INTRODUCTION TO COMPILER

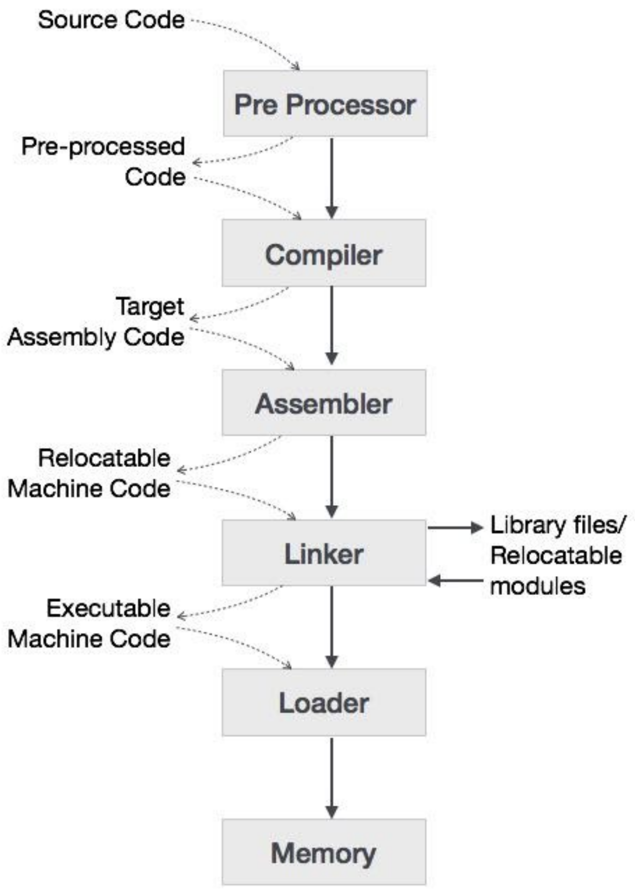
Compiler

A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.

Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end.

Language Processing System

We have learnt that any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.



The high-level language is converted into binary language in various phases. A **compiler** is a program that converts high-level language to assembly language. Similarly, an **assembler** is a program that converts the assembly language to machine-level language.

Let us first understand how a program, using C compiler, is executed on a host machine.

- User writes a program in C language (high-level language).
- The C compiler, compiles the program and translates it to assembly program (low-level language).
- An assembler then translates the assembly program into machine code (object).
- A linker tool is used to link all the parts of the program together for execution (executable machine code).
- A loader loads all of them into memory and then the program is executed.

Before diving straight into the concepts of compilers, we should understand a few other tools that work closely with compilers.

Preprocessor

A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

Interpreter

An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it. whereas a compiler reads the whole program even if it encounters several errors.

Assembler

An assembler translates assembly language programs into machine code.The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

Linker

Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

Loader

Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

Cross-compiler

A compiler that runs on platform (A) and is capable of generating executable code for platform (B) is called a cross-compiler.

Source-to-source Compiler

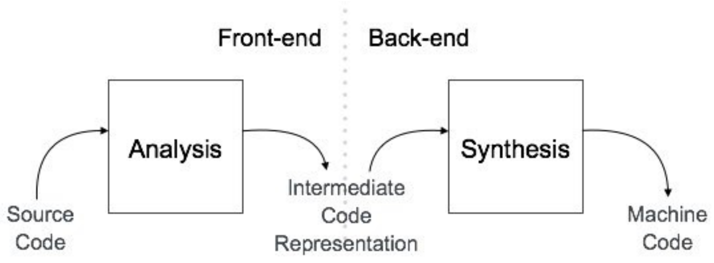
A compiler that takes the source code of one programming language and translates it into the source code of another programming language is called a source-to-source compiler.

Phases of Compiler

A compiler can broadly be divided into two phases based on the way they compile.

Analysis Phase

Known as the front-end of the compiler, the **analysis** phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors.The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.



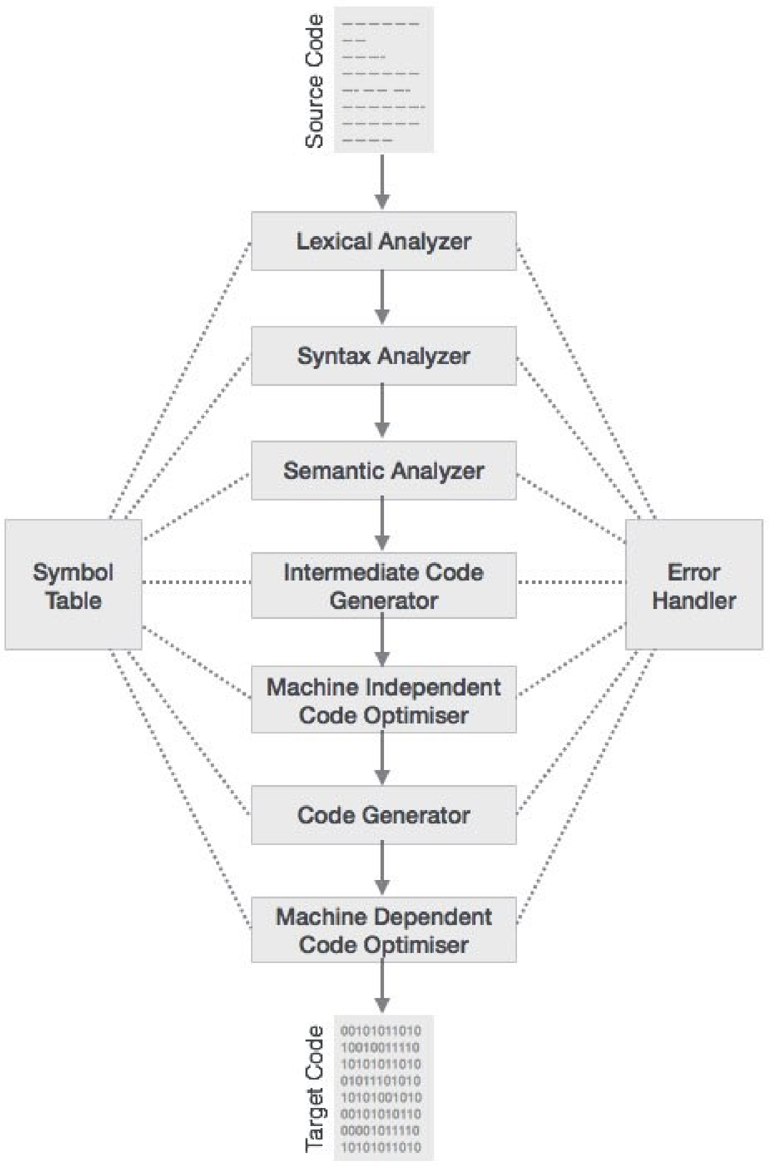
Synthesis Phase

Known as the back-end of the compiler, the **synthesis** phase generates the target program with the help of intermediate source code representation and symbol table.

A compiler can have many phases and passes.

- **Pass** : A pass refers to the traversal of a compiler through the entire program.
- **Phase** : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.



Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

```
<token-name, attribute-value>
```

Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

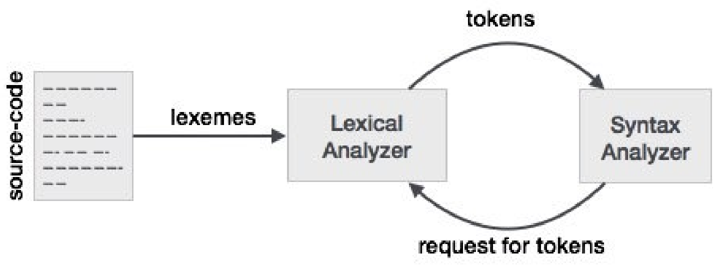
Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

Lexical Analysis-Theory

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Tokens

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration linecontains the tokens:

```
int value =100;
int(keyword), value (identifier),=(operator),100(constant)and;(symbol).
```

Specifications of Tokens

Let us understand how the language theory undertakes the following terms:

Alphabets

Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorialspoint is 14 and is denoted by |tutorialspoint| = 14. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

Special Symbols

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=
Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifier	&
Logical	&, &&, |, ||, !
Shift Operator	>>, >>>, <<, <<<

Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

Longest Match Rule

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

For example:

```
        int intvalue;
```

While scanning both lexemes till 'int', the lexical analyzer cannot determine whether it is a keyword *int* or the initials of identifier int value.

The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

The lexical analyzer also follows **rule priority** where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

Regular Expression

The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules.

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as **regular grammar**. The language defined by regular grammar is known as **regular language**.

Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages. The specification of regular expressions is an example of a recursive definition. Regular languages are easy to understand and have efficient implementation.

There are a number of algebraic laws that are obeyed by regular expressions, which can be used to manipulate regular expressions into equivalent forms.

Operations

The various operations on languages are:

- Union of two languages L and M is written as

  L U M = {s | s is in L or s is in M}

- Concatenation of two languages L and M is written as

  LM = {st | s is in L and t is in M}

- The Kleene Closure of a language L is written as

  L* = Zero or more occurrence of language L.

Notations

If r and s are regular expressions denoting the languages L(r) and L(s), then

- **Union** : (r)|(s) is a regular expression denoting L(r) U L(s)
- **Concatenation** : (r)(s) is a regular expression denoting L(r)L(s)
- **Kleene closure** : (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denoting L(r)

Precedence and Associativity

- *, concatenation (.), and | (pipe sign) are left associative

- * has the highest precedence
- Concatenation (.) has the second highest precedence.
- | (pipe sign) has the lowest precedence of all.

Representing valid tokens of a language in regular expression

If x is a regular expression, then:

- x* means zero or more occurrence of x.

  i.e., it can generate { e, x, xx, xxx, xxxx, … }

- x+ means one or more occurrence of x.

  i.e., it can generate { x, xx, xxx, xxxx … } or x.x*

- x? means at most one occurrence of x

  i.e., it can generate either {x} or {e}.

  [a-z] is all lower-case alphabets of English language.

  [A-Z] is all upper-case alphabets of English language.

  [0-9] is all natural digits used in mathematics.

Representing occurrence of symbols using regular expressions

letter = [a – z] or [A – Z]

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]

sign = [+ | -]

Representing language tokens using regular expressions

Decimal = $(\text{sign})^?(\text{digit})^+$

Identifier = (letter)(letter | digit)*

The only problem left with the lexical analyzer is how to verify the validity of a regular expression used in specifying the patterns of keywords of a language. A well-accepted solution is to use finite automata for verification.

Finite automata

Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly. Finite automata is a **recognizer** for regular expressions. When a regular expression string is fed into finite automata, it changes its state for each literal. If the input string is successfully processed and the automata reaches its final state, it is accepted, i.e., the string just fed was said to be a valid token of the language in hand.

The mathematical model of finite automata consists of:

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

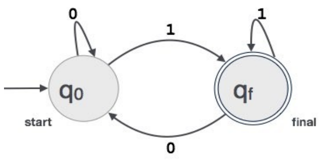
The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), Q × Σ ➔ Q

Finite Automata Construction

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 starts, 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 automata 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 points to the destination state. If automata stays on the same state, an arrow pointing from a state to itself is drawn.

Example :We assume FA accepts any three digit binary value ending in digit 1. FA = {Q(q_0, q_f), Σ(0,1), q_0, q_f, δ}



Introduction To LEX

Before 1975 writing a compiler was a very time-consuming process. Then Lesk [1975] and Johnson [1975] published papers on **lex and yacc**. These utilities greatly simplify compiler writing. Implementation details for lex and yacc may be found in Aho [2006]. Flex and bison, clones for lex and yacc, can be obtained for free from GNU and Cygwin.

Cygwin is a 32-bit Windows ports of the GNU software. In fact Cygwin is a port of the Unix operating system to Windows and comes with compilers gcc and g++. To install simply download and run the setup executable. Under devel install bison, flex, gcc-g++, gdb, and make. Under editors install vim. Lately I've been using flex and bison under the Cygwin environment.

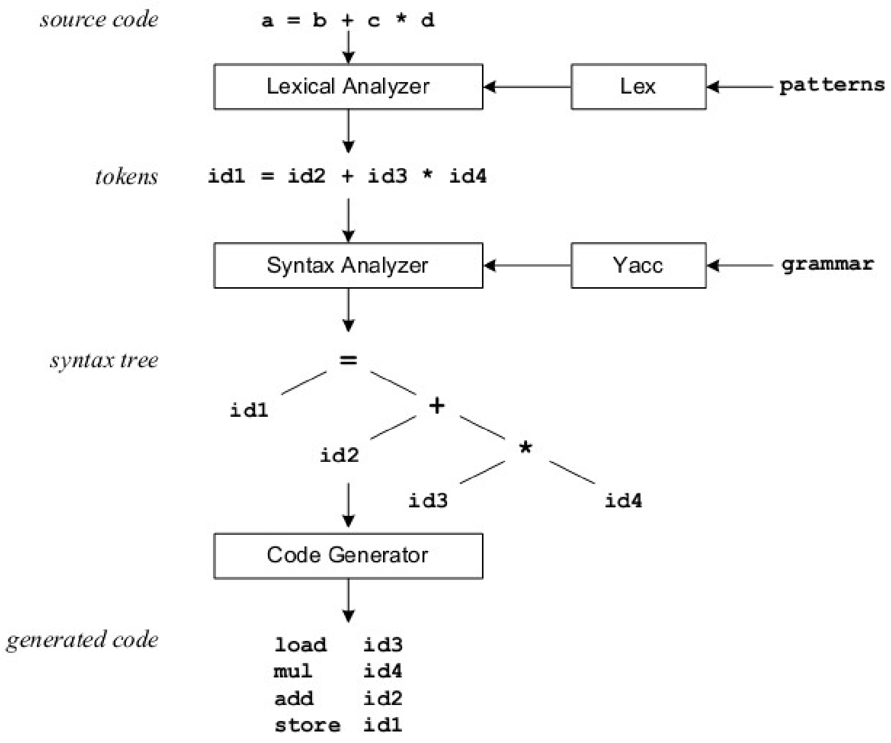


Figure 1: Compilation Sequence

The patterns in the above diagram is a file you create with a text editor. Lex will read your patterns and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on your patterns, and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing.

When the lexical analyzer finds identifiers in the input stream it enters them in a symbol table. The symbol table may also contain other information such as data type (integer or real) and location of each variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index.

The grammar in the above diagram is a text file you create with a text edtior. Yacc will read your grammar and generate C code for a syntax analyzer or parser. The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree. The syntax tree imposes a hierarchical structure the tokens. For example, operator precedence and associativity are apparent in the syntax tree. The next step, code generation, does a depth-firstwalk of the syntax tree to generate code. Some compilers produce machine code, while others, as shown above, output assembly language.

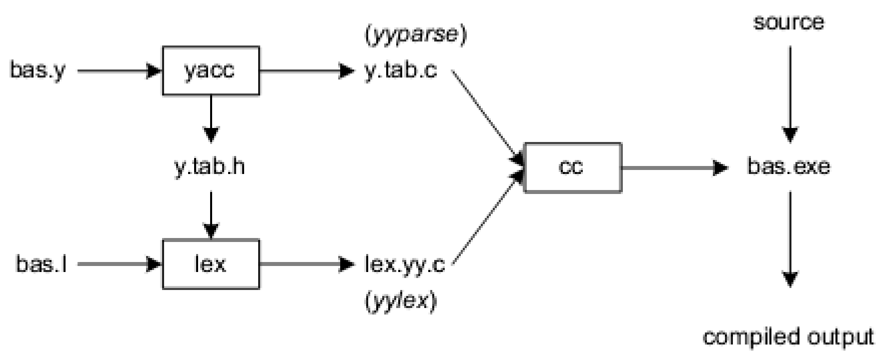


Figure 2: Building a Compiler with Lex/Yacc

Figure 2 illustrates the file naming conventions used by lex and yacc. We'll assume our goal is to write a BASIC compiler. First, we need to specify all pattern matching rules for lex (bas.l) and grammar rules for yacc (bas.y). Commands to create our compiler, bas.exe, are listed below:

```
yacc -d bas.y                    # create y.tab.h, y.tab.c
lex bas.l                        # create lex.yy.c
cc lex.yy.c y.tab.c - lfl        # compile/link
```

Yacc reads the grammar descriptions in bas.y and generates a syntax analyzer (parser), that includes function yyparse, in file y.tab.c. Included in file bas.y are token declarations. The –doption causes yacc to generate definitions for tokens and place them in file y.tab.h. Lex reads the pattern descriptions in bas.l, includes file y.tab.h, and generates a lexical analyzer, that includes function yylex, in file lex.yy.c.

Finally, the lexer and parser are compiled and linked together to create executable bas.exe. From main we call yyparse to run the compiler. Function yyparse automatically calls yylex to obtain each token.

LexTheory

During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value.

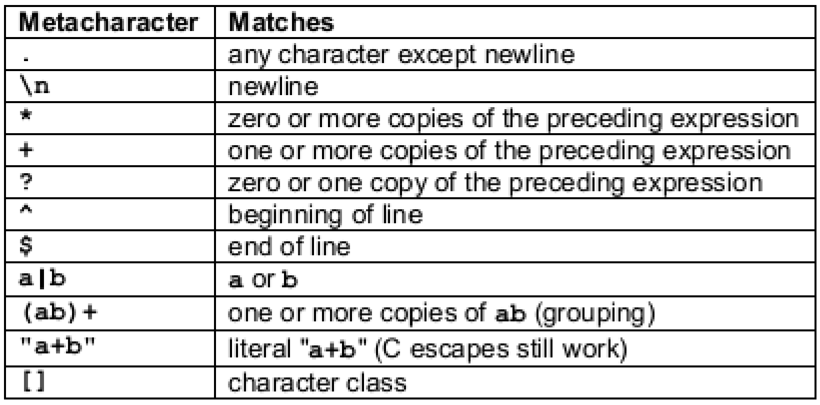
The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

letter(letter|digit)*

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

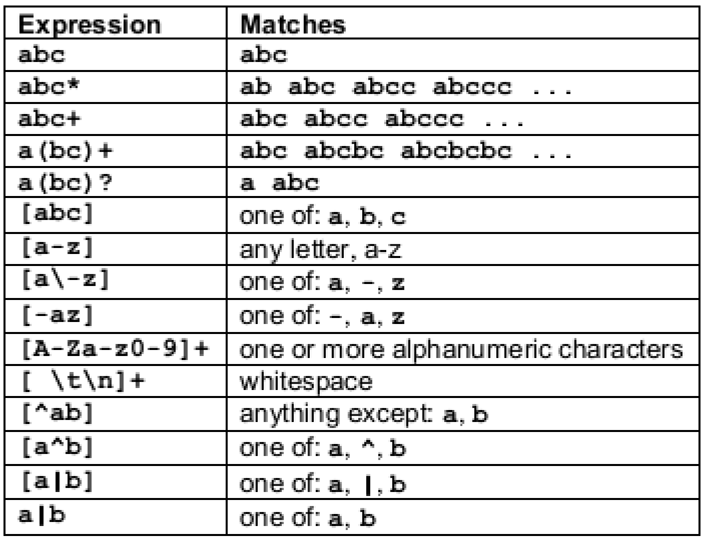
- repetition, expressed by the “*” operator
- alternation, expressed by the “|” operator
- concatenation

Practice



Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[]	character class

Table 1: Pattern Matching Primitives



Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc
[abc]	one of: a, b, c
[a-z]	any letter, a-z
[a\-z]	one of: a, -, z
[-az]	one of: -, a, z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: a, b
[a^b]	one of: a, ^, b
[a|b]	one of: a, |, b
a|b	one of: a, b

Table 2: Pattern Matching Examples

Regular expressions in lex are composed of metacharacters (Table 1). Pattern-matching examples are shown in Table 2. Within a character class normal operators lose their meaning. Two operators allowed in a character class are the hyphen (“-”) and circumflex (“^”). When used between two characters the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string, the longest match wins.

In case both matches are the same length, then the first pattern listed is used.

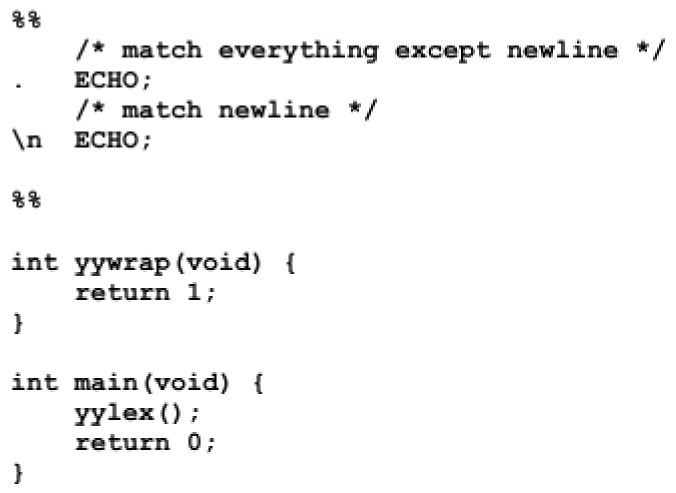
... definitions ...

```
%%
... rules ...
%%
... subroutines ...
```

Input to Lex is divided into three sections with %% dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file:

```
%%
```

Input is copied to output one character at a time. The first %% is always required, as there must always be a rules section. However if we don’t specify any rules then the default action is to match everything and copy it to output. Defaults for input and output are stdin and stdout, respectively. Here is the same example with defaults explicitly coded

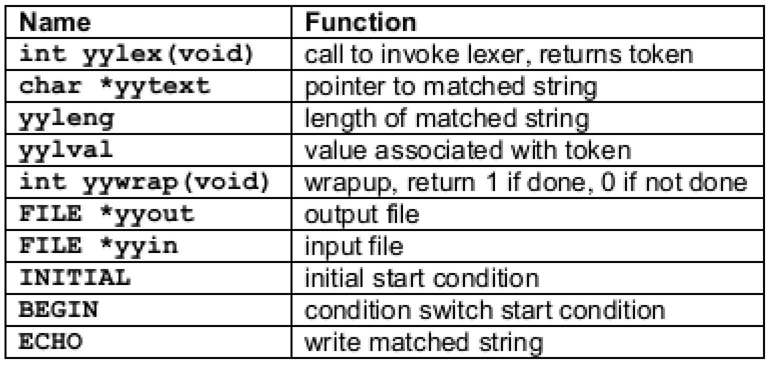


```
%%
    /* match everything except newline */
.   ECHO;
    /* match newline */
\n  ECHO;

%%

int yywrap(void) {
    return 1;
}

int main(void) {
    yylex();
    return 0;
}
```

Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by whitespace (space, tab or newline) and an optional action associated with the pattern. The action may be a single C statement, or multiple C statements, enclosed in braces. Anything not starting in column one is copied verbatim to the generated C file. We may take advantage of this behavior to specify comments in our lex file. In this example there are two patterns, “.” and “\n”, with an ECHO action associated for each pattern. Several macros and variables are predefined by lex. ECHO is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, ECHO is defined as:

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

Variable yytext is a pointer to the matched string (NULL-terminated) and yyleng is the length of the matched string. Variable yyout is the output file and defaults to stdout. Function yywrap is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required. Every C program requires a main function. In this case we simply call yylex that is the main entry-point for lex. Some implementations of lex include copies of main and yywrap in a library thuseliminating the need to code them explicitly. This is why our first example, the shortest lex program, functioned properly.



Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN`	condition switch start condition
`ECHO`	write matched string

Table 3: Lex Predefined Variables

Experiment No 1

AIM

Implementation a lexical analyzer for given language using C

ALGORITHM

1. Start
2. Read the input file.
3. Check whether the input symbol is a keyword, print it is a keyword.
4. Check whether the current input is digit, then print it is a digit.
5. Check whether the current input contains any special symbols, then print it is a special symbol.
6. Check whether the current input contains any identifier, then print it is an identifier.
7. Check whether the current input contains any operator, then print it is an operator.

Experiment No 2

AIM

Implementation a lexical analyzer for given language using LEX tool

ALGORITHM

Token declaration Part

1. Declare token named Head as #include”<”.*”>”
2. Declare token named Key as “int”|”float”|”char”|”while”|”if”|”double”|”for”|”do”|”goto”|”void”
3. Declare token named Digit as [0-9]+
4. Declare token named Id as [a-z][A-Z0-9a-z0-9]*|_[a-z][A-Z0-9a-z0-9]*
5. Declare token named Op as “+”|”-“|”*”|”/”

Translation Rule Part

1. If input lexeme matches with token {Head} then print it is a header file.
2. If input lexeme matches with token {Key} then print it is a keyword.
3. If input lexeme matches with token {Digit} then print it is a Digit.
4. If input lexeme matches with token {Id} then print it is an Identifier.
5. If input lexeme matches with token {Op} then print it is an operator.
6. If input lexeme matches with token “.”, then skip other patterns.

Auxiliary Procedure Part

8. Start
9. Read the input file.
10. If input symbol is a keyword, print it is a keyword.
11. Check whether the current input is #include “<”.*”>”, then print it is a header file.
12. Check whether the current input is digit, then print it is a digit.
13. Check whether the current input contains any identifier, then print it is an identifier.
14. Check whether the current input contains any operator, then print it is an operator.
15. Repeat the steps 2-8 until EOF.
16. Stop.

Experiment No 3

AIM

Implementation of a calculator using LEX tool

ALGORITHM

Token declaration Part

1. Declare token named Digit as [0-9]+|([0-9]*)”.”([0-9]+)
2. Declare token named Add as “+”

3. Declare token named Sub as “-“
4. Declare token named Mul as “*”
5. Declare token named Div as “/”
6. Declare token named Pow as “ ^”
7. Declare token named nl as “ \n”

Translation Rule Part

1. If input lexeme matches with token {Digit} then call the function eval().
2. If input lexeme matches with token {Add} then set op=1.
3. If input lexeme matches with token {Sub} then set op=1.
4. If input lexeme matches with token {Mul} then set op=1.
5. If input lexeme matches with token {Div} then set op=1.
6. If input lexeme matches with token {Pow} then set op=1.
7. If input lexeme matches with token {nl} then print result as a.

Auxiliary Procedure Part

1. Start
2. Initialize op=0, i as integer and a and b as float.
3. Read the input file.
4. Check whether the current input is a digit, then call the function eval().
5. Check whether the current input is +, the set op=1.
6. Check whether the current input is -, the set op=2.
7. Check whether the current input is *, the set op=3.
8. Check whether the current input is /, the set op=4.
9. Check whether the current input is ^, the set op=5.
10. Repeat 5-11 lines until EOF.
11. Stop.

Function eval()

```
1.  Start
2.  If op=0, then convert the ASCII value into float value and store in a.
3.  Else begin
    3.1.        Convert the ASCII value into float value and store in b.
    3.2.        If op=1, then a=a+b
    3.3.        If op=2, then a=a-b
    3.4.        If op=3, the a=a*b
    3.5.        If op=4, then a=a/b
    3.6.        If op=5, then a=a^b
        3.6.1.  Initialize i=a
        3.6.2.  Check whether b>1 then
            3.6.2.1.    a=a*i
        3.6.3.  Decrement b.
    3.7.        Op=0.
```

4. Repeat the steps until EOF.
5. Stop.

INTRODUCTION TO YACC

The yacc program creates parsers that define and enforce structure for character input to a computer program. To use this program, you must supply the following inputs:

Grammar file	A source file that contains the specifications for the language to recognize. This file also contains the main, yyerror, and yylex subroutines. You must supply these subroutines.
main	A C language subroutine that, as a minimum, contains a call to the yyparse subroutine generated by the yacc program. A limited form of this subroutine is available in the yacc library.
yyerror	A C language subroutine to handle errors that can occur during parser operation. A limited form of this subroutine is available in the yacc library.
yylex	A C language subroutine to perform lexical analysis on the input stream and pass tokens to the parser. You can generate this lexical analyzer subroutine using the lex command.

When the yacc command gets a specification, it generates a file of C language functions called y.tab.c. When compiled using the cc command, these functions form the yyparse subroutine and return an integer. When called, the yyparse subroutine calls the yylex subroutine to get input tokens. The yylex subroutine continues providing input until either the parser detects an error or the yylex subroutine returns an end-marker token to indicate the end of operation. If an error occurs and the yyparse subroutine cannot recover, it returns a value of 1 to main. If it finds the end-marker token, the yyparse subroutine returns a value of 0 to main.

Experiment No 4

AIM

Implementation of a calculator using YACC tool

ALGORITHM

YACC Program

Declaration Part

1. Declare interger variable I, j=1
2. Declare token ID
3. Declare operators + and – as left associative
4. Declare operators * and / as left associative
5. Declare operators ^ as left associative
6. Declare operators UMINUS as right associative

Grammar Part

```
1.  s:e'\n'{printf("%d",$1);}
2.  e:    e '+' e      {$$=$1+$3;}
          |e '-' e      {$$=$1-$3;}
          |e '*' e      {$$=$1*$3;}
          |e '/' e       {$$=$1/$3;}
          |e '^' e      {
                                int i,j=$1;
                                for(i=1;i<$3;i++)
                                 {
                                     j=j*$1;
                                    $$=j;
                                  }
                            }
            |'('e')'        {$$=$2;}
            |ID             {$$=$2;}       ;
```

Auxiliary Procedure Part

Display message “ Enter the Expression”
Read The input and parse the Expression

LEX Program

Token declaration Part

Declare external variable yylval

Translation Rule Part

If lexeme matches with pattern [0-9] then return token ID
If lexeme matches with pattern [+-/*^\n] then return *yyrtext
If lexeme matches with pattern [.]then return yytext(0)

Experiment No 5

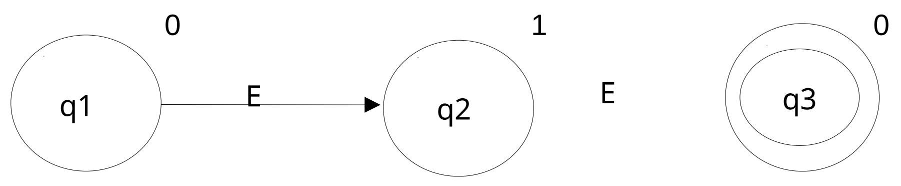
AIM

Program to find ε – closure of all states of any given NFA with ε transition

ALGORITHM

1. Start
2. Declare structure and various functions
3. Declare function for find closure,insert trantbl, findalpha, print closure
4. Declare local variables
5. Read no of alphabets and then alphabets in automata(use e for epsilon)
6. Read no of states
7. Read no of transitions
8. Read transition as matrix form
9. Call the function to insert transition table
10. For each state in the automata call the function find closure and then call the function e-closure

Sample



Experiment No 6

AIM

Program to convert ε-NFA to NFA

ALGORITHM

A e-NFA is a nondeterministic finite automaton which has e-transitions in addition to the nondeterministic transitions it already had, which means, a transition can take place from one state to another even when there is no input symbol at all(meaning the input is a null string).

Sample

No Change In Initial State

No Change In The Total No. Of States a b c

May Be Change In Final States E E)

Let M=(Q,Σ,δ,q0,F) – ε-NFA

M1= (Q1,Σ,δ1,q01,F1) – NFA

1) Initial State

q01=q0

2) Construction Of δ11

δ11

(q,x)=ε-Closure(δ(ε-Closure(q),x)

3) Final State

Every State Who's ε-Closure Contain Final State Of ε-NFA Is Final State In NFA

1. Start
2. Declare structure and various functions
3. Declare function for find closure,insert trantbl, findalpha,union closure,find final closure and print closure
4. Declare local variables
5. Read no of alphabets and then alphabets in automata(use e for epsilon)
6. Read no of states

7. Read no of transitions
8. Read transition as matrix form
9. Call the function to insert transition table
10. For each state in the automata call the function find closure
11. And then call the function unionclosure ,print e closure and findfinalstate function

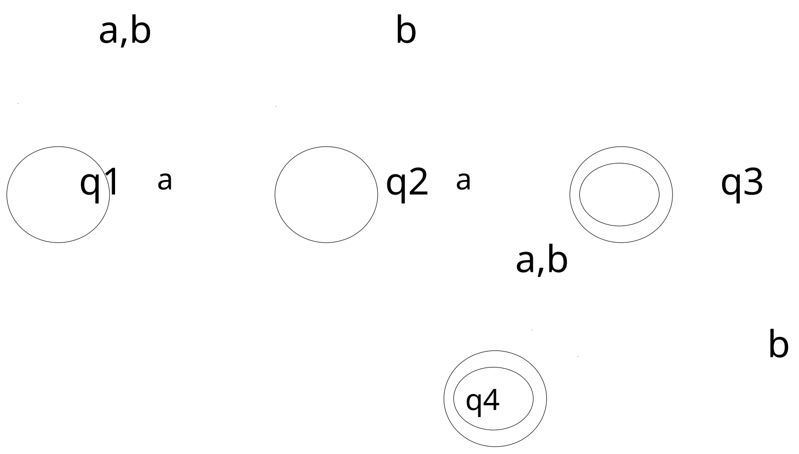
Experiment No 7

AIM

Write a Program to convert NFAto DFA

ALGORITHM

1. Take ∈ closure for the beginning state of NFA as beginning state of DFA.
2. Find the states that can be traversed from the present for each input symbol (union of transition value and their closures for each states of NFA present in current state of DFA).
3. If any new state is found take it as current state and repeat step 2.
4. Do repeat Step 2 and Step 3 until no new state present in DFA transition table.
5. Mark the states of DFA which contains final state of NFA as final states of DFA.



sample

Link : https://mycodetechniques.blogspot.com/search/label/Compiler

Experiment No 8

AIM

Implementation of Intermediate code generation for simple expressions

ALGORITHM

1. Start
2. Declare the arrays for operation, arguments and result
3. Create an input file as input.txt
4. Create an output file as blank for output the ICG code
5. Read from the input file and store the values for various operation
6. Check each operation and implement the operation code
7. Write the corresponding code to output file
8. Repeat the steps 6 and 7 till end of file(input.txt)
9. Stop

Experiment No 9

AIM

Implementation of Language Acceptance ($a^n b$)

ALGORITHM

1. Start
2. Define variables A,B & NL as tokens
3. Define the rules for accepting and rejecting the given string
4. Read the string entered by the user
5. Print “String matched” if it obeys the rule otherwise print “invalid input”.
6. Stop

Experiment No 10

AIM

Program to find Simulate First and Follow of any given grammar.

ALGORITHM

1. Start
2. Declare the functions to find the first and follow
3. Local variable declarations for rules count and for string(rules)
4. Declare character array to store first and follow after calculation
5. Read the rules
6. Initialize the loop to find the first of each grammer(non terminals) and print
7. Then initialize the loop to the followt of each grammer(non terminals) and print
8. Stop

Production Rules:

```
E=TR
R=+TR
R=#
T=FY
Y=*FY
Y=#
 F=(E)
F=i
```

Experiment No 11

AIM

Recursive descent parsing using C for the given grammar

E -> E+T | T

T -> T*F | F

F -> (E) | id

ALGORITHM:

First we have to avoid left recursion

E -> TE'

E' -> +TE' | ε

T -> FT'

T' -> *FT' | ε

F -> (E) | id

After eliminating Left recursion, we have to simply move from one character to next by checking whether it follow the grammar. In this program, ε is indicated as $.

CODES

1. Implementation of Lexical Analyser using C

```
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>

main()
{
        FILE *f1;
char c,str[100];
int num=0,i=0,lineno=1;
        f1=fopen("input.c","r");
while((c=getc(f1))!=EOF)
        {
if(isdigit(c))
                {
num=c-48;
                        c=getc(f1);
while(isdigit(c))
                        {
num=num*10+(c-48);
                                c=getc(f1);
                        }
printf("%d is a number \n",num);
ungetc(c,f1);
                }
else if(isalpha(c))
                {
str[i++]=c;
                        c=getc(f1);
while(isdigit(c)||isalpha(c)||c=='_'||c=='$')
                        {
str[i++]=c;
                                c=getc(f1);
                        }
str[i++]='\0';
                        if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||
strcmp("int",str)==0||strcmp("float",str)==0||strcmp("char",str)==0||strcmp("double",str)==0||
strcmp("static",str)==0||strcmp("switch",str)==0||strcmp("case",str)==0)
printf("%s is a keyword \n",str);
else
printf("%s is a identifier \n",str);
ungetc(c,f1);
                        i=0;
```

```
               }
else if(c==' '||c=='\t')
printf("\n");
else if(c=='\n')
lineno++;
else
printf("%c is a special symbol \n",c);
        }
printf("Total no. of lines are: %d \n",lineno);
fclose(f1);

}
```

Output

```
[cs1633@LabServer ~]$ cc lex.c
[cs1633@LabServer ~]$ ./a.out
 Enter the file name of program
lex.c
# is a special symbol
include is identifier
< is a special symbol
ctype is identifier
. is a special symbol
h is identifier
> is a special symbol
# is a special symbol
include is identifier
< is a special symbol
stdio is identifier
. is a special symbol
h is identifier
> is a special symbol
# is a special symbol
include is identifier
.
.
..
..
```

<u>2.Implementation of Lexical Analyser using LEX tool</u>

```
%{
#include<stdio.h>
#include<string.h>
char key[100][100],head[100][100],dig[100][100],op[100][100],id[100][100];
int i=0,j=0,k=0,l=0,a=0,b=0,c=0,d=0,m=0,n=0;
%}

KW "int"|"while"|"if"|"else"|"for"|"char"|"float"|"case"|"switch"
HF "#include<".*">"
```

```
OP "+"|"-"|"*"|"/"|"="
DIG [0-9]*|[0-9]*"."[0-9]+
ID [a-zA-Z][a-zA-Z0-9]*

%%
{KW} {strcpy(key[i],yytext);i++;}
{HF} {strcpy(head[j],yytext);j++;}
{DIG} {strcpy(dig[k],yytext);k++;}
{OP} {strcpy(op[m],yytext);m++;}
{ID} {strcpy(id[n],yytext);n++;}
. {}
%%

main()
{
	yyin=fopen("input.c","r+");
	yylex();
	printf("\nThe keywords are");
	for(a=0;a<i;a++)
	{
		printf("\n%s",key[a]);
	}
	printf("\nThe headerfiles are ");
	for (b=0;b<j;b++)
	{
		printf("\n%s",head[b]);
	}

	printf("\nThe digits are");
	for(c=0;c<k;c++)
	{
		printf("\n%s",dig[c]);
	}
	printf("\noperators ...");
	for (d=0;d<m;d++)
	{
		printf("\n%s",op[d]);
	}
	printf("\nidentifiers....");
	for(d=0;d<n;d++)
	{
		printf("\n%s",id[d]);
	}
}

int yywrap()
{
	printf("Errors..\n");
	return 1;
}
```

Output

```
[cs1636@LabServer s7comp]$ lex lex.l
[cs1636@LabServer s7comp]$ cc lex.yy.c
[cs1636@LabServer s7comp]$ ./a.out

The keywords are
int
float
The headerfiles are
#include<stdio.h>
The digits are
0
0.00
operators...

=
=
identifiers...

void
void
void
void
void
void
void[cs1636@LabServer s7comp]$
```

<u>3. Implementation of Calculator using LEX</u>

```
%{
        #include<stdio.h>
        #include<stdlib.h>
        int op=0;
        float a,b,n,i;
        void digit();

%}
digit [0-9]+|[0-9]*"."[0-9]+
add "+"
sub "-"
mul "*"
div "/"
pow "^"
end "\n"
%%

{digit} {digit();}
{add} {op=1;}
{sub} {op=2;}
```

```
{mul} {op=3;}
{div} {op=4;}
{pow} {op=5;}
{end} {printf("Result is %f",a);printf("\nEnter new equation\n");}
. {exit(0);}
%%

int main()
{
        printf("Enter the equation\n");
        yylex();
}
int vywrap()
{
        return(1);
}
void digit()
{
        if(op==0)
        {
                a=atof(yytext);
        }
        else
        {
                b=atof(yytext);
                switch(op)
{
                case 1:
                        a=a+b;
                        break;
                case 2:
                        a=a-b;
                        break;
                case 3 :
                        a=a*b;
                        break;
                case 4:
                        a=a/b;
                        break;
                case 5:
                        i=1;
                        n=a;
                        while(i<b)
                        {
                                a=a*n;
                                i++;
                        }
                        break;
                default:
                        printf("invalid operation\n");
```

```
		}
		op=0;
}
}
```

Output

```
[cs1633@LabServer s7comp]$ lex cal.l
[cs1633@LabServer s7comp]$ cc lex.yy.c
[cs1633@LabServer s7comp]$ ./a.out
Enter the equation
4+7
Result is 11.000000
```

4. Implementation of Desktop calculator using YACC

LEX file

```
%{
#include "y.tab.h"
extern int yylval;
%}
DIGIT [0-9]+
OP [+|-|*|/]
%%
{DIGIT} {yylval=atoi(yytext); return NUM;}
{OP} {return *yytext;}
[\n] {return NL;}
%%

int yywrap()
{
return 1;
}
```

YACC file

```
%{
#include<stdio.h>
#include<stdlib.h>
int yylval;
%}
%token NUM NL
%left '+' '-'
%left '/''*'
%right UMINUS
%%
S:E NL{printf("result is %d",$$);}
```

```
E:E'+'E{$$=$1+$3;}
|E'-'E{$$=$1-$3;}
|'-'E %prec UMINUS{$$=-$2;}
|NUM {$$=$1;}
%%
void main()
{
yyparse();
}
int yyerror()
{
printf("input error");
}
```

Output

```
[cs1633@LabServer s7comp]$ lex calyac.l
[cs1633@LabServer s7comp]$ yacc -d calyac.y
[cs1633@LabServer s7comp]$ cc lex.yy.c y.tab.c -ll
[cs1633@LabServer s7comp]$ ./a.out
6+6
result is 12
```

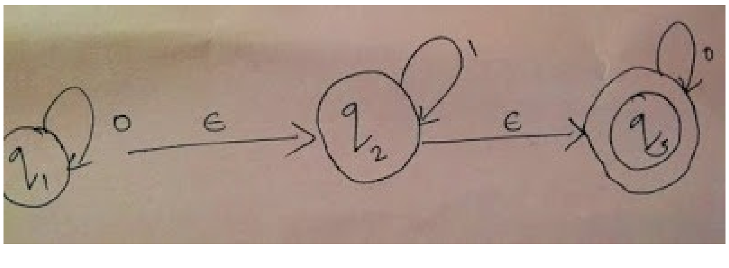
<u>5. C Program to findɛ Closure of all states of NFA</u>

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
	int st;
	struct node *link;
};

void findclosure(int,int);
void insert_trantbl(int,char,int);
int findalpha(char);
void print_e_closure(int);

static int set[20],nostate,noalpha,s,notransition,c,r,buffer[20];
char alphabet[20];
static int e_closure[20][20]={0};
struct node *transition[20][20]={NULL};

void main()
{
	int i,j,k,m,t,n;
	struct node *temp;
	printf("Enter the number of alphabets?\n");
	scanf("%d",&noalpha);
	getchar();
	printf("NOTE:-[ use leter e as epsilon]\n");
	printf("NOTE:- [e must be last character,if it is present]\n");
	printf("\nEnter alphabets?\n");
	for(i=0;i<noalpha;i++)
	{
		alphabet[i]=getchar();
		getchar();
	}
	printf("\nEnter the number of states?\n");
	scanf("%d",&nostate);
	printf("\nEnter no of transition?\n");
	scanf("%d",&notransition);
	printf("NOTE:- [Transition is in the form--> qno albhabet qno]\n",notransition);
	printf("NOTE:- [States number must be greater than zero]\n");
	printf("\nEnter transition?\n");
	for(i=0;i<notransition;i++)
	{
		scanf("%d %c%d",&r,&c,&s);
		insert_trantbl(r,c,s);
	}
	printf("\n");
	printf("e-closure of states...........\n");
	printf("------------\n");
	for(i=1;i<=nostate;i++)
	{
		c=0;
		for(j=0;j<20;j++)
		{
			buffer[j]=0;
			e_closure[i][j]=0;
		}
		findclosure(i,i);
		printf("\ne-closure(q%d): ",i);
		print_e_closure(i);
	}
}

void findclosure(int x,int sta)
{
	struct node *temp;
	int i;
	if (buffer[x])
		return;

	e_closure[sta][c++]=x;
	buffer[x]=1;
	if (alphabet[noalpha-1]=='e' && transition[x][noalpha-1]!=NULL)
	{
	temp=transition[x][noalpha-1];
	while(temp!=NULL)
```

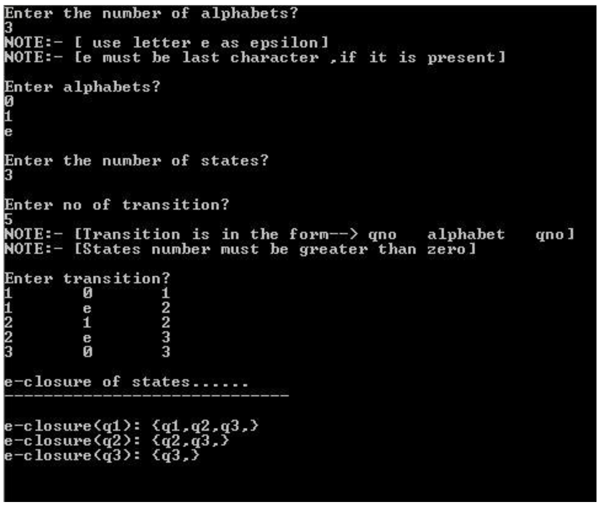
```
        {
                findclosure(temp->st,sta);
                temp=temp->link;
        }
}}
void insert_trantbl(int r,char c,int s)
{
        int j;
        struct node *temp;
        j=findalpha(c);
if(j==999)
        {
                printf("error\n");
                exit(0);
        }
        temp=(struct node *)malloc(sizeof(struct node));
        temp->st=s;
        temp->link=transition[r][j];
        transition[r][j]=temp;

}
int findalpha(char c)
{
        int i;
        for(i=0;i<noalpha;i++)
        if (alphabet[i]==c)
                return i;
        return(999);
}
void print_e_closure(int i)
{
        int j;
        printf("{");
        for (j=0;e_closure[i][j]!=0;j++)
        printf("q%d,",e_closure[i][j]);
        printf("}");
}
```

Output

```
[cs1636@LabServer s7comp]$ cc eclosure.c
 [cs1636@LabServer s7comp]$ ./a.out
Enter the number of alphabets?
3
NOTE:- [ use letter e as epsilon]
NOTE:- [e must be last character ,if it is
present]

Enter alphabets?
0
1
```



```
e

Enter the number of states?
3

Enter no of transition?
5
NOTE:- [Transition is in the form–> qno alphabet
qno]
NOTE:- [States number must be greater than zero]

Enter transition?
1 0 1
1 e 2
2 1 2
2 e 3
3 0 3

e-closure of states……
————————————

e-closure(q1): {q1,q2,q3,}
e-closure(q2): {q2,q3,}
e-closure(q3): {q3,}[cs1636@LabServer s7comp]$
```



```
Enter the number of alphabets?
3
NOTE:- [ use letter e as epsilon]
NOTE:- [e must be last character ,if it is present]

Enter alphabets?
0
1
e

Enter the number of states?
3

Enter no of transition?
5
NOTE:- [Transition is in the form--> qno   alphabet   qno]
NOTE:- [States number must be greater than zero]

Enter transition?
1       0       1
1       e       2
2       1       2
2       e       3
3       0       3

e-closure of states......
-----------------------------

e-closure(q1): {q1,q2,q3,}
e-closure(q2): {q2,q3,}
e-closure(q3): {q3,}
```

6. C Program for converting ε-NFA to NFA

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int st;
struct node *link;
};
void findclosure(int,int);
void insert_trantbl(int,char,int);
int findalpha(char);
void findfinalstate(void);
void unionclosure(int);
void print_e_closure(int);
static int set[20],nostate,noalpha,s,notransition,nofinal,start,finalstate[20],c,r,buffer[20];
char alphabet[20];
static int e_closure[20][20]={0};
struct node * transition[20][20]={NULL};
void main()
{
```

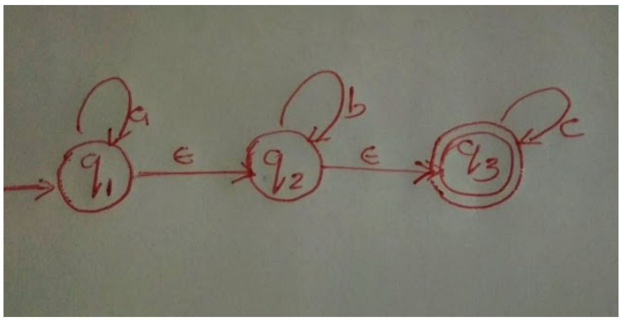
```
int i,j,k,m,t,n;
struct node *temp;
printf("enter the number of alphabets?\n");
scanf("%d",&noalpha);
getchar();
printf("NOTE: - [ use letter e as epsilon)\n");
printf("NOTE:- [ e must be last character ,if it is present)\n ");
printf("\nEnter alphabets?\n");
for(i=0;i<noalpha;i++)
        {
alphabet[i]= getchar();
getchar();
        }
printf("Enter the number of states ?\n");
scanf("%d" ,&nostate);
printf("Enter the start state ?\n");
scanf( "%d",&start);
printf("Enter the number of final states?\n");
scanf("%d",&nofinal);
printf("Enter the final states?\n");
for(i=0;i<nofinal;i++)
        {
scanf("%d",&finalstate[i]);
printf("Enter no of transition?\n");
scanf("%d",&notransition);
printf("NOTE:- [Transition is in the form-- > qno alphabet qno]\n" ,notransition);
printf( "NOTE:- [States number must be greater than zero]\n");
printf("\nEnter transition?\n");
for(i=0;i<notransition;i++)
                {
scanf("%d %c%d",&r,&c,&s);
                        insert_trantbl(r,c,s);
                }
printf("\n");
for(i=1;i<=nostate;i++)
                {
                        c=0;
for(j=0;j<20;j++)
                        {
buffer[j]=0;
                                e_closure[i][j]=0;
                        }
findclosure(i,i);
                }
printf("Equivalent NFA without epsilon\n");
                printf("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - \n");
printf("start state:");
                print_e_closure(start);
printf("\nAlphabets:");
for(i=0;i<noalpha;i++)
```

```
        {
printf("%c",alphabet[i]);
printf("\nStates:");
        }
for(i=1;i<=nostate;i++)
        {
            print_e_closure(i);
        }
printf("\nTransitions are...:\n");
for(i=1;i<=nostate;i++)
        {
for(j=0;j<noalpha-1;j++)
            {
for(m=1;m<=nostate;m++)
                {
set[m]=0;
                }
for(k=0;e_closure[i][k]!=0;k++)
                {
                    t=e_closure [i][k];
temp=transition[t][j];
while(temp!=NULL)
                    {
unionclosure(temp->st);
temp=temp->link;
                    }
                }
printf("\n");
                print_e_closure(i);
printf("%c\t",alphabet[j]);
printf("{");
for(n=1;n<=nostate;n++)
                {
if(set[n]!=0)
                    {
printf("q%d,",n);
                    }
                }
printf("}");
            }
        }
printf("\nFinal states:");
findfinalstate();
    }
}
void findclosure(int x,int sta)
{
struct node *temp;
int i;
if(buffer[x])
```

```
return;
        e_closure[sta][c++]=x;
buffer[x]=1;
if(alphabet[noalpha-1]=='e'&&transition[x][noalpha-1]!=NULL)
        {
temp =transition[x][noalpha-1];
while(temp!=NULL)
                {
findclosure(temp->st,sta);
temp=temp->link;
    }
        }
}
void insert_trantbl(int r,char c,int s)
{
int j;
struct node *temp;
        j=findalpha(c);
if(j==999)
        {
printf("error\n");
exit(0);
        }
temp=(struct node *) malloc(sizeof(struct node));
temp->st=s;
temp-> link=transition[r][j];
transition[r][j]=temp;
}
int findalpha(char c)
{
int i;
for(i=0;i<noalpha;i++)
        {
if(alphabet[i]==c)
                {
return i;
return(999);
                }
        }
}
void unionclosure(int i)
{
int j=0,k;
while(e_closure[i][j]!=0)
        {
                k=e_closure[i][j];
set[k]=1;
j++;
        }
}
```

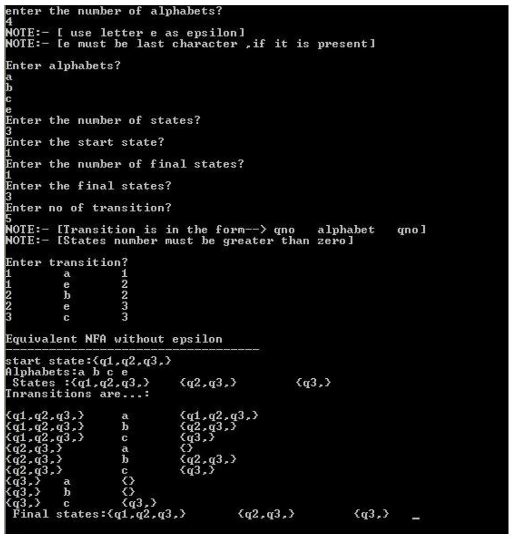
```
void findfinalstate()
{
int i,j,k,t;
for(i=0;i<nofinal;i++)
        {
for(j=1;j<=nostate;j++)
                {
for(k=0;e_closure[j][k]!=0;k++)
                        {
if(e_closure[j][k]==finalstate[i])
                                {
                                        print_e_closure(j);
                                }
                        }
                }
        }
}
void print_e_closure(int i)
{
int j;
printf("{");
for(j=0;e_closure[i][j]!=0;j++)
        {
printf("q%d,",e_closure[i][j]);
printf("\t");
                        }
printf("}");
                                }
```

Output



```
[cs1636@LabServer s7comp]$ ./a.out
enter the number of alphabets?
4
NOTE:- [ use letter e as epsilon]
NOTE:- [e must be last character ,if it is
present]

Enter alphabets?
a
b
c
e
Enter the number of states?
3
Enter the start state?
1
Enter the number of final states?
1
Enter the final states?
```

```
3
Enter no of transition?
5
NOTE:- [Transition is in the form--> qno   alphabet   qno]
NOTE:- [States number must be greater than zero]

Enter transition?
1 a 1
1 e 2
2 b 2
2 e 3
3 c 3

Equivalent NFA without epsilon
-----------------------------------
start state:{q1,q2,q3,}
Alphabets:a b c e
 States :{q1,q2,q3,}    {q2,q3,}        {q3,}
Tnransitions are...:

{q1,q2,q3,}     a       {q1,q2,q3,}
{q1,q2,q3,}     b       {q2,q3,}
{q1,q2,q3,}     c       {q3,}
{q2,q3,}        a       {}
{q2,q3,}        b       {q2,q3,}
{q2,q3,}        c       {q3,}
{q3,}   a       {}
{q3,}   b       {}
{q3,}   c       {q3,}
 Final states:{q1,q2,q3,}       {q2,q3,}        {q3,}   [cs1636@LabServer s7comp]$
```



```
enter the number of alphabets?
4
NOTE:- [ use letter e as epsilon]
NOTE:- [e must be last character ,if it is present]

Enter alphabets?
a
b
c
e
Enter the number of states?
3
Enter the start state?
1
Enter the number of final states?
1
Enter the final states?
3
Enter no of transition?
5
NOTE:- [Transition is in the form--> qno   alphabet   qno]
NOTE:- [States number must be greater than zero]

Enter transition?
1       a       1
1       e       2
2       b       2
2       e       3
3       c       3

Equivalent NFA without epsilon
-----------------------------------
start state:{q1,q2,q3,}
Alphabets:a b c e
 States :{q1,q2,q3,}    {q2,q3,}        {q3,}
Tnransitions are...:

{q1,q2,q3,}     a       {q1,q2,q3,}
{q1,q2,q3,}     b       {q2,q3,}
{q1,q2,q3,}     c       {q3,}
{q2,q3,}        a       {}
{q2,q3,}        b       {q2,q3,}
{q2,q3,}        c       {q3,}
{q3,}   a       {}
{q3,}   b       {}
{q3,}   c       {q3,}
 Final states:{q1,q2,q3,}       {q2,q3,}        {q3,}   _
```

7. NFA to DFA

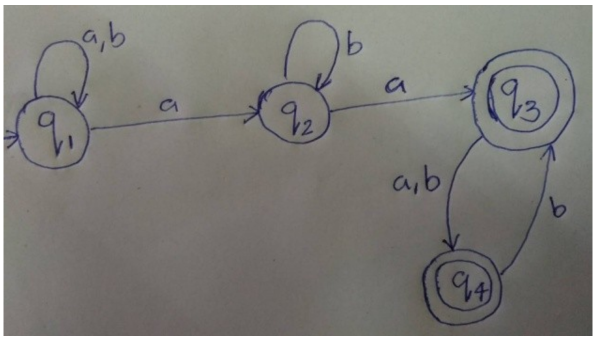
```c
#include<stdio.h>
#include<stdlib.h>
struct node
{
	int st;
	struct node *link;
};
struct node1
{

	int nst[20];
```

```
};

void insert(int ,char, int);
int findalpha(char);
void findfinalstate(void);
int insertdfastate(struct node1);
int compare(struct node1,struct node1);
void printnewstate(struct node1);
static int set[20],nostate,noalpha,s,notransition,nofinal,start,finalstate[20],c,r,buffer[20];
int complete=-1;
char alphabet[20];
static int eclosure[20][20]={0};
struct node1 hash[20];
struct node * transition[20][20]={NULL};
void main()
{
        int i,j,k,m,t,n,l;
        struct node *temp;
        struct node1 newstate={0},tmpstate={0};

        printf("Enter the number of alphabets?\n");
        printf("NOTE:- [ use letter e as epsilon]\n");
        printf("NOTE:- [e must be last character ,if it is present]\n");
        scanf("%d",&noalpha);
        getchar();
        printf("\nEnter alphabets?\n");
        for(i=0;i<noalpha;i++)
         {

                alphabet[i]=getchar();
                getchar();
        }
        printf("Enter the number of states?\n");
        scanf("%d",&nostate);
        printf("Enter the start state?\n");
        scanf("%d",&start);
        printf("Enter the number of final states?\n");
        scanf("%d",&nofinal);
        printf("Enter the final states?\n");
        for(i=0;i<nofinal;i++)
        scanf("%d",&finalstate[i]);
        printf("Enter no of transition?\n");

        scanf("%d",&notransition);
        printf("NOTE:- [Transition is in the form–> qno alphabet qno]\n",notransition);
        printf("NOTE:- [States number must be greater than zero]\n");
        printf("\nEnter transition?\n");


        for(i=0;i<notransition;i++)
```

```
	{


		scanf("%d %c%d",&r,&c,&s);
		insert(r,c,s);

	 }
	for(i=0;i<20;i++)
	{
		for(j=0;j<20;j++)
			hash[i].nst[j]=0;
	}
	complete=-1;
	i=-1;
	printf("\nEquivalent DFA.....\n");
	printf(".......................\n");
	printf("Trnsitions of DFA\n");

	newstate.nst[start]=start;
	insertdfastate(newstate);
	while(i!=complete)
	{
		i++;
		newstate=hash[i];
		for(k=0;k<noalpha;k++)
		{
			 c=0;
			for(j=1;j<=nostate;j++)
				set[j]=0;
			for(j=1;j<=nostate;j++)
			{
				 l=newstate.nst[j];
				if(l!=0)
				{
					temp=transition[l][k];
					while(temp!=NULL)
					{
						if(set[temp->st]==0)
						{
							c++;
							set[temp->st]=temp->st;
						}
						temp=temp->link;


					  }
				}
			}
			printf("\n");
			if(c!=0)
```

```
                        {
                                for(m=1;m<=nostate;m++)
                                        tmpstate.nst[m]=set[m];

                                insertdfastate(tmpstate);

                                printnewstate(newstate);
                                printf("%c\t",alphabet[k]);
                                printnewstate(tmpstate);
                                        printf("\n");
                        }
                        else
                        {
                                printnewstate(newstate);
                                printf("%c\t", alphabet[k]);
                                printf("NULL\n");
                        }

                }
          }
        printf("\nStates of DFA:\n");
        for(i=0;i<complete;i++)
        printnewstate(hash[i]);
        printf("\n Alphabets:\n");
        for(i=0;i<noalpha;i++)
        printf("%c\t",alphabet[i]);
        printf("\n Start State:\n");
        printf("q%d",start);
        printf("\nFinal states:\n");
        findfinalstate();

}

int insertdfastate(struct node1 newstate)
{
        int i;
        for(i=0;i<=complete;i++)
        {
                if(compare(hash[i],newstate))
                return 0;
        }
        complete++;
        hash[complete]=newstate;
        return 1;
}
int compare(struct node1 a,struct node1 b)
{
        int i;
```

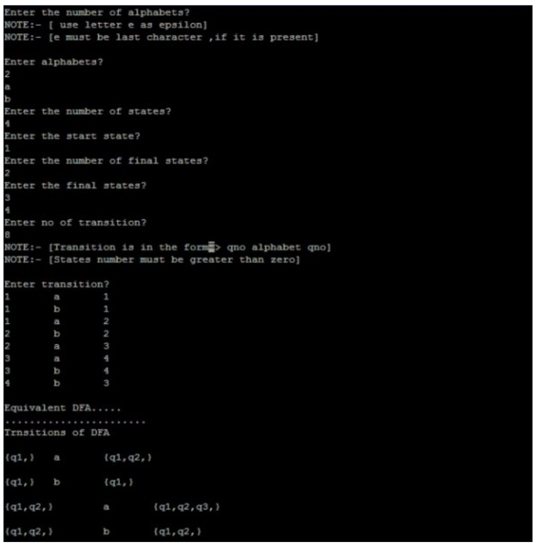
```
            for(i=1;i<=nostate;i++)
            {
                        if(a.nst[i]!=b.nst[i])
                        return 0;
            }
            return 1;


}

void insert(int r,char c,int s)
{
int j;
struct node *temp;
       j=findalpha(c);
if(j==999)
       {
                        printf("error\n");
                        exit(0);
       }
temp=(struct node *) malloc(sizeof(struct node));
temp->st=s;
temp->link=transition[r][j];
transition[r][j]=temp;
}

int findalpha(char c)
{
int i;
for(i=0;i<noalpha;i++)
            if(alphabet[i]==c)
            return i;

return(999);


}


void findfinalstate()
{
int i,j,k,t;

for(i=0;i<=complete;i++)
  {
            for(j=1;j<=nostate;j++)
            {
                        for(k=0;k<nofinal;k++)
                        {
                                    if(hash[i].nst[j]==finalstate[k])
                        {
                                printnewstate(hash[i]);
                                printf("\t");
                                j=nostate;
                                break;
                        }
                }
        }
  }
}


void printnewstate(struct node1 state)
{
        int j;
        printf("{");
        for(j=1;j<=nostate;j++)
        {
                if(state.nst[j]!=0)
                printf("q%d,",state.nst[j]);
        }
        printf("}\t");

}
```



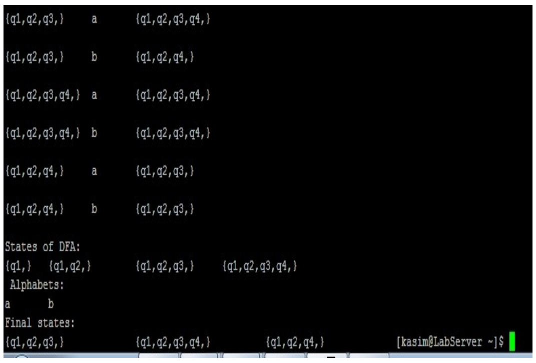
Output

```
[cs1636@LabServer s7comp]$ ./a.out
Enter the number of alphabets?
NOTE:- [ use letter e as epsilon]
NOTE:- [e must be last character ,if it is present]
2
Enter the alphabets
a
b
Enter the number of states?
4
Enter the start state?
1
Enter the number of final states?
2
Enter the final states?
3
4
Enter no of transition?
8
NOTE:- [Transition is in the form–> qno alphabet qno]
NOTE:- [States number must be greater than zero]
```

```
Enter transition?
1 a 1
1 b 1
1 a 2
2 b 2
2 a 3
3 a 4
3 b 4
4 b 3

Equivalent DFA.....
.......................
Trnsitions of DFA

{q1,}   a       {q1,q2,}

{q1,}   b       {q1,}

{q1,q2,}        a       {q1,q2,q3,}

{q1,q2,}        b       {q1,q2,}

{q1,q2,q3,}     a       {q1,q2,q3,q4,}

{q1,q2,q3,}     b       {q1,q2,q4,}

{q1,q2,q3,q4,}  a       {q1,q2,q3,q4,}

{q1,q2,q3,q4,}  b       {q1,q2,q3,q4,}

{q1,q2,q4,}     a       {q1,q2,q3,}

{q1,q2,q4,}     b       {q1,q2,q3,}

States of DFA:
{q1,}   {q1,q2,}        {q1,q2,q3,}     {q1,q2,q3,q4,}  {q1,q2,q4,}
 Alphabets:
a       b
 Start State:
q1
Final states:
{q1,q2,q3,}             {q1,q2,q3,q4,}          {q1,q2,q4,}             [cs1636@LabServer s7comp]$
```



```
Enter the number of alphabets?
NOTE:- [ use letter e as epsilon]
NOTE:- [e must be last character ,if it is present]

Enter alphabets?
2
a
b
Enter the number of states?
4
Enter the start state?
1
Enter the number of final states?
2
Enter the final states?
3
4
Enter no of transition?
8
NOTE:- [Transition is in the form=> qno alphabet qno]
NOTE:- [States number must be greater than zero]

Enter transition?
1       a       1
1       b       1
1       a       2
2       b       2
2       a       3
3       a       4
3       b       4
4       b       3

Equivalent DFA.....
.......................
Trnsitions of DFA

{q1,}   a       {q1,q2,}

{q1,}   b       {q1,}

{q1,q2,}        a       {q1,q2,q3,}

{q1,q2,}        b       {q1,q2,}
```



```
{q1,q2,q3,}     a       {q1,q2,q3,q4,}

{q1,q2,q3,}     b       {q1,q2,q4,}

{q1,q2,q3,q4,}  a       {q1,q2,q3,q4,}

{q1,q2,q3,q4,}  b       {q1,q2,q3,q4,}

{q1,q2,q4,}     a       {q1,q2,q3,}

{q1,q2,q4,}     b       {q1,q2,q3,}

States of DFA:
{q1,}   {q1,q2,}        {q1,q2,q3,}     {q1,q2,q3,q4,}
 Alphabets:
a       b
Final states:
{q1,q2,q3,}             {q1,q2,q3,q4,}          {q1,q2,q4,}             [kasim@LabServer ~]$
```

<u>8. Intermediate Code Generation</u>

```
#include<ctype.h>
#include<stdio.h>
```

```
#include<string.h>
char op[2],arg1[5],arg2[5],result[5];
void main()
{
        FILE *fp1,*fp2;
        fp1=fopen("input.txt","r");
        fp2=fopen("output.txt","w");
while(!feof(fp1))
        {
fscanf(fp1,"%s%s%s%s",op,arg1,arg2,result);
if(strcmp(op,"+")==0)
                {
fprintf(fp2,"\nMOV R0,%s",arg1);
fprintf(fp2,"\nADD R0,%s",arg2);
fprintf(fp2,"\nMOV %s,R0",result);
                }
if(strcmp(op,"*")==0)
                {
fprintf(fp2,"\nMOV R0,%s",arg1);
fprintf(fp2,"\nADD R0,%s",arg2);
fprintf(fp2,"\nMOV %s,R0",result);
                }
if(strcmp(op,"-")==0)
                {
fprintf(fp2,"\nMOV R0,%s",arg1);
fprintf(fp2,"\nSUB R0,%s",arg2);
fprintf(fp2,"\nMOV %s,R0",result);
                }
if(strcmp(op,"/")==0)
                {
fprintf(fp2,"\nMOV R0,%s",arg1);
fprintf(fp2,"\nDIV R0,%s",arg2);
fprintf(fp2,"\nMOV %s,R0",result);
                }
if(strcmp(op,"=")==0)
                {
fprintf(fp2,"\nMOV R0,%s",arg1);
fprintf(fp2,"\nMOV %s,R0",result);
                }
        }
fclose(fp1);
fclose(fp2);
}
```

<u>Output</u>

<u>input.txt</u>

```
+ a b t1
* c d t2
- t1 t2 t
= t ?x
```

<u>output.txt</u>

```
MOV R0,a
ADD R0,b
MOV t1,R0
MOV R0,c
ADD R0,d
MOV t2,R0
MOV R0,t1
SUB R0,t2
MOV t,R0
MOV R0,t
MOV x,R0
```

9. Language Acceptance($a^n b$)

Lex File

```
%{
	#include"y.tab.h"
%}

%%
"a" { return A;}
"b" { return B;}
[\n] { return NL;}
. {}
%%
```

Yacc File

```
%{
	#include<stdio.h>
int a,b;
%}
%token A B NL

%%
S: C B NL {printf("String Matched\n");};
C: A | C A;
%%

main()
{
printf("Enter String:");

yyparse();
}
void yyerror()
{
printf("Invalid input");
}
```

Output

```
 [cs15071@LabServer ~]$ lex lang.l
[cs15071@LabServer ~]$ yacc -d lang.y
[cs15071@LabServer ~]$ cc lex.yy.c y.tab.c -ll
[cs15071@LabServer ~]$ ./a.out
```

```
Enter String
aab
String Matched
[cs15071@LabServer ~]$ ./a.out
Enter a string
aba
Invalid input[cs15071@LabServer ~]$
```

<u>10. First and Follow</u>

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>

void followfirst(char, int, int);
void follow(char c);

void findfirst(char, int, int);

int count, n = 0;

char calc_first[10][100];

char calc_follow[10][100];
int m = 0;

char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;

int main(int argc, char **argv)
{
int jm = 0;
int km = 0;
int i, choice;
char c, ch;
count = 8;

strcpy(production[0], "E=TR");
strcpy(production[1], "R=+TR");
strcpy(production[2], "R=#");
strcpy(production[3], "T=FY");
strcpy(production[4], "Y=*FY");
strcpy(production[5], "Y=#");
strcpy(production[6], "F=(E)");
strcpy(production[7], "F=i");

int kay;
```

```
char done[count];
int ptr = -1;
for(k = 0; k < count; k++) {
for(kay = 0; kay < 100; kay++) {
            calc_first[k][kay] = '!';
        }
    }
int point1 = 0, point2, xxx;

for(k = 0; k < count; k++)
    {
        c = production[k][0];
        point2 = 0;
xxx = 0;

for(kay = 0; kay <= ptr; kay++)
if(c == done[kay])
xxx = 1;

if (xxx == 1)
continue;

findfirst(c, 0, 0);
ptr += 1;

done[ptr] = c;
printf("\n First(%c) = { ", c);
        calc_first[point1][point2++] = c;

for(i = 0 + jm; i < n; i++) {
int lark = 0, chk = 0;

for(lark = 0; lark < point2; lark++) {

if (first[i] == calc_first[point1][lark])
            {
chk = 1;
break;
            }
        }
if(chk == 0)
        {
printf("%c, ", first[i]);
            calc_first[point1][point2++] = first[i];
        }
    }
printf("}\n");
jm = n;
point1++;
    }
```

```
printf("\n");
printf("-----------------------------------------------\n\n");
char donee[count];
ptr = -1;

for(k = 0; k < count; k++) {
for(kay = 0; kay < 100; kay++) {
            calc_follow[k][kay] = '!';
        }
    }
    point1 = 0;
int land = 0;
for(e = 0; e < count; e++)
    {
ck = production[e][0];
        point2 = 0;
xxx = 0;

for(kay = 0; kay <= ptr; kay++)
if(ck == donee[kay])
xxx = 1;

if (xxx == 1)
continue;
land += 1;

follow(ck);
ptr += 1;

donee[ptr] = ck;
printf(" Follow(%c) = { ", ck);
        calc_follow[point1][point2++] = ck;


for(i = 0 + km; i < m; i++) {
int lark = 0, chk = 0;
for(lark = 0; lark < point2; lark++)
            {
if (f[i] == calc_follow[point1][lark])
                {
chk = 1;
break;
                }
            }
if(chk == 0)
            {
printf("%c, ", f[i]);
                calc_follow[point1][point2++] = f[i];
            }
        }
```

```
printf(" }\n\n");
km = m;
point1++;
    }
}

void follow(char c)
{
int i, j;

if(production[0][0] == c) {
f[m++] = '$';
    }
for(i = 0; i < 10; i++)
    {
for(j = 2;j < 10; j++)
        {
if(production[i][j] == c)
            {
if(production[i][j+1] != '\0')
                {

followfirst(production[i][j+1], i, (j+2));
                }

if(production[i][j+1]=='\0' && c!=production[i][0])
                {

follow(production[i][0]);
                }
            }
        }
    }
}

void findfirst(char c, int q1, int q2)
{
int j;

if(!(isupper(c))) {
first[n++] = c;
    }
for(j = 0; j < count; j++)
    {
if(production[j][0] == c)
        {
if(production[j][2] == '#')
            {
if(production[q1][q2] == '\0')
first[n++] = '#';
```

```
else if(production[q1][q2] != '\0'
&& (q1 != 0 || q2 != 0))
                {

findfirst(production[q1][q2], q1, (q2+1));
                }
else
first[n++] = '#';
            }
else if(!isupper(production[j][2]))
            {
first[n++] = production[j][2];
            }
else
            {

findfirst(production[j][2], j, 3);
            }
        }
    }
}

void followfirst(char c, int c1, int c2)
{
int k;

if(!(isupper(c)))
f[m++] = c;
else
    {
int i = 0, j = 1;
for(i = 0; i < count; i++)
        {
if(calc_first[i][0] == c)
break;
        }

while(calc_first[i][j] != '!')
        {
if(calc_first[i][j] != '#')
            {
f[m++] = calc_first[i][j];
            }
else
            {
if(production[c1][c2] == '\0')
                {

follow(production[c1][0]);
                }
```

```
else
                {

followfirst(production[c1][c2], c1, c2+1);
                }
            }
j++;
        }
    }
}
```

Output

```
[cs15071@LabServer ~]$ cc ff.c
[cs15071@LabServer ~]$ ./a.out

First(E) = { (, i, }

First(R) = { +, #, }

First(T) = { (, i, }

First(Y) = { *, #, }

First(F) = { (, i, }

-----------------------------------------------

Follow(E) = { $, ),  }

Follow(R) = { $, ),  }

Follow(T) = { +, $, ),  }

Follow(Y) = { +, $, ),  }

Follow(F) = { *, +, $, ),  }
```

Production Rules:

```
E=TR
R=+TR
R=#
T=FY
Y=*FY
Y=#
 F=(E)
```

11 Construct a Recursive Parser

recursive.c

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>

char input[10];
int i,error;
void E();
void T();
void Eprime();
void Tprime();
void F();
        main()
        {
i=0;
error=0;
             printf("Enter an arithmetic expression   :  "); // Eg: a+a*a
             gets(input);
             E();
             if(strlen(input)==i&&error==0)
                    printf("\nAccepted..!!!\n");
             else printf("\nRejected..!!!\n");
                     }


void E()
{
    T();
    Eprime();
}
void Eprime()
{
    if(input[i]=='+')
    {
    i++;
    T();
    Eprime();
    }
    }
void T()
{
    F();
    Tprime();
}
void Tprime()
{
    if(input[i]=='*')
    {
                i++;
                F();
                Tprime();
                }
                }
    void F()
    {
        if(isalnum(input[i]))i++;
        else if(input[i]=='(')
        {
        i++;
        E();
        if(input[i]==')')
        i++;

        else error=1;
          }

        else error=1;
        }
```

Output

```
[cs15071@LabServer ~]$ cc ff.c
[cs15071@LabServer ~]$ ./a.out

a+(a*a)  a+a*a    Accepted

a+a+a*a+a          Accepted

a***a              Rejected
```