEX Example 2

%{
FILE *declfile;
%}

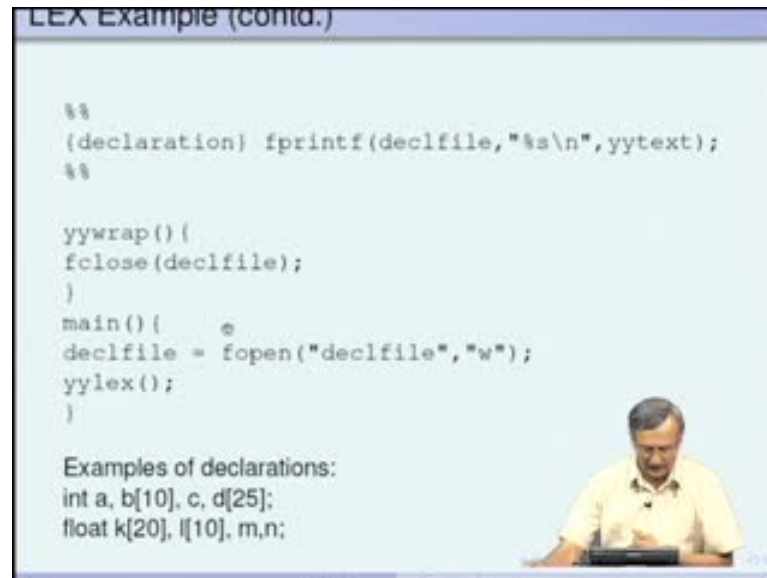
blanks [ \t]*
letter [a-z]
digit [0-9]
id ((letter)|_)((letter)|(digit)|_)*
number (digit)+
arraydeclpart (id)*["{number}"]
declpart ((arraydeclpart)|(id))
decllist ((declpart){blanks}*, "{blanks
        {blanks}{declpart}{blanks}
declaration (("int")|("float"))(blanks
        {decllist}{blanks}
```

Whatever we described in the syntax of a programming language as a context-free grammar, is not necessarily always context free, it can even be specified in the form of regular expressions. That is what I want to show here.

For example, we have blanks, which are nothing but a blank or tab any number of times, a letter, digit, then identifier, number; these are all the usual tokens that are recognized by a LEX specification. Here comes the next one. You are still in the definitions part. Array declaration part is actually an identifier followed by right bracket followed by a number, which is nothing but the number of dimensions of the array followed by the right square bracket; whereas, a declaration part is array declaration part or just a simple name.

A declaration list is a list of such declaration parts and a complete declaration says – integer or float followed by blanks followed by a declaration list followed by blanks again.

(Refer Slide Time: 24:55)



```
LEX Example (contd.)

%%
(declaration) fprintf(declfile, "%s\n", yytext);
%%

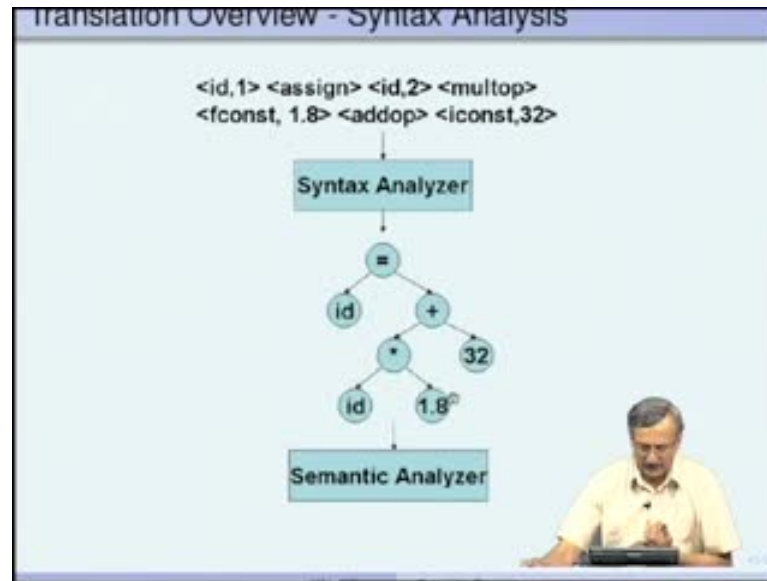
yywrap() {
fclose(declfile);
}
main() {
    e
    declfile = fopen("declfile", "w");
    yylex();
}

Examples of declarations:
int a, b[10], c, d[25];
float k[20], l[10], m, n;
```

This is the declaration, which actually is parsed by the lexical analyzer; LEX specifications that we have written here. These are all legal specifications. So, once declaration is found, it writes it into the text file and then ignores all the others. So, it writes it into a text file called declaration file and ignores all others.

The rest of the LEX specification is simply... In the main program, you open a file and call yylex. In the yywrap, you just do the wrapping routine, close the file, and get out. This is an example to show that it is possible to use LEX to parse even sizable parts of a programming language specification such as declaration, but I must hasten to add that not every declaration is so easy to parse within a LEX specification; some of these can be. I hope it conveys the essence of a LEX tool.

(Refer Slide Time: 26:04)



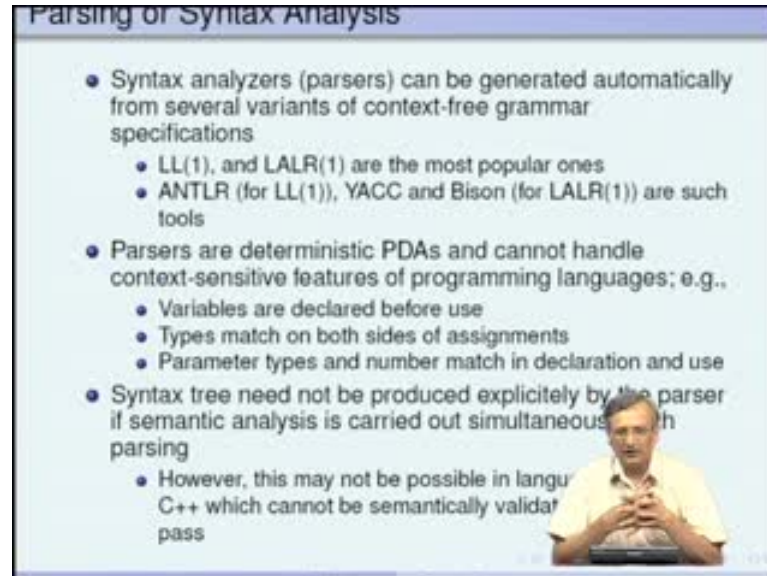
Let us move on and let us talk about syntax analysis. The lexical analyzer returns these tokens. The same assignment statement that we considered before – id assign id multop fconst addop iconst; these are all our tokens. fconst is the floating point constant and iconst is the integer constant. These are fed to the syntax analyzer. The syntax analyzer make sure that the assignment statement is indeed correct in syntax. In other words, there is an identifier on the left side of an assignment, there is an identifier on the right side of an assignment followed by an expression or whatever operator, and so on and so forth.

In general, the programming language constructs are complex. There is if then else, there is fall loop and so on and so forth. The lexical analyzer does not worry about such constructs. It simply returns tokens for most of these constructs. For example, if it is if then else, **it is then safe**. Then, for the entire expression, it returns a number of tokens followed by then and followed by the number of tokens for statement, and so on and so forth.

The syntax analyzer would look at the stream of tokens that is coming into it. It uses a context-free grammar to check whether the rules are all appropriately satisfied and then it constructs what is known as a syntax tree. Here is a syntax tree (Refer Slide Time: 27:41) for the assignment statement. The assignment has left child as an identifier, the right child as a plus operator. The plus operator has left child as a star and its right child is the constant. The star operator has identifier on the left-hand side and then the constant 1.8

on the right-hand side. So, such syntax trees are produced and fed to the semantic analyzer.

(Refer Slide Time: 28:15)



Syntax analyzers can be generated automatically from context-free grammar specifications. As I said, context-free grammar is the basis of parsing. A pushdown automaton is constructed from such a context-free grammar specification and then it is fed a sequence of tokens and it contains

There are many tools, which can do this. For example, ANTLR is a tool which takes LL 1 context-free grammar and produces a top-down parser. YACC and Bison; YACC is a tool with unix and bison is the corresponding tool available from GNU. These take LALR 1 form of context-free grammar and produces a parser for such grammars. These parsers are all deterministic pushdown automaton, but the main problem with these parsers are – they cannot handle any semantic features of programming languages, which are known as context sensitivity features of a programming language.

For example: If you have variables in the program, obviously you would have number of them; to check whether your variables have been declared before they are used in the program is a context sensitive feature. You really cannot check whether it is possible or such a declaration exists. Secondly, whether the left and right sides of an assignment match; it is something that we really cannot check in a context-free grammar and using a pushdown automaton. The reason is – for a context-free grammar and a pushdown

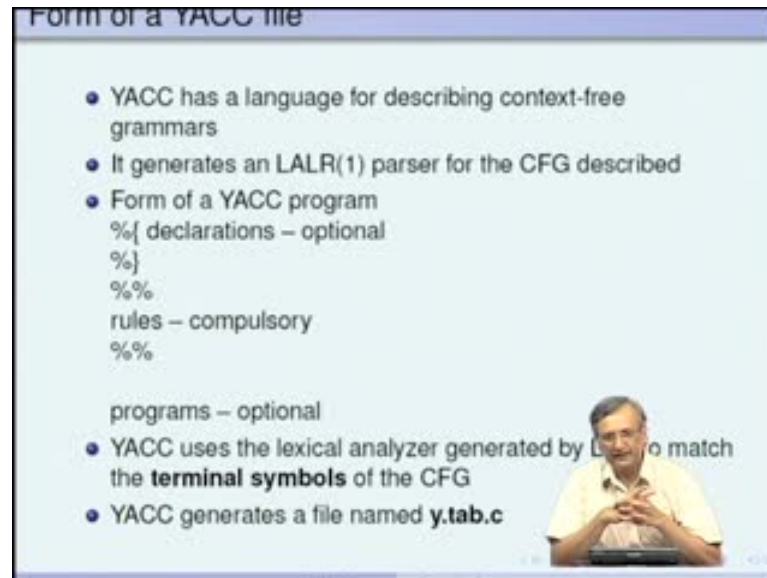
automaton, which is produced by it, whether the left hand side is an array name or whether it is a simple integer name is not known. That information cannot be captured in a context-free grammar. Therefore, checking whether the right-hand side also happens to be an array of values or whether it is simple arithmetic expression producing an integer value, cannot be checked by the same context-free grammar. For doing this, we need special types of grammars called attribute grammar and we will see a simple example very soon.

Third example of a context sensitivity feature is regarding parameter. You would have a number parameters in a function and you would actually put down the declaration of the function and its parameters and then call the function with the actual parameter list. Whether the types of parameters in the usage match the types of parameters in the declaration is a context sensitive feature. This cannot be captured in a context-free grammar and therefore, we need the next phase of compiler called the semantic analyzer phase.

A syntax tree as I said, will be produced as the output from a syntax analyzer, but I must add that this does not always happen. In some cases, if the entire compiler is a one-pass compiler; in other words, it produces even the machine code in one-pass, it is not necessary to produce the syntax tree explicitly. However, if there are language constructs such as in C plus plus, which says – you can use the variables and put the declaration elsewhere, perhaps much later; that is possible in the class in C plus plus; such constructs cannot be validated semantically in a single pass.

We need to produce the syntax tree decorated with some of the semantic information available from the program and pass this entire thing to the semantic analyzer for validation. So, that is really what is we need to see next.

(Refer Slide Time: 33:05)



Form of a YACC file

- YACC has a language for describing context-free grammars
- It generates an LALR(1) parser for the CFG described
- Form of a YACC program

```
%{ declarations – optional
%}
%%
rules – compulsory
%%

programs – optional
```
- YACC uses the lexical analyzer generated by LEX to match the **terminal symbols** of the CFG
- YACC generates a file named **y.tab.c**


Before that, let us see how a parser specification is written for a very simple expression parser. Let us use YACC for this. Yet Another Compiler-Compiler is the expansion of the acronym YACC. YACC has a language for describing context-free grammars. The productions are all going to be described by this particular language. It generates an LALR 1 parser for the context-free grammar that we describe. Its description is very similar. There are declarations very similar to that of LEX. There are declarations, which are optional, then the rules are context-free grammar productions, which are compulsory, and then some programs.

What is important is that YACC uses the lexical analyzer generated by LEX with great ease. The terminal symbols of the context-free grammar, which are specified by YACC should actually be produced as tokens by the lexical analyzer. Finally, YACC generates a file called y dot tab dot c.

(Refer Slide Time: 34:25)

```
YACC Example: LEX Specification

number [0-9]+\.[0-9]*|[0-9]*\.[0-9]+
name [A-Za-z][A-Za-z0-9]*
%%
[ ] /* skip blanks */
(number) {sscanf(yytext,"%lf",&yyval.dval);
         return NUMBER;}
(name) {struct symtab *sp =symlook(yytext);
        yyval.symp = sp; return NAME;}
"++" (return POSTPLUS;)
"--" (return POSTMINUS;)
"$" (return 0;)
\n|. (return yytext[0];)
```




This is a LEX specification for the expression parser that we already saw. It has NUMBER, it has NAME, then POSTPLUS, POSTMINUS as its tokens.

(Refer Slide Time: 34:40)

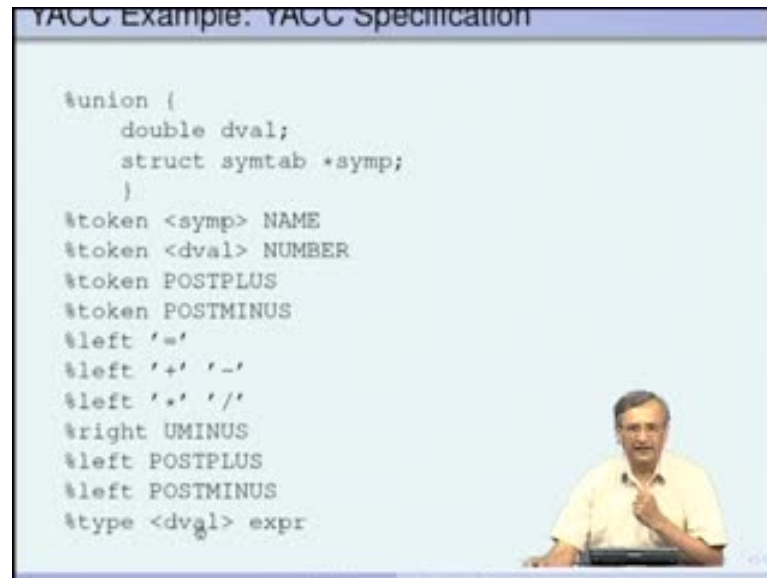
```
YACC Example: YACC Specification

%{
#define NSYMS 20
struct symtab {
    char *name; double value;
}symboltab[NSYMS];
struct symtab *symlook();
#include <string.h>
#include <ctype.h>
#include <stdio.h>
%}
```



Here is the YACC specification. To begin with, there are some declarations of a symbol table, then a routine called symlook, and then there are some include statements. These are all part of the user code, which is supplied.

(Refer Slide Time: 34:56)



```
YACC Example: YACC Specification

%union {
    double dval;
    struct symtab *symp;
}

%token <symp> NAME
%token <dval> NUMBER
%token POSTPLUS
%token POSTMINUS
%left '='
%left '+' '-'
%left '*' '/'
%right UMINUS
%left POSTPLUS
%left POSTMINUS
%type <dval> expr
```

Now, we start declaring tokens. We will be come to this union a little later. There are tokens called NAME, NUMBER, POSTPLUS and POSTMINUS. Then, we have this equal to, plus, minus, star, slash, then unary minus, then POSTPLUS, POSTMINUS, so many of them. We have also talked about the left associativity and right associativity of some of these operators.

Tokens are NAME, NUMBER, POSTPLUS, POSTMINUS, equal to, plus, minus, star and slash. It says that equal to, plus, minus, star and slash are left associative and it says that UMINUS is right associative, POSTPLUS and POSTMINUS are left associative. Then, it is possible to attach some semantic information to non-terminals and terminals.

For terminals such as NAME and NUMBER, there is a dval field, which is described in the union statement above (Refer Slide Time: 36:10); it is a double field. This symp field is a pointer into the symbol table. So, for NUMBERS, double is the declaration of the value of the token and for NAMES, a pointer into the symbol table is the value of the token. For non-terminals such as expression, again dval, which is double is the semantic information associated with it.

(Refer Slide Time: 36:37)

```
YACC Example: YACC Specification

%%
lines: lines expr '\n' {printf("%g\n", $2);}
      | lines '\n'
      | /* empty */
      | error '\n'
      {yyerror("reenter last line:"); yyerrok; }
;

expr : NAME '=' expr {$1 -> value = $3; $$ = $3;}
      | NAME {$$ = $1 -> value;}
      | expr '+' expr {$$ = $1 + $3;}
      | expr '-' expr {$$ = $1 - $3;}
      | expr '*' expr {$$ = $1 * $3;}
      | expr '/' expr {$$ = $1 / $3;}
      | '(' expr ')' {$$ = $2;}
      | '-' expr %prec UMINUS {$$ = - $2;}

```

Here is the specification. Let us look at expression to begin with. NAME equal to expression; that is the assignment statement, expression plus expression, expression minus expression, expression star expression and expression slash expression. These are all the right-hand sides of the various rules and then of course, parentheses expression parentheses, minus expression.

(Refer Slide Time: 37:09)

```
YACC Example: YACC Specification

      | NUMBER
      | NUMBER POSTPLUS %prec POSTPLUS
        {$$ = $1 + 1;}
      | NUMBER POSTMINUS %prec POSTMINUS
        {$$ = $1 - 1;}
;

%%
void initsytab()
{int i = 0;
 for(i=0; i<NSYMS; i++) symboltab[i].name = NULL;
}
int yywrap(){return 1;}
yyerror( char* s) { printf("%s\n",s);}
main() {initsytab(); yyparse(); }

#include "lex.yy.c"

```

There are some more – NUMBER, NUMBER POSTPLUS, NUMBER POSTPLUS; these are the various right-hand sides of the productions.

(Refer Slide Time: 37:16)

```
YACC Example: YACC Specification

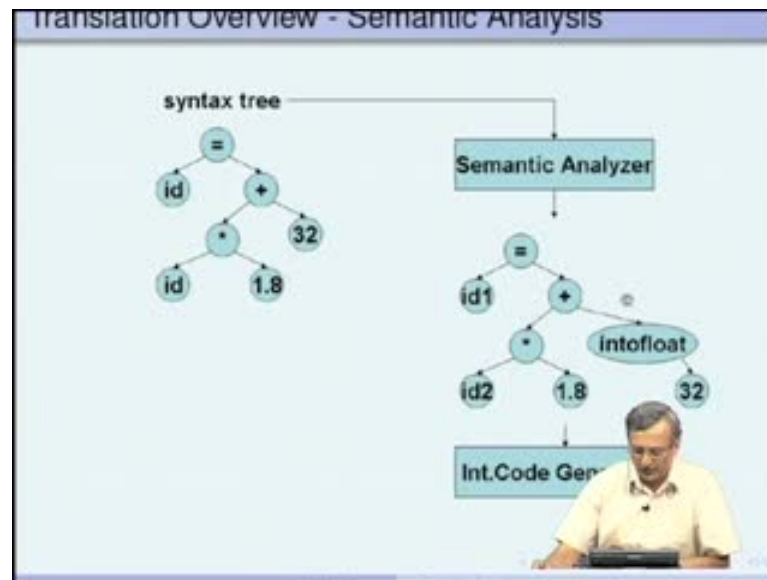
%%
lines: lines expr '\n' {printf("%g\n", $2);}
      | lines '\n'
      | /* empty */
      | error '\n'
        {yyerror("reenter last line:"); yyerrok; }
;
expr : NAME '=' expr {$1 -> value = $3; $$ = $3;}
      | NAME {$$ = $1 -> value;}
      | expr '+' expr {$$ = $1 + $3;}
      | expr '-' expr {$$ = $1 - $3;}
      | expr '*' expr {$$ = $1 * $3;}
      | expr '/' expr {$$ = $1 / $3;}
      | '(' expr ')' {$$ = $2;}
      | '-' expr %prec UMINUS {$$ = - $2;}
;

```

The left-hand side is common; that is expression. So, expression going to NAME equal to expression, expression going to NAME; this is the way to read this particular grammar specification. What is happening here? lines is nothing but a couple of productions added to make sure that the calculator does not stop prematurely. It will actually be going into an infinite cycle until you press some strange characters. The YACC specification also has an action part here. For example, NAME is equal to expression; it says that dollar 1 pointer value is dollar 3 and dollar dollar is dollar 3. So, dollar dollar is the value of the left-hand side symbol expression, dollar 1 pointer value is the value of the token name, and dollar 3 is the value of the expression in the right-hand side of the production, NAME equal to expression. So, this simple says – the value of the left-hand side non-terminal expression is nothing but the value produced by the expression on the right-hand side, which is fine with us; that is the way it should be. Similarly, expression plus expression says - the value of the left-hand side is dollar dollar equal to dollar 1 plus dollar 3, which is the sum of the two values produced by the two expressions. So, this is the way it continues. This is just to give you a sample of how YACCs specifications are returned.

I am going to skip the symbol table routines because they are not really important for our discussion.

(Refer Slide Time: 39:16)



After the syntax analysis part of the translation, we move on to what is known as the semantic analysis. In the case of semantic analysis, the input to the semantic analyzer is the syntax tree and which has information from the program. Finally, it validates this particular syntax tree along with the information available from the program and then produces what is known as a semantically validated syntax tree, which is the input to the next phase of the compiler namely, the intermediate code generator.

(Refer Slide Time: 40:03)

-
- Semantic Analysis**
- Semantic consistency that cannot be handled at the parsing stage is handled here
 - Type checking of various programming language constructs is one of the most important tasks
 - Stores type information in the symbol table or the syntax tree
 - Types of variables, function parameters, array dimensions, etc.
 - Used not only for semantic validation but also for subsequent phases of compilation
 - Static semantics of programming languages can be specified using attribute grammars
 - Semantic analyzers can be generated automatically from attributed translation grammars
 - If declarations need not appear before use (as in C++), semantic analysis needs more than one pass

What is semantic analysis? Semantic analysis handles actually the features of a program, which cannot be handled at the syntax level. As I mentioned, type checking - whether the left-hand side of an assignment is the same as right-hand side of an assignment. The type (()) whether I am wrongly assigning an array some value, which is not an array or am I assigning a character value to an integer variable. These are all the kind of checks that I want to do in semantic analysis.

During semantic analysis, we also need a huge table called the symbol table, which stores the names of the variables and their types, parameters of functions and their types, dimensions of an array, and so on and so forth. This particular symbol table is useful not only during compilation, but also for other purposes such as debugging. For example, when you turn on the debugger in GCC, the compiler actually includes the entire symbol table in the assembly code, which is produced by it. That is how actually the debugger can know – what is a variable, which variable is it, what is its type and so on and so forth. Otherwise, it is impossible for the binary code to find out the types of such variables.

The specifications, which can be used in semantic analysis can be provided by what are known as attribute grammars. Attribute grammars can specify what are known as static semantics of programming languages, but not dynamic semantics. Dynamic semantics are – what happens at run time; that cannot be specified by attribute grammar and there are no suitable specifications for these either. We will have to actually generate code to check such violations in the code itself.

It is possible to generate semantic analyzers automatically from attributed translation grammars and we will very soon see an example of how this can be done. If declarations need not appear before use as in c plus plus, semantic analysis actually needs more than one phase. It may not be possible to do this semantic analysis in just one phase, we may need more than one phase.

(Refer Slide Time: 42:52)

Example of an Attribute Grammar

- Let us first consider the CFG for a simple language
 - $S \rightarrow E$
 - $E \rightarrow E + T \mid T \mid \text{let } id = E \text{ in } (E)$
 - $T \rightarrow T * F \mid F$
 - $F \rightarrow (E) \mid \text{number} \mid id$
- This language permits expressions to be nested inside expressions and have scopes for the names
 - *let A = 5 in ((let A = 6 in (A*7)) - A)* evaluates correctly to 37, with the scopes of the two instances of A being different
- It requires a scoped symbol table for implementation
- The next slide shows an abstract attribute grammar for the above language and the slide following it shows an implementation of the abstract AG using a YACC-style translation grammar
- Abstract AGs permit both inherited and synthesized attributes, whereas YACC-style grammars permit only synthesized attributes

YN. Srikant Compiler Overview

Here is an example of an attribute grammar. To begin with, we have a context-free grammar; S going to E, E going to E plus T, or T, or it is a let expression; so let id is equal to expression in expression, etcetera. Then, we have a non-terminal T going T star F or F and finally, F goes to parentheses expression parentheses or number or id. What are the specialties of this language.

This language actually permits expressions to be nested inside another expression. So, we have nested expressions possible and we can also have scopes for the names inside these expressions. Here is an example - Let A equal to 5 in let A equal to 6 in A star 7. So, the inner A has this restricted scope of let A equal to 6 in A star 7 and the outer A actually cannot interfere with the expression A star 7. So, the inner A rules there and the outer A rules only in the outer level of the expression; that is, the entire expression 5 in let A equal to 6 in A star 7 minus A. The second A is actually 5. This evaluates correctly to 41 provided the scopes of the two instances of A are treated as different. So, if the inner A retains the value 6 and A star 7 is evaluated with that 6 and if the outer A retains the value 5 and it is used for the outer A, then the value of the expression is 41.

Such a programming language of expression requires a scope symbol table for implementation. Let us see how an abstract attribute grammar is used to specify such a language and then see how this works. Abstract attribute grammars use what are known

as inherited attributes and synthesized attributes, whereas YACC permits only synthesized attributes and its specifications.

(Refer Slide Time: 45:41)

An Abstract Attribute Grammar

- 1. $S \rightarrow E \{ E.syntab \downarrow := \phi; S.val \uparrow := E.val \uparrow \}$
- 2. $E_1 \rightarrow E_2 + T \{ E_2.syntab \downarrow := E_1.syntab \downarrow; E_1.val \uparrow := E_2.val \uparrow + T.val \uparrow; T.syntab \downarrow := E_1.syntab \downarrow \}$
- 3. $E \rightarrow T \{ T.syntab \downarrow := E.syntab \downarrow; E.val \uparrow := T.val \uparrow \}$
- 4. $E_1 \rightarrow \text{let } id = E_2 \text{ in } (E_3) \{ E_1.val \uparrow := E_3.val \uparrow; E_2.syntab \downarrow := E_1.syntab \downarrow; E_3.syntab \downarrow := E_1.syntab \downarrow \setminus \{ id.name \uparrow \rightarrow E_2.val \uparrow \} \}$
- 5. $T_1 \rightarrow T_2 * F \{ T_1.val \uparrow := T_2.val \uparrow * F.val \uparrow; T_2.syntab \downarrow := T_1.syntab \downarrow; F.syntab \downarrow := T_1.syntab \downarrow \}$
- 6. $T \rightarrow F \{ T.val \uparrow := F.val \uparrow; F.syntab \downarrow := T.syntab \downarrow \}$
- 7. $F \rightarrow (E) \{ F.val \uparrow := E.val \uparrow; E.syntab \downarrow := F.syntab \downarrow \}$
- 8. $F \rightarrow \text{number} \{ F.val \uparrow := \text{number.val} \uparrow \}$
- 9. $F \rightarrow \text{id} \{ F.val \uparrow := F.syntab \downarrow [id.name \uparrow] \}$

Here is an attribute grammar. The first production is S to E. The attribute grammar format is simple; the context-free grammar rule is written followed by the attribute competition rules.

The attributes with down arrow in front of them are inherited attributes. The attributes with up arrow in front of them are synthesized attributes. For example, in the first production, E dot syntab is inherited and S dot val E dot val are both synthesized. So, the rule corresponding to the production S going to E is very simple. It initializes the symbol table to a null symbol table and it says – whatever expression is produced by E, is the value produced by S as well; S dot val equal to E dot val.

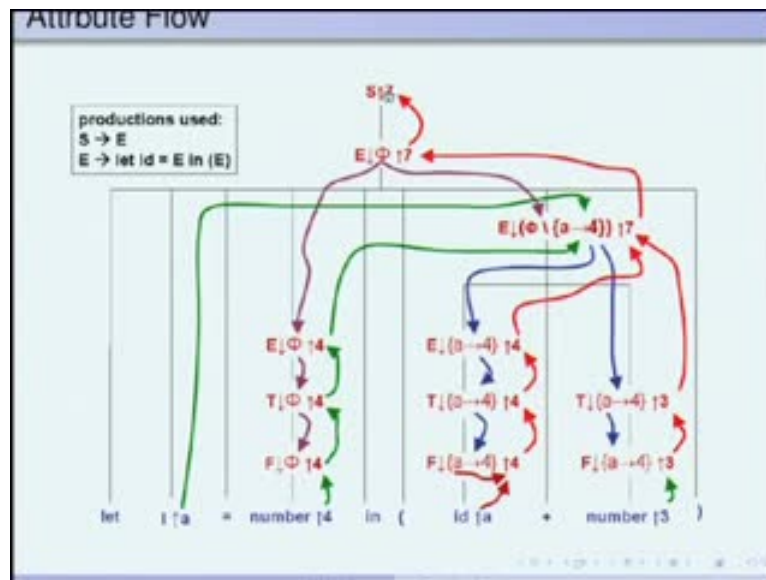
Let us directly go to the next production, E 1 going to E 2 plus T. E 1 and E 2 are the two instances of the non-terminal E. Here whatever is inherited from E 1; for example, here this is E 1, (Refer Slide Time: 47:06) it has a symbol table coming in and that is given to E 2 as a symbol table with which it should operate and that is also given to T as the symbol table with which it should operate. That is why, there are two statements: E 2 dot syntab equal to E 1 dot syntab and then T dot syntab equal to E 1 dot syntab. By the way, I should also mention that the order in which these statements are all written does

not correspond to a sequence of statements. The order of executing these statements is actually found out later by an attribute evaluator.

What is the value of E 1? E 1 dot val is the value, which is nothing but E 2 dot val and T dot val; added together. Similarly, let us take the most complicated expression E 1 going to let id equal to E 2 in E 3. The value of E 1 is the value of E 3. That is why, E 1 dot val is E 3 dot val. The symbol table for E 2, which has all the names including ones from outside is nothing but E 1 dot symtab. However, the symbol table for E 3 is very different. The symbol table for E 3 is the symbol table of E 1 with the name E 2 dot val pair overriding any similar name within E 1 dot symtab.

This operator (Refer Slide Time: 48:47), back slash is nothing but the overriding operator that we are going to define. So, if there is a name inside E 1 dot symtab, which is same as id dot name, that name is temporarily overridden by this particular new id dot name. The new id dot name will have the value E 2 dot val associated with it. So, this is the structure of the symbol table (Refer Slide Time: 49:15).

(Refer Slide Time: 49:22)



Let us now look at a simple example to see how exactly the attributes flow. The productions, which are used here are two of them: S to E and E going let id equal to E in E.

Here is a complete syntax tree for this particular sentence – let a equal to 4 in a plus 3; a simple sentence. The productions are S to E and then E going to this entire thing (Refer Slide Time: 49:50). So, that happens in several steps; E going to... It actually says – let id equal to E in again E. This is the first level. Then, this E expands further to T and then F and then number. This E expands to E plus T and then finally, to T, F and a. On this side, it expands to F and 3. So, this is the syntax tree.

We begin with a null symbol table. The null symbol table is handed over to E and it is also handed over to this side E (Refer Slide Time: 50:36). This side, the symbol table continues to be null or empty and when we get 4, the value is actually handed over to F, which in turn goes to T, which in turn goes to E. This side, the value which is handed over as phi; the symbol value, which is handed over as phi, actually now gets updated. Let see how?

Here is a (Refer Slide Time: 51:06). Once we say this is 4, which is actually synthesized from E to T to F to 4 and this identifier a, which is already available are combined into an association a to 4. This is the overriding operator. So, phi overridden with a to 4 is the new symbol table, which is given to E. That symbol table continues to be handed over to its successors. So, a to 4 is the new symbol table, which goes down. As a to 4 goes down, it meets another a. So, E to T to F to a. During F to a, there is a symbol, which is produced here (Refer Slide Time: 51:51), a; this a produces the value 4, when it is looked up in this particular symbol table and the value of 4 goes up. Number 3 goes up without any difficulty; it does not need a symbol table. These two numbers are combined into the value 7 because of the production E going E plus T. So, the values of these two are added, 7 is produced here (Refer Slide Time: 52:15). This 7 is passed on to the root as the value produced by this start symbol.

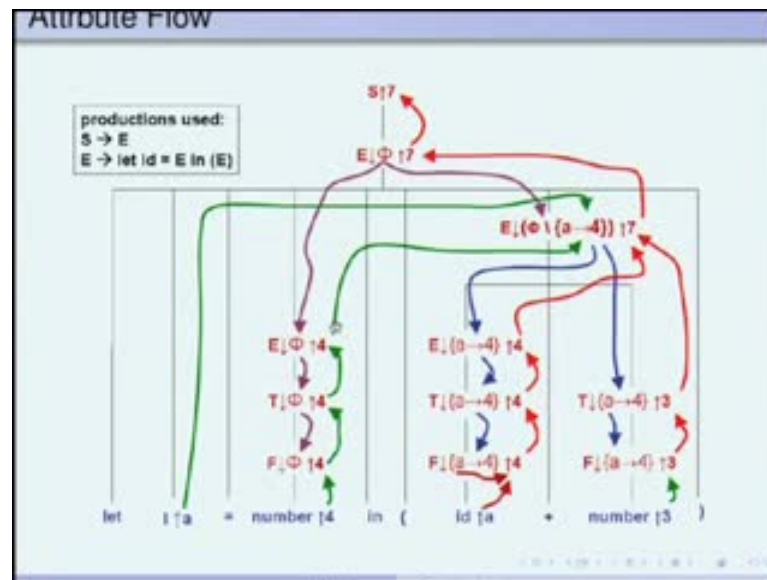
(Refer Slide Time: 52:23)

An Abstract Attribute Grammar	
1	$S \rightarrow E \{ E.symtab \downarrow := \phi; S.val \uparrow := E.val \uparrow \}$
2	$E_1 \rightarrow E_2 + T \{ E_2.symtab \downarrow := E_1.symtab \downarrow; E_1.val \uparrow := E_2.val \uparrow + T.val \uparrow; T.symtab \downarrow := E_1.symtab \downarrow \}$
3	$E \rightarrow T \{ T.symtab \downarrow := E.symtab \downarrow; E.val \uparrow := T.val \uparrow \}$
4	$E_1 \rightarrow \text{let } id = E_2 \text{ in } (E_3) \{ E_1.val \uparrow := E_3.val \uparrow; E_2.symtab \downarrow := E_1.symtab \downarrow; E_3.symtab \downarrow := E_1.symtab \downarrow \setminus \{ id.name \uparrow \rightarrow E_2.val \uparrow \} \}$
5	$T_1 \rightarrow T_2 * F \{ T_1.val \uparrow := T_2.val \uparrow * F.val \uparrow; T_2.symtab \downarrow := T_1.symtab \downarrow; F.symtab \downarrow := T_1.symtab \downarrow \}$
6	$T \rightarrow F \{ T.val \uparrow := F.val \uparrow; F.symtab \downarrow := T.symtab \downarrow \}$
7	$F \rightarrow (E) \{ F.val \uparrow := E.val \uparrow; E.symtab \downarrow := F.symtab \downarrow \}$
8	$F \rightarrow \text{number} \{ F.val \uparrow := \text{number.val} \uparrow \}$
9	$F \rightarrow \text{id} \{ F.val \uparrow := F.symtab \downarrow [id.name \uparrow] \}$

Now, we can go through this little more. For example, we saw that F to number; the value of F is nothing but the number itself. So, the number value is passed on. What is the semantics of F to id? F dot val is the value produced by looking up this particular name inside the symbol table F dot symtab.

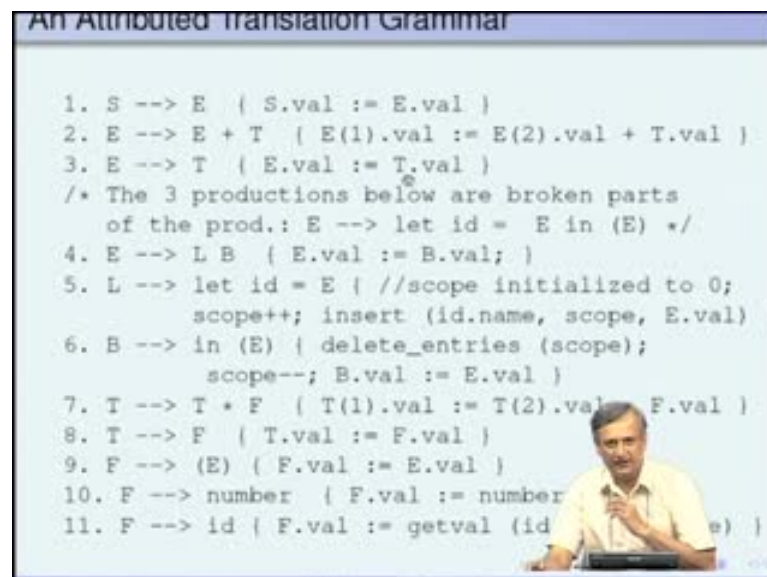
What is the value of T to F? Whatever value produced by F is passed on to T. What is the value of T 2 star F? We take the value of T 2 dot val, we take the value of F dot val, and then we add up these two values and that is the value which is produced as the value of T 1.

(Refer Slide Time: 53:21)



This is the way the attributes are computed and then they are passed on to the **start symbol**.

(Refer Slide Time: 53:27)



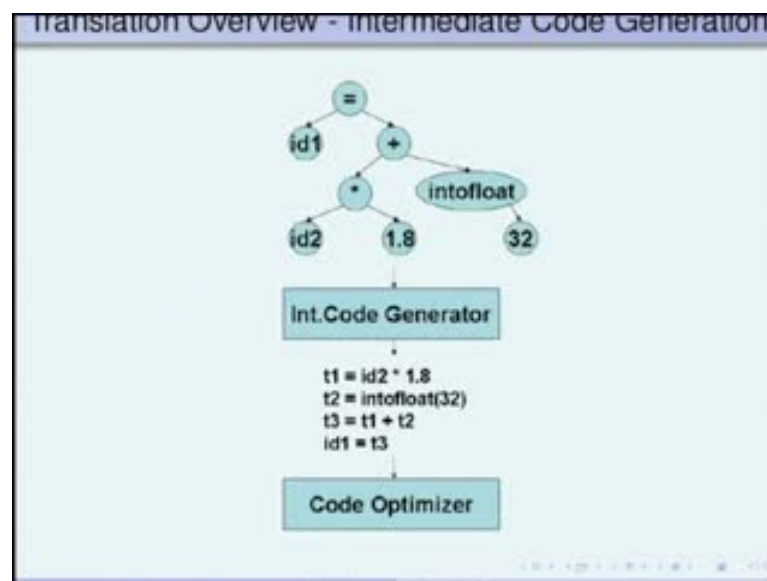
How are these translation grammars implemented? Let us say – we consider YACC. In YACC, how do we implement such translation grammars?

The most important thing to observe is – these are all very simple and these can be read and understood very easily because they are very similar to what we had before. The symbol table is going to be a global structure here. The other important thing that we

need to do is to make sure that we understand this production, which breaks the single production E going to let id equal to E in E into these three productions: E to L B, L to let id equal to E and B going to in E. In these two productions everything else happens.

In this production (Refer Slide Time: 54:15), there is a new scope, which is generated and the name is inserted with the new scope. Once we complete this entire production, the entries of the previous scope are deleted, the scope number is reduced and we return. In other words, this breakage of productions is essential because YACC permits addition of rules only at the end of a context-free grammar production. Further, it allows only synthesized attribute. In the inherited attribute, the symbol table is implemented in the form of a global variable. Similarly, in order to make sure that it is available wherever it is used, we have to make it a global variable. These are the two reasons why we need to make this symbol table into a global variable.

(Refer Slide Time: 55:25)



We will stop the lecture at this point with a picture saying that in the next class, we will be looking at conversion of the semantically validated syntax tree into intermediate code and then look at what happens to intermediate code when it goes through the machine code generation phase and optimization phase, etcetera.

Thank you.