Laboratory Manual
For
Compiler Design Lab

Department of Computer Science & Engineering

LIST OF EXPERIMENTS

Sr. No.	Title of Experiment	Corresponding CO
1	Implementation of lexical analyzer for IF statement and Arithmetic expression.	C320.1
2	Construction of NFA from Regular Expression	C320.1
3	Construction of DFA from NFA	C320.1
4	Construction of recursive descent parsing for the grammar	C320.2
5	Write a C program to implement operator precedence parsing	C320.2
6	Implementation of shift reduce parsing algorithm	C320.2
7	Design a code optimizer for implementing constant propagation	C320.3
8	Write a program to perform loop unrolling for code optimization	C320.3
9	Implementation Code Generator	C320.4
Content beyond syllabus		
10	Write a C program for implementing the functionalities of predictive parser	C320.2
11	Write a C program for constructing of LL (1) parsing	C320.2
12	Write a program to Design LALR Bottom up Parser	C320.2

INTRODUCTION

The language Processors comprises assemblers, compilers and interpreters. It deals with the recognition the translation, and the execution of formal languages It is closely related to compiler construction. Compiler is system software that converts high level language into low level language. We human beings can't program in machine language (low level lang.) understood by computers so we program in high level language and compiler is the software which bridges the gap between user and computer.

Students will gain the knowledge of

- Lexical Analysis for decomposing a character stream into lexical units
- Syntax analysis for recovering context-free structure from an input stream, error correction
- Semantic analysis for enforcing non-context-free requirements, attribute grammars.
- Semantic definition, for describing the meaning of a phrase (we rely on interpretive definitions)
- Implementation of programming concepts, control structures
- Data representation, implementation of data structures
- Partial evaluation, for removing interpretation overhead
- Code generation: instruction selection, register allocation
- Further semantic analysis: document validation, type checking.

PREFACE

This lab is as a part of B.Tech VI semester for CSE students. Compiler design principles provide an in-depth view of translation and optimization process. This lab enables the students to practice basic translation mechanism by designing complete translator for a mini language and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end with recommended systems/software requirements following the university prescribed textbooks. The expected outcomes from the students are:

1. By this laboratory, students will understand the practical approach of how a compiler works.

2. This will enable him to work in the development phase of new computer languages in industry.

DO'S AND DONT'S

DO's

1. Conform to the academic discipline of the department.
2. Enter your credentials in the laboratory attendance register.
3. Read and understand how to carry out an activity thoroughly before coming to the laboratory.
4. Ensure the uniqueness with respect to the methodology adopted for carrying out the experiments.
5. Shut down the machine once you are done using it.

DONT'S

1. Eatables are not allowed in the laboratory.
2. Usage of mobile phones is strictly prohibited.
3. Do not open the system unit casing.
4. Do not remove anything from the computer laboratory without permission.
5. Do not touch, connect or disconnect any plug or cable without your faculty/laboratory technician's permission.

GENERAL SAFETY INSTRUCTIONS

1. Know the location of the fire extinguisher and the first aid box and how to use them in case of an emergency.

2. Report fires or accidents to your faculty /laboratory technician immediately.

3. Report any broken plugs or exposed electrical wires to your faculty/laboratory technician immediately.

4. Do not plug in external devices without scanning them for computer viruses.

Details Of The Experiments Conducted

(To Be Used By The Students In Their Records)

S.No	Date Of Conduction	Expt. No	Title Of The Experiment	Page No.	Marks Awarded (10)	Faculty Signature With Remark
1						
2						
3						
4						
5						
6						
7						
8						
9						
10						

Compiler Design Lab (RCS 652)

Name	
Roll No.	
Section- Batch	

INDEX

Experiment No.	Experiment Name	Date of Conduction	Date of Submission	Faculty Signature

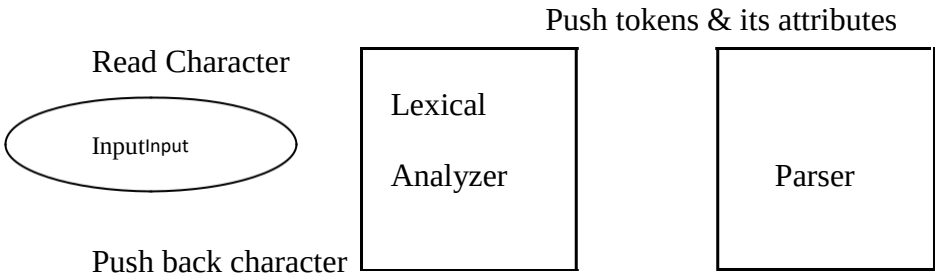
EXPERIMENT #1

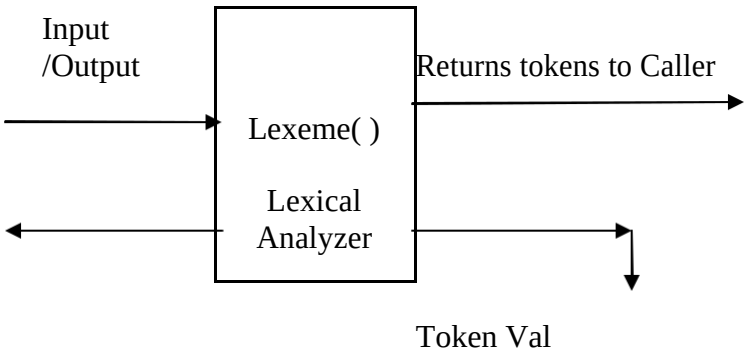
AIM: Implementation of lexical analyzer for IF statement & arithmetic expression.

DESCRIPTION:

Lexical analysis or scanning is the process where the stream of characters making up the source program is read from left-to-right and grouped into tokens. Tokens are sequences of characters with a collective meaning. There are usually only a small number of tokens for a programming language: constants (integer, double, char, string, etc.), operators (arithmetic, relational, logical), punctuation, and reserved words.

INTERFACE OF LEXICAL ANALYZER:





PRE EXPERIMENT

To develop a Lexical Analyzer one should possess the basic knowledge of the followings.

Q1. What is the Lexical Analyser's Basic Mechanism?

Q2. What are lexemes, Patterns and Tokens? Specification of Tokens in form of Regular expressions?

Q3. What is the functioning of Deterministic Finite Automata?

Q4. What are the additional functionalities of a Lexical Analyser?

ALGORITHM

Step 1: Declare the necessary variables.

Step2:Declare an array and store the keywords in that array

Step 3: Open the input file in read open

Step 4:read the string from the file till the end of file.

Step 4.1:If the first character in the strings # then print that string as header file.

Step 4.2:If the string matches with any of the keywords print .That string isa keyword Step 4.3:If the string matches with operator and special Symbols print the corresponding message Step 4.4 : If the string is not a keyword then prints that as an identifier.

INPUT & OUTPUT:

Input:

```
 Enter Program ,$ for termination:
 { int a[3],t1,t2;
t1=2; a[0]=1; a[1]=2; a[t1]=3;
 t2=-(a[2]+t1*6)/(a[2]-t1);
if t2>5 then
                    print(t2);
          else
                  { int t3;
                    t3=99;
                    t2=-
                    25;
                     print(-t1+t2*t3); /* this is a comment on 2
                   lines */ } endif
 }
$
```

Output:

```
 Variables : a[3] t1 t2 t3
 Operator : - + * / >
 Constants : 2 1 3 6 5 99 -25
 Keywords : int if then else endif
 Special Symbols : , ; ( ) { }
```

Comments : this is a comment on 2 lines

POST EXPERIMENT

Q1. How are comments handled at lexical analysis phase? Write the steps for removing the comments?

Q2. Write a program to reduce multiple new lines of a given string into a single stream of characters by replacing a new line into a blank space & also reducing contiguous blank spaces into a single space.

Q3. Design a regular grammar and implement Deterministic Finite Automaton that recognize c variable declaration made up only of following legal combinations of following keywords.

int, float, for while, if

EXPERIMENT #2

AIM: To write a C program to construct a Non Deterministic Finite Automata (NFA) from Regular Expression.

DESCRIPTION:

A nondeterministic finite automaton (NFA), or nondeterministic finite state machine, does not need to obey these restrictions. In particular, every DFA is also an NFA.

Using the subset construction algorithm, each NFA can be translated to an equivalent DFA, i.e. a DFA recognizing the same formal language. Like DFAs, NFAs only recognize regular languages. Sometimes the term NFA is used in a narrower sense, meaning an automaton that properly violates an above restriction i.e. that is *not* a DFA.

PRE EXPERIMENT

Q1. What do you understand by NFA?

Q2. Define DFA with diagram.

Q3. Can we simulate NFA to DFA.

ALGORITHM:

Step1: Start the Program.

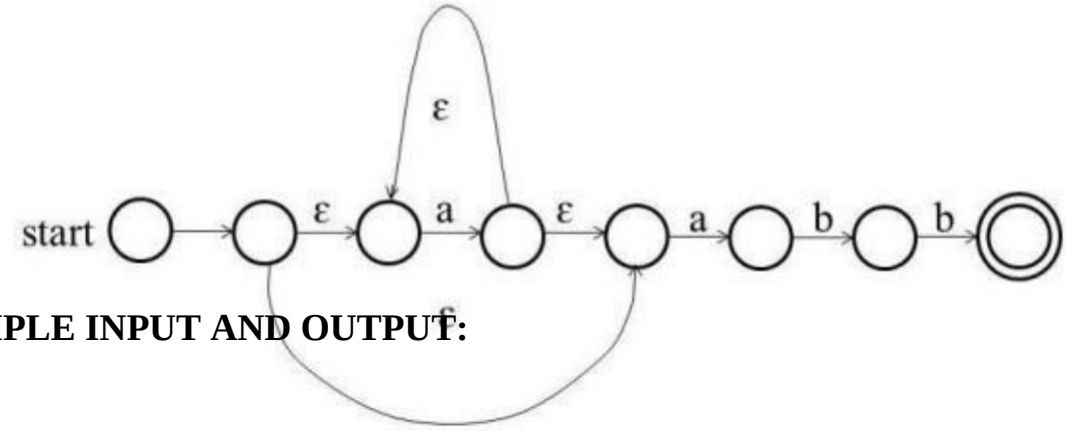
Step2: Enter the regular expression R over alphabet E.

Step3: Decompose the regular expression R into its primitive components

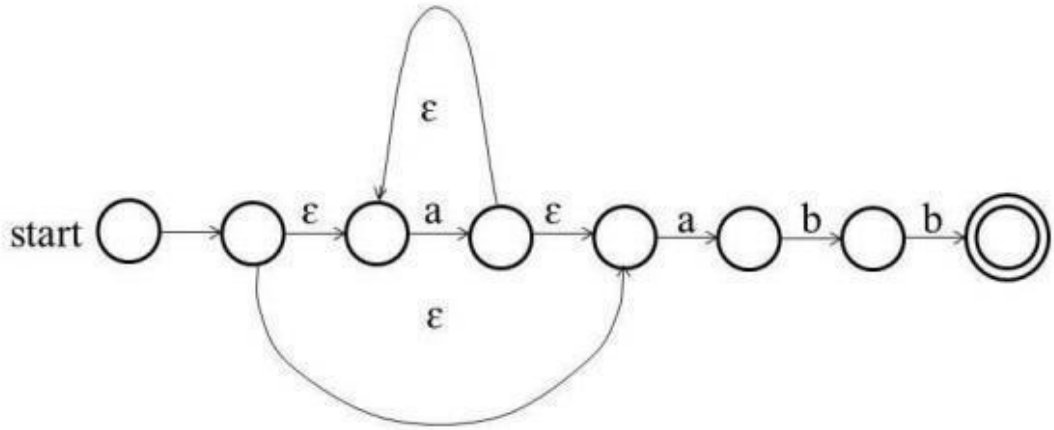
Step4: For each component construct finite automata.

Step5: To construct components for the basic regular expression way that corresponding to that way compound regular expression.

Step6: Stop the Program



SAMPLE INPUT AND OUTPUT:



Result: The above C program was successfully executed and verified

POST EXPERIMENT:

Q1. Differentiate the uses of Regular Expressions ad Grammar

Q2. Define Context Free Grammar.

Q3. What is Closure?

EXPERIMENT #3

AIM: To write a C program to construct a DFA from the given NFA.

DESCRIPTION:

A Deterministic finite automaton (DFA) can be seen as a special kind of NFA, in which for each state and alphabet, the transition function has exactly one state. Thus clearly every formal language that can be recognized by a DFA can be recognized by an NFA.

PRE EXPERIMENT

Q1. Differentiate between NFA and DFA

Q2. What do you mean by Transition Table?

Q3. What is Finite Automata?

ALGORITHM

Step1: Start the program.

Step2: Accept the number of state A and B.

Step3: Find the E-closure for node and name if as A.

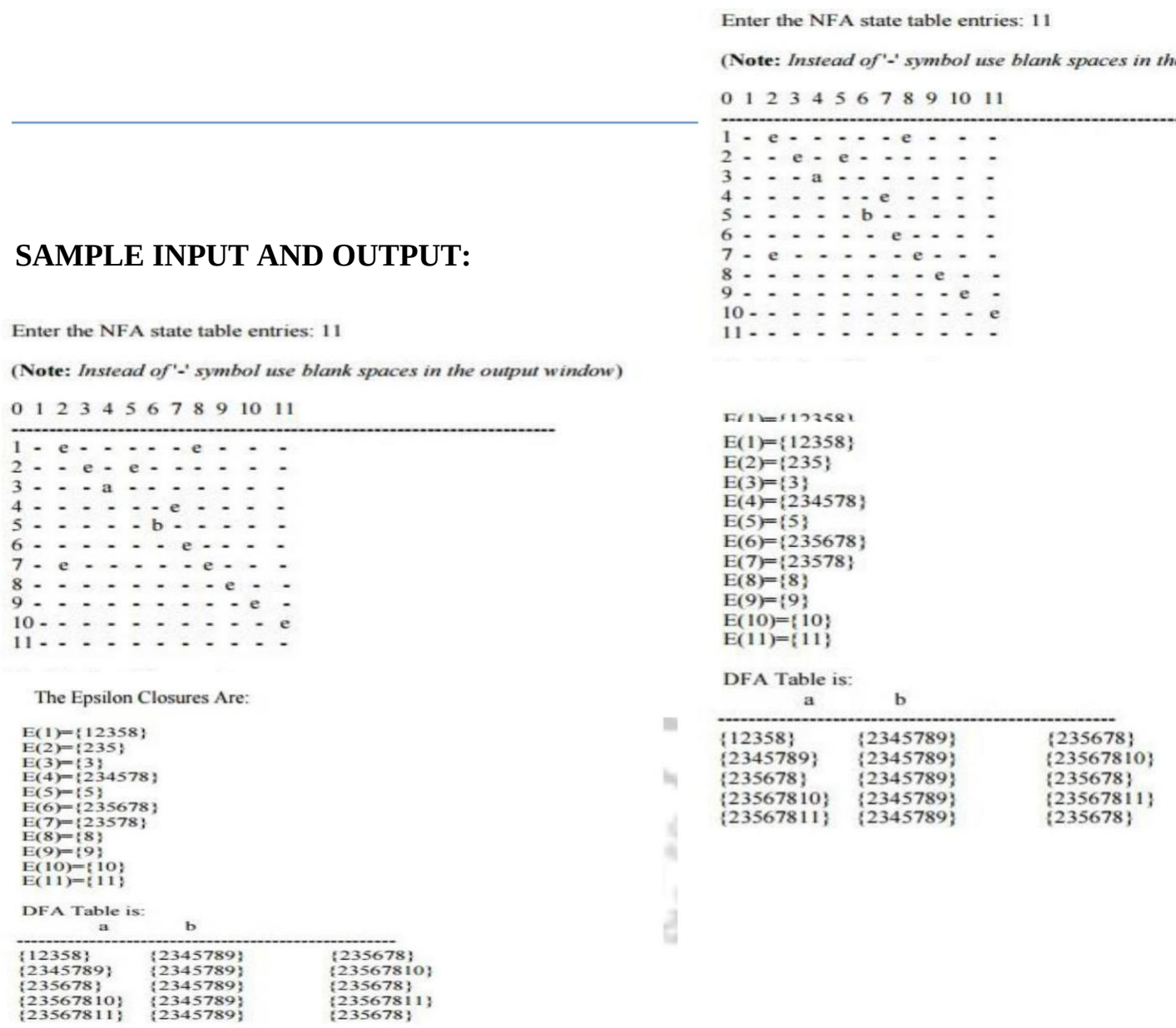
Step4: Find v(a,a) and (a,b) and find a state.

Step5: Check whether a number new state is obtained.

Step6: Display all the state corresponding A and B.

Step7: Stop the program.

SAMPLE INPUT AND OUTPUT:



Enter the NFA state table entries: 11

(**Note:** *Instead of '-' symbol use blank spaces in the output window*)

```
0 1 2 3 4 5 6 7 8 9 10 11
-------------------------------------------------------------------------
1 - e - - - - - e - - -
2 - - e - e - - - - - -
3 - - - a - - - - - - -
4 - - - - - - e - - - -
5 - - - - - b - - - - -
6 - - - - - - - e - - -
7 - e - - - - - - e - -
8 - - - - - - - - - e -
9 - - - - - - - - - - e
10 - - - - - - - - - - -
11 - - - - - - - - - - -
```

The Epsilon Closures Are:

```
E(1)={12358}
E(2)={235}
E(3)={3}
E(4)={234578}
E(5)={5}
E(6)={235678}
E(7)={23578}
E(8)={8}
E(9)={9}
E(10)={10}
E(11)={11}
```

DFA Table is:

```
          a          b
-------------------------------------------------------
{12358}      {2345789}      {235678}
{2345789}    {2345789}      {23567810}
{235678}     {2345789}      {235678}
{23567810}   {2345789}      {23567811}
{23567811}   {2345789}      {235678}
```

Enter the NFA state table entries: 11

(**Note:** *Instead of '-' symbol use blank spaces in th*

```
0 1 2 3 4 5 6 7 8 9 10 11
-------------------------------------------------------------------------
1 - e - - - - - e - - -
2 - - e - e - - - - - -
3 - - - a - - - - - - -
4 - - - - - - e - - - -
5 - - - - - b - - - - -
6 - - - - - - - e - - -
7 - e - - - - - - e - -
8 - - - - - - - - - e -
9 - - - - - - - - - - e
10 - - - - - - - - - - -
11 - - - - - - - - - - -
```

```
E(1)={12358}
E(2)={235}
E(3)={3}
E(4)={234578}
E(5)={5}
E(6)={235678}
E(7)={23578}
E(8)={8}
E(9)={9}
E(10)={10}
E(11)={11}
```

DFA Table is:

```
          a          b
-------------------------------------------------------
{12358}      {2345789}      {235678}
{2345789}    {2345789}      {23567810}
{235678}     {2345789}      {235678}
{23567810}   {2345789}      {23567811}
{23567811}   {2345789}      {235678}
```

Result: The above C program was successfully executed and verified

POST EXPERIMENT

Q1. What are the types of Finite Automata?

Q2. Define Recognizer.

Q3. Define Context Free Grammar.

EXPERIMENT # 4

AIM: Construction of recursive descent parsing for the following

grammar E->TE'

E'->+TE/@ "@ represents null character"

T->FT'

T`->*FT'/@

F->(E)/ID.

DESCRIPTION:

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.

PRE-EXPERIMENT

Q1. What is Recursive descent parser?

Q2. What type of parsing it is?

ALGORITHM:

Step 1: start.

Step 2: Declare the prototype functions E() , EP(),T(), TP(),F()

Step 3: Read the string to be parsed.

Step 4: Check the productions

Step 5: Compare the terminals and Non-terminals

Step 6: Read the parse string.

Step 7: stop the production

INPUT & OUTPUT

Recursive descent parsing for the following grammar

E->TE'

E'->+TE'/@

T->FT'

T'->*FT'/@

F->(E)/ID

Enter the string to be checked:

(a+b)*c

String is accepted

Recursive descent parsing for the following grammar

E->TE'

E'->+TE'/@

T->FT'

T'->*FT'/@

F->(E)/ID

Enter the string to be checked:

a/c+d

String is not accepted

POST EXPERIMENT

Q1. Recursive descent parsing may require backtracking of the input string?

Q2. What are the problems for parsing expressions by recursive descent parsing?

EXPERIMENT #5

AIM: - Write a C program to implement operator precedence parsing

DESCRIPTION:

An operator precedence grammar is a context-free grammar that has the property (among others) that no production has either an empty right-hand side or two adjacent nonterminals in its right-hand side. These properties allow precedence relations to be defined between the terminals of the grammar. A parser that exploits these relations is considerably simpler than more general-purpose parsers such as LALR parsers. Operator-precedence parsers can be constructed for a large class of context-free grammars.

PRE EXPERIMENT

Q1. What is user of operator grammar?

Q2. How to construct Precedence relation table?

ALGORITHM:

Step 1: Push # onto stack

Step 2: Read first input symbol and push it onto stack

Step 3: Do

Obtain OP relation between the top terminal symbol on the stack and the next input symbol

If the OP relation is < or =

i. Pop top of the stack into handle, include non-terminalsymbol if appropriate.

ii. Obtain the relation between the top terminal symbol on the stack and the leftmost terminal symbol in the handle .

iii. While the OP relation between terminal symbols is =o Do

- Pop top terminal symbol and associated non-terminal symbol on stack into handle.
- Obtain the OP relation between the top terminal symbol on the stack and the leftmost terminal symbol in the handle.

iv. Match the handle against the RHS of all productions v. Push N onto the stack

Step 4: Until end-of-file and only and N are on the stack

INPUT & OUTPUT:

INPUT:

Enter the arithmetic expression

(d*d)+d$

Output:

(d*d)+d$

Precedence input:$<(<d>*<d>)>+<d>$

$<(E*<d>)>+<d>$

$<(E*E)>+<E>$

$E+<E>$

$E+E$

Precedence input:$<+>$

E

success

POST EXPERIMENT

Q1. The input string: id1 + id2 * id3after inserting precedence relations becomes?

Q2. Having precedence relations allows identifying handles?

EXPERIMENT #6

AIM : Implementation of shift reduce parsing algorithm.

DESCRIPTION:

Shift-reduce parsing attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root. In other words, it is a process of "reducing" (opposite of deriving a symbol using a production rule) a string w to the start symbol of a grammar.

PRE EXPERIMENT:

Q1. What type parser it is?

Q2. What is BOOTOM_UP PARSER?

Q3. What is SHIFT?

Q4. What is REDUCE?

Q5. What is ACCEPT?

Q6. What is error?

ALGORITHM:

STEP1: Initial State: the stack consists of the single state, s0; ip points to the first character in w.

STEP 2: For top-of-stack symbol, s, and next input symbol, a case action of T[s,a]

STEP 3: **shift x:** (x is a STATE number) push a, then x on the top of the stack and advance ip to point to the next input symbol.

STEP 4: **reduce y:** (y is a PRODUCTION number) Assume that the production is of the form A ==> beta pop 2 * |beta| symbols of the stack.

STEP 5: At this point the top of the stack should be a state number, say s'. push A, then goto of T[s',A] (a state number) on the top of the stack.

INPUT & OUTPUT

Input:

Enter the GRAMMAR:

E□E+E

EE/EE

E*E E

a/b

Enter the input symbol: a+a

Output:
Stack implementation table

stack	input symbol	action
$	a+a$	--
$a	+a$	shift a
$E	+a$	E->a
$E+	a$	shift +
$E+a	$	shift a
$E+E	$	E->a
$E	$	E->E*E
$E	$	ACCEPT

POST EXPERIMENT

Q1. What are the two types of conflicts?

Q2. What are its limitations?

EXPERIMENT #7

AIM: Design a code optimizer for implementing constant propagation.

DESCRIPTION:

The algorithm we shall present basically tries to find for every statement in the program a mapping between variables, and values of N T ∪⊥ { , } . If a variable is mapped to a constant number, that number is the variables value in that statement on every execution. If a variable is mapped to T (top), its value in the statement is not known to be constant, and in the variable is mapped to ⊥ (bottom), its value is not initialized on every execution, or the statement is unreachable. The algorithm for assigning the mappings to the statements is an iterative algorithm, that traverses the control flow graph of the algorithm, and updates each mapping according to the mapping of the previous statement, and the functionality of the statement. The traversal is iterative, because non-trivial programs have circles in their control flow graphs, and it ends when a "fixed-point" is reached – i.e., further iterations don't change the mappings.

PRE EXPERIMENT

Q1. What is a compiler?

Q2. What is an interpreter?

ALGORITHM:

Step1: start

Step2: declare a=30,b=3,c

Step3: display result of propagation

Step4:end

INPUT & OUTPUT:

Input:

Enter the code

Output:

Result of constant propagation is 4

POST EXPERIMENT

Q1. What are the advantages of constant propagation?

EXPERIMENT #8

AIM : Write a program to perform loop unrolling for code optimization.

DESCRIPTION:

Loop unrolling transforms a loop into a sequence of statements. It is a parallelizing and optimizing compiler technique where loop unrolling us used to eliminate loop overhead to test loop control flow such as loop index values and termination conditions technique was also used to expose instruction-level parallelism i.e. Syntax tree.

PRE EXPERIMENT

Q1. Define loop unrolling

Q2. Give example of loop unrolling

ALGORITHM:

Step1: Start

Step2: Declare n

Step3: Enter n value

Step4: Loop rolling display countbit1 or move to next step 5

Step5: Loop unrolling display countbit2

Step6: End

Input & Output:

INPUT:

Enter a value: 5

OUTPUT:

Loop rolling output: 2

Loop unrolling output: 2

POST EXPERIMENT

Q1. What are the advantages of loop unrolling?

EXPERIMENT # 9

AIM: To write a C program to implement Simple Code Generator.

DESCRIPTION:

In computing, code generation is the process by which a compiler's code generator converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine.

The input to the code generator typically consists of a parse tree or an abstract syntax tree. The tree is converted into a linear sequence of instructions, usually in an intermediate language such as three-address code. Further stages of compilation may or may not be referred to as "code generation", depending on whether they involve a significant change in the representation of the program.

PRE EXPERIMENT:

Q1. Describe Code Generation Phases.

Q2. Explain Intermediate Code Generation Phase.

ALGORITHM:

Input: Set of three address code sequence.

Output: Assembly code sequence for three address codes (opd1=opd2, op, opd3).

Method:

Step1: Start

Step2: Get addresses code sequence.

Step3: Determine current location of 3 using address (for 1st operand).

Step4: If current location not already exist generate move (B,O).

Step5: Update address of A (for 2nd operand).

Step6: If current value of B and () is null, exist.

Step7: If they generate operator () A, 3 ADPR.

Step8: Store the move instruction in memory

Step9: Stop.

SAMPLE INPUT & OUTPUT:

Enter the Three Address Code:

a=b+c

c=a*c

Exit

The Equivalent Assembly Code is:

Mov R0, b

Add c, R0

Mov a, R0

Mov R1, a

Mul c, R1

Mov c, R1

Result: The above C program was successfully executed and verified

POST EXPERIMENT:

Q1. What are the Compiler Generator Tool?

Q2. What are the functions of Linker?

EXPERIMENT #10

AIM: Write a C program for implementing the functionalities of predictive parser

DESCRIPTION:

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. The goal of predictive parsing is to construct a top-down parser that never backtracks. To do so, we must transform a grammar in two ways:

1. Eliminate left recursion, and
2. Perform left factoring.

These rules eliminate most common causes for backtracking although they do not guarantee a completely backtrack-free parsing.

PRE EXPERIMENT

Q1. What is top-down parsing?
Q2. What are the disadvantages of brute force method?
Q3. What is context free grammar?
Q4. What is parse tree?
Q5. What is ambiguous grammar?
Q6. What are the derivation methods to generate a string for the given grammar?
Q7. What is the output of parse tree?

ALGORITHM:

STEP 1: X symbol on top of stack,a current input symbol

STEP 2: Stack contents and remaining input called parser configuration (initially $S on stack and complete inputstring)

2.1.If X=a=$ halt and announce success

2.2.If X=a≠$ pop X off stack advance input to next symbol

2.3.If X is a non-terminal use M[X ,a] which contains production

STEP 3: X->rhs or error replace X on stack with rhs or call error routine, respectively.

EX: X->UVW replace X with WVU(U on top) output

the production (or augment parse tree)

INPUT & OUTPUT:

The following is the predictive parsing table for the following grammar:

S->A

A->Bb

A->Cd

B->aB

B->@

C->Cc

C->@

Predictive parsing table is

A	b	c	d	$
S	S->A	S->A	S->A	S->A
A	A->Bb	A->Bb	A->Cd	A->Cd
B	B->aB	B->@	B->@	B->@
C	C->@	C->@	C->@	

POST EXPERIMENT:

Q1. What is Predictive parser?

Q2. How many types of analysis can we do using Parser?

Q3. What is Recursive Decent Parser?

Q4. How many types of Parsers are there?

Q5. What is LR Parser?22

EXPERIMENT #11

AIM: Write a C program for constructing of LL (1) parsing.

DESCRIPTION:

An LL parser is a top-down parser for a subset of context-free languages. It parses the input from Left to right, performing Leftmost derivationof the sentence. An LL parser is called an LL(*k*) parser if it uses *k* tokens of look-ahead when parsing a sentence. If such a parser exists for a certain grammar and it can parse sentences of this grammar without backtracking then it is called an LL(*k*) grammar. LL(*k*) grammars can generate more languages the higher the number *k* of lookahead tokens.

PRE EXPERIMENT:

Q1. What is LL(1)?

Q2. How to element left factoring for give grammar?

INPUT & OUTPUT:

Enter the input string:

i*i+i

Stack	**INPUT**
$bt	i*i+i$
$bcf	i*i+i$
$bci	i*i+i$

$bc	*i+i$
$bcf*	*i+i$
$bcf	i+i$
$bci	i+i$
$bc	+i$
$b	+i$
$bt+	+i$
$bt	i$
$bcf	i$
$ bci	i$
$bc	$
$b	$
$	$

success

POST EXPERIMENT:

Q1. How to find FIRST set of terminals for a given grammar?

Q2. How to find FOLLOW set of terminals for a given grammar?

Q3. Is a given parsing table LL(1) or not?

EXPERIMENT #12

AIM: - Write a program to Design LALR Bottom up Parser.

DESCRIPTION:

An LALR parser or Look-Ahead LR parser is a simplified version of a canonical LR parser, to parse (separate and analyze) a text according to a set of production rules specified by a formal grammar for a computer language. ("LR" means left-to-right, rightmost derivation.)

- Works on intermediate size of grammar
- Number of states are same as in SLR(1)

PRE EXPERIMENT

Q1. What is full form LALR?

Q2. What is CLOSURE?

Q3. What are Kernel Items?

Q4. What are Non-kernel Items?

Q5. What is an augmented grammar?

ALGORITHM:

STEP 1: Represent I_i by its CLOSURE, those items that are either the initial item[S′ → .S; eof] or do not have the . at the left end of the rhs.

STEP 2 : Compute shift, reduce, and goto actions for the state derived from I_i directly from CLOSURE(I_i)

INPUT & OUTPUT

Enter the input:

i*i+1

Output :

Stack	**input**
0	i*i+i$
0i5	*i+i$

0F3	*i+i$
0T2	*i+i$
0T2*7	i+i$
0T2*7i5	+i$
0T2*7i5F10	+i$
0T2	+i$
0E1	+i$
0E1+6	i$
0E1+6i5	$
0E1+6F3	$
0E1+6T9	$
0E1	$ accept the input*/

POST EXPERIMENT QUESTIONS

Q1. What is LALR parsing?

Q2. What is Shift reduced parser?

Q3. What are the operations of Parser?

Q4. What is the use of parsing table?

Q5. What is bottom up parsing?

REFERENCES:

(1) Aho,Ullman, Sethi ------ Compiler Construction.

(2) Allen I.Holub----------Compiler Construction.

APPENDIX-I:

SYLLABUS (AS PER AKTU)

1. Implementation of LEXICAL ANALYZER for IF STATEMENT
2. Implementation of LEXICAL ANALYZER for ARITHMETIC EXPRESSION
3. Construction of NFA from REGULAR EXPRESSION
4. Construction of DFA from NFA
5. Implementation of SHIFT REDUCE PARSING ALGORITHM
6. Implementation of OPERATOR PRECEDENCE PARSER
7. Implementation of RECURSIVE DESCENT PARSER
8. Implementation of CODE OPTIMIZATION TECHNIQUES
9. Implementation of CODE GENERATOR

****