UNIT – III

Role of lexical analysis -parsing & Token, patterns and Lexemes & Lexical Errors, regular definitions for the language constructs & strings, sequences, Comments & Transition diagram for recognition of tokens, reserved words and identifiers, examples Introduction to Compilers and Interpreters: General Model of Complier, Program interpretation, Comparison of compiler and Interpreter, Use of Interpreter and components of Interpreter. Case Study: Overview of LEX and YACC specification and features.

INTRODUCTION :

Compiler is a translator program which converts the high level language into an equivalent program in another language (the object or target language).

An important part of any compiler is the detection and reporting of errors; Commonly, the source language is a high-level programming language (i.e. a problem-oriented language), and the target language is a machine language or assembly language (i.e. a machine-oriented language). Thus compilation is a fundamental concept in the production of software: it is the link between the (abstract) world of application development and the low-level world of application execution on machines.

The following section deals with the compilation procedure of any program.

Preprocessor

Preprocessing performs (usually simple) operations on the source file(s) prior to compilation.

Typical preprocessing operations include:

Expanding macros. For example, in C,

```
#define foo(x,y) (3*x+y*(2+x))
```

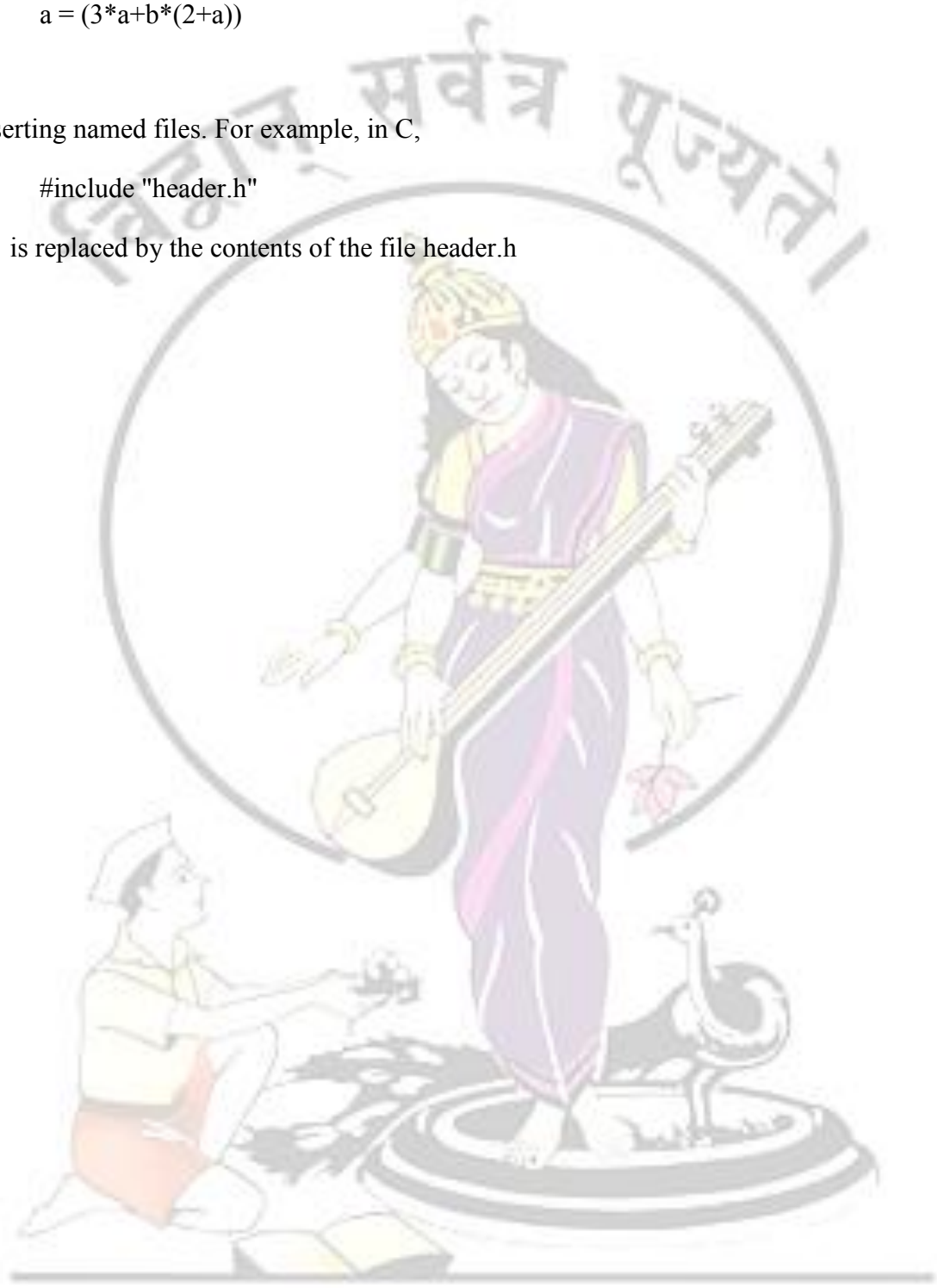
defines a macro foo, that when used in later in the program, is expanded by the preprocessor. For example, a = foo(a,b) becomes

a = (3*a+b*(2+a))

Inserting named files. For example, in C,

#include "header.h"

is replaced by the contents of the file header.h



Linker

Linker is used to link different object program to produce binary equivalent code.

Compilation process

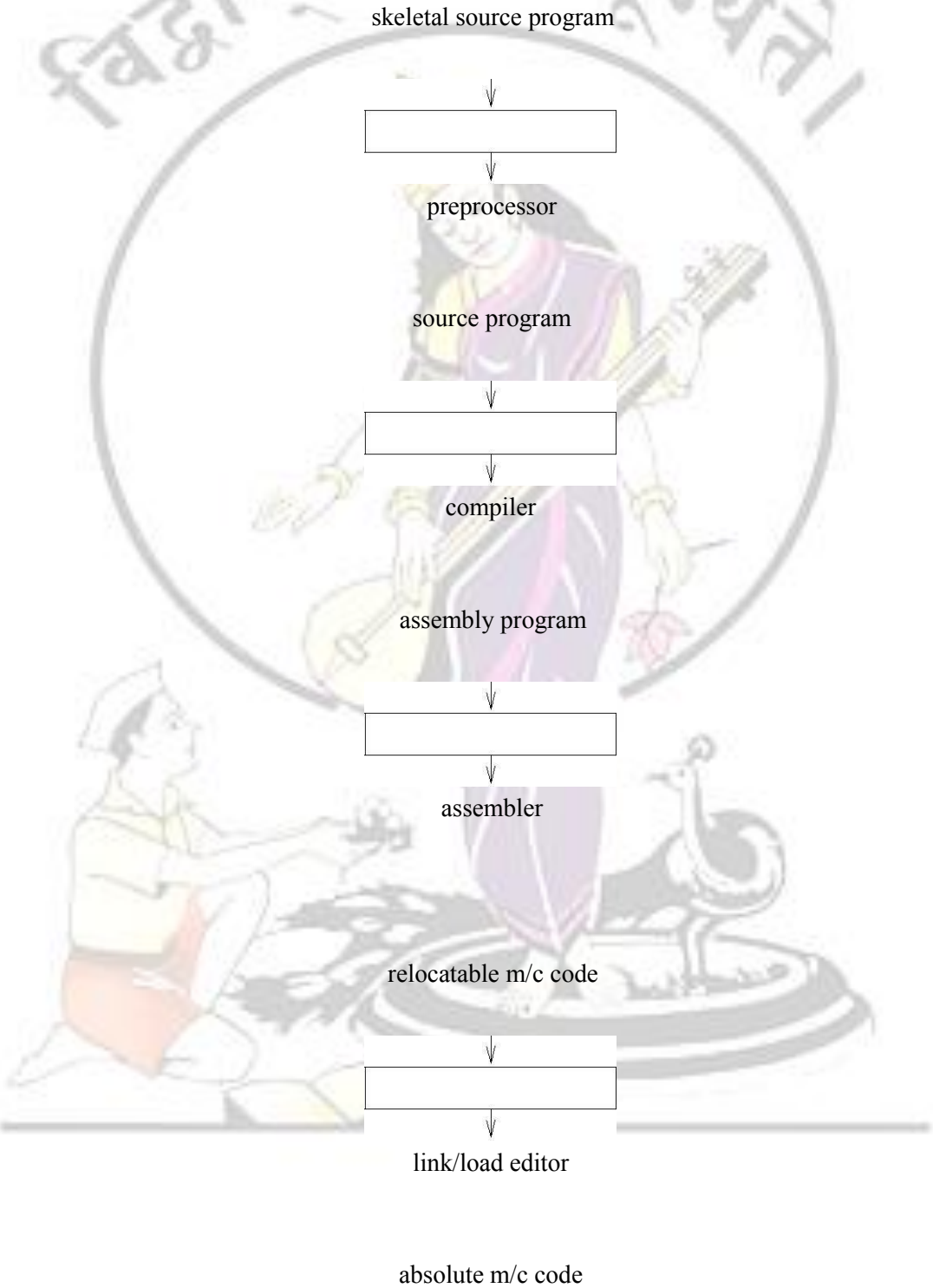


Fig 3.1 Compilation process

The above figure deals with compilation procedure of any program

Loader

Loader is used to load the program in main memory for execution

The Phases of a Compiler

The process of compilation is split up into six phases, each of which interacts with a symbol table manager and an error handler. This is called the analysis/synthesis model of compilation. There are many variants on this model, but the essential elements are the same.

The various phases of compilers are

Lexical phase or scanner

Syntax and semantic phase (or) parser

Intermediate code generation

Code optimization

Code generation

All the phases of compiler uses symbol table. Symbol table is a data structure which contains information about symbols. Apart from symbol table all the phases uses its own error recovery management mechanism. The following Figure 3.2 explains the structure of compiler.

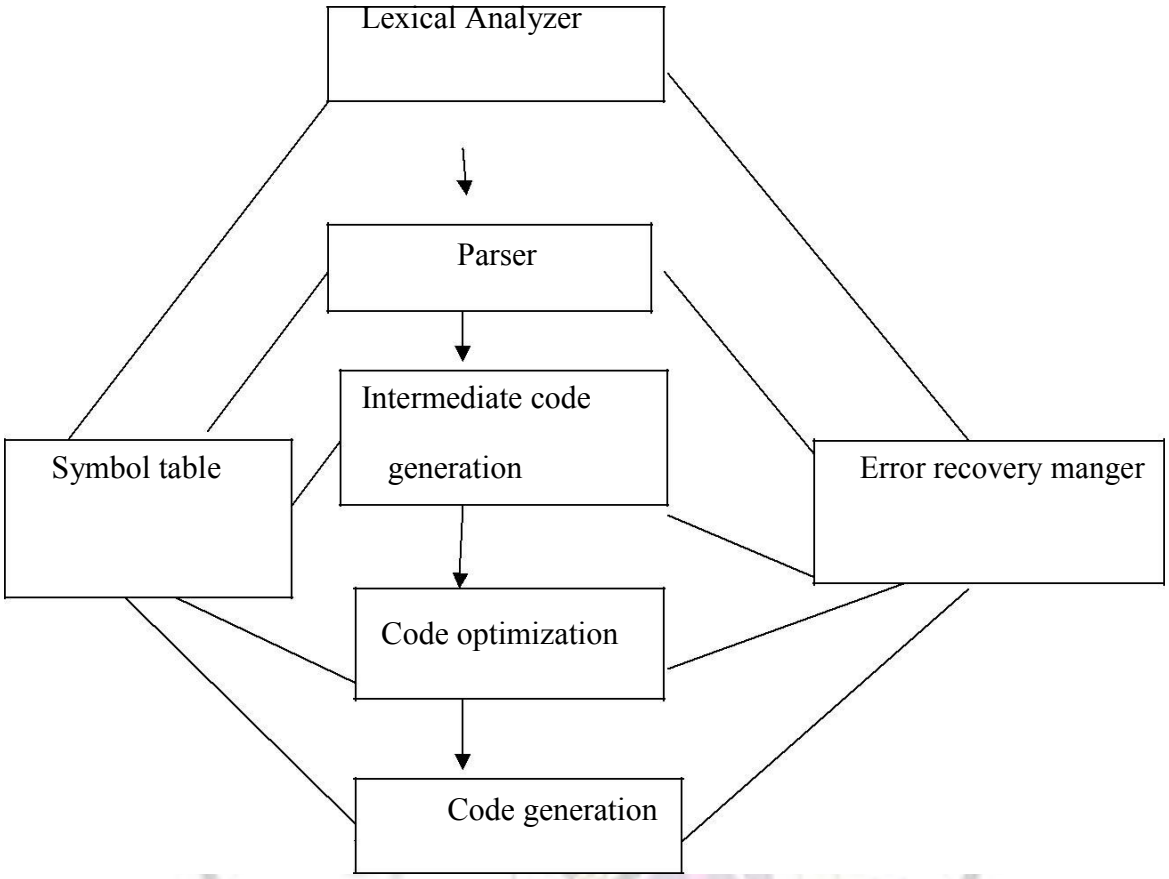


Fig 3.2 structure of compiler

Lexical Analysis

A lexical analyser or scanner is a program that groups sequences of characters into lexemes, and outputs (to the syntax analyser) a sequence of tokens. Here:

(a) Tokens are symbolic names for the entities that make up the text of the program; e.g. if for the keyword if, and id for any identifier. These make up the output of the lexical analyser.

(b) A pattern is a rule that specifies when a sequence of characters from the input constitutes a token; e.g the sequence i, f for the token if , and any sequence of alpha numerics starting with a letter for the token id.

(c) A lexeme is a sequence of characters from the input that match a pattern (and hence constitute an instance of a token); for example if matches the pattern for if , and foo123bar matches the pattern for id.

Consider the following example

program foo(input,output);var x:integer;begin

Lexeme	Token	Pattern
program	program	p, r, o, g, r, a, m
		newlines, spaces, tabs
foo	id (foo)	letter followed by seq. of alphanumerics
(	lef tpar	a left parenthesis
input	input	i, n, p, u, t
,	comma	a comma
output	output	o, u, t, p, u, t
)	rightpar	a right parenthesis
;	semicolon	a semi-colon
var	var	v, a, r


विद्वान् सर्वत्र पूज्यते।

x	id (x)	letter followed by seq. of alphanumerics
:	colon	a colon
integer	integer	i, n, t, e, g, e, r
;	semicolon	a semi-colon
begin	begin	b, e, g, i, n

Lexeme

Lexeme is a sequence of characters which matches the input

Token

The various components used in the given program are called as tokens. The various tokens are

Keywords

Identifier

Operators

Literals

Punctuations

Special symbols

numbers

Pattern

Pattern is a rule for forming token

Consider the following input

a+b*c+60

The output of lexical analysis

a is an identifier

2. is an operator

b is an identifier

*is an operator

3. is an operator

60 is a number

During scanning if there is any identifier is encountered then the symbol is entered in to a symbol table along with its various attributes. Other values are entered in to a table during other phases of compiler.

Symbol table

Symbol name	Type	Address	value
Id1	int		
Id2	Int		
Id3	int		

Parser

Parser is a program which takes an input string and produces the output as parse tree if the string belongs to given grammar, otherwise it produces error.

The grammar for the above input string is denoted as

E->E+E |E-E|E*E|(E)|-E|id

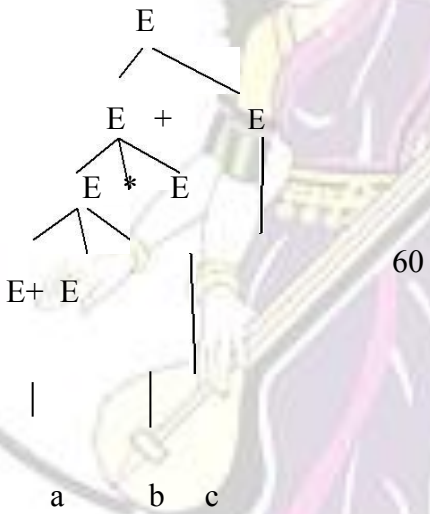


Fig 3.

Intermediate code generation

It breaks the source code in to an intermediate code. One form of intermediate code is three address code.

The output is

T=b*c

T1=t+60

T2=a+t1

T3=t2

Code optimization

The various code optimization strategy are applied to produce an efficient object code

Code generation

Some compiler produces assembly code, while some other compiler produces straightway an object code .

```
MOV b,Ro

MOV c,R1

MUL Ro,R1

ADD Ro,#60

MOV a,R1

ADD Ro,R1
```

Functions of lexical phase

1. Separate tokens

2. Deletion of comment lines

3. Keeping track of line numbers

4. Produce the output listing if necessary

Once the tokens are generated these tokens are given to parser to produce parse tree.

Symbol Table

A symbol table is a data structure containing all the identifiers (i.e. names of variables, procedures etc.) of a source program together with all the attributes of each identifier.

For variables, typical attributes include:

- its type,
- how much memory it occupies,
- its scope.

For procedures and functions, typical attributes include:

(a) the number and type of each argument (if any),
(b) the method of passing each argument, and
(c) the type of value returned (if any).

The purpose of the symbol table is to provide quick and uniform access to identifier attributes throughout the compilation process. Information is usually put into the symbol table during the lexical analysis and/or syntax analysis phases.

Role of lexical Analyzer:

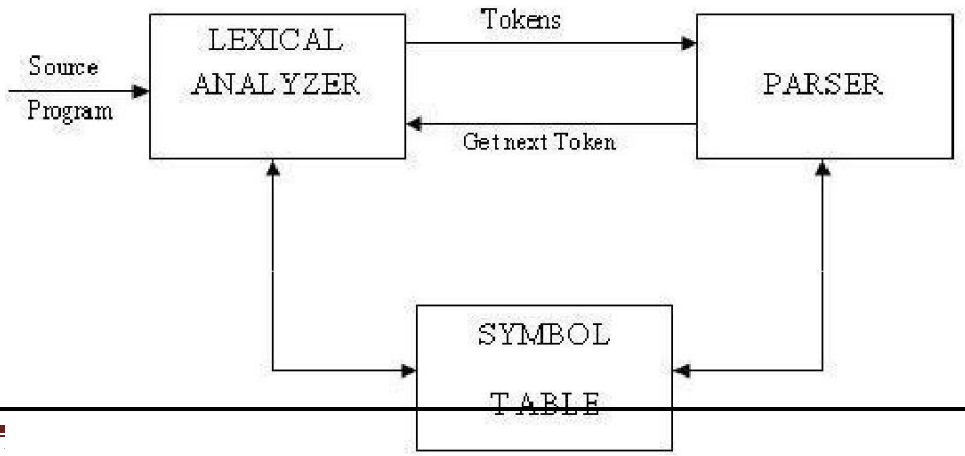


Fig 3.4 Role of lexical analyzer

The scanner is the first phase of a compiler. The main task is to read the input character

and produce as output a sequence of tokens that the parser uses for syntax analysis. Upon receiving a get next token‟ command form the parser, the lexical analyzer reads the input character until it can identify the next token. The Lexical analyzer (LA) return to the parser representation for the token it has found. The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon. LA may also perform certain secondary tasks as the user interface. One such task is striping out from the source program the commands and white spaces in the form of blank, tab and new line characters. Another is correlating error message from the compiler with the source program.

LEXICAL ERRORS:

Lexical errors are the errors thrown by the lexer when unable to continue, which means that there's no way to recognise a *LEXEME* as a valid *TOKEN* for the lexer. Syntax errors, on the other side, will be thrown by the scanner when a given set of already recognised valid tokens don't match any of the right sides of the grammar rules. Simple panic-mode error handling system requires that we return to a high-level parsing function when a parsing or lexical error is detected

Error-recovery actions are:

i. Delete one character from the remaining input.

ii. Insert a missing character in to the remaining input.

iii. Replace a character by another character.

iv. Transpose two adjacent characters

Strings,languages and operations of languages

String is a collection of characters.

Eg: set of names

Language is a collection of strings

Eg: L={abc,xyz}

The above language describes two strings, they are abc and xyz

Operations of languages

A language L is said to be regular, if it is closed under union,concatenation and kleene closure.

Union:

L={a|b}

The above language L represents either a or b

Concatenation:

L1={abc}

L2={xyz}

Set of strings both are in L1 and L2

Then the concatenation of two languages represented by L1.L2.

After concatenation the language becomes abcxyz.

Closure:

There are two types of closure operation

1.Kleene closure

a* (zero or more occurrences of a)

a*= {^, a,aa,aaaa, aaaaa……}

2.Positive closure

a+ (one or more occurrences of a)

a+ = {a,aa,aaa,aaaa,aaaaaa…….}

Regular expression

Regular expression is used to describe the structure of tokens.

Formal definition

1.^ is said to be a regular expression

2.Φ is said to be a regular expression

3.a in ∑ is said to be regular expression.

4. Let L1 be the language for representing regular expression r1. Let L2 be the language for representing the regular expression r2. Then r1|r2 is said to be a regular expression.

5. Let L1 be the language for representing regular expression r1. Let L2 be the language for representing the regular expression r2. Then r1.r2 is said to be a regular expression.

6. Let L1 be the language for representing regular expression r1. Then r1* is said to be a regular expression.

The regular expression is made up of only by using above definitions.

Properties of regular expression

Let R,S and T are regular expressions for the respective languages. Then the regular expression follows the various properties

R|S=S|R (Commutative property)

R(S|T)=R|S(T) (Associative property)

R(ST)=RS(T) (Associative property)

R(S|T)=RS|RT (Distributive property)

R.^=^.R=R (Identity)

REGULAR DEFINITIONS

For notational convenience, we may wish to give names to regular expressions and

to define regular expressions using these names as if they were symbols. Identifiers are the set or string of letters and digits beginning with a letter. The following regular definition provides a precise specification for this class of string.

Recognition of tokens:

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.

Stmt if expr then stmt

| If expr then else stmt

| ϵ

Expr term relop term

|term

Term id

|number

Examples of regular definitions

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 --></>/<=/>=/==/< >

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the "token" we defined by:

ws blank/tab/newline)

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 .

Implementation of lexical analyzer using transition diagram:

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 labeled 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.

Components of transition diagrams

Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions between the lexeme Begin and forward pointers we always indicate an accepting state by a double circle. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a * near that accepting state. One state is designated as the start state or initial state. It is indicated by an edge labeled "start" entering from nowhere.

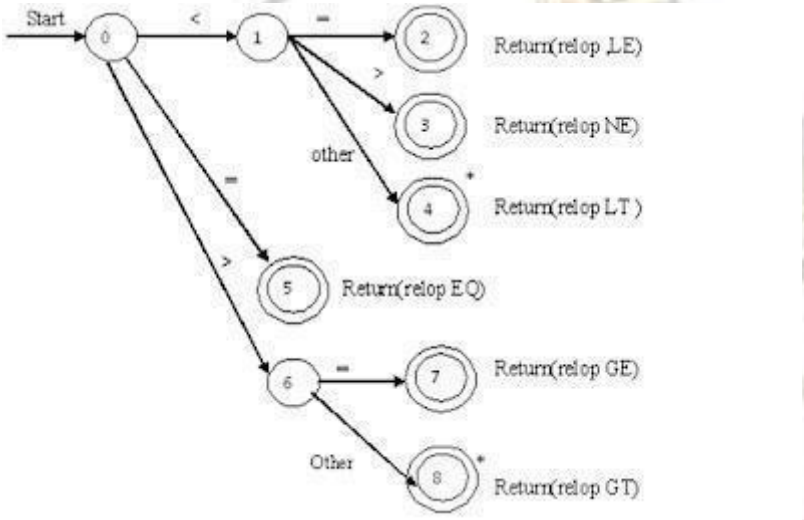


Fig 3.5 Transition diagram for Keywords

Consider the following transition diagram. The transition diagrams shows how the lexical analyzer recognizes token of type id. Once id is encountered, it is entered in to a symbol table.

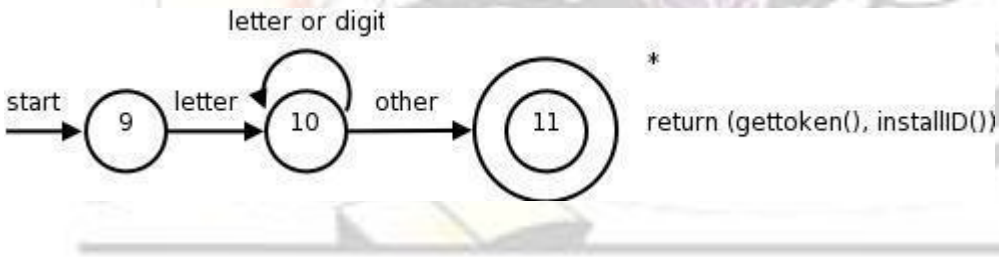


Fig 3.6 Transition diagram for an identifier


विद्वान् सर्वत्र पूज्यते।

state 9: c=getchar();

if letter (c),then go to state (10)

else fail();

state 10: if letter (c) or digit (c) then go to state (10)

else if delimiter (c) then go to state (11)

else

fail();

state 11: return (id, INSTALL());

Install() is a procedure which is used to enter the symbol in to a symbol table.

Te above transition diagram for an identifier, defined to be a letter followed by any no of letters or digits. A sequence of transition diagram can be converted into program to look for the tokens specified by the diagrams. Each state gets a segment of code. Id = letter (letter | digit) *

Num = digit +

Recognizer

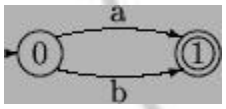
Recognizer is a program for a Language L, which takes an input string w, and says answer "yes" if w∑ G, otherwise it says no. If the string is accepted by the given grammar G then it is valid token, otherwise it is invalid.

For recognizing token the concept of finite automata is used.

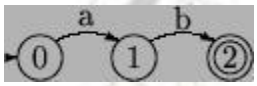
Finite automata

There are two types of finite automata

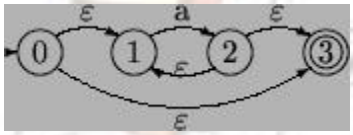
1. Non deterministic finite automata (NFA)
2. Deterministic finite automata (DFA)



(a) Concatenation simply inv olves connecting one NFA to the other; eg. AB is:



(b) The Kleene closure must allow for taking zero or more instances of the l etter from the input; thus A* looks like:



Conversion of NFA to DFA

Co nsider the following regular expression (a|b)*ab b#

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

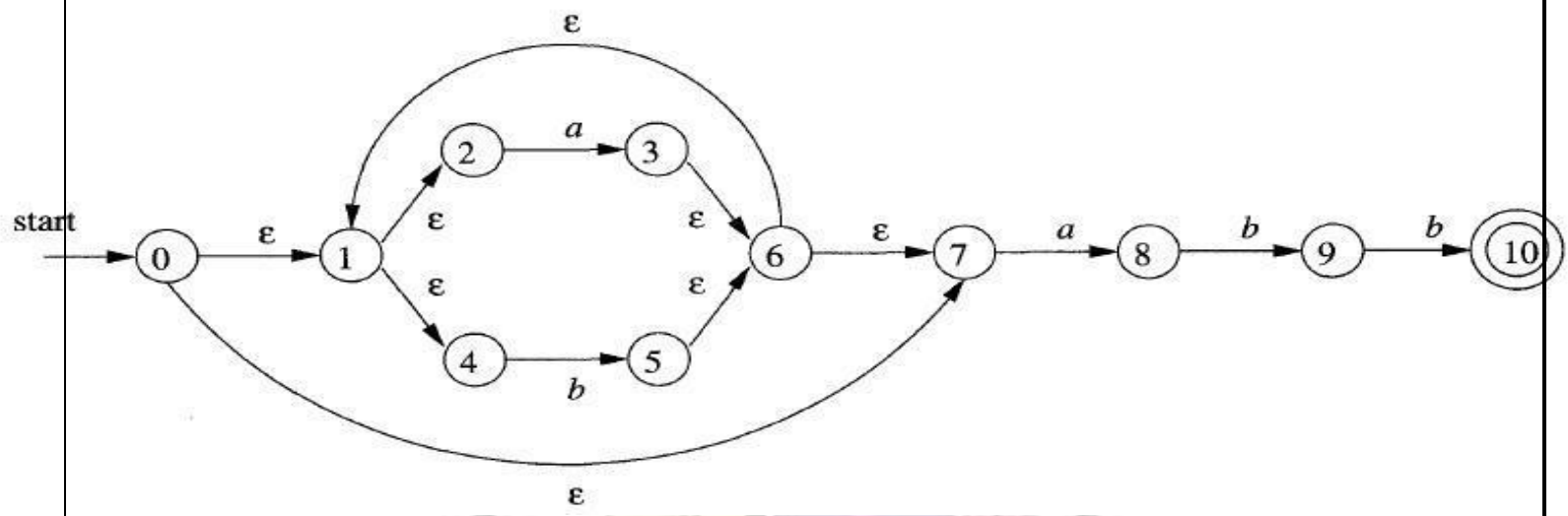


Fig 3.7 NFA for (a|b)*abb#

€-closure(s):

It is the set of € transitions starting from the given state s and reachable from s.

Computation of €-closure (0) = {0,1,2,4,7} ----------(A)

There are only two input symbols a,b for the above example

Find the transition on input symbol a when we are in state A

∂[A,a]={3,8}

Find the transition on input symbol b when we are in state A

∂[A,b]={5}

compute €-closure(3,8)={1,2,3,4,6,7,8}----------(B)

compute €-closure(5)={1,2,4,5,6,7}----------(C)

∂[B,a]={3,8}

∂[B,b]={5,9}

Then compute €-closure(5,9)= {1,2,4,5,6,7,9}----------(D)

∂[C,a]= {3,8}

∂[C,b]={5}

∂[D,a]={3,8}

∂[D,b]={5,10}

Then compute €-closure(5,10) ={1,2,4,5,6,7,10}----------(E)

∂[E,a] ={3,8}

∂[E,b]={5}

The above procedure is repeated until no new state can be found

Step 2: Convert the above NFA in to DFA.

Transition table

Input Symbol

STATES	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

The above transition table represented by the Fig 3.8. Here state 'E' is the accepting state for DFA.

NFA STATE	DFA STATE	*a*	*b*
{0, 1, 2, 4, 7}	*A*	*B*	*C*
{1, 2, 3, 4, 6, 7, 8}	*B*	*B*	*D*
{1, 2, 4, 5, 6, 7}	*C*	*B*	*C*
{1, 2, 4, 5, 6, 7, 9}	*D*	*B*	*E*
{1, 2, 3, 5, 6, 7, 10}	*E*	*B*	*C*

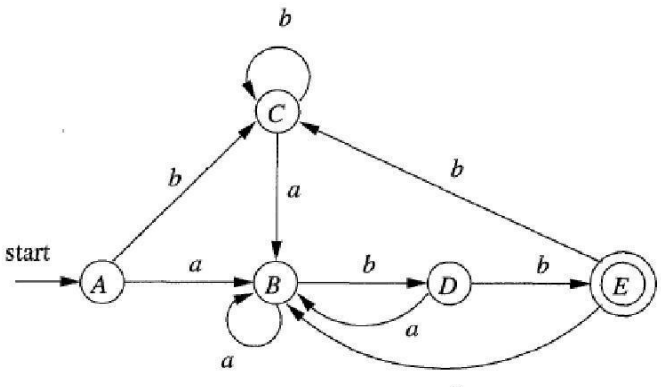


Fig 3.8 DFA for (a|b)*abb#

Fig 3.9 Minimized DFA

Step 3: Convert the above DFA in to minimized DFA by applying the following algorithm.

Minimized DFA algorithm

Input: DFA with 's' no of states

Output: Minimized DFA with reduced no of states.

Steps:

1. Partition the set of states in to two groups. They are set of accepting states and non accepting states.

2. For each group G of π do the following steps until $\pi=\pi_{new}$.

begin

divide G in to as many groups as possible, such that two states s and t are in the same group only when for all states s and t have transitions for all input symbols 's' are in the same group itself. Place newly formed group in π_{new}.

end

3. Choose representative state for each group.

4. Remove any dead state from the group.

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

Input Symbol

STATES	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Exercises

Convert the following regular expression in to minimized state DFA

(f) (a|b)*
(g)(b|a)*abb(b|a)*
(h) ((a|c)*)ac(ba)*

Context free grammar

Grammar is a set of rules and regulations which is used to define the syntax of any programming language.

Advantages of grammar

1.to define the syntax of programming language

2.efficient parser can be constructed with properly designed grammar

3. which imparts the structure of program useful for detecting the error

There are different types of grammar; one such grammar is called as context free grammar (CFG). It is otherwise called as BNF. (Back us Naur Form)

Definition of context free grammar

The grammar G is defined as which consists of four tubles i.e G=(V,T,S,P)

Where V is a non terminal,

T is a terminal,

S is a starting symbol of the given grammar, S ∑ V

P is a set of productions or rules,such that every production is of the form

P:A->α | α∑ (VUT)*

Non-terminal:

Set of symbols which is used to define further

e.g statement, expression, operators etc, because statement, expression can be defined further.

Generally all upper case symbol in the given grammar are treated as non terminal.

Terminal:

All tokens are called as tokens. Generally all lower case symbol in the given grammar are treated as terminal.

e.g identifiers, literals, numbers, punctuations etc.

Productions:

Set of rules which is used to define the grammar

e.g S-> if (condn) then S

|if (condn) then S else S

The above grammar consists of two productions.

Any sentence can be derived from starting symbol of the grammar only.

Derivation

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

Deciding the non-terminal which is to be replaced.

Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options.

Left sentential form:

While deriving some string w using left most derivation, for each step combination of terminals and non terminals derived. The combination of terminals and non terminals is called as left sentential form.

Right sentential form

While deriving some string w using left most derivation, for each step combination of terminals and non terminals derived. The combination of terminals and non terminals is called as left sentential form.

Left-most Derivation (LMD)

If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

Right-most Derivation (RMD)

If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

Consider the following grammar which is used to evaluate arithmetic expression

E->E+E|E-E|E*E|E/E|(E)|-E|id

Derive the following sentence using LMD and RMD

id+id*id

E -> E+E

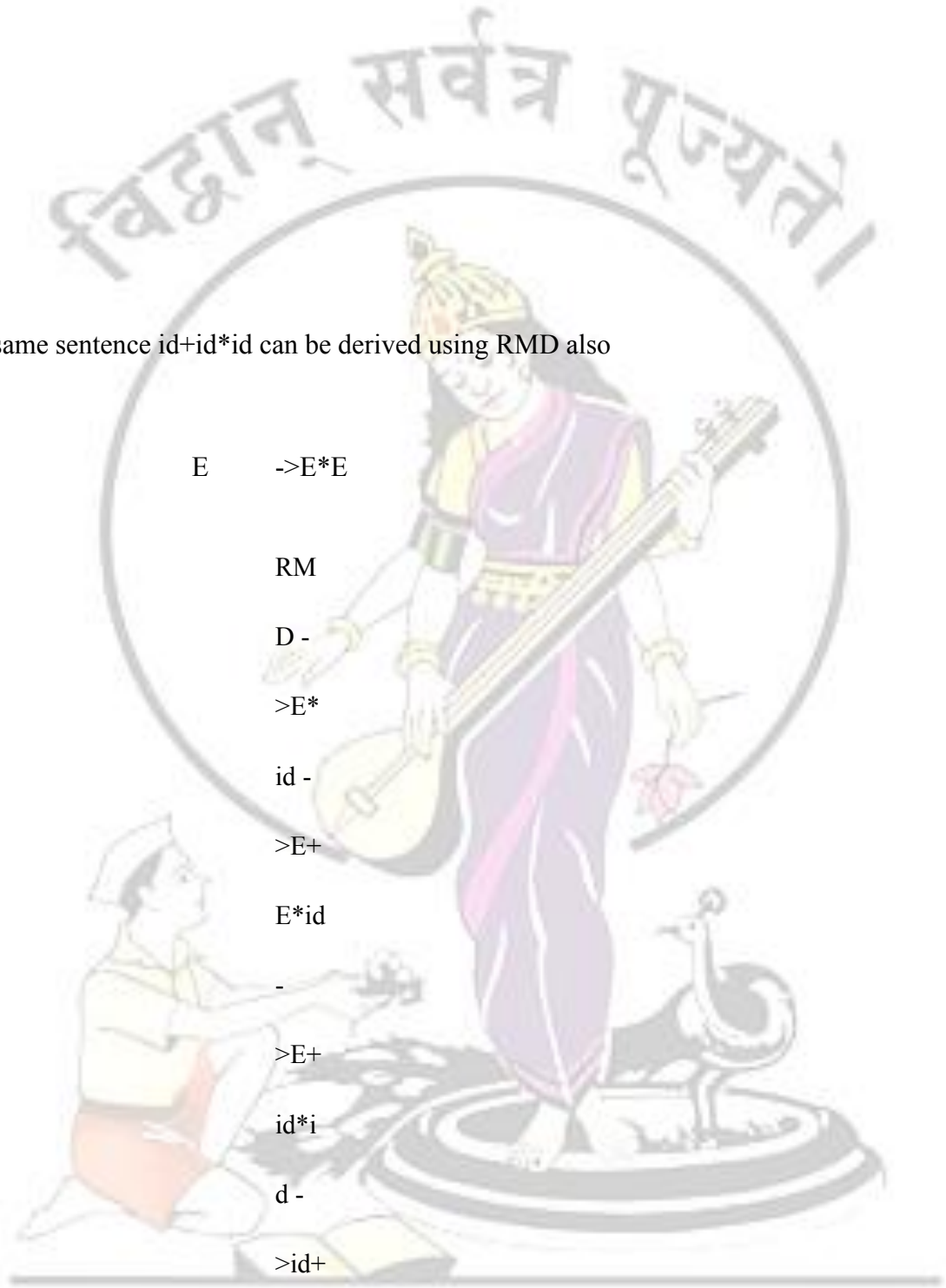
LMD

->id+E

->id+E*E

->id+id*E

->id+id*id



The same sentence id+id*id can be derived using RMD also

E ->E*E

RM

D ->E*

id ->E+

E*id

->E+

id*i

d ->id+

id*i

d

Ambiguous grammar

The grammar G is said to be ambiguous, if there is a more than one parse tree for deriving the same sentence w. The above grammar is said to be an ambiguous grammar. But before parsing the grammar should be converted in to unambiguous grammar.

Unambiguous grammar

The grammar G is said to be unambiguous, if there is an at most one parse tree for deriving the same sentence w.

e.g E->E+T|T

T->T*F|F

F->(E)|a is an example of unambiguous grammar.

Context free grammar versus regular expression

Consider the regular expression (a|b)*abb, and the corresponding CFG is

A0-> aA0|bA0|aA1

A1->bA2

A2->bA3

A3->€

We can construct mechanically a grammar to recognize the same language as non deterministic finite automata. The grammar above was constructed from the NFA using the following procedure.

(a) For each state i of the NFA ,create a non terminal Ai
(b) If state i has a transition to state j on input a , add the production Ai->aAj. If state i goes to state j on input € ,add the production Ai->Aj
(c) If i is an accepting state ,add Ai->€
(d) If I is the start state ,make Ai be the start symbol of the grammar

Capabilities of context free grammar over regular expression

1. Separating the syntactic structure of a language in to lexical and non lexical parts provides a convenient way of modularizing the front end of a compiler into two manageable sized components

3. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammar.Regular expressions generally provides a more concise and easier to understand notation for tokens than grammar

4. More efficient lexical analyzer can be constructed automatically from regular expressions than from arbitrary grammar.