KALASALINGAM

ACADEMY OF RESEARCH AND EDUCATION

(DEEMED TO BE UNIVERSITY)

Under sec. 3 of UGC Act 1956. Accredited by NAAC with "A++" Grade

Anand Nagar, Krishnankoil, Srivilliputtur (Via), Virudhunagar (Dt) - 626126, Tamil Nadu | info@kalasalingam.ac.in | www.kalasalingam.ac.in

SCHOOL OF COMPUTING

Department of Computer Science and Engineering

Automata and Compiler Design Lab Record

(212CSE3304)

Student Name	: ………………………………………………………
Register Number	: ………………………………………………………
Slot / Section	: ………………………………………………………..

KALASALINGAM
ACADEMY OF RESEARCH AND EDUCATION
(DEEMED TO BE UNIVERSITY)
Under sec. 3 of UGC Act 1956. Accredited by NAAC with "A++" Grade

Anand Nagar, Krishnankoil, Srivilliputtur (Via), Virudhunagar (Dt) - 626126, Tamil Nadu | info@kalasalingam.ac.in | www.kalasalingam.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BONAFIDE CERTIFICATE

Bonafide record of work done by

of **III Year / VI Semester** in **212CSE3304 / Automata and Compiler Design**

during **Even** Semester in the Academic Year **2024-2025**

Staff In-charge

Submitted to the End Semester Practical Examination held at Kalasalingam Academy of Research and Education, Krishnankoil on ---------------------------

REGISTER NUMBER

INTERNAL EXAMINER **EXTERNAL EXAMINE**

<u>EXPERIMENT EVALUATION SUMMARY</u>

S.No.	Date	Experiment	Marks (100)	Faculty Signature
1		Design a DFA and NFA, Recognize an input string from Automata		
2		Automata Translation using JFLAP		
3		Automata and Regular Expression Translation using JFLAP		
4		Proving Language not to be Regular		
5		Installing Lex, Simple Lex Program, Implementation of Lexical Analyzer		
6		Compute First and Follow Function using Predictive Parsing		
7		Construction of Predictive Parsing Table		
8		Construction of SLR Parsing Table		
9		Generation of 3-address code		
10		Implementation of 3-address code		
11		Generation of Target Code		
12		Implementations of Optimization Techniques		

COURSE OBJECTIVES

- Understand the different mathematical models of computation.
- Design Finite Automata and recognize the Regular Expression and Languages.
- Explore the principles, algorithms, and data structures involved in the design and construction of compilers.

COURSE OUTCOMES(COS)

CO1: Understand the different forms of proof and construct Finite Automata, Deterministic Finite Automata and Non-Deterministic Finite Automata.

CO2: Evaluate Regular Expression and Languages using Finite Automata and their types.

CO3: Understand the different phases of compilers and Evaluate Lexical Analysis.

CO4: Apply Various Parsing Techniques to the Context Free Grammar (CFG).

CO5: Create the Various Code Generation Schemes and apply the various optimization techniques for the generated code.

CO6: Analyze the Various Mathematical Computational Machine Model using JFLAP simulation tools.

CO7: Implement all the Compiler Phases in programming language efficiently

ASSESSMENT METHOD:

S.No	Assessment	Split up
1	Internal Assessment (15 marks)	Regular Lab Exercises (5)
		Model Lab (10)
2	External Assessment (15 marks)	Record (5)
		End Semester program and output (10)

LIST OF EXPERIMENTS

S.No.	Experiment Details	Number of Periods	Cumulative Number of Periods
1	Design a DFA and NFA, Recognize an input string from Automata	2	2
2	Automata Translation using JFLAP	2	4
3	Automata and Regular Expression Translation using JFLAP	2	6
4	Proving Language not to be Regular	2	8
5	Installing Lex, Simple Lex Program, Implementation of Lexical Analyzer	4	12
6	Compute First and Follow Function using Predictive Parsing	2	14
7	Construction of Predictive Parsing Table	2	16
8	Construction of SLR Parsing Table	4	20
9	Generation of 3-address code	2	22
10	Implementation of 3-address code	2	24
11	Generation of Target Code	2	26
12	Implementations of Optimization Techniques	4	30

RUBRICS FOR INDIVIDUAL EXPERIMENTS

Modules	Unacceptable	Fair	Acceptable	Excellent
Level of understanding	Very little background information provided or information is incorrect (1)	Some introductory information, but still missing some major points (4)	Introduction is nearly complete, missing some minor points (7)	Introduction complete, provides all necessary background principles for the experiment (10)
Algorithm	Several major aspects of the exercise are missing, student displays a lack of understanding about how to write an algorithm (2)	Algorithm misses one or more major aspects of carrying out the exercise (6)	Algorithm is nearly complete, missing some minor points (10)	Algorithm is complete and well-written; provides all necessary background principles for the exercise (15)
Design principles & Program Logic	Missing several important experimental details or not written in proper logic in program (10)	Written in proper logic, still missing some important details (20)	Written in proper logic, important details are covered, some minor details missing (30)	Program Logic is well written, all details are covered (40)
Output	Output contains errors or are poorly constructed, (2)	Partial output; missing some important output features (4)	Output is good but some minor problems or could still be improved (7)	Output is excellent (10)
Discussion/ Viva	Answered for less than 40% of the questions indicating a lack of understanding of results (2)	Answered for 60% of the questions. but incomplete understanding of results is still evident (4)	Answered for 60% of the questions. Still need some improvements (7)	Answered for more than 90% of the questions correctly, good understanding of results is conveyed (10)

Ex.No.1 DESIGN A DFA AND NFA, RECOGNIZE AN INPUT STRING FROM AUTOMATA

DATE:

Aim:

To construct the Deterministic Finite Automata (DFA) and Non-Deterministic Finite Automata (NFA) for the given language and to verify the given string is in the language of automata or not using JFLAP.

Algorithm/ Procedure:

1. **Deterministic Finite Automata:**

Problems on DFA: (Any Three)

1. **Construct the DFA, that accepts the below languages over ∑={a,b} (Draw the Transition diagram using JFLAP and recognize the string in Language)**

   (i) |w| =2 (String length exactly equals to 2)

   (ii) |w| <=2 (String length with atmost 2)

   (iii) |w| >=2 (String length with atleast 2)

   (iv) |w| mod 2 =0 (String length divisible by 2 or even string length)

   (v) |w| mod 2 =1 (String length not divisible by 2 or odd string length)

   (vi) $N_a(w)$= 0 mod 2 (Even numbers of a's or No. of a's divisible by 2)

   (vii) $N_a(w)$= 1 mod 2 (odd numbers of a's or No. of a's not divisible by 2)

   (viii) $N_b(w)$= 0 mod 2 (Even numbers of b's or No. of b's divisible by 2)

   (ix) $N_b(w)$= 1 mod 2 (odd numbers of b's or No. of b's not divisible by 2)

   (x) $N_a(w)$= 0 mod 3 (No. of a's divisible by 3)

   (xi) $N_a(w)$= 1 mod 3 (No. of a's divisible by 3 with remainder 1)

   (xii) $N_a(w)$= 2 mod 3 (No. of a's divisible by 3 with remainder 2)

   (xiii) $N_b(w)$= 0 mod 3 (No. of b's divisible by 3)

   (xiv) $N_b(w)$= 1 mod 3 (No. of b's divisible by 3 with remainder 1)

   (xv) $N_b(w)$= 2 mod 3 (No. of b's divisible by 3 with remainder 2)

   (xvi) $N_a(w)$ and $N_b(w)$ both are even

   (xvii) $N_a(w)$ and $N_b(w)$ both are odd

   (xviii) $N_a(w)$ is odd and $N_b(w)$ is even

   (xix) $N_a(w)$ is even and $N_b(w)$ is odd

Solution (Q.No.)

Transition Diagram and Transition Table **Transition Function**

Solution (Q.No.)

Transition Diagram and Transition Table

Transition Function

Solution (Q.No.)

Transition Diagram and Transition Table

Transition Function

2. Non-Deterministic Finite Automata (NFA):

Problem:(Any Three)

2. Construct the NFA that accepts the below language over ∑={a,b}

(i) Strings start with ‘a’

(ii) Strings ends with ‘ab’

(iii) Strings that have 3 consecutive a’s

(iv) String that has ‘ab’ as sub-string

Solution (Q.No.)

Transition Diagram and Transition Table **Transition Function**

Solution (Q.No.)

Transition Diagram and Transition Table **Transition Function**

Solution (Q.No.)

Transition Diagram and Transition Table **Transition Function**

VIVA QUESTIONS

1. State 5 tuples of Automata.
2. Define DFA.
3. Define NFA.
4. Compare DFA and NFA.
5. Which is more powerful? NFA or DFA.

RESULT:

EVALUATION

Assessment	**Marks Scored**
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) **(Technical – 5 and Communications - 5)**	
Total (100)	

Ex.No.2 **AUTOMATA TRANSLATION USING JFLAP**

(NFA TO DFA)

DATE:

Aim:

To convert the Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA) using JFLAP.

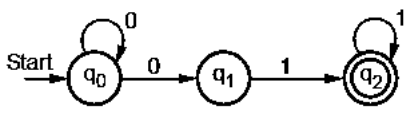
Algorithm/ Procedure:

<u>Steps in conversion of NFA to DFA:</u>

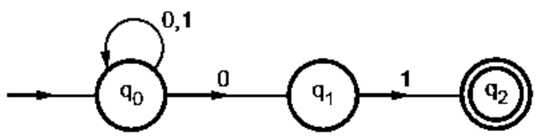
Problems:

1. **Convert the following NFA to DFA (Any one)**

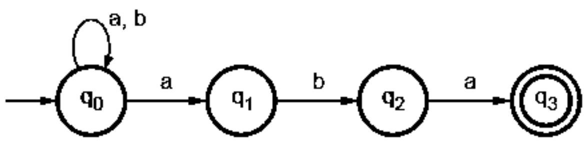
(i)



(ii)



(iii)



Solution (Q.No.)

<u>Step 1:</u> Construct Transition Table:

<u>Step 2:</u> Identify the new states from transition table

<u>Step 3:</u> Find the next state for new states in step 2 using transition function (Repeat until no more new states are found)

<u>Step 4:</u> Construct the new Transition Table and find the reachable states

<u>Step 5:</u> Draw the DFA using the table obtained in Step 4

VIVA QUESTIONS

1. Which has a greater number of states? NFA or DFA.
2. List the steps involved in NFA to DFA conversion.
3. Which state in NFA becomes the starting state in DFA?
4. How will you find the final state of resulting DFA?
5. List the three different ways of representing finite automata.

RESULT:

EVALUATION

Assessment	**Marks Scored**
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) **(Technical – 5 and Communications - 5)**	
Total (100)	

Ex.No.3 **AUTOMATA AND REGULAR EXPRESSION TRANSLATION USING JFLAP**

DATE:

Aim:

(i) To convert the Regular Expression (RE) to Finite Automata and Vice-Versa using JFLAP.

(ii) To convert the Regular Expression (RE) to Epsilon NFA using JFLAP.

Algorithm/ Procedure:

<u>Regular Expression-Definition:</u>

<u>Closure:</u>

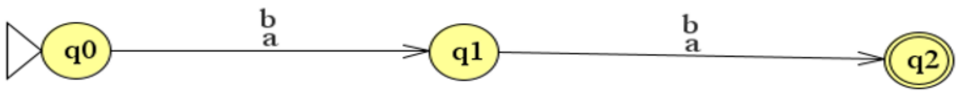
Kleene Closure:

Positive Closure:

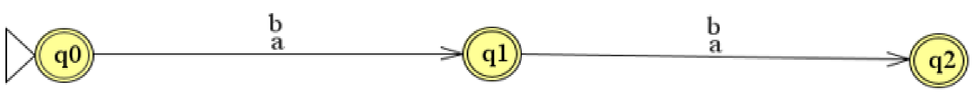
(i) <u>Finite Automata to Regular Expression:</u>

Problems:

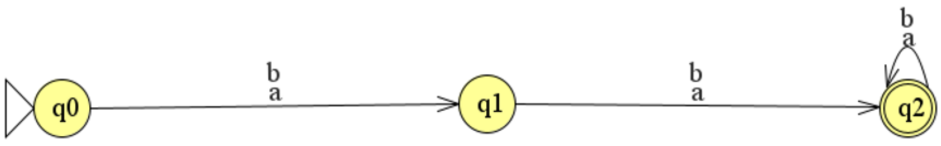
1. Convert the following Finite Automata to Regular Expression



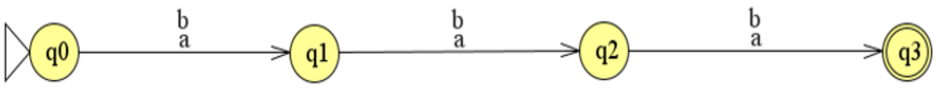
(i)



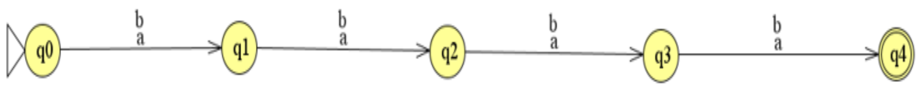
(ii)



(iii)



(iv)

(v) 

Solutions:

(i) **Language L= { }**

Language Definition:

Regular Expression:

(ii) **Language L= { }**

Language Definition:

Regular Expression:

(iii) **Language L= { }**

Language Definition:

Regular Expression:

(iv) **Language L= { }**

Language Definition:

Regular Expression:

(v) **Language L= { }**

Language Definition:

Regular Expression:

(ii) <u>Regular Expression to Epsilon NFA:</u>

Procedure:

Thomson Construction Method

(**Precedence order:** Closure, Union, Concatenation)

(i) Union Case:

(ii) Concatenation Case:

(iii) Closure Case:

<u>Kleene Closure:</u>

<u>Positive Closure:</u>

Problem:

Convert the Regular Expression (a+b)*abb to epsilon NFA.

Solution:

<u>Union Case: (a+b)</u>

R1= a **R2=b**

R3= (R1 + R2) => (a+b)

Closure Case: (a+b) * [Kleene Closure]

R4 = (R3) *

Concatenation Case: (abb)

R5=abb

Final Epsilon NFA:

R6= R4.R5

VIVA QUESTIONS

1. Define Regular Expression.
2. List the operations of RE.
3. Construct the RE for the language that accepts even number of a’s over ∑= {a, b}
4. Construct the FA for the Regular Expression (a+b) *ab
5. Define Epsilon Closure.

RESULT:

EVALUATION

Assessment	Marks Scored
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) **(Technical – 5 and Communications - 5)**	
Total (100)	

Ex.No.4 **PROVING LANGUAGE NOT TO BE REGULAR**

DATE:

Aim:

To prove the given language is regular or not with Pumping Lemma using JFLAP.

Algorithm/ Procedure:

<u>Pumping Lemma -Definition:</u>

Problems:

1. **Prove that the given Language is not Regular using Pumping Lemma**

   $L = \{a^n b^n \mid n > 0\}$

Solution:

Language L ={ }

Case 1: (V contains Only 'a')

Case 2: (V contains Only 'b')

Case 3: (V contains both ‘a’ and ‘b’)

2. Prove that the given Language is not Regular using Pumping Lemma

$L = \{a^n b^k c^{n+k} \mid n,k > 0\}$

Solution:

Language L ={ }

Case 1: (V contains Only 'a')

Case 2: (V contains Only 'b')

Case 3: (V contains both ‘a′ and ‘b′)

Case 3: (V contains only ‘c′)

VIVA QUESTIONS

1. State Pumping Lemma for Regular Language.
2. Pumping Lemma is used to prove a language as ------------------------------.
3. Define Regular Language with example
4. L ={$a^n b^m$ | m, n >0} is Regular or not?
5. List the properties of regular language.

RESULT:

EVALUATION

Assessment	**Marks Scored**
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) **(Technical – 5 and Communications - 5)**	
Total (100)	

Ex.No.5 INSTALLING LEX, SIMPLE LEX PROGRAM, IMPLEMENTATION OF LEXICAL ANALYZER

DATE:

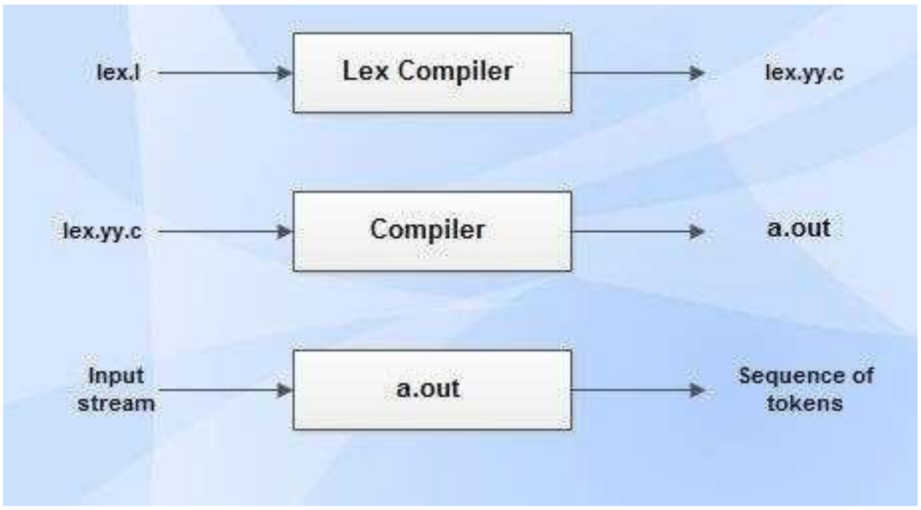
AIM

To write a program to identify tokens in the source program using LEX tool.

THEORY:

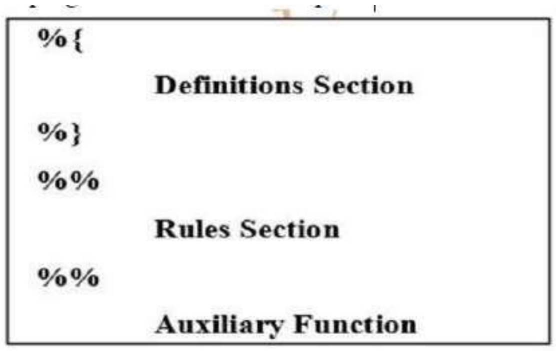
Introduction:

LEX stands for Lexical Analyzer. LEX is a UNIX utility which generates the lexical analyzer. LEX is a tool for generating scanners. Scanners are programs that recognize lexical patterns in text. These lexical patterns (or regular expressions) are defined in a particular syntax. A matched regular expression may have an associated action. This action may also include returning a token. When Lex receives input in the form of a file or text, it attempts to match the text with the regular expression. It takes input one character at a time and continues until a pattern is matched. If a pattern can be matched, then Lex performs the associated action (which may include returning a token). If, on the other hand, no regular expression can be matched, further processing stops and Lex displays an error message. Lex and C are tightly coupled. A lex file (files in Lex have the .l extension eg: first.l) is passed through the lex utility, and produces output files in C (lex.yy.c). The program lex.yy.c basically consists of a transition diagram constructed from the regular expressions of first.l These file is then compiled object program a.out, and lexical analyzer transforms an input streams into a sequence of tokens as show in figure. To generate a lexical analyzer two important things are needed. Firstly it will need a precise specification of the tokens of the language. Secondly it will need a specification of the action to be performed on identifying each token



LEX Specifications:

The Structure of lex programs consists of three parts:



```
%{
            Definitions Section
%}
%%
            Rules Section
%%
            Auxiliary Function
```

Definition Section:

The Definition Section includes declarations of variables, start conditions regular definitions, and manifest constants (A manifest constant is an identifier that is declared to represent a constant e.g.# define PIE 3.14).

C code: Any indented code between %{ and %} is copied to the C file. This is typically used for defining file variables, and for prototypes of routines that are defined in the code segment.

Definitions: A definition is very much like # define cpp directive. For example□

letter [a-zA-Z]+

digit [0-9]+

These definitions can be used in the rules section: one could start a rule

{letter}{printf("n Wordis = %s",yytext);}

State definitions: If a rule depends on context, it‟s possible to introduce states and incorporate those in the rules. A state definition looks like %s STATE, and by default a state INITIAL is already given.

Rule Section:

Second section is for translation rules which consist of regular expression and action with respect to it. The translation rules of a Lex program are statements of the form:

p1 {action 1}

p2 {action 2}

p3 {action 3}

… …

… …

pn {action n}

Where, each p is a regular expression and each action is a program fragment describing what action the lexical analyzer should take when a pattern p matches a lexeme. In Lex the actions are written in C.

Auxiliary Function (User Subroutines):

Third section holds whatever auxiliary procedures are needed by the actions. If the lex program is to be used on its own, this section will contain a main program. If you leave this section empty, you will get the default main as follow:

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

In this section we can write a user subroutines its option to user e.g. yylex() is a unction automatically get called by compiler at compilation and execution of lex program or we can call that function from the subroutine section.

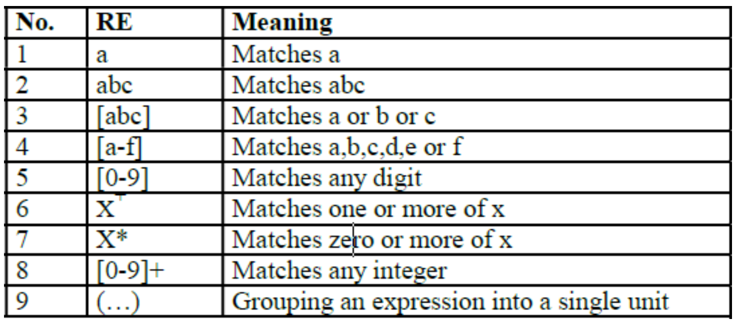
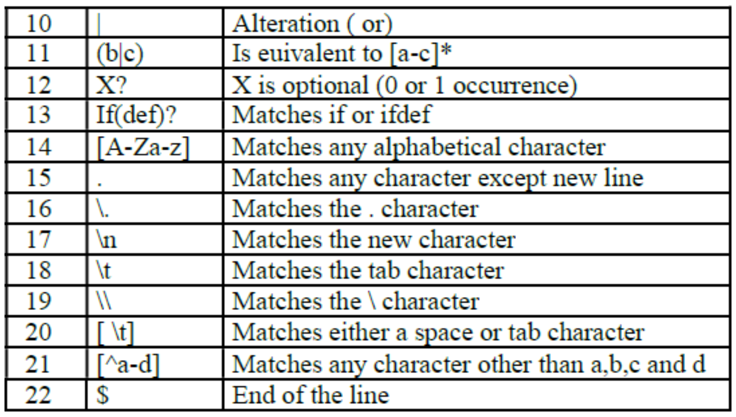
2. Built - in Functions:

No.	Function	Meaning
1	yylex()	The function that starts the analysis. It is automatically generated by Lex.
2	yywrap()	This function is called when end of file (or input) is encountered. If yywrap() returns 0, the scanner continues scanning, while if it returns 1 the scanner returns a zero token to report the end of file.
3	yyless(int n)	This function can be used to push back all but first „n" characters of the read Token.
4	yymore()	This function tells the lexer to append the next token to the current token.
5	yyerror()	This function is used for displaying any error message.

2. Built - in Variables:

No.	Variables	Meaning
1	yyin	Of the type FILE*. This point to the current file being parsed by the lexer. It is standard input file that stores input source program.
2	yyout	Of the type FILE*. This point to the location where the output of the lexer will be written. By default, both yyin and yyout point to standard input and output.
3	yytext	The text of the matched pattern is stored in this variable (char*) i.e. When lexer matches or recognizes the token from input token the lexeme stored in null terminated string called yytext. OR This is global variable which stores current token
4	yyleng	Gives the length of the matched pattern. (yyleng stores the length or number of character in the input string)The value in yyleng is same as strlen() functions.
5	yylineno	Provides current line number information. (May or may not be supported by the lexer.)
6	yylval	This is a global variable used to store the value of any token.

Regular Expression:

No.	RE	Meaning
1	a	Matches a
2	abc	Matches abc
3	[abc]	Matches a or b or c
4	[a-f]	Matches a,b,c,d,e or f
5	[0-9]	Matches any digit
6	X^+	Matches one or more of x
7	X*	Matches zero or more of x
8	[0-9]+	Matches any integer
9	(…)	Grouping an expression into a single unit
10	|	Alteration (or)
11	(b|c)	Is euivalent to [a-c]*
12	X?	X is optional (0 or 1 occurrence)
13	If(def)?	Matches if or ifdef
14	[A-Za-z]	Matches any alphabetical character
15	.	Matches any character except new line
16	\.	Matches the . character
17	\n	Matches the new character
18	\t	Matches the tab character
19	\\	Matches the \ character
20	[\t]	Matches either a space or tab character
21	[^a-d]	Matches any character other than a,b,c and d
22	$	End of the line

Steps to Execute the program:

```
$ lex filename.l (eg: first.l)
$cc lex.yy.c–ll or gcc lex.yy.c–ll
$./a .out
```

ALGORITHM:

1. Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%.

The format is as follows:

```
definitions
%%
rules
%%
user_subroutines
```

2. In definition section, the variables make up the left column, and their definitions make up the right column. Any C statements should be enclosed in %{..}%. Identifier is defined such that the first letter of an is alphabet and remaining letters are alphanumeric.

3. In rules section, the left column contains the pattern to be recognized in an input file to yylex(). The right column contains the C program fragment executed when that pattern is recognized. The various patterns are keywords, operators, new line

character, number, string, identifier, beginning and end of block, comment statements, preprocessor directive statements etc.

4. Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.

5. When yylex() matches a string in the input stream, it copies the matched text to an external character array, yytext, before it executes any actions in the rules section.

6. In user subroutine section, main routine calls yylex(). yywrap() is used to get more input.

7. The lex command uses the rules and actions contained in file to generate a program, lex.yy.c, which can be compiled with the cc command. That program can then receive input, break the input into the logical pieces defined by the rules in file, and run program fragments contained in the actions in file.

PROGRAM: (lexid.l)

INPUT:

Lexyi.txt

```
int a=10;
```

OUTPUT:

```
C:/FlexWindow:/EditPlusPortable> lex lexid.l
C:/FlexWindow:/EditPlusPortable> cc lex.yy.c
C:/FlexWindow:/EditPlusPortable> a


Identifier 1
Digit 1
Keyword 1
Operator 1
Delimiter 1
```

PROGRAM TO RECOGNIZE A VALID ARITHMETIC EXPRESSION THAT USESOPERATOR +, - , * AND / USING YACC

ALGORITHM:

1. Declare the variables which are used for C programs in the declaration section %{...%}.
2. Define the tokens, precedence and associativity of operators used in yacc.
3. Include the pattern (Context Free Grammar) in the transition rule section for validating the expression between %%..%%
4. In main function get the expression from the user for validating it.
5. Call the yyparse() function to parse the given expression and it construct the LALR parsing table using the grammar defined in transition rule .
6. Then call the yylex() function, it get the current token and store its value in yylval variable and it is repeated until the value for given expression is computed.
7. Then it validates the expression with constructed LALR parser.
8. Print the expression is VALID if the expression given by user is derived by the grammar, else print INVALID.
9. Stop the program.

PROGRAM:

OUTPUT:

```
C:\Flex Windows\EditPlusPortable>yacc -d arithval.y

C:\Flex  Windows\EditPlusPortable>cc y.tab.c

C:\Flex Windows\EditPlusPortable>a

ENTER AN EXPRESSION TO VALIDATE 5+9

VALID

4+6

VALID

5-

INVALID
```

VIVA QUESTIONS

1. Define LEX.
2. Give the syntax of LEX program.
3. List the built-in functions in LEX.
4. List the built-in variables in LEX.
5. Give the regular expression for the identifiers.

RESULT:

EVALUATION

Assessment	**Marks Scored**
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) **(Technical – 5 and Communications - 5)**	
Total (100)	

Ex.No.6 COMPUTE FIRST AND FOLLOW FUNCTION USING PREDICTIVE PARSING

DATE:

AIM

To find first and follow of a given context free grammar

THEORY:

<u>Why FIRST?</u>

To avoid backtracking in parsing we need to calculate first.

S -> cAd

A -> bc|a

And the input string is “cad”.

If the compiler would have come to know in advance, that what is the “first character of the string produced when a production rule is applied”, and comparing it to the current character or token in the input string it sees, it can wisely take decision on which production rule to apply.

Computing the Function *FIRST*

If X is Grammar Symbol, then First (X) will be –

- If X is a terminal symbol, then FIRST(X) = {X}
- If $X \rightarrow \varepsilon$, then FIRST(X) = {ε}
- If X is non-terminal & $X \rightarrow a\,\alpha$, then FIRST (X) = {a}
- If $X \rightarrow Y_1, Y_2, Y_3$, then FIRST (X) will be

(a) If Y is terminal, then

FIRST (X) = FIRST (Y_1, Y_2, Y_3) = {Y_1}

(b) If Y_1 is Non-terminal and

If Y_1 does not derive to an empty string i.e., If FIRST (Y_1) does not contain ε then,

FIRST (X) = FIRST (Y_1, Y_2, Y_3) = FIRST(Y_1)

(c) If FIRST (Y_1) contains ε, then.

FIRST (X) = FIRST (Y_1, Y_2, Y_3) = FIRST(Y_1) – {ε} ∪ FIRST(Y_2, Y_3)

Similarly, FIRST (Y_2, Y_3) = {Y_2}, If Y_2 is terminal otherwise if Y_2 is Non-terminal then

- FIRST (Y_2, Y_3) = FIRST (Y_2), if FIRST (Y_2) does not contain ε.
- If FIRST (Y_2) contain ε, then
- FIRST (Y_2, Y_3) = FIRST (Y_2) – {ε} ∪ FIRST (Y_3)

<u>Why FOLLOW?</u>

The parser faces one more problem. Let us consider below grammar to understand this problem.

```
A -> aBb
B -> c | ε
```

And suppose the input string is “ab” to parse.

As the first character in the input is a, the parser applies the rule A->aBb.

```
    A
  / | \
 a  B   b
```

Now the parser checks for the second character of the input string which is b, and the Non-Terminal to derive is B, but the parser can’t get any string derivable from B that contains b as first character.

But the Grammar does contain a production rule B -> ε, if that is applied then B will vanish, and the parser gets the input “ab”, as shown below. But the parser can apply it only when it knows that the character that follows B in the production rule is same as the current character in the input.

In RHS of A -> aBb, b follows Non-Terminal B, i.e. FOLLOW(B) = {b}, and the current input character read is also b. Hence the parser applies this rule. And it is able to get the string “ab” from the given grammar.

```
     A               A
   / | \           /   \
 a   B   b   =>   a     b
     |
     ε
```

So FOLLOW can make a Non-terminal vanish out if needed to generate the string from the parse tree.

Computing the Function *FOLLOW*

1. First, put $ (the end of input marker) in Follow(S) (S is the start symbol)
2. Suppose there is a production rule of A → aBB, (where a can be a whole string) then everything in FIRST(B) except for ε is placed in FOLLOW(B).
3. Suppose there is a production rule of A → aB, then everything in FOLLOW(A) is in FOLLOW(B)
4. Suppose there is a production rule of A → aBC, where FIRST(C) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)

Program:

SAMPLE INPUT

Consider the expression grammar ,

A -> aAaB

B -> bBaB

A-> a

B -> b

SAMPLE OUTPUT

First (A) = a
First (B) = b
Follow (A)= $, a, b
Follow (B) =$, a, b

VIVA QUESTIONS

1. What is the need of calculating FIRST?
2. What is the need for FOLLOW?
3. Compare Top Down and Bottom-up parser?
4. Does Top-Down Parser handle Left Recursive Grammar?
5. State the rule for eliminating Left Recursion.

RESULT:

EVALUATION

Assessment	**Marks Scored**
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) **(Technical – 5 and Communications - 5)**	
Total (100)	

Ex.No.: 7 **CONSTRUCTION OF PREDICTIVE PARSING TABLE**

DATE:

AIM

To implement the Predictive Parsing Table using any programing language.

ALGORITHM:

Program:

Output :

VIVA QUESTIONS

1. Does Predictive Parser can handle Left Recursive Grammar?
2. What do you mean Left Recursive Grammar?
3. Does Predictive Parser can handle Left Factoring Grammar?
4. What do you mean Left Factoring Grammar?
5. What do you mean LL [1] Grammar?

RESULT:

EVALUATION

Assessment	**Marks Scored**
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) **(Technical – 5 and Communications - 5)**	
Total (100)	

Ex.No.:8 CONSTRUCTION OF SLR PARSING TABLE

DATE:

AIM

To write a program for implementing SLR bottom up parser for the given grammar

THEORY:

LR parsers:

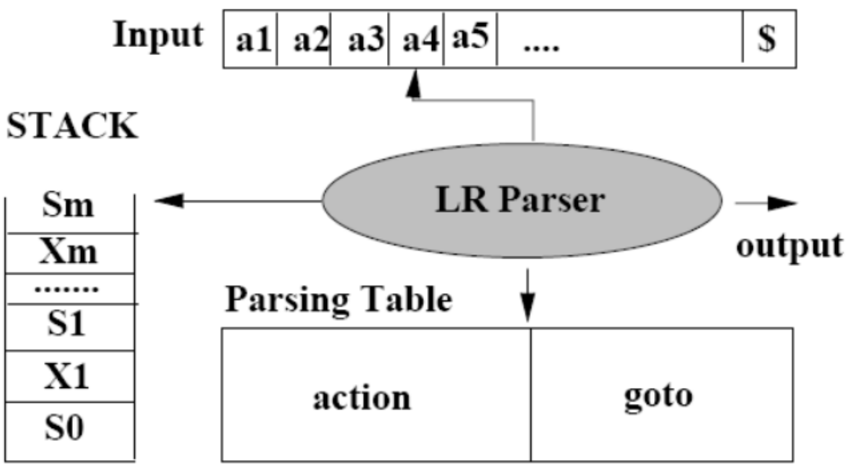


Fig: LR Parser

It is an efficient bottom-up syntax analysis technique that can be used to parse large classes of context free grammar is called LR(0) parsing.

L stands for the left to right scanning

R stands for rightmost derivation in reverse

0 stands for no. of input symbols of lookahead

Advantages of LR parsing:

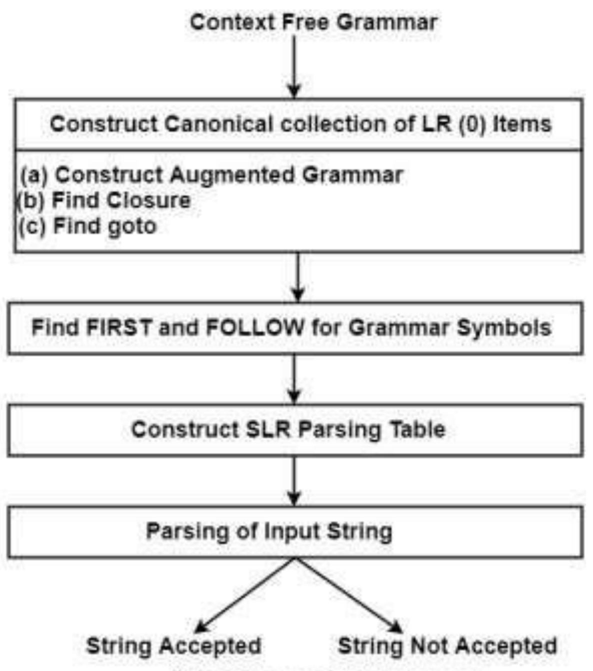
- It recognizes virtually all programming language constructs for which CFG can be written
- It is able to detect syntactic errors
- It is an efficient non-backtracking shift reducing parsing method.

Types of LR parsing methods:

1. SLR
2. CLR
3. LALR

SLR Parser:

SLR is simple LR. It is the smallest class of grammar having few number of states. SLR is very easy to construct and is similar to LR parsing. The only difference between SLR parser and LR(0) parser is that in LR(0) parsing table, there's a chance of 'shift reduced' conflict because we are entering 'reduce' corresponding to all terminal states. We can solve this problem by entering 'reduce' corresponding to FOLLOW of LHS of production in the terminating state. This is called SLR (1) collection of items



Context Free Grammar
Construct Canonical collection of LR (0) Items
(a) Construct Augmented Grammar
(b) Find Closure
(c) Find goto
Find FIRST and FOLLOW for Grammar Symbols
Construct SLR Parsing Table
Parsing of Input String
String Accepted
String Not Accepted

Working of SLR Parser

Steps for constructing the SLR parsing table:

1. Writing augmented grammar
2. LR (0) collection of items to be found
3. Find FOLLOW of LHS of production
4. Defining 2 functions: goto[list of terminals] and action[list of non-terminals] in the parsing table

EXAMPLE – Construct LR parsing table for the given context-free grammar

S–>AA

A–>aA|b

Solution:

STEP1 – Find augmented grammar

The augmented grammar of the given grammar is:-

S'–>.S [0th production]

S–>.AA [1st production]

A–>.aA [2nd production]

A–>.b [3rd production]

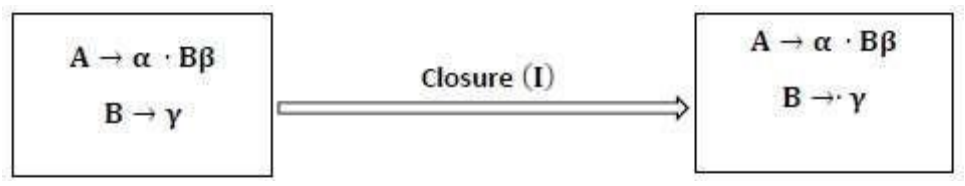
STEP2 – Find LR(0) collection of items

Below is the figure showing the LR(0) collection of items. We will understand everything one by one.

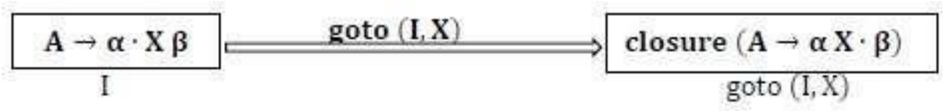
Closure – For a Context-Free Grammar G, if I is the set of items or states of grammar G, then

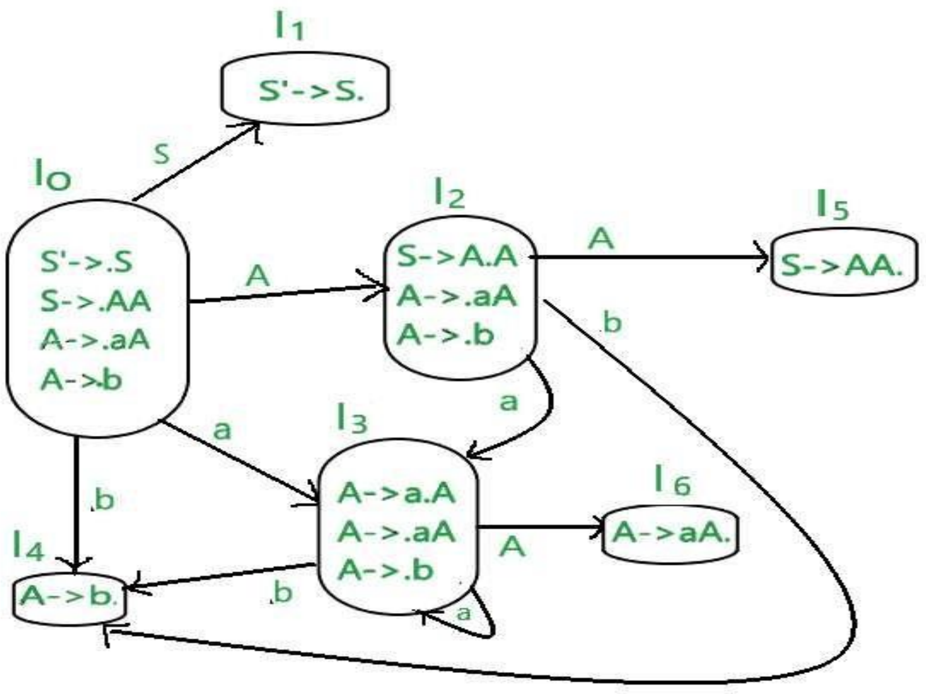
- Every item in I is in the closure (I).
- If rule A → α. B β is a rule in closure (I) and there is another rule for B such as B → γ

then closure (I) will consist of A → α. Bβ and B → . γ



goto (I, X) – If there is a production A → α · X β in I then goto (I, X) is defined as closure of the set of items of A → α X · β where I is set of items and X is grammar symbol (non-terminal).





STEP3 – Find FOLLOW of LHS of production

FOLLOW(S)=$

FOLLOW(A)=a,b,$

STEP 4-

Defining 2 functions: goto[list of non-terminals] and action[list of terminals] in the parsing table. Below is the SLR parsing table.

Algorithm

Input – An Augmented Grammar G′

Output – SLR Parsing Table

Method

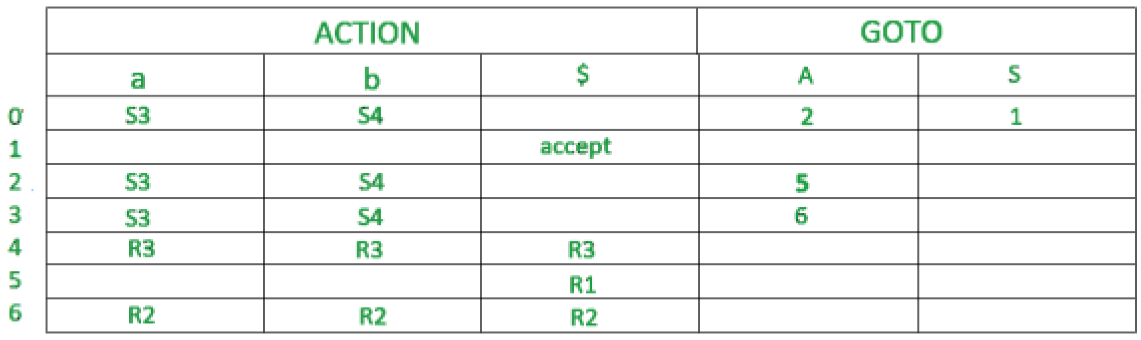
- Initially construct set of items

C = {I_0, I_1, I_2 … … I_n} where C is a set of LR (0) items for Grammar.

- Parsing actions are based on each item or state I_1.

Various Actions are –

- If A → α · a β is in I_i and goto (I_i, a) = I_j then set Action [i, a] = shift j".
- If A → α · is in I_i then set Action [i, a] to "reduce A → α" for all symbol a, where a ∈ FOLLOW (A).
- If S′ → S · is in I_i then the entry in action table Action [i, $] = accept".
- The goto part of the SLR table can be filled as– The goto transition for the state i is considered for non-terminals only. If goto (I_i, A) = I_j then goto [i, A] = j
- All entries not defined by rules 2 and 3 are considered to be "error. "



	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S3	S4		5	
3	S3	S4		6	
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

Program:

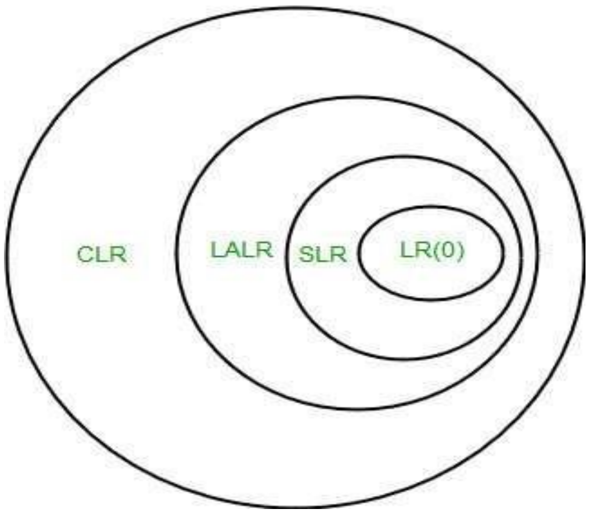
OUTPUT:

```
Enter any Stringv a+a*a$

0                   a+a*a$
0a5                 +a*a$
0F3                 +a*a$
0T2                 +a*a$
0E1                 +a*a$
0E1+6               a*a$
0E1+6a5             *a$
0E1+6F3             *a$
0E1+6T9             *a$
0E1+6T9*7           a$
0E1+6T9*7a5         $
0E1+6T9*7F10        $
0E1+6T9             $
0E1                 $
Given String is accept
```

NOTE:

1. Even though CLR parser does not have RR conflict but LALR may contain RR conflict.
2. If number of states LR(0) = n1,
   number of states SLR = n2,
   number of states LALR = n3,
   number of states CLR = n4 then,
   n1 = n2 = n3 <= n4



VIVA QUESTIONS

1. Define LR Parser.
2. State the advantages of LR Parser.
3. List the types of LR Parser.
4. Compare LR and LL Parser.
5. What do you mean by GOTO operation?

RESULT:

EVALUATION

Assessment	**Marks Scored**
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) **(Technical – 5 and Communications - 5)**	
Total (100)	

Ex.No.:9 **CONSTRUCT THE THREE ADDRESS CODE FOR THE GIVEN EXPRESSION**

DATE:

AIM

To write program to construct the three-address code for the given expression

THEORY:

Three address code

- Three-address code is an intermediate code. It is used by the Code Optimizer.
- In three-address code, the given expression is broken down into several separate instructions. These instructions can easily translate into assembly language.
- Each Three address code instruction has at most three operands. It is a combination of assignment and a binary operator.

They use maximum three addresses to represent any statement. They are implemented as a record with the address fields**.**

General Form-

In general, Three Address instructions are represented as-

```
a = b op c
```

Here,

- a, b and c are the operands.
- Operands may be constants, names, or compiler generated temporaries.
- op represents the operator.

Examples-

Examples of Three Address instructions are-

- a = b + c
- c = a x b

Common Three Address Instruction Forms-

The common forms of Three Address instructions are-

1. Assignment Statement-

```
x = y op z and x = op y
```

Here,

- x, y and z are the operands.
- op represents the operator.

It assigns the result obtained after solving the right side expression of the assignment operator to the left side operand.

2. Copy Statement-

```
x = y
```

Here,

- x and y are the operands.
- = is an assignment operator.

It copies and assigns the value of operand y to operand x.

3. Conditional Jump-

```
If x relop y goto X
```

Here,

- x & y are the operands.
- X is the tag or label of the target statement.
- relop is a relational operator.

If the condition “x relop y” gets satisfied, then-

- The control is sent directly to the location specified by label X.
- All the statements in between are skipped.

If the condition “x relop y” fails, then-

- The control is not sent to the location specified by label X.
- The next statement appearing in the usual sequence is executed.

4. Unconditional Jump-

```
goto X
```

Here, X is the tag or label of the target statement.

On executing the statement,

- The control is sent directly to the location specified by label X.
- All the statements in between are skipped.

5. Procedure Call-

```
param x call p return y
```

Here, p is a function which takes x as a parameter and returns y.

Example:

1. Write Three Address Code for the following expression-

(a x b) + (c + d) – (a + b + c + d)

Three Address Code for the given expression is-

(1) T1 = a x b

(2) T2 = uminus T1

(3) T3 = c + d

(4) T4 = T2 + T3

(5) T5 = a + b

(6) T6 = T3 + T5

(7) T7 = T4 – T6

2. Write Three Address Code for the following expression

If A < B and C < D then t = 1 else t = 0

(1) If (A < B) goto (3)

(2) goto (4)

(3) If (C < D) goto (6)

(4) t = 0

(5) goto (7)

(6) t = 1

(7)

Program:

Output

VIVA QUESTIONS

1. Define Three Address Code with Example.
2. What is the need for intermediate code?
3. What are the types of three address code?
4. Is three address code machine dependent? Justify.
5. Write the three address code for the following.

   If A < B then 1 else 0

RESULT:

EVALUATION

Assessment	**Marks Scored**
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) **(Technical – 5 and Communications - 5)**	
Total (100)	

Ex.No.:10 IMPLEMENTATION OF 3-ADDRESS CODE

(TRIPLES, QUADRUPLES AND INDIRECT TRIPLES)

DATE:

AIM

To write a program to implement a code that converts the given expression to triples, quadruples and indirect triples

THEORY:

The commonly used representations for implementing Three Address Code are-

1. Quadruples
2. Triples
3. Indirect Triples

1. Quadruple –

It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage –

- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

Disadvantage –

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

Example:

a + b x c / e ↑ f + b x c

Three Address Code for the given expression is-

T1 = e ↑ f

T2 = b x c

T3 = T2 / T1

T4 = b x a

T5 = a + T3

T6 = T5 + T4

Location	Op	Arg1	Arg2	Result
(0)	↑	e	f	T1
(1)	x	b	c	T2
(2)	/	T2	T1	T3
(3)	x	b	a	T4
(4)	+	a	T3	T5
(5)	+	T5	T4	T6

3. **Triples –**

This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Disadvantage –

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Location	Op	Arg1	Arg2
(0)	↑	e	f
(1)	x	b	c
(2)	/	(1)	(0)
(3)	x	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

3. Indirect Triples-

This representation is an enhancement over triples representation.

- It uses an additional instruction array to list the pointers to the triples in the desired order.
- This representation makes use of pointer to the listing of all references to computations which is made separately and stored.
- Thus, instead of position, pointers are used to store the results.

Advantages

- It allows the optimizers to easily re-position the sub-expression for producing the optimized code
- Its similar in utility as compared to quadruple representation but requires less space than it.
- Temporaries are implicit and easier to rearrange code.

Statement	
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

Location	**Op**	**Arg1**	**Arg2**
(0)	↑	e	f
(1)	x	b	e
(2)	/	(1)	(0)
(3)	x	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

Program:

VIVA QUESTIONS

1. Compare Triples and Indirect Triples.
2. What are the ways of representation of Intermediate code? (Postfix notation, Syntax tree, Three-address code)
3. State the advantages and disadvantages of Quadruples.
4. Translate the following expression to quadruple, triple and indirect triple-
   a = b x – c + b x – c
5. State the advantages of Indirect Triples.

RESULT:

EVALUATION

Assessment	Marks Scored
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) **(Technical – 5 and Communications - 5)**	
Total (100)	

Ex .No 11 **GENERATION OF TARGET CODE**

DATE:

AIM

To write a C program for implementing back end of the compiler which takes three address codes as input and produces 8086 assembly language instruction.

THEORY:

A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.

Code Generator determines the values that are to be stored in the registers.

• For example: consider the three-address statement a := b+c It can have the following sequence of codes:

ADD Rj, Ri Cost = 1

ADD c, Ri Cost = 2

MOV c, Rj Cost = 3

ADD Rj, Ri

Register and Address Descriptors:

• A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.

• An address descriptor stores the location where the current value of the name can be found at run time.

A code-generation algorithm:

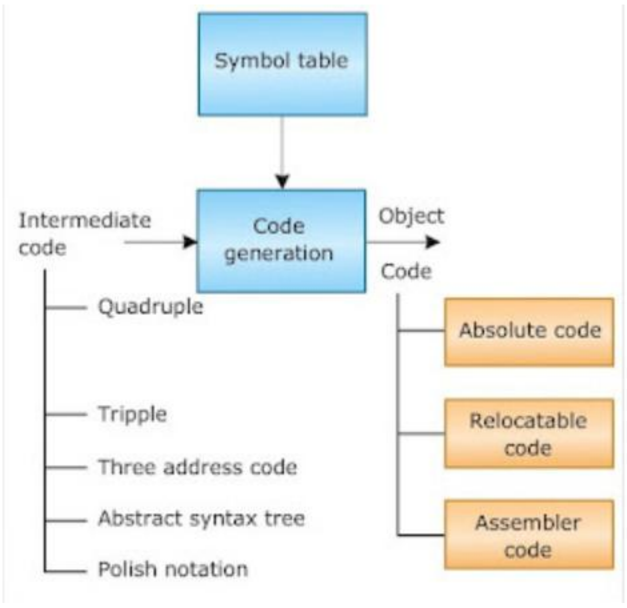
The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form x : = y op z, perform the following actions:

1. Invoke a function getreg to determine the location L where the result of the computation y op z should be stored.

2. Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction MOV y' , L to place a copy of y in L.

3. Generate the instruction OP z' , L where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z



Example:

Generating Code for Assignment Statements:

The assignment statement d:= (a-b) + (a-c) + (a-c) can be translated into the following sequence of three address code:

1. t:= a-b
2. u:= a-c
3. v:= t +u
4. d:= v+u

Code sequence for the example is as follows:

Statement	Code Generated	Register descriptor Register empty	Address descriptor
t:= a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u:= a - c	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
v:= t + u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R1
d:= v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

Target Machine

- The target computer is a type of byte-addressable machine. It has 4 bytes to a word.
- The target machine has n general purpose registers, R0, R1,...., Rn-1. It also has two-address instructions of the form:

1. op source, destination

Where, op is used as an op-code and source and destination are used as a data field.

- It has the following op-codes:

  ADD (add source to destination)

  SUB (subtract source from destination)

  MOV (move source to destination)

- The source and destination of an instruction can be specified by the combination of registers and memory location with address modes.

MODE	FORM	ADDRESS	EXAMPLE	ADDED COST
absolute	M	M	Add R0, R1	1
register	R	R	Add temp, R1	0
indexed	c(R)	C+ contents(R)	ADD 100 (R2), R1	1

indirect register	*R	contents(R)	ADD * 100	0
indirect indexed	*c(R)	contents(c+ contents(R))	(R2), R1	1
literal	#c	c	ADD #3, R1	1

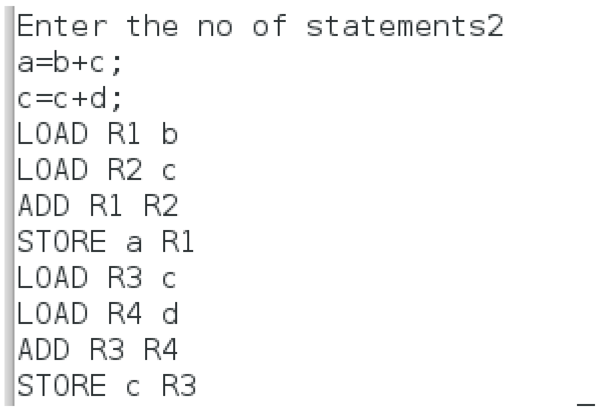
- Here, cost 1 means that it occupies only one word of memory.
- Each instruction has a cost of 1 plus added costs for the source and destination.
- **Instruction cost = 1 + cost is used for source and destination mode.**

ALGORITHM:

1. Start the program
2. Include the necessary header files.
3. Get the number of statements from the user.
4. For each variable allocate a separate register using Load or Move Instructions LD R,a or Mov R,a
5. If the expression contains operator "+", then generate the assembly code as ADD
6. If the expression contains operator "-", then generate the assembly code as SUB
7. If the expression contains operator "*", then generate the assembly code as MUL
8. If the expression contains operator "/", then generate the assembly code as DIV
9. Result of the operand is stored to any variables ST x, R0.
10. Stop the program.

PROGRAM:

OUTPUT:



```
Enter the no of statements2
a=b+c;
c=c+d;
LOAD R1 b
LOAD R2 c
ADD R1 R2
STORE a R1
LOAD R3 c
LOAD R4 d
ADD R3 R4
STORE c R3
```

VIVA QUESTIONS

1. What is the purpose of code generator?
2. List the issues in code generator.
3. Compare Register and Address descriptor.
4. What do you mean by next use information?
5. Write the assembly code for the given expression and find the cost. C=a+b*6
6. Name the technique used for allocating registers efficiently.

   Linear Scan Algorithm

RESULT:

EVALUATION

Assessment	**Marks Scored**
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) **(Technical – 5 and Communications - 5)**	
Total (100)	

Ex .No 12 IMPLEMENTATIONS OF OPTIMIZATION TECHNIQUES

DATE:

AIM

To write a program to implement a code optimizer to perform possible optimization like dead code elimination, common sub expression elimination, etc,.

THEORY:

Reasons for Optimizing the Code

- Code optimization is essential to enhance the execution and efficiency of a source code.
- It is mandatory to deliver efficient target code by lowering the number of instructions in a program.

When to Optimize?

Code optimization is an important step that is usually performed at the last stage of development.

Role of Code Optimization

- It is the fifth stage of a compiler, and it allows you to choose whether or not to optimize your code, making it really optional.
- It aids in reducing the storage space and increases compilation speed.
- It takes source code as input and attempts to produce optimal code.
- Functioning the optimization is tedious; it is preferable to employ a code optimizer to accomplish the assignment.

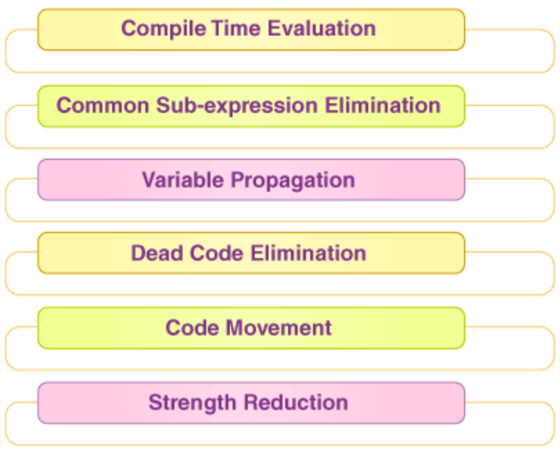
Different Types of Optimization

Optimization is classified broadly into two types:

- Machine-Independent
- Machine-Dependent

Machine-Independent Optimization

It positively affects the efficiency of intermediate code by transforming a part of code that does not employ hardware parts. It usually optimises code by eliminating tediums and removing unneeded code.



Machine-Dependent Optimization

After the target code has been constructed and transformed according to the target machine architecture, machine-dependent optimization is performed. It makes use of CPU registers and may utilise absolute rather than relative memory addresses. Machine-dependent optimizers work hard to maximise the perks of the memory hierarchy.

Loop Optimization

- Invariant code/Code Motion or Frequency Reduction
- Induction analysis
- Strength reduction

ALGORITHM:

PROGRAM:

OUTPUT:

```
enter no of values 5
left    a right:  9
left    b right: c+d
left    e right: c+d
left    f right: b+e
left    r right: f
intermediate Code
a=9
b=c+d
e=c+d
f=b+e
r=f

after dead code elimination
b       =c+d
e       =c+d
f       =b+e
r       =f
eliminate common expression
b       =c+d
b       =c+d
f       =b+b
r       =f
optimized code
b=c+d
f=b+b
r=f
```

RESULT:

VIVA QUESTIONS

1. State the role of Optimizer
2. Compare Machine dependent and independent optimizer.
3. List the machine dependent optimization techniques.
4. List the machine independent optimization techniques.
5. What do you mean by common sub expression? Give Example.
6. What is code motion and Dead code?

EVALUATION

Assessment	Marks Scored
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) **(Technical – 5 and Communications - 5)**	
Total (100)	