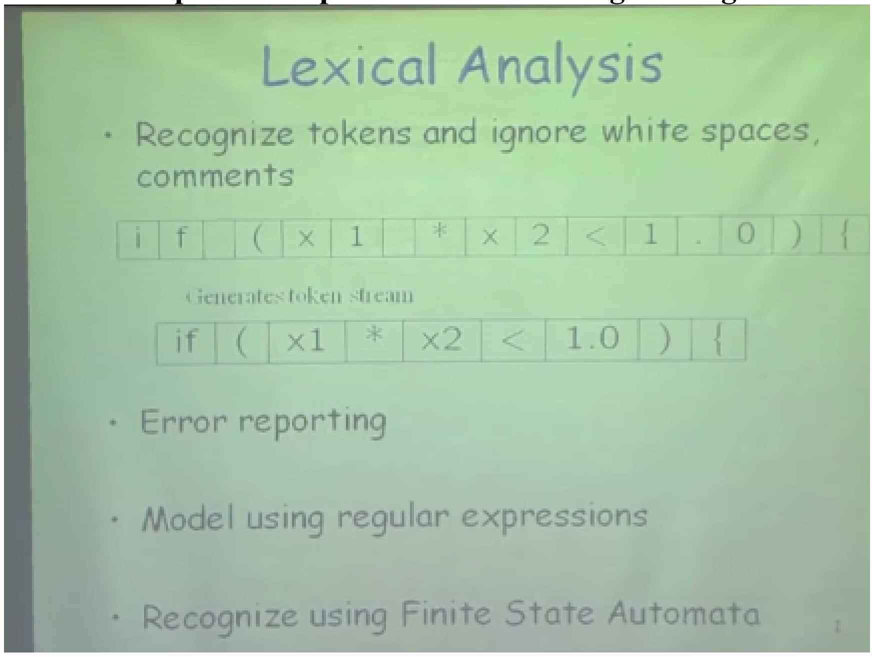
Indian Institute of Technology Kanpur
NP-TEL
National Programme
On
Technology Enhanced Learning
<u>Course Title</u>
Compiler Design
Lecture – 04
By
Prof. S.K Aggarwal.
Dept. of Computer Science and Engineering



Lexical Analysis

- Recognize tokens and ignore white spaces, comments

i	f		(	x	1		*	x	2	<	1	.	0	)	{

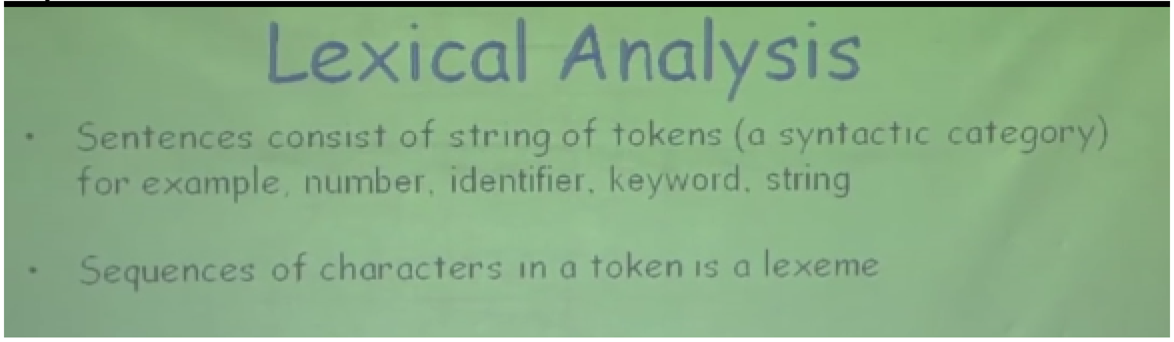
Generates token stream

if	(	x1	*	x2	<	1.0	)	{

- Error reporting
- Model using regular expressions
- Recognize using Finite State Automata

So once again we need to recaptured, this is what I've taken from my introduction, this is the input to the lexical analyzer which is sequence of characters and what it generate is a sequence of tokens and the process it does error reporting whenever it finds that some token talk properly form and the whole thing is model using regular expressions and we then use Finite State Machine of Finite State Automata tool, implement lex, okay.
Now before we get into really the implementation we need to understand few terms, okay, so one common term or I will actually introduce few terms to you, so one is what we know as the token, another is lexeme, and the third term is going to be rules for construction of tokens. So we need to understand clearly what are this terms and what does it mean, what do these terms mean, okay.
So once again let me go back and start taking an example, and introduce these terms by an example, suppose when I write an expression like this and I'm trying to tokenize this, I'm trying to compile this particular expression and obviously remember that this is insert to context so I'm not looking at the whole program I'm just focusing on this part. And typically what I'll be doing is, I will be passing on information by saying that there is an identifier, there is another identifier, here is another identifier, this is an assignment operator, and this is technician

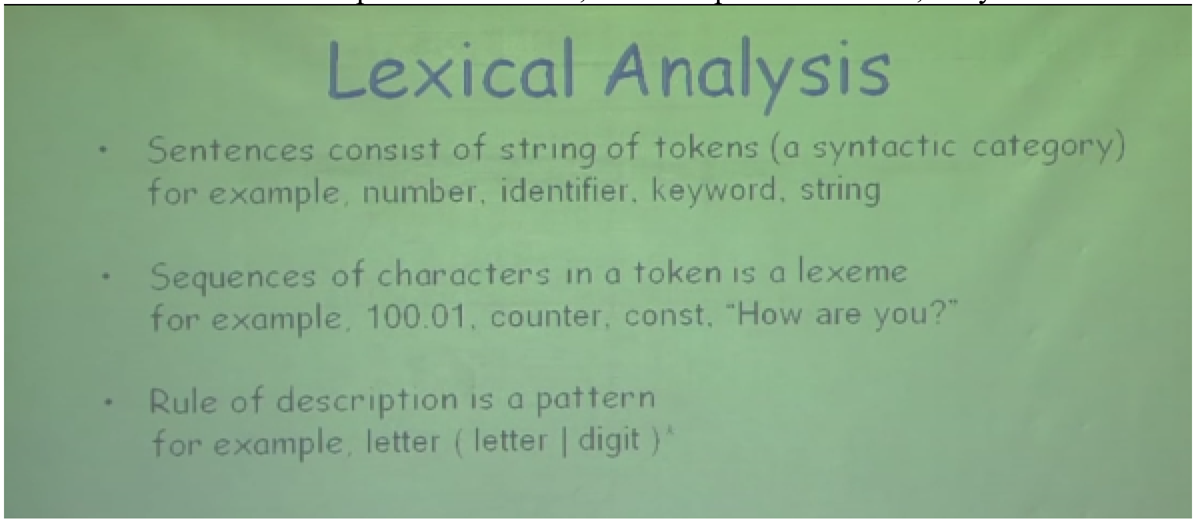
operates, so my stream of tokens is going to be ID assign ID addop ID, okay, so this is what my input to this syntax analyzer, okay.
So what we have up if I look at this, this and this, these are really nothing but the tokens, okay, but if I look at this string, if I look at this string and if I look at this string, these are really the lexemes which are associated with these tokens, okay, so tokens are this class of, this class which we are going to pass on the syntax analyzer associated with each of these and it is going to be a strength which we call lexeme. And then we have rules which we say that all these tokens are going to get constructed, okay. So this is the overall three terms we are going to use very formally, okay, so sentence consist of a string of tokens which is really is syntactic category, so if I look at identifier here, and this is nothing but syntactic category, okay, so this is one we may use, second will be so here is an example where we have numbers, identifiers, keywords, string and so on, okay, and when I look at sequence of categories in tokens which is mainly this



Lexical Analysis

- Sentences consist of string of tokens (a syntactic category) for example, number, identifier, keyword, string
- Sequences of characters in a token is a lexeme

sequence which is a lexeme so for example if I say 100.01 that's a number, okay, or if I say counter which is an identifier or I may say a keyword which is a constant of the, there must be a skilled, so skill always comes within codes and here you can say that this is really the lexeme which is associated with this particular lexeme, with this particular token, okay. And what are

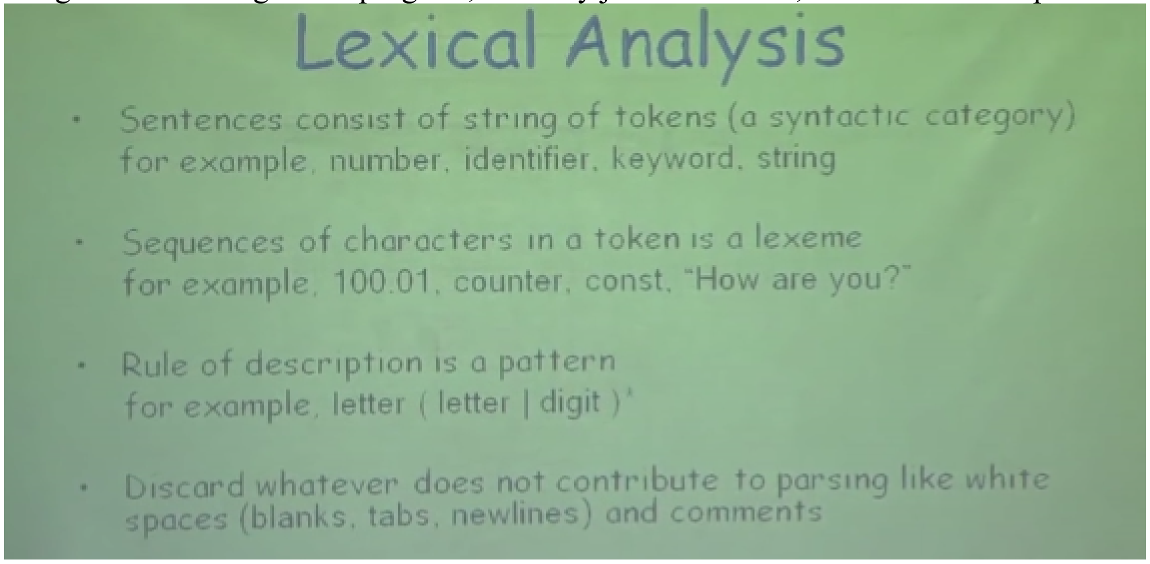


Lexical Analysis

- Sentences consist of string of tokens (a syntactic category) for example, number, identifier, keyword, string
- Sequences of characters in a token is a lexeme for example, 100.01, counter, const, "How are you?"
- Rule of description is a pattern for example, letter (letter | digit)*

the rules of description so I may have a rule like this it says that how an identifier is going to get constructed, so each identifier must start with the letter, it must be followed by 0 or more occurrences of letter or term, okay.
And in the process what we are going to do is when I tokenize my input I also want to discard all the useless information, so what is useless information here? So all the spacers here maybe

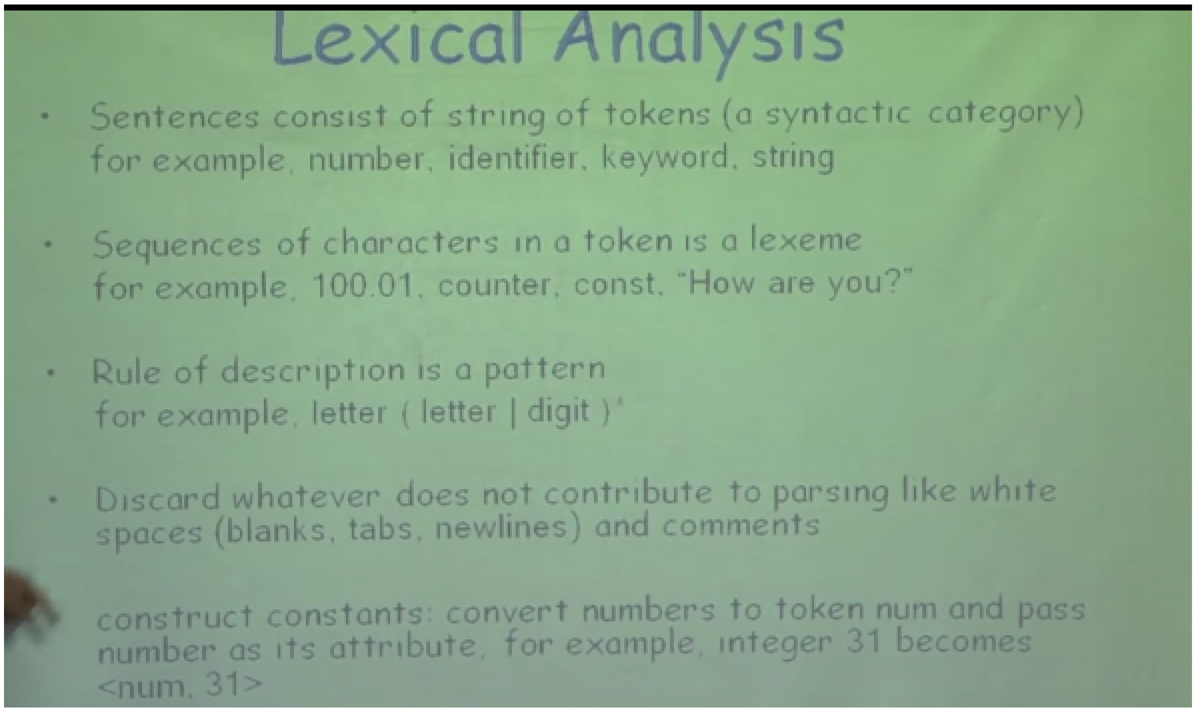
useless, so I can put one blank or I can put two blanks, I may put a tab, but tab doesn't have anything to the meaning of the program, so I may just decide that, so all this white spaces



Lexical Analysis

- Sentences consist of string of tokens (a syntactic category) for example, number, identifier, keyword, string
- Sequences of characters in a token is a lexeme for example, 100.01, counter, const, "How are you?"
- Rule of description is a pattern for example, letter (letter | digit)*
- Discard whatever does not contribute to parsing like white spaces (blanks, tabs, newlines) and comments

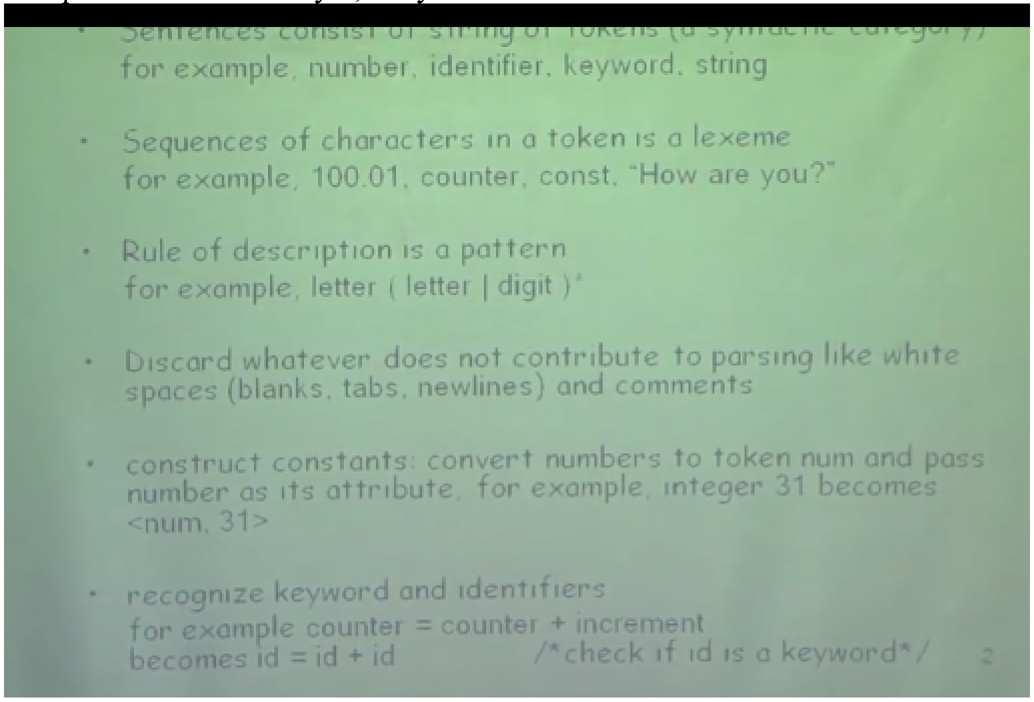
which are blanks, tabs, newline characters, all these are going to be with parsing, unless they occur us part of a string, I can be part of the string, then that is really the lexeme I cannot discard that, and also what we want to do is whenever we deal with numbers I don't want to pass



Lexical Analysis

- Sentences consist of string of tokens (a syntactic category) for example, number, identifier, keyword, string
- Sequences of characters in a token is a lexeme for example, 100.01, counter, const, "How are you?"
- Rule of description is a pattern for example, letter (letter | digit)*
- Discard whatever does not contribute to parsing like white spaces (blanks, tabs, newlines) and comments
- construct constants: convert numbers to token num and pass number as its attribute, for example, integer 31 becomes <num, 31>

this as a character sequence, but I want to convert this into a number. So for example when I'm reading this I am going to read six characters which will be one zero zero dot zero one, but when I pass on this information to this syntax analyzer, I will say I have read a number whose value is 100.01, so I should be able to convert this character sequence into a number, so for all numbers I will do this conversion so when I pass on this information, I'll say there is a token which is number and the value of the number is 31, I'll not say value of this number is, as a character sequence P followed by 1, okay.



- Sentences consist of string of tokens (a syntactic category) for example, number, identifier, keyword, string
- Sequences of characters in a token is a lexeme for example, 100.01, counter, const, "How are you?"
- Rule of description is a pattern for example, letter (letter | digit)*
- Discard whatever does not contribute to parsing like white spaces (blanks, tabs, newlines) and comments
- construct constants: convert numbers to token num and pass number as its attribute, for example, integer 31 becomes <num, 31>
- recognize keyword and identifiers for example counter = counter + increment becomes id = id + id /*check if id is a keyword*/

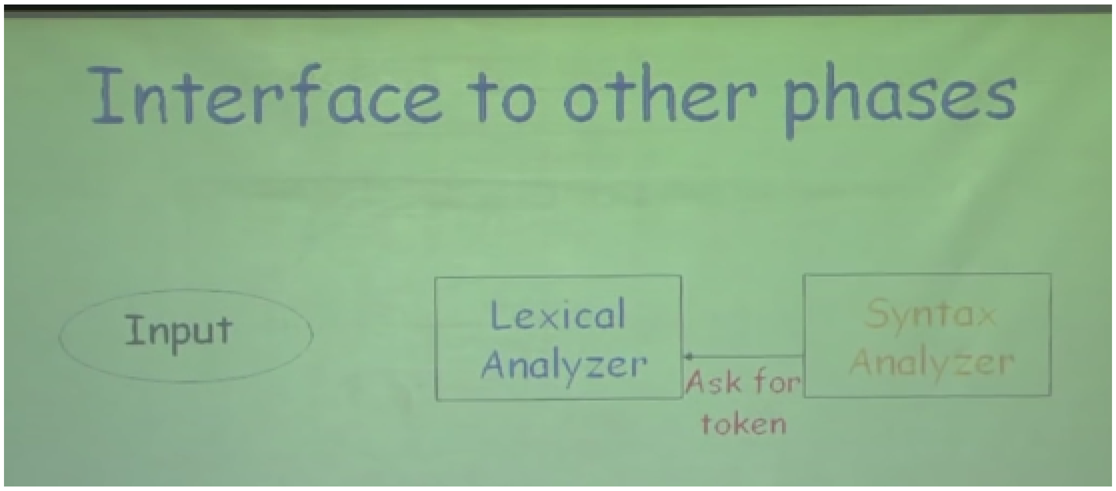
2

And we also need to recognize whether I have certain keywords, now the rules of constructions of keywords are, similar to rules of construction of identifiers, okay, but certain keywords in some languages or in most languages if not some they are going to be reserve, so we will have to take an input like this, so if I say that my input is counter mean assign counter + increment, so this was the common thing you will see X is high X+1, right.

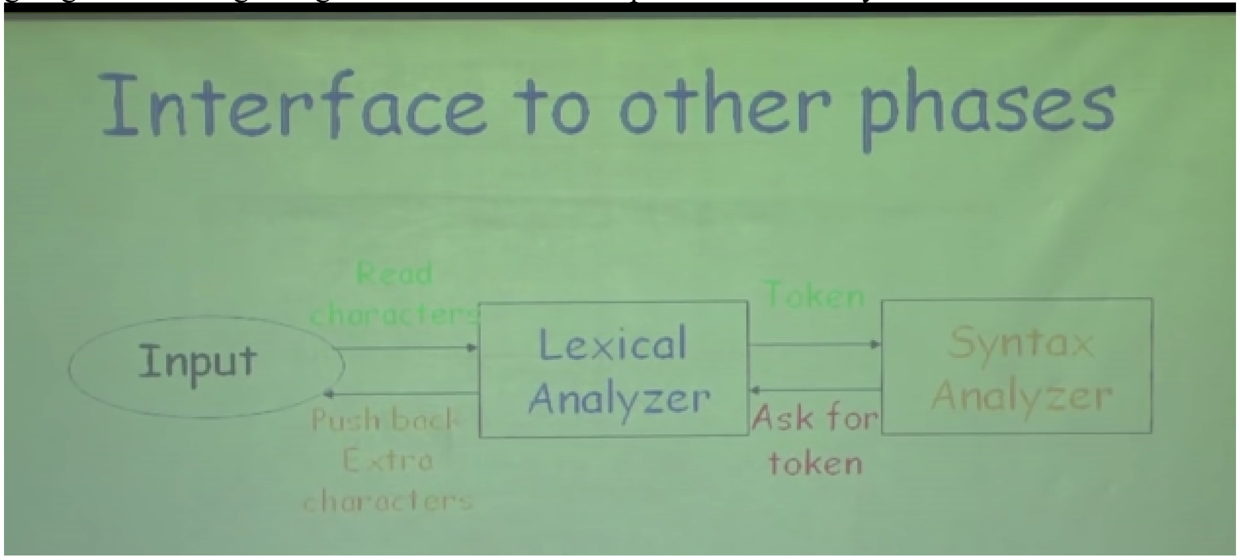
So what I will pass to this syntax analyzer, this is what I am going to pass through syntax analyzer, and every time I do that I'll have to check whether I'm dealing with one of the identifiers or one of the keywords, if I say this is the keyword in my language, then I don't want to perhaps deal with this kind of situations, okay. Some languages still permit that, but most languages will not, okay, so I need to figure out that whether I am dealing with an identifier or I'm dealing with a keyword. And keyword is a list of words which are reserved in the language, right, so functional specifications clear to everyone, what we are trying to do here, okay.

So let's move on and let's see how does interface and other phases of the compiler, okay, so if you go back and recall the compiler structure we had, we had somewhere to the first phase in this compiler which was the lexical analyzer and then we had syntax analyzer, so really if I am looking at lexical analyzer it is dealing with two entities, one which is input, and another is syntax analyzer, okay, it does not have to deal with any other phase other than the symbol table where you have to put all the information, okay.

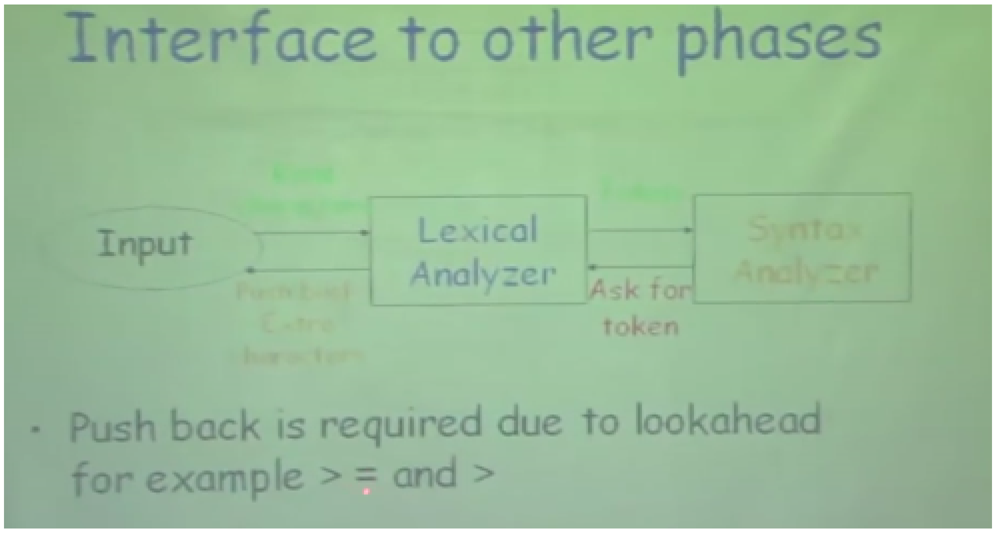
So this is how the lexical analyzer looks, it is feeding information to syntax analyzer and it's taking input from the sequence of characters, okay. Now assume that the syntax analyzer is the one which is driving the whole process, okay. So look at it this way that when I'm starting my process of compilation, syntax analyzer says I want to pass something give me a token and syntax analyzer is asking for a sequence of tokens and who will you trust? Obviously the lexical analyzer, so what it says is it just asks for a token and now to generate token what



lexical analyzer has to do, it has to start taking characters from input, okay, so it starts taking characters from input, okay, and it starts falling now a token, okay, so for example if I say I want to compile this, syntax analyzer says give me a token and lexical analyzer then reads A, but by reading it will it know that it is a token or not, doesn't know, it has to re-go, okay, so it has to continue to read, so it will read then the next character and then it suddenly realizes that I've reached the word boundary and A was dealing between the word boundary was, between A and assignment symbol, okay. So in the process what it has done, there is an extra character, logically, okay. So what happens is that it has formed a token which is A and in the process it also has to say that whatever extra I have read push that back to input stream because this is going to be the beginning of a new token. Is this point clear to everyone?



So what we are trying to do here is that when lexical analyzer is trying to read characters and forming the token, it identifies the token and in some cases not always, in some cases it will also have to pushback all these extra characters into the input sequence, okay, logically that is what is happening we will see how the implementation happens.
So pushback is required due to look ahead because here I am saying that I’ll have to do a look head to find out this word

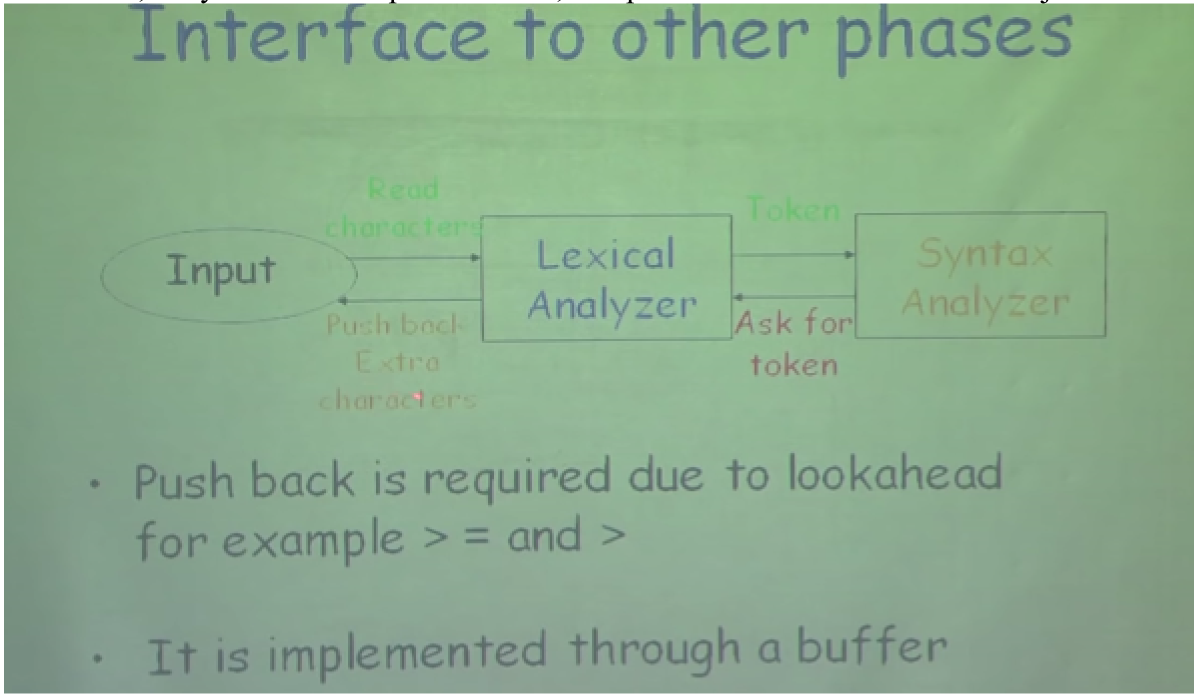


Interface to other phases

Read characters
Input
Lexical Analyzer
Token
Syntax Analyzer
Push back Extra characters
Ask for token

- Push back is required due to lookahead for example > = and >

boundary, okay, and here is an example, so if I say I'm trying to read greater than equal 2 or greater than, I will not know unless I have read the next character whether it is part of the same token or not, okay. So this is implementation, this pushback extra character can be just

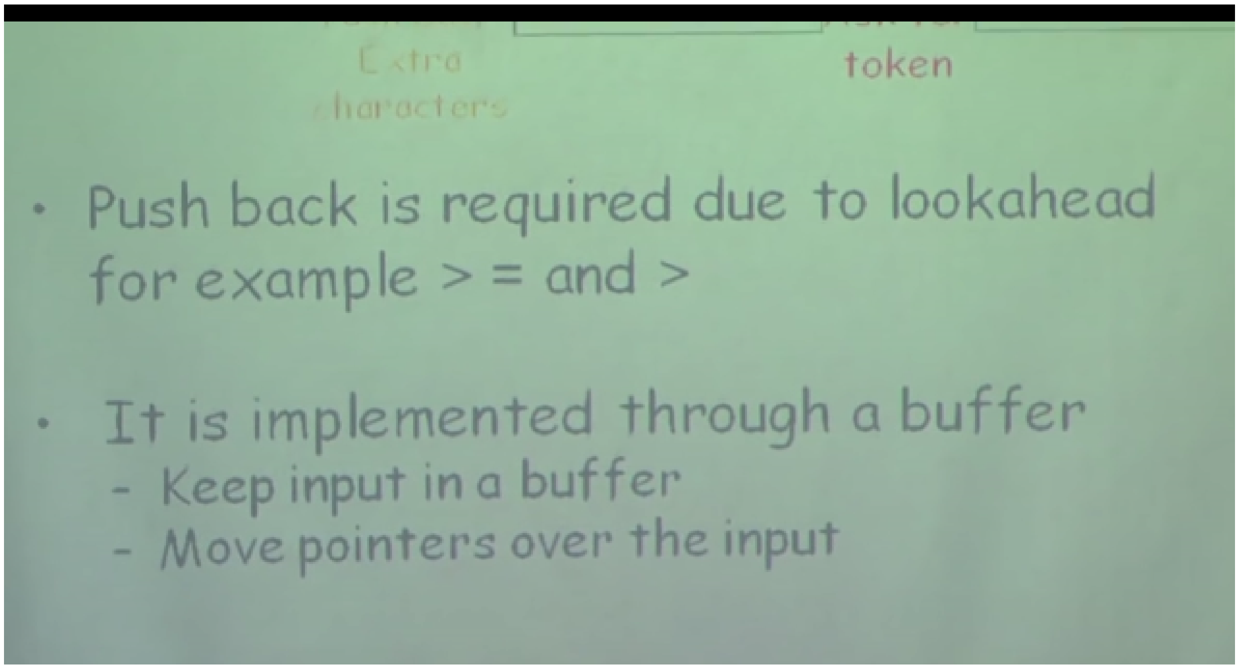


Interface to other phases

Read characters
Input
Lexical Analyzer
Token
Syntax Analyzer
Push back Extra characters
Ask for token

- Push back is required due to lookahead for example > = and >
- It is implemented through a buffer

implemented to a buffer and all that means is when I'm reading a character or pushing back a character what it means is that I have an input pointer which will move left or right, okay. So pushing back is not uncapped as, trying to do NC, okay, you can just use a buffer to implement it and I had to take care of this input point, okay, so keep input in the buffer and keep moving this pointer over the buffer, left over, okay.

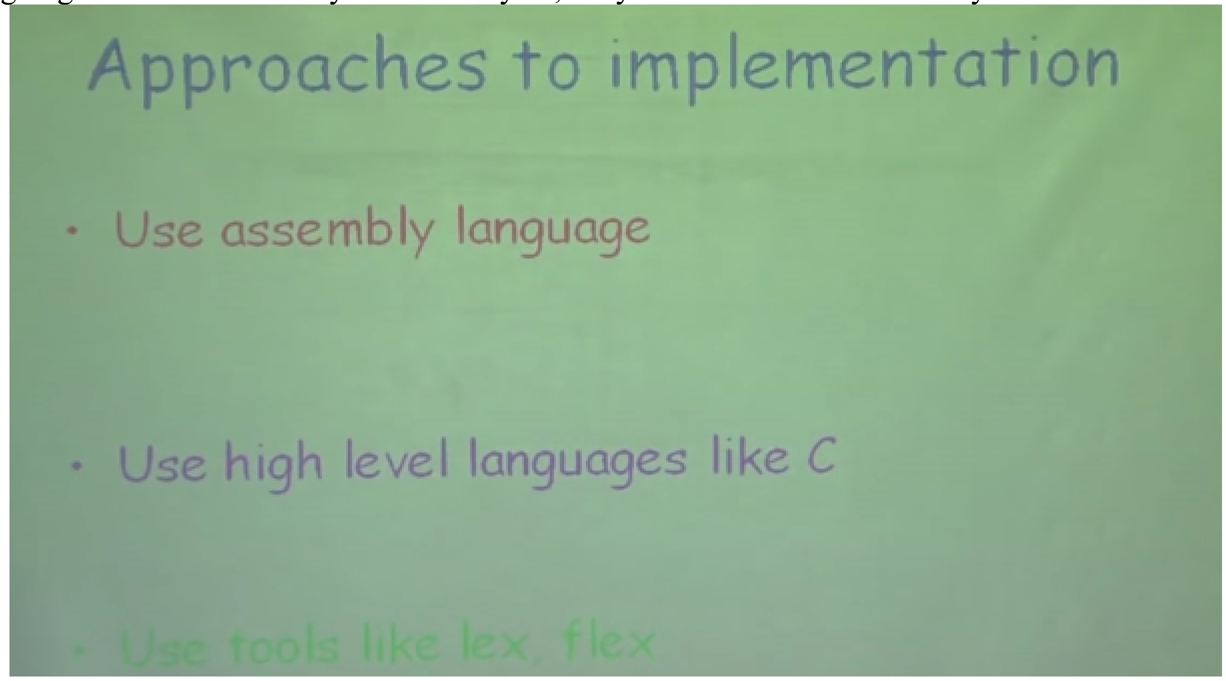


Extra characters

token

- Push back is required due to lookahead for example > = and >
- It is implemented through a buffer
  - Keep input in a buffer
  - Move pointers over the input

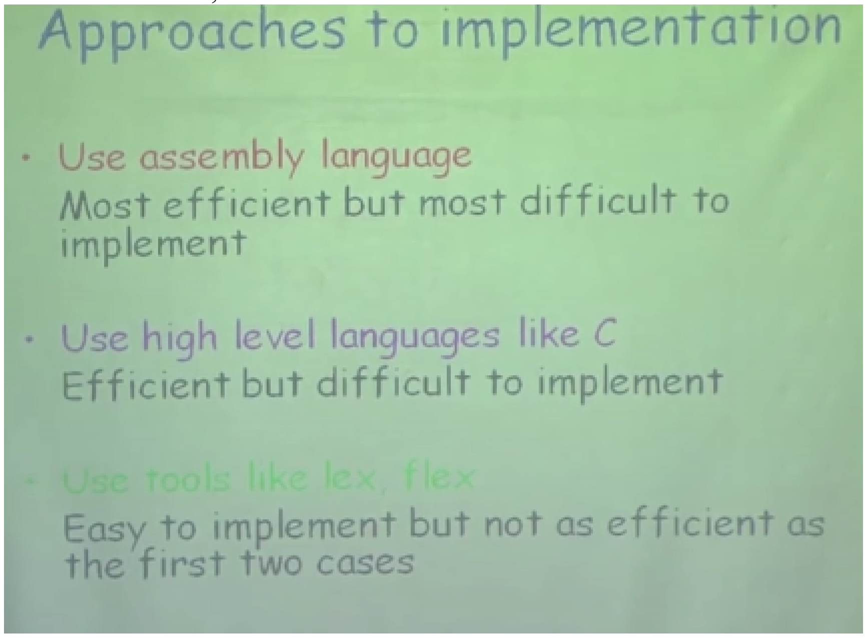
So how do I implement lexical analyzer? So we understand what lexical analyzer does, let's start looking at how I am using implement and I'll talk about three approaches to implementation, one is that since I am doing low-level IO and I want this process to be very efficient I can always program this in assembly language, okay, that is one option that is available to me, another option is obviously I can use a high level language like a programming language C or I can use tools like lex or flex which are where I just write specification in R going to then increment by lexical analyzer, okay. So remember that not only I'm worried about



Approaches to implementation

- Use assembly language
- Use high level languages like C
- Use tools like lex, flex

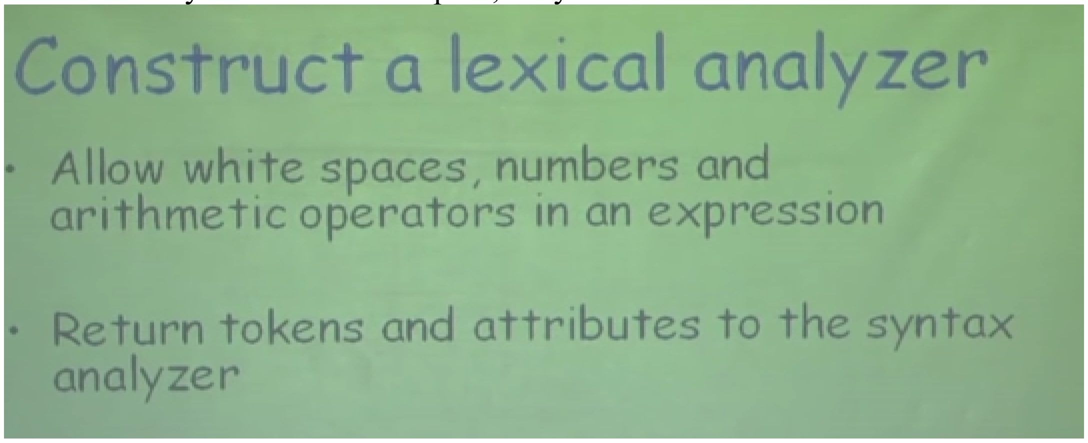
functional correctness of the phase, I'm also worried about the speed of the phase, how fast it can read my tokens because you don't want the situations where you have a program which is very slowly reading your characters and that becomes the bottom line because remember this phase involves lot of test file, okay, your program is going to be your phase, so this is lot of file which will take place, so what should I take? Any ideas? First one, okay, third, okay but what about efficiency? Development time will be very less, true, but runtime? So what do I wanted? So, oh can be very different, language, so typing in assembly program to discuss it, what this a character mean in the high level language, there will be noticeable discuss in later the time, okay, so normally if I look at those icons, okay, this one is obviously the most efficient because you can do low level IO, but also it's going to be most difficult to implement, this is definitely efficient not as efficient as this, and



Approaches to implementation

- Use assembly language
  Most efficient but most difficult to implement
- Use high level languages like C
  Efficient but difficult to implement
- Use tools like lex, flex
  Easy to implement but not as efficient as the first two cases

it's difficult to implement, not as difficult as this, once again, but if I go to the third approach, okay, it's easy to implement but it's not going to be as efficient as the first two cases, here we don't have no control over IO, I'm only hoping that my tool will be going good IO, tool will be doing good, function madness and that is why you see we had tools like flex, which says fast flex, okay, because lex did not implement but not any efficient changes, okay, so this is also goes back to the first point that define that your tool is not efficient and you get a better tool, then you can actually have a faster compiler, okay.

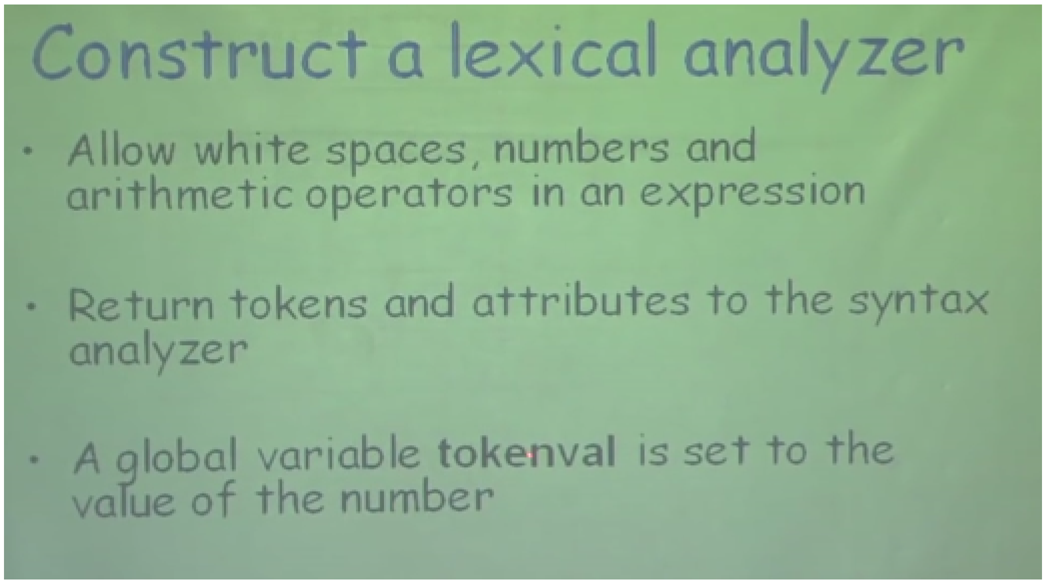


Construct a lexical analyzer

- Allow white spaces, numbers and arithmetic operators in an expression
- Return tokens and attributes to the syntax analyzer

So normally what happens is in practice, okay, when we start implementing, we'll start with the third approach, okay, because we don't want to become bottleneck for the consequences, okay, but as the rest of the compiler is getting developed, okay, you keep moving to a high level language, and keep implementing at least I/O processes in assembly language, okay. So to make sure that normally or functionally correct, you are also high efficient, okay, so always start from this but slowly migrate at least to this level and then put all your high level, low level make sure that I/O does not become bottom level, whatever language is, okay. So we need to therefore understand how to implement lexical analyzer using this approach and how to implement lexical analyzer using this approach in a systematic manner, so even when I want to write programs, I want to write C programs to implement the lexical analyzer, I just want to be writing arbitrary C program but I want to have certain structures over it, okay, and we will talk about both these approaches of implementation, okay, this point clear to everyone? Any comments, questions here, okay.

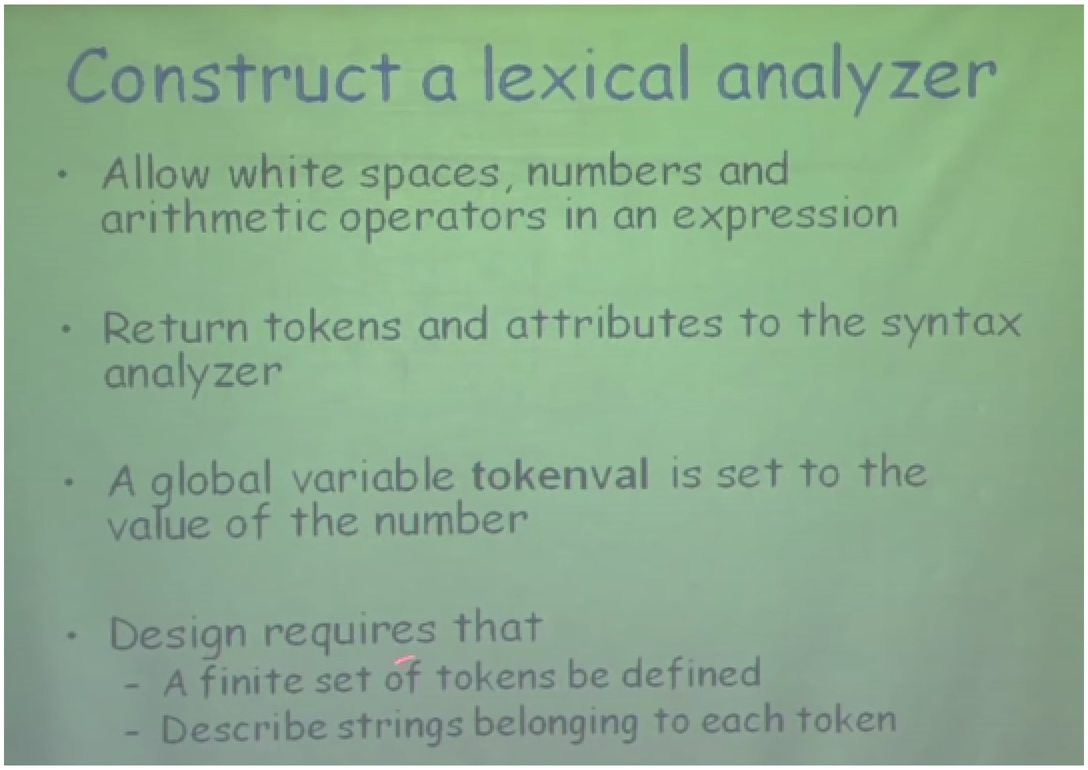
So let's move on, and let's actually take a small language and try to construct the lexical analyzer first, where and what goes on in lexical analyzer, okay, so this language is going to allow white spaces, it's going to have numbers and it will have arithmetic operators in an expression, okay, and what it is going to do is, it is going to return tokens and it is going to return an attributes to syntax analyzer, so token will be something of this class, okay so it will say it's an identifier and my attribute maybe saying either it's a lexeme or it may be an attribute to the symbol table, okay or attribute when you say that so if I write something like this A = B + 46 it may say that here token is number and attribute is number 46, okay, so this is the information it is going to return, okay, and what we'll do is we will assume that I have a global variable just for the sake of the implementation of this program instead of returning a value I'll say there is a



Construct a lexical analyzer

- Allow white spaces, numbers and arithmetic operators in an expression
- Return tokens and attributes to the syntax analyzer
- A global variable **tokenval** is set to the value of the number

global variable where I'm going to copy this value and then subsequent values are just going to read this global variable and subsequent of this set, so token value by global value which is going to set to the value of

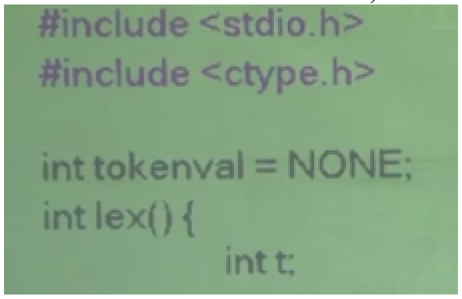


Construct a lexical analyzer

- Allow white spaces, numbers and arithmetic operators in an expression
- Return tokens and attributes to the syntax analyzer
- A global variable **tokenval** is set to the value of the number
- Design requires that
  - A finite set of tokens be defined
  - Describe strings belonging to each token

the number and what it requires is that we have a set of tokens which are defined and then we describe strings belonging to each of the tokens, okay.

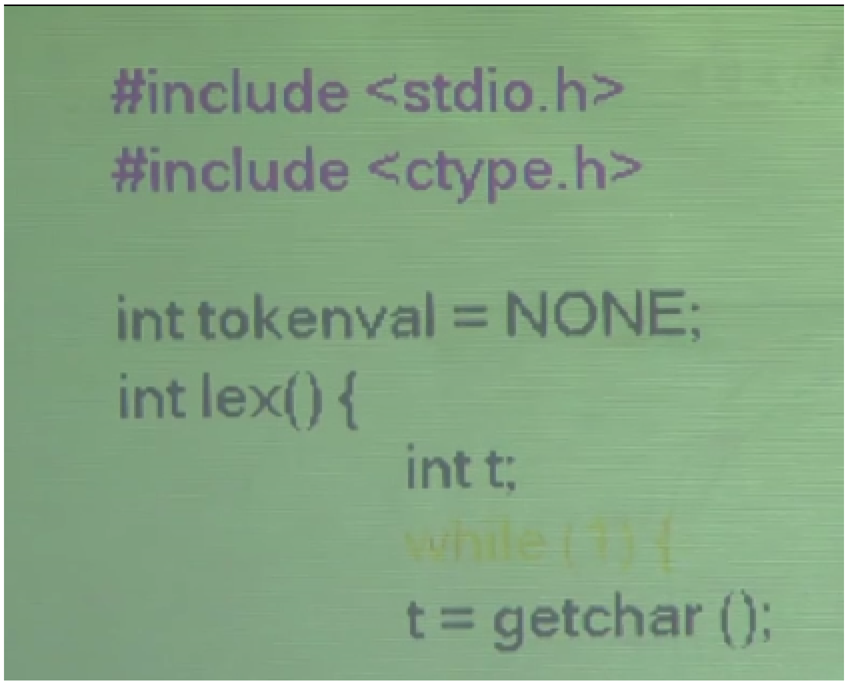
So let's just look at the structure of such as echo, okay, so I'm just writing the standard C program without worrying too much about the structure and nearly the point of showing you this program is that how one does it can be to write the lexical analyzer, it just involves so few tokens, okay, and then motivate you to use high level specifications and some tools, which can then make sure that this job is done correctly and efficiently, okay, so we'll have all these hash in to statements and I have initialized this tokenval to none, none is some identifier character,



```
#include <stdio.h>
#include <ctype.h>

int tokenval = NONE;
int lex() {
            int t;
```

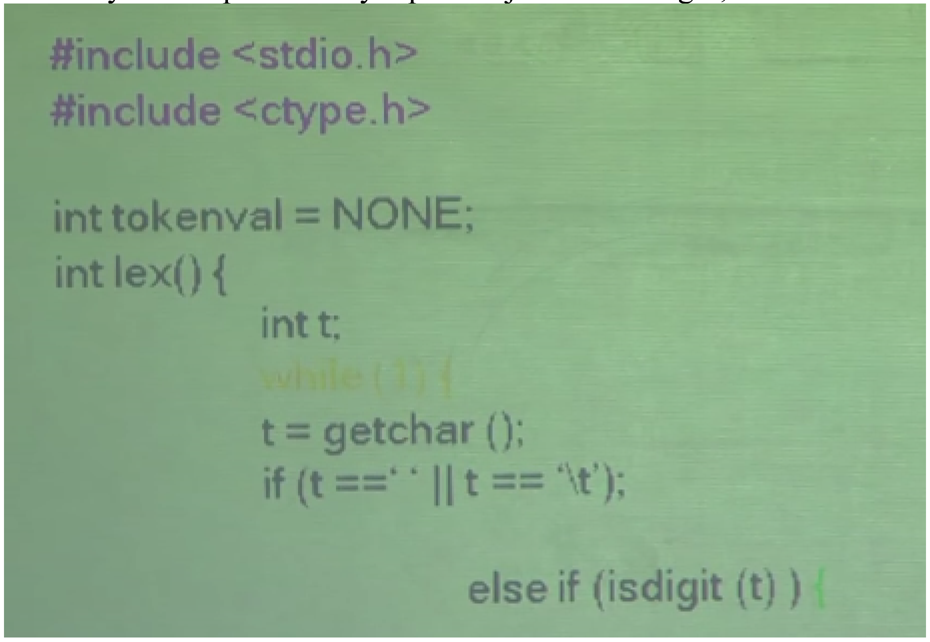
which is an integer now, okay, and here is a function lex, okay where I have T as in integer define, and this is where the function closes, okay.

Now what are the things I need to do? I'm sorry, this is not visible, because of brightness what this is while 1 that means going an infinite loop and keep reading a character coming, okay, so this is reading a character and first thing I do is, I do a getchar right I read the first character and then I



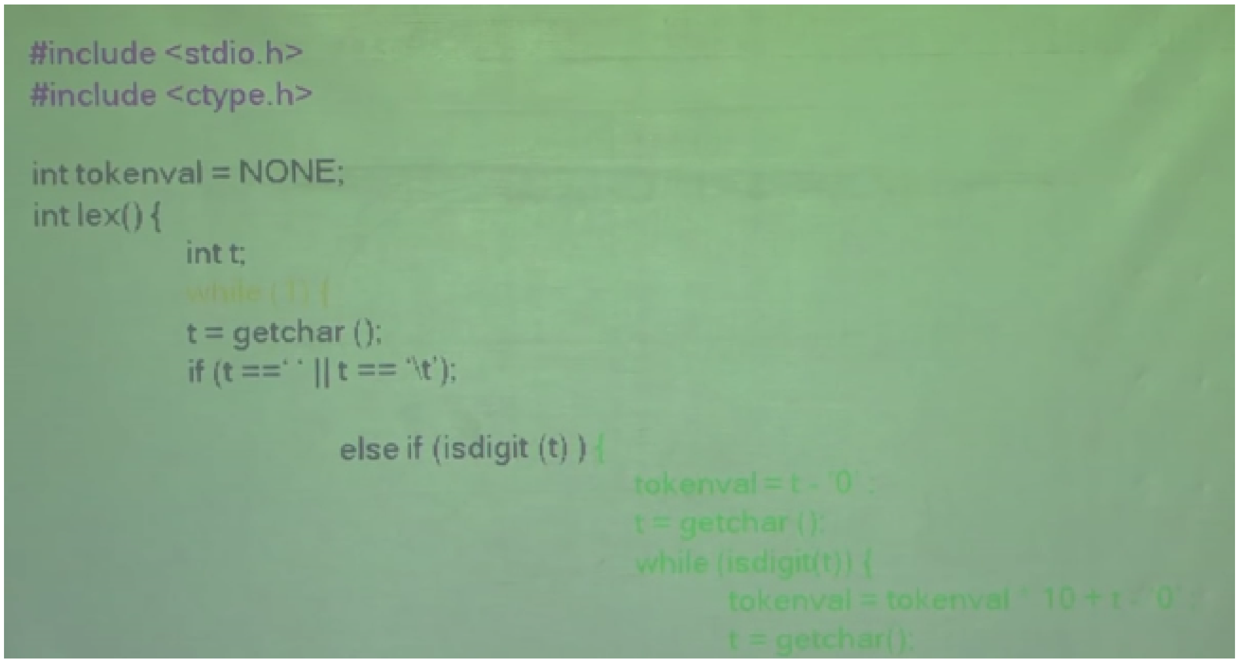
```
#include <stdio.h>
#include <ctype.h>

int tokenval = NONE;
int lex() {
            int t;
            while ( 1 ) {
            t = getchar ();
```

find out whether this is a blank or a tab character, this is a blank or tab, I just ignore it, okay, yeah I don't want any white spaces in my input I'm just discarding it, otherwise if this is a digit



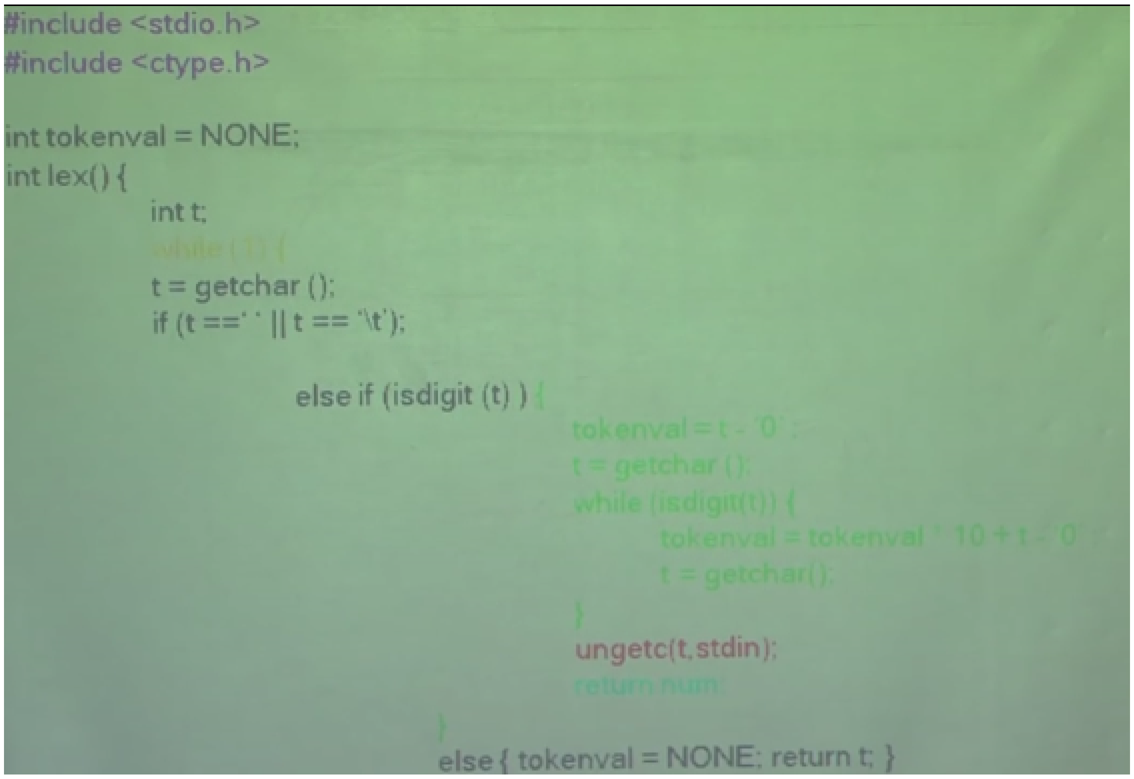
```
#include <stdio.h>
#include <ctype.h>

int tokenval = NONE;
int lex() {
            int t;
            while ( 1 ) {
            t = getchar ();
            if (t ==' ' || t == '\t');

                    else if (isdigit (t) ) {
```

okay then what do I have to do? Is my input is a digit then what do I do? Make a decision like, I cannot wait, there is nothing waiting there. So I've to read the next character and find out whether that is a digit or that was forming a number, okay, so I keep reading as long as I get digits, and I'm reading I'll also keep converting this digits to a number, right, and whenever I get something which is non-digit then I'll say I've reached the word boundary and I must start beginning of a new token, okay, so this is what we do here we say tokenval is T minus whatever is asking value of zero, and T is then getchar and then I'll say while I keep on getting digits, keep on constructing tokenval, and how do I construct tokenval? I'd say simple as arithmetic expression I take the previous value whatever I have read, previous value is



```
#include <stdio.h>
#include <ctype.h>

int tokenval = NONE;
int lex() {
            int t;
            while (1) {
            t = getchar ();
            if (t ==' ' || t == '\t');

                        else if (isdigit (t) ) {
                                    tokenval = t - '0' ;
                                    t = getchar ();
                                    while (isdigit(t)) {
                                          tokenval = tokenval * 10 + t - '0' ;
                                          t = getchar();
```

multiplied by 10, whatever I have read that is added and I go in this loop, okay, and I keep on doing it so long as I keep on getting digits, okay.
Now you can see that when I come out of this loop I have the number as tokenval, but I'll also I would have read at least one extra character to find out whether it was digit or not, okay, then only I'll come out to the loop, so that's an extra character I have read, okay, so that is the first thing I'll have to do, then I'll have to put it back into the loop, so I can getchar, right, okay. Now once I do that, then what happens so what other situations could have been there in the input, okay, I request everyone to please switch off your mobile phones, and remember it from the next class onwards if you enter the room just switch it off. So what other characters I'd have got, I have a blank, I have a tab, which are white spaces, I have a digit and what else I could have in my specification, I felt that assembles, right so here I know that I have recognize the number which I am returning, okay otherwise what I do is I say that tokenval is none, and



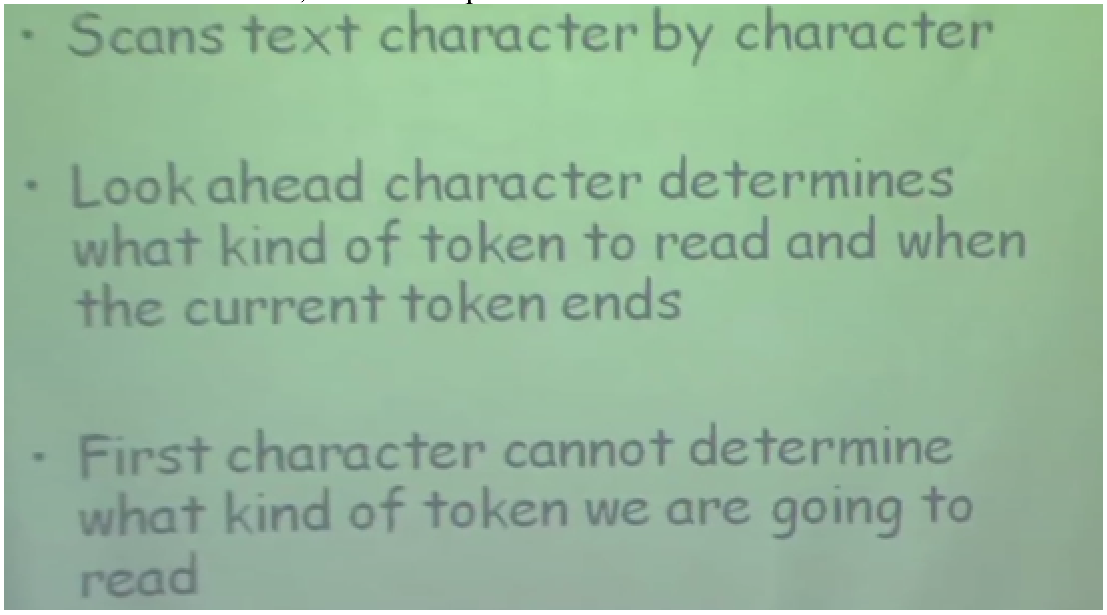
```
#include <stdio.h>
#include <ctype.h>

int tokenval = NONE;
int lex() {
            int t;
            while (1) {
            t = getchar ();
            if (t == ' ' || t == '\t');

                        else if (isdigit (t) ) {
                                    tokenval = t - '0' ;
                                    t = getchar ();
                                    while (isdigit(t)) {
                                            tokenval = tokenval * 10 + t - '0' ;
                                            t = getchar();
                                    }
                                    ungetc(t,stdin);
                                    return num;
                        }
                        else { tokenval = NONE; return t; }
```

tokenval is got a number, and whatever is the T I just returned that, so if I read a plus for example I just say return plus, here if I read a minus, I'll return minus, otherwise, I return a number, okay, so this is only specification I had, I had expressions, which consisted of just the numbers and arithmetic operators, and starting white spaces.

Now you can see that just to do this I have to write the program C program which is like 10 to 15 times of 4, okay but more interesting part okay, I have to write, I have to use all these data structures, okay, I had to use this iteration here, I have to do an I/O here, okay, so this is what really makes lexical analyzer complicated. Now imagine you have full specifications of programming language like C or parsing, okay, I read already, okay and if you start writing a C program like this the chances that you will make an error, okay, somewhere in your declaration, somewhere not using the loop properly, not returning the character properly to the input buffer and so on, okay, so this approach is not clearly something we want, right, okay. So what do we do now, okay, so one extra thing which I am bringing at the end, now read this, okay, what it is saying is that if my input character is a newline character then I am saying increment some line number by 1 and I have initialized my line number 2, okay, now why do I need line numbers here in lexical analyzer, I am just returning this program right and I'm passing the structure. So for example if I had something like this suppose my input was something like this, okay. Are we going to return the same set of tokens, so why I am calculating this information like line number, how does it help me in doing parsing and rest of the compilation?

So when I have to do debugging at that point of time I must remember that for which line number what was the code which was generated, because my view as the user is going to be that I want to take breaks at certain, I've used GDs, how many of you are familiar with GD mean? Almost everyone, right? So when you set a breakpoint you say break at certain line number, okay, so how do I map that line number in code? Somewhere I start generating this information, okay so this information will also be communicated subsequently it will remain some as some tab within my code so that I can then we'll start taking at certain line number and

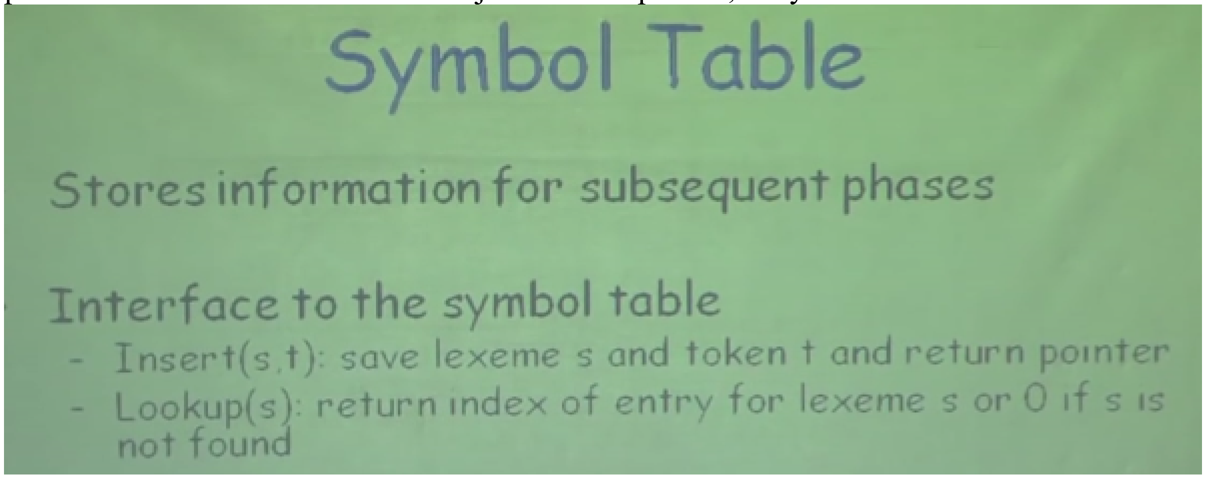
this is if you now go back and start reading what I showed you in one of the files that compiler is part of overall program development environment and it has to feed information to all the fields, okay so debugging was one of the phases there, makes sense?
Okay, so far, okay we have seen some effects of what lexical analyzer is doing, but let's also see what are the kind of pitfalls here to be aware of, what are the problems in a phase in right to lexical analyzer, so when I start looking at specifications of a language I must make certain flags in, certain situations other, okay so what was the kind of problems I will face here? One is obviously that I am reading my input character by character, so your I/O has to be very efficient, okay, also look ahead character is going to determine what kind of token to read and when the current token ends, so for example



- Scans text character by character
- Look ahead character determines what kind of token to read and when the current token ends
- First character cannot determine what kind of token we are going to read

this is, if I'm reading this and then I read this, this is going to determine what kind of token I had, okay, and this also is saying what is beginning off the new tokens.
So when I start for example something like XYZ is assign something, by reading this character I have some idea of what kind of token or what kind of character I am expecting, okay, that'll also give me a hint of, what is the finite state machine I am going to use to identify this particular tool, okay, and first character alone cannot determine what kind of token we are going to get, okay, so if I read for example first character of this, so the example I already gave you if I have this and this is not clear just by reading either this or this what is it that I am going to get, unless I read at least few extra characters, okay, in this case I am assuming to begin with that look ahead of one will be sufficient to find out, but we'll see situations when look ahead of characters is not going to be sufficient, okay.
So next issue that comes is that how do I interface with the symbol table, okay, so what was my interface with the symbol table? So if you recall very quickly what I had was a symbol table here and symbol table lexical analyzer is to put information in this symbol table, okay, and what kind of information it can put into symbol table, what information lexical analyzer has at any point of time, when we knows that, let's see end of token, it has no other information, okay, so what will I put in the symbol table? So when I say that I encountered A = B + C and it says A is a token, what more information I have is the Lexi right so I should be able to then in my symbol table say that I have token which is of type identifier and it has lexeme here, right. But now imagine this situation that I have this A = A+C and this is what I'm tokenize, okay, now again

you will say I have an identifier where lexeme is 8, but that must have already been entered in the symbol table when I process this part, so do I make a duplicate entry in the symbol table? No, so how do I know that this already exists in the symbol table? See that information is there, pointers are there but this is a table right which has multiple records, so I must be able to look up this table and say whether such a token or such a lexeme already exists, right because if a token with this lexeme already exists then I don't want to make an entry but when I come to this I'll say again insert this identifier with a lexeme C, right so these are the two functions I need, I should be able to look up in my symbol table and I should be able to insert something in my symbol table, what do I insert? The only thing I can say is insert this token and this lexeme and lookup will say lookup this particular string, okay, and these two functions should be sufficient to interface as far as lexical analyzer is concerned, those symbol table I don't have to worry about rest of the structure of the symbol table, I don't have to worry about the rest of the free symbol table, alright, this are the only two fields I'm going to deal with as far as lexical analyzer is concern.
So this is what we do that I'm going to store information for subsequent phases because I need to know what my lexemes are, okay and I need two functions which will say that same this particular lexeme and this token and just return a pointer, okay and before I insert I also want to



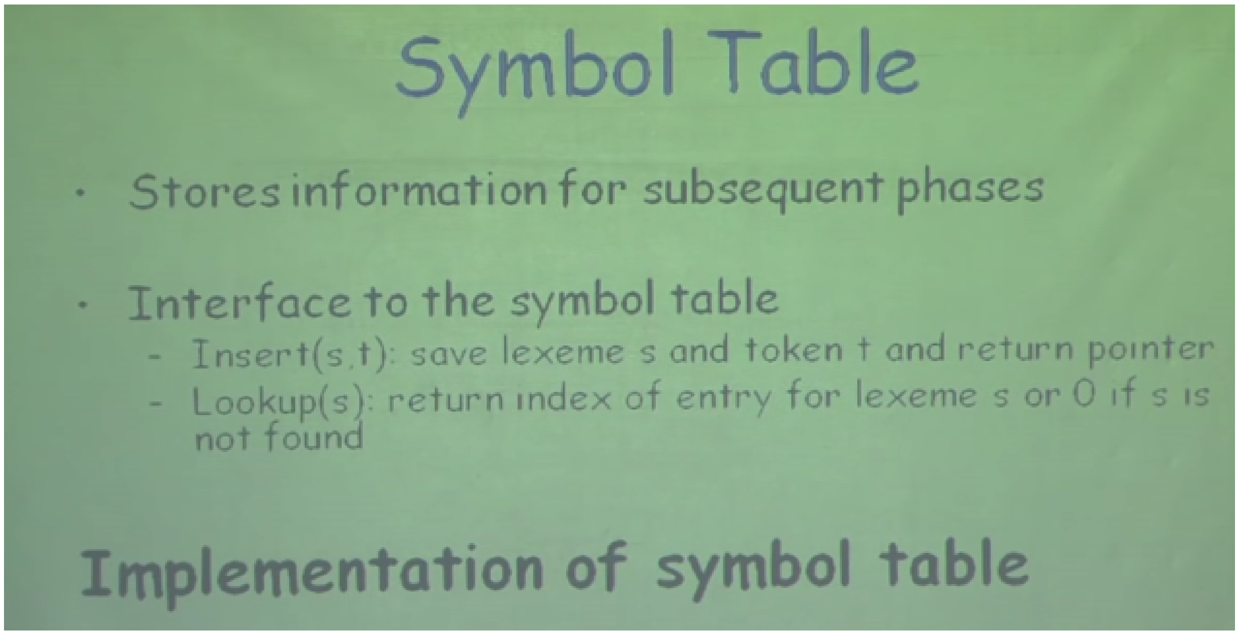
Symbol Table

Stores information for subsequent phases

Interface to the symbol table

- Insert(s,t): save lexeme s and token t and return pointer
- Lookup(s): return index of entry for lexeme s or 0 if s is not found

look up and say that if this already exists then just give me an entry to the symbol table, if it does not exist okay then give me another pointer, then you say that is not exist, okay. And these two functions are sufficient interface, okay, and how do I implement symbol table, okay.
Now if I look at symbol table one very preliminary implementation I gave you was that I want



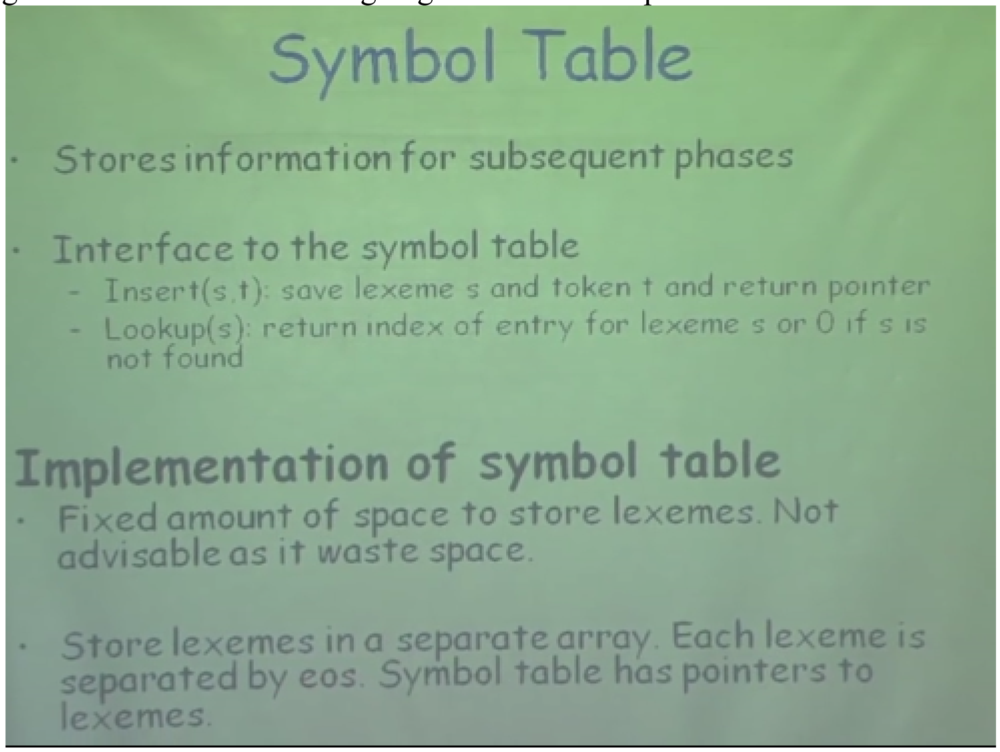
Symbol Table

- Stores information for subsequent phases
- Interface to the symbol table
  - Insert(s,t): save lexeme s and token t and return pointer
  - Lookup(s): return index of entry for lexeme s or 0 if s is not found

Implementation of symbol table

to have some kind of structure of adding of structures, so what I want to do here is, I want to have let’s say one rope corresponding to each of the variables I have in my program and this will consist of information like saying what is the token, what is the lexeme, and then I’ll have more information, so I’ll have for example type, I’ll have address, and some other information which is part of the symbol table, but right now let's focus on this part okay.
Now if this is the kind of space I am using, okay, we will not worry about how efficient this is at this point of time that we should be let this later so assume that I have now an adding, okay and you can see that one thing when I do a look up here, okay, is going to be linear look up right unless I continuously sort the symbol table and do some by the reason, okay, so assuming that I have linear lookup because that's really not the issue at this point of time, okay. When I am implementing this, okay look at how much space I am consuming, okay and look up because at this point of time I am concerned only with these two fits, okay let's look up how much space I am consuming here, okay.
Now if I look at token how much space do I need for tokens? To store a token, so let's talk in terms of bytes, bit, bytes, that’s about low level language, right, how much space do I need to store for token? Depends on the number of tokens I have in my language, okay, suppose I have 24 different kind of token in my language, okay then how many bits I need? 32 is the maximum, 32 is something which is close to this, so just 2 to the power 5 I can do everything in 5 bits, okay, so if I have 24 kind of tokens then 5 bits are sufficient, okay, so that is the input, okay, I can always and for each language I know predetermine, I can predetermine what are the kind of tokens I’ll encounter, I can encode it in terms of bits, and I can have this conflict information, so there is really no space over it here, okay.
Now what about the lexeme part? How much space do I reserve for lexeme? How much space do I required? Depends on the, depends on the span, stored my identifiers here, it depends upon how many bytes I can have in each identifier, and most programing language today we will target up to 32 bytes, so you can have a variable which is 32 character, so that means I need to store, I need to reserve space of 32 bytes, okay, but if I could average size of variables, when you have return programs, right, so what is the average size of variable do you use? Variable and, 5, 6 really more than that, okay so average will be 5, 6 some variables maybe, most

variables will be like IJK and some variables may be like 7, 8, okay and if you take average it will come to 5, 6. Now 32 bytes, reserving 32 bytes when I know for sure that on the average I'll not be using 5 to 6 bytes, this highly inefficient space wise, okay, so I need to come up with some better data structure than this, so what do I do? Give me some ideas, okay, so this clearly I mean yourself are saying there is lot of space which is being wasted here I want to recover this space, so what do I do? Make it dynamic, so what do I do to make a dynamic?
Yeah, you were saying something? Okay, so very good, so what we can do is that I can have some separate memory which I can keep allocating for my identifiers and all I need to do is have a pointer here which will point to this, so this is what my lexeme is going to be, so let me say this is identifier 1, so identifier 2 and so on, and each one will have a pointer here, right, okay. Now you can see that what is my overhead? Pointer typically is going to be 4 bytes, okay so if your pointer is only 4 bytes long then you know that what you are using is very compact and overhead on the average for each lexeme is going to be this 4 bytes, okay, and this is what is implemented almost in all symbol tables that fixed amount of space to store lexeme is not something which is advisable as it is going to waste lot of space and therefore store all these



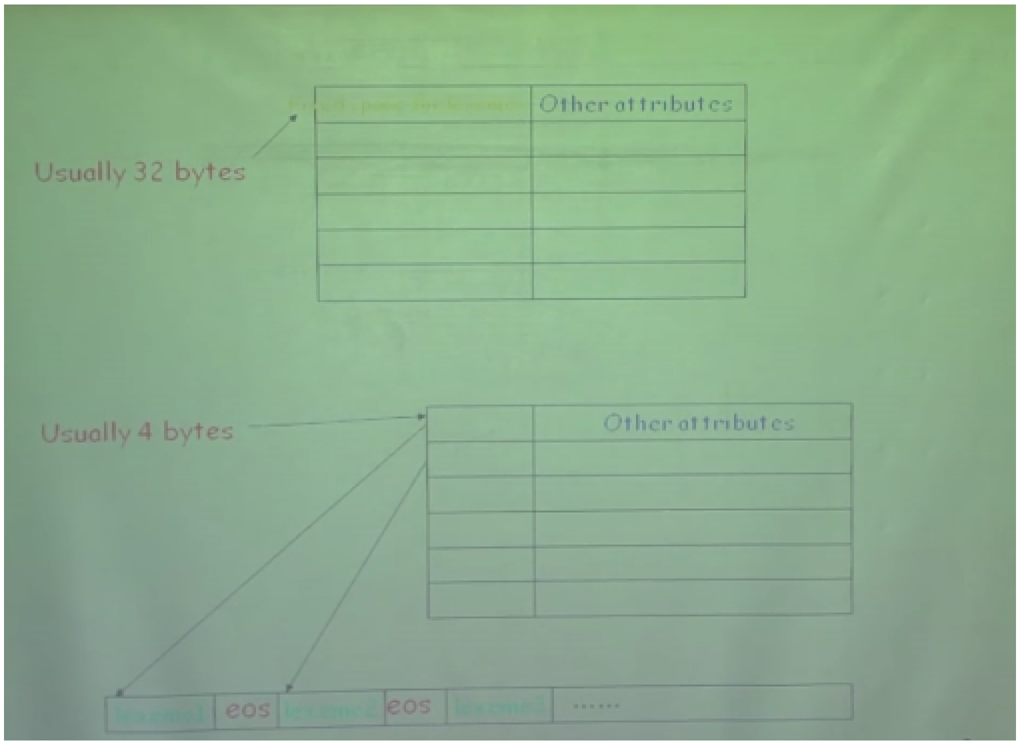
Symbol Table

- Stores information for subsequent phases
- Interface to the symbol table
  - Insert(s,t): save lexeme s and token t and return pointer
  - Lookup(s): return index of entry for lexeme s or 0 if s is not found

Implementation of symbol table

- Fixed amount of space to store lexemes. Not advisable as it waste space.
- Store lexemes in a separate array. Each lexeme is separated by eos. Symbol table has pointers to lexemes.

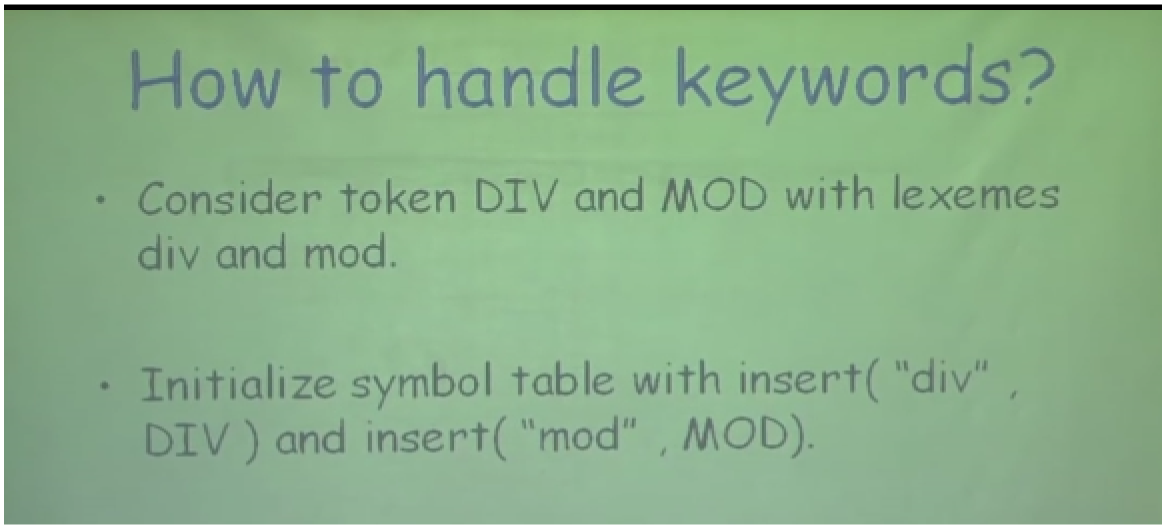
lexemes in a separate space and each lexeme is subject by some character because I don't want to store information like what in the length of the lexeme, okay and symbol table has just pointer to lexemes okay. So one implementation could have been like I have this fixed space for



lexeme and all other attributes versus here which is usually going to be 32 bytes once I have all these separate spaces for lexemes which is stored separately and I have just the pointers to all this symbol tables, now all this lexemes, okay, so that makes my implementation, space wise more efficient, okay.

Our next issue that comes is how do I handle keywords, okay, so how do I handle keywords, okay so if I say so when I wrote something like A = B + C suppose I write A = if+5, okay, and as I start spending my input from left to right, I say that I encounter if and suppose in my language if is a result keyword, cannot be used as an identifier. Now how do I know that if is and as far as rules are concerned, rule of construction of this is same as rule of construction of this, okay, so this will say it's an identifier, how do I know that this is not an identifier, how do I handle this situation, so whatever we have seen so far.

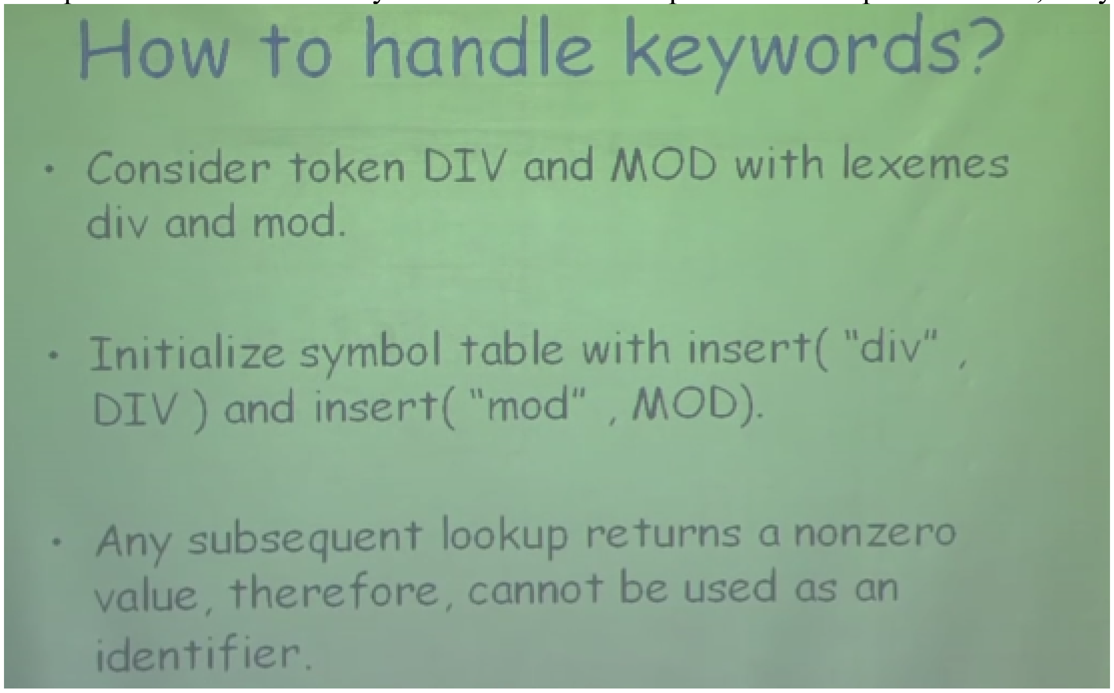
Yes, so I need to maintain a list of keywords, if I maintain just a list of keywords, okay then I know that first before saying that this is a identifier check it against list, okay and how do I maintain this list, okay, I can always initialize my symbol table by inserting all the keywords initially in it and whenever I will look up what will happen, lookup will just say that this already exists in my symbol table and it is a keyword, so I don't have to worry here, okay, so this is really what we do that when I am saying that insert these as lexemes, okay I just initialize my symbol table with all the keywords, okay and since I have this lookup function, lookup



How to handle keywords?

- Consider token DIV and MOD with lexemes div and mod.
- Initialize symbol table with insert("div" , DIV) and insert("mod" , MOD).

function is going to ensure that I never complicate between a keyword and identifier, lookup function will always make sure because before I insert I always have to do a lookup so if I just loop a sequence of inserts in the symbol table before the process of compilation starts, okay,

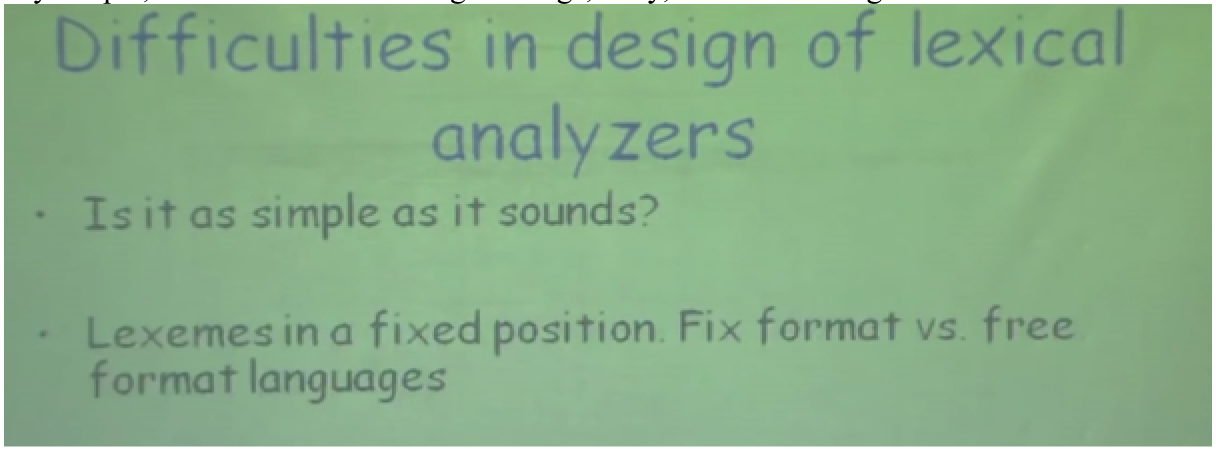


How to handle keywords?

- Consider token DIV and MOD with lexemes div and mod.
- Initialize symbol table with insert("div" , DIV) and insert("mod" , MOD).
- Any subsequent lookup returns a nonzero value, therefore, cannot be used as an identifier.

then I have taken care of this and any subsequent lookup is going to return a nonzero value and therefore I will say that this cannot be used as a token, or this cannot be used as identifier, okay, so as far as lexical analyzer is concerned if this is an error, no there's not an error, right, who will catch that this is an error, which phase? Pascal will say that you cannot use a keyword in an expression, right, as far as lexical analyzer is concerned this is going to just return the sequence of tokens which is a identifier assign keyword and number and that's it, okay, so you must also remember that what was the kind of errors which lexical analyzer can catch and what are the kind of errors is subsequent phases are going to hand, okay so using a keyword inside an expression is as far as lex is concerned or lexical analyzer is concerned is not an error, but some subsequent phases are going to capture this information provided lexical analyzer is supplied information that this is a keyword and not an identifier, okay.

Yes, okay, so what are the kind of difficulties I may face, what are the kind of flags I need to raise before I start designing a lexical analyzer, okay, so far everything sounds reasonable if not very simple, so let's start now raising red flags, okay, the first red flag that comes is can I use a

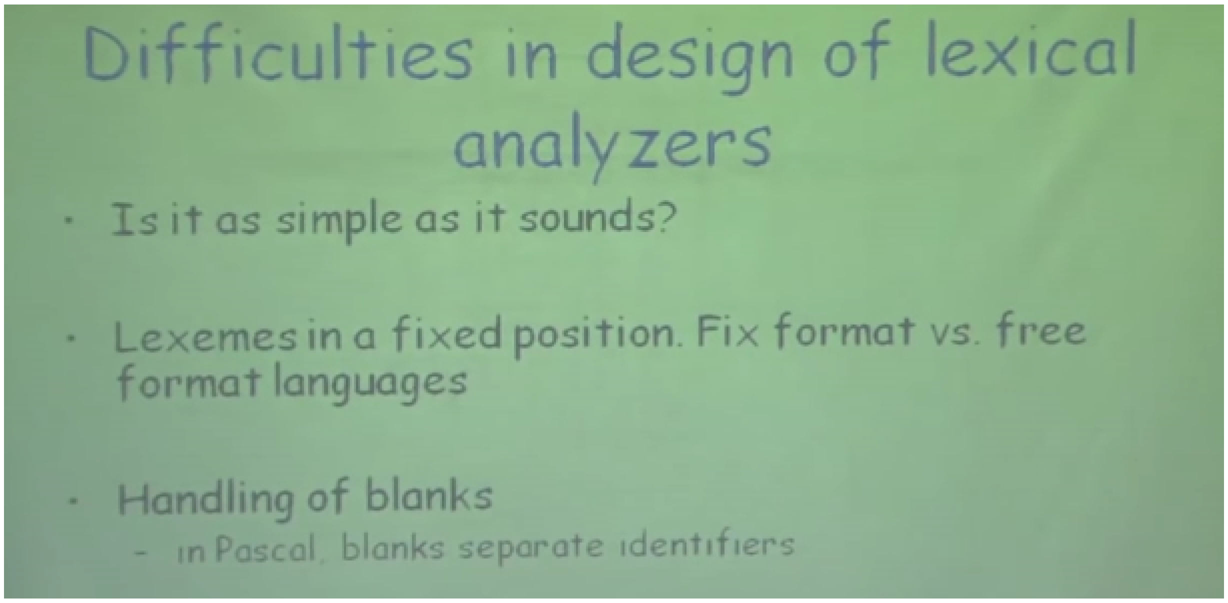


Difficulties in design of lexical analyzers

- Is it as simple as it sounds?
- Lexemes in a fixed position. Fix format vs. free format languages

format which is this, what’s this, a format which is this, these are two formats which express the same language, okay, could I have written this also in a form which will say, so this, this and this, do they need the same thing? Depends on the implementation of what, lexical, it’s nothing to do with implementation, the reason is this part of the specification of the language, this language tells me that whether I can write an expression in this form or this form or this form, okay, if all the three forms are valid then I should be able to write designed lexical analyzers which will doesn't matter, what is the input you give me, even if you give me one character for input or one word for input, for line of input I’m okay with that, what's this language is if you say no, you cannot use this format, you cannot use this format, you must use only this format, okay, so that issue is like whether I am dealing with free format languages versus fixed format languages. Now this is not a part of implementation issue, the part of language specification and lexical analyzer must implement whatever is specified by the language, lexical analyzer is not supposed to deviate from whatever is the language specification, so this part of language specification which says that whether lexeme are in fixed positions versus it's a free format language, right.

Now do we know any fixed format language? Python and the first language was the fixed format language, of course it was a fixed format, okay, so whenever you deal with these formats, you need to worry about whether you are in fix format versus free format, okay so typically if I'll show you more examples of fixed format language and what kind of difficulties they can introduce, next issue we have to deal with this I said right in the beginning my input is going to be a sequence of characters, and I need to tokenize it, and then we said that I'm going to define certain word boundaries and how do I determine word boundaries, I’ll say either I encounter a



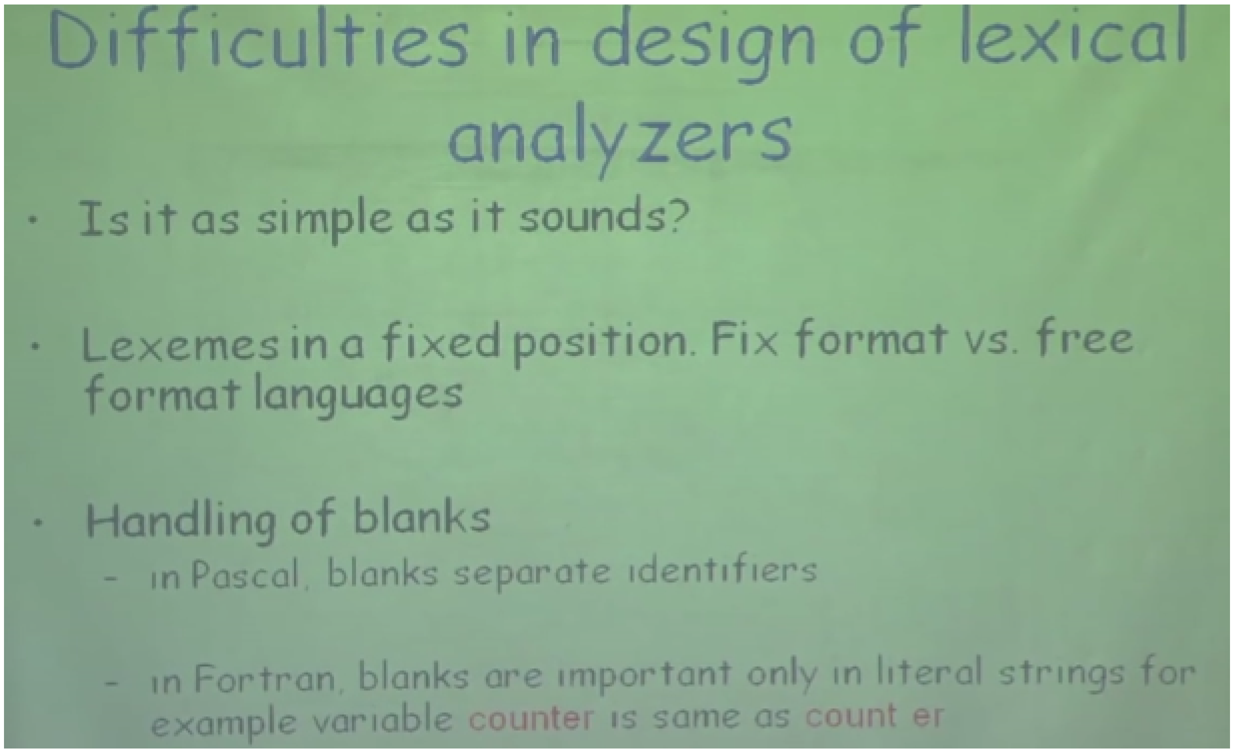
Difficulties in design of lexical analyzers

- Is it as simple as it sounds?
- Lexemes in a fixed position. Fix format vs. free format languages
- Handling of blanks
  - in Pascal, blanks separate identifiers

blank or some blank space that means either a sequence of blanks or tabs or newline that will be a word boundaries or some kind of punctuation, which is like comma or full stop, semicolon or so on, or I encounter a character from a different class of characters, okay so if I say I have A here and I come to assign that I know it's a different class of characters this is the word boundary although there is no blank type okay, but blanks typically most languages are useless word separators, okay.

Now let's look at a slightly different situation, so it is one expression, let me write one more expression, are the two same? Okay, so answer is, I don't know whether they are the same or not depends on the language specification, or the languages we will say that blanks even when it comes in an identifier, let me just ignore, makes life more complicated, okay, blank can be put anywhere and as designer of the lexical analyzer I have to figure out what it means, okay, so blank here means saying that this is count, ignore these blanks, but don't ignore it, okay, this is to be treated like a blank, similarly this blank, and this blank can be ignored to say this is count, but this blank cannot be ignored and this says this is end of part.

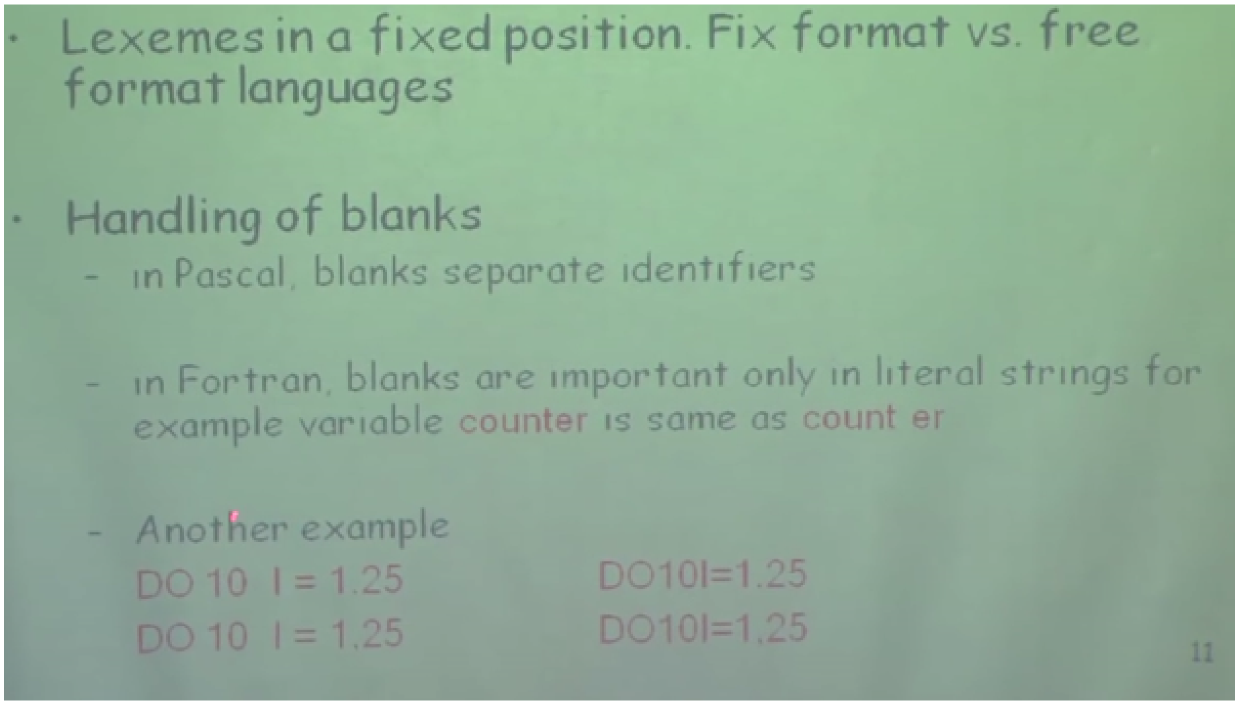
So even blank now becomes context sure, when it occurs that makes lot of difference, so things like important, blanks are just important in literal strings and rest of the places I just



Difficulties in design of lexical analyzers

- Is it as simple as it sounds?
- Lexemes in a fixed position. Fix format vs. free format languages
- Handling of blanks
  - in Pascal, blanks separate identifiers
  - in Fortran, blanks are important only in literal strings for example variable counter is same as count er

ignore blanks, okay as if they don’t exist, okay, so when I write counter versus count blank ER, they are actually the same identifier, okay. And any idea, why such kind of format was prevalent? By design, by accident, what was the reason? I mean see as computer scientist and as student of this any subject I mean we must also know bit of history, okay, so I give you one part of history like how compilers will design initially, right, so is it by accident or by design? No, actually it makes your program highly unreadable, okay, so programmers don't want spaces. Earlier programs were not type then, they were written by hand, then how do I pass it on to the computer? See whatever program you write it, okay, earlier programs and today’s programs are all written by hand, even today we expect that before you go to the lab at least you will have some structure of the code in mind, okay, you will have some structure before you go and sit on terminal and start this, dining on the keyword, okay, but what happened was earlier, okay so, here is another example and this is really much more complex, okay, so take this, okay.

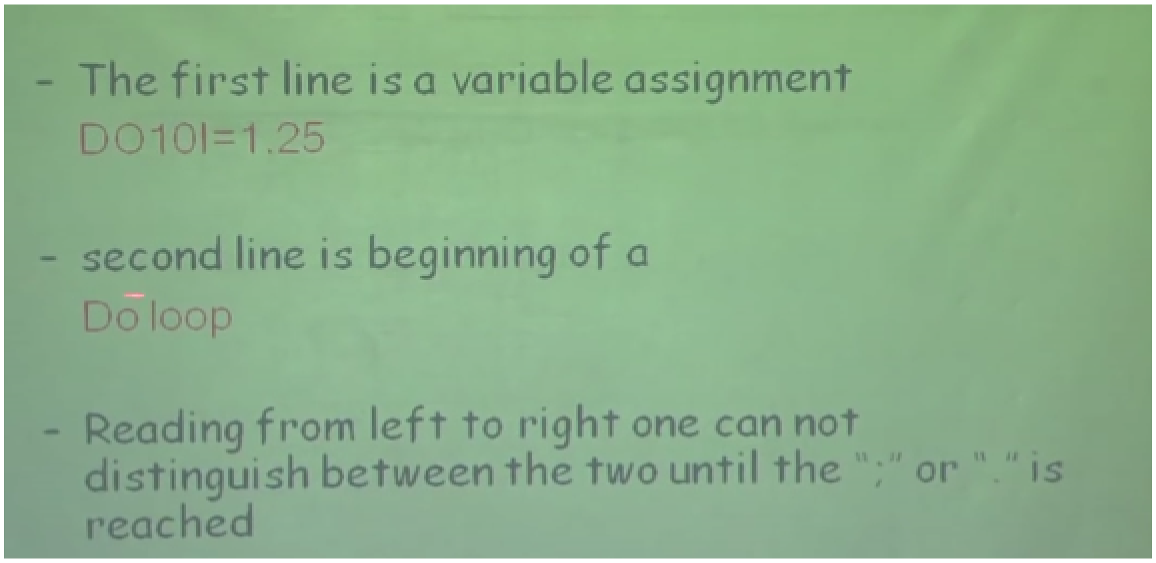


- Lexemes in a fixed position. Fix format vs. free format languages
- Handling of blanks
  - in Pascal, blanks separate identifiers
  - in Fortran, blanks are important only in literal strings for example variable counter is same as count er
  - Another example

```
DO 10  I = 1.25          DO10I=1.25
DO 10  I = 1,25          DO10I=1,25
```

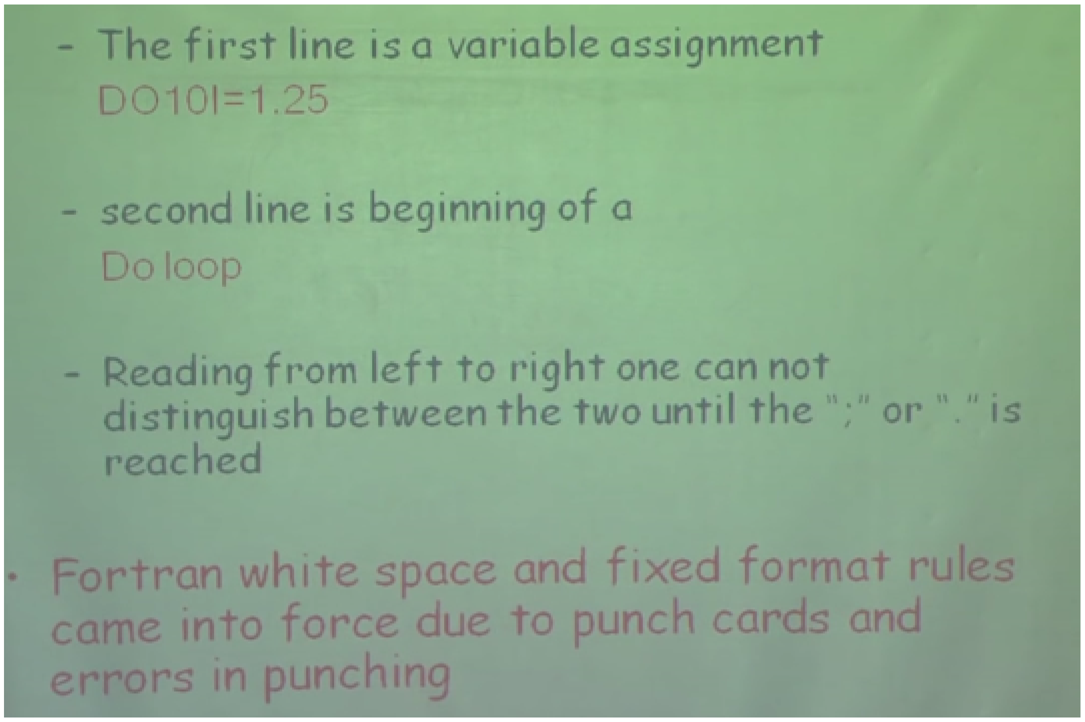
11

And I will come back to this point of blanks once again, okay so take this when I say, DO10I and there's a blank between DO10 and I = 1.25 and this is DO10I = 1,25, okay. Now what in the first case is this is really an identifier which is DO10I which is the assigned value 1.25, variable identifier, okay, and the second one says this is actually a for loop which says that I want to, I take from this statement up to the statement whose level is 10 and the iteration space is given by word bound as 1, and upper bound is 25 in steps of 1, so this is do everything from this statement up to statement 10 for values of I varying from 1 to 25, so that's like a Fortran, but interesting part is that I can also write DO10I in this form, I don't have to put blanks here, okay so this now when I see this format to figure out what I'm dealing with is not an assignment but I am dealing with a Fortran, okay so I have to go back and tokenize this, I must say that oh DO is a keyword and 10 is a label and I is an identifier. So not only blanks are not important but in certain situation even if I don't put blanks here that does not mean that I have the same token I, we need to take DO10I break that into three tokens, okay, and when will I know this, when I encounter either dot here, or a comma here. Here's interesting situation, okay, let me say it doesn't matter how you write, compiler is smart enough to figure out what you meant, right that I could have written in any of these formats and compilers was able to figure out what was the language or what is it and what was the intent of the program, okay, and it really happened the reason when we say it happens by design what it meant was, I mean that was designed because we wanted to prevent certain accidents, okay, so the first line is a variable assignment second line is beginning of a Do loop and reading from left to right one really cannot distinguish



- The first line is a variable assignment
  DO10I=1.25
- second line is beginning of a
  Do loop
- Reading from left to right one can not distinguish between the two until the ";" or "." is reached

between these two till I encounter either this comma or dot, okay and why haven't we used these fixed

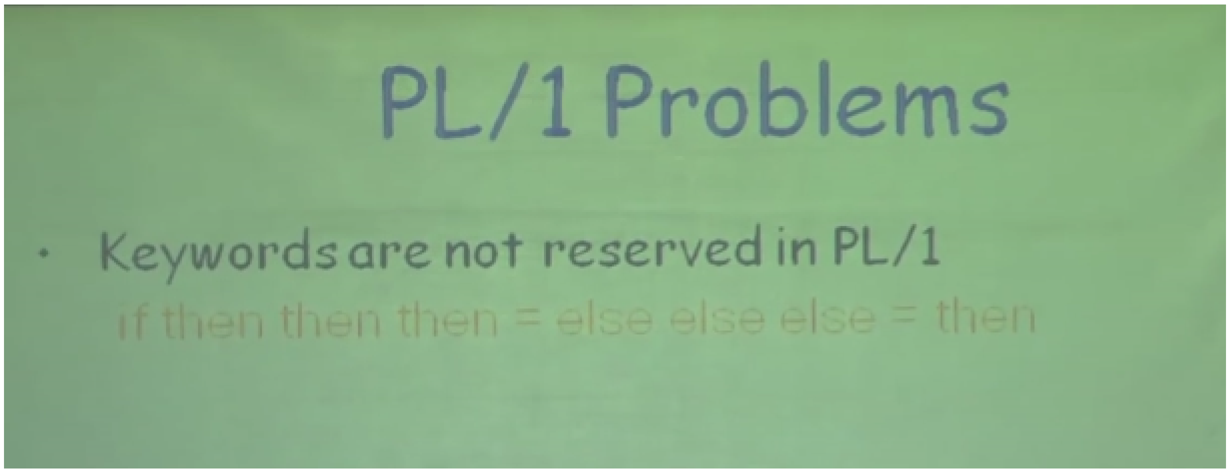


- The first line is a variable assignment
  DO10I=1.25
- second line is beginning of a
  Do loop
- Reading from left to right one can not distinguish between the two until the ";" or "." is reached

- Fortran white space and fixed format rules came into force due to punch cards and errors in punching

formats and we use these ignoring the blanks and not ignoring the blanks in some ways because at that point of time we did not have these terminals, today I mean you have this keyword and terminal and you have editor and you just go and type your program and visually you can see when your program is at least space wise correct or not, and then you can do various kind of formatting, you can do editing before you pass it on to the compiler, okay earlier we did not have that luxury, and really the way programs or type was that we had these punch cards, so you go and type your program in the punch card and it is almost impossible to correct a punch card, you cannot correct a punch card you have to retype the punch card and people did not have that kind of time, okay. So situation was something like this that I typed my program on punch cards each card will have one statement and having the thousand line program is no big

deal, right, if you have thousand lines of program you have thousand punch cards, okay. Now thousand punch card will be a stack of something like toffee, okay.
So this is typically our punch card look, okay, I still have some old punch cards lying with me for historical reasons and just take thousands of them so each punch card will have this one statement and you have thousands stack sim, and then you pass it on to an operator and go, this operator at some point of time it will put it in card readers so there are hundreds of thousands of cards which are submitted every day, this operator puts it and then suppose we have this kind of situation that you made a mistake while typing the punch card, okay, you had no way of correcting it, okay. Now if you come back next day and you find that only error your program was because certain blanks were not inserted in right places, okay, the whole day is wasted, so you replace that punch card come again, so whatever errors programmers would have made while typing punch cards compiler was trying to take care of that, okay I mean today it's not required, okay but for historical reasons when it persists because this language has, you still has some old code and it process, okay.
So really what happened here was that equal punching their program on the start, and the start was the program of punch, okay, and you wanted to make sure that whatever errors you had, during punching those are not really the errors which relate to execution and therefore all such errors you will try to catch in the faith of compilation and you just ignore that, because we did not have scenery to set that point of time, okay and that was an important clues of punch card.
Anyone knows history of all these punch cards? What is the name of this card? Anyone knows what is this card called? Who designed it? Why? Why it was designed? It was designed before computers, it was designed some time in 1880, for sewing machines had cards, but not these kind of cards, they are bolts, so sewing machines have designed cards, okay, but not this kind of cards, I'm talking about this specific card, so this was designed by a statistician called Paul Meier in 1880 and here the machine called tabulator, and the first major use of this tabulator machine and this punch card was done was during US census data in 1880, okay. And really what happened was that if you have this punch card with certain holes you put it in such as stand up and some there'll be some connections which will be made so some needles will go and they can penetrate the holes, they had the connection will be made and certain computation will happen, okay.
So for the first time regulation of all the census data was done, and later on this company okay which was making these tabulators then after several modules, actually emerged into idea, so that was perhaps beginning of IBM in some small data, okay.
So let's move on okay and let’s talk about different kind of problems, okay. So one problem we are saying is all this blank, okay, now what about this? Is this a valid statement in a program? If it is keyword in my programming language, is this a valid statement in my programs? Well, I mean answer is if you say no that's not the correct answer, okay you immediately have to ask the question is keyword a reserved word in my language, so this keyword is not a reserved word, then there is nothing wrong with this and their language is like PL1 which say keyword is not reserved, and if they are not reserved I can write statement like this, which is valid statement,

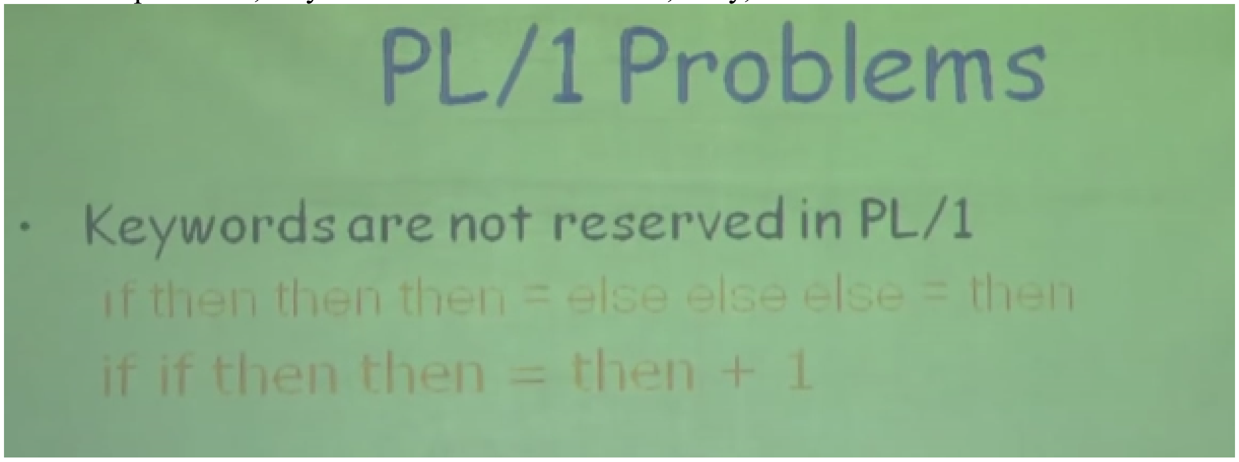


PL/1 Problems

- Keywords are not reserved in PL/1

  if then then then = else else else = then

okay. And what is the statement, this is if then then then = else else else = then. Now it is for compiler to figure out which is a keyword, which is the boolean operator, we can identify, okay, so it says if that is true then then = else else else = then, interesting right? So answer is when I say that whether this is valid or not, answer is not loop but answer is if a result keyword in my language or not, okay and if it is then you know it is not valid but if it is not then I can write any kind of expressions, so you have to be aware of this, okay, here is



PL/1 Problems

- Keywords are not reserved in PL/1

  if then then then = else else else = then

  if if then then = then + 1

another one, if if then then = then + 1, okay, right this is also valid, declarations, okay, here is oh, we are running short of time, so I think we'll have to close here and then we'll move on, and I think the next class is waiting outside, so tomorrow we will start our decision on this one. And those of you who wish can keep these cards as implement in case, I have a lot

Acknowledgement
Ministry of Human Resource & Development
Prof. Phalguni Gupta
Co-ordinator, NPTEL IIT Kanpur
Satyaki Roy
Co Co-ordinator, NPTEL IIT Kanpur
Camera
Ram Chandra
Dilip Tripathi
Padam Shukla
Manoj Shrivastava

Sanjay Mishra

<u>Editing</u>

Ashish Singh
Badal Pradhan
Tapobrata Das
Shubham Rawat
Shikha Gupta
Pradeep Kumar
K.K Mishra
Jai Singh
Sweety Kanaujia
Aradhana Singh
Sweta
Preeti Sachan
Ashutosh Gairola
Dilip Katiyar
Ashutosh Kumar

<u>Light & Sound</u>

Sharwan
Hari Ram

<u>Production Crew</u>

Bhadra Rao
Puneet Kumar Bajpai
Priyanka Singh

<u>Office</u>

Lalty Dutta
Ajay Kanaujia
Shivendra Kumar Tiwari
Saurabh Shukla

<u>Direction</u>

Sanjay Pal

<u>Production Manager</u>

Bharat Lal

an IIT Kanpur Production