COMPILER DESIGN

UNIT-I :-

Introduction: Language processors, the structure of a compiler, the science of building a compiler, Programming language basics.

Lexical Analysis:- The Role of the Lexical Analyzer, Input Buffering, Recognition of Tokens, The Lexical Analyzer Generator Lex, Finite Automata, From Regular expressions to Automata, Design of a Lexical Analyzer generator, optimization of DFA-Based Pattern matchers.

=

What is a Compiler?

* The software that accepts text in some language as input and produces text in another language as output while preserving the actual meaning of the original text is called a translator.
* The input language is called the source language; the output language is called the target or objet language.
* A compiler basically a translator. It translates a source program written in some high-level programming language such as pascal/C/C++ into machine language for computers, such as the Intel pentium IV/AMD Processor machine

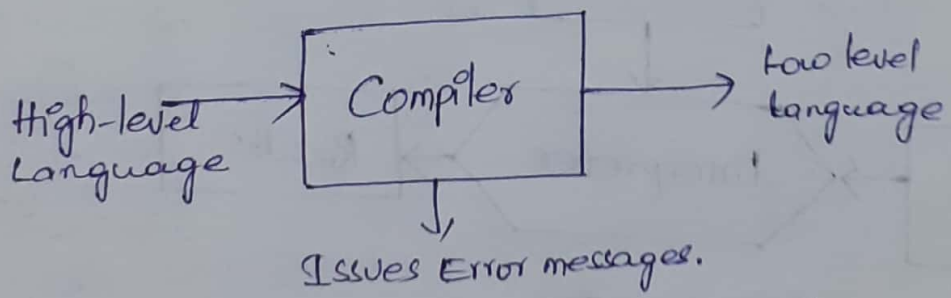


Fig: Compiler Input and output

* The generated source code can be used for execution as many times as needed. In addition to this, a compiler even reports about errors in the source program.

Compiler Vs. Interpreter

* An interpreter is another way of implementing a programming language
* An interpreter works similar to a compiler in most of the phases like Lexical, syntax, and semantic. This analysis is performed for each single statement and when the statement is error free, instead of generating the corresponding code, the statement is executed and the result of execution is displayed.
* In an interpreter, since every time the program has to be processed for and has to be checked for errors, it is slower than the compiler program.
* writing an interpreter program is simple task than writing a compiler program because a compiler processes ~~whole~~ the program as a whole, where as an interpreter process a program one line at a time.
* An interpreter first reads the source program line by line, written in a high level language; it also reads the data for this program; then it runs or executes the program directly against the data to produce results.
* Interpreter does not produce machine code or object code.

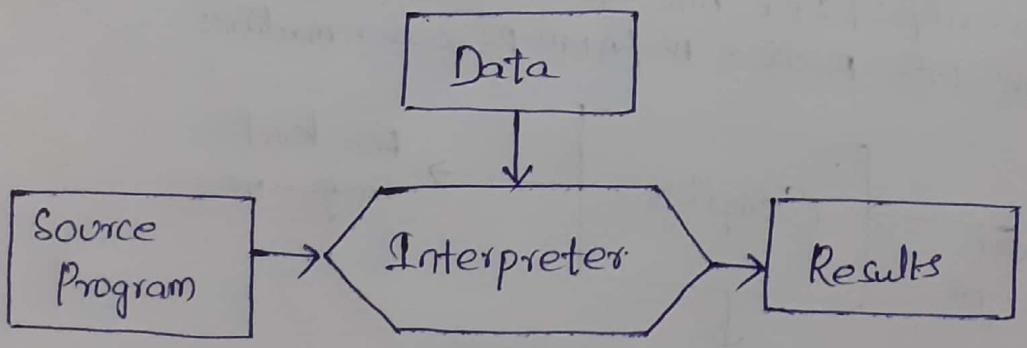


Fig: Interpreter

Examples of Interpreter, Unix shell Interpreter, Python Interpreter. JavaScript, Lisp, Basic, Perl, SMALLTALK, APL etc.

=

Typical Language processing System.

To understand the importance of a compiler, It is essetial to go through the typical language processing system.

* Given a high-level language program, Let us see how we get the executable code. In addition to a compiler, other programs (translators) are needed to generate an executable code.

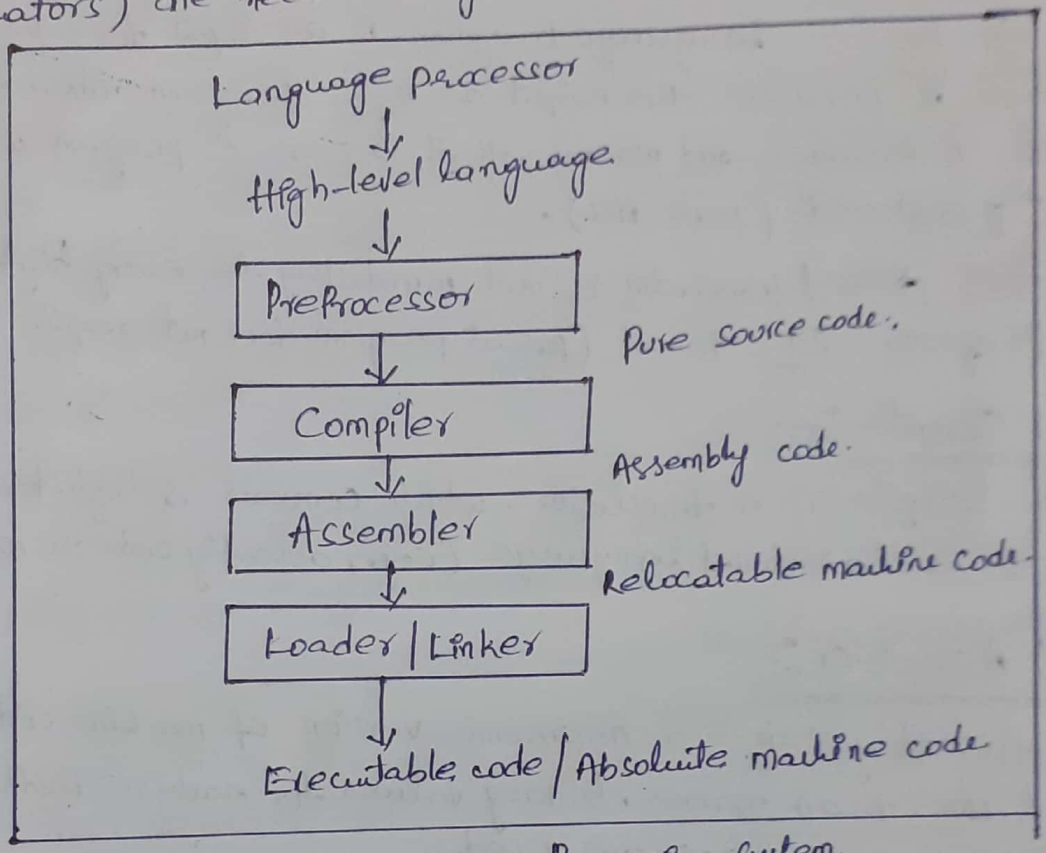


Fig: Typical Language Processing System.

* In this typical language Processing system, It requires different ~~Soo~~ Software. The first translator needed here is the Preprocessor.

Preprocessor :-

* A ~~Soo~~ source program may be divided into modules stored in separate files and may consist of macros.
* A Preprocessor produces input to a compiler ie, Preprocessor processes the source code before the compilation and produces a code that can be more efficiently used by the compiler.

* A preprocessor performs various operations like Macro Processing (macros are shorthands for longer constructs), file inclusion (code from all files is appended in text), Language Extensions (A preprocessor can add new functionalities to extend the language by built-in macros - for example, adding database query facilities to the language).
* If the 'C' language program is an input for a preprocessor, then it produces the output as a 'C' program where there are no #I includes and macros, that is, a C program with only C statements (pure HLL).
* This phase/translator is not mandatory for every high-level language program. Ex: Pascal (Pascal program does not require Preprocessing)

(ii) Compiler :-

* Compiler is a translator, which converts a high-level language to a low-level language (e.g., assembly code or machine code)

(iii) Assembler :-

* Assembly code is a mnemonic version of machine code. In which rather than names, binary values for machine instructions and memory addresses are used.
* Assembler translates assembly code to machine code.
* An assembler assigns memory locations or addresses to symbols / identifiers. It should use these addresses in generating the target language (machine language)
* Assembler should ensure that the same address must be used for all the occurences of a given identifier, and no two identifiers are assigned with the same address.
* Assembler ensures this mechanism in two different passes. In first pass, whenever a new identifier is encountered, it assigns address to it. Store the identifier along with address in a symbol table. during second pass, whenever an identifier is seen, then its address is retrieved from the symbol table and that value is used in the generated machine code.

(iv) Loader / Linker :—

* TO Convert the relocatable machine code to the executable code, one more translator is required and this is called the loader / linker.
* Linking is a process where a linker takes several object files and libraries as input and produces one executable object file.
* Loading is a process where a loader loads an executable file into memory, initializes the register, heap, data etc. and starts the execution of the program.
* Above all discussed are the different translators must be used for execution of the program.
* Among all designing a compiler is the most complex task.

The design of the first FORTRAN compiler took 18 man years the complexity of the design depends on the source language.

* currently many automated tools are available, with modern compilers tools like YACC (Yet another compiler compiler), LEX (Lexical Analyzer) and data flow engine, makes the design of a compiler easy.

=

** Design phases of a Compiler.

Designing a compiler is a compler task. The design is divided into simpler subtasks called phases.

* A phase converts one form of represetation to another form. The task of a compiler is divided into six phases as shown in the given figure.

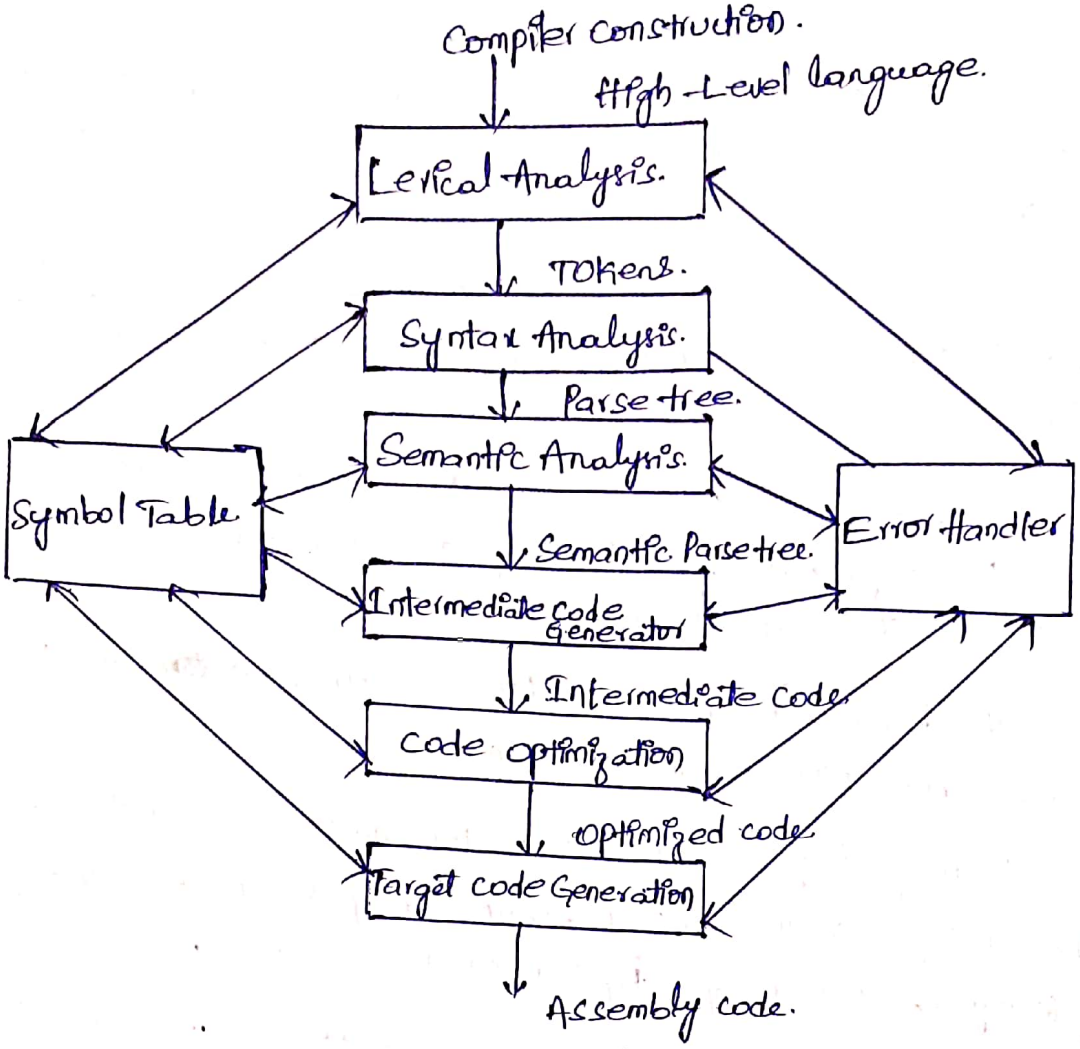
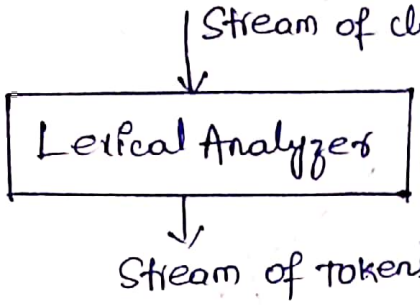


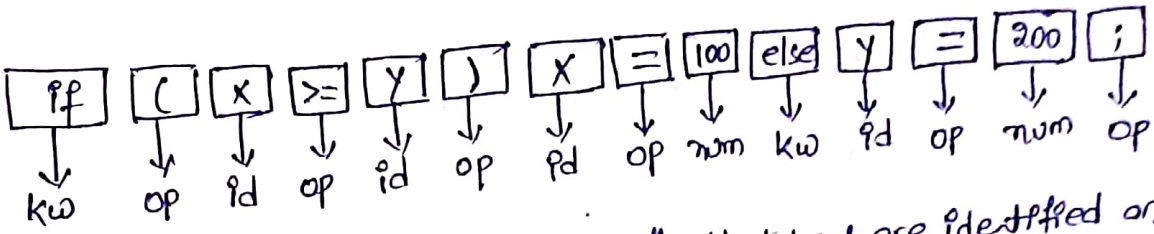
Fig: phases of a compiler.

(1) Lexical Analysis:—

* Lexical analyzer is also called as Scanner
* The scanner groups the input stream of characters into a stream of tokens and construct a symbol table.
* The tokens in any high-level language can include
  * Keywords
  * identifiers
  * Operators
  * Constants: numeric, character and special characters.
  * Comments
  * punctuation symbols.
* Lexical analyzer groups input stream of characters into logical units called tokens.

kw - keyword
op - operator
id - identifier
num - number.





Id, op, kw, num are the tokens. Totally 14 tokens are identified and sent to the next phase.

(ii) * Regular Expressions are used to define the rules for recognizing the tokens by a Scanner (or lexical analyzer)

* The Scanner is implemented as a finite state automata.
* Automated Scanner generator tools are Lex and Flex. Lex is a faster version of Flex.

(ii) Syntax Analysis :—

* Lexical Syntax Analyzer is also called as Parser., it checks the syntax of a given statement.
* The parser groups stream of tokens into a hierarchical structure called the parse tree.
* Inorder to check the syntax, first the standard syntax of all statements must be known. Generally, the syntax rules for any High level language are given by Context-free grammers (CFG) or Backus-Naurform (BNF)
* The CFG, which describes all assignment statements with arithematic expressions is as follows.

Ex: To check the syntax of the statement 6x(8+4) CFG is

Expr → Expr op Expr
| (expr)
| num

op → + | - | * | /

Using this grammar, the syntax of the given statement is verified by deriving the statement from grammar.

Ex:

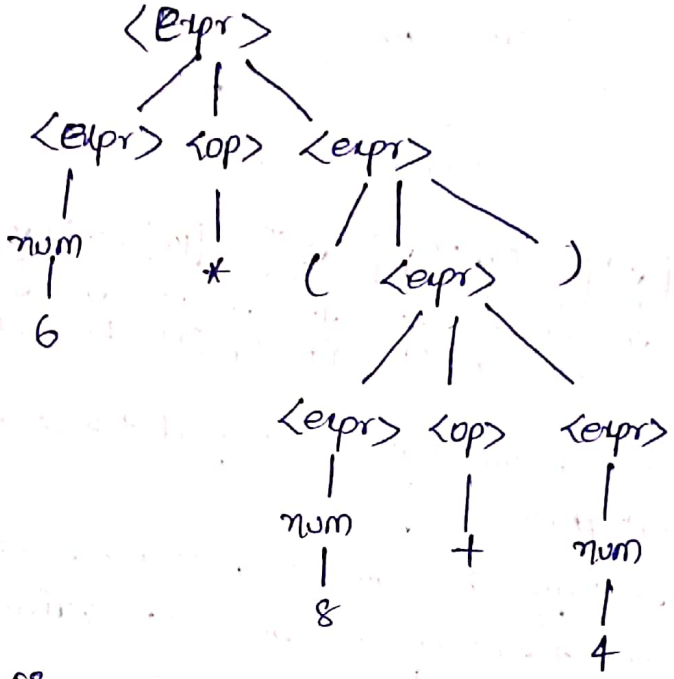


Fig: parse tree for 6*(8+4)

* So the output of the parser is a parse tree representation of the program.
* The parser is generally represented as push-down automata. many automated tools are available for designing parsers. YACC and Bison are widely used tools. They are used for generating bottom-up parsers in 'C' Language.
* Bison is a faster version of YACC.

(ii) Semantic Analysis :-

* Semantic Analyzer checks semantics, that is whether the language constructs are meaningful.

for example, given a statement A+B, let us see how a Parser and a semantic analyzer work.

* Semantic analyzer checks whether the two operands are type-compatible or not. Static type checking is the basic function of the semantic analyzer. The output of the semantic analysis phase is an ~~ano~~ annotated parse tree.
* Attribute grammars are used to describe the semantics of a Program

(IV) Intermediate Code Generator :-

Intermediate code generator converts the parse tree into a linear representation called Intermediate Representation (IR) of the program.

* Intermediate representations may be the assembly language or an abstract syntax tree.
* An intermediate language/code is often used by many compilers for analyzing and optimizing the source program.
* IR can be represented in different ways :
  1) As a syntax tree
  2) As a directed acyclic graph (DAG)
  3) As a postfix notation.
  4) As a three-address code.

Example of IR in three-address code is as follows.

```
temp1 = int2float (sqrt(t));
temp2 = g * temp1;
temp3 = 0.5 * temp2;
dist  = temp3;
```

* The main purpose of IR is to make target code generation easier.

(V) Code optimizer :-

* Compile time code optimization involves static analysis of the intermediate code to remove extraneous operations.
* The main goal of code optimizer is to improve the efficiency hence, it always tries to reduce the number of instructions.
- Once the number of instruction is reduced, the code runs faster and occupies less space.

Eg: constant folding.

I := 4+j-5; can be optimized as I := j-1;

or

I = 3; j = I+2; can be optimized as I=3; j=5

=

(vi) Target Code Generator :-

The code generator's main function is to translate the intermediate representation of the source code to the target machine code. The machine code may be an executable binary or assembly code, or another high-level language.

* This phase takes each IR statement and generates an equivalent machine instruction.

for the following source text

```
if (x<=3) then y:=2*x;
else y:=5;
```

The target code generated is

Address	Instruction
10	Cmp x 3
11	JG 14
12	MUL 2 x temp
13	JMP 16
14	Mov y 5

* To design a compiler, all the above phases are necessary; additionally, two more routines are also import. They are, symbol table manager and error handler.

(vii) Symbol table manager and Error Handler.

The Symbol table manager is used to store the record for each identifier, procedure encountered in the source program. with relavant attributes.

* The attributes of identifier can be name, token, type, address, and so on.
* For procedures, a record is stored in the symbol table with attributes like return type, number of parameters, type of each Parameter, and so on.
* whenever any phase encounters new information, records are stored in the symbol table and ~~where~~ whenever any phase

~~encounters new information~~ requires any information, records are retrieved from the Symbol table.

* Syntax analyzer stores the type of information; this is later used by the Semantic analyzer for type checking.

*The Error Handler:- Is used to report and recover from errors encountered in the source program. A good compiler should not stop on seeing the very first error. Each phase may encounter errors, so to deal with errors and to continue further, error handlers are required in each phase.

Compiler Design Tools.

Many automated tools are available to design and enhance the performance of a compiler. Some of them are as follows.

1. Automatic parser Generators:-

With the help of CFG, these generators produce Syntax analyzers (parse tree). Out of all phases of a compiler, the most complex is parsing. The tools available for parsing are YACC and Bison.

2) Scanner Generators:-

These generators automatically generate lexical analyzers (the stream of tokens), for example, Lex, Flex.

3) Syntax - Directed Translation Engine:-

These engines take the parse tree as input and produce intermediate code with the three-address format.

4) Automatic code Generators:-

These take collection of rules that define the translation of each operation of the intermediate language for

the target machine.

5) Data-flow Engines:-

To perform good code optimization, we need to understand how information flows across different parts of a program. The "data flow engines" gathers the information about how values are transmitted from one part to other parts of the program.

6) Global optimizers:-

Global optimizer is a language and compiler independent. It can be retargeted and it supports a number of architectures. It is useful if you need a back end for the compiler that handles code optimization.

=

Retargeting:- Creating more and more compilers for the same source language but for different machines is called retargeting.

* Retargetable compiler, also called cross compiler. It is a compiler that can be modified easily to generate code for different architecture. But the code generated is less efficient.

For example:- If there is a 'C' compiler that produces code for poor machine T800, 8-bit processor, then we want a compiler for a better machine, that is P860, 68-bit processor.

* To design a 'C' compiler for P860, we don't have to start the design from lexical analysis. As front end is independent of target machine and completely dependent on source language, take the front end of T800 and design the back end for P860 machine. Combine the front end and back end to get the 'C' compiler for P860. With this advantage we can create as many as compilers for different architectures.

=

Two-Stage Model of Compilation

1. Analysis :- The analysis part of compiler analyses breaks up the source program into constituent pieces and creates an intermediate representation of source text.

2. Synthesis :- Synthesis part of compiler constructs the desired target machine code from the intermediate represetation.

* Most compilers are designed as two-stage systems. The first stage focuses on analyzing the source program and transforming it into intermediate represetation. This is called the front end of the system.
* The second stage focuses on the synthesis part where the intermediate code is converted to the target code.
* The front end is designed with a focus on the source language to check for errors and generate a common intermediate represetation that is error free to the back end.
* The backend can be designed to take the intermediate code that may be from any source language and generate the target code that is efficient and correct.
* Both front end and back end can use the common optimizer, which works on intermediate represetation.

Analysis of the source program.

Analysis deals with analyzing the source program and preparing the intermediate form.

* Lexical Analysis
* Syntax Analysis
* Semantic Analysis.

Synthesis

Synthesis concerns issues involving generating code in the target language. It usually consists of the following phases:

* Intermediate code generation
* Code optimization
* Final code generation.

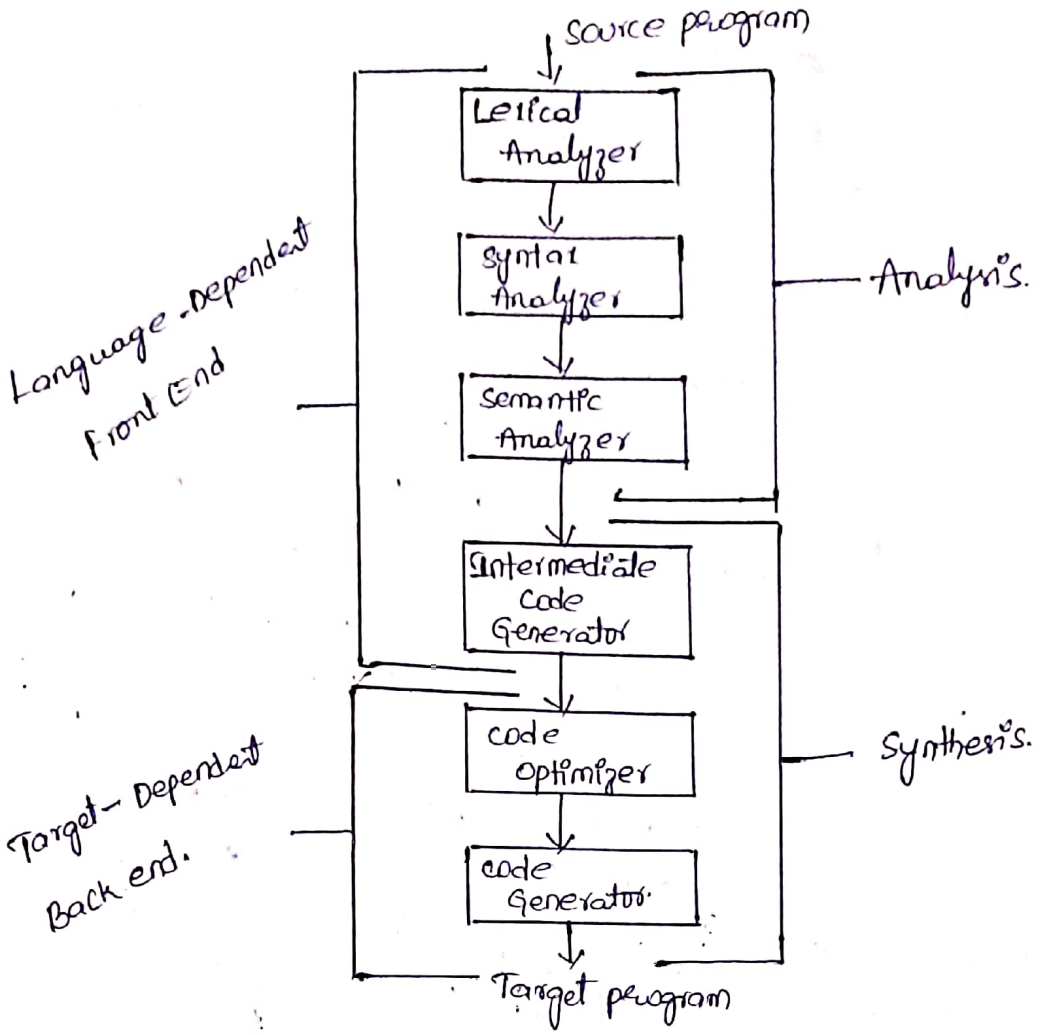


Fig: compiler front end and Back end.

Example: compiler phases.

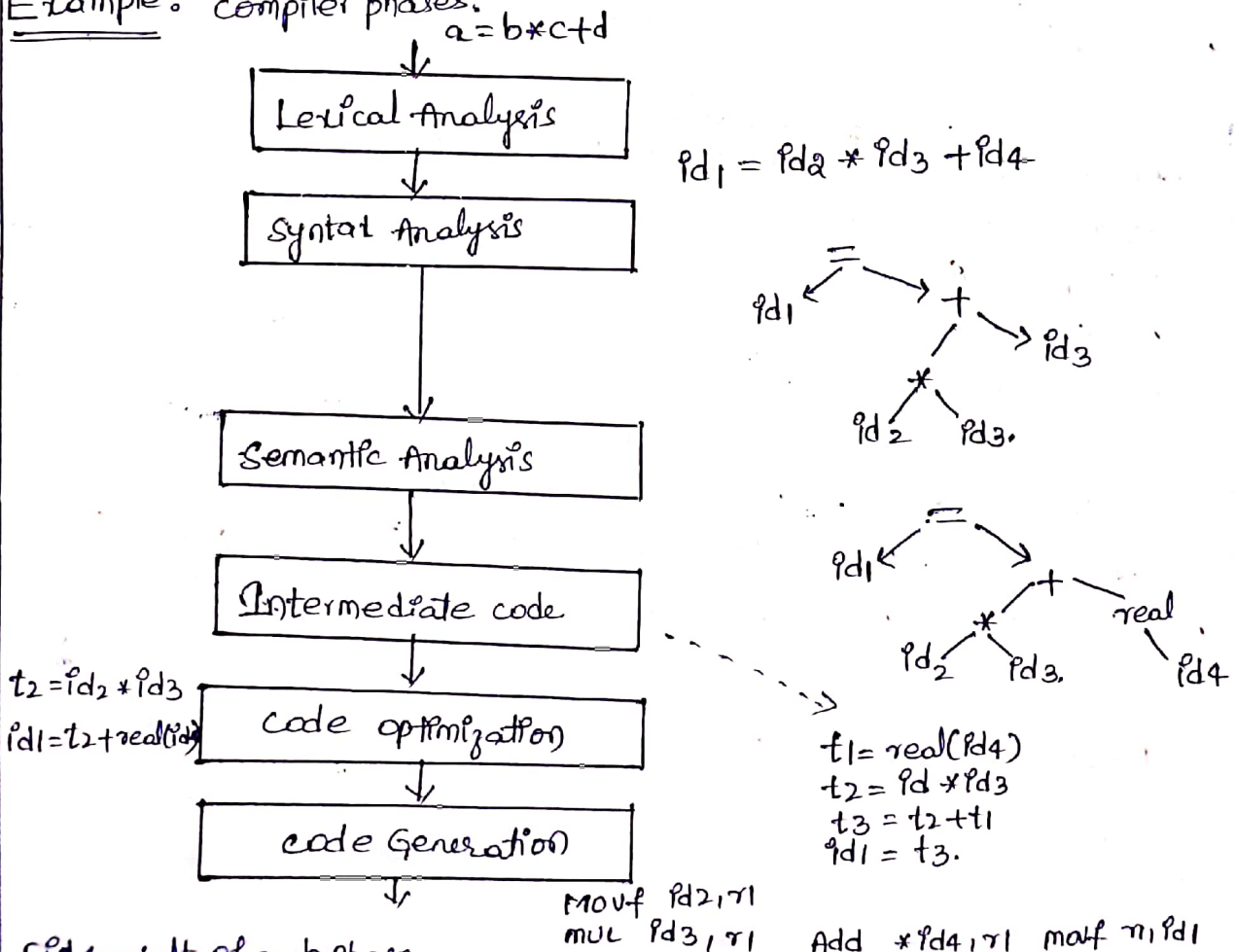


Fig: result of each phase.

The Science of building a compiler.

Building a compiler is a challenging task. The main job of the compiler is to accept the source program of any length (or size) and convert into suitable target program.

* compilers study is focused mainly on study of how to design the correct mathematical model and choose correct algorithms, in order to balance the need of generality and efficiency.
* Correct mathematical models like, finite-state machines and regular expressions are available for describing tokens in a lexical analysis for a compiler.
* In compiler design, the code optimization produce code which is more efficient than the previous code. This code should be faster than any other code that performs the same task.

The objectives to be fulfilled by the compiler optimizations are

① The meaning of the compiled program must be preserved

② Optimization should improve program performance

③ Time required for compilation should be reasonable.

④ Management of engineering effort is must.

Lexical Analysis.

Role of the lexical Analyzer.

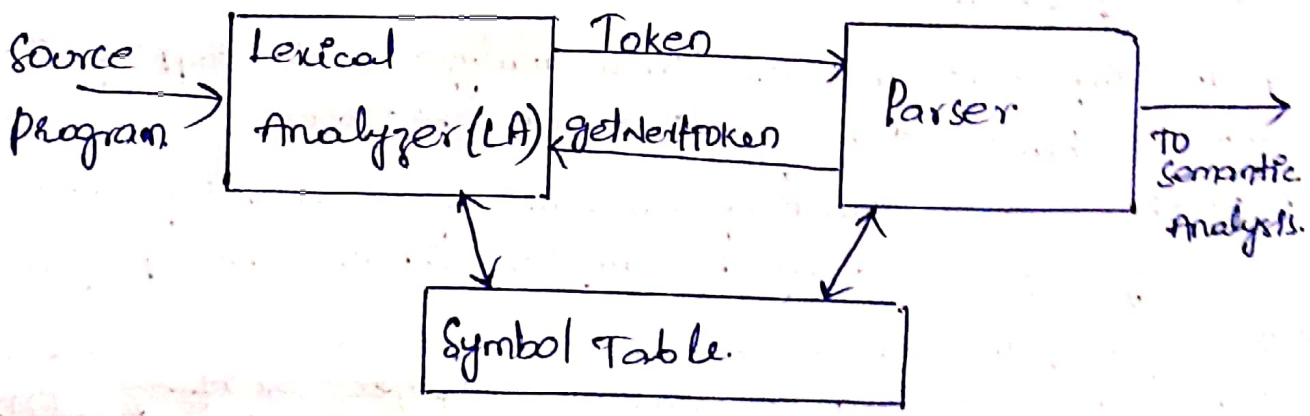


Fig: Interaction b/w the LA and Parser.

* Lexical Analyzer is the first phase of compiler. It reads stream of characters as source and converts into stream of tokens. LA is also called as Scanner.
* Don't think that LA reads the complete text and sends stream of tokens. only on getting request from the parser the lexical analyzer reads the next token and sends the next token to the parser.
* If a token is an Identifier or a procedure, then lexical analyzer stores the token in the Symbol table.
* Here, 'getNextToken' command helps LA to read characters from its input until it identifies next lexeme and produce next token for it.

For Example: If there is a source text A+B. Lexical Analyzer doesn't read A+B at once and sends tokens id, +, id.

* on getting request from parser, the LA reads the first string. A and recognizes it as the token id and stores it in symbol table, and sends the token id to the Parser.
* on the next request it reads + and sends the operator as it is as a token.
* on the third request it reads string B, recognizes that as token id, stores it in the symbol table and sends the token id to the parser.

out of all phases, a compiler spends much time in lexical analysis phase. Hence if this phase is implemented efficiently, this contributes to the overall efficiency of the compiler.

Functions of LA are

* The important task of lexical analyzer is to stripping out comments, blanks, tabs, newlines and whitespaces.
* Another task is ~~correlating error messages~~ correlating error messages generated by the compiler with the source program.
* It may keep track of number of newline characters and line numbers.

* converts sequence of characters into sequence of tokens and also keep track of line numbers.

Lexical Analysis can also be divided into two phases.

① Scanning ② Lexical Analysis.

1) Scanning :- The scanner is responsible for doing very simple tasks and doesn't require tokenization of the input.

Eg: Deletion of comments combining consequitive white spaces to one white space.

2) Lexical Analysis :- more complex operation is performed Scanner produce sequence of tokens as output. LA includes information about identifiers into the symbol table.

Lexical Analysis Vs parsing.

Lexical Analysis and Parsing can be combined in one phase but there are some advantages for doing it separately.

① Both of them do similar things. But the LA deals with simple non-recursive constructs of the language and parser deals with recursive constructs of the language

② Efficiency of the compiler is improved. Input characters can be read by specialized buffering techniques, which speed up the compiler.

③ portability of the compiler is enhanced

④ LA recognizes the smallest meaningful units called tokens and parser works on these tokens to recognize meaningful structures in programming language.

Tokens, Patterns and lexemes :-

When talking about lexical analysis, most often we use the following terms : lexemes, tokens and patterns.

Token:- It is a group of characters with logical meaning
Token is a logical building block of the language.
Example: Id, Keyword, Num etc.

Pattern:- It is a rule that describes the characters that can produce a token.
→ It is expressed as regular expression
→ Input stream of characters are matched with Patterns and tokens are identified

Example:- pattern/rule for identifier is that it should start with a letter followed by any number of letters or digits. this is given by the regular expression : $[A-Za-z][A-Za-z0-9]^*$
using this pattern, the given input strings "xyz", abcde, a7b82 are recognized as tokenid.

Lexeme:- A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
Ex: int is identified as token keyword. there int is lexeme, keyword is token.

Example:-

float key = 1.2;

lexeme	token	pattern.
float	keyword	float
key	Id	A letter followed by number of letters or digits.
=	relop	<|>|<=|>=|=|!=|==
1.2	num	Any numeric constant
;	punctuation	;

* Design the complexity of lexical analyzer mainly dependent on language conventions.

Attributes for tokens

When a token represents many lexemes, additional information must be provided by the scanner or lexical analyzer about the particular lexeme of a token.

Ex: The token id matches with x or y or z. But it is essential to the code generator to know which string is actually matched.

Example: $A = B + D^3$

$A = B + D * * 3.$

This is written as sequence of pairs as shown below.

<id, pointer to symbol_table for A>

<assign-op>

<id, pointer to symbol table for B>

<add-op>

<id, pointer to symbol table entry for D>

<exp-op>

<number, integer value 3>

=

Lexical Errors

Error recovery in lexical analysis.

Generally, errors often detected in lexical analysis are as follows.

1) Numeric literals that are too long
2) Long identifiers
3) Input characters that are not in the source language

How to recover from errors

There are four approaches to recover from lexical errors

1) Delete:- This is the most simplest and most important operation. If there is an unknown character, simply delete it. Deletion of character is also called panic-mode.

This is used by many compilers. However it has certain disadvantages.

* The meaning of the program may get changed
* The whole input may get deleted in the process.

Example:- "charr" can be corrected as 'char' by deleting r.

Insert:- Insert an extra or missing character to group into a meaningful token.

ex: 'cha' can be corrected as 'char' by inserting r
'els' " " " " 'else' " " e.

Transpose:- Based on certain rules, we can transpose two characters. Like 'whiel' can be corrected to 'while' by the transpose method.

Replace:- In some cases, it may require replacing one character by another.

Ex: 'chrr' can be corrected as 'char' by with a
'ipt' can be " " int by replacing with n.

Strategies for implementing a lexical Analyzer.

Lexical Analyzer can be designed in 3 ways.

(1) Use a Scanner generator tool like Lex|Flex to produce a lexical analyzer.
→ The specification can be given using a regular expression,
→ This is easy to implement but least significant.

(2) Design a lexical analyzer in high-level programming language, like 'C' and use the input and buffering techniques provided by the language.
→ This is intermediate in terms of efficiency.

(3) writing a lexical analyzer in assembly language is the most efficient method. It can explicitly manage input buffering
→ This is very complex approach to implement

Designing a lexical analyzer either by hand or automated tools mainly involve two steps.

(1) Describe rules for tokens using regular expressions
(2) Design a recognizer for such rules, that is for tokens.

Designing a recognizer corresponds to converting

→ regular expressions to Finite Automata
→ processing can be speeded up if the regular expression is represented in Deterministic Finite automata.
→ Convert regular expression to NFA with ε
→ Convert NFA with ε to NFA without ε
→ convert NFA to DFA

=

Input Buffering.

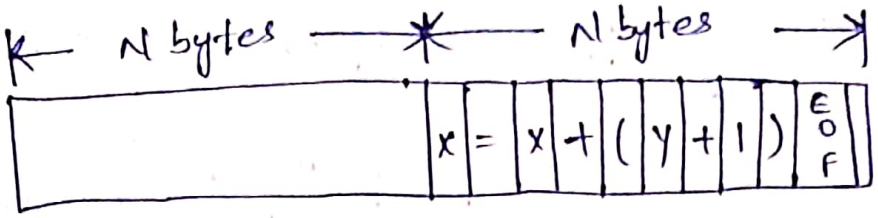
In order to recognize a token, a scanner has to look ahead several characters from the current character many times.

Eg: 'char' is a keyword in 'C' while the term 'char' may be a variable name. when the character 'c' is encountered, the scanner cannot decide whether it is a variable, keyword or function name until it reads three more characters.

* It takes a lot of time to read character by character from a file, so specialized techniques are developed. These buffering techniques, makes the reading process easy and also reduces the ~~ab~~ amount of overhead required to process.

Look ahead with 2N buffering :—

we use a buffer divided into two N-character halves (i.e buffer pair)



Typically N is the Number of characters on one disk block.

* we read N input characters into each half of the buffer with one system read command, rather than invoking a read command for each input character.
* If less than N characters remain in the input buffer then special characters EOF is read into the buffer after the input characters.
* EOF is end of end of input characters.
* Two pointers. "lexeme" and "fwd" to the input buffer are maintained. To find the lexeme, first initialize both the pointers with the first character of the input. keep incrementing the fwd pointer until a lexeme is found.
* once character not matching is found, stop incrementing the pointer and extract the string between the 'lexeme' and 'fwd' pointers.
* once lexeme is found, lexeme and fwd pointers process for next lexemes.

Sentinels:-

The sentinel is a special character that cannot be part of the source program and a natural choice is the character eof.

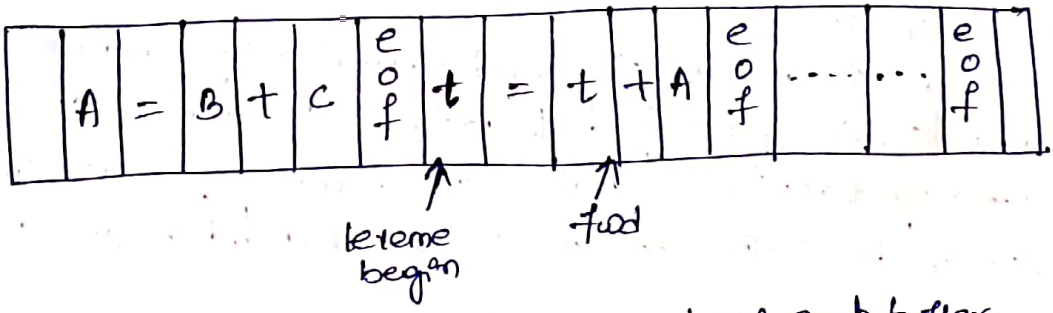


Fig: Sentinels at the end of each buffer.

Specification of Tokens :—

Token type and its attribute uniquely identify a lexeme. Regular expressions are widely used to specify patterns. Following describes notations of Regular Expression.

(i) String and Language :—

⇒ An ALPHABET is a finite non-empty set of symbols denoted by the 'Σ'.

Example: $\Sigma = \{0,1\}$ represents binary alphabet.

⇒ A STRING is a finite sequence of symbols on an alphabet.

ε is an empty string.

Example :— If $\Sigma = \{0,1\}$ then possible strings from Σ is 0, 1, 00, 01, 10, 111, 000, ...

$|\omega|$ denotes the length of the string. ω is a string.

If ω = 'compiler' then $|\omega| = 8$.

⇒ A Language is a set of strings over some alphabet.

If x = compiler and y = Design are two strings then.

x.y = CompilerDesign is known as concatenation

(ii) Operations on Languages :—

In Lexical Analysis, important operations on languages are : Union, Concatenation, and closure.

Union : If L_1 and L_2 are languages then ~~$L_1 \cup L_2 = \{s \mid s \in L_1$ or~~

$L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \}$

Concatenation : $L_1 L_2 = L_1 \cdot L_2 = \{ s_1 s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$.

Kleene closure : $L^* = \cup L^i$

L^* is the language consisting of all words that are concatenations of 0 or more words in the original language. (Including null string ε).

Positive closure : $L^+ = L^* - \{\epsilon\}$

$L^* = L^+ + \{\epsilon\}$

Example:

(i) $\Sigma = \{a, b\}$

$\Sigma^* = \{\epsilon, a, b, aa, ab, bb, ba, aaa, aab, \ldots\}$.

(ii) $\Sigma = \{x\}$

$\Sigma^* = \{\epsilon, x, xx, xxx, \ldots\}$.

(iii) $L_1 = \{\text{smart}, \text{ugly}\}$ $L_2 = \{\text{boy}, \text{girl}\}$

$L_1 \cup L_2 = \{\text{smart}, \text{ugly}, \text{boy}, \text{girl}\}$

$L_1 L_2 = \{\text{smart boy}, \text{smart girl}, \text{ugly boy}, \text{ugly girl}\}$.

(III) Regular Grammar:-

A regular grammar (G) is defined as:

$G = \langle V, T, P, S \rangle$ where

V = set of variables (or non terminals)

T = set of terminals.

P = set of productions

S = start symbol.

Example:

Let $A \rightarrow aB$.

$B \rightarrow b | \epsilon$

Then the Grammar G is defined as.

$V = \{A, B\}$

$T = \{a, b\}$

$P = \{A \rightarrow aB, B \rightarrow b | \epsilon\}$

$S = \{A\}$.

This is called as Regular language and it can be represented by some DFA. A Regular Grammar is a grammar which can represented by using finite automata.

(IV) Regular Expressions (RE):-

Regular Expressions is used to describe tokens of a Programming language. RE is built or consists of smaller RE (using defining rules).

* Each RE denotes a language.

RE is defined as:-

1. ϕ is a RE denoting an empty language.
2. ϵ (Epsilon) is a RE denoting a language which has an empty string.
3. 'a' is a RE denoting a language containing only {a}.
4. If R and S are RE, then L_R and L_S denote language for RE R & S, then:
   (i) R+S is RE for language $L_R \cup L_S$.
   (ii) R.S is RE for language $L_R . L_S$.
   (iii) R^* is RE for language L_R^*.

Eg. If $\Sigma = \{0, 1\}$ then
- $(0|1)(0|1)$ denotes $L = \{0, 1, 001, 110, \cdots\}$
- 1^* denotes $L = \{\epsilon, 1, 11, 1111, \cdots\}$.

Language defined by RE is Regular set. Algebraic Laws for RE are:

1. Commutative law: $R|S = S|R$ [| is commutative]
2. Associative law: $R|(S|U) = (R|S)|U$
3. Concatenation is associative: $R(SU) = (RS)U$
4. Concatenation distributes over |:
   $R(S|U) = RS|RU$; $(S|U)R = SR|UR$.
5. $\epsilon R = R\epsilon = R$
6. $R^* = (R|\epsilon)^*$
7. $R^{**} = R^*$ (* is idempotent).

=

(V) Regular Definition :-

A REGULAR DEFINITION is a sequence of the definitions of the form:

$$d_1 \rightarrow R_1$$
$$d_2 \rightarrow R_2$$
$$\vdots$$
$$d_n \rightarrow R_n.$$

where d_i = distinct name.

R_i = RE over symbols in $\Sigma \cup \{d_1, d_2, \ldots d_{n-1}\}$

Basic symbols ↑ ↑ previously defined names.

Example:-

letter → A|B|---|Z|a|b|---|z.
digit → 0|...|9
id → letter|~~digit~~(letter|digit)*

⇓ can be simplified as

letter → [A-Z|a-z]
digit → [0-9]
id → letter(letter|digit)*

Regular Expressions problems :-

1). Describe the language denoted by following RE :-

$a(a|b)^*a$

Solution:- L = {aa, aba, abbaa, ...}

This RE denotes strings of a's & b's such that language starts with 'a' and ~~ends~~ followed by any number of 'a and 'b' and ends with 'a'.

(2) $(a|b)^*\, a(a|b)(a|b)$

Solution:- L = {aaa, abb, abbaba, ...}

This RE denotes strings of a's & b's such that the third letter from right is 'a' and minimum length of string is three.

3) write regular definitions for : All strings of lower case letters in which the letters are in descending order.

Solution:-

- letter → z|y|x|---|b|a.

  ↓↓

  letter → [z-a].

Recognition of Tokens.

Lexical Analysis can be performed with pattern matching through the use of regular expressions.

* Therefore, a lexical analyzer can be defined and represented as a DFA.
* Recognition of tokens implies implementing a regular expression recognizer. This entails implementation of a DFA.

Example: consider the following grammar.

S → iEtS | iEtSeS | ε

E → T relop T | T

T → id | num.

i=if; t=then; e=else

Here terminals are : i, t, e, relop, id, num.

They generate set of strings given by the following regular definitions

i → if

t → then

e → else.

relop → =|<|>|>=|<=|<>

id → letter|(letter|digit)*

num → digit (. digit+) (E [+|-] digit+).

where, letter → [A-Z a-z]

digit → [0-9]

For this language, the lexical analyzer will recognize the keywords if, then and else, as well as lexemes that match the patterns for relop, id and num.

In addition, we assign the lexical analyzer the job of stripping all white space by recognizing the "token" we defined by ws → (blank | tab | newline).

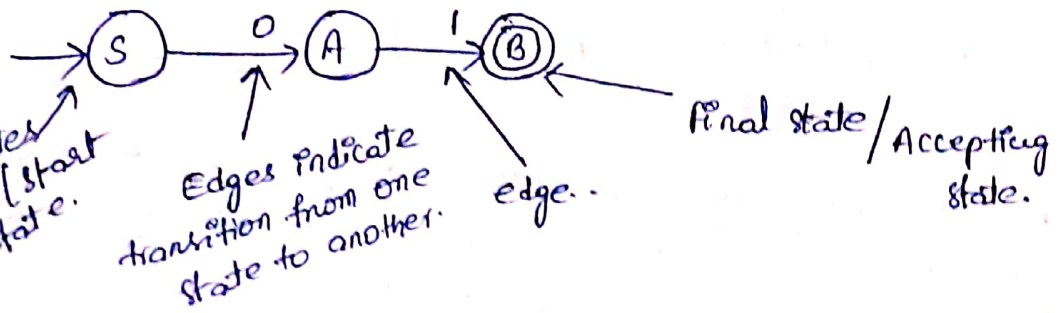
Lexemes	Token name	Attribute Value.
Any ws	-	-
If	If	-
then	then	-
else	else	-
Any id	id	Pointer to table entry
Any num	num	Pointer to table entry
<	relop	LT
>	relop	GT
<=	relop	LE
>=	relop	GE
<>	relop	NE

Table: shows lexemes, token and their attribute values.

TRANSITION DIAGRAMS.

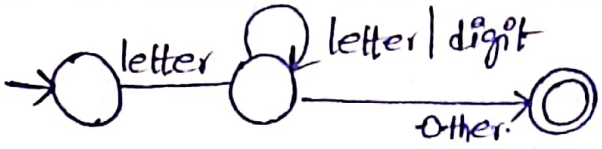
An intermediate step in the construction of a lexical analyzer, we first convert patterns into flowcharts called "transition diagrams".

Transition diagram have a collection of nodes or circles called states. Each state represents a condition that could occur during the process of scanning the inputs.



S, A and B are different states. Start state = {S}; indicated by → at the start. Final state = {B}; indicated by ◎

Transition diagram of an Identifier

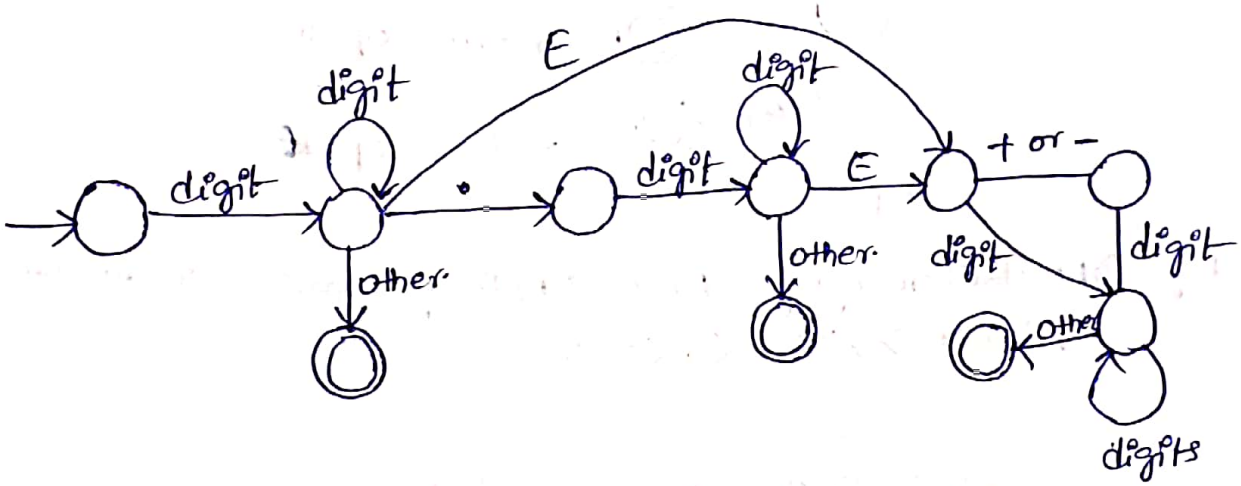


Transition diagram for unsigned numbers

Unsigned number is like set of digits or set of digits followed by decimal point followed by set of digits.

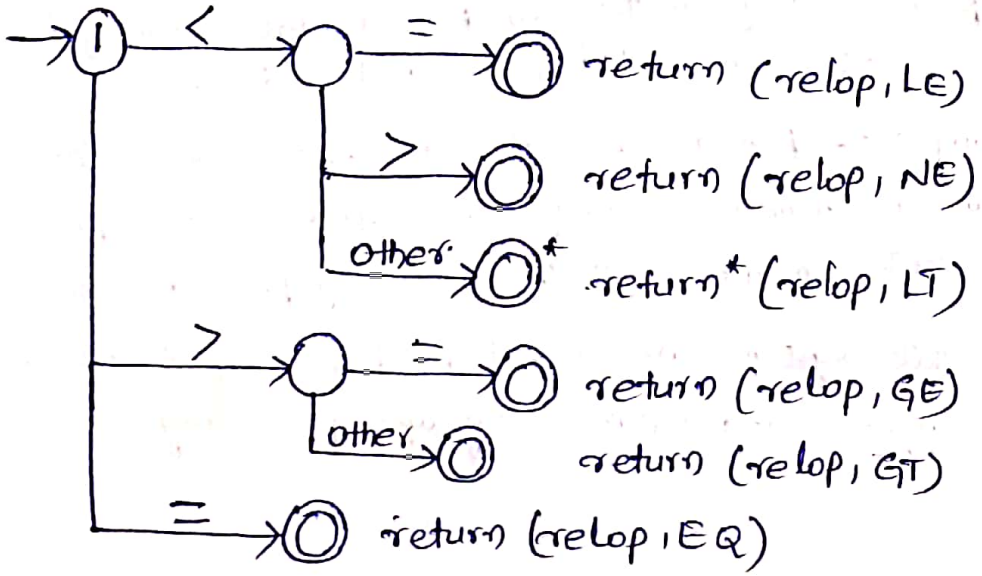
Ex:
341
341.02
341.02E+001
341.02E-20
341E+10

} All these are possible unsigned numbers.



Transition diagram for relational operators

relational operators are (<, <=, >, >=, =, <>)



Implementing finite automata :- we can hand code DFA as each state corresponds to a labeled code fragment and state transitions represented as control transfers as follows.

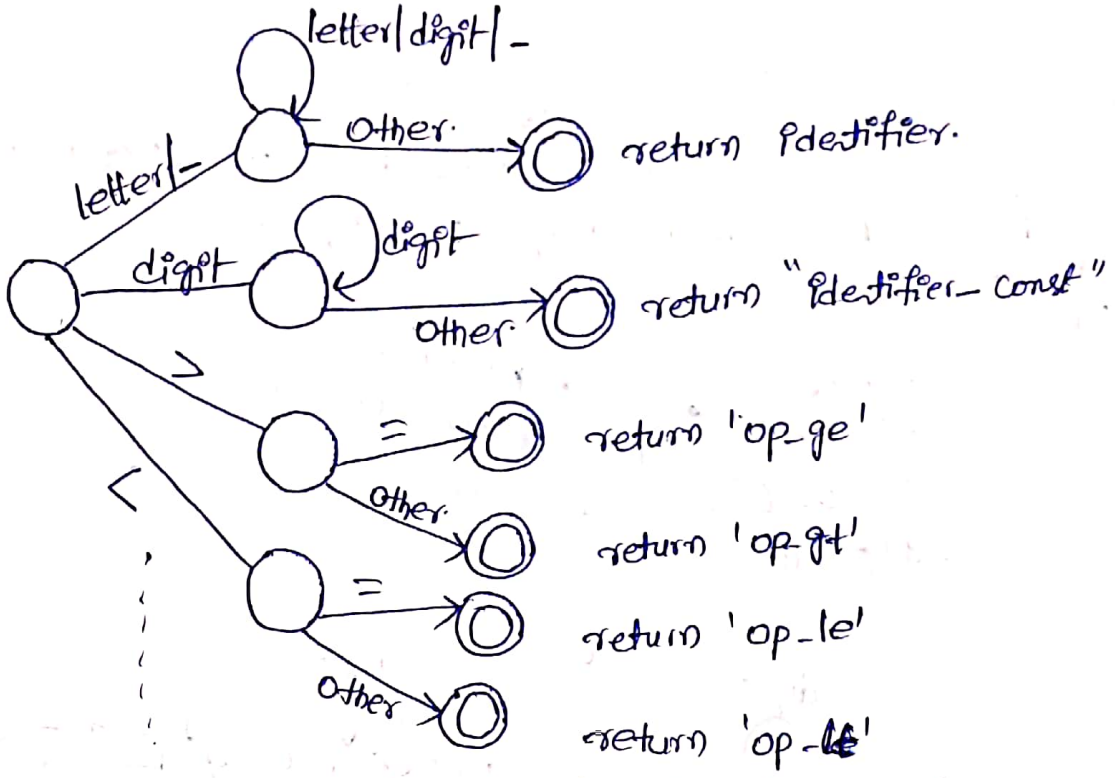


Fig: DFA that recognizes tokens Id, Integer constant, relational operator >, >=, <, <= etc.

Finite State Machine.

A finite state system is a mathematical model of a system with certain input and gives certain output finally.

* The input processed given to an FSM is processed by various states. These states are called intermediate states.
* A good example of FSM is a control mechanism of elevator. This mechanism remembers only current floor number pressed and it does not remember all the previously pressed numbers.
* The finite state systems are useful in design of text editors, lexical analyzers, and natural language processing.

Definition:- A finite automation is formerly defined as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

Q - is a finite set of states which is not empty.

Σ - is input alphabet

q_0 - is initial state

F - is a set of final states and $F \subseteq Q$.

δ - is a transition function (or) mapping function.

Ex:- Lexical Analyzer can be shown as FA. consider the Lexical Analyzer that matches words like "the", "this", "that", and "to".

Systems are called finite Automation. as the number of possible states and the number of letters in the alphabet are both finite. It is automation because the change of state is totally governed by the input.

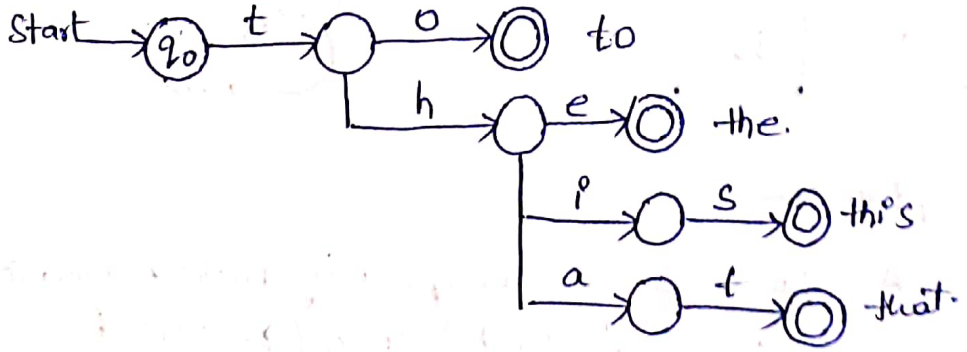


Fig: DFA recognizes strings to, the, this, that.

* Properties of the transition function, 'δ'

1. $\delta(q, \varepsilon) = q$, the states of FA are changed only by an input symbol.

2. for all strings w and i/p symbol a, $\delta(q, aw) = \delta(\delta(q, a), w)$.

An FA can be represented by a.

a) transition diagram

b) transition table.

(a) Transition Diagram

A transition graph contains a set of states as circles. start state q_0 with arrow → (q).
final state by double circle (q)

→ A finite set of transitions (edges/labels) that show to go from some state to other.

Transition Table:—

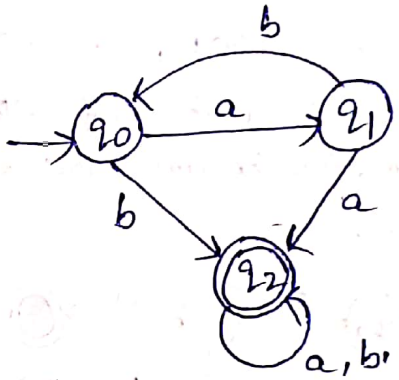
Ex: $M = \{\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_2\}\}$

$\delta(q_0, a) = q_1$ $\delta(q_0, b) = q_2$

$\delta(q_1, a) = q_2$ $\delta(q_1, b) = q_0$

$\delta(q_2, a) = q_2$ $\delta(q_2, b) = q_2$

Δ/Σ	a	b
→q_0	q_1	q_2
q_1	q_2	q_0
*q_2	q_2	q_2

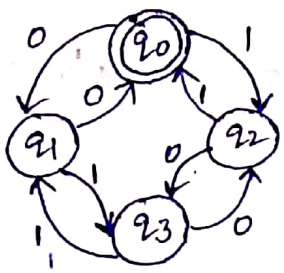


Language Acceptance:—

A string ω is accepted by Finite Automation U given as $U = \{Q, \Sigma, \delta, q_0, F\}$ if $\delta(q_0, \omega) = P$ for some P in F.

* This includes that the string is accepted when it enters into the final state on the last input element.

Example:—



$q_0 \xrightarrow{1} q_2 \xrightarrow{0} q_3 \xrightarrow{1} q_1 \xrightarrow{0} q_0$

Let us check if input string 1010 is accepted or not

$\delta(q_0, 1010) = \delta(q_2, 010) = \delta(q_3, 10) = \delta(q_1, 0) = q_0$

Hence q_0 is the final state. Hence the string is accepted.

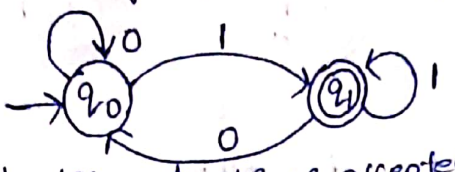
(b) check 11111

$\delta(q_0, 11111) = \delta(q_2, 1111) = \delta(q_0, \epsilon 111) = \delta(q_2, 11) = \delta(q_0, 1) = q_2$

$q_2 \notin F$. Hence, the string rejected.

Example 2:-

Give the language defined by the following FA



we list different strings accepted by this automation

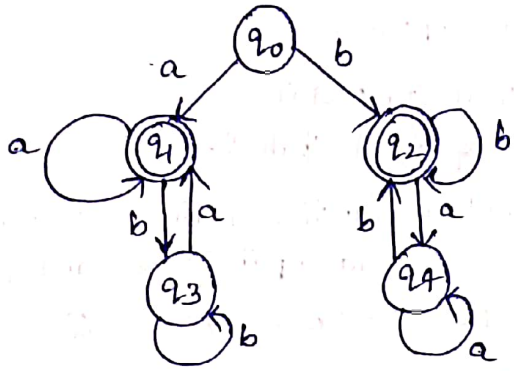
we get {1, 01, 00001, 11011001}.

If we observe, all strings that are accepted always end with 1

$L(M) = \{ w | w$ ends with 1 on $\Sigma = \{0,1\} \}$

Language accepted by machine M is L(M), which is the set of strings

Example: Define language accepted by the following machine.



$L(m) = \{ w | w$ contains all strings that start and end with the same symbol$\}$.

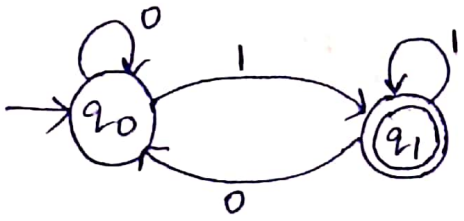
=

Finite Automation is of Two types

a) Deterministic Finite Automation

b) Non deterministic finite Automation.

In DFA there will be unique transition in any state on an input symbol, whereas in NFA there can be more than one transition on an input symbol. Hence, DFA is faster than NFA.



Eg: Fig1: DFA

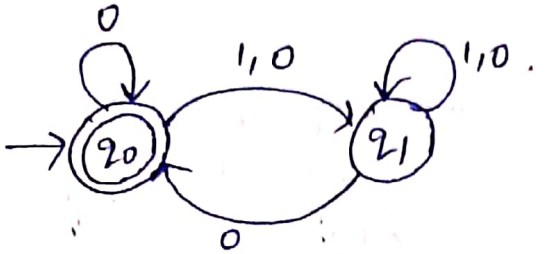


Fig2: NFA

Fig1 denotes example of DFA and Fig2 denotes example of NFA.

(i) Deterministic Finite Automation (DFA)

Deterministic finite automation can be defined as quintuple.

$$M = (Q, \Sigma, \delta, q_0, F)$$

where Q – Non empty finite set of states.

Σ – input alphabet

q_0 – initial start state

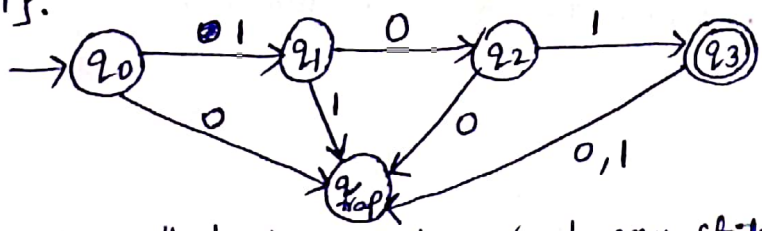
F – set of final states.

δ – transition function that takes two arguments a state and input symbol and returns output as state i.e, $\delta : Q \times \Sigma \rightarrow Q$

Example: $\delta(q_1, a) = q_1$ DFA can be used as finite acceptor because its sole job is to accept certain input strings and reject other strings.

It is called language recognizer because it merely recognizes whether the input strings are in the language or not.

Example 5:- Design a DFA that accepts only input 101 over the set $\{0,1\}$.



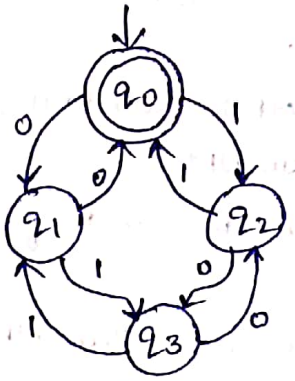
Here q_{trap} is called trap state or dummy state where unnecessary transitions are thrown away.

Example:- Design a DFA that accepts even number of 0's and even number of 1's.

Solution:- This FA will have four different possibilities while reading 0's & 1's as input. The possibilities could be

Even number of 0's and even number of 1's - q_0
Odd number of 0's and even number of 1's - q_1
Even number of 0's and odd number of 1's - q_2
Odd number of 0's and odd number of 1's - q_3.

where states are q_0, q_1, q_2, and q_3. Since the state q_0 indicates the condition of even number of 0's and even number of 1's this state is made as final state. The DFA is given by.



=

Nondeterministic finite Automation (NFA) :-

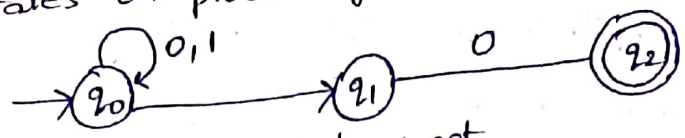
For a given input symbol, there can be more than one transition from a state. Such automation is called Nondeterministic Finite Automation. NFA is mathematically described as quintuple. Nondeterministic finite automation can be defined as quintuple.

$$M = (Q, \Sigma, \delta, q_0, F)$$

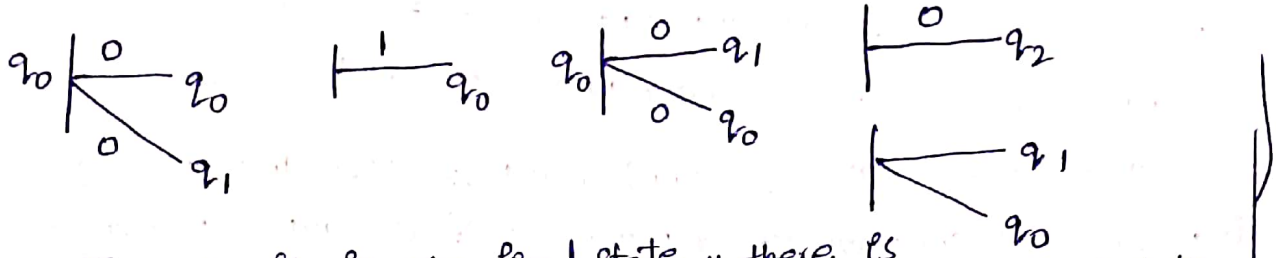
where Q = Non empty finite set of states.
Σ = Input alphabet
q_0 - Initial start state.
F - set of final states.
δ - transition function that takes two arguments a state and input symbol and returns output as state i.e, $\delta: Q \times \Sigma \rightarrow 2^Q$.

Acceptance of NFA

Acceptance of a string is defined as reaching to the final states on processing the input string.



check 0100 is accepted or not



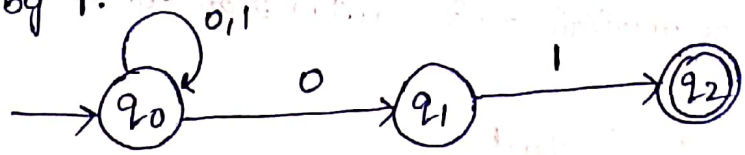
Since q_2 is in the final state ∴ there is atleast one path from the initial state to one of the final state. Hence the given string is accepted.

Note: It is easy to construct NFA than DFA, but the processing time of the string is more than DFA.

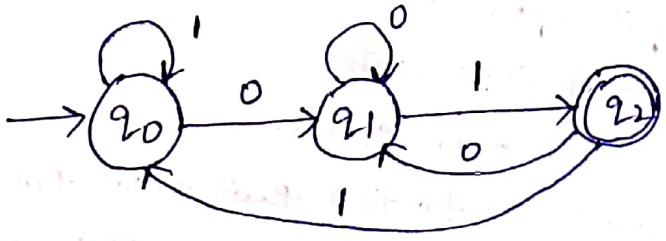
* Every language that can be described by NFA can be described by some DFA
* DFA in practice has some more states than NFA
* Equivalent DFA can have at most 2^n states, whereas NFA has only 'n' states.

Example:- Design NFA accepting all strings ending with 01 over $\Sigma = \{0,1\}$

we can have any string on '0' or '1' but should end with 0 followed by 1.



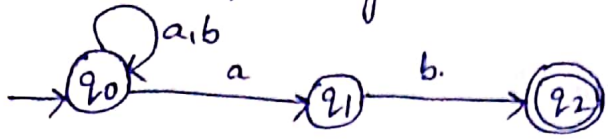
corresponding DFA is



So Drawing NFA is simple than DFA.

Problem on NFA to DFA Conversion:-

1) Convert the following NFA to DFA



Solution:-

$Q = 2^3 = 8$ states = all subsets of q_0, q_1, q_2

$= \{\phi, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$.

$\Sigma = \{0, 1\}$

$q_0 = \{q_0\}$.

$F = \{\{q_2\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$

δ = is given by $\delta_D(\{q_1, q_2\}, a) = \delta_n(q_1, a) \cup \delta_n(q_2, a)$

	0	1
ϕ	ϕ	ϕ
$\rightarrow\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	ϕ	$\{q_2\}$
$\{q_2\}$	ϕ	ϕ
$*\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	ϕ	$\{q_2\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

To get this simplified DFA construct the states of DFA as follows:

(i) Start with the initial state. Do not add all subsets of states as there ~~are~~ may be unnecessary states.

(ii) After finding the transition on this initial state, include only the resultant states into the list until no new state is added to the list.

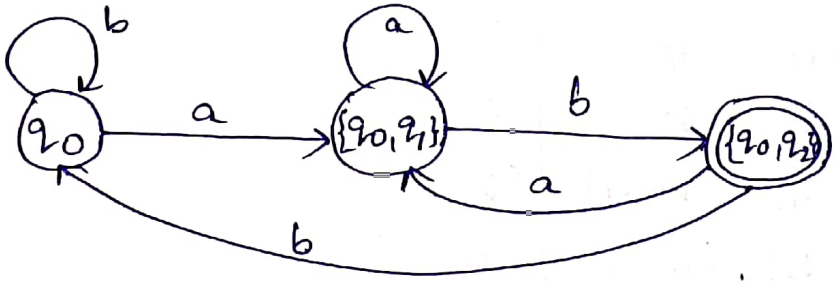
For example, if $\delta(q_0, a) = \{q_0, q_1\}$, then add this as new state in DFA. Then find transition from this state on input symbol.

(iii) Declare the states as final if it has at least one final state of NFA

rewritten table for NFA

	a	b
$\rightarrow\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

The states $\{\emptyset\}$, $\{q_1\}$, $\{q_2\}$, $\{q_1, q_2\}$ and $\{q_0, q_1, q_2\}$ are not reachable from start stated and hence cannot define any strings. So they can be ignored. Hence the DFA can be simplified as



DFA

Example2:

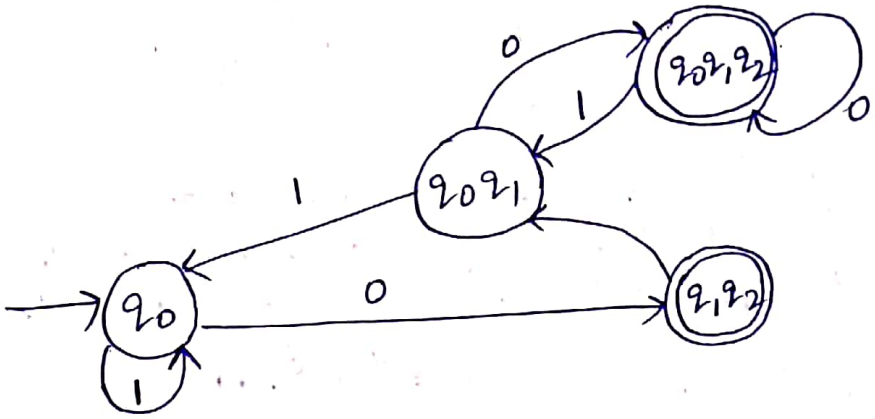
Convert the following NFA to DFA.

δ	0	1
$\rightarrow q_0$	$\{q_1, q_2\}$	$\{q_0\}$
q_1	$\{q_0 q_1\}$	$\emptyset$
$*q_2$	q_1	$\{q_0 q_1\}$

DFA is

DFA Ps

δ	0	1
$\rightarrow\{q_0\}$	$\{q_1 q_2\}$	$\{q_0\}$
$*\{q_1 q_2\}$	$\{q_0 q_1\}$	$\{q_0, q_1\}$
$\{q_0 q_1\}$	$\{q_0 q_1 q_2\}$	$\{q_0\}$
$*\{q_0 q_1 q_2\}$	$\{q_0 q_1 q_2\}$	$\{q_0 q_1\}$.



=

NFA with Epsilon (ϵ) -Transitions:-

* we can extend NFA by introducing a "ϵ-moves" that allows us to make a transition on the empty string. There would be an edge ~~leads~~ labeled ϵ between two states, which allows the transition from one state to another even ~~there~~ without receiving an input symbol.
* This Ps another mechanism that allows NFA to be in multiple states at once.
* Constructing such NFA Ps easier but not that powerful.
* The NFA with ϵ-moves Ps given by

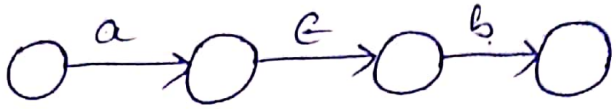
$M = (Q, \Sigma, \delta, q_0, F)$ where δ Ps defined as

$$Q \times \Sigma \cup \{\epsilon\} \rightarrow 2^Q.$$

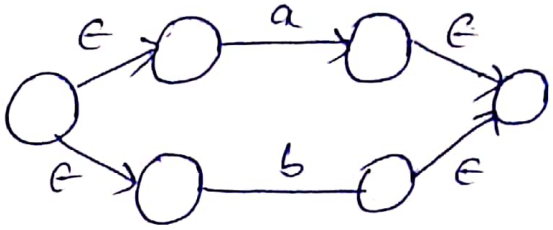
NFA

ϵ- transitions for operations of regular expressions are.

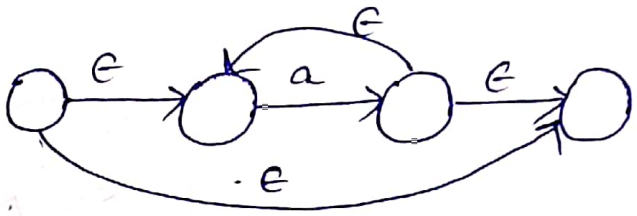
concatenation ab -



Union a+b -



closure a* -



The above construction of ϵ-NFA is defined from Thompson construction method. for any kind of Regular expressions this method can be applicable and can be obtained ϵ-NFA.

example : for regular expression $(a+b)^*a$. ϵ-NFA is,

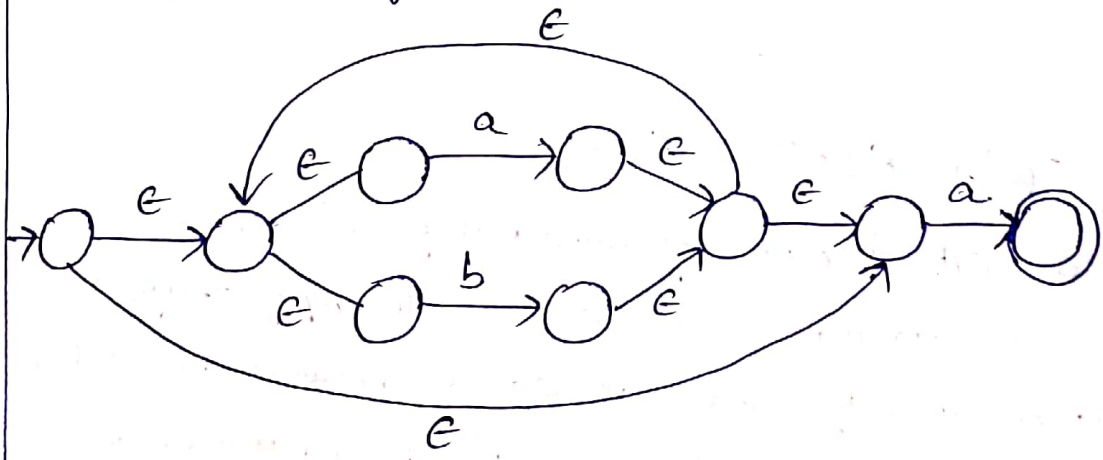


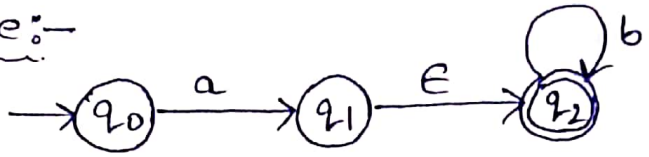
Fig: ϵ-NFA for $(a+b)^*a$.

=

NFA with ϵ-Transitions moves to without ϵ-Transition

For each state, compute ϵ-closure(q) on each input symbol $a \in \Sigma$. If the ϵ-closure of a state contains a final state then make the state as final.

Example:-



Solution: Transition table for given NFA is.

δ	a	b	ϵ
→ q0	q1	ϕ	ϕ
q1	ϕ	ϕ	q2
* q2	ϕ	q2	ϕ

(i) Finding ϵ-closure.

ϵ-closure $(q_0) = \{q_0\}$

ϵ-closure $(q_1) = \{q_1, q_2\}$

ϵ-closure $(q_2) = \{q_2\}$

(ii) Extended function.

$$\hat{\delta}(q, x) = \epsilon\text{-closure}(\delta(\hat{\delta}(q, \epsilon), x))$$
$$\hat{\delta}(q, \epsilon) = \epsilon\text{-closure}(q)$$

$\hat{\delta}(q_0, a)$ = ϵ-closure $(\delta(\hat{\delta}(q_0, \epsilon), a))$
= ϵ-closure $(\delta(\epsilon\text{-closure}(q_0), a))$
= ϵ-closure $(\delta(q_0, a))$
= ϵ-closure (q_1)
= $\{q_1, q_2\}$

$\hat{\delta}(q_0, b)$ = ϵ-closure $(\delta(\hat{\delta}(q_0, \epsilon), b))$
= ϵ-closure $(\delta(\epsilon\text{-closure}(q_0), b))$
= ϵ-closure $(\delta(q_0, b))$
= ϵ-closure (ϕ)
= ϕ.

$\hat{\delta}(q_1, a) = \epsilon\text{-closure}(\delta(\hat{\delta}(q_1, \epsilon), a))$

$= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_1), a))$

$= \epsilon\text{-closure}(\delta(q_1, q_2), a))$

$= \epsilon\text{-closure}(\delta(q_1, a) \cup \delta(q_2, a))$

$= \epsilon\text{-closure}(\phi \cup \phi)$

$= \phi$

$\hat{\delta}(q_1, b) = \epsilon\text{-closure}(\delta(\hat{\delta}(q_1, \epsilon), b))$

$= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_1), b))$

$= \epsilon\text{-closure}(\delta(q_1, q_2), b))$

$= \epsilon\text{-closure}(\delta(q_1, b) \cup \delta(q_2, b))$

$= \epsilon\text{-closure}(\phi \cup q_2)$

$= \epsilon\text{-closure}(q_2)$

$= q_2.$

$\hat{\delta}(q_2, a) = \epsilon\text{-closure}(\delta(\hat{\delta}(q_2, \epsilon), a))$

$= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_2), a))$

$= \epsilon\text{-closure}(\delta(q_2, a))$

$= \epsilon\text{-closure}(\phi)$

$= \phi$

$\hat{\delta}(q_2, b) = \epsilon\text{-closure}(\delta(\hat{\delta}(q_2, \epsilon), b))$

$= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_2), b))$

$= \epsilon\text{-closure}(\delta(q_2, b))$

$= \epsilon\text{-closure}(q_2)$

$= q_2$

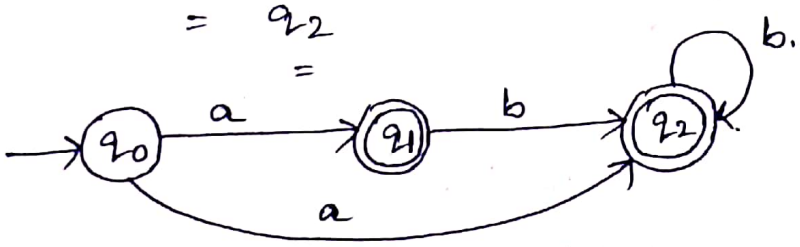


Fig: NFA without ϵ-moves.

Final states are q_1, q_2 because

ϵ-closure (q_1) contains final states

ϵ-closure (q_2) contains final states.