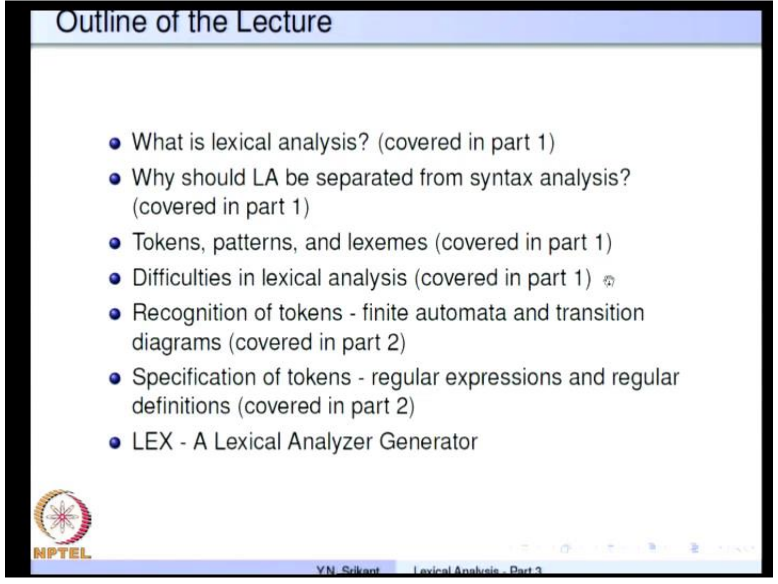
Principles of Compiler Design
Prof. Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

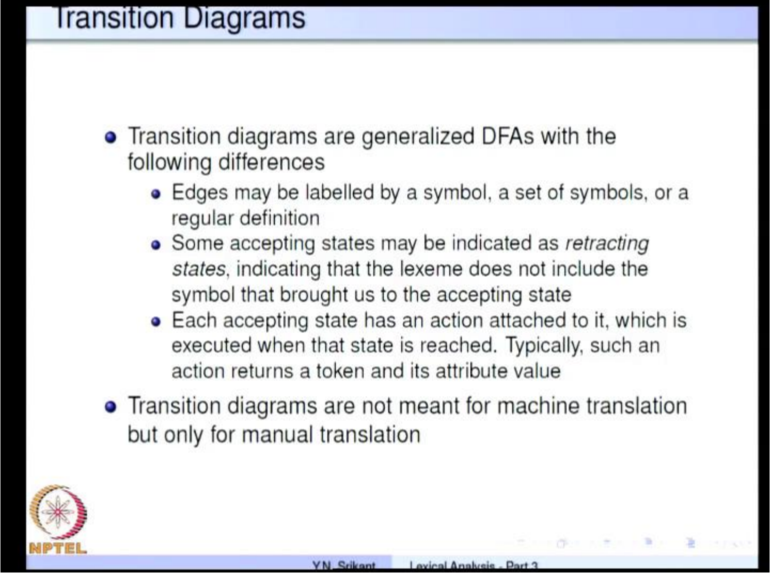
Lecture - 4
Lexical Analysis-Part-3

(Refer Slide Time: 00:20)



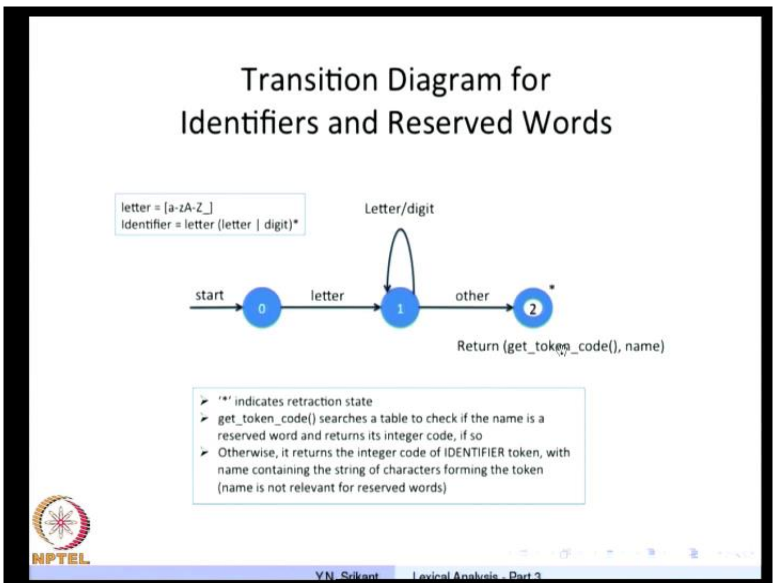
Welcome to the lecture on lexical analysis part three. So, in part one we covered the basics of lexical analysis the motivation, etcetera, and we covered the theoretical fundamentals, finite automata, regular expressions, and transition diagrams in part 2. Today we will continue a little more about transition diagrams and generation of lexical analyzers and then study ((Refer Time: 00:46)), which generates lexical analyzers.

(Refer Slide Time: 00:49)



So, to do a bit of recap transition diagrams are generalized deterministic finite automata, but there are a few differences. The edges may be labeled by a symbol set of symbols or a regular definition, some accepting states may be indicated as retracting states. And when we reach a retracting state we really do not consume the symbol which brings us to that particular state. And then with each accepting state there is an action which is executed when that state is reached. Typically we use this action to return a token and its attribute value, but very important transition diagrams are meant for you know manual translation and not for machine translation.

(Refer Slide Time: 01:48)



So, let us understand how exactly transition diagrams can be translated to lexical analyzers. So, the first transition diagram will be considering is the one for reserved words and identifiers. So, this a fairly simple and straight forward transition diagram. It starts in state 0 then on a letter we reach state one we keep on consuming letters and digits in state one and when we finally, get some other symbol other than letter or digit we reach state two, where we return the token code or say for the reserved word or for the identifier itself. So, how does this really translate to a program?

(Refer Slide Time: 02:21)

Lexical Analyzer Implementation from Trans. Diagrams

```
TOKEN gettoken() {
   TOKEN mytoken; char c;
   while(1) { switch (state) {
     /* recognize reserved words and identifiers */
       case 0: c = nextchar(); if (letter(c))
               state = 1; else state = failure();
               break;
       case 1: c = nextchar();
               if (letter(c) || digit(c))
               state = 1; else state = 2; break;
       case 2: retract(1);
               mytoken.token = search_token();
               if (mytoken.token == IDENTIFIER)
               mytoken.value = get_id_string();
               return(mytoken);
```

So, the lexical analyzer you know is called a get token and it returns a type called token. So, inside the lexical analyzer we have two local variables my token and c. So, there is a while loop, which runs until the end of file. So, which has a switch on the state. So, to begin with we will always be in state 0 and then you know for this particular transition diagram, we will be in state 0. And as we read a character in state 0 if it is a letter we change to state one, otherwise we report it as a failure that is because here the only legal character is a letter.

And in state one we check whether it is a letter or digit after reading a character if serviced we remain in the same state otherwise we switch to state two. In state two we have actually we have got ready to announce a token. So, the symbol which brought us to state two is not consumed it is push back to the input. The token is obtained by searching the table of tokens. So, if it is a reserved word then you know the reserved

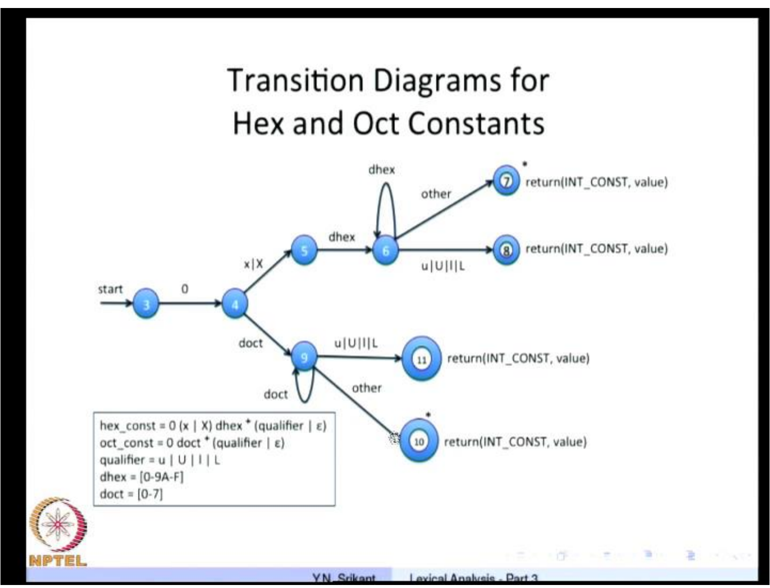
word token is returned by the search token and if it is an identifier then we simply get the string corresponding to the identifier, put it in my token dot value and return my token.

(Refer Slide Time: 03:56)

Lexical Analyzer Implementation from Trans. Diagrams

```
/* recognize hexa and octal constants */
  case 3: c = nextchar();
          if (c == '0') state = 4; break;
          else state = failure();
  case 4: c = nextchar();
          if ((c == 'x') || (c == 'X'))
          state = 5; else if (digitoct(c))
          state = 9; else state = failure();
          break;
  case 5: c = nextchar(); if (digithex(c))
          state = 6; else state = failure();
          break;
```

(Refer Slide Time: 04:00)



Similarly, when we want to… You know catch constants various types of constants hexa, octa, etcetera, etcetera we start in state 0 and little X or a X we go to the hex constant part. And on d oct the octal digit we come to the octal part, we keep consuming digits here until we reach return a token. So, this is the scheme both for hexadecimal and octal you know digits rather constants. So, this is fairly straight forward, we remain you know

if the character is a 0 than it is an octal constant. So, we go to state four and if it is not a 0 then it is a failure because it is an illegal character and in state four we check you know whether it is an X or x. So, it could be a hexadecimal constant or it could be an octal constant.

So, in you know if it is a hexadecimal we go to state 5 and if it is octal the we go to state 9 and. So, the two constants hexadecimal and octal are separated by you know one of them the hexadecimal is prefixed by X whereas, the octal constant is not prefixed by X. So, if neither of these appear in the input then it is a failure in state 5 we go on checking the for the hexadecimal digits and we go to state 6 on receiving hexadecimal digit.

(Refer Slide Time: 05:36)

Lexical Analyzer Implementation from Trans. Diagrams

```
    case 6: c = nextchar(); if (digithex(c))
            state = 6; else if ((c == 'u')||
            (c == 'U')||(c == 'l')||
            (c == 'L')) state = 8;
            else state = 7; break;
    case 7: retract(1);
/* fall through to case 8, to save coding */
    case 8: mytoken.token = INT_CONST;
            mytoken.value = eval_hex_num();
            return(mytoken);
    case 9: c = nextchar(); if (digitoct(c))
            state = 9; else if ((c == 'u')||
            (c == 'U')||(c == 'l')||(c == 'L'))
            state = 11; else state = 10; break;
```

NPTEL

Y.N. Srikant Lexical Analysis - Part 3

We remain in state 6 until we get hexadecimal digits and then we either go to state 8 or state seven depending on the qualifier presence or otherwise. So, here again you know we return the token called integer constant because hexadecimal constants are also integers. So, after and we evaluate hexadecimal number and return its value as the attribute of the token. So, in state 9 we do something similar and check the octal digits here.

(Refer Slide Time: 06:12)

Lexical Analyzer Implementation from Trans. Diagrams

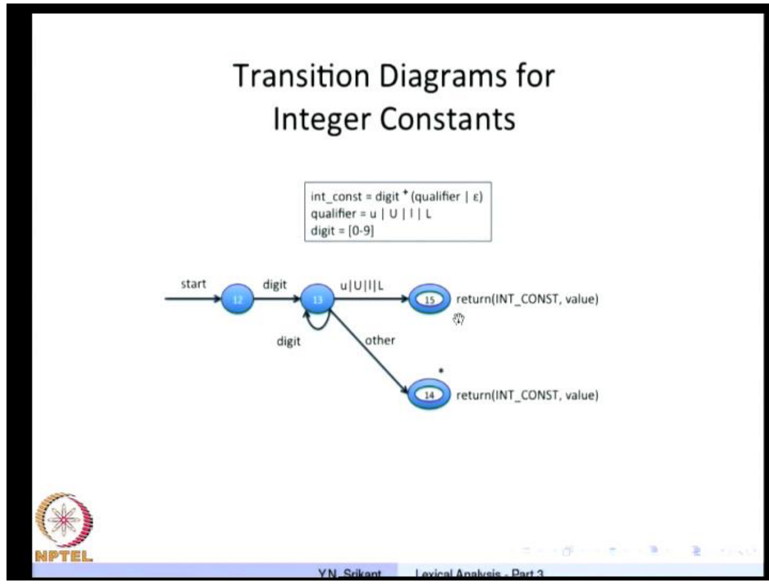
```
    case 10: retract(1);
/* fall through to case 11, to save coding */
    case 11: mytoken.token = INT_CONST;
             mytoken.value = eval_oct_num();
             return(mytoken);
```

NPTEL

Y.N. Srikant   Lexical Analysis - Part 3

So, and finally, we return int const the evaluated octal number as the token and its value.

(Refer Slide Time: 06:22)



So, for integer constants it is even simpler n number of digits are consumed and then the qualifier or otherwise is checked and we return in int const.

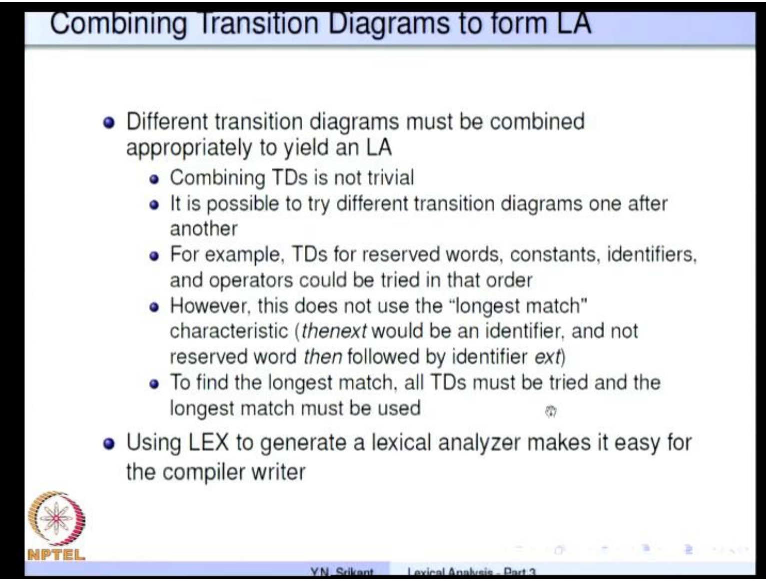
(Refer Slide Time: 06:34)

Lexical Analyzer Implementation from Trans. Diagrams

```
/* recognize integer constants */
    case 12: c = nextchar(); if (digit(c))
             state = 13; else state = failure();
    case 13: c = nextchar(); if (digit(c))
             state = 13;else if ((c == 'u')||
             (c == 'U')||(c == 'l')||(c == 'L'))
             state = 15; else state = 14; break;
    case 14: retract(1);
/* fall through to case 15, to save coding */
    case 15: mytoken.token = INT_CONST;
             mytoken.value = eval_int_num();
             return(mytoken);
    default: recover();
    }
}
```

So, the integer constants are very easy we check whether it is a digit otherwise it is a failure, we consume digits, and finally in state 15 we get the we return the integer constant token along with its value.

(Refer Slide Time: 06:53)



So, this an overview of how the transition diagrams are translated into you know lexical analyzer programs, but there are some reality checks that must be done here. So, we actually saw program segments corresponding to different you know transition diagrams. So, these transition diagrams must be combined appropriately to make a big transition

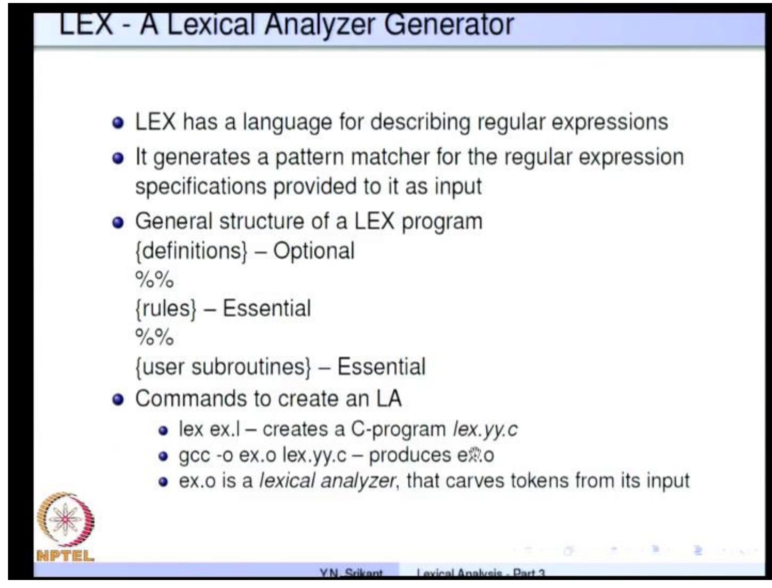
diagram. And then that transition diagram will have to be translated to manually into a lexical analyzer program. Unfortunately combining the transition diagrams is definitely not trivial and it is possible to, you know one possibility is order the transition diagrams in particular order say transition diagrams for reserved words, then constants then identifiers and operators, try each of these transition diagrams in a in that order.

So, in other words the programs are also listed exactly in that order. So, when there is a failure it goes to the next transition diagram and starts looking at that particular diagram. So, if this is followed you know it this is fairly easy to program; however, this does not use the longest match characteristics. In other word the word t h e n e x t would be an identifier and not a reserved word then followed by identifier e x t.

So, it should not be rather, but what happens is if you actually order the transition diagram for identifier first the next would be an identifier. But if you order the reserved word transition diagrams first then you know you would have the reserved word then followed by identifier e x t really, in reality it is better to use the longest match so. In fact, the next should be identifier rather than then followed by identifier e x t. So, how do we get this longest match? The transition diagrams must be tried in you know all of them must be tried, then all the matched must be recorded and the longest match must be used.

So, if this is done or if the programmer is able to order the transition diagrams appropriately, in either case the longest match can be used. Using lex to generate lexical analyzers, really makes it easy for the compiler writer. So, we will see how this works in the next few slides.

(Refer Slide Time: 09:46)



So, now you know. So, far we saw how to generate regular how, how to generate lexical analyzers you know manually using transition diagrams, this or that approach is fine as far as small lexical analyzers small lexical analyzer small languages are concerned. However, for professional languages such as c c plus plus java the lexical analyzers are very difficult to write by hand.

So, there is a language and corresponding tool available for describing lexical analyzers. So, lex is such a tool available in unix. So, lex has a language for describing regular expressions, which are at the heart of the lexical analysis. So, you just write down you know all the regular expression, specifications for each of the patterns that we are going to detect in lexical analysis, then it generates a pattern matcher for the regular expression specifications, which are given to the lex tool. And once it is done the lex tool generates programs, which are appropriate for the lexical analysis. We are going to see how this is done.

The general structure of a lex program is that you have definitions we will see what these are then we have what are known as rules and finally, we have user subroutines. Out of these definitions are optional, but rules and user subroutines are essential parts of a lex program are lex specification. And on a unique system how do we use lex to create a lexical analyzer. So, you just type lex e x dot l and it creates a c program by the name lex

dot y y dot c then compile the program using g c c it produces e x dot o and e x dot o is your lexical analyzer which carves out tokens from its input

(Refer Slide Time: 12:01)

LEX Example

```
/* LEX specification for the Example */
%%
[A-Z]+    {ECHO; printf("\n");}
.|\n       ;
%%
yywrap(){}
main(){yylex();}

/* Input */                    /* Output */
wewevWEUFWIGhHkkH              WEUFWIG
sdcwehSDWEhTkFLksewT           H
                               H
                               SDWE
                               T
                               FL
                               T
```

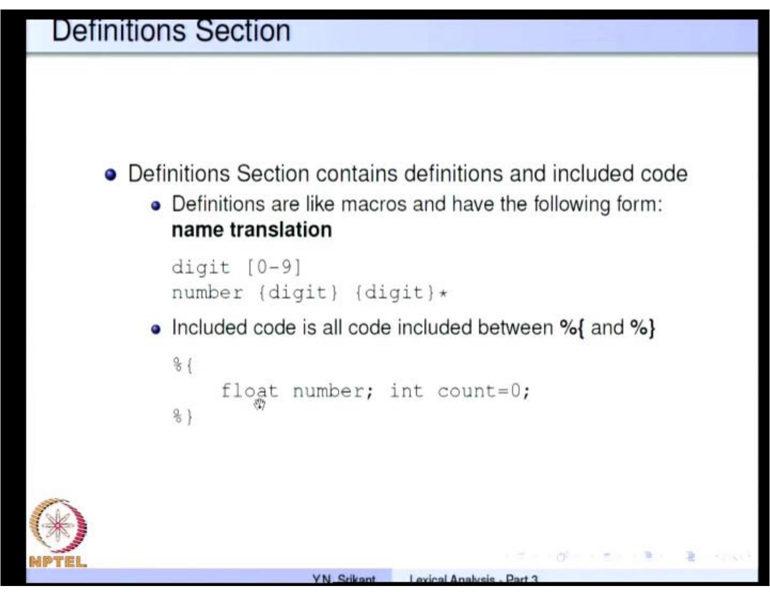
Y.N. Srikant   Lexical Analysis - Part 3

So, now let’s see how exactly a simple lex program looks like and then go into some details. So, here is a comment next specification for the example this is a c style comment and we do not have any definitions in this particular program. So, between these two percentage marks we have the patterns and then we have a little bit tough user written code. So, this is very simple there is a main program which calls lex and there is a y y wrap program, which wraps up the lexical analysis. And in fact, y y wrap hardly has anything a unless we want some files to be closed explicitly, we want we will see examples of this little later.

Coming back to the rules section or the patters section this is a pattern a dash z plus, which stands for you know any of the letters a to z any number of times one hour more. So, this is a set notation a you know a to z. So, all the characters a to z are in that set and a to z plus implies any of these characters any numbers of times once or more times. So, if this pattern is detected then the echo statement generates you know echo’s the pattern, which is rather the text reaches matching this pattern. And then its prints out a new line character dot slash you know bar slash n indicates dot indicates any character, but new line and slash n indicates new line.

So, if in the absence of a match here all other characters are actually matching here and we ignore them. The semicolon is just empty code. So, it ignores all other characters. So, here is a sample. So, we have input here which contains both lower case and upper case characters, but the output filters and produces only the upper case characters that is very easy to see. So, a to z plus matches all these w you know w o e o e etcetera, etcetera. And then for each of the matches it prints out the characters which match and followed by a new line, where and it ignores all other lower case or any numbers characters etcetera, etcetera.

(Refer Slide Time: 14:37)

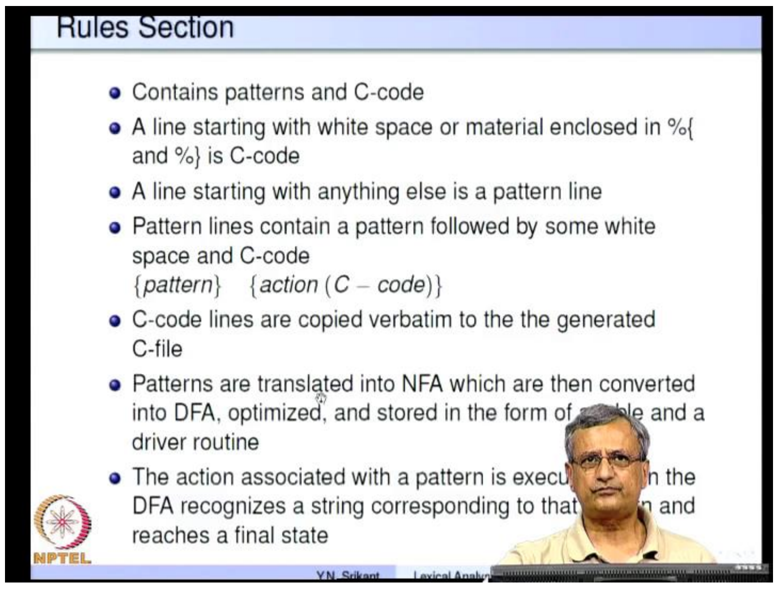


The definition section of the lex program, it contains you know we have already seen regular definitions. So, they are similar to that there are similar definitions written here and there is also some code which can be included for initialization and other purposes. So, definitions are like macros and they are actually like short hands. So, and they have the form name followed by its translation there are two simple examples here the name digit really stands for the set 0 to 9. And the name number stands for you know the digit pattern which is defined here followed by digit star, which implies 0 or more occurrences of digit.

So, here we are writing a regular expression digit digit star and the digit part is defined here. So, these two together define a number as 0 to 9 followed by you know any one of 0 to 9 followed by any 1 of 0 to 9 0 or more times. So, that is the number and this is the

you know definition for that number. So, when we use such definitions it become easier for us to write bigger regular expressions as they are going to see very soon. So, any code for initialization etcetera that be include here any variable that we want to use are all included between this percent bracket and percent bracket. So, that is the initialization part.

(Refer Slide Time: 16:12)



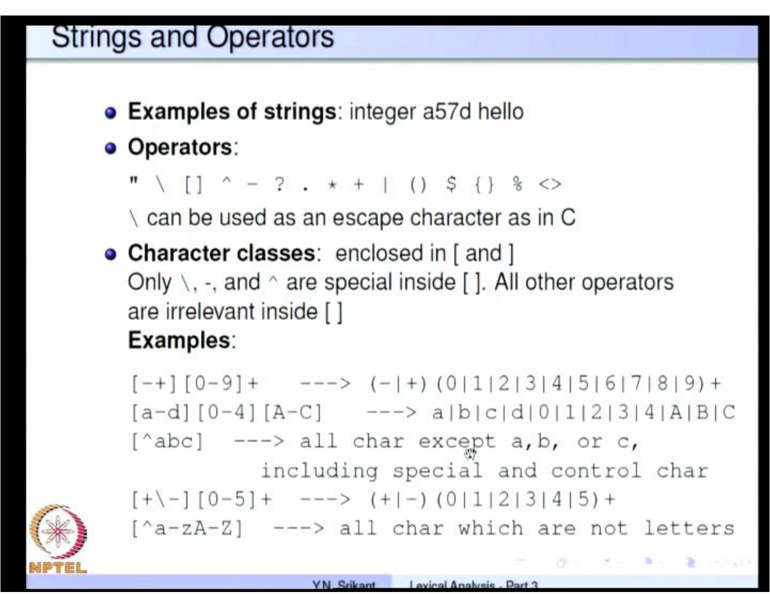
The rules part it contains the is the heart of any lex specification or lex program, it contains patterns and it contains c code. So, a line starting with white space or material enclosed in percent bracket etcetera is c code. So, any c code lines are copied verbatim to the output they are to the generated c file they are not changed in by any fashion. And a line starting with anything else is a pattern line that is inside the rule section. So, pattern lines contain a pattern followed by some white space and then followed by some c code which is optional. So, that is here. So, there is a pattern followed by some white then there is c-code.

So, this c code and the initialization c code are all copied exactly in the same order to the output c code file, there is no change at all. So, what happens for the patterns? The patterns are nothing but regular expressions they are first translated to NFA's then the NFA's are converted to DFA's. So, we have not studied the you know particular algorithm for optimization of DFA, but let me tell you that DFA's can be comprised and optimized. So, that they are compacted rather compacted and optimized, it is the number

of the states of the DFA's the number of transitions they make etcetera can all be made you know optimal. And it so happens that you can find for same language any other NFA or DFA will always be optimized to a particular unique DFA.

So, these DFA's are stored in the form of table and a driver routine as well. So, this is easy to understand the driver routine on a particular state looks up the table and then finds out what the action is and performs that particular action on that symbol. The action associate with the pattern is executed when DFA recognizes a string, which brought you to that particular final state. So, once we reach a final state we may want to announce a token and that can be done with the help of this particular action.

(Refer Slide Time: 18:46)



So, now let us go into details of the syntax of a lex and then take up adequate number of examples to understand them. So, strings in lex are nothing but concatenation of various characters. So, integer when a57d hello these are all examples of strings any symbols which actually you know can be put together and made into a string, but there are some operator symbols which need to be handled in a special way. So, for example, the double cote the back slash the square brackets this caret etcetera, etcetera have a star plus mid this you know bar etcetera, they are all special operators we are going to understand the their work and you know usage as we go on.

So, to begin with this back slash is an escape character. So, if you want to use any you know its usage similar to that of c. So, if you want to use any of these special characters

in your string representation, then you can say back slash question mark or a back slash dot to get that particular dot as a character into a string, character into as a string. Now very important notations is tough character classes. So, we have left brackets square bracket and the right square bracket anything with in enclosed in between is character classes. So, in side only you know the back slash the dash and the caret have special meaning all others are just characters inside the set notation.

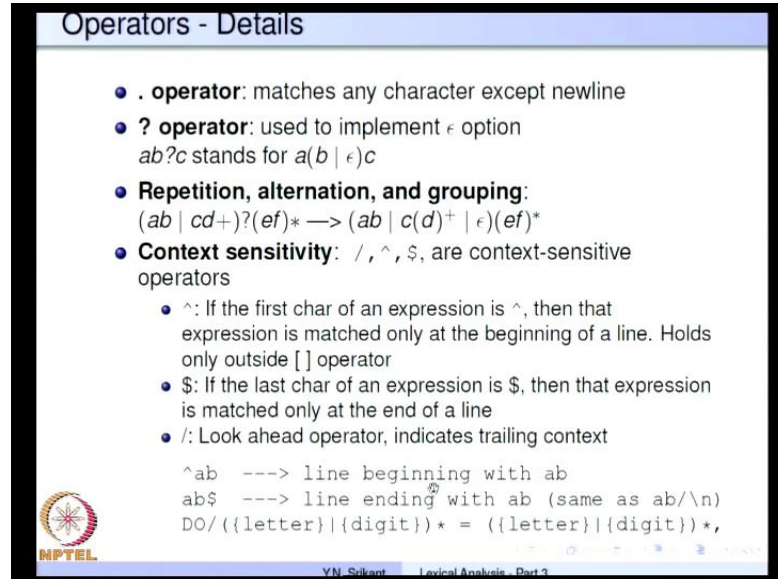
So, for example, the notation minus plus in one square bracket pair and another pair contain 0 dash 9 plus. So, this stands for minus plus is set containing minus or plus and 0 to 9 plus is set containing 0 or 1 or 2 etcetera and up to 9 followed by plus this is a regular expression as we can see. So, it is start with whether it’s minus or plus followed by you know any of the digits any number of times.

So, similarly a to d 0 to 4 a to c. So, this says any character a to d followed by any of the let character, you know any of the character 0 to 4 and finally, any of the characters capital A to capital C. So, this is the set of characters corresponding to this regular expression notation, this notation says the character a b c inside the square brackets. So, this simply says its complement operator all characters except a b or c. So, a, b, c as usual stands for a or b or c and caret front complements inside it says all characters a except a or b or c it including all special and control characters as well two more examples plus back slash minus followed by 05 plus.

So, I said this back slash really stands for the black slash minus really stands for the minus operated itself. So, it is either plus or minus. So, here difference is the minus started in beginning. So, it was actually understood as the minus character, but here we wanted to make it that second character, which cannot be done because it is use to you know ah used as when it is used as when it is used as second character that should be third character as well. So, get right of this problem we used the back slash to put the order of the put plus to the minus characters and the 0 to 5 plus is 0, 1, 2, 3, 4 or 5 plus.

So, again caret a to z and a to z says all characters which are not letters. So, this is a to z is all lower case characters capital A to Z is all says as upper case characters and character complements them. So, it all characters which are not letters so; that means, you know all the digits special characters etcetera or all included in the set.

(Refer Slide Time: 23:03)



There is dot operator, which matches any character except newline and question mark operator is used to implement the epsilon or null string option. For example, a, b question mark c stands for a followed by b or epsilon followed by c then we also have repetition alternation and grouping characters. So, a b bar c d plus question mark e f star. So, this regular expression, which stands for. So, observe that we have used ordinary parenthesis not the square brackets.

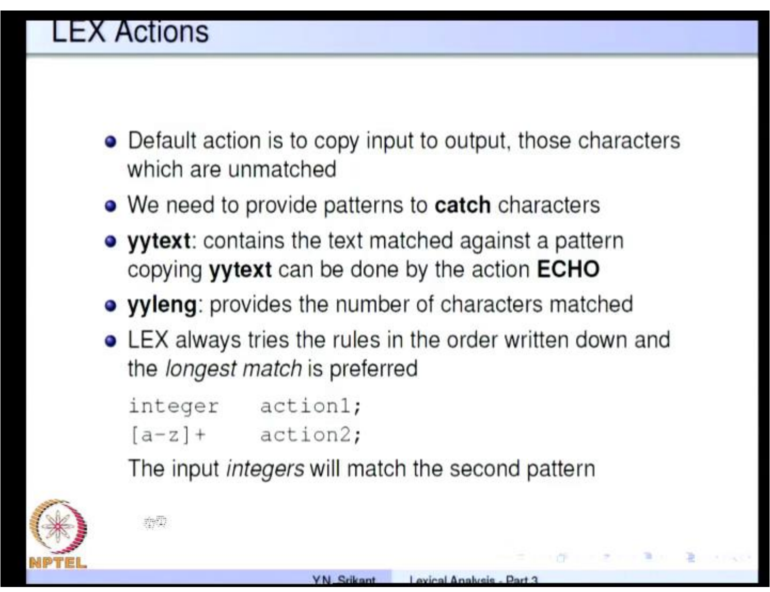
So, a, b is the string a b, c is the string c d plus stands for the d any number of times, then the question mark says epsilon place of any of the that is bar epsilon and then e f star says repeated number of times. So, that is e f star. So, this is the regular expression notation in the ordinary form, where as this is the lex notation. There are context sensitivity operators as well slash then and the caret and dollar. So, if we use caret as the first character of the first expression, then the expression is matched only at the beginning of the line. So, for example, if we say caret a b is the pattern on its own remember caret character inside the square bracket as the different meaning, this is outside the square bracket.

So, this means line beginning with a b. So, it is matched only if the line starts with a b, a b dollar the dollar is end of the line end of you know it matches to the last ends with the a b. So, dollar if the last character of an expression is matched only the end of the line. So, then the look ahead operator is little more complicated we want to say that d o is the

pattern it should be match provided there is a pattern following it string matches this particular pattern. So, that is letter or digits star equal or letter or digit star comma. So, if this pattern entire big pattern mattes after the d o then the d o is a pattern, where which string which is to be where pattern which is to be matches if not otherwise.

So, in that in the sense slash operator is a look ahead operator which it looks at all the all the pattern, the pattern which is following, but which does not consumes the symbols which match the following pattern. It consumes only two characters which is d o the rest i is going to be i u know match again after the d o match is finalized.

(Refer Slide Time: 25:54)



What are the actions of lex? For example, the default action is to copy input and output. So, those characters which are unmatched. So, if there is no match. So, just copy to the output. So, we need to provide patterns which really catch characters that is our purpose and what is caught that retend the buffer y y text. So, echo actually empty’s is the y y text into the output y y length contains the number of characters which are matched and whose character string is present in y y text.

So, lex always tries the rules in the order written down and the longest match is preferred. So, we remember we discussed this longest match requirement in the lexical analyzer and transistor you know the transition diagram part. So, for example, here this is a word integer and here is a regular expression corresponding to any lower case character

a to z plus . So, corresponding to the integer action one and corresponding to the second one is action two.

So, if you have a minor variation and we say the integers then really speaking integer there are several possible matches integer matches the first part. And s matches the second part it is also possible to matches the entire thing integers in the second with the second pattern as well. And since the longest pattern is always used it this is the pattern which is pattern actually which is matches to the our input integers.

(Refer Slide Time: 27:38)

LEX Example 1: EX-1.lex

```
%%
[A-Z]+     {ECHO; printf("\n";}
.|\n       ;
%%
yywrap(){}
main(){yylex();}

/* Input */                     /* Output */
wewevWEUFWIGhHkkH               WEUFWIG
sdcwehSDWEhTkFLksewT            H
                                H
                                SDWE
                                T
                                FL
                                T
```

NPTEL Y.N. Srikant Lexical Analysis - Part 3

So, now let us start looking at several examples to understand how exactly lex performs its matching. In fact, the files which contains this all programs match and here X 1 2 and etcetera, etcetera. So, all the available in NPTEL repository and you can download them compile them and execute them to see whether the output generated properly try with different variations so on and so forth. So, for example, this is program which already saw. So, let us on discuss with once more it captures all the upper case characters in the input. So, a to z plus is printed out along with a new line. So, this is the input. So, all the upper case characters are printed out along with new lines that is really simple straight forward program.

(Refer Slide Time: 28:42)

LEX Example 2: EX-2.lex

```
%%
^[ ]*\n
\n    {ECHO; yylineno++;}
.*    {printf("%d\t%s",yylineno,yytext);}
%%

yywrap(){}
main(){ yylineno = 1; yylex(); }
```

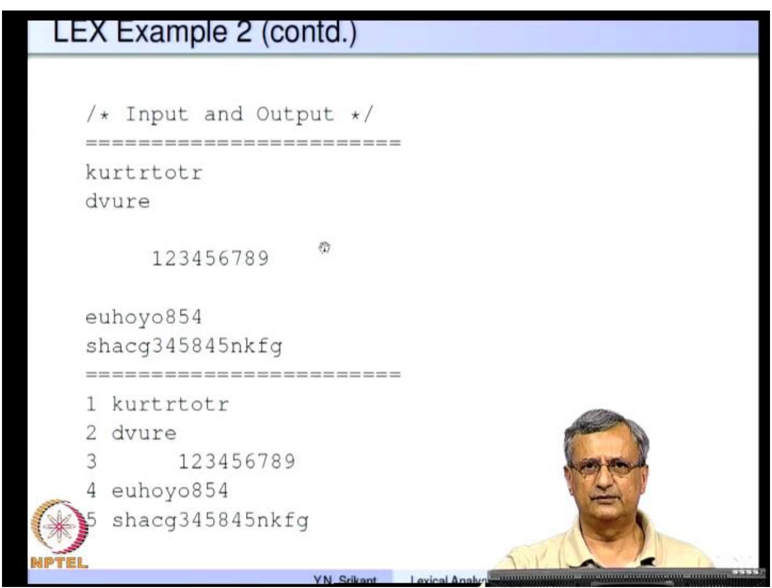
So, let us take a another simple example. So, the first pattern says you know beginning with any number of blanks 0 or more blanks followed by a new line. So, this is the pattern. So, in other words the simple explanation is all blank lines. So, they may contain a blank space character or they may contain only a new line character or they may contain several blank spaces followed by new line, but in the sense they must all be blank lines completely blank lines without containing any other character. So, if this is the pattern is called pattern this is ignore this is no action corresponding to it, then if they remember the ordering of these as well.

So, if it is a new line character, the new line is a echo and variable y y line number is incremented. Any other character apart from new line dot star any number of them is called in the third pattern and when this matches it prints out y y line number corresponding to that particular line that particular time and y y is a text, which matches the patterns dot star. So, this is printed out. So, the y y where have the patterns does nothing it just you know its require for to complete some formality and in the main program we initialize y y line number to one and call y y lex. So, the way this operates the in the input all the blank lines are ignored.

And then since the longest match is suppose there is a reject non blank lines containing other character as well as along with blank, then there would match this, but the new line will not match as dot star as I told you dot matches only all characters accept new line.

So, all the characters all the line which contain non blank character will match this first for all the non blank character accept new line and this next further and new line character. So, between this to where printout the character and also the line number. So, let see how the input is transform to an output.

(Refer Slide Time: 31:05)



So, here is a non blank line this is another non blank line then we have a blank line then there are a few blank spaces followed by non blank character, another blank line non blank line and another non blank line. So, over all there are 1, 2, 3, 4 and 5 non blank lines. So, we have 1 the first non blank line 2 flowed by the second blank non blank line three the third, four the fourth and finally, 5 the 5. So, remember the blank character which are here you know in the input are also copied, what is important is that they contain non blank characters. Suppose we had after this a couple of blanks followed by a b this particular line would still be written on at as all this followed by 2 blank or 3 blank whatever is the present in the input followed by a b.

So, the blanks in the non blank line are copied as they are, but if the input is a complete blank line. For example, forward line is here and the fifth line here we are complete blank we are not copied to the output they are ignored. So, this is what this was supposed to do, all the non blank line are caught here and the new line caught here, but all blank character blank line caught here.

(Refer Slide Time: 32:37)

LEX Example 3: EX-3.lex

```
%{
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}","{blan
                 {blanks}{declpart}{
declaration (("int")|("float")){blan
                 {decllist}{blanks};
```

So, now the example becomes little more complicated. So, here we have a code part in which we describe you know there is a declaration called file star decifile. So, we declare a file variable for use our action later on and then we have a remember the this declaration is a user written code and it is enclosed in this 2 special markers, following that are a number of declaration rather patterns.

(Refer Slide Time: 33:30)

LEX Example 3 (contd.)

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

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

So, finally, let us go to the next first the in the role section what we have is a regular expression called declaration, which is explained in the previous slide we will come go

back to it in a minute. So, whenever a declaration is caught it is printed out to a file called decifile which we have already declared and what is printed out the corresponding text, you know which was matched. In the y y wrap routine we just call f close to close the declaration decifile variable and in the main program we open the decifile for write purposes then call the lexical analysis. So, let us look at the output and go back to the patterns after that.

(Refer Slide Time: 34:23)

LEX Example 3: Input, Output, Rejection

```
wjwkfblwebg2; int ab, float cd, ef;
ewl2efo24hg2jhrto;ty;
int ght,asjhew[37],fuir,gj[45]; sdkvbwrkb;
float ire,dehj[80];
sdvjkjkw
==========================================
float cd, ef;
int ght,asjhew[37],fuir,gj[45];
float ire,dehj[80];
==========================================
wjwkfblwebg2; int ab,
ewl2efo24hg2jhrto;ty;
 sdkvbwrkb;
sdvjkjkw
```

Y.N. Srikant Lexical Analysis - Part 3

So, this is the input this is the you know matched output and this is the rejected input. So, as you if look at the matched input which is printed as the output. So, we have a float c d coma e f right then we have I n t g h t coma a s j h e w 37 coma f u i r coma g j 45 followed by semi colon. Then we have a float i r e coma d e h j 80 semicolon. Now, let see what happened what is present in the input the number of you know variable followed by semi colon, then there is I n t a b coma float c d coma e f and then another set of characters another i n t g h t etcetera.

So, in this input which seemingly seems to be meaningless, there actually some c type declaration. So, the propose of the program that now going to discuss is to extract this meaningful c declaration from this seemingly meaningless input and rest of it is ignored. So, this is our excise. So, let see how we can do it.

(Refer Slide Time: 35:50)

LEX Example 3: EX-3.lex

```
%{
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};
```

So, there are a number of short hand notation that we write down for example, blanks. So, blank or tab observe the you know back slash escape character for tab. So, blank or tab any number of times letter is a any lower case letter a to z that is the notation for set any way the digit short hand is 0 put a 9, then identifier would be a letter or an underscore followed by letter or digit or underscore any number of time. So, this is a standard you know pattern for identifier that we you know of any later followed by later or digit star, but we have also included the underscore here, then we have a number which is one or more digits. So, digit plus this is the regular expression notation digit digit star.

Then we permit a array declaration part which says an identifier followed by the left square bracket followed by a number followed by right square bracket. So, observe that this is a pattern, which uses pattern declared before. So, in other words for the array declaration part we must have a name followed by a number. So, that exactly that we have in our output here name followed by a number name followed by a number etcetera. So, that is the array declaration part and then the short hand notation declaration part. So, either in array declaration or a simple name. So, the simple name is you know any name that we have already seen here.

For example, this g h t i r e d e h etcetera are all you know yeah sorry f u i r etcetera are all simple names. Then the declaration list basically it is a number of declarations. So,

declaration part followed by any number of blanks then there was be a coma, any number of blanks again and then you know followed by this entire thing declaration coma is repeated any number of time including 0 followed by declaration part. So, in other words we want to generate a list of declarations. So, this help us in doing that when a complete declaration would be beginning with a result word i n t or float followed by a declaration list. So, this gives us something like this you know float c d coma e f i n t g h t a s j h e which 37, etcetera, etcetera.

So, this is the int this is they are the declaration list containing either a name simple i d or an array declaration part. So, separated by this semicolon and ending with a sorry separating by a coma and ending with a semicolon. So, each of this is a repeated in any number of times. So, there is a one declaration is here another here third here. So, you could have as many of them as you wish. So, that is our declaration and then we have a you know the pattern in the rules section which is a declaration. So, here this is what we are looking at this is the declaration that we know of. So, any number of them. So, declaration whenever there is a match for the declaration it prints out that particular declaration.

So, observe here each declaration has int or float followed by a declaration list. So, this is one declaration. So, whenever this declaration is followed by a of course, a semicolon here. So, whenever this declaration is matched in the input, it prints out that particular declaration into the declaration file all others are a characters which are ignored. So, now, let us see how it works here. So, this sequence you know is a sequence of characters it is a an identifier, but then it suddenly ends with a semicolon we are going to start processing it as a declaration only when it starts with int or float.

So, this part is ignored. So, this starts very promisingly with an int then there is a an identifier also and then there is a comma as well, but then suddenly you know instead of a semicolon we have a float. So, therefore, even this sequence of characters is ignored and copied into the rejected part whereas, again it starts with a float. So, a new declaration pattern is you know going to start here and this happens to be a correct declaration because it is just float c d comma e f semicolon. So, that part is separated out as a declaration. So, after the semicolon it ends. So, we start processing a new declaration if possible.

So, this entire thing does not correspond to a declaration because it does not begin with int this begins with int. So, int g h t a s j h e w 37 f u i r comma g j 45 and semicolon this entire thing is a valid declaration, it is copied to the output as valid this is in you know invalid. So, it is rejected this is valid. So, it is copied here and this is invalid. So, this is also rejected. So, the example really shows that for processing or catching declarations in a programming language you know you can actually write lex programs for it.

Even though we are going to do this more meaningfully using yacc a little later the that is after we learn parsing, we will see how to do all this using yacc which is actually a more meaningful way. But it is not as if lex is less powerful lex can do quite a bit of work, it can actually catch all the declarations in the program etcetera that is simply because the declaration part of a program is really a regular in nature you know is this corresponds to a regular language. Whereas, when we get into a nesting of statements and records etcetera it does not happen to be regular language anymore, and we will require a more complicated machinery.

(Refer Slide Time: 43:27)

LEX Example 4: Identifiers, Reserved Words, and Constants (id-hex-oct-int-1.lex)

```
%{
int hex = 0; int oct = 0; int regular =0;
%}
letter          [a-zA-Z_]
digit           [0-9]
digits          {digit}+
digit_oct       [0-7]
digit_hex       [0-9A-F]
int_qualifier   [uUlL]
blanks          [ \t]+
identifier      {letter}({letter}|{digit})*
integer         {digits}{int_qualifier}?
hex_const       0[xX]{digit_hex}+{int_qualifier}?
oct_const       0{digit_oct}+{int_qualifier}?
```

NPTEL Y.N. Srikant Lexical Analysis - Part 3

Now, we move on to the next example that we have that is the you know how do we combine identifiers reserved words, then hex constants oct constants, normal integers etcetera. So, I told you that this a fairly intricate problem when we dealt with transition diagrams. So, let us see how you know you are writing a lex specification simplifies this problem and makes it easy for us to write the specification. So, this lex program also has

an initialization code. So, we have a variable hex which is initialized to 0 variable oct which is initialized to 0 and variable regular, which is initialized to 0 then we have a host of these patterns we will discuss them shortly.

(Refer Slide Time: 44:25)

LEX Example 4: (contd.)

```
%%
if           {printf("reserved word:%s\n",yytext);}
else         {printf("reserved word:%s\n",yytext);}
while        {printf("reserved word:%s\n",yytext);}
switch       {printf("reserved word:%s\n",yytext);}
{identifier} {printf("identifier :%s\n",yytext);}
{hex_const}  {sscanf(yytext,"%i",&hex);
     printf("hex constant: %s = %i\n",yytext,hex);}
{oct_const}  {sscanf(yytext,"%i",&oct);
     printf("oct constant: %s = %i\n",yytext,oct);}
{integer}    {sscanf(yytext,"%i",&regular);
    printf("integer : %s = %i\n",yytext, regular);}
.|\n ;
%%
yywrap(){}
int main(){yylex();}
```

YN Srikant Lexical Analysis - Part 3

So, here I have a host of these patterns corresponding to reserved words and then the other constants as well. And finally, you know a main program which calls y y lex.

(Refer Slide Time: 44:40)

LEX Example 4: Input and Output

```
uorme while
0345LA 456UB 0x786lHABC
b0x34
========================
identifier :uorme
reserved word:while
oct constant: 0345L = 229
identifier :A
integer : 456U = 456
identifier :B
hex constant: 0x786l = 1926
identifier :HABC
identifier :b0x34
```

YN Srikant Lexical Analysis - Part 3

So, let us also see the input and output and understand it before we discuss the patterns. So, obviously you this is the input and this part is the output u or me happens to be a

simple identifier it is caught as an identifier. While is a reserved word. So, it is printed out as a reserved word we have 0, 3, 4, 5, l a. So, out of this it actually the 0 3 5 for 3 4 5 l happens to be an octal constant 229, but the A part is a simple identifier which is printed out as an identifier. Similarly we have 456 u b the 456 u part is a normal integer 4 5 6 and the b is an identifier b, here is a hex constant 0 X 786 you know l.

So, this much is a hex constant 1 9 2 6 and then the h a b c part is an identifier. And now watch this we have b followed by 0 X 34 here it is not recognized as b an identifier followed by a hex constant because of the longest match characteristic, b followed by this entire sequence of 0 X 34 is actually matched against an identifier and it is denoted as an identifier. So, you know this is how the longest match is useful in identifying various tokens. So, let us go back to the patterns and study them in full.

So, here is a letter which is already well known a to z or a to z or followed by and you know or underscore. So, any little a lower case or upper case character or underscore is a letter for us digit is 0 to 9, digits is a digit any number of times one or more digit octal is a just 0 to 7, 0 1 2 3 4 5 6 or 7 digit hex would be 0 to 9 or a to f. So, any of these are hexa digits integer qualifier is a u capital U lower case l or capital L blanks as usual blank or tab any number of times one or more now. Now, we look at identifiers. So, letter followed by letter or digit star. So, this a well known regular expression for identifiers, then we have a integers digits followed by integer qualifier optional. So, this question mark remember implements epsilon.

So, this is really speaking integer qualifier or epsilon digits followed by integer qualifier or epsilon would make up an integer x constant is 0 followed by either this is remember this is square bracket x X square bracket. So, either little X or bigger X followed by hexa digits any number of times once or more followed by integer qualifier either you know or epsilon. So, we have any 0 followed by X followed by any number of X digits of course, integer qualifier is a optional octal constant similarly is 0 followed by octal digits any number of times followed by integer qualifier optional.

Then we have the reserved words if else while and switch. So, now, we come to the rules section with a percent percent. So, for the 4, 5 reserved 1 2 3 4, 4 reserved words the action is simply print out the reserved word and then the identifier. So, since we have

written down the reserved words and then the identifier, the characters which correspond to these will never be matched against identifier.

Whereas, if we have a word such as switches then even though part of it matches here it will be the longest match will be that of an identifier. So, if we place the identifier before i f that is the reserved words here be sure to try it out once, these four will never match everything matches against the identifier. Now, the hexadecimal constants when they are caught the s scan f is used to read the integer inside the hexadecimal notation and print it out along with the text. Similarly octal constants are converted to decimal constants using the scan f function and print it out integers are also converted to the normal integer the characters are converted to normal integer and print it out. Finally, any other character which is caught is ignored. So, dot or new line they are all ignored. So, this is what we just now went through. So, the patterns are all caught and the output is generated.

(Refer Slide Time: 50:09)

LEX Example 5: Floats in C (C-floats.lex)

```
digits          [0-9]+
exp             ([Ee](\+|\-)?{digits})
blanks          [ \t\n]+
float_qual      [fFlL]
%%
{digits}{exp}{float_qual}?/{blanks}
        {printf("float no fraction:%s\n",yytext);}
[0-9]*\.{digits}{exp}?{float_qual}?/{blanks}
        {printf("float with optional
                 integer part :%s\n",yytext);}
{digits}\.[0-9]*{exp}?{float_qual}?/{blanks}
        {printf("float with
                 optional fraction:%s\n",yytext);}
.|\n          ;
%%
yywrap(){} int main(){yylex();}
```

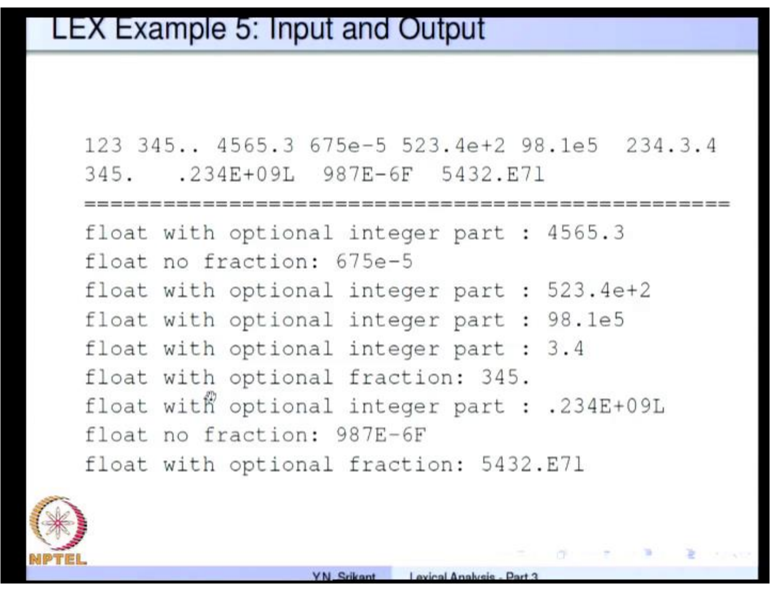
NPTEL

Y.N. Srikant   Lexical Analysis - Part 3

So, let us look at the floating point numbers which are a little more complicated than normal integers. So, as usual we have lexical analyzer program for floats in c the digits are 0 to 9 plus and then we have an exponent, which is either big E or small e followed by plus or minus, it is optional the sign is optional followed by digits. So, if we write e plus 9 or e 9 it amounts to the same number, same exponent then the blanks blank tab or

new line any number of time and floating qualifier is f F l L, right? So, then there are the patterns. So, let us look at the output and then go back to the pattern it is self.

(Refer Slide Time: 51:02)



So, 1 2 3 then three 45 dot dot 4565 dot 3. So, really the first output is generated here because we do not catch pure integers, we do not catch anything, which does not have a blank after a dot. So, this is ignored. So, 4565.3 is you know optional integer part. So, this could have been a dot three as well 675 e minus is float with no fraction then 523.4 e plus 2 again with a optional integer part. Next one is also with optional integer part 234.3.4 corresponds to 3.4234 dot should have had a blank after that. So, it does not. So, that is ignored.

345 dot has a blank so that is caught here as optional float with a optional fraction then dot34e plus 09L is caught as an optional integer you know possibly no integer part here. And similarly 987 is without fraction and the last one with optional fraction. So, let us study the patterns here. So, these are the first part first one is digits followed by exponent followed by optional floating qualifier followed and then followed by blanks. So, remember the look ahead operator. So, this pattern is valid provided it is followed by blanks, otherwise it is not valid. So, this will match float with no fraction. So, digits followed by exponent. So, this is and this of course, is optional.

So, this corresponds to floats with no fraction. So, float with no fraction is here 987 e minus 6 f here we have a number of digits you know 0 to 9 star 0 or more time iterations

here followed by a dot compulsorily. So, before the dot there can be an integer part or no integer part. So, this is optional integer part, but once there is an optional integer part we must definitely have digits followed by an optional exponent followed by an optional floating qualifier follow and of course, it matches only if there are blanks after that.

So, the integer part is optional, but the rest is compulsory at least the digits part is compulsory exponent is optional, here we have digits followed by dot followed by the optional fraction 0 to 9 star, optional exponent optional floating qualifier, but followed by a blanks compulsorily. So, these are the various patterns which match here. So, one with no fraction, second one with optional integer part, third one with optional fraction. So, these are the various outputs that we already looked at…

(Refer Slide Time: 54:23)

LEX Example 6: LA for Desk Calculator

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

NPTEL

Y.N. Srikant   Lexical Analysis - Part 3

So, let us look at the another example which is used with the desk calculator as a lexical analyzer to generate a lexical analyzer for the desk calculator, which we are going to study later, number is 0 to 9 plus followed by dot optionally. So, this is dot or a epsilon followed by 0 to 9 star r sorry this is a r. So, this is the bar here which gives you two options. So, either 0 to 9 plus followed by a dot or 0 to 9 star followed by A dot then followed by a number of digits.

So, the difference between the two is the integer part here is compulsory where as the fraction is not here the fractional the integer part is optional, but the fractional part is compulsory you cannot have both of them as optional, otherwise you would have just a

dot. Name is as usual a to z a to z followed by a to z a to z 0 to 9 star. So, this is the identifier part. So, when a number is caught it is converted to a number using s scan f and it returns a token called number, the value actually you know is here you know. So, that is read into a variable called y y l val dot d val which is a parser variable. So, I am introducing this example just to show how interfacing with yacc happens.

So, yylval is a parser variable used by yacc and this value of number is put into that variable. Whenever there is a name caught it is looked up in a symbol table and if the name is present you know that pointer is return to is otherwise if the name is absent in the symbol table, it is entered into the symbol table and the token name is returned. For plus plus we return post plus minus minus we return post minus for a dollar we return a 0 and new line are any other character, we just return that simple character itself. So, this is just to show you that it is possible to do some symbol table operations along with name search etcetera inside a lex program as well. So, lex programs can become as technical and as complicated as we want them to be. So, with this background let us stop the lecture in the next lecture, we will start studying parsers.

Thank you.