ACADEMIC YEAR: 2025-26
REGULATION: 2023

IFET Autonomous College of Engineering
Permenantly Affiliated to Anna University | Approved by AICTE

DEPARTMENT OF CSE
SUBJECT CODE / NAME: 23CS6401 / COMPILER DESIGN
YEAR / SEM: III / VI

UNIT I INTRODUCTION TO COMPILER DESIGN

Compilers – Analysis of the Source Program – Phases of a Compiler- Types of Compiler-Role of Lexical Analyzer – Input Buffering – Specification of Tokens – Recognition of Tokens -Finite Automata – Regular Expressions to Automata – Minimizing DFA. Language for Specifying Lexical Analyzers - Design of Lexical analyzer generator(LEX) - Recent trends in Compiler Design.

1. COMPILER

Compiler is a translator which is used to convert programs in high-level language to low-level language. It translates the entire program and also reports the errors in source program encountered during the translation.

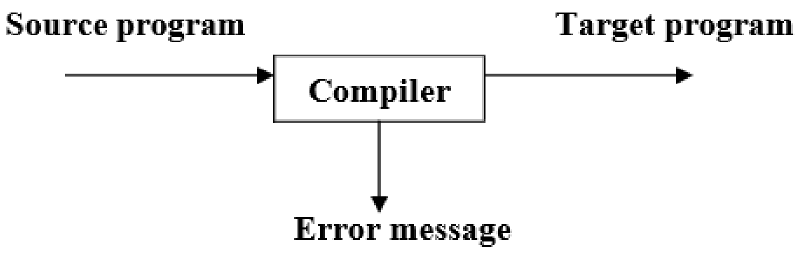


Fig 1.1 Compiler

Properties of a Compiler

- The compiler itself must be **bug free.**
- It must generate **correct machine code**.
- The generated machine code must **run fast**.
- The compiler itself must **run fast** (compilation time must be proportional to program size)
- The compiler must be **portable** (i.e modular, supporting separate compilation)
- It must give **good diagnostics** and error messages.
- The generated code must work well existing **debugger**s.
- It must have **consistent optimization**.

Features of a good compiler

- Compiler should generate accurate code corresponding to the source code provided.
- Compiler should perform better optimization that would contribute to the improvement of the performance as well as in conservation of space.
- Compiler should be bug-free by itself.
- Output of the compiler should not take much time to run.

- Compiler should be able to provide details of the error occurring such that they are sufficient to locate that error in the source program.
- Compiler should generate machine code by using CPU registers efficiently & should not use redundant LOAD or STORE statements for data.

Classifications of compiler

Single pass compiler: Passes through the parts of each compilation unit only once, immediately translating each part into its final machine code

Multi pass compiler: Processes the source code or abstract syntax tree of a program several times

Load-and-go compiler: A load and go compiler generates machine code and then immediately executes it

Debugging or optimizing compiler: Minimize or maximize some attributes of an executable computer program

1.2 ANALYSIS OF THE SOURCE PROGRAM

The source program can be analysed in three phases –

1) Linear analysis: In this type of analysis the source string is read from left to right and grouped into tokens.

For example- Tokens for a language can be identifiers, constants, relational operators, keywords.

2) Hierarchical analysis: In this analysis, characters or tokens are grouped hierarchically into nested collections for checking them syntactically.

3) Semantic analysis: This kind of analysis ensures the correctness of meaning of the program.

1.3 PHASES OF A COMPILER

A compiler operates in various phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. There are two phases of compilation.

- Analysis (Machine Independent/Language Dependent)
- Synthesis (Machine Dependent/Language Independent) Compilation process is partitioned into no-of-sub processes called **'phases'**. **Analysis part**

The source program is read and broken down into constituent pieces. The syntax and the meaning of the source string is determined and then an intermediate code is created from the input source program. It is also termed as **front end** of compiler.

Again the analysis is carried out in four phases:

- Lexical analysis
- Syntax analysis
- Semantic analysis
- Intermediate code generation

Synthesis part

The intermediate form of the source language is taken and converted into an equivalent target program. During this process if certain code has to be optimized for efficient execution then the required code is optimized. It is also termed as **back end** of compiler.

Again the synthesis is carried out in two phases:

- Code optimization
- Code generation

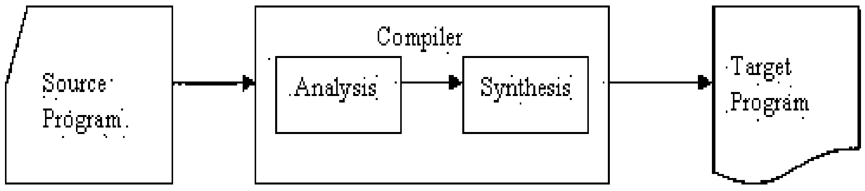


Fig 1.2 Analysis and Synthesis model of compiler

Different phases of compiler can be grouped together to form a front end and back end. The front end and back end model of the compiler is very much advantages because of the following reasons.

- By keeping the same front end and attaching different back ends one can produce a compiler for same source language on different machines.
- By keeping different front ends and same back end one can compile several different languages on the same machine.

The different phases of compiler are as follows,

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generation
5. Code generation
6. Code optimization

All of the mentioned phases involve the following tasks,

- Symbol table management

- Error handling

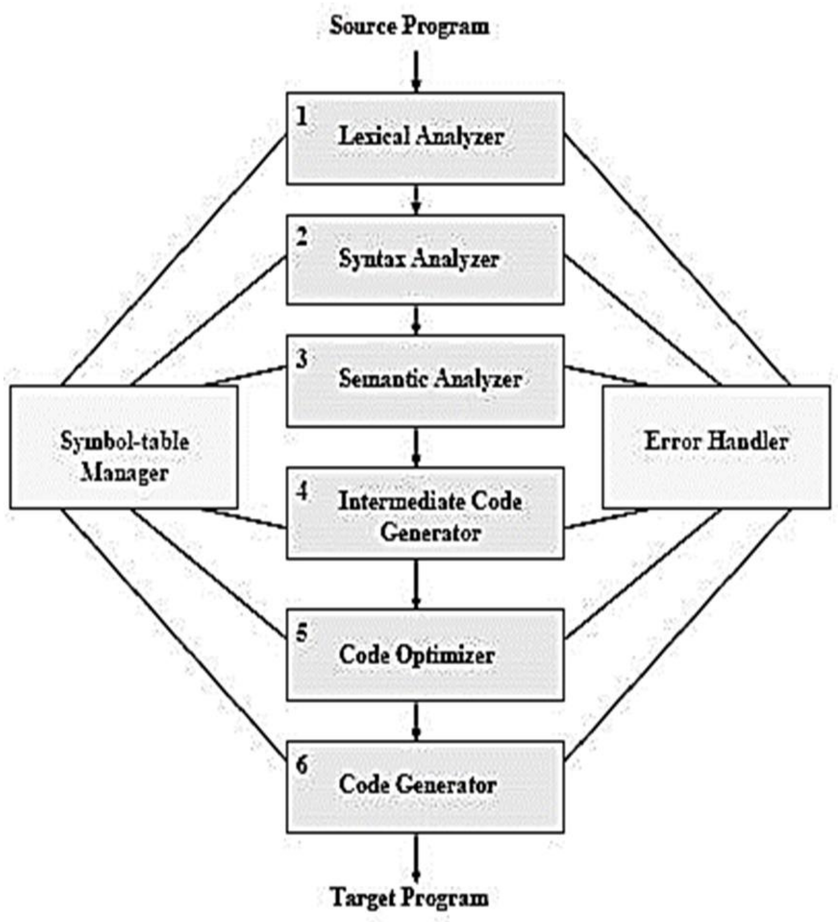


Fig 1.3 Phases of compiler

1. Lexical analysis

- Lexical analysis is the first phase of compiler.
- Lexical analysis is also called as **Scanning/ linear analysis.**
- Source program is scanned to read the stream of characters and those characters are grouped to form a sequence called lexemes which produces token as output.
- ***Token*** is a sequence of characters that can be treated as a single logical entity. The tokens are,

  1) Identifiers 2) Keywords 3) Operators 4) Special symbols 5) Constants
- ***Pattern:*** A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.
- ***Lexeme:*** A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
- Once a token is generated the corresponding entry is made in the symbol table.

Input: stream of characters

Output: Token
Token Template:<token-name, attribute-value>
Table Token Separation
Give Input: position = initial + rate * 60

Table.1.1 Token Separation

Lexemes	**Tokens**
Position	Identifier
=	assignment symbol
Initial	Identifier
+	+(addition symbol)
Rate	Identifier
*	*(multiplication symbol)
60	60(number)

Hence, <id, 1><=><id, 2><+><id, 3><*><60>

Example: Construct the lexical tokens corresponding to each of the following Java source inputs:

Solution:
(a) for (i=1; i<5.1e3; i++) func1(x);
(b) if (sum!=133) /* sum = 133 */
(a) for (i=1; i<5.1e3; i++) func1(x);
keyword for
special char (
identifier i
operator =
numeric const 1
special char ;
identifier i
operator<
numeric const 5.1e3
special char ;
identifier i
operator ++
special char)
identifier func1
special char (
identifier x
special char)

special char ;
(b) if (sum!=133) /* sum = 133 */
keyword if
special char (
identifier sum
operator !=
numeric const 133
special char)
comment /* sum = 133 */

2. Syntax analysis

- The second phase of the compiler is ***syntax analysis or parsing or hierarchical analysis.***
- Parser converts the tokens produced by lexical analyzer into a tree like representation called parse tree.
- The hierarchical tree structure generated in this phase is called syntax tree or parse tree.
- A parse tree describes the syntactic structure of the input.
    - *Input: Tokens*
    - *Output: Syntax tree*
- In a syntax tree, each interior node represents an operation and the children of the node represent the arguments of the operation.

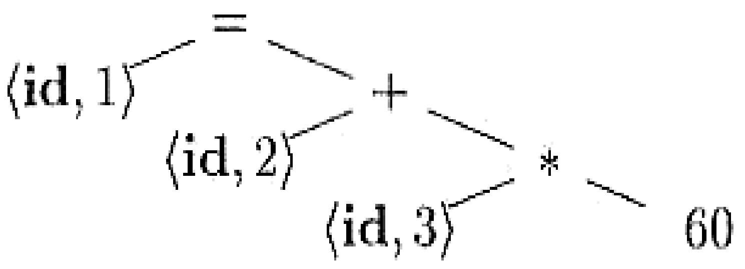


Fig 1.4 Syntax tree for position = initial + rate * 60

The tree has an interior node labeled * with <id, 3> as its left child and the integer 60 as its right child. The node <id, 3> represents the identifier rate. Similarly <id,2> and <id, 1> are represented as in tree. The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifier position

Syntax tree and Parse tree

Table 1.2 Difference between Syntax tree and Parse Tree

S. No	Syntax tree	Parse tree
1.	A syntax tree records the structure of the input and is insensitive to the grammar that produced it.	A parse tree is a record of the rules (and tokens) used to match some input text.

2.	Interior nodes are "operators", leaves are operands.	Interior nodes are non-terminals, leaves are terminals.
3.	When representing a program in a tree structure usually use a syntax tree.	Rarely constructed as a data structure.
4.	Represents the abstract syntax of a program (the semantics)	Represents the concrete syntax of a program.

Example: Show the sequence of atoms put out by the parser, and show the syntax tree corresponding to each of the following Java source inputs:

i) a = (b+c) * d;
(ADD, b, c, T1)
(MUL, T1, d, T2)
(MOV, T2, , a)
ii) if (a<b) a = a + 1;
(TST, a, b, , 2, L1)
(JMP, L2)
(LBL, L1)
(ADD, a, 1, T1)
(MOV, T1, , a)
(LBL, L2)

Solution:

*i) a = (b+c) * d;*
(ADD, b, c, T1)
(MUL, T1, d, T2)
(MOV, T2, , a)

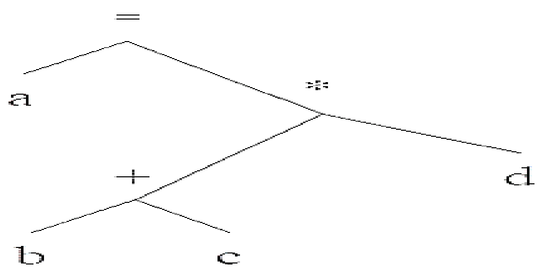


Fig 1.5 Syntax tree for a = (b+c) * d

3. Semantic Analysis

- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
- It ensures the correctness of the program, matching of the parenthesis is also done in this phase.
- It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.
- An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands.
- The compiler must report an error if a floating-point number is used to index an array. The language specification may permit some type conversions like integer to float for float addition is called coercions.

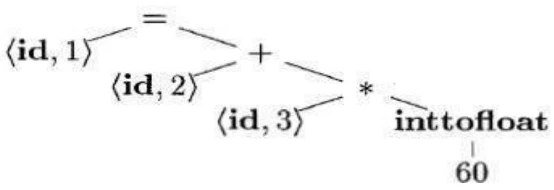


Fig 1.6 Semantic tree for position = initial + rate * 60

- The operator * is applied to a floating-point number rate and an integer 60. The integer may be converted into a floating-point number by the operator **int to float.**

Table 1.3 Syntax vs Semantic analysis

Syntax Analysis	Semantic Analysis
A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams.	Semantics of a language provide meaning to its constructs, like tokens and syntax structure.
The parser analyzes the source code (token stream) against the production rules to detect any errors in the code.	Semantics help interpret symbols, their types, and their relations with each other.

4. Intermediate code generation

- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation.
- Three-address code is one of the intermediate representations, which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register.
- The output of the intermediate code generator consists of the three-address code sequence for position = initial + rate * 60

```
t1 = int to float(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Properties of good intermediate representations

- It should be easy to produce,
- And easy to translate into target program.
- It should not include too much machine specific detail.
- Convenient to produce in the semantic analysis phase.
- Convenient to translate into code for the desired target architecture.
- Each construct must have a clear and simple meaning such that optimizing transformations that rewrite the intermediate representation can be easily specified and implemented.
- It provides a separation between front and back ends which helps compiler portability.

5. Code optimization

- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.
- Optimization has to improve the efficiency of code so that the target program running time and consumption of memory can be reduced.
  t1 = id3 * 60.0
  id1 = id2 + t1
- The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the int to float operation can be eliminated by replacing the integer 60 by the floating-point number 60.0.

Example: Show how each of the following C source inputs can be optimized using global optimization techniques:

a) if (x>0) {x = 2; y = 3;}
else {y = 4; x = 2;}
b) if (x>0) x = 2;
else if (x<=0) x = 3;
else x = 4; **Solution:**
a) if (x>0) {x = 2; y = 3;}
else {y = 4; x = 2;}

The optimized code is,
```
if (x>0)
y = 3;
else
y = 4;
x = 2;
```
b) if (x>0) x = 2;
else if (x<=0) x = 3;
else x = 4;

The optimized code is,
```
if (x>0) x = 2;
else x = 3;
```

6. Code Generation

- Code generation is the final phase of compiler
- The code generator takes as input from code optimization and produces target code or object code as output.
- If the target language is machine code, then the registers or memory locations are selected for each of the variables used by the program.
- The intermediate instructions are translated into sequences of machine instructions.
- For example, using registers R1 and R2, the intermediate code might get translated into the machine code

```
LDF R2, id3
MULF R2, R2 , #60.0
LDF Rl, id2
ADDF Rl, Rl, R2
STF id1, Rl
```

Examples: Which of the following Java source errors would be detected at compile time, and which would be detected at run time?

(a) a = b+c = 3;
(b) if (x<3) a = 2;
(c) if (a>0) x = 20;
else if (a<0) x = 10;
else x = x/a;
(d) MyClass x [] = new MyClass[100];
x[100] = new MyClass;

Solution:

(a) a = b+c = 3;	Compile time error
(b) if (x<3) a = 2	
else a = x;	Compile time error
(c) if (a>0) x = 20;	
else if (a<0) x = 10;	
else x = x/a;	Run time error
(d) MyClass x [] = new MyClass[100];	
x[100] = new MyClass;	Run time error

Symbol-Table Management

- The symbol table, which stores information about the entire source program, is used by all phases of the compiler.
- An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.
- These attributes may provide information about the storage allocated for a name, its type, its scope.
- In the case of procedure names, such things as the number and types of its argument, the

method of passing each argument (for example, by value or by reference), and the type returned are maintained in symbol table.

- The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.
- A symbol table can be implemented in one of the following ways:
    - Linear (sorted or unsorted) list
    - Binary Search Tree
    - Hash table
- Among the above all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.
- A symbol table may serve the following purposes depending upon the language in hand:
    - To store the names of all entities in a structured form at one place.
    - To verify if a variable has been declared.
    - To implement type checking, by verifying assignments and expressions.
    - To determine the scope of a name (scope resolution).

Error Handler

- As programs are written by human beings therefore they cannot be free from errors. In compilation, each phase detects errors.
- These errors must be reported to error handler whose task is to handle the errors so that the compilation can proceed. Normally, the errors are reported in the form of message. It worksin conjunction with all the phases to handle the errors at the respective phases.

1.4 TYPES OF COMPILER

1.4.1 Incremental Compiler

Incremental compiler is a compiler which performs the recompilation of only modified source rather than compiling the whole source program.
The basic features of incremental compiler are,
1. It tracks the dependencies between output and the source program.
2. It produces the same result as full recompile.
3. It performs less task than the recompilation.
4. The process of incremental compilation is effective for maintenance.

1.4.2 Cross Compiler

Basically there exists three types of languages
1. Source language i.e. the application program.
2. Target language in which machine code is written.
3. The Implementation language in which a compiler is written.

There may be a case that all these three languages are different. In other words there may be a compiler which run on one machine and produces the target code for another machine. Such a compiler is called cross compiler. Thus by using cross compilation technique platform S independency can be achieved.

To represent cross compiler T diagram is shown in Fig. 1.7(a) and Fig. 1.7(b)

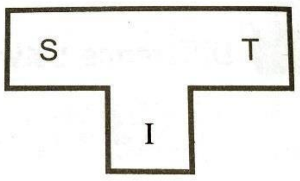


Fig. 1.7 (a) T diagram with as source, T as target and I as implementation language

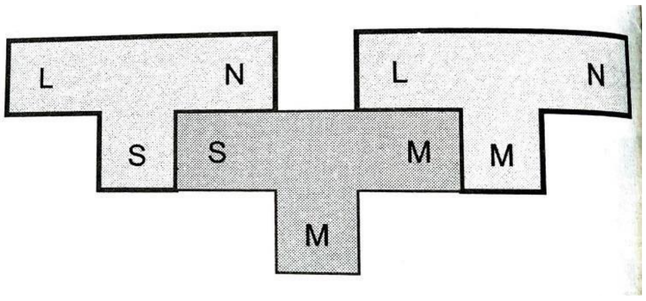


Fig. 1.7 (b) Cross compiler

For source language L the target language N gets generated which runs on machine M.

For example:

For the first version of EQN compiler, the compiler is written in C and the command are generated for TROFF, which is as shown in Fig. 1.8.

The cross compiler for EQN can be obtained by running it on PDP-11 through compiler, which produces output for PDP-11 as shown below –

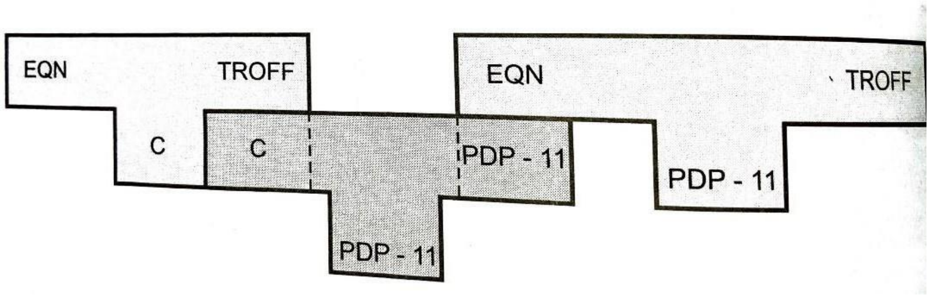


Fig. 1.8 Cross compiler for EQN

1.5 ROLE OF LEXICAL ANALYZER

Lexical Analyzer

The main task of lexical analyser is to read the source program, scan the input characters, group them into lexemes and produce the token as output. The stream of tokens is sent to the parser for syntax analysis. It interacts with the symbol table. When it discovers a lexeme consisting as identifier, it needs to enter that lexeme into symbol table. In some cases, information regarding kind of identifier may be read from the symbol table by lexical analyser to assist it in determining the proper token it must pass to the parser. The interaction between lexical analyser and the parser is implemented by the getNextToken command (Fig 1.9). When the command is issued by parser, the lexical analyser to read characters from its input until it identify the next lexeme and produce for it the next token, which it returns to the parser.

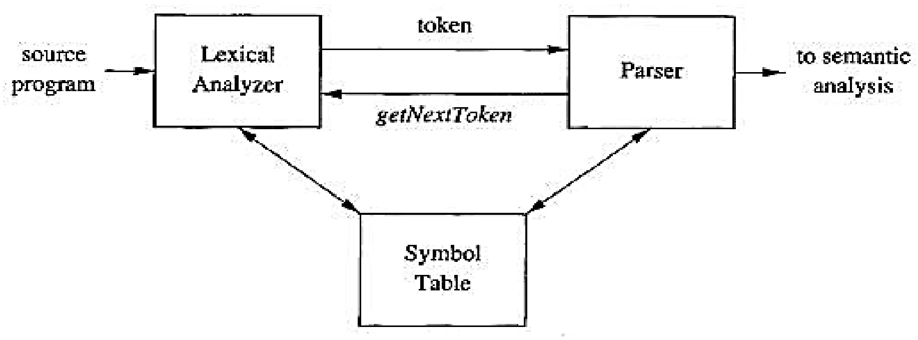


Fig 1.9 Interaction between the lexical analyser and parser

Functions of Lexical analyser

The following are the some other tasks performed by the lexical analyser

- It eliminates whitespace and comments.
- It generates symbol table which stores the information about identifiers, constants encountered in the input.
- It keeps track of line numbers.
- It reports the error encountered while generating the tokens.
- It expands the macros when the source program uses a macro-pre-processor

Processes of lexical analyser

Sometimes, lexical analyser is divided into a cascade of two processes:

- **Scanning**: It performs reading of input characters, removal of white spaces and comments
- **Tokenization:** It is the more complex portion, where the scanner produces the sequence of tokens as output

Lexical Analysis versus Parsing

There are a number of reasons why analysis portion of a complier is normally separated into lexical analysis and parsing (syntax analysis) phases

1. **Simplicity of design** is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks. The removal of white spaces and comments enables the syntax analyzer for efficient syntactic constructs.
2. **Compiler efficiency is improved**. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly
3. **Compiler portability is enhanced**. Input-device-specific peculiarities can be restricted to the lexical analyser

Tokens, Patterns and Lexemes

When discussing lexical analysis, we use three related but distinct terms:

- **Token:** Token is a valid sequence of characters which are given by lexeme. In a programming language, Keywords, constant, identifiers, numbers, operators and punctuations symbols are possible tokens to be identified.

- **Pattern:** Pattern describes a rule that must be matched by sequence of characters (lexemes) to form a token. It can be defined by regular expressions or grammar rules
- **Lexeme:** Lexeme is a sequence of characters that matches the pattern for a token i.e., instance of a token.

Table 1.4 Examples of token

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Attributes of tokens:

- The lexical analyzer collects information about tokens into their associated attributes. The tokens influence parsing decisions and attributes influence the translation of tokens.
- Usually a token has a single attribute i.e. pointer to the symbol table entry in which the information about the token is kept

Lexical Errors

- A character sequence that cannot be scanned into any valid token is a lexical error.
- Lexical errors are uncommon, but they still must be handled by a scanner.
- Misspelling of identifiers, keyword, or operators are considered as lexical errors.

Usually, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

Lexical error handling approaches

Lexical errors can be handled by the following actions:

- Deleting one character from the remaining input.
- Inserting a missing character into the remaining input.
- Replacing a character by another character.
- Transposing two adjacent characters.

1.6 INPUT BUFFERING

➢ Some efficiency issues concerned with the buffering of input.
  - Speed of lexical analysis is a concern.
  - Lexical analysis needs to look ahead several characters before a match can be announced

- To ensure that a right lexeme is found, one or more characters have to be looked up beyond the next lexeme.
    - A two-buffer input scheme that is useful when lookahead on the input is necessary to identify tokens.
    - Techniques for speeding up the lexical analyser, such as the use of sentinels to mark the buffer end.
- There are three general approaches for the implementation of a lexical analyzer:
    - By using a lexical-analyzer generator, such as lex compiler to produce the lexical analyzer from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.
    - By writing the lexical analyzer in a conventional systems-programming language, using I/O facilities of that language to read the input.
    - By writing the lexical analyzer in assembly language and explicitly managing the reading of input.

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. The following fig 1.10 shows the buffer pairs which are used to hold the input data.

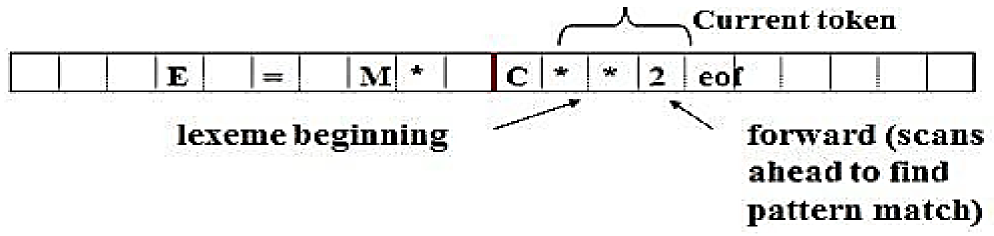


Fig 1.10 Buffer Pairs

Scheme

- Consists of two buffers, each consists of N-character size which are reloaded alternatively.
- N-Number of characters on one disk block, e.g., 4096.
- N characters are read from the input file to the buffer using one system read command.
- eof is inserted at the end if the number of characters is less than N.

Pointers

- Two pointers lexemeBegin and forward are maintained.
- The pointer lexeme Begin points to the beginning of the current lexeme which is yet to be found.
- The pointer forward scans ahead until a match for a pattern is found.
- Once a lexeme is found, lexemebegin is set to the character immediately after the lexeme which is just found and forward is set to the character at its right end.
- Current lexeme is the set of characters between two pointers.

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 first half ;
move forward to beginning of first half
end
else forward : = forward + 1 ;
```

Sentinels:

- With the previous algorithm, we need to check each time we move the forward pointer that
- We have not moved off one half of the buffer. If so, then we must reload the other half.
- This can be reduced, if we extend each buffer half to hold a sentinel character at the end.
- Code performs only one test to see whether forward points to an eof.

1.7 SPECIFICATION OF TOKENS

Regular expressions are a notation to represent lexeme patterns for a token. Regular expressions are used to represent the language for lexical analyser. It assists in finding the type of token that accounts for a particular lexeme

Strings and Languages

An alphabet or a character class is a finite set of symbols. Typical examples of symbols are letters, digits and punctuation. The set {0, 1} is the binary alphabet. ASCII and EBCDIC are two examplesof an alphabets.

A string over some alphabet is a finite sequence of symbol taken from that alphabet. In language theory, the terms "sentence" and "word" are often used as synonyms for "string”. The length of a string s, usually written |s|, is the number of occurrences of symbols in s. For example, banana is a string of length six. The empty string, denoted ε, is the string of length zero.

A language is a set of strings over some fixed alphabet. The language may contain a finite or an infinite number of strings

The following string-related terms are commonly used:

- A prefix of string s is any string obtained by removing zero or more symbols from the endof string s. For example, ban is a prefix of banana.
- A suffix of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, nana is a suffix of banana.
- A substring of s is obtained by deleting any prefix and any suffix from s. For example,nan is a substring of banana.
- The proper prefixes, suffixes, and substrings of a string s are those prefixes, suffixes, and substrings, respectively of s that are not ε or not equal to s itself.
- A subsequence of s is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, baan is a subsequence of banana.

Operations on Languages

The following are the operations that can be applied to languages:

Union

Union of two languages Land M produces the set of strings which may be either in language L or in language M or in both.

Example:

Let L and M be two languages where L = {dog, ba, na} and M = {house, ba} then

Union of L & U=LUM = {dog, ba, na, house}

Concatenation

Concatenation of two languages L and M, produces a set of strings which are formed by merging the strings in L with strings in M (strings in L must be followed by strings in M)

Example:

Concatenation of L & M = LM = {doghouse, dogba, bahouse, baba, nahouse, naba}

Kleene closure

Kleene closure refers to zero or more occurrences of input symbols in a string, i.e., it includes empty string Ɛ (set of strings with 0 or more occurrences of input symbols). The kleene closure of language L, denoted by L*. For example, If L = {a, b}, then

L* = { , a, b, aa, ab, ab, ba, bb, aaa, aba, baa, . . . }

Positive closure

Positive closure indicates one or more occurrences of input symbols in a string, i.e., it excludes empty string Ɛ (set of strings with 1 or more occurrences of input symbols). The positive closure of Language L denoted by L+. For example, If L = {a, b}, then

L+ = {a, b, aa, ba, bb, aaa, aba, . . . }

Precedence of operators

- Unary operator (*) is having highest precedence.
- Concatenation operator (.) is second highest and is left associative.
- Union operator (| or U) has least precedence and is left associative.

Based on the precedence, the regular expression is transformed to finite automata when implementing lexical analyzer

Regular Expressions

Regular expressions are a combination of input symbols and language operators such as union, concatenation and closure. It can be used to describe the identifier for a language. The identifier is a collection of letters, digits and underscore which must begin with a letter. Hence, the regular expression for an identifier can be given by,

Letter_ (letter | digit)*

Note: Vertical bar (|) refers to 'or' (Union operator).

The regular expressions are built recursively out of smaller regular expressions. Each regular expression **r** denotes a language **L(r).** There are the two rules that define the regular expressions over some alphabet **Σ** and the languages that those expressions denote:

Basis: There are two rules that form the basis:

1. **ε** is a regular expression, and L(ε) is { ε }, that is, the language whose sole member is the empty string.
2. If **a** is a symbol in Σ, then **a** is a regular expression, and L(**a**) = {a}, that is, the language with one string, of length one, with a in its one position.

Induction: There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then,

1. (r)|(s) is a regular expression denoting the language L(r) U L(s).
2. (r)(s) is a regular expression denoting the language L(r)L(s).
3. (r)* is a regular expression denoting (L(r))*.
4. (r) is a regular expression denoting L(r).

As defined, regular expressions often contain unnecessary pairs of parentheses. We may drop certain pairs of parentheses if we adopt the conventions that

a) The unary operator * has highest precedence and is left associative.
b) Concatenation has second highest precedence and is left associative.
c) | has lowest precedence and is left associative

A Language that can defined by regular expression is called a regular set. If two regular expressions r and s denotes the same regular set, we say they are equivalent. A language which cannot be described by any regular expression is a non-regular set. Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

Algebraic laws of regular expressions

Law	Description
r|s = s|r	| is commutative
r|(s|t)=(r|s)|t	| is associative
r(st)=(rs)t	Concatenation is associative
r(s|t)=rs|rt;(s|t)r=sr|tr	Concatenation is distributive
$\varepsilon r = r\varepsilon = r$	ε is identity for concatenation
r*=(r|ε)*	ε is guaranteed in closure
r**=r*	* is idempotent

Table 1.5 Algebraic laws for regular expressions.

Regular Definitions

Giving names to regular expressions is referred to as a Regular definition. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

dl → r1

d2 → r2

………

dn → rm

Where, Each di is a distinct name and ri is a regular expression over the alphabet Σ U {dl, d2,. . .

, di-1}.

Example:

Identifiers are the set of strings of letters, underscore and digits beginning with a letter.

Regular definition for this set:

letter → A | B | …. | Z | a | b | …. | z |

digit → 0 | 1 | …. | 9

id → letter_(letter_ | digit) *

The pattern for the Pascal unsigned token can be specified as follows:

digit → 0 | 1 | 2 | . . . | 9

digit → digit digit*

Optimal-fraction → .digits | ε

Optimal-exponent → (E (+ | - |ε) digits) | ε

num → digits optimal-fraction optimal-exponent.

This regular definition says that

- An optimal-fraction is either a decimal point (dot) followed by one or more digits or it is missing (i.e., an empty string).
- An optimal-exponent, if not missing, is the letter E followed by an optimal + or - sign, followed by one or more digits.

Extensions of Regular Expressions

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational short hands for them.

One or more instances (+):

The unary postfix operator + means "one or more instances of". If r is a regular expression that denotes the language L(r), then (r)+ is a regular expression that denotes the language (L (r))+. Thus the regular expression a+ denotes the set of all strings of one or more a‟s. The operator + has the same precedence and associativity as the operator *.

Zero or one instance (?):

The unary postfix operator (?) means "zero or one instance of". The notation r? is a shorthand for r | ε. If „r‟ is a regular expression, then (r)? is a regular expression that denotes the language

Character Classes:

The notation [abc] where a, b and c are alphabet symbols denotes the regular expression a | b | c. Character class such as [a – z] denotes the regular expression a | b | c | d | ….|z. We can describe identifiers as being strings generated by the regular expression, [A–Za–z][A– Za–z0–9]*

Examples:

1. Describe the languages denoted by the following regular expressions:

a. a(a|b)*a

b. ((ε|a)b*)*

c. (a|b)*a(a|b)(a|b)

d. a*ba*ba*ba*

e. !! (aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*

Answer

a. String of a's and b's that start and end with a.

b. String of a's and b's.

c. String of a's and b's that the character third from the last is a.

d. String of a's and b's that only contains three b.

e. String of a's and b's that has a even number of a and b.

2. write a regular expression for a language containing the strings of length two over Σ={0,1}

Soln:

RE=(0+1)(0+1)

3. Write a regular expression for a language containing the strings which ends with "abb"over Σ={a,b}

Soln:

RE=(a+b)*abb

4. Design the regular expression for a language accepting all combinations of „a‟ s except the null string over Σ= {a}

Soln: L={a,aa,aaa,….}

RE=a+

Where + is the positive closure

5. Design the regular expression for a language containing all the strings with any number of a‟s and b‟s

Soln:

L={

RE=(a+b)*

6. Construct a regular expression for the language containing all the strings having any number of a‟s and b‟s except null string

Soln:

RE=(a+b)+

1.8 RECOGNITION OF TOKENS

Recognition of token explains how to take the patterns for all needed tokens. It builds a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.

Rules for conditional statement or branching statement can be given as follows

```
Stmt→ if expr then stmt
      | If expr then stmtelse stmt
      | ϵ
Expr→term relop term
      | term
      Term→id
      |number
```

For relop, we use the comparison operations of languages like Pascal or SQL where = is "equals"

and <> is "not equals" because it presents an interesting structure of lexemes.
The terminal of grammar, which are if, then, else, relop, id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are described using regular definitions.
The patterns for token

digit -->[0,9]

digits -->digit+

number -->digit(.digit)?(e.[+-]?digits)?

letter -->[A-Z,a-z]

id -->letter(letter/digit)*

if --> if

then -->then

else -->else

relop -->< | > | <= | >= |= |= | <>

For easy recognition, keywords are considered as reserved words even through their lexeme match with the pattern for identifiers. In addition, we assign the lexical analyzer the job of stripping out white space, by recognizing the "token" ws defined by:

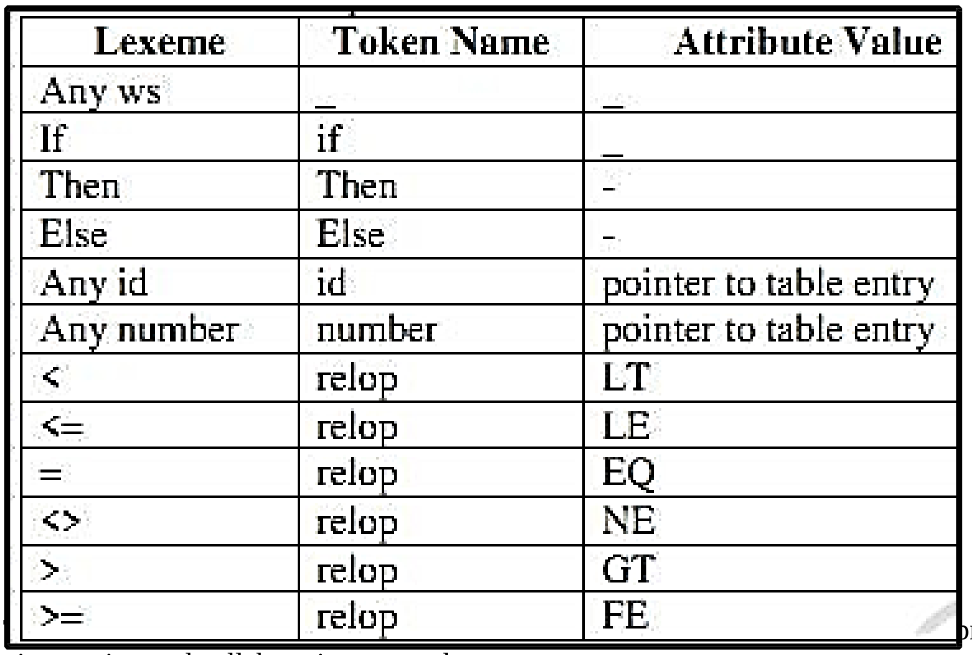
ws$\rightarrow$ (blank |tab| newline)+

or

delim$\rightarrow$ blank| tab | newline

ws$\rightarrow$delim+

Here, blank, tab and newline are abstract symbols that we use to express the ASCII characters of the same names. Token **ws** is different from the other tokens in that ,when we recognize it, we do not return it to parser ,but rather restart the lexical analysis from the character that follows the white space.



Lexeme	Token Name	Attribute Value
Any ws	_	_
If	if	_
Then	Then	-
Else	Else	-
Any id	id	pointer to table entry
Any number	number	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	FE

Table 1.6 Tokens, their patterns and attributes values

Transition Diagram

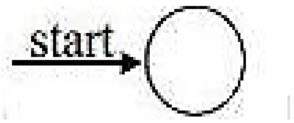
As an intermediate step in the construction of a lexical analyzer, we first produce flowchart, called a Transition diagram. It depicts the actions that take place when a lexical analyzer is called by the parser to get the next token.

Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.

Edges are directed from one state of the transition diagram to another. Each edge is labelled by a symbol or set of symbols. If we are in one state **s**, and the next input symbol is **a**, we look for an edge out of state **s** labeled by **a**. if we find such an edge, we advance the forward pointer and enter the state of the transition diagram to which that edge leads. It uses to keep track of information about characters that are seen as the forward pointer scans the input. It dose that by moving from position to position in the diagram as characters are read

Components of Transition Diagram (Some important conventions about transition diagrams)

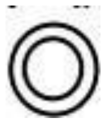
1. One state is labeled the Start State; it is the initial state of the transition diagram where control resides when we begin to recognize a token.



2. Positions in a transition diagram are drawn as circles and are called states.

3. The states are connected by arrows called edges. Labels on edges are indicating the input characters.

4. The Accepting states in which the tokens has been found.



5. Retract one character use * to indicate states on which this input retraction.

Examples:

1. **Transition diagram for relop**

It is fairly clear how to write code corresponding to relop diagram. We look at the first character, if it is <, then look at the next character.

If that character is =, then return (relop, LE) to the parser.

If instead that character is >, then return (relop, NE). If it is another character, return (relop, LT) and adjust the input buffer so that we will read this character again since we have not used it for the current lexeme.

If the first character was =, then return (relop, EQ).

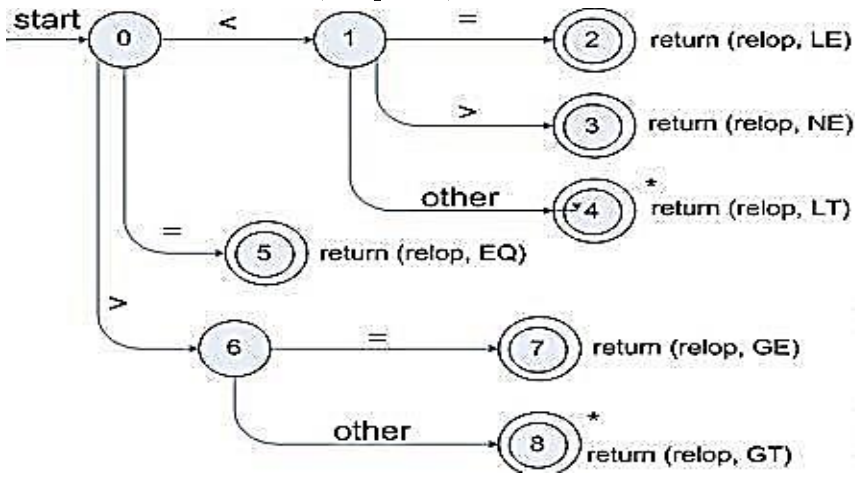


Fig 1.11 Transition diagram for relop

2. Transition diagram for reserved words and identifiers

Recognizing keywords and identifiers presents a problem. Usually, keywords like if or then are reserved, so they are not identifiers even though they look identifiers. Then although we typically use a transition diagram like that in following figure to search for identifier lexemes this diagram will also recognize the keyword if then else of an running example: return (get Token (), installID())

There are two ways that we can handle reserved words that look like identifiers.

1. Install the reserved word in the symbol table initially

2. Create separate transition diagram for each keyword adjust the input buffer so that we will read this character again since we have not used it for the current lexeme. If the first character was =, then return (relop, EQ).

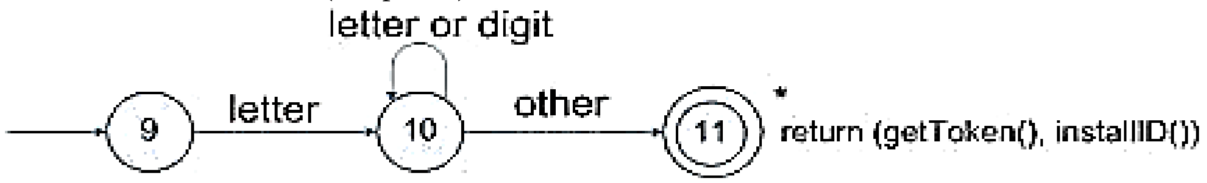


Fig 1.12 A Transition diagram for id's and keywords

3. Transition diagram for unsigned numbers

Beginning in state 12, if we see digit, we go to state13. In that state, we can read any number of additional digits. However, if we see anything but a digit or a dot, we have seen a number in form of an integer. That case is handled by entering state 20, where we return token number. If we instead see a dot in state 13, then we have an "optional fraction." State 14 is entered, and we look for one or more additional digits; state 15 is used for that purpose. If we see an E, then we have an "optional exponent," whose recognition is the job of states 16 through 19. Should we, in state 15, instead see anything but E or a digit, then we have come to the end of the fraction, there is no exponent and we return the lexeme found, via state 21.

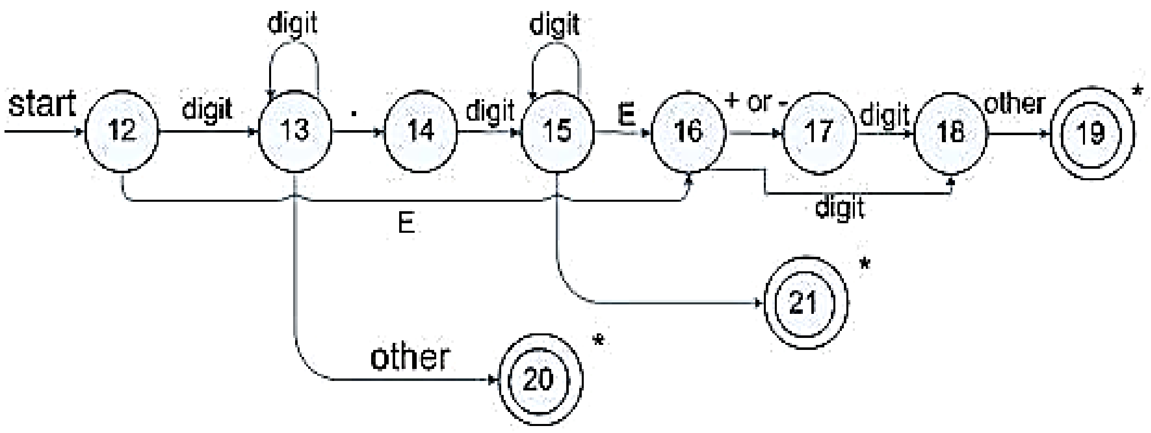


Fig 1.13 A transition diagram for unsigned numbers

4. Transition diagram for whitespace

The diagram itself is quite simple reflecting the simplicity of the corresponding regular expression.

- The delim in the diagram represents any of the whitespace characters, say space, tab, and newline.
- The final star is there because we needed to find a non-whitespace character in order to know when the whitespace ends and this character begins the next token.
- There is no action performed at the accepting state. Indeed the lexer does not return to the parser, but starts again from its beginning as it still must find the next token

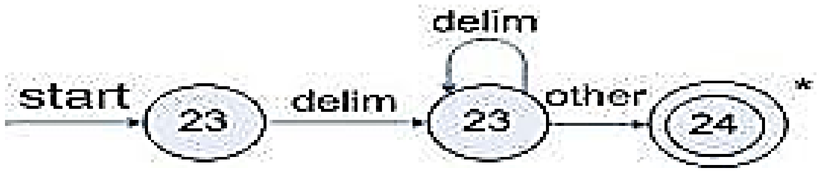


Fig 1.14 A transition diagram for whitespace

1.9 FINITE AUTOMATA

Finite automata are recognizer; they simply say “yes” or “no” about each possible input string.

There are two types of finite automata:

a) Nondeterministic finite automata (NFA)

b) Deterministic finite automata (DFA)

The mathematical model of finite automata consists of:

- Finite set of states (Q)
- Finite set of input symbols (Σ)
- One initial state (q0)
- Set of final states (qf)
- Transition function (δ)

The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ),

$Q \times \Sigma \rightarrow Q$

Construction of Finite Automata

Let L(r) be a regular language recognized by some finite automata (FA).

States: States of FA are represented by circles. State names are written inside circles.

Start state: The state from where the automata start is known as the start state. Start state has an arrow pointed towards it.

Intermediate states: All intermediate states have at least two arrows; one pointing to and another pointing out from them.

Final state: If the input string is successfully parsed, the automaton is expected to be in this state. Final state is represented by double circles. It may have any odd number of arrows pointing to it and even number of arrows pointing out from it. The number of odd arrows are one greater than even, i.e. odd = even+1.

Transition: The transition from one state to another state happens when a desired symbol in the input is found. Upon transition, automata can either move to the next state or stay in the same state. Movement from one state to another is shown as a directed arrow, where the arrows point to the destination state. If automata stay on the same state, an arrow pointing from a state to itself isdrawn.

1.9.1 NONDETERMINISTIC FINITE AUTOMATA (NFA)

A nondeterministic finite automaton (NFA) consists of:

1. A finite set of states S.
2. A set of input symbols Σ, the input alphabet. We assume that e, which stands for the empty string, is never a member of Σ.
3. A transition function that gives, for each state, and for each symbol in Σ U {e} a set of next states.
4. A state so from S that is distinguished as the start state (or initial state).
5. A set of states F, a subset of S that is distinguished as the accepting states (or final states).

Representation of NFA

Transition graph or Transition diagram or state diagram

We can represent either an NFA or DFA by a transition graph, where the notes are states and the labelled edges represent the transition function. There is an edge labeled a from state **s** to state **t** if and only if **t** is one of the next states for state s and input **a**. This graph is very much like a transition diagram except:

- The same symbol can label edges from one state to several different stages
- An edge may be labelled by **ϵ,** the empty string, instead of , or in addition to, symbols from input alphabet

Transition table

We can also represent an NFA by a transition table, whose rows correspond to states and whose columns corresponding to the input symbols and **ϵ**. The entry for a given state and input is the value of the transition function applied to these arguments. If the transition function has no information about that state-input pair, we put Φ in the table for the pair

Table 1.7 Transition table

STATE	a	B	ϵ
0	{0, 1}	{0}	Φ
1	Φ	{2}	Φ
2	Φ	{3}	Φ
3	Φ	Φ	Φ

1.9.2 DETERMINISTIC FINITE AUTOMATA (DFA)

The finite automata which satisfy the following conditions can be called as Deterministic finite automata

- It should have one start state
- It should have one or more accepting or final states
- It should not allow € transitions
- It should allow only one transitions on a particular i/p symbol

Formal Definition of DFA

M= (Q, Σ, δ, q0, F)

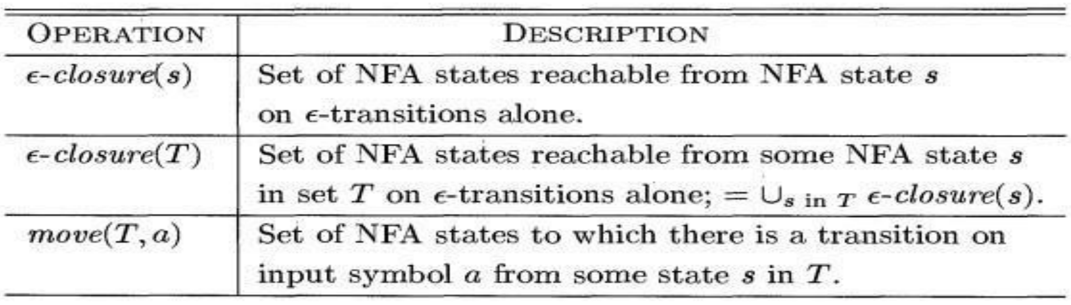
1) The finite set of states which can be denoted by Q.
2) The finite set of input symbols Σ.
3) The start state q_0 such that $q_0 \epsilon$ Q.
4) A set of final states F such that F ϵ Q.
5) The mapping function or transition function denoted by δ.

1.10 REGULAR EXPRESSIONS TO AUTOMATA

1.10.1 CONVERSION OF AN NFA TO A DFA: The Subset Construction Algorithm

The algorithm for constructing a DFA from a given NFA such that it recognizes the same language is called subset construction. The reason is that each state of the DFA machine corresponds to a set of states of the NFA. The DFA keeps in a particular state all possible states to which the NFA makes a transition on the given input symbol. In other words, after processing a sequence of input symbols the DFA is in a state that actually corresponds to a set of states from the NFA reachable from the starting symbol on the same inputs.

Table 1.8 Three operations that can be applied on NFA states

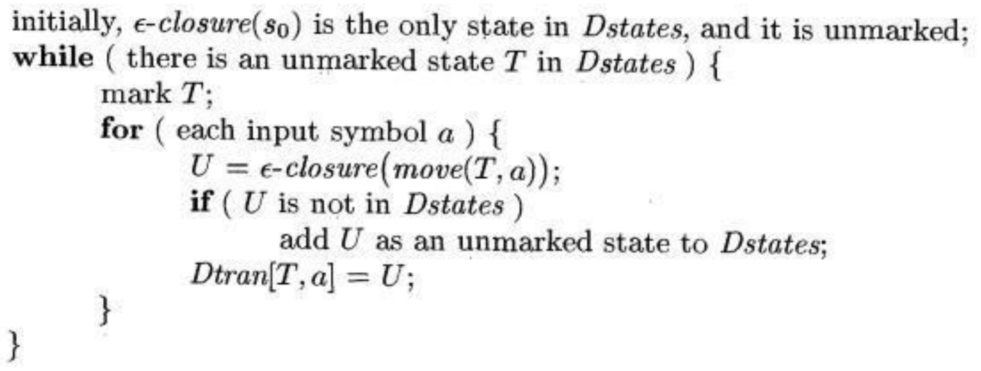


OPERATION	DESCRIPTION
ϵ-*closure*(s)	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-*closure*(T)	Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T}\ \epsilon\text{-}closure(s)$.
move(T, a)	Set of NFA states to which there is a transition on input symbol a from some state s in T.

We must explore those sets of states that *N* can be in after seeing some input string. As a basis, before reading the first input symbol, *N* can be in any of the states of ϵ *-closure(so)*, where *S0* is its start state. For the induction, suppose that *N* can be in set of states *T* after reading input string *x*. If it next reads input a, then *N* can immediately go to any of the states in *move(T, a)*. However, after reading a, it may also make several ϵ -transitions; thus *N* could be in any state of ϵ*-closure(move(T, a))* after reading input *xa*. Following these ideas, the construction of the set of D’s states, *Dstates*, and its transition function *Dtran*, is shown below.

The start state of *D* is ϵ *-closure(so)*, and the accepting states of *D* are all those sets of N's states that include at least one accepting state of *N*. To complete our description of the subset construction, we need only to show how initially, ϵ *-closure(s_0)* is the only state in *Dstates*, and it is unmarked;

Subset construction



```
initially, $\epsilon$-closure($s_0$) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state T in Dstates ) {
        mark T;
        for ( each input symbol a ) {
                U = $\epsilon$-closure(move(T, a));
                if ( U is not in Dstates )
                        add U as an unmarked state to Dstates;
                Dtran[T, a] = U;
        }
}
```

Computing ϵ *-closure(T)*

Example: Convert the NFA for the expression: (a|b)*abb into a DFA using thesubset construction algorithm.

Step 1: Convert the above expression in to NFA using Thompson rule constructions.

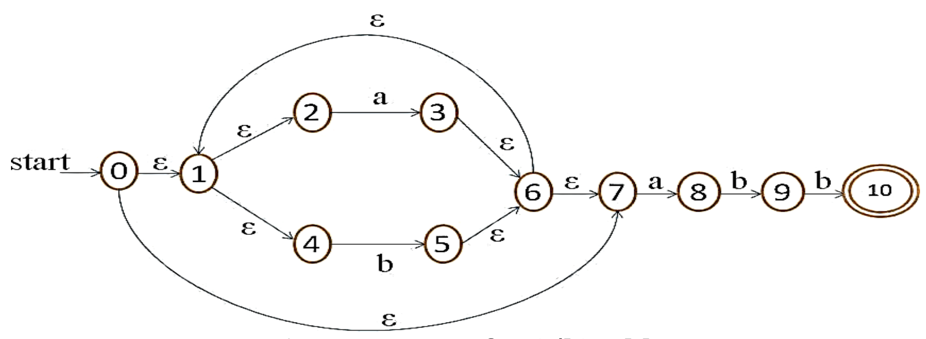


Fig.1.15 NFA N for (a|b)*abb

Step 2: Start state of equivalent DFA is **ε-closure(0)**

ε-closure(0) ={ 0,1,2,4,7}

Step 2.1: Compute **ε-closure(move(A,a))**

move(A,a) ={3,8}

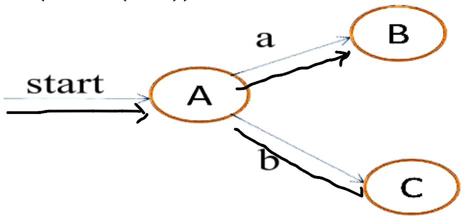
ε-closure(move(A,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}

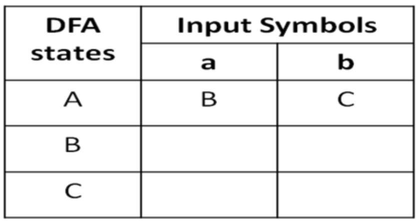
ε-closure(move(A,a))

={1,2,3,4,6,7,8}

Dtran[A,a]=B

-Step 2.2: Compute **ε-closure(move(A,b))**
move(A,b) ={5}
ε-closure(move(A,b)) = ε-closure(5) = {5,6,7,1,2,4}
ε-closure(move(A,a)) ={1,2,4,5,6,7} Dtran[A,b]=C.

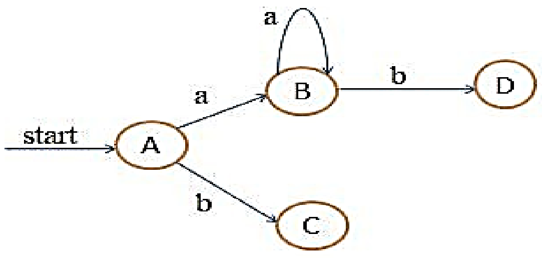


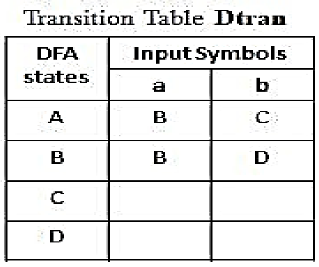


DFA states	Input Symbols	
	a	b
A	B	C
B		
C		

Fig.1.16 DFA and Transition table after step 2

Step 3: Compute Transition from **state B** on
input symbol {a,b} B ={1,2,3,4,6,7,8}
Step 3.1: Compute **ε-closure(move(B,a))**
move(B,a) ={3,8}
ε-closure(move(B,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}
ε-closure(move(B,a)) ={1,2,3,4,6,7,8} Dtran[B,a]=B
Step 3.2: Compute **ε-closure(move(B,b))** move(B,b)
={5,9}
ε-closure(move(B,b)) = ε-closure(5,9) = {5,9,6,7,1,2,4}
ε-closure(move(B,b)) ={1,2,4,5,6,7,9} Dtran[B,b]=D



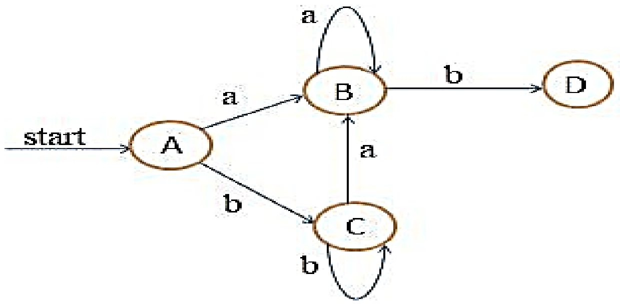


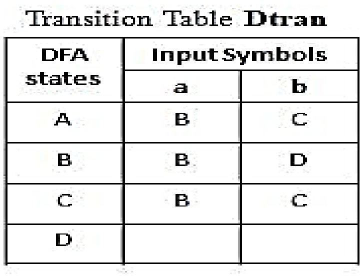
Transition Table Dtran

DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C		
D		

Fig.1.17 DFA and Transition table after step 3
Step 4: Compute Transition from **state C** on
input symbol {a,b} C = {1,2,,4,5,6,7}
Step 4.1: Compute **ε-closure(move(C,a))**
move(C,a) ={3,8}
ε-closure(move(C,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}
ε-closure(move(C,a))={1,2,3,4,6,7,8}
Dtran[C,a]=B
Step 4.2: Compute **ε-closure(move(C,b))**
move(C,b) ={5}

ε-closure(move(C,b)) = ε-closure(5) = {5,6,7,1,2,4}
ε-closure(move(C,b))={1,2,4,5,6,7}
Dtran[C,b]= C
DFA and Transition table after step 4 is shown below.




Transition Table Dtran

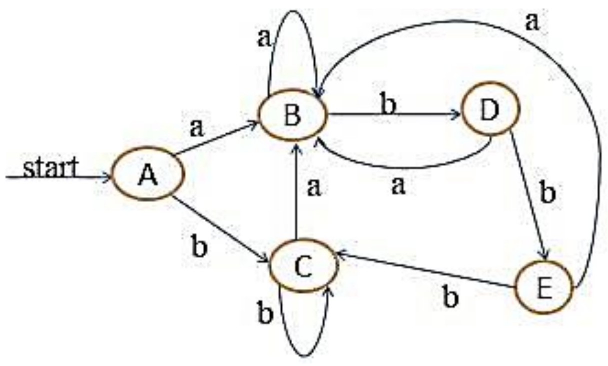
DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C	B	C
D		

Fig.1.18 DFA and Transition table after step 4

Step 5: Compute Transition from **state D** on input symbol {a,b}
D = {1,2,,4,5,6,7,9}
Step 5.1: Compute **ε-closure(move(D,a))**
move(D,a) ={3,8}
ε-closure(move(D,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}
ε-closure(move(D,a)) ={1,2,3,4,6,7,8}
Dtran[D,a]= B
Step 5.2: Compute **ε-closure(move(D,b))**
move(D,b) ={5,10}
ε-closure(move(D,b)) = ε-closure(5,10) = {5,10,6,7,1,2,4}
ε-closure(move(C,b))={1,2,4,5,6,7,10}
Dtran[D,b]= E
Step 6: Compute Transition from **state E** on
input symbol {a,b} E = {1,2,,4,5,6,7,10}

Step 6.1: Compute **ε-closure(move(E,a))**
move(E,a) ={3,8}
ε-closure(move(E,a)) = ε-closure(3,8) = {3,8,6,7,1,2,4}
ε-closure(move(E,a)) ={1,2,3,4,6,7,8}
Dtran[E,a]= B
Step 6.2: Compute **ε-closure(move(E,b))**
move(E,b) ={5}
ε-closure(move(E,b)) = ε-closure(5) = {5,6,7,1,2,4}
ε-closure(move(E,b)) ={1,2,4,5,6,7}
Dtran[E,b]= C

DFA and Transition table after step 6 is shown below.



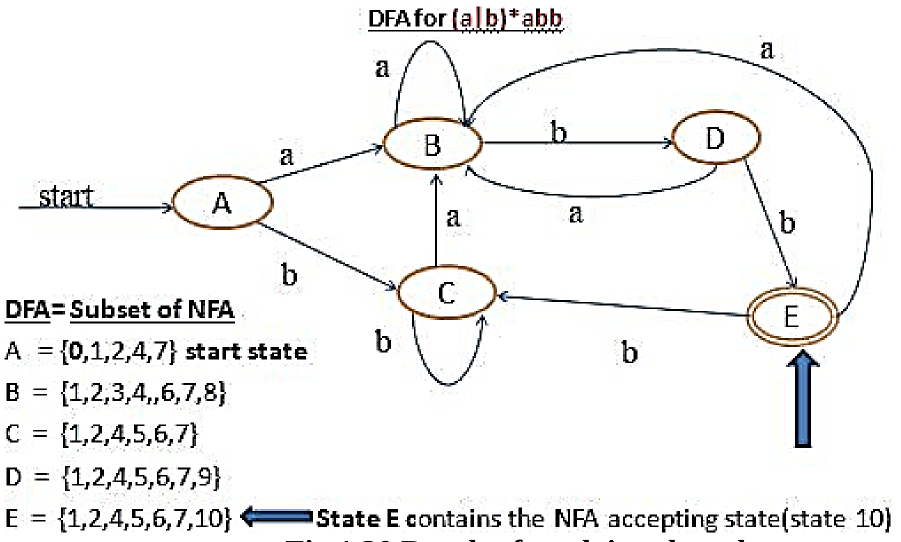
DFA states	Input Symbols	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Fig.1.19 DFA and Transition table after step 6

Step 7: No more new DFA states are formed.

Stop the subset construction method.

The start state and accepting states are marked the DFA.



DFA= Subset of NFA

A = {0,1,2,4,7} start state

B = {1,2,3,4,,6,7,8}

C = {1,2,4,5,6,7}

D = {1,2,4,5,6,7,9}

E = {1,2,4,5,6,7,10} State E contains the NFA accepting state(state 10)

Fig.1.20 Result of applying the subset construction

1.10.2 CONSTRUCTION OF NFA FROM REGULAR EXPRESSION

The McNaughton-Yamada-Thompson algorithm is used to convert a regular expression to anNFA.

Thompson algorithm

Input: A regular expression r over alphabet Σ

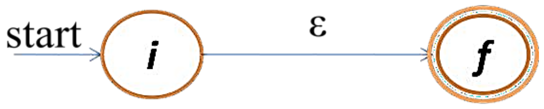
Output: An NFA N accepting (r)

Method:

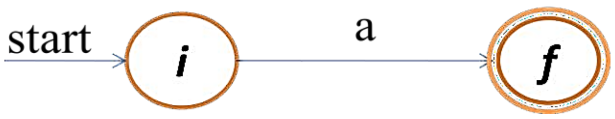
- Begin by parsing r into its constituent sub-expressions
- The rules for constructing an NFA consist of basis rules for handling sub expressions with no operators
- Induction rules for constructing larger NFA‟s from the NFA‟s for the immediate Sub

expressions of a given expression with operators

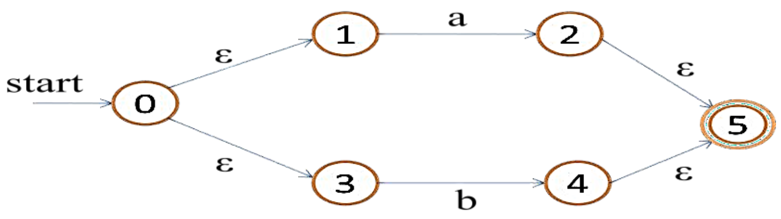
1. R = ε



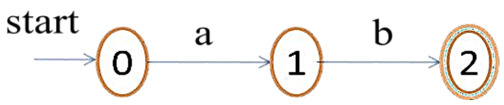
2. R = a



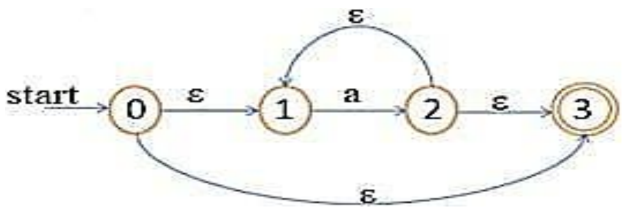
3. R= a | b



4. R=ab



5. R= a*



In general the rules if Thompson's construction method is given below

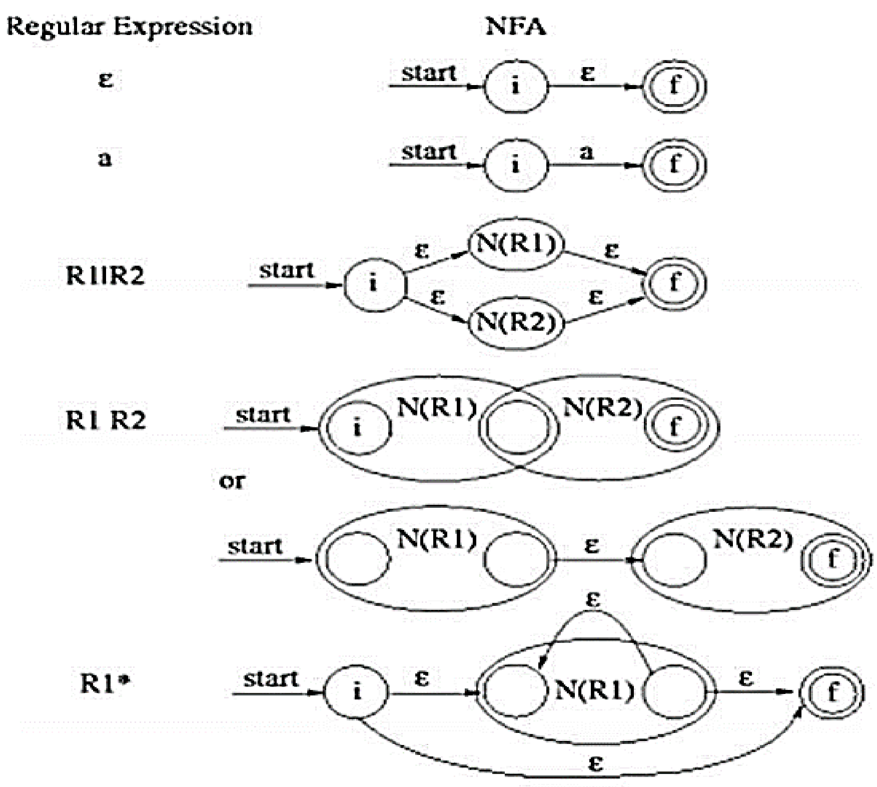


Fig.1.21 Thompson's construction method

Problems:

1. Construct NFA for: $(a+b)^{*}abb$

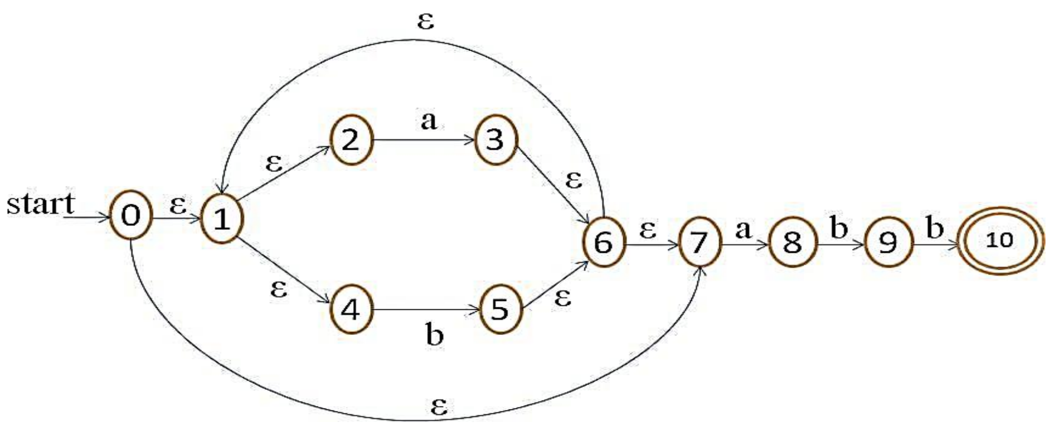


Fig.1.22 NFA for (a+b)*abb

1.11 MINIMIZING DFA

- Lexical analyser can be implemented by DFA. The number of states of constructed DFA must be minimized as for each state entry is made in a table which describes the lexical analyser
- Minimization of DFA focuses on reducing the number of states in the given finite automata

- The states which are equivalent are grouped thereby enabling the reduction of states
- Two states are said to be equivalent if on taking the same input symbol they transit to same group of states (final or non-final). Otherwise, they are distinguishable states

Algorithm:

1. Start with initial partition Π with two groups, F and Q-F, the accepting and non-accepting states of D
2. Construct a new partition Π new as follows:

Let Πnew=Π

For each group G of Π

{partition G into subgroups such that two states p and q are in same group iff for all input symbols a, states p and q have transitions to states in same group of Π.

Replace G in Πnew by the set of subgroups formed.}

3. If Πnew=Π stop partitioning, otherwise repeat step 2
4. Choose representative states from each group and construct finite automata.

Example: After applying minimized DFA algorithm for the regular expression (a|b)*abb the transition table for the minimized DFA becomes Transition table for Minimized state DFA:

DFA states	Input Symbols	
	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Minimized DFA:

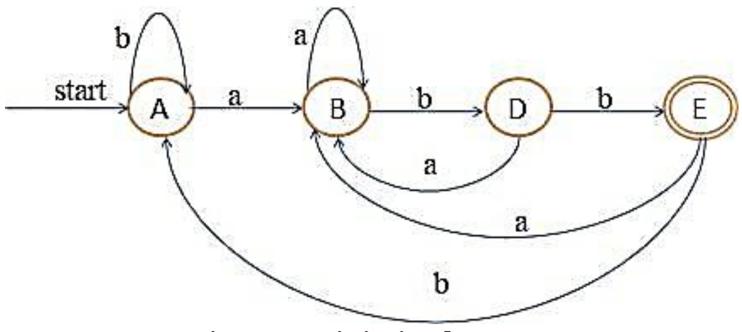


Fig.1.23 Minimized DFA

1.12 Language for Specifying Lexical Analyzers

A specialized **pattern-description language** used to define tokens and automatically generate lexical analyzers, mainly implemented using **LEX** (or FLEX in updated systems).

1.12.1 Purpose

- Helps **describe token patterns** using **regular expressions**.
- Automatically produces a **scanner** (lexical analyzer) that converts input characters → meaningful

lexemes (tokens).

1.12.2 Structure of LEX Program

A LEX specification consists of **three sections**:

```
%{
  Declarations (C code, headers)
%}



Definitions
%%
Pattern      Action
Pattern      Action
%%
User Code (optional)
```

1.12.2.1 Declarations Section

- Written in { }.
- Includes C header files, variable declarations, prototypes.

1.12.2.2 Definitions Section

Contains **regular definitions / macros** to simplify patterns.

Example:

```
DIGIT   [0-9]
  ID      [a-zA-Z_][a-zA-Z0-9_]*
```

1.12.2.3 Rules Section

- **Core part**: patterns with actions
- Pattern = Regular expression, Action = C code executed when pattern matches.

Example:

```
  %%
[0-9]+      { printf("NUMBER"); }
[a-zA-Z_][a-zA-Z0-9_]*  { printf("IDENTIFIER"); }
"+"        { printf("PLUS"); }
%%
```

1.12.3 Working / Process

1.12.3.1 Write LEX program (.l file).

1.12.3.2 Run

```
lex filename.l
cc lex.yy.c -ll
```

1.12.3.3 Output: **lex.yy.c**

Contains code for lexical analyzer.

1.12.3.4 Executable scanner reads input and prints tokens.

1.12.4 Patterns Used (Regular Expressions)

Symbol	**Meaning**
*	zero or more occurrences
+	one or more occurrences

?	one or more occurrences
[abc]	any character a/b/c
[^abc]	any character except a/b/c
( )	grouping
.	any character except newline

1.12.4.1 Example Program (Simple Token Identifier)

```
%{
#include <stdio.h>
%}

DIGIT   [0-9]
ID      [a-zA-Z_][a-zA-Z0-9_]*
%%
{DIGIT}+       { printf("NUMBER "); }
{ID}           { printf("IDENTIFIER "); }
"+"            { printf("PLUS "); }
.              { /* ignore other chars */ }
%%
int main() {
    yylex();
    return 0;
      }
```

1.12.5 Advantages of LEX Language

- Automates scanner creation: **no need to hand-code loops/matches**.
- Handles complex token patterns using simple regular expressions.
- Integrates with **YACC** for full compiler construction.

1.12.6 Output of LEX

A **token stream** passed to parser (e.g., identifiers, numbers, operators).

1.13. Design of Lexical analyzer generator(LEX)

Lexical Analyzer Generator is typically implemented using a tool. There are some standard tools available in the UNIX environment. Some of the standard tools are

- LEX – it helps in writing programs whose flow of control is regulated by the various definitions of regular expressions in the input stream. The wrapper programming language is C.
- FLEX – It is a faster lexical analyzer tool. This is also a C language version of the LEX tool.
- JLEX – This is a Java version of LEX tool.

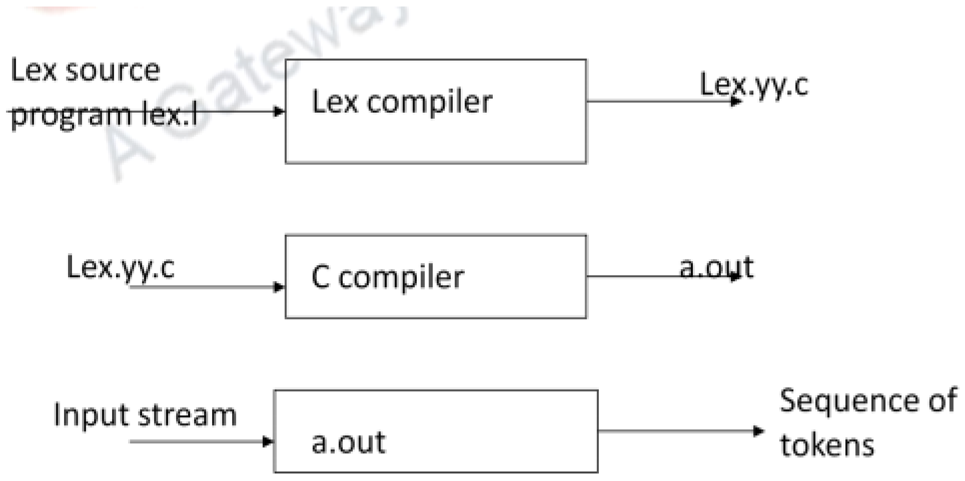


Fig. 1.24 Control flow of a lex program

The input will be a source file with a ―.l‖ extension. This will be compiled by a Lex compiler and the output of this compiler will be a source file in C named as ―lex.yy.c‖. This can be compiled using a C compiler to get the desired output.

1.13.1.1Creating a lexical analyzer with Lex (or) Specification of LEX

A Lex program (the lex.l file) consists of three parts:

```
%{
auxiliary declarations
%}
regular definitions
%%
translation rules
%%
auxiliary procedures
```

1.13.1.2 Declarations section:

It includes declarations of variables, manifest constants (A manifest constant is an identifier that is declared to represent a constant e.g. # define PIE 3.14), the files to be included and definitions of regular expressions.

1.13.1.3 Transition rules section:

The translation rules of a Lex program are statements of the form:

```
p1        {action 1}
p2        {action 2}
p3        {action 3}
..          ...
```

Where each P1 is a regular expression called a pattern, over the alphabet consisting of Σ and the auxiliary definition names. The patterns describe the form of the tokens.

• Each action 1is a program fragment describing what action the lexical analyzer should take when token P's found. The actions are written in a conventional programming language, rather than any particular language, we use pseudo language.

• To create the lexical analyzer L, each of the actions must be compiled into machine code

1.13.1.4 Auxiliary procedures:

The third section holds whatever auxiliary procedures are needed by the actions. Alternatively, these procedures can be compiled separately and loaded with the lexical analyzer. The auxiliary procedures are written in C language.

Auxiliary Definition

Letter= A|B|...|Z

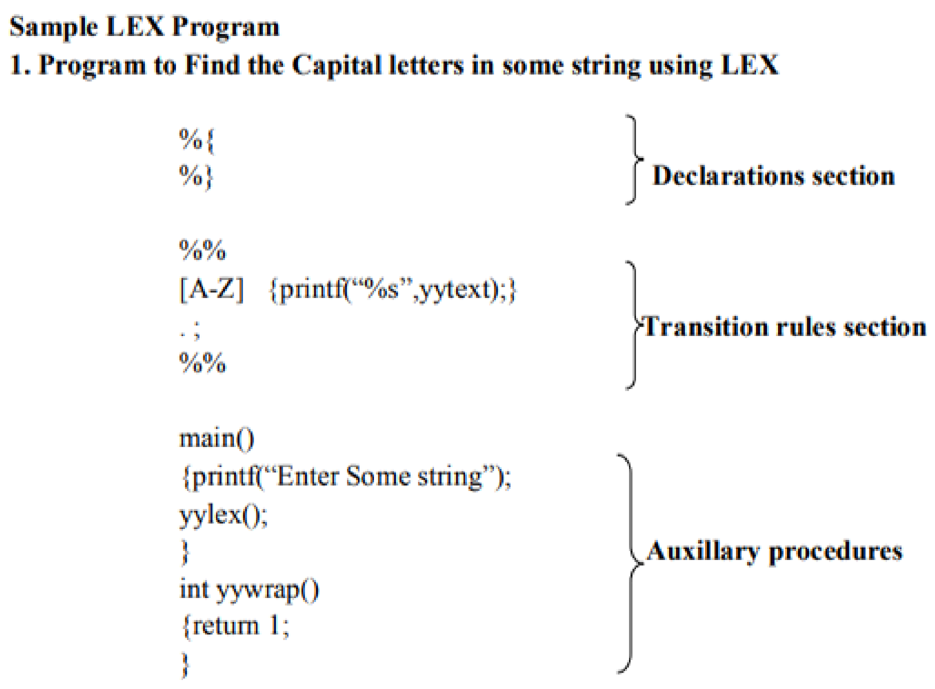
Digit= 0|1|...|9

1.13.1.5 How does this Lexical analyzer work?

- The lexical analyzer created by Lex behaves in concert with a parser in the following manner. When activated by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it has found the longest prefix of the input that is matched by one of the regular expressions p.
- Then it executes the corresponding action. Typically, the action will return control to the parser.
- However, if it does not, then the lexical analyzer proceeds to find more lexemes, until an action causes control to return to the parser. The repeated search for lexemes until an explicit return allows the lexical analyzer to process white space and comments conveniently.
- The lexical analyzer returns a single quantity, the token, to the parser. To pass an attribute value with information about the lexeme, we can set the global variable yylval.

1.13.1.6 LEX Actions

- yytext() is a variable that is a pointer to the first character of the lexeme.
- yywrap() yywrap is called when lexical analyzer reach end of file. It yywrap returns a then lexical analyzer continue scanning. When yywrap return 1 means end of file is encountered.
- yyleng() is an integer telling how long the lexeme is.
- yyin() –It is used to read the source program from file and then stored in yyin.



Sample LEX Program

1. Program to Find the Capital letters in some string using LEX

```
%{
%}

%%
[A-Z]   {printf("%s",yytext);}
. ;
%%

main()
{printf("Enter Some string");
yylex();
}
int yywrap()
{return 1;
}
```

Input:
Enter Some String
IFET College of Engineering and Technology
Output:
ICET

1.14 Recent trends in Compiler Design.

Compiler design has evolved far beyond simple translation of high-level code into machine instructions, driven by advances in artificial intelligence, heterogeneous hardware, and modern software ecosystems. A major trend is the integration of **AI and machine learning** into compiler optimization phases, where models assist in tasks such as register allocation, instruction scheduling, and performance tuning, enabling smarter and adaptive optimization strategies tailored to specific hardware profiles. Another significant development is the use of **multi-level intermediate representations (MLIR)** and modular compiler infrastructures that improve code reuse, optimization across abstraction layers, and support for diverse targets including CPUs, GPUs, and domain-specific accelerators. Hybrid compilation techniques like **Just-In-Time (JIT) and Ahead-Of-Time (AOT)** compilation are increasingly balanced to combine runtime adaptability with predictable performance, while **WebAssembly** has emerged as a universal, secure compilation target for web and edge applications. Research and practice also focus on **compilers for domain-specific languages (DSLs)** and emerging architectures like quantum processors, expanding compiler applicability beyond traditional software into specialized AI and scientific domains. Furthermore, safety and correctness have gained attention through techniques such as static analysis, fuzz testing, and formal verification to ensure robust code generation in trusted computing environments. These trends reflect a shift where compilers act not just as translators but as intelligent, architecture-aware systems that optimize for performance, portability, and security in complex modern computing landscapes.