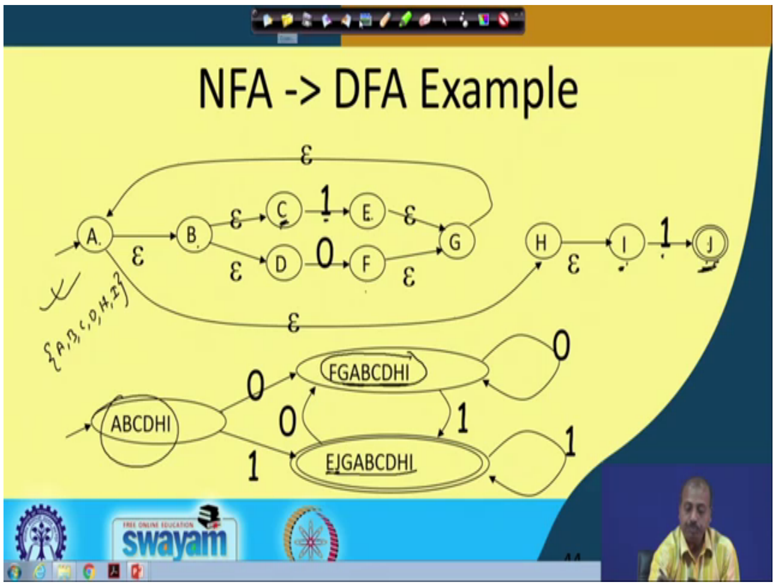
Compiler Design
Prof. Santanu Chattopadhyay
Department of E & EC Engineering
Indian Institute of Technology, Kharagpur

Lecture – 12
Lexical Analysis (Contd)

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

(Refer Slide Time: 00:41)


NFA -> DFA Example

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

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

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

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

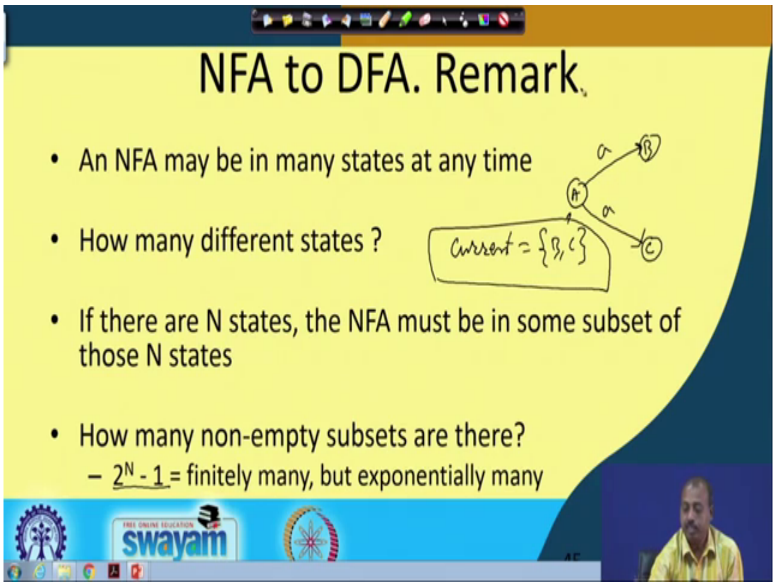
Now, from this state or for let us come back to this F G A B C D H I now what happens on a 0. So, since 0 transition is defined only from D to F so, from D it can go to F and once you are in F from epsilon transition so, you can come to G from G you can come to A. So, essentially it gives rise to the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

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

So, if this is the same state that is created for the DFA. So, this way we can convert one NFA into DFA using some algorithm. And you see that whether this DFA is more complex or not than that NFA so, that depends on the type of regular expression that we are considering. In this case it is not complex because, now I have got only 3 states in the DFA compared to 1 2 3 4 5 6 7 8 9 10 states in the NFA.

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

(Refer Slide Time: 06:19)



NFA to DFA. Remark

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

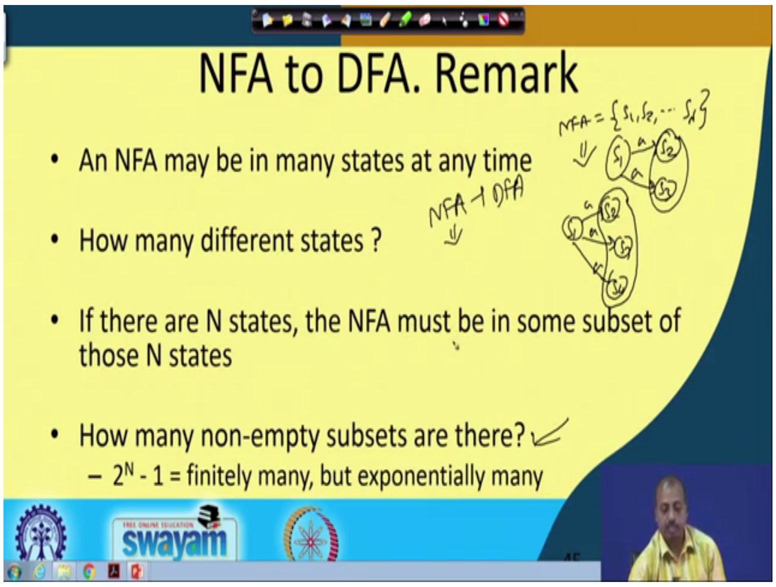
a
a
A
B
C
Current = {B, C}
swayam

Some remarks on this NFA to DFA conversion and NFA maybe in many states at any time. So, what I mean is that suppose I have got a transition in the NFA from one state a there is a transition on input symbol A to state B and another transition on input symbol a to state C. So, if you are simulating this NFA and you are currently at state A if your current state is A and the next input symbol is a small a, then the next the current state becomes the set BC.

So, this is the meaning that you have if you try to stimulate the NFA then you have to consider in terms of current set of states in which the system may be there the NFA the auto motor may be there. So, then NFA may be in many states at any time, but how many different states if there are N states the NFA must be in some subset of those N states and how many subsets can be there?

So, there can be 2 to the power N minus 1 finitely many, but exponentially many subsets because it can do many of those any of those subsets like. So, any of these like if I have got say what I want to mean is suppose my NFA has got set of states S 1 S 2 up to S N.

(Refer Slide Time: 07:47)



NFA to DFA. Remark

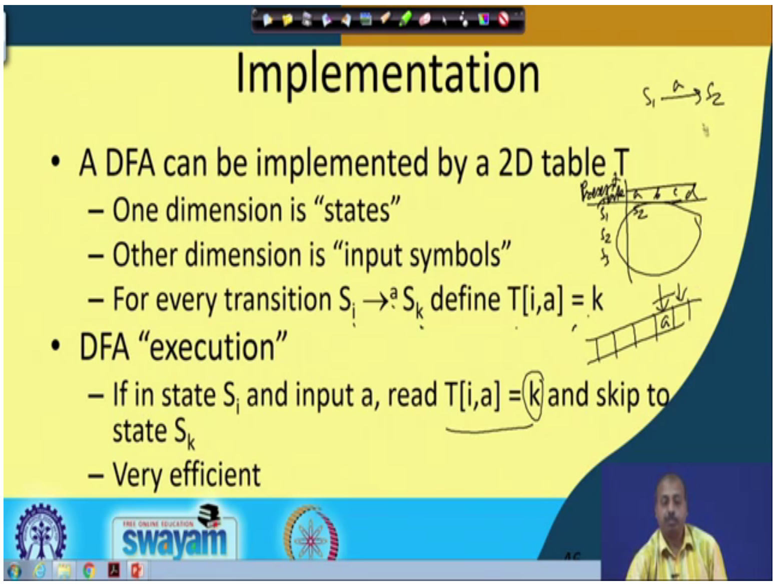
- An NFA may be in many states at any time
- How many different states ?
- If there are N states, the NFA must be in some subset of those N states
- How many non-empty subsets are there?
  - $2^N - 1$ = finitely many, but exponentially many

Now, while you are doing the transitions so, it can be in any of the subsets because from S 1 it may be from S 1 it may go to it may be going to S 2 on a and it may be going to another state S 3 on a or it may be that on S 1it goes to S 2 S 3 and S 4 on a.

So, this way so, these are the different subsets. So, S 2 S 3 becomes the subset than S 2 S 3 S 4 becomes the subset so, these are the possibility. So, in the worst case so, you can have 2 power N minus 1 different subsets that are possible and it can go to any of those states any of those state of state of subsets of states on a particular input symbol. So, and in fact, when you are doing this conversion from NFA to DFA so, what we are essentially doing is that we are constructing the equivalent set of equivalent subsets of the NFA.

And then for each of these set of sub subsets so each of these the subset of states so, we are calling it to be a state in the DFA. So, DFA that is why DFA number of state can become exponential. So, in this case so, this can happen. So, you can it can give rise to exponentially many number of a subsets so, that create that makes it difficult for simulating the NFA.

(Refer Slide Time: 09:29)



Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbols"
  - For every transition $S_i \to^a S_k$ define T[i,a] = k
- DFA "execution"
  - If in state S_i and input a, read T[i,a] = k and skip to state S_k
  - Very efficient

Whereas, for DFA we do not have this problem a DFA if you are trying to implement it can be realised by a two dimensional table T one dimension is the states and the other dimension is the is the input symbol. So, what I say is that you maintain a table for a DFA that this is the current state or present state this is the present state and you have got the input symbols say this a b so, these are say c d suppose I have got 4 input symbol.

So, these are the different states S 1 S 2 S 3 etcetera. So, you just note down on say from Si to Sk say from S 1 to S 2 there is a transition on input symbol a. So, I write here the next state is S 2. So, accordingly we define that in this table i k i a equal to k that is from state i on getting input a it will go to state k. So, I can maintain a 2 dimensional table where this T i a equal to k or this. So, a table corresponding to the state S 1 and input a is S 2.

So, this is how I can represent the DFA and for the DFA execution for simulating the DFA if we are in state Si with an input is a with a read Ti a equal to a and skip to the state Sk. So, so, this is how the simulation will be done. So, we I we also have got an input string so, this is the input string that we have and at present the input pointer is somewhere here.

So, if this is the this symbol is a at the present state is Si and we will consult this table and find out what is the entry for Tia in the table. So, and if the entry is say k then we will we tell we will update the current state to state k and we will advance the input pointer to the next position. So, that we can have a very efficient algorithm for simulating the DFA where as for simulating NFA you need to remember the current set of states. So, there can be a number of present states at which the system may be there so, it makes it difficult.

(Refer Slide Time: 11:47)

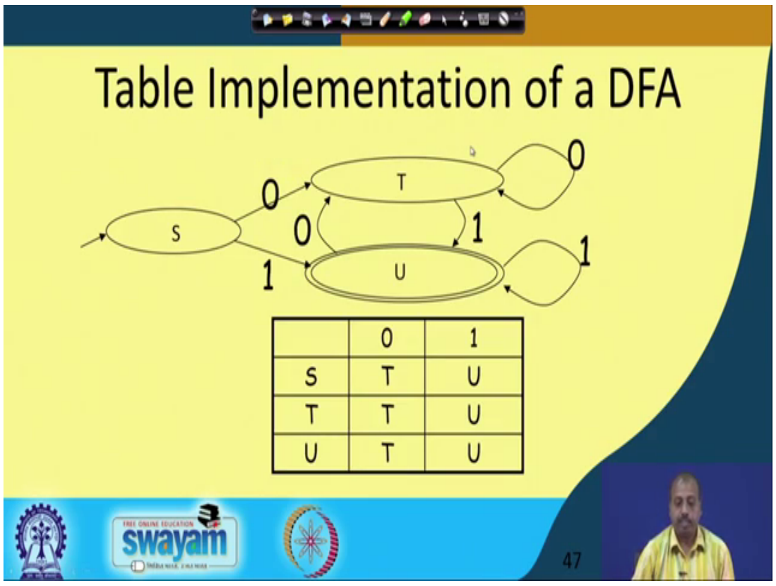


Table Implementation of a DFA

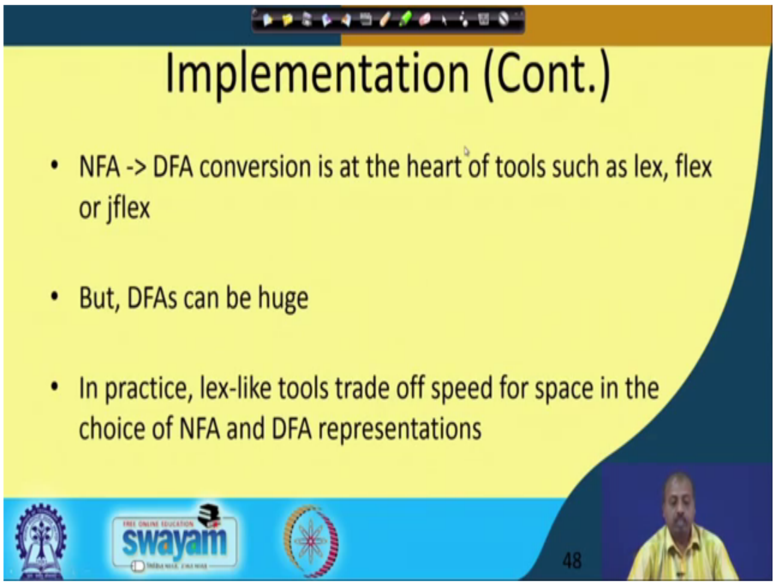
$$\begin{tikzcd} \arrow[r] & S \arrow[r, "0"] \arrow[dr, "1"'] & T \arrow[loop right, "0"] \arrow[d, bend left, "1"] \\ & & U \arrow[u, bend left, "0"] \arrow[loop right, "1"] \end{tikzcd}$$

	0	1
S	T	U
T	T	U
U	T	U

So, for example, this is the DFA that we have constructed and we if we name the states as S T and U S T and U so, I can have a table like this. So, there only in my alphabet there are only two symbols 0 and 1. So, from state S on input 0 it goes to state T on say input 1 it goes to state U.

Similarly, from the state T on input 0 it remains in state T on input 1 it comes to state U from state U on input 0 it remains in the state it comes to state T and on 1 it remains in state U. So, this way I can have a very simple table that can represent the DFA and then this DFA that can be simulated as we have discussed previously.

(Refer Slide Time: 12:39)



Implementation (Cont.)

- NFA -> DFA conversion is at the heart of tools such as lex, flex or jflex
- But, DFAs can be huge
- In practice, lex-like tools trade off speed for space in the choice of NFA and DFA representations

So, NFA to DFA conversion is the heart of tools such as lex flex or jflex. So, this lex flex and jflex so, they are some automated tools that have been developed by compiler designers. So, even these they come as some utility in many of the operating systems. So, they were originally proposed to come with the UNIX operating system and later on many other operating system. So, so they have integrated them as some tools for system development. So, if you are trying to develop a compiler for a language so, if the lexical analyzer part can be designed using these tools.

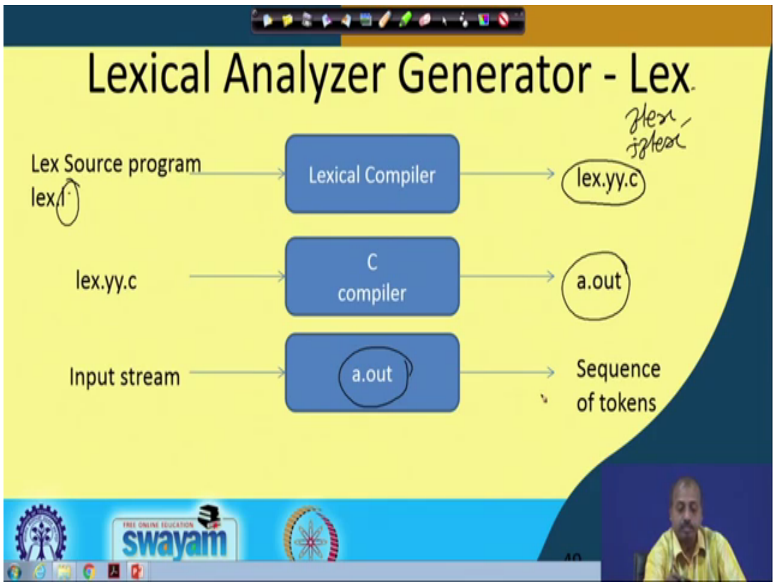
And these tools so, they have the capability to convert NFA to DFA. So, they first take the regular expressions for the language for which you are going to get the lexical analyzer and then convert those regular definite regular expressions to their corresponding NFA and convert those NFA’s to DFA.

However, the DFA‘s can be large can be huge because them and they normally it is produced as a text file as a C program or C + + program or java program and the program size becomes very large and it is very difficult to understand like what is there in the code, but we most of the time we take it blindly that this is the code produced is correct and since the tools are running for quite some time over the years.

So, they are time tested so, there is very I should say every little chance that there are some bugs in those tools. So, we can use those tools with a lot of faith on them and most of the time what happens is if there is some problem the problem is due to the regular expressions that we have written. So, there is some problem in the regular expressions themselves.

So, this lex like tools so, the trade off speed for space in the choice of NFA and DFA representation. So, we have there is a trade off between space and speed of operation. So, they do this a they do this a trade off to see like what can be a reasonably good speed and reasonably good representation the NFA or DFA.

(Refer Slide Time: 14:47)



Lexical Analyzer Generator - Lex

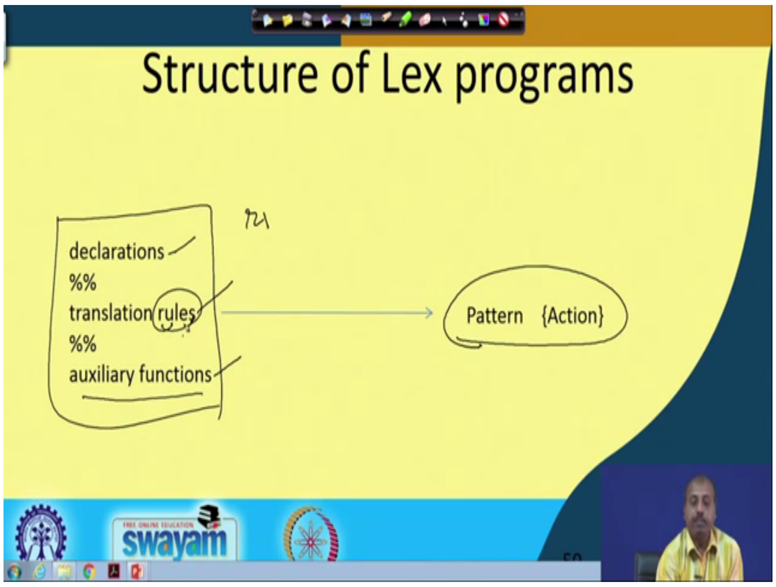
So, next we will look into one particular tool lexical analyzer generator called lex. So, this is the original tool as I said that this was proposed in the UNIX operating system this came with the UNIX operating system. So, this is a lexical analyzer generator that is it can give generate ul lexical analyzer given the source specification file.

So, for the source specification file so, you should have this extension dot l there is a lex source program. So, that is given to a lexical compiler the tool is a. So, we have got the 2 lex and there are some more tools like some improved version flex and jflex like that so, there are some improved version. But, what they essentially do is that they generate a c program lex dot yy dot c this lex and flex they generate the c program. So, that lex dot yyy dot c and this lex dot y y dot c being a c program you can pass it through a compiler c compiler to generate the executable version a dot out and this a dot out is a lexical analyzer for the source a lex specification file.

So, if you if you give some input to this program which follows the above the specification that we have in this lex dot l then it can accept it can accept it can detect the tokens that are appearing in the input stream and return those tokens. And like many other jobs can be done some format manipulation and also everything can be done. So, this is basically a lexical analysis tool.

So, this input stream will be coming they will be given this a dot out will analyse this input stream and it will give you the sequence of tokens. So, it will act as the lexical analyzer.

(Refer Slide Time: 16:43)



Structure of Lex programs

```
declarations
%%
translation rules
%%
auxiliary functions
```

Pattern {Action}

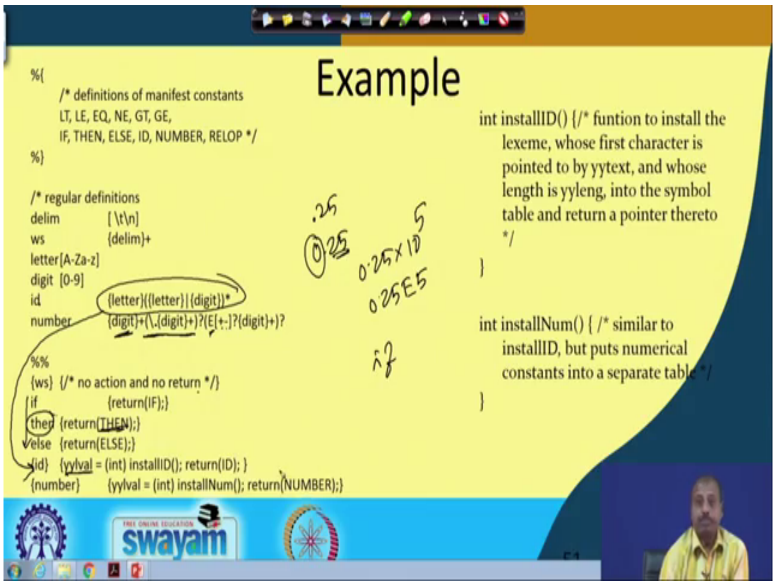
So, we will see some sample form sample structure of this lex program. So, there is no we do not have scope for going to a detailed discussion on this lex tool. But, what we will do is that we will see some overview that how this lex tool operates or the specification file.

So, this is the lex specification file. So, this is the lex specification file. So, we have got it it has got 3 parts the declaration part the translation rules part and the auxiliary functions part. So, this declarations part so, these 3 parts they are separated by this double percentage marks so, this is the format. So, all the regular definitions and declarations so, they should be put in the declaration part.

Then what are the trans translation rules at what do you do when you see a particular token particular lex in matching with a rule then what do you have particular definition. For example, here I have a put a regular expression say r r 1. So, here I have to tell for r 1 what are you going to do? So, these rules part we will be telling what to do with the when we see this particular thing. For example, if you find say some variable declaration and then maybe the action for that is to install the regular install the corresponding variable in to the symbol table.

So, that way we can do it. So and there are some auxiliary functions so, which may be used by this transition rules that we are writing here. So, these actions are written in a c code and those c codes get integrated into the lexical analyzer tool. And as an output so, it produces the patterns and the corresponding actions. So, this ultimately generates a switch case type of statement big switch case statement for each of these translation rules. So, it will have the corresponding actions which are given here so, this way this a lex programs will be structured.

(Refer Slide Time: 18:43)



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, this is a typical example. So, the first part the definition part is up to the first up to this much is the definition part ok. So, here you can say you can definitions of a manifest constant like LT LE EQ so, this way we can define all the tokens basically. So, these definitions they can they can be useful for this parts are for the time being they do not have any meaning.

So, this then comes this is this regular definitions. So, this one is so, do we are defining a regular definition delim delimiter which is blank tab and newline character. And they are put with put within this square bracket means it is the alternatives. So, we blank the blank character the tab character and the new line character. So, they will constitute the set of delimiters.

Then the white space is a regular definition whose actual definition is like this delim plus. So, that is 1 or more occurrence of this delimiter characters so, that is a white space. Then we are defining letter so, letter is the definition and the corresponding stream sequences is it can be this A to Z and then and then small a to small z.

So, that we can have this sequence of characters giving this regular definition then digit is 0 to 9 now an ID can be letter followed by letter or digit whole star. So, I do not have any other symbols. So, it says that the identifier it starts with a letter and the remaining characters are have to be letter or digit and the that is the star. We can define the number so, number is defined like this that I should have at least 1 digit followed by a dot or then digit plus.

So, if you have a dot then you have got a digit in the 1 or more digits then there is a question mark. So, question mark means this entire thing this part there may be 0 or 1 occurrence. So, that will take care of the situation that is at sometimes we do not have the fractional part after decimal point. So, it will be taking care of that and if you are having fractional part then definitely has to be 1 digit before that.

So, you cannot write like 0.25 like that. So, you have to write like 0.25 because, this 0 in that case will match with this digit plus part and this then this point will be there and this digit plus will match with this 25 ok. But, if you write like 0.25 then this first digit is not there so, it will not be matching. So, this type of minute details may be there. So, that you have to see and this whenever we are writing the specification the lex specification we have to be careful at those points. Then comes the exponentiation part that is at 10 to the power part so, this symbol E must be there then you can have plus minus and there is a their question mark means it is 0 or 1 occurrence.

So, we can have a sign or we may not have a sign. So, if we have a sign that can be plus or minus or there if there is no sign then that is also fine in that case it is taken as plus. Then followed by digit plus so, you should have at least 1 digit available. So, whenever you are writing say 0.25 into 10 to 10 to the power 5. So, we have to write like 0.25 then E 5. So, you have to do it like this.

So, now we have to come to the action part. So, for whitespace we do not have any action so, no action and no return. So, it is the simply written as no action. Then if you have the token i if you have the symbol like a regular definition if. So, if it matches with the if the input is like i if state if word then it will return the token if ok. So, it is there so, if it gets the if it gets the sequence t it is sequence then and then this particular regular expression will match and in that case the action part is to return the token THEN.

So, in this way whatever regular definitions we have. So, you can put it here the corresponding regular expressions and in the action part we have to give the corresponding c code. So, here it is returning else as we know that the parser will called the lexi analysis tool for the next token and this is how that tokens will be returned. So, it is then else etcetera so, they are all defined as tokens in the language and so, it will be it is in the parser will know these tokens and the lexical analyzer will also know the tokens. So, it will be returning those tokens to the parser.

Now, this id so, this id whenever you are putting within this brace; so, it mean that this will be this definition will be replaced. So, it will be taking this definition and it will be replaced here. So, now, it is doing it like this that yy l val. So, this is a special variable that this lexical analyse analyser tool and parsing tool they will understand. So, it is a some value of it is the value of the token that is returned.

So, I said that every token has got some attributes. So, this value is one of the attribute for the token and the value is returned by the variable y y l val. So, we are setting the yy l val to be the install ID function will be called. So, that will be installing the corresponding identifier into the symbol table and it will return the corresponding index of the table and it will return the identifier. And in case of number so, it will be converting it will install the number into the number table and it will return the token number.