LEXICAL ANALYZER

EX. NO. 1

SHUSHRUT KUMAR (RA1811028010049)

AIM: To write a program to implement a lexical analyzer.

ALGORITHM:

1. Start.
2. Get the input program from the file prog.txt.
3. Read the program line by line and check if each word in a line is a keyword, identifier, constant or an operator.
4. If the word read is an identifier, assign a number to the identifier and make an entry into the symbol table stored in sybol.txt.
5. For each lexeme read, generate a token as follows:

a. If the lexeme is an identifier, then the token generated is of the form <id, number>

b. If the lexeme is an operator, then the token generated is <op, operator>.

c. If the lexeme is a constant, then the token generated is <const, value>.

d. If the lexeme is a keyword, then the token is the keyword itself.

6. The stream of tokens generated are displayed in the console output.
7. Stop.

CONVERSION FROM REGULAR EXPRESSION TO NFA

EX. NO. 2

SHUSHRUT KUMAR (RA1811028010049)

AIM: To write a program for converting Regular Expression to NFA.

ALGORITHM:

1. Start
2. Get the input from the user
3. Initialize separate variables and functions for Postfix , Display and NFA
4. Create separate methods for different operators like +,*, .
5. By using Switch case Initialize different cases for the input
6. For ' . ' operator Initialize a separate method by using various stack functions do the same for the other operators like ' * ' and ' + '.
7. Regular expression is in the form like a.b (or) a+b
8. Display the output
9. Stop

CONVERSION OF NFA TO DFA

EX. NO. 3

SHUSHRUT KUMAR (RA1811028010049)

AIM: To write a program for converting NFA to DFA.

ALGORITHM:

```
1. Start
2. Get the input from the user
3. Set the only state in SDFA to “unmarked”.
4. while SDFA contains an unmarked state do:
a. Let T be that unmarked state
b. for each a in % do S = e-Closure(MoveNFA(T,a))
c. if S is not in SDFA already then, add S to SDFA (as an “unmarked” state)
d. Set MoveDFA(T,a) to S
5. For each S in SDFA if any s & S is a final state in the NFA then, mark S an a final state in
the DFA
6. Print the result.
7. Stop the program
```

ELIMINATION OF LEFT RECURSION

EX. NO. 4(a)

SHUSHRUT KUMAR (RA1811028010049)

AIM: A program for Elimination of Left Recursion.

ALGORITHM:

```
1. Start the program.
2. Initialize the arrays for taking input from the user.
3. Prompt the user to input the no. of non-terminals having left recursion and no. of
productions for these non-terminals.
4. Prompt the user to input the production for non-terminals.
5. Eliminate left recursion using the following rules:-
        A->Aα1| Aα2 | . . . . . |Aαm
        A->β1| β2| . . . . .| βn
        Then replace it by
        A-> βi A’ i=1,2,3,…..m
        A’-> αj A’ j=1,2,3,…..n
        A’-> Ɛ
6. After eliminating the left recursion by applying these rules, display the productions
without left recursion.
7. Stop.
```

LEFT FACTORING

EX. NO. 4(b)

SHUSHRUT KUMAR (RA1811028010049)

AIM : A program for implementation Of Left Factoring

ALGORITHM :

1. Start
2. Ask the user to enter the set of productions
3. Check for common symbols in the given set of productions by comparing with:

   A->aB1|aB2
4. If found, replace the particular productions with:

   A->aA'

   A'->B1 | B2|ε
5. Display the output
6. Exit

FIRST AND FOLLOW

AIM: To write a program to perform first and follow using any language.

ALGORITHM:

For computing the first:

1. If X is a terminal then FIRST(X) = {X}

Example: F -> I | id

We can write it as FIRST(F) -> { (, id)

2. If X is a non-terminal like E -> T then to get FIRSTI substitute T with other productions until you get a terminal as the first symbol

3. If X -> ε then add ε to FIRST(X).

For computing the follow:

1. Always check the right side of the productions for a non-terminal, whose FOLLOW set is being found. (never see the left side).

2. (a) If that non-terminal (S,A,B…) is followed by any terminal (a,b…,*,+,(,)…) , then add that terminal into the FOLLOW set.

(b) If that non-terminal is followed by any other non-terminal then add FIRST of other nonterminal into the FOLLOW set.

EXPERIMENT-7 PREDICTIVE PARSING

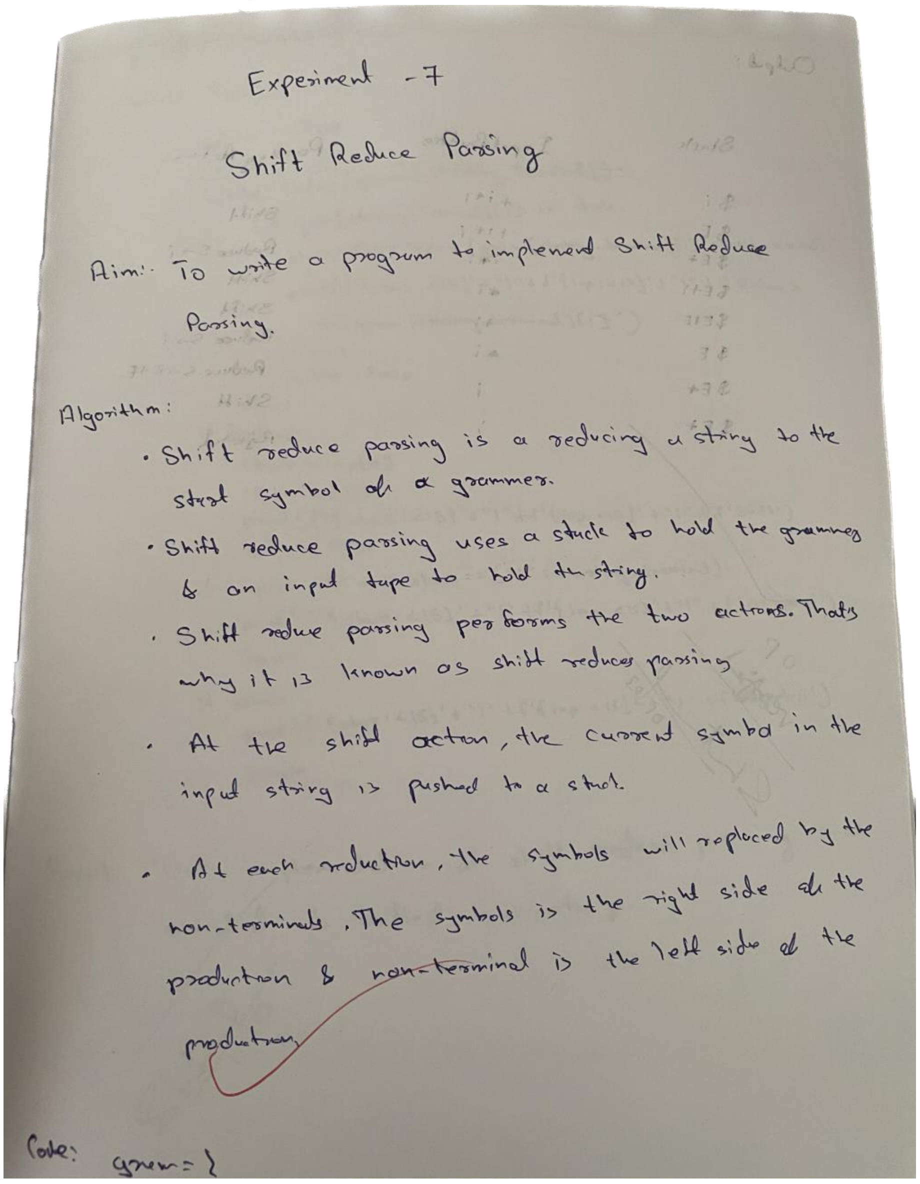
Aim: A program for Predictive Parsing

Algorithm:-

1. Start the program.
2. Initialize the required variables.
3. Get the number of coordinates and productions from the user.
4. Perform the following

```
for (each production A → α in G) {
for (each terminal a in FIRST(α))
add A → α to M[A, a];
if (ε is in FIRST(α))
for (each symbol b in FOLLOW(A))
add A → α to M[A, b];
```

5. Print the resulting stack.
6. Print if the grammar is accepted or not.
7. Exit the program.



Experiment -7

Shift Reduce Parsing

Aim:- To write a program to implement Shift Reduce Parsing.

Algorithm:

- Shift reduce parsing is a reducing a string to the start symbol of a grammer.
- Shift reduce parsing uses a stack to hold the grammer & an input tape to hold the string.
- Shift reduce parsing performs the two actions. That's why it is known as shift reduces parsing
- At the shift action, the current symbol in the input string is pushed to a stack.
- At each reduction, the symbols will replaced by the non-terminals. The symbols is the right side of the production & non-terminal is the left side of the production.

Code: gram = {

LAB EXP 8 : LEADING AND TRAILING

AIM : A program to implement Leading and Trailing

ALGORITHM :

1. For Leading, check for the first non-terminal.
2. If found, print it.
3. Look for next production for the same non-terminal.
4. If not found, recursively call the procedure for the single non-terminal present before the
comma or End Of Production String.
5. Include it's results in the result of this non-terminal.
6. For trailing, we compute same as leading but we start from the end of the production to the beginning.
7. Stop

Computation of LR(0) Items

Aim: A program to implement LR(0) items

Algorithm:-

1. Start.
2. Create structure for production with LHS and RHS.
3. Open file and read input from file.
4. Build state 0 from extra grammar Law S' -> S $ that is all start symbol of grammar and one Dot (.) before S symbol.
5. If Dot symbol is before a non-terminal, add grammar laws that this non-terminal is in Left Hand Side of that Law and set Dot in before of first part of Right Hand Side.
6. If state exists (a state with this Laws and same Dot position), use that instead.
7. Now find set of terminals and non-terminals in which Dot exist in before.
8. If step 7 Set is non-empty go to 9, else go to 10.
9. For each terminal/non-terminal in set step 7 create new state by using all grammar law that Dot position is before of that terminal/non-terminal in reference state by increasing Dot point to next part in Right Hand Side of that laws.
10. Go to step 5.
11. End of state building.
12. Display the output.
13. End.

Intermediate code generation – Postfix, Prefix

Aim: A program to implement Intermediate code generation – Postfix, Prefix.

Algorithm:-

1. Declare set of operators.
2. Initialize an empty stack.
3. To convert INFIX to POSTFIX follow the following steps
4. Scan the infix expression from left to right.
5. If the scanned character is an operand, output it.
6. Else, If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
7. Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack.
8. If the scanned character is an '(', push it to the stack.
9. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
10. Pop and output from the stack until it is not empty.
11. To convert INFIX to PREFIX follow the following steps
12. First, reverse the infix expression given in the problem.
13. Scan the expression from left to right.
14. Whenever the operands arrive, print them.
15. If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
16. Repeat steps 6 to 9 until the stack is empty

Intermediate code generation – Quadruple, Triple, Indirect triple

Aim: Intermediate code generation – Quadruple, Triple, Indirect triple

Algorithm:-

The algorithm takes a sequence of three-address statements as input. For each three address statements of the form a:= b op c perform the various actions. These are as follows: 1. Invoke a function getreg to find out the location L where the result of computation b op c should be stored.

2. Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction MOV y' , L to place a copy of y in L.
3. Generate the instruction OP z' , L where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptors.
4. If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of x : = y op z those register will no longer contain y or z.

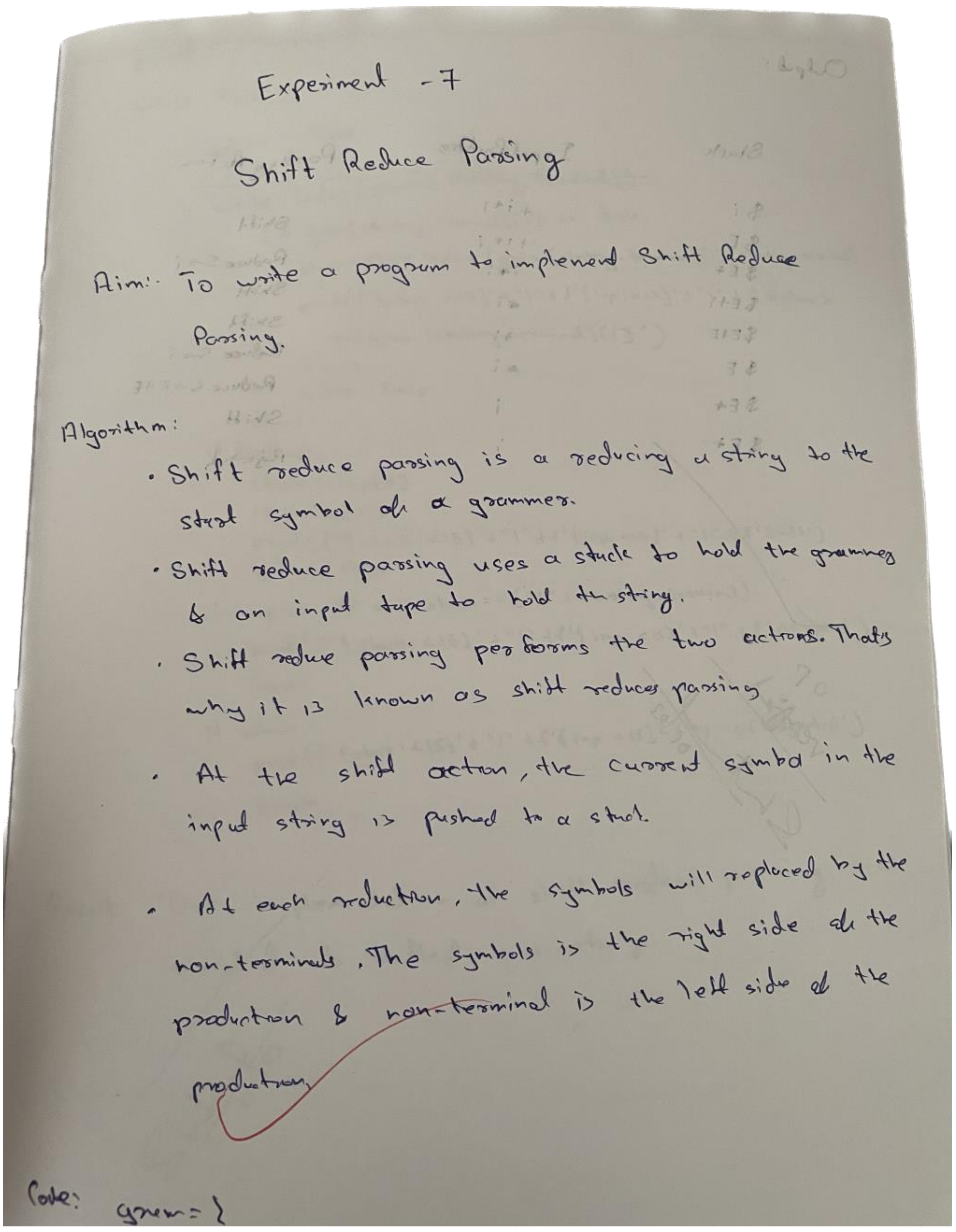
Implementation of DAG

File display

Aim: A program to Implementation of DAG

Algorithm:-

1. The leaves of a graph are labeled by a unique identifier and that identifier can be variable names or constants.
2. Interior nodes of the graph are labeled by an operator symbol.
3. Nodes are also given a sequence of identifiers for labels to store the computed value.
4. If y operand is undefined then create node(y).
5. If z operand is undefined then for case(i) create node(z).
6. For case(i), create node(OP) whose right child is node(z) and left child is node(y).
7. For case(ii), check whether there is node(OP) with one child node(y).
8. For case(iii), node n will be node(y).
9. For node(x) delete x from the list of identifiers. Append x to attached identifiers list for the node n found in step 2. Finally set node(x) to n.



Experiment - 7

Shift Reduce Parsing

Aim:- To write a program to implement Shift Reduce Parsing.

Algorithm:

- Shift reduce parsing is a reducing a string to the start symbol of a grammer.
- Shift reduce parsing uses a stack to hold the grammer & an input tape to hold the string.
- Shift reduce parsing performs the two actions. That's why it is known as shift reduces parsing
- At the shift action, the current symbol in the input string is pushed to a stack.
- At each reduction, the symbols will replaced by the non-terminals. The symbols is the right side of the production & non-terminal is the left side of the production.

Code: gram = {