Experiment 01 : First () and Follow() Set

Learning Objective: Student should be able to Compute First () and Follow () set of given grammar.
Tools: Jdk1.8,Turbo C/C++, Python, Notepad++

Theory:

1. **Algorithm to Compute FIRST as follows:**
   - Let a be a string of terminals and non-terminals.
   - First (a) is the set of all terminals that can begin strings derived from a.

   Compute FIRST(X) as follows:

   a) if X is a terminal, then FIRST(X)={X}

   b) if X→ε is a production, then add ε to FIRST(X)

   c) if X is a non-terminal and $X \rightarrow Y_1 Y_2 ... Y_n$ is a production, add FIRST(Y_i) to FIRST(X) if the preceding Y_js contain ε in their FIRSTs

2. **Algorithm to Compute FOLLOW as follows:**

   a) FOLLOW(S) contains EOF

   b) For productions A→αBβ, everything in FIRST (β) except ε goes into FOLLOW (B)

   c) For productions A→αB or A→αBβ where FIRST (β) contains ε, FOLLOW(B) contains everything that is in FOLLOW(A)

Original grammar:

E→E+E
E→E*E
E→(E)
E→id

This grammar is left-recursive, ambiguous and requires left-factoring. It needs to be modified before we build a predictive parser for it:

Step 1: Remove Ambiguity.

E→E+T
T→T*F
F→(E)
F→id

Grammar is left recursive hence Remove left recursion:

E→TE'

E'→+TE'|ε
T→FT'

T'→*FT'|ε
F→(E)
F→id

Step 2: Grammar is already left factored.

Step 3: Find First & Follow set to construct predictive parser table:-

FIRST (E) = FIRST(T) = FIRST(F) = {(, **id**}

FIRST (E') = {+, ε}

FIRST (T') = {*, ε}

FOLLOW (E) = FOLLOW(E') = {**$**, **)**}

FOLLOW (T) = FOLLOW(T') = {+, **$**, **)**}

FOLLOW (F) = {*, +, **$**, **)**}

Example:

E→TE'

E'→+TE'|ε
T→FT'

T'→*FT'|ε
F→(E)
F→id

FIRST (E) = FIRST (T) = FIRST(F) = {**(**, **id**}

FIRST (E') = {+, ε}

FIRST (T') = {*, ε}

FOLLOW (E) = FOLLOW (E') = {**$**, **)**}

FOLLOW (T) = FOLLOW (T') = {+, **$**, **)**}

FOLLOW (F) = {*, +, **$**, **)**}

Application: To design Top Down and Bottom up Parsers.

Design:

Implementation:

Source Code:

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

        // The Input grammar
        strcpy(production[0], "X=TnS");
        strcpy(production[1], "X=Rm");
        strcpy(production[2], "T=q");
        strcpy(production[3], "T=#");
        strcpy(production[4], "S=p");
        strcpy(production[5], "S=#");
        strcpy(production[6], "R=om");
        strcpy(production[7], "R=ST");

        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;

                // Checking if First of c has
                // already been calculated
                for (kay = 0; kay <= ptr;
kay++)
                        if (c == done[kay])
                                xxx = 1;

                if (xxx == 1)
                        continue;

                // Function call
                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;

        // The case where we
        // encounter a Terminal
        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 {
                                // Recursion to
calculate First of
                                // New
Non-Terminal we encounter
                                // at the
beginning

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

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

        // The case where we encounter
        // a Terminal
        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] !=                                    }
'#') {                                                                         else {
                        f[m++] =
calc_first[i][j];                                   followfirst(production[c1][c2], c1,
                    }
                    else {                                  c2 + 1);
                        if                                                     }
(production[c1][c2] == '\0') {                          }
                            // Case                     j++;
where we reach the                                      }
                            // end              }
of a production                                 }

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

Output:

```
First(X) = { q, n, o, p, #, }

First(T) = { q, #, }

First(S) = { p, #, }

First(R) = { o, p, q, #, }

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

Follow(X) = { $,  }

Follow(T) = { n, m,  }

Follow(S) = { $, q, m,  }

Follow(R) = { m,  }


...Program finished with exit code 0
Press ENTER to exit console.
```

Result and Discussion:

Learning Outcomes: The student should have the ability to

LO1: Identify type of grammar G.
LO2: Define First () and Follow () sets.
LO3: ***Find***First () and Follow () sets for given grammar G.
LO4: ***Apply*** First () and Follow () sets for designing Top Down and Bottom up Parsers

Course Outcomes: Upon completion of the course students will be able to analyze the analysis and synthesis phase of the compiler for writing application programs and construct different parsers for given context free grammars.

Conclusion:

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				

Experiment 02: Code optimization Techniques

Learning Objective: Student should be able to Analyze and Apply code optimization techniques to increase efficiency of compiler.

Tools: Jdk1.8, Turbo C/C++, Python, Notepad++

Theory:

Code optimization aims at improving the execution efficiency. This is achieved in two ways.

1. Redundancies in a program are eliminated.

2. Computations in a program are rearranged or rewritten to make it execute efficiently.

The code optimization must not change the meaning of the program.

Constant Folding:

When all the operands in an operation are constants, operation can be performed at compilation time.

Elimination of common sub-expressions:

Common sub-expression are occurrences of expressions yielding the same value.

Implementation:

1) Expressions with same value are identified
2) Their equivalence is determined by considering whether their operands have the same values in all occurrences
3) Occurrences of sub-expression which satisfy the criterion mentioned earlier for expression can be eliminated.

Dead code elimination

Code which can be omitted from the program without affecting its result is called dead code. Dead code is detected by checking whether the value assigned in an assignment statement is used anywhere in the program

Frequency Reduction

Execution time of a program can be reduced by moving code from a part of program which is executed very frequently to another part of program, which is executed fewer times. For ex. Loop optimization moves, loop invariant code out of loop and places it prior to loop entry.

Strength reduction

The strength reduction optimization replaces the occurrence of a time-consuming operations by an occurrence of a faster operation. For ex. Replacement of Multiplication by Addition

Example:

A=B+C

B=A-D

C=B+C

D=A-D

After Optimization:

A=B+C

B=A-D

C=B+C

D=B

Application: To optimize code for improving space and time complexity.

Result and Discussion:

Learning Outcomes: The student should have the ability to

LO1: **Define** the role of Code Optimizer in Compiler design.
LO2: **List** the different principal sources of Code Optimization.
LO3: ***Apply*** different code optimization techniques for increasing efficiency of compiler.
LO4: ***Demonstrate*** the working of Code Optimizer in Compiler design.

Course Outcomes: Upon completion of the course students will be able to Evaluate the synthesis phase to produce object code optimized in terms of high execution speed and less memory usage.

Conclusion:

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				

Experiment 03 : Two Pass Assembler

Learning Objective: Student should be able to Apply 2 pass Assembler for X86 machine.
Tools: Jdk1.8,Turbo C/C++, Python, Notepad++
Theory:
An assembler performs the following functions

1 Generate instructions

a. Evaluate the mnemonic in the operator field to produce its machine code.
b. Evaluate subfields- find value of each symbol, process literals & assign address.

2 Process pseudo ops: we can group these tables into passed or sequential scans over input associated with each task are one or more assembler modules.

Format of Databases:

a) POT (Pseudo Op-code Table):-

POT is a fixed length table i.e. the contents of these table are altered during the assembly process.

Pseudo Op-code (5 Bytes character)	Address of routine to process pseudo-op-code. (3 bytes= 24 bit address)
"DROPb"	P1 DROP
"ENDbb"	P1 END
"EQUbb"	P1 EQU
"START"	P1 START
"USING"	P1 USING

- The table will actually contain the physical addresses.
- POT is a predefined table.
- In PASS1 , POT is consulted to process some pseudo opcodes like-DS,DC,EQU
- In PASS2, POT is consulted to process some pseudo opcodes like DS,DC,USING,DROP

b) MOT (Mnemonic Op-code Table):-

MOT is a fixed length table i.e. the contents of these tables are altered during the assembly process.

Mnemonic Op-code (4 Bytes character)	Binary Op-code (1 Byte Hexadecimal)	Instruction Length (2 Bits binary)	Instruction Format (3 bits binary)	Not used in this design (3 bits)
"Abbb"	5A	10	001	
"AHbb"	4A	10	001	
"ALbb"	5E	10	001	
"ALRb"	1E	01	000	

b- Represents the char blanks.

Codes:-

Instruction Length
01= 1 Half word=2 Bytes
10= 2 Half word=4 Bytes
11= 3 Half word=6 Bytes

Instruction Format
000 = RR
001 = RX
010 = RS
011= SI
100= SS

- MOT is a predefined table.
- In PASS1 , MOT is consulted to obtain the instruction length.(to Update LC)
- In PASS2, MOT is consulted to obtain:
    a) Binary Op-code (to generate instruction)
    b) Instruction length (to update LC)
    c) Instruction Format (to assemble the instruction).

C) Symbol table (ST):

Symbol (8 Bytes charaters)	Value (4 Bytes Hexadecimal)	Length (1 Byte Hexadecimal)	Relocation (R/A) (1 Byte character)
"PRG1bbb"	0000	01	R
"FOURbbbb"	000C	04	R

- ST is used to keep a track on the symbol defined in the program.
- In pass1- whenever the symbol is defined an entry is made in the ST.
- In pass2- Symbol table is used to generate the address of the symbol.

D) Literal Table (LT):

Literal	Value	Length	Relocation (R/A)
= F '5'	28	04	R

-
- LT is used to keep a track on the Literals encountered in the program.
- In pass1- whenever the literals are encountered an entry is made in the LT.
- In pass2- Literal table is used to generate the address of the literal.

E) Base Table (BT):

Register Availability (1 Byte Character)	Contents of Base register (3 bytes= 24 bit address hexadecimal)
1 'N'	-
2 'N'	-
15 'N'	00

- Code availability-
- Y- Register specified in USING pseudo-opcode.

- N--Register never specified in USING pseudo-opcode.
- BT is used to keep a track on the Register availability.
- In pass1- BT is not used.
- In pass2- In pass2, BT is consulted to find which register can be used as base registers along with their contents.

F) Location Counter (LC):

- LC is used to assign addresses to each instruction & address to the symbol defined in the program.
- LC is updated only in two cases:-
  a) If it is an instruction then it is updated by instruction length.
  b) If it is a data representation (DS, DC) then it is updated by length of data field

Pass 1: Purpose - To define symbols & literals

1) Determine length of machine instruction (MOTGET)
2) Keep track of location counter (LC)
3) Remember values of symbols until pass2 (STSTO)
4) Process some pseudo ops. EQU
5) Remember literals (LITSTO)

Pass 2: Purpose - To generate object program

1) Look up value of symbols (STGET)
2) Generate instruction (MOTGET2)
3) Generate data (for DC, DS)
4) Process pseudo ops. (POT, GET2)

Data Structures:

Pass 1: Database

1) Input source program
2) Location counter (LC) to keep the track of each instruction location
3) MOT (Machine OP table that gives mnemonic & length of instruction
4) POT (Pseudo op table) which indicate mnemonic and action to be taken for each pseudo-op
5) Literals table that is used to store each literals and its location
6) A copy of input to be used later by pass-2

Pass 2: Database

1) Copy of source program from Pass1
2) Location counter
3) MOT which gives the length, mnemonic format op-code
4) POT which gives mnemonic & action to be taken
5) Symbol table from Pass1
6) Base table which indicates the register to be used or base register
7) A work space INST to hold the instruction & its parts
8) A work space PRINT LINE, to produce printed listing
9) A work space PUNCH CARD for converting instruction into format needed by loader
10) An output deck of assembled instructions needed by loader.

Algorithm:

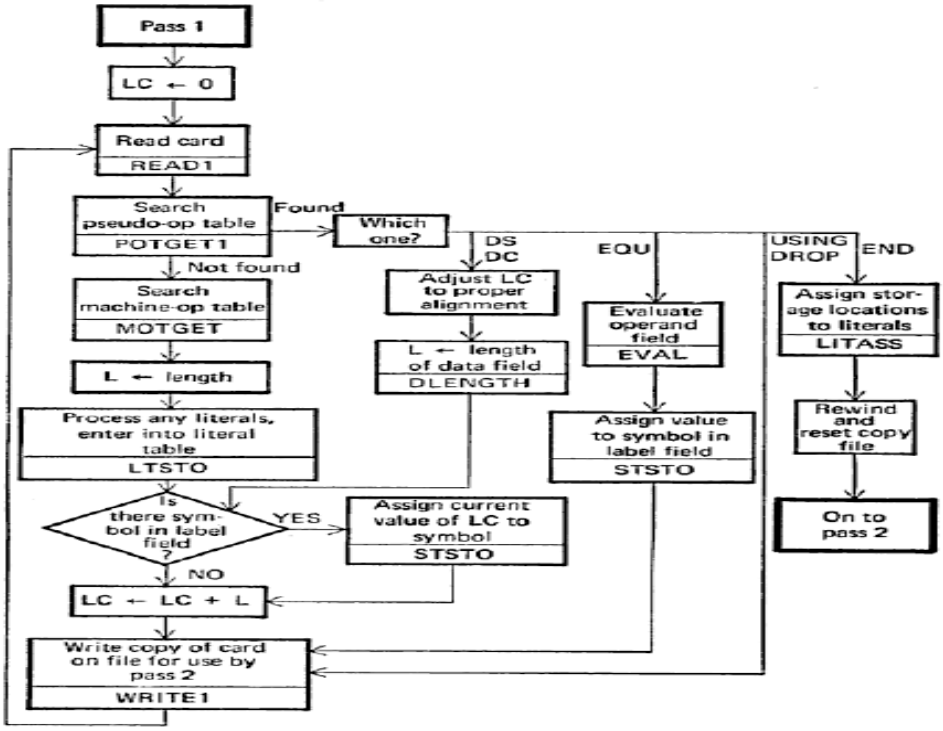
Pass 1

1. Initialize LC to 0
2. Read instruction
3. Search for pseudo-op table and process it.
    a. If its a USING & DROP pseudo-op then pass it to pass2 assembler
    b. If its a DS & DC then Adjust LC and increment LC by L
    c. If its EQU then evaluate the operand field and add value of symbol in symbol table
    d. If its END then generates Literal Table and terminate pass1
4. Search for machine op table
5. Determine length of instruction from MOT
6. Process any literals and enter into literal table
7. Check for symbol in label field
    a. If yes assign current value of LC to Symbol in ST and increment LC by length
    b. If no increment LC by length
8. Write instruction to file for pass 2
9. Go to statement 2

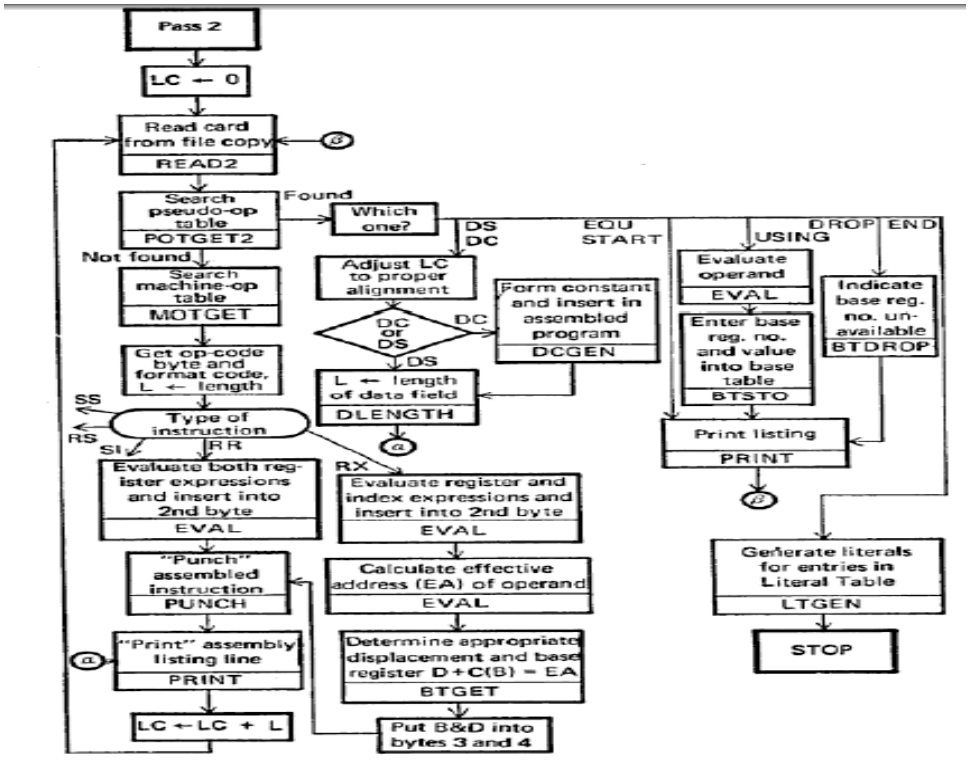
Pass 2

1. Initialize LC to 0.
2. Read instruction
3. Search for pseudo-op table and process it.
    a. If it's a USING then check for base register number and find the contents of the base register
    b. If it's a DROP then base register is not available
    c. If it's a DS then find length of data field and print it
    d. If DC then form constant and insert into machine code.
    e. If its EQU and START then print listing
    f. If its END then generates Literal Table and terminate pass1
    g. Generate literals for entries in literal table
    h. stop
4. Search for machine op table
5. Get op-code byte and format code
6. Set L = length
7. Check for type of instruction
    a. evaluate all operands and insert into second byte
    b. increment LC by length
    c. print listing
    d. Write instruction to file
8. Go to step 2

Flowchart:Pass1



Flowchart: Pass 2:



Example:

```
JOHN        START   0
            USING   *, 15
L           1, FIVE
            A       1, FOUR
            ST      1, TEMP
FOUR        DC      F '4'
FIVE        DC      F '5'
TEMP        DS      1F
            END
```

Application: To design 2-pass assembler for X86 processor

Learning Outcomes: The student should have the ability to

LO1: ***Describe*** the different database formats of 2-pass Assembler with the help of examples.
LO2: **Design** 2 pass Assembler for X86 machine.
LO3: ***Develop***2-pass Assembler for X86 machine.
LO4: Illustrate the working of 2-Pass Assembler.

Course Outcomes: Upon completion of the course students will be able to Describe the various data structures and passes of assembler design.

Conclusion:

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	
Marks Obtained				

Experiment 04 : Two Pass MacroProcessor

Learning Objective: Student should be able to Apply Two-pass Macro Processor.

Tools: Jdk1.8,Turbo C/C++, Python, Notepad++

Theory:

Macro processor is a program which is responsible for processing the macro.

There are four basic tasks/ functions that any macro instruction processor must perform.

1. **Recognize macro definition:**

   A macro instruction processor must recognize macro definitions identified by the MACRO and MEND pseudo-ops.

2. **Save the definitions**:

   The processor must store the macro instruction definitions, which it will need for expanding macro calls.

3. **Recognize calls:**

   The processor must recognize macro calls that appear as operation mnemonics. This suggests that macro names be handled as a type of op-code.

4. **Expand calls and substitute arguments**:

   The processor must substitute for dummy or macro definition arguments the corresponding arguments from a macro call; the resulting symbolic (in this case, assembly language) text is then substituted for the macro call. This text, of course, may contain additional macro definitions or calls.

In summary: the macro processor must recognize and process macro definitions and macro calls.

Two Pass Macro processor:

The macro processor algorithm will two passes over the input text, searching for the macro definition & then for macro calls.

Data bases required for Pass1 & Pass2 Macro processor:

The following data bases are used by the two passes of the macro processor:

Pass 1 data bases:

1. The input macro source deck
2. The output macro source deck copy for use by pass 2
3. The Macro Definition Table (MDT), used to store the body of the macro definitions
4. The Macro Name Table (MNT), used to store the names of defined macros
5. The Macro Definition Table Counter (MDTC), used to indicate the next available entry in the MDT
6. The Macro Name Table Counter (MNTC), used to indicate the next available entry in the MNT
7. The Argument List Array (ALA), used to substitute index markers for dummy arguments before storing a macro definition

Pass 2 data bases:

1. The copy of the input macro source deck
2. The output expanded source deck to be used as input to the assembler
3. The Macro Definition Table (MDT), created by pass 1
4. The Macro Name Table (MNT), created by pass 1
5. The Macro Definition Table Pointer (MDTP), used to indicate the next line of text to be used during macro expansion
6. The Argument List Array (ALA), used to substitute macro call arguments for the index markers in the stored macro definition

Format of Databases:

1) Argument List Array:

- The Argument List Array (ALA) is used during both pass 1 and pass 2.

During pass 1, in order to simplify later argument replacement during macro expansion, dummy arguments in the macro definition are replaced with positional indicators when the definition is stored: The ith dummy argument on the macro name card 'is represented in the body of the macro by the index marker symbol #.

Where # is a symbol reserved for the use of the macro processor (i.e., not available to the programmers).

- These symbols are used in conjunction with the argument list prepared before expansion of a macro call. The symbolic dummy arguments are retained on the macro name card to

enable the macro processor to handle argument replacement by name rather than by position.

- During pass 2 it is necessary to substitute macro call arguments for the index markers stored in the macro definition.

Argument List Array:

Index	8 bytes per entry
0	*"bbbbbbbb"* (all blank)
2	"DATA3bbb"
3	"DATA2bbb"
4	"DATA1bbb"

2) Macro Definition Table:

- The Macro Definition Table (MDT) is a table of text lines.
- Every line of each macro definition, except the MACRO line, is stored in the MDT. (The MACRO line is useless during macro expansion.)
- The MEND is kept to indicate the end of the definition; and the macro name line is retained to facilitate keyword argument replacement.

Macro Definition Table

8O bytes per entry

Index		Card	
.		.	
.		.	
15	&LAB	INCR	&ARG1,&AAG2,&AAG3
16	#0	A	1, #1
17		A	2, #2
18		A	3,#3
19		MEND	
.		.	
.		.	
.		.	

2) The Macro Name Table (MNT) :

- MNT serves a function very similar to that of the assembler's Machine-Op Table (MOT) and Pseudo-Op Table(POT).
- Each MNT entry consists of a character string (the macro name) and a pointer (index) to the entry in the MDT that corresponds to the beginning of the macro definition.

Index	8 Bytes	4 Bytes
.	.	.
.	.	.
3	"INCRbbbb"	15
.	.	.
.	.	.

ALGORITHM

PASS I-MACRO DEFINITION: The algorithm for pass-l tests each input line. If-it is a MACRO pseudo-op:

1) The entire macro definition that follows is saved in the next available locations in the Macro Definition Table (MDT).
2) The first line of the definition is the macro name line. The name is entered into the Macro Name Table (MNT), along with a pointer to the first location of the MDT entry of the definition.
3) When the END pseudo-op is encountered, all of the macro definitions have been processed so control transfers to pass 2 in order to process macro calls.

PASS2-MACRO CALLS AND EXPANSION:

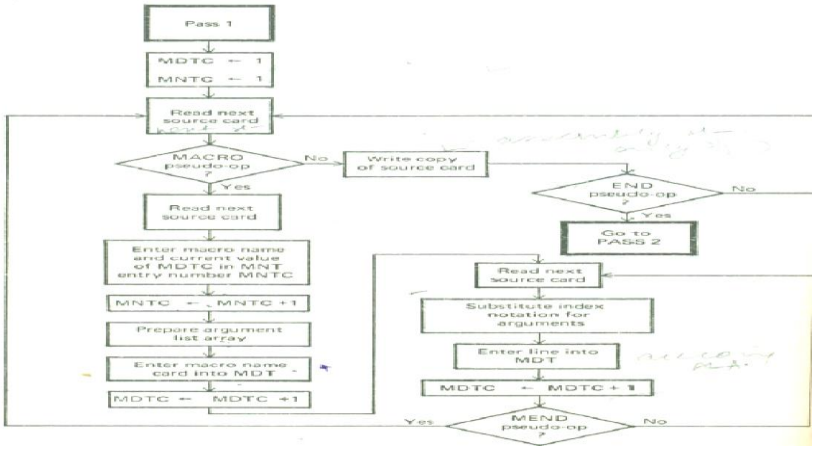
The algorithm for pass 2 tests the operation mnemonic of each input line to see if it is a name in the MNT. When a call is found:-

1) The macro processor sets a pointer, the Macro Definition Table Pointer (MDTP) , to the corresponding macro definition stored in the MDT. The initial value of the MDTP is obtained from the "MDT index" field of the MNT entry.
2) The macro expander prepares the Argument List Array(ALA) consisting of a table of dummy argument indices and corresponding arguments to the call.

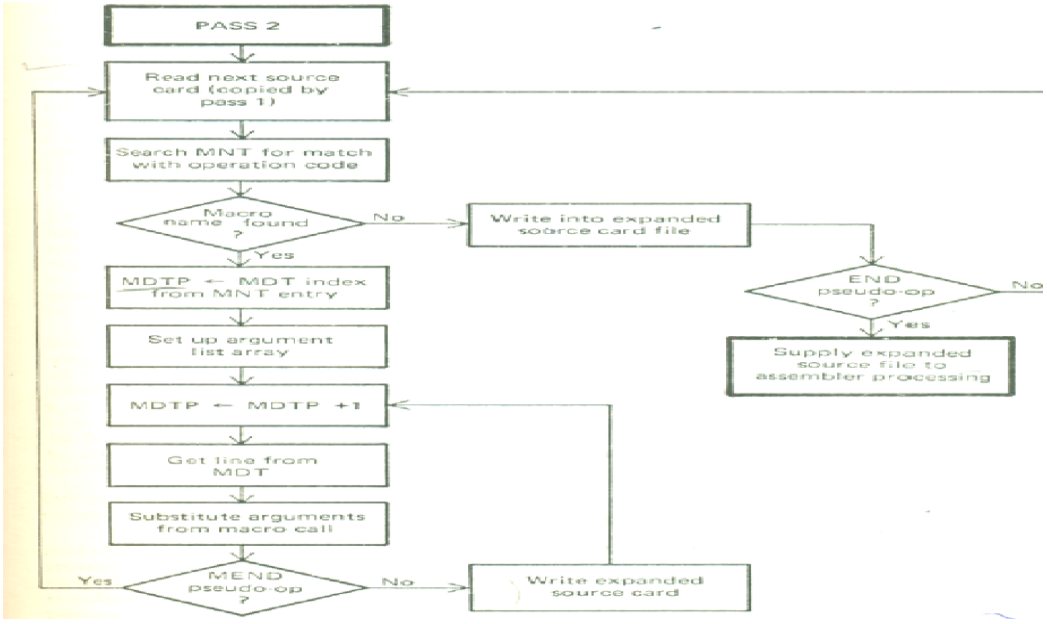
3) Reading proceeds from the MDT; as each successive line is read, the values from the argument list are substituted for dummy argument indices in the macro definition.
4) Reading of the MEND line in the MDT terminates expansion of the macro, and scanning continues from the input file.
5) When the END pseudo-op is encountered, the expanded source deck is transferred to the assembler for further processing.

Flowchart of Pass1 & Pass2 Macro processor:

Pass 1: processing macro definitions



Pass2 : processing macro calls and expansion



Application: To design two-pass macroprocessor.

Design:

Result and Discussion:

Learning Outcomes: The student should have the ability to

LO1: ***Describe*** the different database formats of two-pass Macro processor with the help of examples.
LO2: **Design** two-pass Macro processor.
LO3: ***Develop*** two-pass Macro processor.
LO4: Illustrate the working of two-pass Macro-processor.

Course Outcomes: Upon completion of the course students will be able to Use of macros in modular programming design

Conclusion:

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				

Experiment 05: Lexical Analyzer

Learning Objective: Students should be able to design a handwritten lexical analyser.
Tools: Jdk1.8,Turbo C/C++, Python, Notepad++

Theory:

Design of lexical analyzer

. Allow white spaces, numbers, and arithmetic operators in an expression
. Return tokens and attributes to the syntax analyzer
. A global variable tokenval is set to the value of the number
. Design requires that
- A finite set of tokens be defined
- Describe strings belonging to each token

Regular Expressions

• We use regular expressions to describe tokens of a programming language.
• A regular expression is built up of simpler regular expressions (using defining rules)
• Each regular expression denotes a language.
• A language denoted by a regular expression is called a **regular set**.

Regular Expressions (Rules)

Regular expressions over alphabet S

Regular Expression	Language it denotes
ε	{ ε }
aЄ∑	S {a}
(r1) | (r2)	L(r1) È L(r2)
(r1) (r2)	L(r1) L(r2)
(r)*	(L(r))*
(r)	L(r)

• (r)+ = (r)(r)*
• (r)? = (r) | ε
• We may remove parentheses by using precedence rules.

*	highest
concatenation	next
|	lowest

How to recognize tokens

Construct an analyzer that will return <token, attribute> pairs
We now consider the following grammar and try to construct an analyzer that will return **<token, attribute> pairs.**

relop	**< | = | = | <> | = | >**
id	**letter (letter | digit)***
num	**digit+ ('.' digit+)? (E ('+' | '-')? digit+)?**
delim	**blank | tab | newline**
ws	**delim+**

Using the set of rules as given in the example above we would be able to recognize the tokens. Given a regular expression R and input string x, we have two methods for determining whether x is in L(R). One approach is to use an algorithm to construct an NFA N from R, and the other approach is using a DFA.

Finite Automata

• A *recognizer* for a language is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise.

– We call the recognizer of the tokens as a *finite automaton*.

• A finite automaton can be: *deterministic(DFA)* or *non-deterministic (NFA)*

• This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.

• Both deterministic and non-deterministic finite automaton recognizes regular sets.

• Which one?

– deterministic – faster recognizer, but it may take more space

– non-deterministic – slower, but it may take less space

– Deterministic automatons are widely used lexical analyzers.

• First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.

Algorithm1: Regular Expression → NFA → DFA (two steps: first to NFA, then to DFA)

Algorithm2: Regular Expression → DFA (directly convert a regular expression into a DFA)

Converting Regular Expressions to NFAs

• Create transition diagram or transition table i.e. NFA for every expression

• Create a zero state as the start state and with an e-transition connect all the NFAs and prepare a combined NFA.

Algorithm:for lexical analysis

1) Specify the grammar with the help of regular expression
2) Create transition table for combined NFA
3) read input character
4) Search the NFA for the input sequence.
5) On finding accepting state
   i. if token is id or num search the symbol table
      1. if symbol found return symbol id
      2. else enter the symbol in the symbol table and return its id.
   ii. Else return token
6) Repeat steps 3 to 5 for all input characters.

Input:

```
#include<stdio.h>
void main()
{
inta,b;
printf("Hello");
getch();
}
```

Output:

Preprocessor Directives: #include
Header File: stdio.h
Keyword : void main intgetch
Symbol: <> , ; ( ) ; }
Message: Hello

Application: To design a lexical analyzer.

Design:

Source Code:

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
bool isValidDelimiter(char ch) {
  if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
  ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
  ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
  ch == '[' || ch == ']' || ch == '{' || ch == '}')
  return (true);
  return (false);
}
bool isValidOperator(char ch){
  if (ch == '+' || ch == '-' || ch == '*' ||
  ch == '/' || ch == '>' || ch == '<' ||
  ch == '=')
  return (true);
  return (false);
}
// Returns 'true' if the string is a VALID
IDENTIFIER.
bool isvalidIdentifier(char* str){
  if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
  str[0] == '3' || str[0] == '4' || str[0] == '5' ||
  str[0] == '6' || str[0] == '7' || str[0] == '8' ||
  str[0] == '9' || isValidDelimiter(str[0]) == true)
  return (false);
  return (true);
}
bool isValidKeyword(char* str) {
    if (!strcmp(str, "if") || !strcmp(str, "else") ||
!strcmp(str, "while") || !strcmp(str, "do") ||
!strcmp(str, "break") || !strcmp(str, "continue") ||
!strcmp(str, "int")
  || !strcmp(str, "double") || !strcmp(str, "float") ||
!strcmp(str, "return") || !strcmp(str, "char") ||
!strcmp(str, "case") || !strcmp(str, "char")
  || !strcmp(str, "sizeof") || !strcmp(str, "long") ||
!strcmp(str, "short") || !strcmp(str, "typedef") ||
!strcmp(str, "switch") || !strcmp(str, "unsigned")
  || !strcmp(str, "void") || !strcmp(str, "static") ||
!strcmp(str, "struct") || !strcmp(str, "goto"))
  return (true);
  return (false);
}
bool isValidInteger(char* str) {
  int i, len = strlen(str);
  if (len == 0)
  return (false);
  for (i = 0; i < len; i++) {
    if (str[i] != '0' && str[i] != '1' && str[i] !=
'2'&& str[i] != '3' && str[i] != '4' && str[i] != '5'
    && str[i] != '6' && str[i] != '7' && str[i] !=
'8' && str[i] != '9' || (str[i] == '-' && i > 0))
  return (false);
  }
  return (true);
}
bool isRealNumber(char* str) {
  int i, len = strlen(str);
  bool hasDecimal = false;
  if (len == 0)
  return (false);
  for (i = 0; i < len; i++) {
    if (str[i] != '0' && str[i] != '1' && str[i] != '2'
&& str[i] != '3' && str[i] != '4' && str[i]      !=
'5' && str[i] != '6' && str[i] != '7' && str[i] != '8'
    && str[i] != '9' && str[i] != '.' || (str[i] == '-'
&& i > 0))
    return (false);
      if (str[i] == '.')
    hasDecimal = true;
  }
  return (hasDecimal);
}
char* subString(char* str, int left, int right) {
  int i;
   char* subStr = (char*)malloc( sizeof(char) *
(right - left + 2));
  for (i = left; i <= right; i++)
    subStr[i - left] = str[i];
  subStr[right - left + 1] = '\0';
  return (subStr);
}
void detectTokens(char* str) {
  int left = 0, right = 0;
  int length = strlen(str);
  while (right <= length && left <= right) {
    if (isValidDelimiter(str[right]) == false)
    right++;
      if (isValidDelimiter(str[right]) == true &&
left == right) {
```

```
        if (isValidOperator(str[right]) == true)
        printf("Valid operator : '%c'
", str[right]);
        right++;
        left = right;
      } else if (isValidDelimiter(str[right]) == true
&& left != right || (right == length && left !=
right)) {
        char* subStr = subString(str, left, right - 1);
        if (isValidKeyword(subStr) == true)
          printf("Valid keyword : '%s'
", subStr);
        else if (isValidInteger(subStr) == true)
          printf("Valid Integer : '%s'
", subStr);
        else if (isRealNumber(subStr) == true)
          printf("Real Number : '%s'
", subStr);
        else if (isvalidIdentifier(subStr) == true
              && isValidDelimiter(str[right - 1]) ==
false)
        printf("Valid Identifier : '%s'
", subStr);
        else if (isvalidIdentifier(subStr) == false
              && isValidDelimiter(str[right - 1]) ==
false)
        printf("Invalid Identifier : '%s'
", subStr);
        left = right;
      }
    }
  return;
}
int main(){
  char str[100] = "float x = a + 1b; ";
  printf("The Program is : '%s'
", str);
  printf("All Tokens are :
");
  detectTokens(str);
  return (0);
}
```

Output:

```
The Program is : 'float x = a + 1b; '
All Tokens are :
Valid keyword : 'float'
Valid Identifier : 'x'
Valid operator : '='
Valid Identifier : 'a'
Valid operator : '+'
Invalid Identifier : '1b'
```

Result and Discussion:

Learning Outcomes: The student should have the ability to

LO1: Appreciate the role of lexical analyzer in compiler design

LO2: Define role of lexical analyzer.

Course Outcomes: Upon completion of the course students will be able to Illustrate the working of the compiler and handwritten /automatic lexical analyzer.

Conclusion:

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	
Marks Obtained				

Experiment 06: Intermediate Code Generator

Learning Objective: Student should be able to Apply Intermediate Code Generator using 3-Address code.

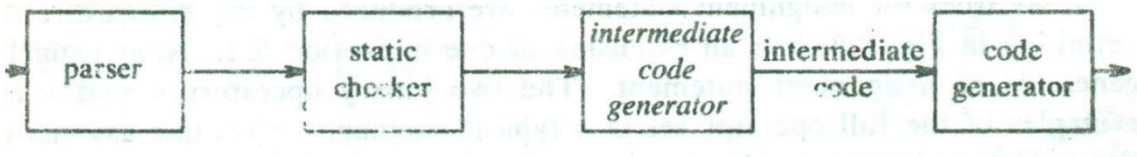
Tools: Jdk1.8, Turbo C/C++, Python, Notepad++

Theory:

Intermediate Code Generation:

In the analysis-synthesis model of a compiler, the front end translates a source program into an intermediate representation from which the back end generates target code. Details of the target language are confined to the backend, as far as possible. Although a source program can be translated directly into the target language, some benefits of using a machine-independent intermediate form are:

1. Retargeting is facilitated; a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation.



Position of intermediate code generator.

(Intermediate Languages) Intermediate Code Representation:

a) Syntax trees or DAG

b) postfix notation

c) Three address code

Three-Address Code:

Three-address code is a sequence of statements of the general form

x:= y op z

where x, y, and z are names, constants, or compiler generated temporaries; op stands for any operator, such as fixed-or floating-point arithmetic operator, or a logical operator on Boolean-valued data.

A source language expression like x+y+z might be translated into a sequence

t1:= y*z

t2:=x+t1

where t1 and t2 are compiler-generated temporary names.

Example: a: =b* -c +b * -c

a := b * -c + b * -c

t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5

Types of Three-Address Statements:

Three-address statements are akin to assembly code. Statements can have symbolic labels and there are statements for flow of control. A symbolic label represents the index of a three-address statement in the array holding intermediate code.

Actual indices can be substituted for the labels either by making a separate pass, or by using "back patching,"

Here are the common three-address statements:

I. Assignment statements of the form x= y op Z, where op is a binary arithmetic or logical operation.

2. Assignment instructions of the form x : = op y, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.

3. Copy statements of the form x : = y where the value of y is assigned to x.

4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.

5. Conditional jumps such as if x relop y goto L. This instruction applies a relational operator «, =. >=, etc.) to x and y, and executes the statement with label L next if x stands in relation relop to y. If not, the three-address statement following if x relop y goto L is executed next, as in the usual sequence.

6. param x and call p, n for procedure calls and return y, where y representing a returned value is optional. Their typical use is as the sequence of three-address statements

param Xl

param X2

param Xn

call p,n

generated as part of a call of the procedure p (Xl, X2, …….. Xn) The integer n indicating the number of actual parameters in "call p, n" is not redundant because calls can be nested.

7. Indexed assignments of the form X : = y[i] and x[i] : = y. The first of these sets x to the value in the location i memory units beyond location y. The statement x[i] : = y sets the contents of the location i units beyond x to the value of y. In both these instructions, x, y, and i refer to data objects. Address and pointer assignments of the form x : = &y, x : = *y, and *x : = y.

Implementations of Three-Address Statements

A three-address statement' is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are quadruples, triples, and indirect triples.

Quadruples:

a) A quadruple is a record structure with four fields:

op, arg 1, arg 2, and result.

b) The op field contains an internal code for the operator.

c) The three-address statement x : = y op z is represented by placing y in arg1, Z in arg 2, and x in result.

d) Statements with unary operators like x : = -y or x : = y do not use arg 2. Operators like param use neither arg 2 nor result.

e) Conditional and unconditional jumps put the target label in result.

f) The quadruples are for the assignment a : = b * - c + b* - c.
They are obtained from the three-address code in Fig. (a).

g) The contents of fieldsarg I, arg 2, and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

Triples:

a) To avoid entering temporary names into the symbol table, refer to a temporary value by the position of the statement that computes it.

b) Three-address statements can be represented by records with only three fields: op, arg1 and arg2, as in Fig.(b).

c) The fieldsarg1 and arg2, for the arguments of op, are pointer to the symbol table. Since three fields are used, this intermediate code format is known as triples.

	op	*arg 1*	*arg 2*	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

(a) Quadruples

	op	*arg 1*	*arg 2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triples

Quadruple and triple representations of three-address statements.

A ternary operation like x[i] := y requires two entries in the triple structure, as shown in Fig.(a), while x : = y[i] is naturally represented as two operations in Fig. (b).

	op	*arg* 1	*arg* 2
(0)	[ ]=	x	i
(1)	assign	(0)	y

(a) x[i] := y

	op	*arg* 1	*arg* 2
(0)	=[ ]	y	i
(1)	assign	x	(0)

(b) x := y[i]

More triple representations.

Indirect Triples:

Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves. This implementation is naturally called indirect triples.

For example, let us use an array statement to list pointers to triples in the desired order.

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg* 1	*arg* 2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Indirect triples representation of three-address statements

Input:

a : = b * - c + b* - c.

Output:

	op	*arg* 1	*arg* 2	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

(a) Quadruples

	op	*arg* 1	*arg* 2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triples

Application: Intermediate code can be easily produced to the target code.

Result and Discussion:

Learning Outcomes: The student should have the ability to

LO1 **Define** the role of Intermediate Code Generator in Compiler design.
LO2: ***Describe*** the various ways to implement Intermediate Code Generator.
LO3: ***Specify*** the formats of 3 Address Code.
LO4: Illustrate the working of Intermediate Code Generator using 3-Address code
.

Course Outcomes: Upon completion of the course students will be able to Evaluate the synthesis phase to produce object code optimized in terms of high execution speed and less memory usage.

Conclusion:

For Faculty Use:

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	
Marks Obtained				

Design:

Input:

Output:

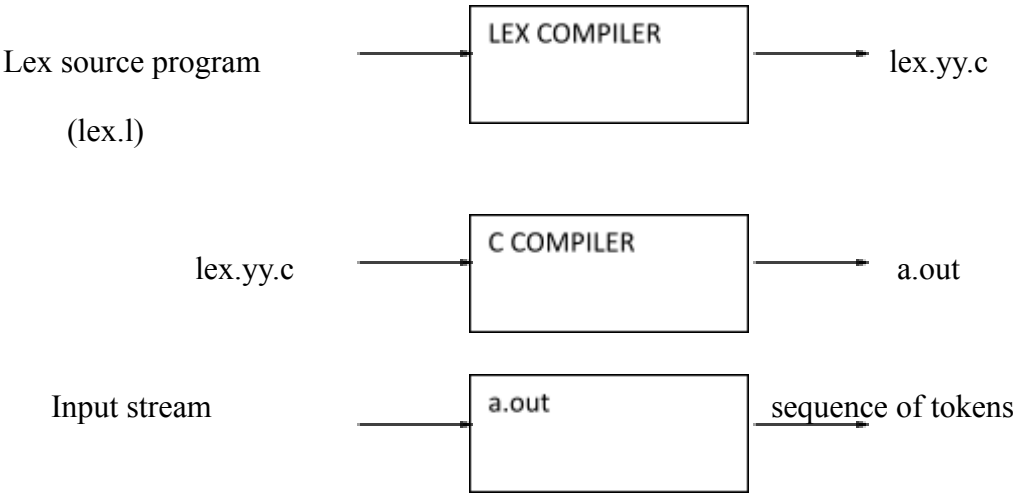
Experiment 07 : Lex Tool

Learning Objective: Students should be able to build a Lexical analyzer using LEX / Flex tool.

Tools: Open Source tool (Ubuntu , LEX tool), Notepad++

Theory:

LEX :A tool widely used to specify lexical analyzers for a variety of languages .We refer to the tool as Lex compiler , and to its input specification as the Lex language.



Steps for creating a lexical analyzer with Lex

Lex specifications:

A Lex program (the .l file) consists of three parts:

declarations

%%

translation rules

%%

auxiliary procedures

1. The declarations section includes declarations of variables,manifest constants(A manifest constant is an identifier that is declared to represent a constant e.g. # define PIE 3.14), and regular definitions.
2. The translation rules of a Lex program are statements of the form :

   p1 {action 1}

   p2 {action 2}

   p3 {action 3}

   … …

   …

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.

3. The third section holds whatever auxiliary procedures are needed by the actions.Alternatively these procedures can be compiled separately and loaded with the lexical analyzer.

How does this Lexical analyzer work?

The lexical analyzer created by Lex behaves in concert with a parser in the following manner. When activated by the parser, the lexical analyzer begins reading its remaining input , one character at a time, until it has found the longest prefix of the input that is matched by one of the regular expressions *p.* Then it executes the corresponding *action.* Typically the *action* will return control to the parser. However, if it does not, then the lexical analyzer proeeds to find more lexemes, until an *action* causes control to return to the parser. The repeated search for lexemes until an explicit return allows the lexical analyzer to process white space and comments conveniently.

The lexical analyzer returns a single quantity, the *token*, to the parser. To pass an attribute value with information about the lexeme, we can set the global variable *yylval.*

e.g. Suppose the lexical analyzer returns a single token for all the relational operators, in which case the parser won't be able to distinguish between "<=",">=","<",">","==" etc. We can set yylval appropriately to specify the nature of the operator.

Note: To know the exact syntax and the various symbols that you can use to write the regular expressions visit the manual page of FLEX in LINUX :

$man flex

The two variables *yytext* **and** *yyleng*

Lex makes the lexeme available to the routines appearing in the third section through two variables *yytext*and *yyleng*

1. *yytext* is a variable that is a pointer to the first character of the lexeme.
2. *yyleng*is an integer telling how long the lexeme is.

A lexeme may match more than one patterns. How is this problem resolved?

Take for example the lexeme *if.* It matches the patterns for both *keyword if* and *identifier*. If the pattern for *keyword if* precedes the pattern for *identifier* in the *declaration* list of the lex program the conflict is resolved in favor of the keyword. In general this ambiguity-resolving strategy makes it easy to reserve keywords by listing them ahead of the pattern for identifiers.

The Lex's strategy of selecting the longest prefix matched by a pattern makes it easy to resolve other conflicts like the one between "<" and "<=".

In the lex program, a *main()* function is generally included as:

main(){

*yyin=fopen(**filename,**"r");*

while(yylex());

}

Here ***filename*** corresponds to input file and the *yylex* routine is called which returns the tokens.

Lex Syntax and Example

Lex is short for "lexical analysis". Lex takes an input file containing a set of lexical analysis rules or regular expressions. For output, Lex produces a C function which when invoked, finds the next match in the input stream.

1. Format of lex input:

(beginning in col. 1) declarations

%%

token-rules

%%
aux-procedures

2. Declarations:
 a) string sets; name character-class
 b) standard C; %{ -- c declarations --
%}

3. Token rules: *regular-expression* { *optional C-code* }

 a) if the expression includes a reference to a character class, enclose the class name in brackets { }
 b) regular expression operators;
 * , + --closure, positive closure
 " " or \ --protection of special chars
 | --or
 ^ --beginning-of-line anchor
 ()--grouping
 $ --end-of-line anchor
 ? --zero or one
 . --any char (except \n)
 {ref} --reference to a named character class (a definition)
 [ ] --character class
 [^] --not-character class

4. Match rules: Longest match is preferred. If two matches are equal length, the first match is preferred. Remember, lex partitions, it does not attempt to find nested matches. Once a character becomes part of a match, it is no longer considered for other matches.

5. Built-in variables: yytext -- ptr to the matching lexeme. (char *yytext;)
yyleng -- length of matching lexeme (yytext). Note: some systems use yyleng

6. **Aux Procedures:** C functions may be defined and called from the C-code of token rules or from other functions. Each lex file should also have a yyerror() function to be called when lex encounters an error condition.

7. Example header file: tokens.h

```
#define NUM     1          // define constants used by lexyy.c
#define ID      2          // could be defined in the lex rule file
#define PLUS    3
#define MULT    4
#define ASGN    5
#define SEMI    6
```

7. Example lex file

```
D     [0-9]                         /* note these lines begin in col. 1 */
A     [a-zA-Z]
%{
#include “tokens.h”
%}
%%
{D}+              return (NUM);       /* match integer numbers */
{A}({A}|{D})*                   return (ID);            /* match identifiers */
"+"               return (PLUS);      /* match the plus sign  (note protection) */
"*"               return (MULT);      /* match the multsign  (note protection
                                                          again) */
: =             return (ASGN);      /* match the assignment string */
;                 return (SEMI);      /* match the semi colon */
.                 ;                   /* ignore any unmatched chars */
%%

voidyyerror ()                        /* default action in case of error in yylex()
  {

               printf (" error\n");
exit(0);
  }

               voidyywrap () { }                  /* usually only needed for some Linux  systems */
```

8. Execution of lex:

(to generate the *yylex()* function file and then compile a user program)

(MS) c:> flexrulefile — (Linux) $ lexrulefile

flexproduceslexyy.c — lexproduceslex.yy.c

The produced .c file contains this function: intyylex()

9. User program:

(The above scanner file must be linked into the project)

```
#include <stdio.h>
#include “tokens.h”

intyylex ();                          // scanner prototype
extern char* yytext;
```

```
main ()
{    int n;
     while ( n = yylex() )                    // call scanner until it returns 0 for
     EOF
        printf (" %d  %s\n", n, yytext);   // output the token code and lexeme
string
}
```

Design:

Result and Discussion:

Learning Outcomes: The student should have the ability to

LO1 **Summarize** different Compiler Construction tools.
LO2: ***Describ*e** the structure of Lex specification.
LO3: ***Apply*** LEX Compiler for Automatic Generation of Lexical Analyzer.
LO4: Construct Lexical analyzer using open source tool for compiler design

Course Outcomes: Upon completion of the course students will be able to Illustrate the working of the compiler and handwritten /automatic lexical analyzer..

Conclusion:

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				

Experiment 08 : YACC Tool

Learning Objective: Student should be able to Build Parser Generator using YACC tool.

Tools: Open Source tool (Ubuntu , LEX tool), Notepad++

Theory:

A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers.

YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

Input File:

YACC input file is divided in three parts.

```
/* definitions */
 ....
%%
/* rules */
....
%%
/* auxiliary routines */
....
```

Input File: Definition Part:

- The definition part includes information about the tokens used in the syntax definition:
- %token NUMBER
  %token ID
- Yacc automatically assigns numbers for tokens, but it can be overridden by
  %token NUMBER 621
- Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should no overlap ASCII codes.

- The definition part can include C code external to the definition of the parser and variable declarations, within **%{** and **%}** in the first column.
- It can also include the specification of the starting symbol in the grammar:

```
%start nonterminal
```

Input File: Rule Part:

- The rules part contains grammar definition in a modified BNF form.
- Actions is C code in { } and can be embedded inside (Translation schemes).

Input File: Auxiliary Routines Part:

- The auxiliary routines part is only C code.
- It includes function definitions for every function needed in rules part.
- It can also contain the main() function definition if the parser is going to be run as a program.
- The main() function must call the function yyparse().

Input File:

- If yylex() is not defined in the auxiliary routines sections, then it should be included:

```
#include "lex.yy.c"
```

- YACC input file generally finishes with:

```
.y
```

Output Files:

- The output of YACC is a file named **y.tab.c**
- If it contains the **main()** definition, it must be compiled to be executable.
- Otherwise, the code can be an external function definition for the function **int yyparse()**
- If called with the **–d** option in the command line, Yacc produces as output a header file **y.tab.h** with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).
- If called with the **–v** option, Yacc produces as output a file **y.output** containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.

Example:

Yacc File (.y)

```
%{
  #include <ctype.h>
  #include <stdio.h>
  #define YYSTYPE double /* double type for yacc stack */
%}

%%
 Lines :  Lines S '\n' { printf("OK \n"); }
       | S '\n'
       | error '\n' {yyerror("Error: reenter last line:");
                     yyerrok; };
 S     :  '(' S ')'
       | '[' S ']'
       |  /* empty */    ;
%%

#include "lex.yy.c"

void yyerror(char * s)
/* yacc error handler */
{
 fprintf (stderr, "%s\n", s);
}
   int main(void)
 {
 return yyparse();
 }
```

Lex File (.l)

```
%{

%}

  %%

[ \t]     { /* skip blanks and tabs */ }

\n|.      { return yytext[0]; }

%%
```

For Compiling YACC Program:

1. Write lex program in a file file.l and yacc in a file file.y
2. Open Terminal and Navigate to the Directory where you have saved the files.
3. type lex file.l
4. type yacc file.y
5. type cc lex.yy.c y.tab.h -ll
6. type ./a.out

Design:

```
%{
#include "y.tab.h"
%}
%%
[0-9]+ {yylval.num=atof(yytext); return number;}
[-+*/] {return yytext[0];}
COS|cos {return cos1; }
SIN|sin {return sin1; }
TAN|tan {return tan1; }
%%
int yywrap(){
return 1;
}
```

Cal.txt:

```
%{
#include<stdio.h>
#include<math.h>
%}
%union {float num;}
%start line
%token cos1
%token sin1
%token tan1
%token <num> number
%type <num> exp
%%
line : exp
| line exp
;
exp : number {$$=$1;}
| exp '+' number {$$=$1+$3;printf("\n%f+%f=%f\n",$1,$3,$$);}
| exp '-' number {$$=$1-$3;printf("\n%f-%f=%f\n",$1,$3,$$);}
| exp '*' number {$$=$1*$3;printf("\n%f*%f=%f\n",$1,$3,$$);}
| exp '/' number {$$=$1/$3;printf("\n%f/%f=%f\n",$1,$3,$$);}
| cos1 number {printf("%f",cos(($2/180)*3.14));}
| sin1 number {printf("%f",sin(($2/180)*3.14));}
| tan1 number {printf("%f",tan(($2/180)*3.14));}
;
%%
int main(){
yyparse();
return 0;
}
int yyerror(){
exit(0);
}
```

Learning Outcomes: The student should have the ability to

LO1 **Define** Context Free Grammar.
LO2: ***Describe*** the structure of YACC specification.
LO3: ***Apply*** YACC Compiler for Automatic Generation of Parser Generator.
LO4: Construct Parser Generator using open source tool for compiler design.

Course Outcomes: Upon completion of the course students will be able to Analyze the analysis and synthesis phase of compiler for writhing application programs and construct different parsers for given context free grammars.

Conclusion: Thus, we have successfully implemented YACC Compiler for Automatic Generation of Parser Generator

For Faculty Use:

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				

Experiment 09: Case Study

Learning Objective: Student need to perform case study on topic optimizer compiler
Tools: Google Docs
Title: Optimizing Compiler
Theory:

History:
Building compilers has been a challenging activity since the advent of digital computers in the late 1940s and early 1950s. At that time, implementing the concept of automatic translation from a form familiar to mathematicians into computer instructions was a difficult task. One needed to figure out how to translate arithmetic expressions into instructions, how to store data in memory, and how to choose instructions to build procedures and functions. During the late 1950s and 1960s these processes were automated to the extent that simple compilers could be written by most computer science professionals. In fact, the concept of "small languages" with corresponding translators is fundamental in the UNIX community.

How does a programmer get the performance he expects from his application?
Initially he writes the program in a straightforward fashion so that the correct execution of the program can be tested or proved. The program is then profiled and measured to see where resources such as time and memory are used, and modified to improve the uses of these resources. After all reasonable programmer modifications have been made, further improvements in performance can come only from how well the programming language is translated into instructions for the target machine.

What is an Optimizing Compiler?
The goal of an optimizing compiler is to efficiently use all of the resources of the target computer. The compiler translates the source program into machine instructions using all of the different computational elements. The ideal translation is one that keeps each of the computational elements active doing useful (and nonredundant) work during each instruction execution cycle.
An optimizing compiler in computing is one that makes an effort to minimize or maximize certain characteristics of an executable computer programme. The execution time, memory footprint, storage space, and power consumption of a programme should all be kept to a minimum (the last three are particularly important for portable computers).
A series of optimizing transformations, or algorithms, are typically used to achieve compiler optimization. These algorithms take an input programme and alter it to create an output programme that is semantically comparable but requires less resources or runs faster. Some code optimization issues have been demonstrated to be NP-complete or even insolvable. In reality, a

compiler's ability to optimize is constrained by things like the programmer's patience in waiting for the compiler to finish its job. The majority of the time, optimization requires a lot of CPU and memory resources. The kind of optimisations that could be carried out in the past were significantly constrained by computer memory constraints.

Due to these reasons, optimisation rarely results in "optimal" output in any sense; in some instances, it may even hurt performance. Instead, they are heuristic techniques for enhancing resource utilization in common programmes.

Types of Optimization:

The different scopes of the techniques employed in optimisation can affect anything from a single statement to the entire programme. In general, localized approaches yield lesser improvements than global ones but are simpler to execute. The following are some scope examples:

- **Peephole Optimization:** These are often carried out after the creation of machine code, late in the compilation process. This type of optimisation looks at a few nearby instructions to see whether they may be replaced by a single instruction or a shorter string of instructions, sort of like "looking through a peephole" at the code. For instance, left-shifting or adding a value to itself may be more effective ways to multiply a number by two (this example also demonstrates strength reduction).
- **Local Optimization:** These only take into account data that is specific to a basic block. Due to the lack of control flow in basic blocks, these optimisations need very little analysis, which saves time and storage space but also means that no information is saved over jumps.
- **Global Optimization:** They also go by the name "intraprocedural methods" and work on entire functions. They now have more data at their disposal, but this frequently necessitates expensive calculations. When function calls take place or global variables are accessed, worst case assumptions must be made because there is little knowledge about them.
- **Loop Optimization:** These, for instance loop-invariant code motion, act on the statements that make up a loop, such as a for loop. Due to the fact that many programmes spend a significant amount of time inside loops, loop optimisations can have a significant impact.
- **Prescient Store Optimization:** These enable earlier store operations than would otherwise be possible when threads and locks are involved. The procedure needs a means to anticipate the value that will be saved by the assignment that it was supposed to follow. This relaxation enables some types of code rearrangement that maintain the semantics of correctly synchronized programmes to be performed by compiler optimisation.

- **Interprocedural, Whole-Program or Link-Time Optimization:** These examine every line of source code in a programme. When optimisations only have access to local information, that is, information contained within a single function, they are less effective than when a larger amount of information is collected. This kind of optimization can also make it possible to use novel methodologies. Consider function inlining, in which a copy of the function body is used in place of a call to the function.
- **Machine Code Optimization and Object Code Optimizer:** After all of the executable machine code has been linked, they examine the program's executable task image. When the complete executable task image is accessible for analysis, some of the approaches that can be used in a more constrained context—like macro compression, which reduces file size by collapsing frequent sequences of instructions—are more effective.

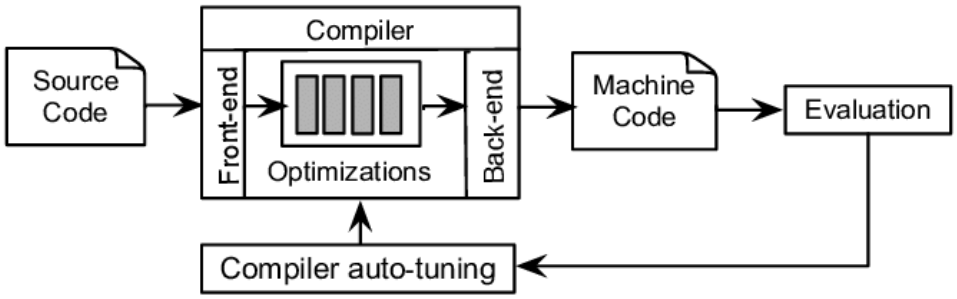


Fig. - Process of compiler optimization exploration

Advantages of Code Optimization in Compilers:

Some of the advantages of an optimizing compiler are:

- Improved performance - An optimizing compiler can analyze the program code and make changes that make it run faster. This can lead to significant improvements in performance, especially for programs that are computationally intensive.
- Efficient Memory Management - Optimizing compilers can also improve memory management in a program. By analyzing the program code, the compiler can determine the most efficient way to use memory, reducing the amount of memory needed and improving program performance.
- Processor-specific optimizations - An optimizing compiler can take advantage of the specific features of a processor to improve performance. For example, it can use vector instructions to perform multiple calculations at once, or it can use specialized instructions to perform certain tasks more efficiently.
- Reduced development time - Optimizing compilers can also help reduce development time by automatically optimizing the code. This can save developers time and effort, allowing them to focus on other aspects of the program.

Disadvantages of Code Optimization in Compilers:

- Increased compilation time: Code optimization can significantly increase the compilation time, which can be a significant drawback when developing large software systems.
- Increased complexity: Code optimization can result in more complex code, making it harder to understand and debug.
- Potential for introducing bugs: Code optimization can introduce bugs into the code if not done carefully, leading to unexpected behavior and errors.
- Difficulty in assessing the effectiveness: It can be difficult to determine the effectiveness of code optimization, making it hard to justify the time and resources spent on the process.

In conclusion, an optimizing compiler is a powerful tool that can help improve the performance of a program. By taking advantage of the specific features of a processor and optimizing the code for efficient memory management, an optimizing compiler can significantly improve program performance.

Conclusion: We learned about the concept of Optimizing Compiler and researched various aspects on this topic. Several concepts related to Compiler construction were revised while performing the experiment.

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	
Marks Obtained				

Mini project
Experiment 10 :Design and development of LR(0) Parser

Learning Objective: Student should be able to apply LR(0) parsing technique and implement it.

Tools: Dev C++,Notepad

Theory:

LR(0) parser: The LR parser is an efficient bottom-up syntax analysis technique that can be used for a large class of context-free grammar. This technique is also 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.

LR(0) parser Algorithm:

1. Construct the given grammar G into augmented grammar
2. Find Item I
3. Compute / Find Closure (I)
4. Apply Goto function to find LR(0) collection
5. Draw DFA Graph
6. Construct LR(0) Table
7. Derive i/p string

Code:

```
#include<iostream>
#include<conio.h>
#include<string.h>

using namespace std;

char prod[20][20],listofvar[26]="ABCDEFGHIJKLMNOPQR";
int novar=1,i=0,j=0,k=0,n=0,m=0,arr[30];
int noitem=0;

struct Grammar
{
        char lhs;
        char rhs[8];
}g[20],item[20],clos[20][10];

int isvariable(char variable)
{
```

```
	for(int i=0;i<novar;i++)
		if(g[i].lhs==variable)
			return i+1;
	return 0;
}
void findclosure(int z, char a)
{
	int n=0,i=0,j=0,k=0,l=0;
	for(i=0;i<arr[z];i++)
	{
		for(j=0;j<strlen(clos[z][i].rhs);j++)
		{
			if(clos[z][i].rhs[j]=='.' && clos[z][i].rhs[j+1]==a)
			{
				clos[noitem][n].lhs=clos[z][i].lhs;
				strcpy(clos[noitem][n].rhs,clos[z][i].rhs);
				char temp=clos[noitem][n].rhs[j];
				clos[noitem][n].rhs[j]=clos[noitem][n].rhs[j+1];
				clos[noitem][n].rhs[j+1]=temp;
				n=n+1;
			}
		}
	}
	for(i=0;i<n;i++)
	{
		for(j=0;j<strlen(clos[noitem][i].rhs);j++)
		{
			if(clos[noitem][i].rhs[j]=='.'                                  &&
isvariable(clos[noitem][i].rhs[j+1])>0)
			{
				for(k=0;k<novar;k++)
				{
					if(clos[noitem][i].rhs[j+1]==clos[0][k].lhs)
					{
						for(l=0;l<n;l++)

	if(clos[noitem][l].lhs==clos[0][k].lhs                                  &&
strcmp(clos[noitem][l].rhs,clos[0][k].rhs)==0)
                                                                    break;
                                                    if(l==n)
                                                    {

	clos[noitem][n].lhs=clos[0][k].lhs;

	strcpy(clos[noitem][n].rhs,clos[0][k].rhs);
                                                                n=n+1;
                                                    }
                                                }
```

```
                                        }
                                }
                        }
                }
                arr[noitem]=n;
                int flag=0;
                for(i=0;i<noitem;i++)
                {
                        if(arr[i]==n)
                        {
                                for(j=0;j<arr[i];j++)
                                {
                                        int c=0;
                                        for(k=0;k<arr[i];k++)
                                                if(clos[noitem][k].lhs==clos[i][k].lhs &&
strcmp(clos[noitem][k].rhs,clos[i][k].rhs)==0)
                                                        c=c+1;
                                        if(c==arr[i])
                                        {
                                                flag=1;
                                                goto exit;
                                        }
                                }
                        }
                }
                exit:;
                if(flag==0)
                        arr[noitem++]=n;
}

int main()
{
                cout<<"ENTER THE PRODUCTIONS OF THE GRAMMAR(0 TO END) :\n";
                do
                {
                        cin>>prod[i++];
                }while(strcmp(prod[i-1],"0")!=0);
                for(n=0;n<i-1;n++)
                {
                        m=0;
                        j=novar;
                        g[novar++].lhs=prod[n][0];
                        for(k=3;k<strlen(prod[n]);k++)
                        {
                                if(prod[n][k] != '|')
                                g[j].rhs[m++]=prod[n][k];
                                if(prod[n][k]=='|')
                                {
```

```
                                g[j].rhs[m]='\0';
                                m=0;
                                j=novar;
                                g[novar++].lhs=prod[n][0];
                        }
                }
        }
        for(i=0;i<26;i++)
                if(!isvariable(listofvar[i]))
                        break;
        g[0].lhs=listofvar[i];
        char temp[2]={g[1].lhs,'\0'};
        strcat(g[0].rhs,temp);
        cout<<"\n\n augumented grammar \n";
        for(i=0;i<novar;i++)
                cout<<endl<<g[i].lhs<<"->"<<g[i].rhs<<" ";

        for(i=0;i<novar;i++)
        {
                clos[noitem][i].lhs=g[i].lhs;
                strcpy(clos[noitem][i].rhs,g[i].rhs);
                if(strcmp(clos[noitem][i].rhs,"ε")==0)
                        strcpy(clos[noitem][i].rhs,".");
                else
                {
                        for(int j=strlen(clos[noitem][i].rhs)+1;j>=0;j--)
                                clos[noitem][i].rhs[j]=clos[noitem][i].rhs[j-1];
                        clos[noitem][i].rhs[0]='.';
                }
        }
        arr[noitem++]=novar;
        for(int z=0;z<noitem;z++)
        {
                char list[10];
                int l=0;
                for(j=0;j<arr[z];j++)
                {
                        for(k=0;k<strlen(clos[z][j].rhs)-1;k++)
                        {
                                if(clos[z][j].rhs[k]=='.')
                                {
                                        for(m=0;m<l;m++)
                                                if(list[m]==clos[z][j].rhs[k+1])
                                                        break;
                                        if(m==l)
                                                list[l++]=clos[z][j].rhs[k+1];
                                }
                        }
```

```
                }
                for(int x=0;x<l;x++)
                        findclosure(z,list[x]);
        }
        cout<<"\n THE SET OF ITEMS ARE \n\n";
        for(int z=0; z<noitem; z++)
        {
                cout<<"\n I"<<z<<"\n\n";
                for(j=0;j<arr[z];j++)
                        cout<<clos[z][j].lhs<<"->"<<clos[z][j].rhs<<"\n";

        }

}
```

OUTPUT:

```
ENTER THE PRODUCTIONS OF THE GRAMMAR(0 TO END) :
S->AA
A->aA|b
0


 augumented grammar

B->S
S->AA
A->aA
A->b
 THE SET OF ITEMS ARE


 I0

B->.S
S->.AA
A->.aA
A->.b

 I1

B->S.
```

```
 I2

S->A.A
A->.aA
A->.b

 I3

A->a.A
A->.aA
A->.b

 I4

A->b.

 I5

S->AA.

 I6

A->aA.

--------------------------------
Process exited after 31.33 seconds with return value 0
Press any key to continue . . .
```

Learning Outcomes: The student should have the ability to

LO1 **Define** the role of Code Generator in Compiler design.
LO2: ***Apply*** the code generator algorithm to generate the machine code.
LO3: ***Generate*** target code for the optimized code, considering the target machines.

Course Outcomes: Upon completion of the course students will be able to evaluate the synthesis phase to produce object code optimized in terms of high execution speed and less memory usage.

Conclusion:

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	
Marks Obtained				