Syllabus

Course Title: Complier Design and Construction
Course no: CSC-352 **Full Marks**: 70+10+20
Credit hours: 3
Pass Marks: 28+4+8

Nature of course: Theory (3 Hrs.) + Lab (3 Hrs.)
Course Synopsis: Analysis of source program. The phases of compiler.
Goal: This course introduces fundamental concept of compiler and its different phases.
Course Contents:

Unit. 1:
1.1 Introduction to compiling: Compilers, Analysis of source program, the phases of compiler, compiler-construction tools. **4 hrs.**
1.2 A Simple One-Pass Compiler: Syntax Definition, Syntax directed translation, Parsing, Translator for simple expression, Symbol Table, Abstract Stack Machines. **5 hrs**

Unit 2:
2.1 Lexical Analysis: The role of the lexical analyzer, Input buffering, Specification of tokens, Recognition of tokens, Finite Automata, Conversion Regular Expression to an NFA and ten to DFA, NFA to DFA, State minimization in DFA, Flex/lex introduction. **8 Hrs.**

2.2 Syntax Analysis: The role of parser, Context frees grammars, Writing a grammars, Top-down parsing, Bottom-up parsing, error recovery mechanism, LL grammar, Bottom up parsing-Handles, shift reduced parsing, LR parsers-SLR,LALR,LR,LR/LALR Grammars, parser generators. **10 Hrs.**

Unit 3:
3.1 Syntax Directed Translation: Syntax-directed definition, Syntax tree and its construction, Evaluation of S-attributed definitions, L-attributed, Top-down translation, Recursive evaluators. **5 Hrs.**
3.2 Type Checking: Type systems, Specification of a simple type checker, Type conversions equivalence of type expression, Type checking Yacc/Bison. **3 Hrs.**

Unit 4:
4.1 Intermediate Code Generation: Intermediate languages, three address code, Declarations, Assignments Statements, Boolean Expressions, addressing array elements, case statements, Back patching, procedure calls. 4 **Hrs.**
4.2 Code Generation and optimization: Issues in design of a code generator, the target machine, Run –time storage management, Basic blocks and flow graphs, next use information's, a simple code generator, Peephole organization, generating code from dags. **6 Hrs.**

Sample question

Third Year/Six Semester

Subject: Compiler Design and Construction **FM: 60**

Time: 3 hours **PM: 24**

Candidates are required to give their answer in their own words as for as practicable.
Attempt all the questions.
Every question contains equal marks.

1. What do mean by compiler? How source program analyzed? Explain in brief.
2. Discuss the role of symbol table in compiler design.
3. Convert the regular expression '0+ (1+0)*00' first into NFA and then into DFA using Thomson's and Subset Construction methods.
4. Consider the grammar:
   S->(L)| a
   L->L, S|S
5. Consider the grammar
   C→AB
   A →a
   B→ a

Calculate the canonical LR (0) items.

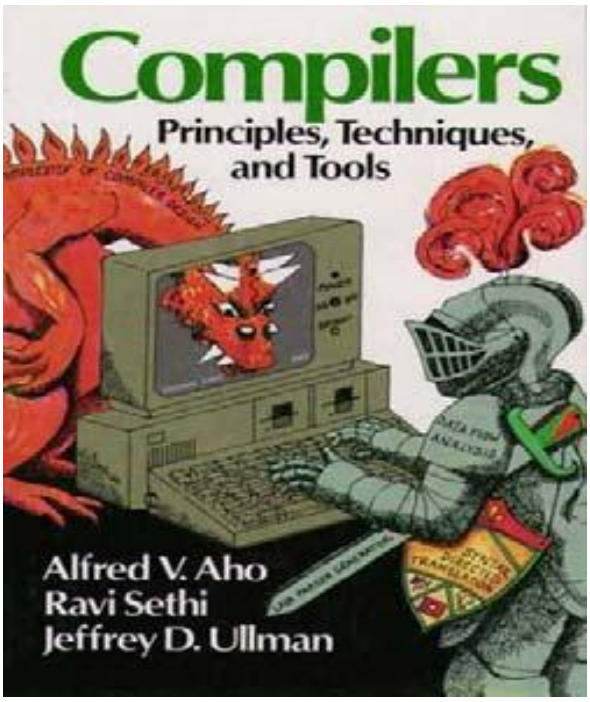
6. Describe the inherited and synthesized attributes of grammar using an example.
7. Write the type expressions for the following types.
   a. An array of pointers to reals, where the array index range from 1 to 100.
   b. Function whose domains are function from characters and whose range is a pointer of integer.
8. What do you mean by intermediate code? Explain the role of intermediate code in compiler design.
9. What is operation of simple code generator? Explain.
10. Why optimization is often required in the code generated by simple code generator? Explain the unreachable code optimization.

Prerequisites

– Introduction to Automata and Formal Languages
– Introduction to Analysis of Algorithms and Data Structures
– Working knowledge of C/C++
--Introduction to the Principles of Programming Languages, Operating System & Computer Architecture is plus

Resources

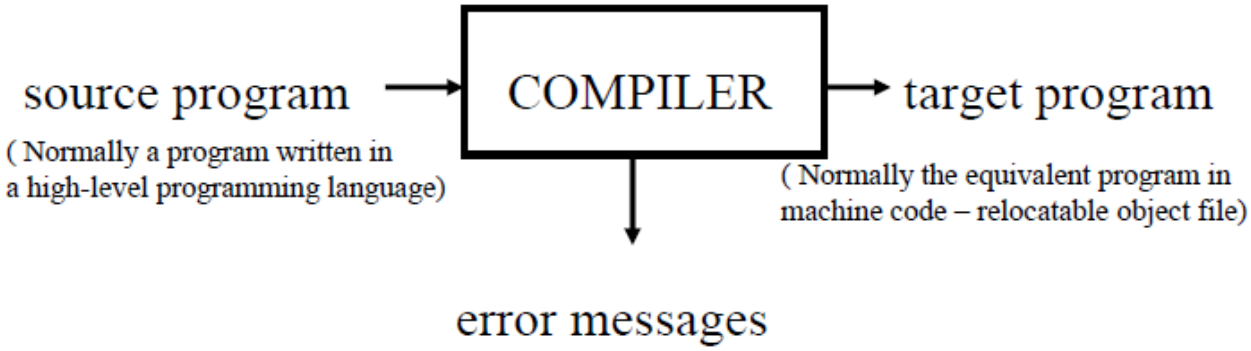
Text Book: Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986



References: Research and technical papers.

What is Compiler?

A **compiler** is a program that takes a program written in a *source language* and translates it into an equivalent low level program in a *target language*.



Phases of a Compiler

There are two major parts of a compiler: **Analysis** and **Synthesis**
– In analysis phase, an intermediate representation is created from the given source program. This phase (Source code analysis phase) is mainly divided into following three parts:

- Lexical Analyzer
- Syntax Analyzer and

- Semantic Analyzer

– In synthesis phase, the equivalent target program is created from this intermediate representation. This phase is divided into following three parts:

- Intermediate Code Generator
- Code Optimizer and
- Final Code Generator

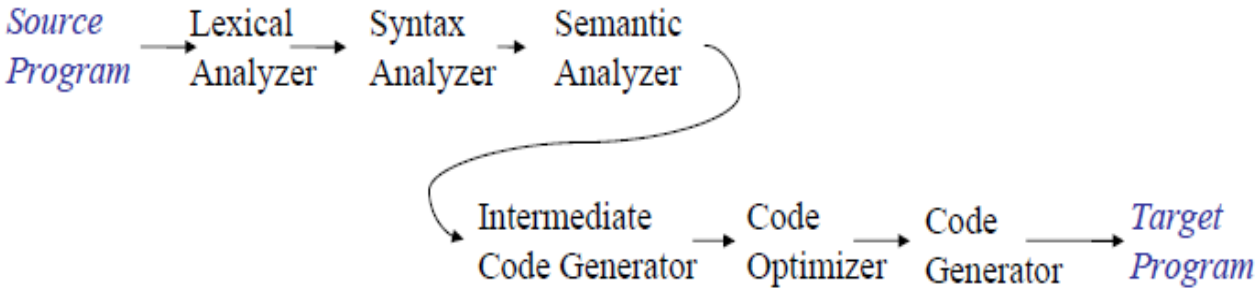


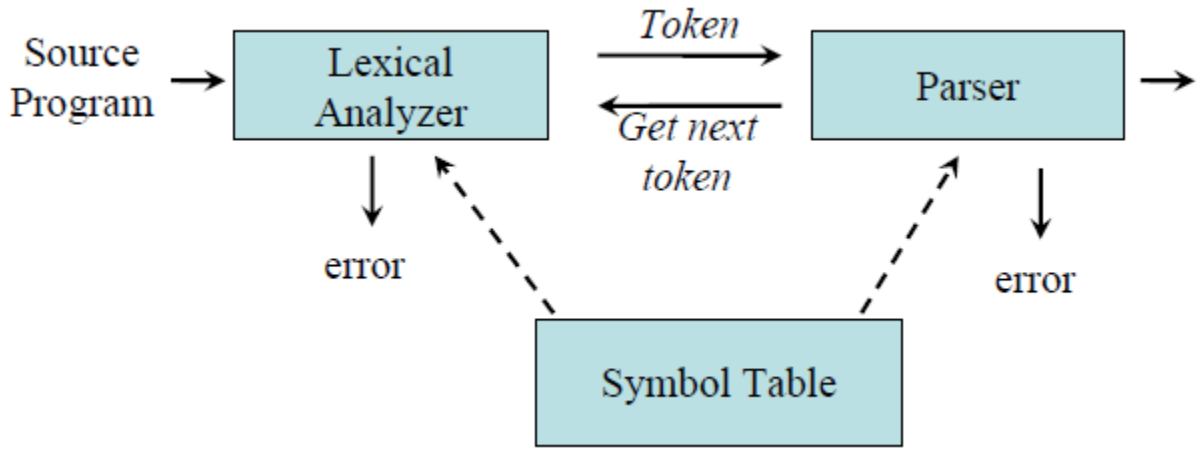
Figure: Phases of a compiler

Lexical Analyzer

Lexical Analyzer reads the source program in character by character ways and returns the *tokens* of the source program.

Normally a lexical analyzer doesn't return a list of tokens, it returns a token only when the parser asks a token from it.

Lexical analyzer may also perform other auxiliary operation like removing redundant white space, removing token separator (like semicolon) etc.



Example:

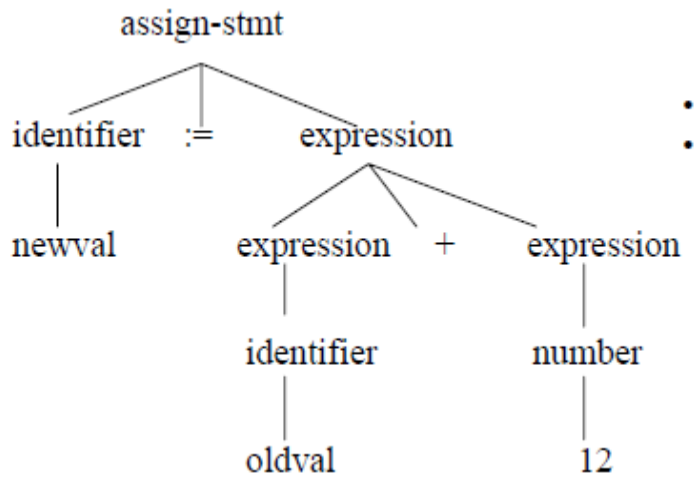
newval := oldval + 12

tokens:	newval	identifier
	:=	assignment operator
	oldval	identifier
	+	add operator
	12	a number

Put information about identifiers into the symbol table.
Regular expressions are used to describe tokens (lexical constructs).
A (Deterministic) Finite State Automaton (DFA) can be used in the implementation of a lexical analyzer.

Syntax Analyzer

A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given source program.
Syntax analyzer is also called the **parser.** Its job is to analyze the source program based on the definition of its syntax. It works in lock-step(fault tolerance of system) with the lexical analyzer and is responsible for creating a parse-tree of the source code.
Ex: newval := oldval + 12



- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar.

The syntax of a language is specified by a **context free grammar** (CFG).
The rules in a CFG are mostly recursive.
A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.
– If it satisfies, the syntax analyzer creates a parse tree for the given program.

Semantic Analyzer

A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
– Type-checking is an important part of semantic analyzer.
Ex: newval := oldval + 12
• The type of the identifier *newval* must match with type of the expression *(oldval+12)*

Synthesis phase

Intermediate Code Generation

An intermediate language is often used by many compiler for analyzing and optimizing the source program. The intermediate language should have two important properties:
– It should be simple and easy to produce.
– It should be easy to translate to the target program
A compiler may produce an explicit intermediate codes representing the source program.
These intermediate codes are generally machine (architecture) independent. But the level of intermediate codes is close to the level of machine codes.

Ex:

newval := oldval * fact + 12

id1:= id2 * id3 + 12
temp1 = intTofloat(12)
temp2 = id2 * id3
temp3 = temp1 + temp2
id1 = temp3

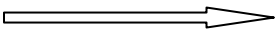
Code Optimization

The process of removing unnecessary part of a code is known as code optimization. Due to code optimization process it decreases the time and space complexity of the program. i.e

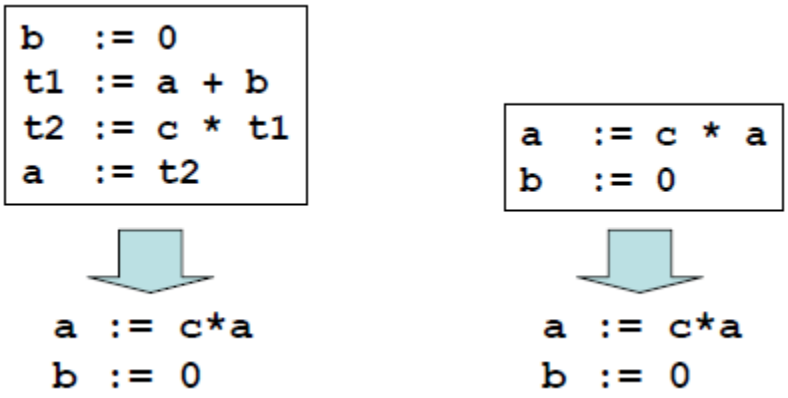
Detection of redundant function calls
Detection of loop invariants
Common sub-expression elimination
Dead code detection and elimination

Ex:

temp1 = intTofloat (12)
temp2 = id2 * id3
temp3 = temp1 + temp2
id1 = temp3



temp1 = id2 * id3
id1 = temp1 + 12



Code Generation

This involves the translation of optimized intermediate code into the target language
The target code is normally is a relocatable object file containing the machine or assembly codes.

Ex:

(Assume that we have an architecture with instructions whose at least one of its operands is a machine register)

```
MOVE    id2, R1
MULT    id3, R1
ADD     #1, R1
MOVE    R1, id1
```

Cousins of Compiler

There are several major kinds of compilers:

Native Code Compiler

Translates source code into hardware (assembly or machine code) instructions. Example: gcc.

Virtual Machine Compiler

Translates source code into an abstract machine code, for execution by a virtual machine interpreter. Example: javac.

JIT Compiler

Translates virtual machine code to native code Example: Sun's Hotspot java machine.

Preprocessor

Translates source code into simpler or slightly lower level source code, for compilation by another compiler. Examples: cpp, m4.

Pure interpreter

Executes source code on the fly, without generating machine code. Example: Lisp.

Compiler Construction Tools

- *Scanner Generators:*
  Generates lexers automatically i.e. tokenize the source program automatically. Example: flex, lex, etc
- *Parser Generators:*
  Produces syntax analyzers automatically. Example: bison, yacc etc.
- *Syntax Directed Translation Engines:*
  Walks through parse tree generated by parser generator and generates intermediate code automatically. Example: bison, yacc, JavaCC etc.
- *Automatic Code Generators:*
  Take optimized intermediate code and generate target code automatically.
- *Data Flow Engines:*
  Optimizers using data flow analysis.

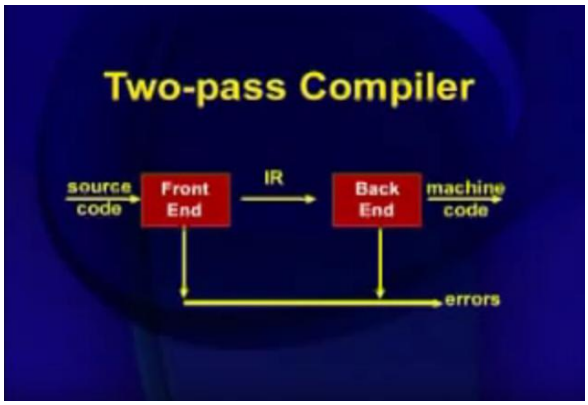
Hello World Program: The Assembly

First we need to write the hello world C program, which can be seen below:

```
#include &lt;stdio.h&gt;

int main() {
  printf("Hello World!");
  return 0;
}
```

```
        .file   "hello.c"
        .intel_syntax noprefix
        .section        .rodata
.LC0:
        .string "Hello World!"
        .text
.globl main
        .type   main, @function
main:
        push    ebp
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
        mov     eax, OFFSET FLAT:.LC0
        mov     DWORD PTR [esp], eax
        call    printf
        mov     eax, 0
        leave
        ret
        .size   main, .-main
        .ident  "GCC: (Gentoo 4.5.4 p1.0, pie-0.4.7) 4.5.4"
        .section        .note.GNU-stack,"",@progbits
```



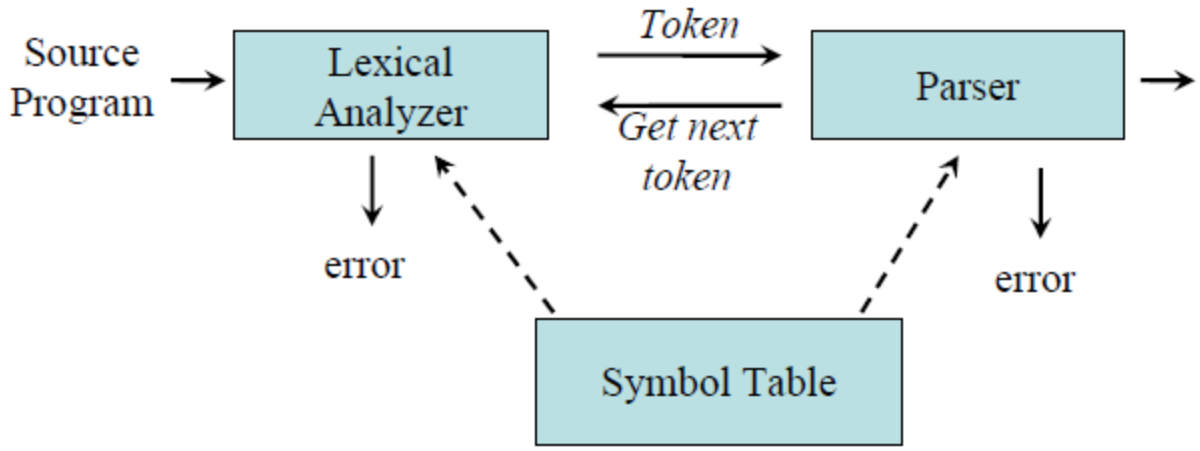
IR – Intermediate represenation

Lexical Analysis

Lexical Analyzer reads the source program in character by character ways and returns the *tokens* of the source program.

Normally a lexical analyzer doesn't return a list of tokens, it returns a token only when the parser asks a token from it.

Lexical analyzer may also perform other auxiliary operation like removing redundant white space, removing token separator (like semicolon) etc.



Example:
newval := oldval + 12

tokens: newval	identifier
:=	assignment operator
oldval	identifier
+	add operator
12	a number

Put information about identifiers into the symbol table.
Regular expressions are used to describe tokens (lexical constructs).
A (Deterministic) Finite State Automaton (DFA) can be used in the implementation of a lexical analyzer.

Tokens, Patterns, Lexemes

-A *token* is a logical building block of language. They are the sequence of characters having a collective meaning.
Eg: identifier, keywords etc

-A sequence of input characters that make up a single token is called a lexeme.
A token can represent more than one lexeme.
Eg: abc, 12 etc
In the statement
Float pi=3.1415
The variable pi is called a lexeme for the token 'identifier'
-Patterns are the rules for describing whether a given lexeme belonging to a token or not.
Regular expressions are widely used to specify patterns.

Attributes of Tokens

When a token represents more than one lexeme, lexical analyzer must provide additional information about the particular lexeme. This additional information is called as the *attribute* of the token.

For simplicity, a token may have a single attribute which holds the required information for that token.
Example: the tokens and the associated attribute for the following statement.
A=B*C+2
<id, pointer to symbol table entry for A>
<assig operator>
<id, pointer to symbol table entry for B>
<mult_op>
<id, pointer to symbol table entry for C>
<add_op>
<num, integer value 2>

Input Buffering

Many times, a scanner has to look ahead several characters from the current character in order to recognize the token.
For example `int` is keyword in C, while the term `inp` may be a variable name. When the character `'i'` is encountered, the scanner cannot decide whether it is a keyword or a variable name until it reads two more characters.
In order to efficiently move back and forth in the input stream, input buffering is used.

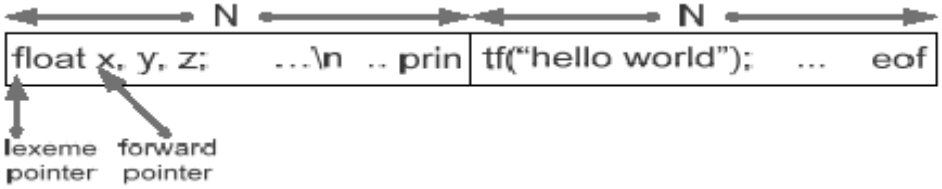


Fig: - An input buffer in two halves

Here, we divide the buffer into two halves with N-characters each.
Rather than reading character by character from file we read N input character at once. If there are fewer than N characters in input eof marker is placed.
There are two pointers (see in above fig.) the portion between lexeme pointer and forward pointer is current lexeme. Once the match for pattern is found, both the pointers points at the same place and forward pointer is moved.
The forward pointer performs tasks like below:

If forward at end of first half then,
    Reload second half
    Forward++
end if
else if forward at end of second half then,
    Reload first half
    Forward=start of first half
end else if
else
    forward++

Specification of Tokens

Stings & Languages

A *string s* is a finite sequence of symbols from Σ
– |*s* | denotes the length of string *s*
– ε denotes the empty string, thus |ε| = 0

Operators on Strings:

– Concatenation: xy represents the concatenation of strings x and y.

s ε = s

ε s = s

– S^n = s s s .. s (n times) s0 = ε

A *language* is a specific set of strings over some fixed alphabet Σ.

Operation on Languages

Concatenation: $L_1L_2 = \{ s_1s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$

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

Exponentiation: $L^0 = \{\varepsilon\}$ $L^1 = L$ $L^2 = LL$

Kleene Closure: $L^* = \cup_{i=0,\ldots,\infty} L^i$

Positive Closure: $L^+ = \cup_{i=1,\ldots,\infty} L^i$

Operation on Languages: Example

L1 = {a, b, c, d} L2 = {1, 2}
L1L2 = {a1, a2, b1, b2, c1, c2, d1, d2}
L1 ∪ L2 = {a, b, c, d, 1, 2}
$L1^3$ = all strings with length three (using a, b, c, d}
$L1^*$ = all strings using letters a, b, c, d and empty string
$L1^+$ = doesn't include the empty string

Regular Expressions

Regular expressions are the algebraic expressions that are used to describe tokens of a programming language.

If *r* and *s* are regular expressions denoting languages *L1(r)* and *L2(s)* respectively, then
– *r*+*s* is a regular expression denoting *L1* (*r*) ∪ *L2(s)*
– *rs* is a regular expression denoting *L1* (*r*) *L2(s)*
– *r** is a regular expression denoting (*L1* (*r*))*
– (*r*) is a regular expression denoting *L1* (*r*)

Properties of Regular Expressions

For regular expression r, s & t
r + s = s + r union is commutative)
r + (s + t) = (r + s) + t (union is associative)
(rs) t = r (st) (concatenation is associative)
r (s + t) = rs + rt (concatenation distributes over union)
ε r = r ε = r (ε is the identity element for concatenation)
r** = r* (closure is idempotent)

Regular Expressions: Examples

Given the alphabet A = {0, 1}
1. 1(1+0)*0 denotes the language of all string that begins with a '1' and ends with a '0'.
2. (1+0)*00 denotes the language of all strings that ends with 00 (binary number multiple of 4)
3. (01)*+ (10)* denotes the set of all stings that describe alternating 1s and 0s
4. (0* 1 0* 1 0* 1 0*) denotes the string having exactly three 1's.
5. (1 1* (0+ ε) 1 1* (0+ ε) 1 1*) denotes the string having at most two 0's and at least three 1's.
6. (A | B | C |……| Z | a | b | c |……| z | _ |). ((A | B | C |……| Z | a | b | c |……| z | _ |) (1 | 2 |…….| 9))* denotes the regular expression to specify the identifier like in C. [TU]

Exercise:
Given the alphabet A = {0, 1} write the regular expression for the following
1. String that either have substring 001 or 100
2. Strings where no two 1s occurs consecutively
3. String which have an odd numbers of 0s
4. String which have an odd numbers of 0s and an even numbers 1s
5. String that have at most 2 0s
6. String that at least 3 1s
7. Strings that have at most two 0s and at least three 1s

Regular Definitions

To write regular expression for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use *regular definitions*.
The regular definition is a sequence of definitions of the form,

$d1 \rightarrow r1$
$d2 \rightarrow r2$
……
$d_n \rightarrow r_n$

Where d_i is a distinct name and r_i is a regular expression over symbols in $\Sigma \cup$ {d1, d2... di-1}
Where, Σ = Basic symbol and
{d1, d2... di-1} = previously defined names.

Regular Definitions: Examples

Regular definition for specifying identifiers in a programming language like C

letter → A | B | C |……| Z | a | b | c |……| z
underscore →'_'
digit →0 | 1 | 2 |…….| 9
id → (letter | underscore).(letter | underscore | digit)*

If we are trying to write the regular expression representing identifiers without using regular definition, it will be complex.

(A | B | C |……| Z | a | b | c |……| z | _ |). ((A | B | C |……| Z | a | b | c |……| z | _ |) (1 | 2 |…….| 9))*

Exercise:

1 Write regular definition for specifying floating point number in a programming language like C

Sol^{n}: digit →0 | 1 | 2 |…….| 9
num→digit* (**.**digit^{+})

2. Write regular definitions for specifying an integer array declaration in language like C

Sol^{n}: letter → A | B | C |……| Z | a | b | c |……| z
underscore →'_'
digit → 1 | 2 |…….| 9
array → (letter | underscore).(letter | underscore | digit)* ([digit^{+}.0*])$^{+}$

Recognition of Tokens

A recognizer for a language is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise.

Recognition of token implies implementing a regular expression recognizer. That entails the implementation of Finite Automation

A finite automaton can be: deterministic (DFA) or non-deterministic (NFA)

This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.

- Deterministic – faster recognizer, but it may take more space
- Non-deterministic – slower, but it may take less space
- Deterministic automatons are widely used lexical analyzers**.**

Design of a Lexical Analyzer

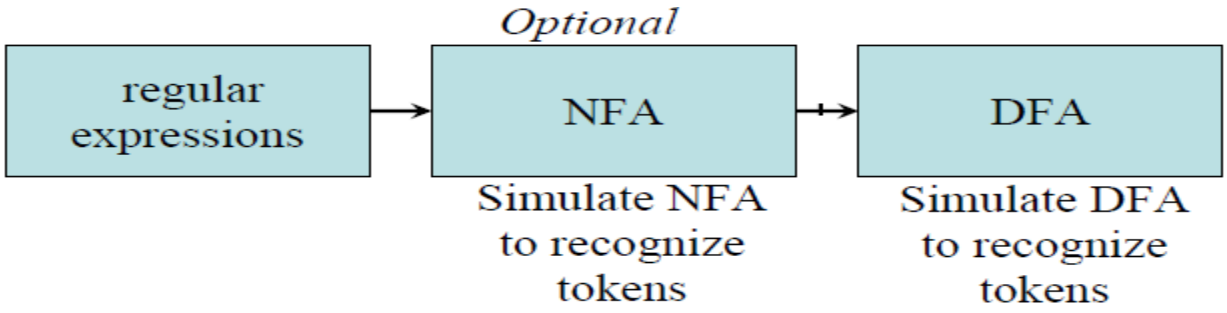
First, we define regular expressions for tokens; then we convert them into a DFA to get a lexical analyzer for our tokens.

Algorithm1:

Regular Expression → NFA → DFA (two steps: first to NFA, then to DFA)

Algorithm2:

Regular Expression → DFA (directly convert a regular expression into a DFA)



Non-Deterministic Finite Automaton (NFA)

An NFA is a 5-tuple (S, Σ, δ, $s0$, F) where

S is a finite set of *states*
Σ is a finite set of symbols
δ is a *transition function*
$s_0 \in S$ is the *start state*
$F \subseteq S$ is the set of *accepting (*or *final) states*

A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states.

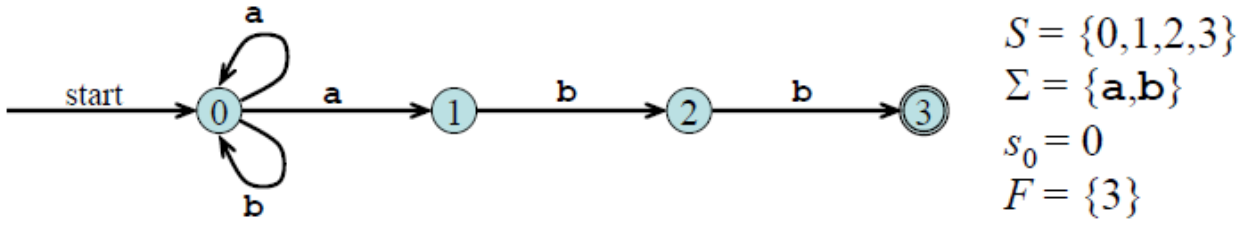


Fig: - NFA for regular expression (a + b)*a b b

ε- NFA

In NFA if a transition made without any input symbol is called ε -NFA.

Here we need ε -NFA because the regular expressions are easily convertible to ε -NFA.

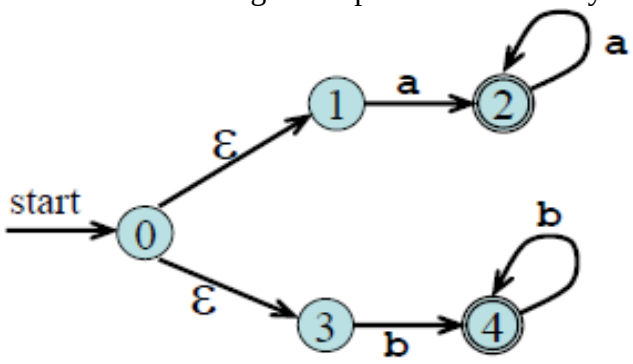


Fig: - ε -NFA for regular expression aa* +bb*

Deterministic Finite Automaton (DFA)

DFA is a special case of NFA. There is only difference between NFA and DFA is in the transition function.
In NFA transition from one state to multiple states take place while in DFA transition from one state to only one possible next state take place.

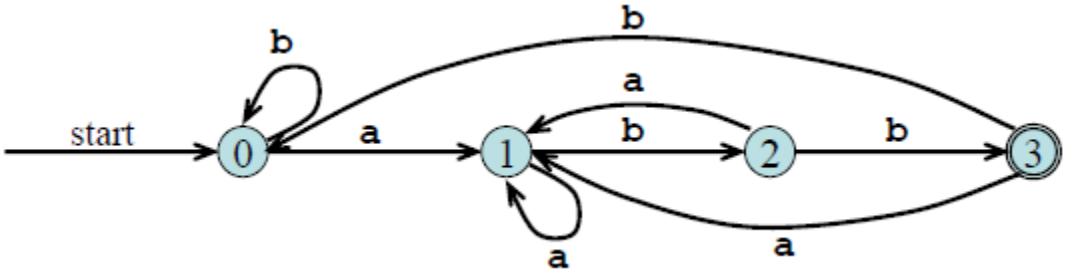


Fig:-DFA for regular expression (**a**+**b**)***abb**

Conversion: Regular Expression to NFA

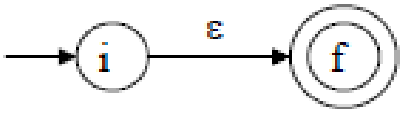
Thomson's Construction

Thomson's Construction is simple and systematic method.
It guarantees that the resulting NFA will have exactly one final state, and one start state.

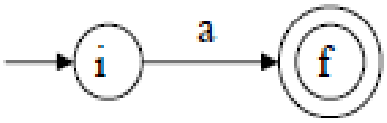
Method:

-First parse the regular expression into sub-expressions
-construct NFA's for each of the basic symbols in regular expression (r)
-Finally combine all NFA's of sub-expressions and we get required NFA of given regular expression.

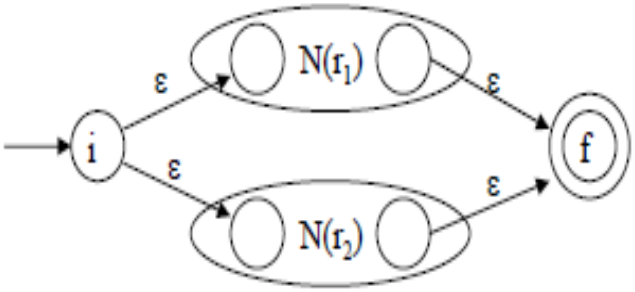
1. To recognize an empty string ε



2. To recognize a symbol a in the alphabet Σ

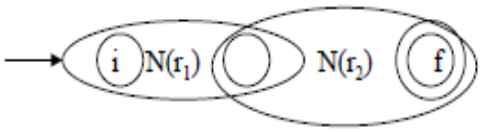


3. If N (r1) and N (r2) are NFAs for regular expressions r1 and r2
   a. For regular expression r1 + r2



$N(r_1 + r_2)$

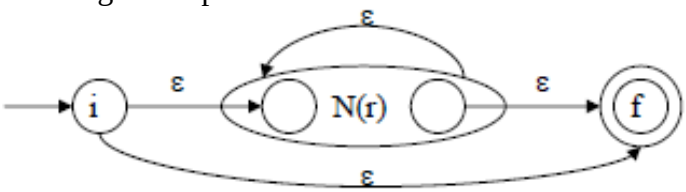
b. For regular expression r1 r2



The start state of $N(r_1)$ becomes the start state of $N(r_1 r_2)$ and final state of $N(r_2)$ become final state of $N(r_1 r_2)$

$N(r_1\ r_2)$

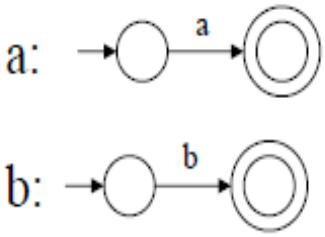
c. For regular expression r*

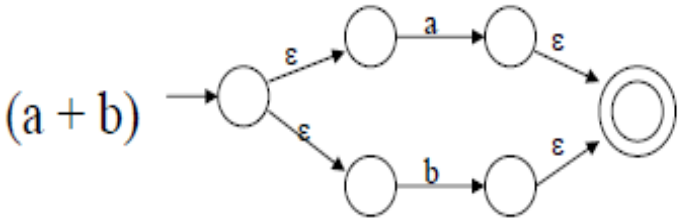


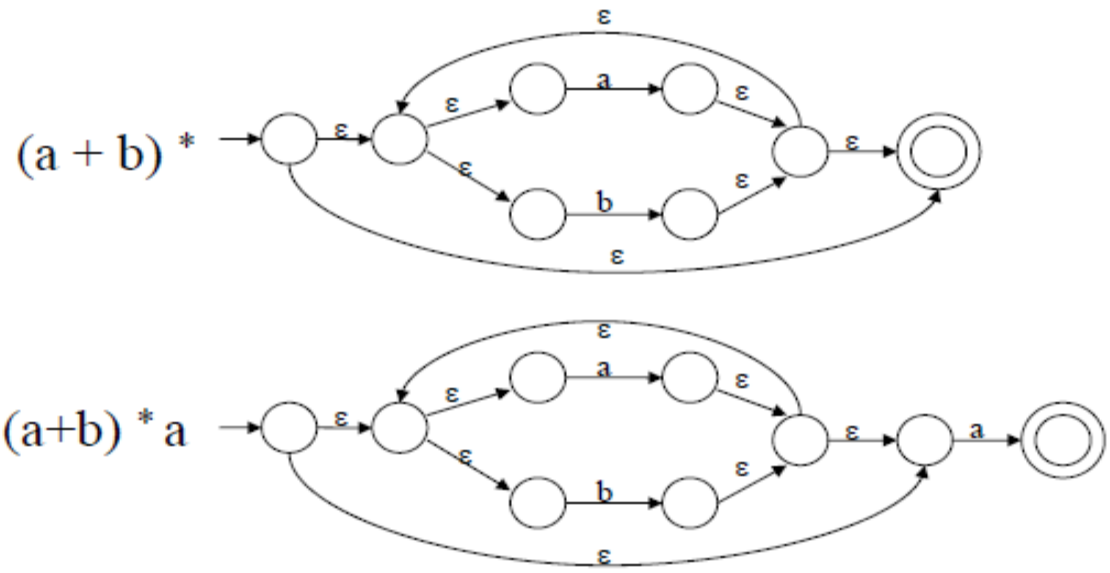
$N\ (r^*)$

Using rule 1 and 2 we construct NFA's for each basic symbol in the expression, we combine these basic NFA using rule 3 to obtain the NFA for entire expression.

Example: - NFA construction of RE **(a + b) * a**







Conversion from NFA to DFA

Subset Construction Algorithm

put ε-*closure*(s0) as an unmarked state in to *Dstates*

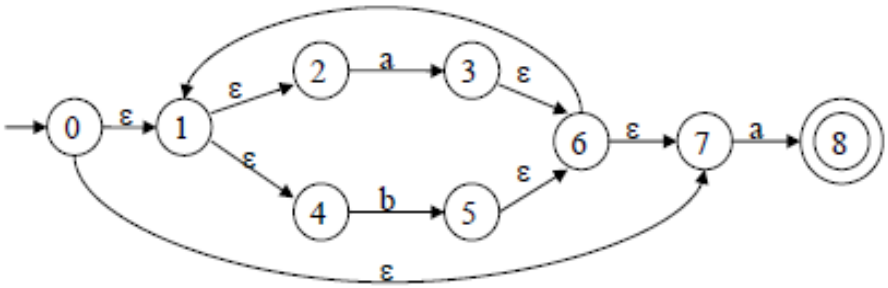
while there is an unmarked state *T* in *Dstates* **do**
mark *T*
for each input symbol *a* ∈ Σ **do**
U = ε-*closure*(*move*(*T*,*a*))
if *U* is not in *Dstates* **then**
add *U* as an unmarked state to *Dstates*
end if
Dtran[*T*,*a*] = *U*
end do
end do

The algorithm produces:

Dstates: Dstates is the set of states of the new DFA consisting of sets of states of the NFA

Dtran: Dtran is the transition table of the new DFA

Subset Construction Example (NFA to DFA)



S_0 = ε-closure({0}) = {0,1,2,4,7} S_0 into *Dstates* as an unmarked state

⇓ mark S_0

ε-closure(move(S_0,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1 S_1 into *Dstates*

ε-closure(move(S_0,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2 S_2 into *Dstates*

Dtran[S_0,a] ← S_1 *Dtran*[S_0,b] ← S_2

⇓ mark S_1

ε-closure(move(S_1,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_1,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

Dtran [S_1,a] ← S_1 *Dtran* [S_1,b] ← S_2

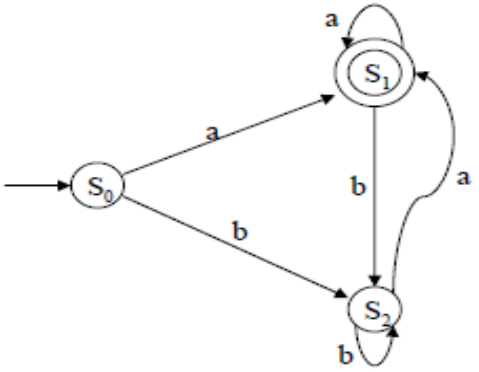
⇓ mark S_2

ε-closure(move(S_2,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_2,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

Dtran[S_2,a] ← S_1 *Dtran*[S_2,b] ← S_2

S0 is the start state of DFA since 0 is a member of S0= {0, 1, 2, 4, 7}
S1 is an accepting state of DFA since 8 is a member of S1 = {1, 2, 3, 4, 6, 7, 8}



this is final DFA

Exercise:
Convert the following regular expression first into NFA and then into DFA

1. 0+ (1+0)*00
2. zero -> 0; one ->1; bit -> zero + one; bits -> bit*

Conversion from RE to DFA Directly

Important States:

A state S of an NFA without ε- transition is called the important state if,

$$move(\{s\}, a) \neq \emptyset$$

In an optimal state machine all states are important states

Augmented Regular Expression:

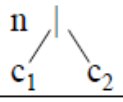
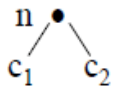
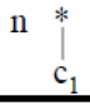
When we construct an NFA from the regular expression then the final state of resulting NFA is not an important state because it has no transition. Thus to make important state of the accepting state of NFA we introduce an 'augmented' character (#) to a regular expression r.
This resulting regular expression is called the augmented regular expression of original expression r.

Conversion steps:

1. Augment the given regular expression by concatenating it with special symbol # i.e. r ->(r) #
2. Create the syntax tree for this augmented regular expression
   In this syntax tree, all alphabet symbols (plus # and the empty string) in the augmented regular expression will be on the leaves, and all inner nodes will be the operators in that augmented regular expression.
3. Then each alphabet symbol (plus #) will be numbered (position numbers)
4. Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*
5. Finally construct the DFA from the *followpos*

To evaluate *followpos*, we need three functions to defined the nodes (not just for leaves) of the syntax tree.

Rules for calculating nullable, firstpos & lastpos

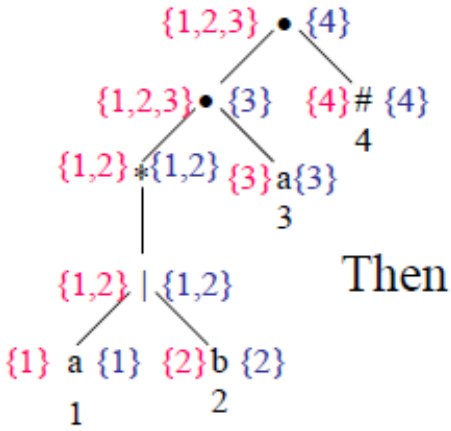
node n	nullable(n)	firstpos(n)	lastpos(n)
is leaf labeled ε	true	Φ	Φ
is leaf labeled with position i	false	{i} (position of leaf node)	{i}
	nullable(c_1) or nullable(c_2)	firstpos(c_1) ∪ firstpos(c_2)	lastpos(c_1) ∪ lastpos(c_2)
	nullable(c_1) and nullable(c_2)	**if** (nullable(c_1)) **then** firstpos(c_1) ∪ firstpos(c_2) **else** firstpos(c_1)	**if** (nullable(c_2)) **then** lastpos(c_1) ∪ lastpos(c_2) **else** lastpos(c_2)
	true	firstpos(c_1)	lastpos(c_1)

How to evaluate followpos

for each node *n* in the tree **do**
    if *n* is a cat-node with left child *c*1 and right child *c*2 **then**
        for each *i* in *lastpos*(*c*1) **do**
            followpos(*i*) := *followpos*(*i*) ∪ *firstpos*(*c*2)
        end do
    else if *n* is a star-node
        for each *i* in *lastpos*(*n*) **do**
            followpos(*i*) := *followpos*(*i*) ∪ *firstpos*(*n*)
        end do
    end if
end do

How to evaluate followpos: Example

For regular expression: **(a | b) * a #**



Then we can calculate followpos

followpos(1) = {1,2,3}
followpos(2) = {1,2,3}
followpos(3) = {4}
followpos(4) = {}

After we calculate follow positions, we are ready to create DFA for the regular expression.

Conversion from RE to DFA Example1

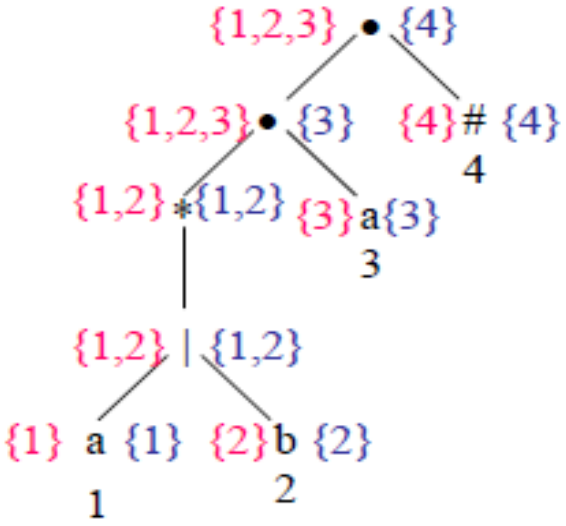
Note: - the start state of DFA is firstpos(root)
the accepting states of DFA are all states containing the position of #

For the RE --- (a | b) * a

Its augmented regular expression is;

(a | b) * a #
1 2 3 4

The syntax tree is:



Now we calculate followpos,

followpos(1)={1,2,3}

followpos(2)={1,2,3}
followpos(3)={4}
followpos(4)={}

S_1=*firstpos(root)={1,2,3}*
mark S_1
for a: followpos(1) $\cup$ *followpos(3)={1,2,3,4}*=S_2 *move*(S_1,*a*)=S_2
for b: followpos(2)={1,2,3}=S_1 *move*(S_1,*b*)=S_1
mark S_2
for a: followpos(1) $\cup$ *followpos(3)={1,2,3,4}*=S_2 *move*(S_2,*a*)=S_2
for b: followpos(2)={1,2,3}=S_1 *move*(S_2,*b*)=S_1

Now
start state: S_1
accepting states: {S_2}
Note:- Accepting states=states containing position of # ie 4.

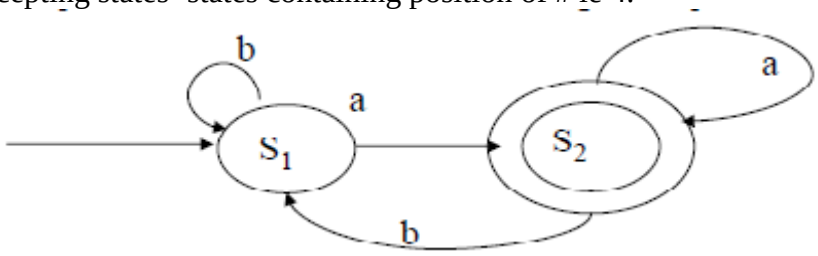


Fig: Resulting DFA of given regular expression

Conversion from RE to DFA

Example2

For RE---- (a | ε) b c* #
1 2 3 4
followpos(1)={2}
followpos(2)={3,4}
followpos(3)={3,4}
followpos(4)={}

S1=firstpos(root)={1,2}
mark S1

for a: followpos(1)={2}=S2 *move(S1,a)=S2*
for b: followpos(2)={3,4}=S3 *move(S1,b)=S3*
mark S2

for b: followpos(2)={3,4}=S3 *move(S2,b)=S3*

mark S3

for c: followpos(3)={3,4}=S3 *move(S3,c)=S3*

start state: S1
accepting states: {S3}

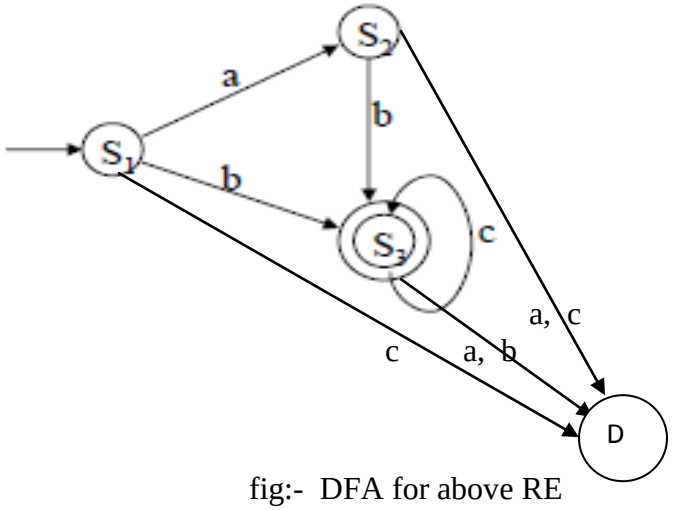


fig:- DFA for above RE