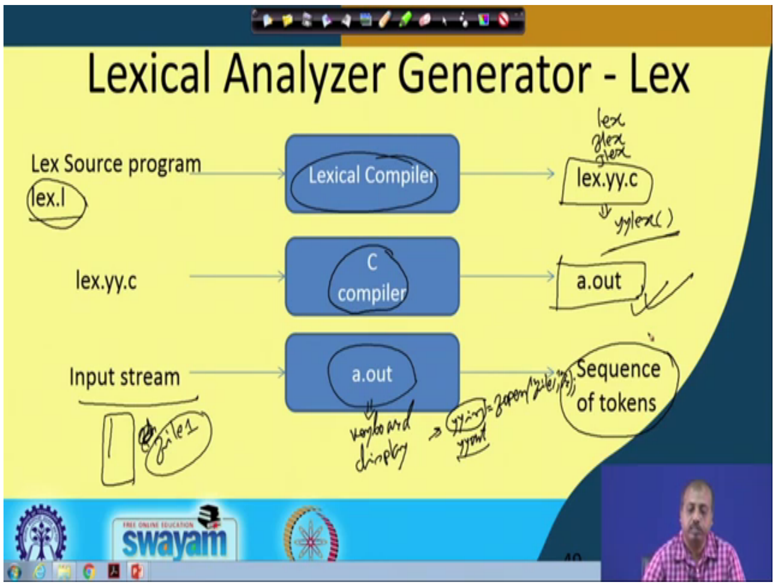
Compiler Design
Prof. Santanu Chattopadhyay
Department of E and EC Engineering
Indian Institute of Technology, Kharagpur

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 converted 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 a 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 generating the compilers automatically.

So, in that direction the first tool that have was 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.

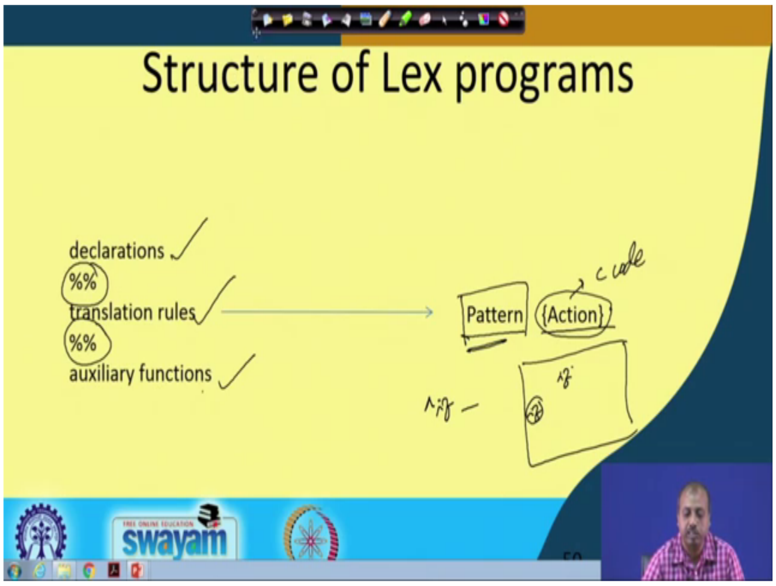
Whatever it is so, this is the executable file that is produced. So, this is actually the lexical analyzer this a dot out is the lexical analyzer. So, if you run this program a dot out and a you give it an input stream then so, this a dot out program. So, it accepts the input from the keyboard. And it 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)



Structure of Lex programs

declarations

%%

translation rules

%%

auxiliary functions

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.

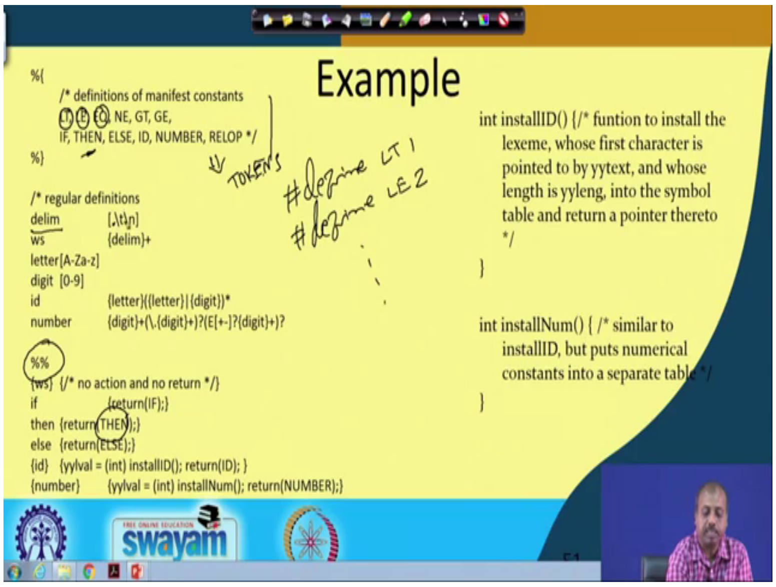
So, this way this pattern is more powerful compared to the regular expression that we are familiar with it theoretically. So, that is why it is written as pattern and in the pattern you can give the corresponding action. So, this translation rules it will have all the patterns that the programs should 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)



Example

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions
delim       [ \t\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}  {yylval = (int) installID(); return(ID); }
{number}    {yylval = (int) installNum(); return(NUMBER);}
```

```
int installID() {/* funtion to install the
    lexeme, whose first character is
    pointed to by yytext, and whose
    length is yyleng, into the symbol
    table and return a pointer thereto
    */
}

int installNum() { /* similar to
    installID, but puts numerical
    constants into a separate table */
}
```

TOKENS

#define LT 1
#define LE 2

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 in 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.

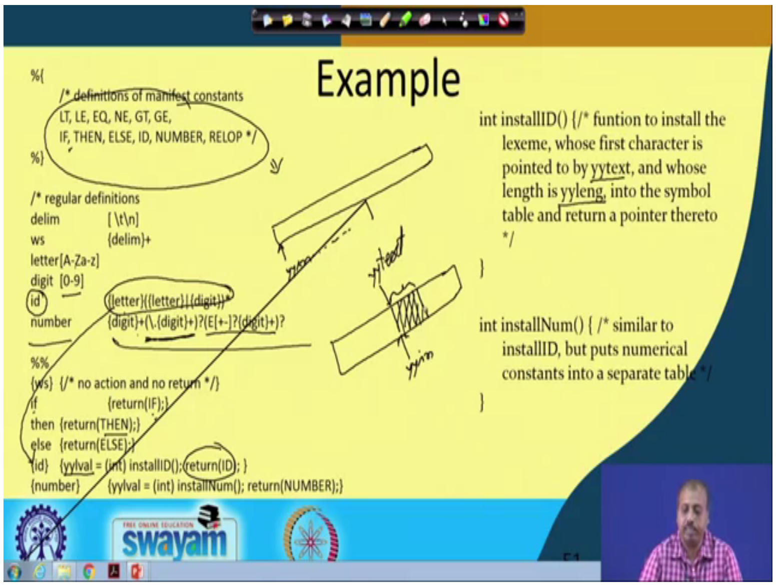
So, here I have got the delimiters. So, delim is a regular definition so, we have got this is a blank this is a blank space tab and newline character. So, to just to say mean that this is a tab not the character t. So, it is written as reverse slash and then t. So, reverse slash in the UNIX operating 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 a 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)



Example

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions
delim       [ \t\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}   {yylval = (int) installID(); return(ID); }
{number}    {yylval = (int) installNum(); return(NUMBER);}
```

```
int installID() {/* funtion to install the
    lexeme, whose first character is
    pointed to by yytext, and whose
    length is yyleng, into the symbol
    table and return a pointer thereto
    */
}

int installNum() { /* similar to
    installID, but puts numerical
    constants into a separate table */
}
```

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.

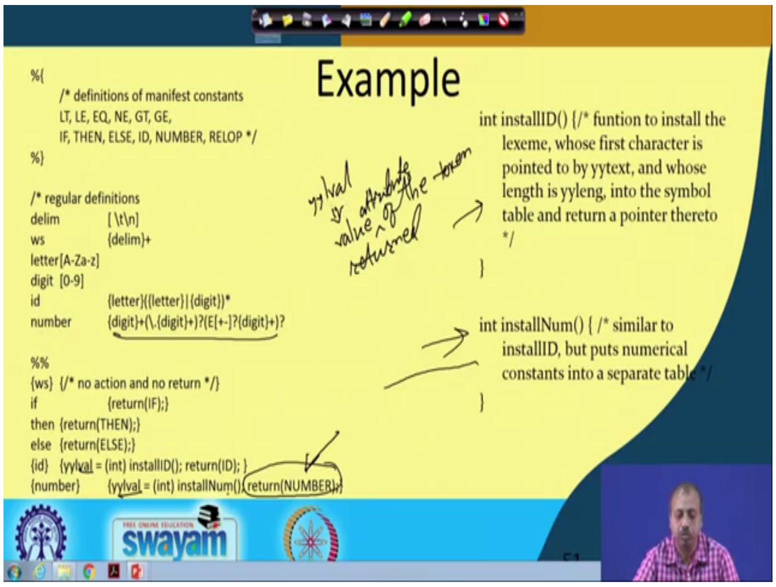
And then the action part you see there are two things; one part is it is returning the constant corresponding to id the token corresponding to id. But, before that it is setting another variable yy l val to this return value of this function install ID. So, install ID is a routine it is an auxiliary 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 l val equal to install number. So, this yy l val so, this is another important variable yy l val.

(Refer Slide Time: 20:51)



Example

```
%{
     /* definitions of manifest constants
     LT, LE, EQ, NE, GT, GE,
     IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions
delim          [ \t\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}  {yylval = (int) installID(); return(ID); }
{number}       {yylval = (int) installNum(); return(NUMBER);}
```

```
int installID() {/* funtion to install the
     lexeme, whose first character is
     pointed to by yytext, and whose
     length is yyleng, into the symbol
     table and return a pointer thereto
     */
}

int installNum() { /* similar to
     installID, but puts numerical
     constants into a separate table */
}
```

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 l val variable can be used.

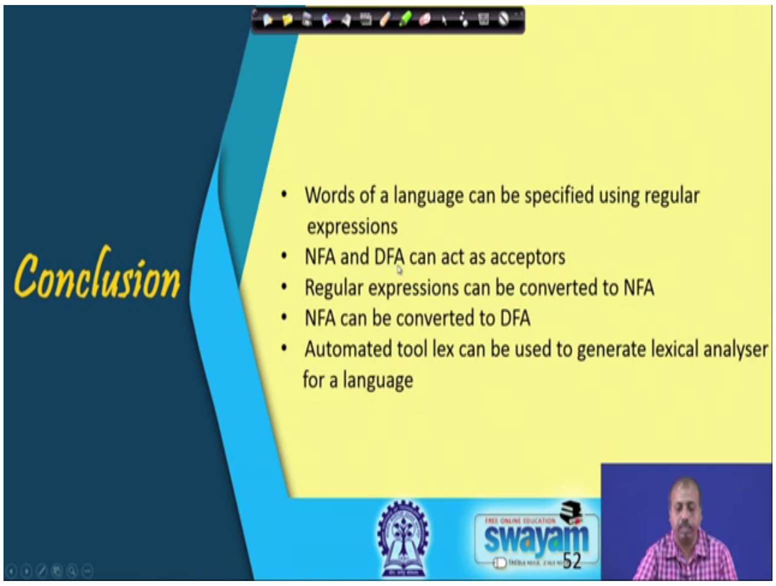
So, normally this parser it has got access to all these variables like yy text, y y leng, yy l val and all that and yy l val is going to be a very important attribute that you can see later. So, here for the number you have given a call to yy this install numbers. So, the number is installed into the 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 l 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 l 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)



Conclusion

- 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

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 is 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 a 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 deter for determining 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.