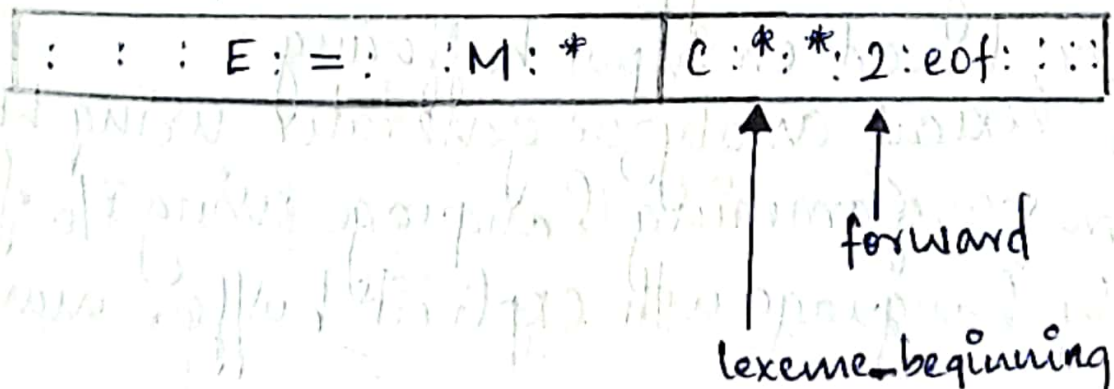
12/12/2025

TUTORIAL - I

Q1. Describe input buffering scheme in lexical analyzer.

Ans: Input buffering is an important technique used in a lexical analyzer to efficiently read input characters and to ensure that the correct lexeme is recognized. To find the right lexeme, it is often necessary to look one or more characters beyond the current lexeme. Hence, buffering techniques are used to safely handle large lookahead and to speed up the lexical analysis process.

To reduce the overhead of reading characters one by one from the input file, a two-buffer scheme is introduced. This scheme consists of 2 buffers, each of size N characters, where N is the number of characters in one disk block. The buffers are filled alternatively using a single system read command. If the number of characters read is less than N, an EOF marker is inserted at the end of the buffer. Techniques such as the use of sentinels are adopted to mark the end of each buffer, which helps in faster processing.



In this scheme, two pointers are maintained:

- lexemeBegin: points to the beginning of the current lexeme
- forward: scans ahead until a pattern match is found

The forward pointer moves through the input to identify a token. Once a lexeme is found, the lexemeBegin pointer is set to the character immediately after the recognized lexeme, and the forward pointer is positioned at the right end of the lexeme. The current lexeme is defined as the set of characters between lexemeBegin and forward.

Code to advance forward pointer:

```
if forward at end of first half then begin
        reload second half;
        forward = forward + 1
end
else if forward at end of second half then begin
        reload second half;
        move forward to beginning of first half.
end
else forward = forward + 1;
```

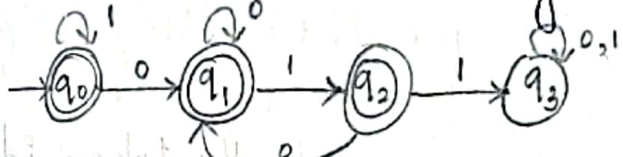
There are 3 general approaches to implement a lexical analyzer, based on input buffering:

- using a lexical analyzer generator using LEX
- a system programming language using I/O facilities
- assembly language with explicit buffer management.

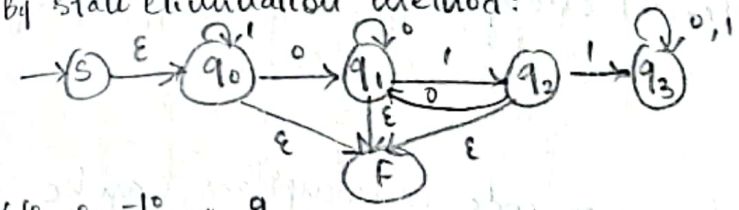
Disadvantages of this scheme:

- Lookahead is limited by buffer size
- Token recognition fails if forward pointer moves beyond buffer length.
- Difficulty to identify tokens needing large lookahead.

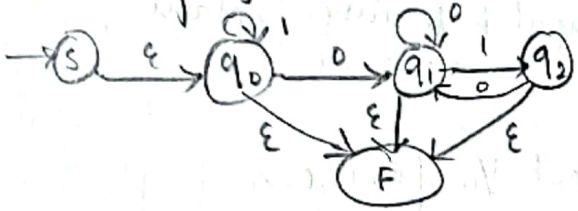
Q2. Construct a regular expression to denote a language L over $\Sigma = \{0, 1\}$ accepting all strings of 0's and 1's that do not contain substring 011.



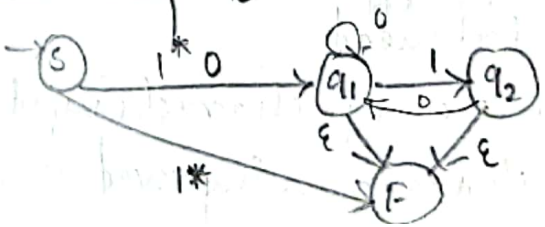
By state elimination method:



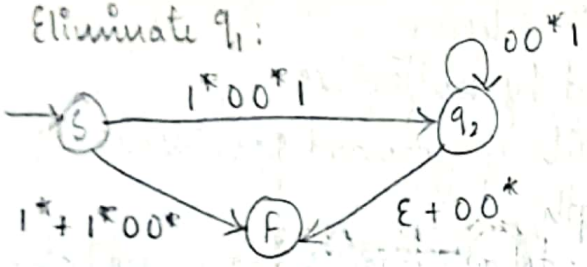
Eliminating q_3:



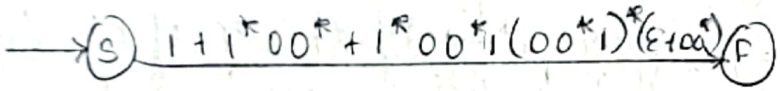
Eliminating q_0:



Eliminate q_1:



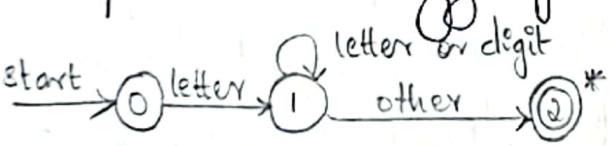
Eliminate q_2:



Regular Expression: $1+1^*00^* + 1^*00^*1(00^*1)^*(\varepsilon+00^*)$

$= 1 + 1^*00^* + 1^*00^*1(00^*1)^*0^*$

Q3. Develop a lexical analyzer for the token identifier.



Q4. Scanning of source code in compilers can be speeded up using input buffering. Explain.

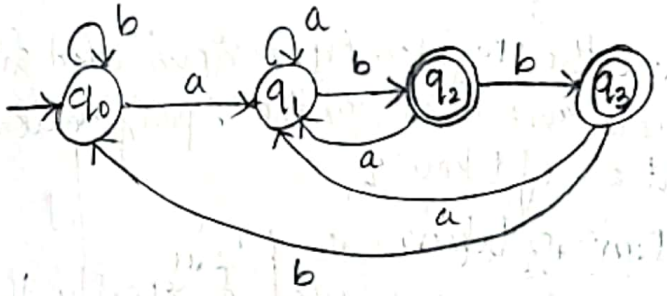
Ans: Input buffering:

- reduces frequent I/O operations
- uses large memory buffers
- allows efficient lookahead
- Eliminates character-by-character input.

Thus, overall scanning performance is improved.

Q5. Draw the DFA for the regular expression $(a|b)^*(abb|a^*b)$

Ans:



Q6. Explain compiler writing tools.

Ans: Compiler writers use software development tools and more specialized tools for implementing various phases of a compiler. Some commonly used compiler writing tools include:

- Parser Generator:
  Input: Grammatical description of a programming language.
  Output: Syntax Analyzers.
- Scanner Generators:
  Input: Regular expression description of the tokens of a language.
  Output: Lexical Analyzers.
- Syntax-directed translation engine
  Input: parse tree
  Output: Intermediate code
- Automatic Code Generator:
  Input: Intermediate language

Output: Machine language.
- Data-flow analysis engines.
- Compiler Construction toolkits.

Q7. Explain how the regular expressions and finite state automata are used for the specification and recognition of tokens?

Ans: Specification of tokens:

Regular expressions are used to specify the structure of tokens. They allow defining patterns for tokens. Each token has a corresponding regular expressions.

Eg: Identifier: letter (letter | digit)*

Relational Operator: < | <= | <> | > | >=

Regular expressions precisely describe the set of valid lexemes for each token.

Recognition of Tokens:

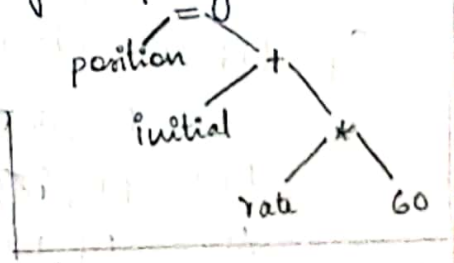
Regular expression are converted into finite automata. These automata read the input characters one by one. If the input string leads to an accepting state, the token is recognized. DFA are commonly used for efficiency.

Q8. Draw the transition diagram for the.

Q8. Explain the working of different phases of a compiler. Illustrate with a source language statement.

Ans: A compiler translates a source program into an equivalent target program through a sequence of well-defined phases. Each phase performs a specific task.

- Lexical Analysis:
  - Reads source program character.
  - Groups characters into lexemes.
  - Produces tokens and removes whitespace and comments.
  - Example token sequence for
    position = initial + rate * 60
    <id,1> <=> <id,2> <+> <id,3> <*> <60>
- Syntax Analysis:
  - Checks grammatical structure of the program.
  - Constructs a syntax tree.
  - Detects syntax errors.



- Semantic Analysis:
  - Checks semantic correctness.
  - Performs type checking.
  - Inserts type conversion if required.
- Intermediate code generation:
  - Converts syntax tree into machine-independent intermediate code.

- uses three-address code.

Eg: $t_1 = \text{inttofloat}(60)$
$t_2 = id3 * t_1$
$t_3 = id2 + t_2$
$id1 = t_3$

- Code optimization:
- Improves intermediate code
- Reduces execution time and code size

$t_1 = id3 * 60.0$
$id1 = id2 + t_1$

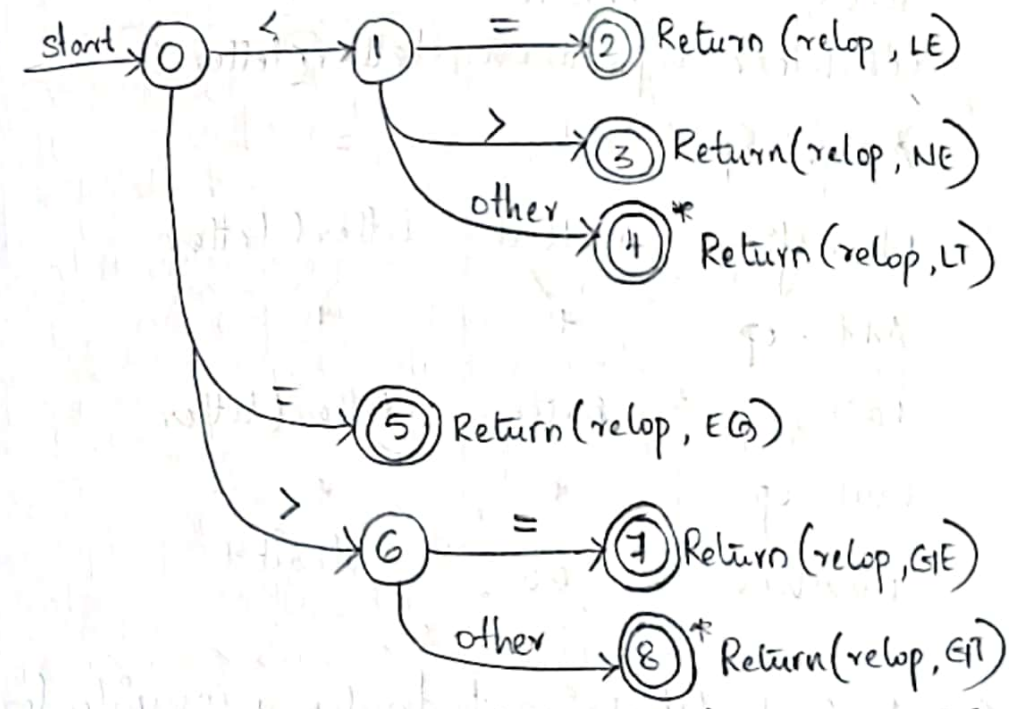
- Target Code Generation
  - Converts optimized code to target machine code
  - Assigns registers and memory.

Eg:
```
LDF  R2, id3
MULF R2, #60.0
LDF  R1, id2
ADDF R1, R2
STF  id1, R1
```

Q9. Draw the transition diagram for the regular definition

relop → <I<=I=I<>I>=I>

Ans:



Q10. With an example source language statement, explain tokens, lexemes and patterns

Ans: Token – A sequence of characters that is treated as a single logical unit by the compiler
eg: Identifier, Keyword, operator, constant.

Lexeme – Actual sequence of characters in the source program that matches the pattern of a token.

Pattern – Rule or regular expressions that defines the structure of lexemes for a token.

For the statement:

position = initial + rate * 60

Token	Lexeme	Pattern
Identifier	position	letter(letter
Assign-op	=	=
Identifier	initial	letter(letter
Add-op	+	+
rate	rate	letter(letter
mul-op	*	*
number	60	digit+

Q11. Apply bootstrapping to develop a compiler for a new high level language P on machine N.

Ans: Bootstrapping is the process of developing a compiler using the language itself.

Steps:

1. Write compiler for P in existing language.
2. Compile using existing compiler.
3. Obtain native compiler for P on machine N.

Q12. Now I have a compiler for P on machine N. Apply bootstrapping to obtain a compiler for P on machine M.

Ans: To obtain a compiler for language P that runs on machine M, bootstrapping is used along with Cross-compiler.

Steps:

1. Assume an existing compiler C that runs on machine N and generates code for machine M.
2. Write the compiler for language P in language P itself.
3. Use compiler C to compile the source of the P compiler.
4. The output is an executable compiler that runs on machine M and produces code for machine M.
5. This compiler can now compile P programs directly on machine M.

Q13. Define cross-compiler.

Ans: A cross-compiler is a compiler that runs on one machine but generates executable code for another machine.

- The machine on which the compiler runs is called the host.
- The machine for which the code is generated is called the target.
- Cross-compilers are commonly used in bootstrapping and embedded-systems.

Q14. For a source language statement, a = b * c − 2, where a, b, and c are float variables, * and − represent multiplication and subtraction on same data types, show the input and output at each of the compiler phases.

Ans:

1. lexical Analysis:

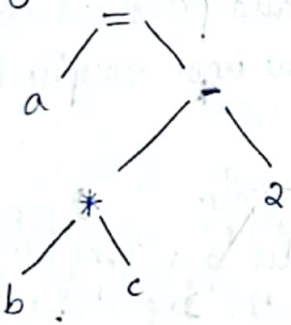
   Input: a = b * c − 2

   Output: <id, a> <=> <id, b> <*> <id, c> <−> <num, 2>

2. Syntax analysis:

   Input: Token stream.

   Output: Syntax tree



3. Semantic Analysis

   Input: Syntax Tree

   Output: Annotated syntax tree (type-checked) (with type conversion if required)

4. Intermediate code Generation:

   Input: Annotated syntax tree

   Output: Three-address code

$$t_1 = b * c$$
$$t_2 = t_1 - 2$$
$$a = t_2$$

5. Code optimization

   Input: Intermediate code

   Output: Optimized intermediate code.

$$t_1 = b * c$$
$$a = t_1 - 2$$

6. Target Code Generation

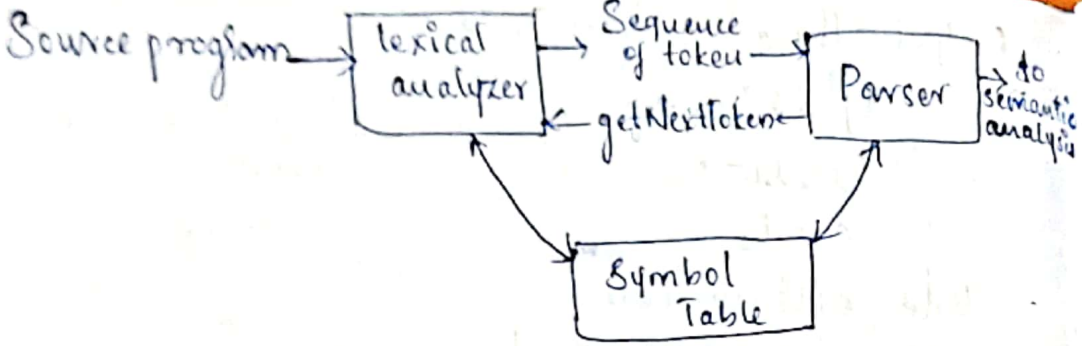
   Input: Optimized intermediate code

   Output: Target Machine code.

```
LOAD R1, b
MUL R1, c
SUB R1, #2
STORE a, R1
```

Q15. Explain the role of lexical analyzer.

Ans The main task of lexical analyzer is to read the input characters of the source program, group them into lexemes and produce as output a sequence of tokens for each lexeme in the source program.
- The stream of tokens is sent to the parser for syntax analysis



- lexical analyzer also interacts with symbol table.

Other tasks:

- Eliminates blanks, tabs, newlines and comments from the input.
- Detects lexical errors like invalid characters or sequences
- The expansion of macros may also be performed by lexical analyzer.

Q 16. Explain bootstrapping with example.

Ans: Bootstrapping is the process of writing a compiler for a programming language in the same language so that it can compile itself. It is used to make a compiler self-hosting and ensures correctness and optimization.

Steps:

1. Write a basic version of the compiler in another language.
2. Write full compiler in the target language itself.
3. Use the basic compiler to compile the full compiler.
4. After this, the compiler can compile its own source code and produce an optimized version.

Eg: GCC is written in C but compiled using GCC itself.

The pascal compiler was first written in assembly, then rewritten in pascal.

Q17. Write a short note on error handling.

Ans: Each phase of a compiler can encounter errors. After detecting an error, the compiler must handle the error and continue compilation so that more errors in the source program can be detected.

A compiler that stops after finding the first error is not helpful.

In lexical analysis, a character sequence that cannot be matched with any valid token is called a lexical error.

The simplest error recovery method is panic mode recovery, where successive characters are deleted from the input until a valid token is found.

Other possible error-recovery actions are:

- Deleting a character
- Inserting a missing character

- Replacing a character
- Transposing two adjacent characters.

Qes. Discuss the role of symbol table in compiler design process.

Ans: The symbol table stores information about all identifiers in the source program including:
- Name, type, storage location and scope.
- Procedure details (arguments, return type)

Uses:
- Lexical Analysis: insert identifiers.
- Semantic Analysis: Type and scope checking.
- Code generation: Retrieve information for target code.

It allows fast lookup and retrieval, aiding all phases of the compiler.