

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

2. Algorithm to Compute FOLLOW as follows:

- a) FOLLOW(S) contains EOF
- b) For productions $A \rightarrow B$, everything in FIRST (B) except goes into FOLLOW (A)
- c) For productions $A \rightarrow B$ or $A \rightarrow B C$ where FIRST (C) contains , FOLLOW(B) contains everything that is in FOLLOW(A)

Original grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E) E \text{ id}$

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.

$E \rightarrow E + T$

$T \rightarrow T * F$

$F \rightarrow (E)$

$F \rightarrow \text{id}$

Grammar is left recursive hence Remove left recursion:

$E' \rightarrow$

$E' \rightarrow +TE' \mid T FT'$

$T' \rightarrow$

$F (E) F \text{ id}$

Step 2: Grammar is already left factored.

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

$(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{ (, \text{id} \}$

$\text{FIRST}(E') = \{ +, \}$

$\text{FIRST}(T') = \{ *, \}$

$\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{ \$,) \}$

$\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{ +, \$,) \}$

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



Example:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid T FT'$

$T' \rightarrow *FT' \mid$

$F \rightarrow (E) F \mid id$

$FIRST(E) = FIRST(T) = FIRST(F) = \{ (, id \}$

$FIRST(E') = \{ +, \}$

$FIRST(T') = \{ *, \}$

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

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

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

Application: To design Top Down and Bottom up Parsers.

Design:

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

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

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



// Functions to calculate Follow

void followfirst(char, int, int);

void follow(char c);

// Function to calculate First

void findfirst(char, int, int);

int count, n = 0;

// Stores the final

result // of the First

Sets char

calc_first[10][100];

// Stores the final result

// of the Follow Sets

char

calc_follow[10][100]; int

m = 0;

// Stores the production rules 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], "E=TR");
```

```
    strcpy(production[1], "R=+TR");
```

```
    strcpy(production[2], "R=#");
```

```
    strcpy(production[3], "T=FY");
```

```
    strcpy(production[4], "Y=*FY");
```

```
    strcpy(production[5], "Y=#");
```

```
    strcpy(production[6], "F=(E)");
```

```
    strcpy(production[7], "F=i");
```

```
    int kay; char
```

```
    done[count]; int
```

```
    ptr = -1;
```

```
    // Initializing the calc_first array
```

```
    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;

```



```

// Adding c to the calculated list

done[ptr] = c;

printf("\n First(%c) = { ", c);

calc_first[point1][point2++] = c;

// Printing the First Sets of the
grammar      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;

// Initializing the calc_follow array
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;

```




```

        // Checking if Follow of ck
        // has already been calculated
        for(kay = 0; kay <= ptr; kay++)
            if(ck == donee[kay])
                xxx = 1;

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

        // Function
        call
        follow(ck);
        ptr += 1;

        // Adding ck to the calculated
        list donee[ptr] = ck; printf("
        Follow(%c) = { ", ck);
        calc_follow[point1][point2++] =
        ck;

        // Printing the Follow Sets of the
        grammar
        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;
```



```

// Adding "$" to the follow
// set of the start symbol
if(production[0][0] == c) {
    f[m++] = '$';
}
for(i = 0; i < 10; i++)
{
    for(j = 2; j < 10; j++)
    {
        if(production[i][j] == c)
        {
            if(production[i][j+1] != '\0')
            {
                // Calculate the first of the next
                // Non-Terminal in the production
                followfirst(production[i][j+1], i, (j+2));
            }
            if(production[i][j+1] == '\0' && c != production[i][0])
            {
                // Calculate the follow of the Non-
                // Terminal
                // in the L.H.S. of the production
                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))
    {
        // Recursion to calculate First of New
        // Non-Terminal we encounter after epsilon
        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;
            }
        }

    //Including the First set of the
    // Non-Terminal in the Follow
    of // the original query
    while(calc_first[i][j] != '!')
    {

```



```

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

```



OUTPUT:

```
First(E) = { (, i, }
First(R) = { +, #, }
First(T) = { (, i, }
First(Y) = { *, #, }
First(F) = { (, i, }

-----

Follow(E) = { $, ), }
Follow(R) = { $, ), }
Follow(T) = { +, $, ), }
Follow(Y) = { +, $, ), }
Follow(F) = { *, +, $, ), }

-----
Process exited after 1.001 seconds with return value 8
Press any key to continue . . .
```

Result and Discussion:

- We have successfully studied follow and first.
- We have implemented this logic in c.
- We have also understood how it will help to generate parse tree.

Learning Outcomes: The student should have the ability

- toLO1: 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

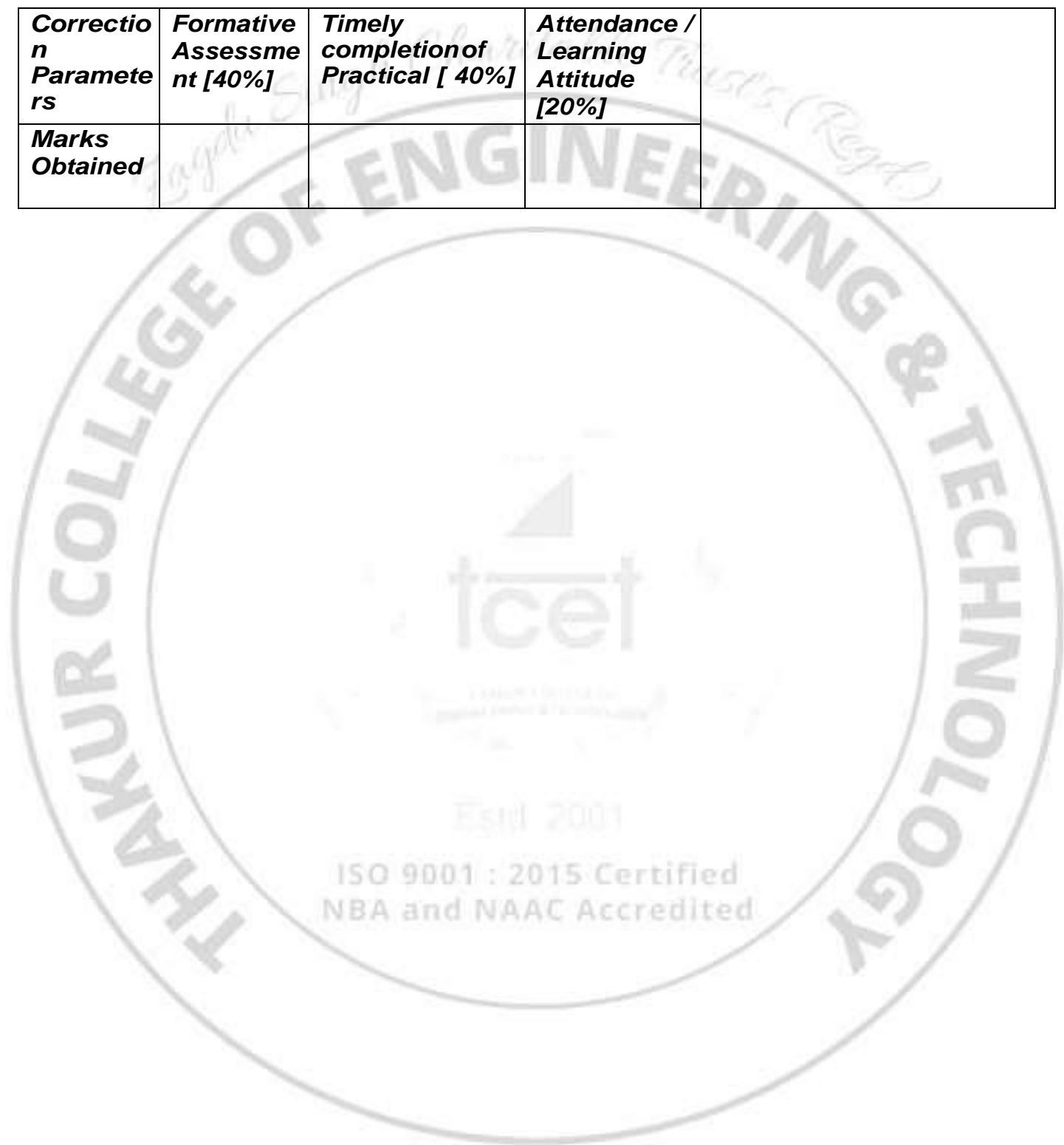
Course Outcomes: Upon completion of the course students will be able to analyze the analysis and synthesis phase of compiler for writhing application programs and construct different parsers for given context free grammars.

Conclusion: Successfully implemented follow and first in c.



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: Students 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, the operation can be performed at compilation time.

Elimination of common subexpressions:

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 the program which is executed very frequently to another part of the program, which is executed fewer times. For example, Loop optimization moves 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 operation by an occurrence of a faster operation. For example, 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 principle sources of Code Optimization.

LO3: **Apply** different code optimization techniques for increasing efficiency of the 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				

Implementation

1. Constant Folding Method

A. Code

```
import time
```

```
# Function for constant folding  
def cf(code):
```

```
    s = time.time()  
    o = ["=", "*", "+", "-", "/", "**"]  
    a = []  
    ans = code.split(" ")
```

```
    for i in ans:  
        if not i.isalpha() and i not in o:  
            a.append(eval(i))  
        else:  
            a.append(i)
```

```
    a = list(map(str,a))  
    print("".join(a))  
    e = time.time()  
    print(e - s)
```

```
print("\nImplementation")  
print(".....")  
code = input("\nEnter a Code Line: ")  
cf(code)  
print("\n")
```

B. Output

```
Run: exp2spcc
C:\Users\Manushi\PycharmProjects\practice\venv\Scripts\python.exe C:/Users/Manushi/PycharmProjects/practice/exp2spcc.py

Implementation

Enter a Code Line: a = 22/7 + 31
a=3.142857142857143+31
0.0

Process finished with exit code 0
```

2. Code Optimization Technique

A. Code

```
def cpo(code):
```

```
    d = {}
```

```
    for i in code:
```

```
        i = i.split("=")
```

```
        d[i[0]] = i[1]
```

```
    ans = []
```

```
    l = len(code)
```

```
    a = ""
```

```
    for t in code[l-1][1:]:
```

```
        if(t in list(d.keys())):
```

```
            a = a + d[t]
```

```
        else:
```

```
            a = a + t
```

```
    a = a.split("=")
```

```
    print(code[l-1][0], "=", eval(a[1]))
```

```
    code = []
```

```
    print("\nImplementation")
```

```
    print(".....")
```

```
    n = int(input("Enter no of line: "))
```

```
    for t in range(n):
```

```
        code.append(input("Enter line: "))
```

```
    cpo(code)
```


B. Output



The screenshot shows a terminal window with the following output:

```
Run: exp2bspcc  
C:\Users\Manushi\PycharmProjects\practice\venv\Scripts\python.exe C:/Users/Manushi/PycharmProjects/practice/exp2bspcc.py  
Implementation  
Enter no of line: 4  
Enter line: a=6  
Enter line: b=8  
Enter line: c=2  
Enter line: d=a+b+c  
d = 16  
Process finished with exit code 0
```



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

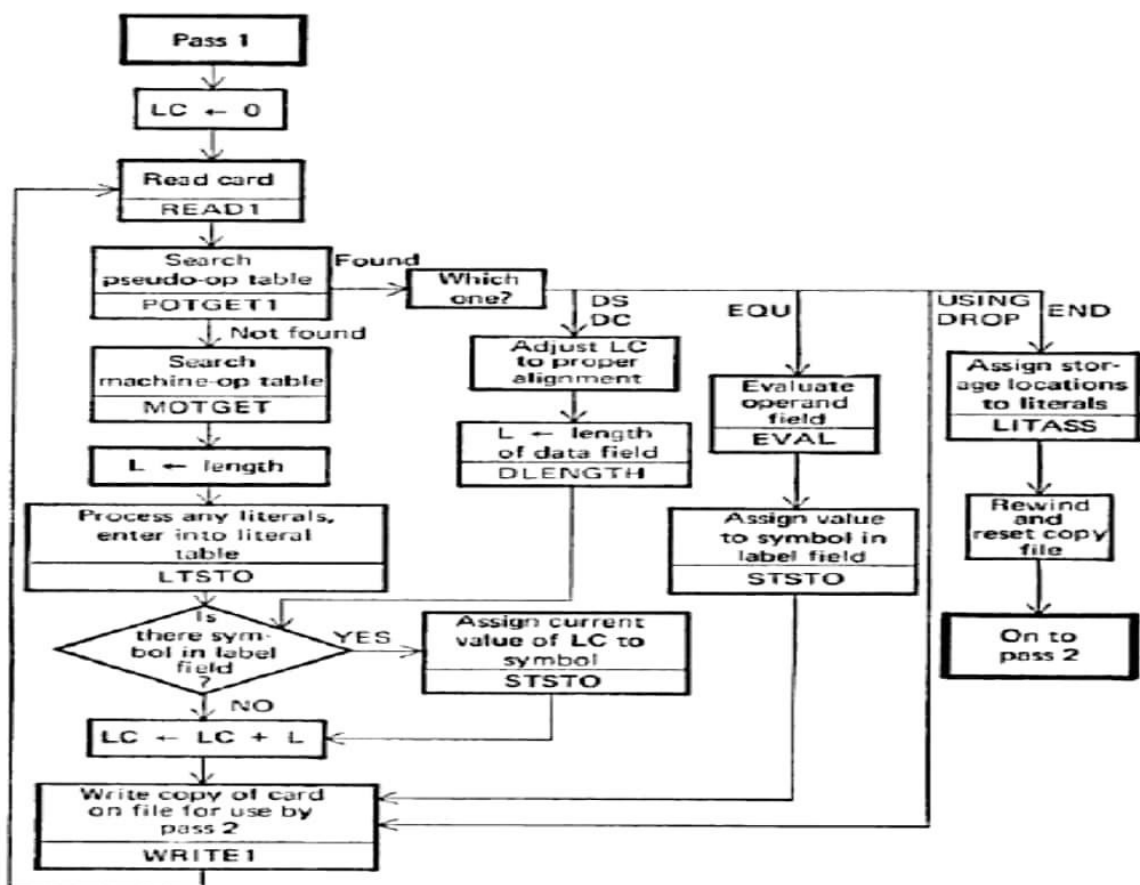
1. Initialize LC to 0
2. Read instruction
3. Search for pseudo-op table and process it.
 - a. If its a USING & DROP pseudo-op then pass it to pass2 assembler
 - b. If its a DS & DC then Adjust LC and increment LC by L
 - c. If its EQU then evaluate the operand field and add value of symbol in symbol 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

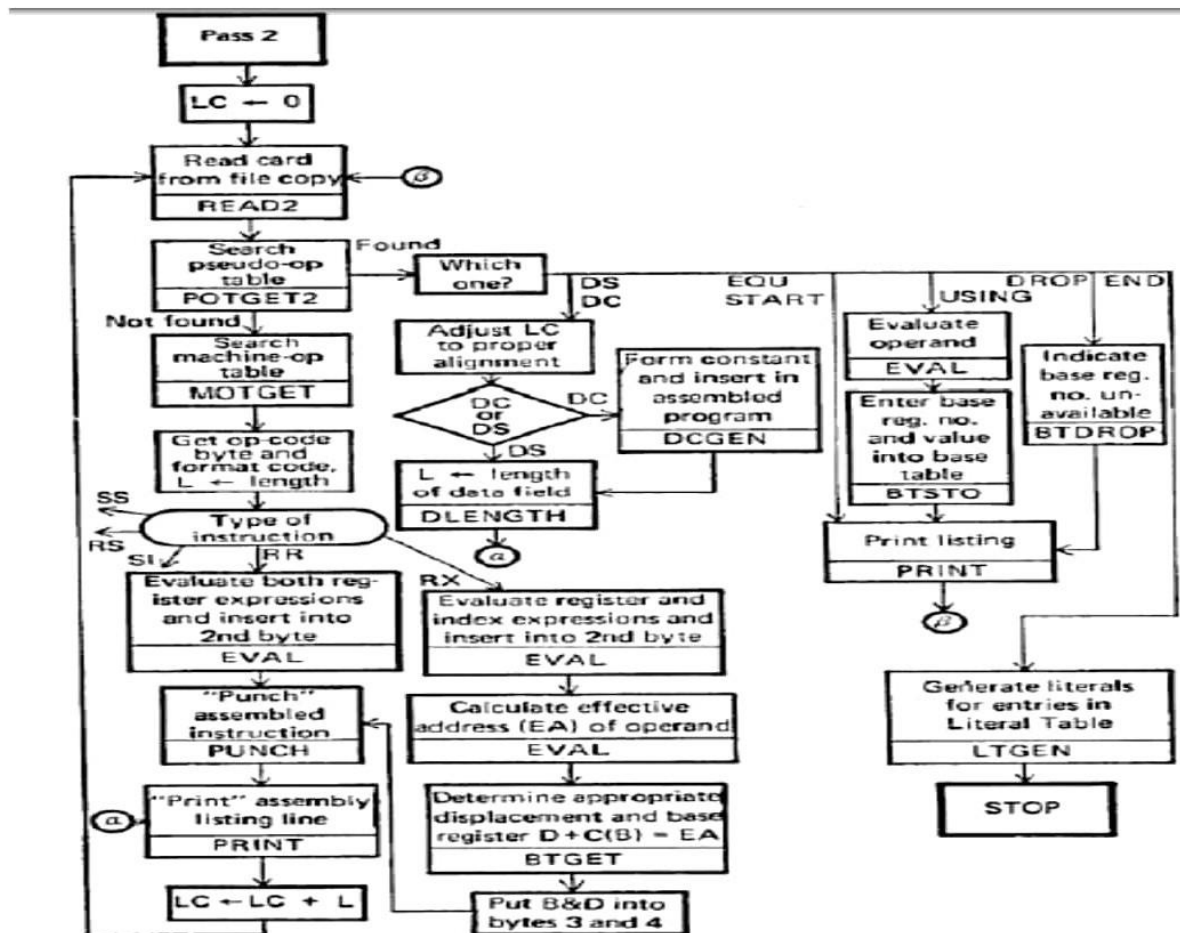
6. Set $L = \text{length}$
7. Check for type of instruction
 - a. evaluate all operands and insert into second byte
 - b. increment LC by length
 - c. print listing
 - d. Write instruction to file
8. Go to step 2

Flowchart:Pass1



ISO 9001 : 2015 Certified
NBA and NAAC Accredited

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		

Output: Display as per above format.

Application: To design 2-pass assembler for X86 processor.

Design:

Code:

```
from sys import exit
```

```
motOpCode = {
```

```
    "MOV": 1,
```

```
    "A": 2,
```

```
    "S": 3,
```

```
    "M": 4,
```

```
    "D": 5,
```

```
    "AN": 6,
```

```
    "O": 7,
```

```
    "ADD": 8,
```

```
    "SUB": 9,
```

```
    "MUL": 10,
```

```
    "DIV": 11,
```

```
    "AND": 12,
```

```
    "OR": 13,
```

```
    "LOAD": 14,
```

```
    "STORE": 15,
```

```
    "DCR": 16,
```

```
    "INC": 17,
```

```
    "JMP": 18,
```

```
    "JNZ": 19,
```

```
    "HALT": 20
```

```
}
```

```
motSize = {
```

```
    "MOV": 1,
```

```
    "A": 1,
```

```
    "S": 1,
```

```
    "M": 1,
```

```
    "D": 1,
```

```
    "AN": 1,
```

```
    "O": 1,
```

```
    "ADD": 1,
```

```
    "SUB": 2,
```

```
    "MUL": 2,
```

```
    "DIV": 2,
```

```
    "AND": 2,
```

```
    "OR ": 2,
```

```
    "LOAD": 3,
```

```

"STORE": 3,
"DCR": 1,
"INC": 1,
"JMP": 3,
"JNZ": 3,
"HALT": 1
}

l = []
relativeAddress = []
machineCode = []
symbol = []
symbolValue = []
RA = 0
current = 0
count = 0
temp = []
n = int(input("Enter the no of instruction lines : "))
for i in range(n):
    instructions = input("Enter instruction line { } : ".format(i + 1))
    l.append(instructions)
l = [x.upper() for x in l]
for i in range(n):
    x = l[i]
    if "NEXT:" in x:
        s1 = ".join(x)
        a, b, c = s1.split()
        a = a[:4]
        l[i] = b + " " + c
        symbol.append(a)
        x = l[i]
    if b in motOpCode:
        value = motOpCode.get(b)
        size = motSize.get(b)
        if len(str(size)) == 1:
            temp = "000" + str(size)
        elif len(str(size)) == 2:
            temp = "00" + str(size)
        elif len(str(size)) == 3:
            temp = "0" + str(size)
        else:
            print("Instruction is not in Op Code.")
            exit(0)
        symbolValue.append(temp)
        previous = size
        RA += current

```

```

current = previous
relativeAddress.append(RA)
if c.isalpha() is True:
    machineCode.append(str(value))
else:
    temp = list(b)
    for i in range(len(temp)):
        if count == 2:
            temp.insert(i, ',')
            count = 0
        else:
            count = count + 1
    s = ".join(temp)
    machineCode.append(str(value) + "," + s)
elif " " in x:
    s1 = ".join(x)
    a, b = s1.split()
    if a in motOpCode:
        value = motOpCode.get(a)
        size = motSize.get(a)
        previous = size
        RA += current
        current = previous
        relativeAddress.append(RA)
        if b.isalpha() is True:
            machineCode.append(str(value))
        else:
            temp = list(b)
            for i in range(len(temp)):
                if count == 2:
                    temp.insert(i, ',')
                    count = 0
                else:
                    count = count + 1
            s = ".join(temp)
            machineCode.append(str(value) + "," + s)
    else:
        print("Instruction is not in Op Code.")
        exit(0)
else:
    if x in motOpCode:
        value = motOpCode.get(x)
        size = motSize.get(x)
        previous = size
        RA += current
        current = previous

```

```

        relativeAddress.append(RA)
        machineCode.append(value)
    else:
        print("Instruction is not in Op Code.")
        exit(0)
print("Symbol Table : \n")
print("\n Symbol      Value(Address)")
for i in range(len(symbol)):
    print(" {}          {}".format(symbol[i], symbolValue[i]))

print("\n Pass-1 machine code output without reference of the symbolic address : \n")
print("Relative Address      Instruction      OpCode")
for i in range(n):
    if "NEXT" in l[i]:
        print("{}          {}          {}".format(
            relativeAddress[i], l[i], machineCode[i]))
    else:
        print("{}          {}          {}".format(
            relativeAddress[i], l[i], machineCode[i]))

print("\n Pass-2 output: Machine code output \n ")
print("Relative Address      Instruction      OpCode")
for i in range(n):
    if "NEXT" in l[i]:
        for j in range(len(symbol)):
            if "NEXT" in symbol[j]:
                pos = j
                print("{}          {}          {}".format(
                    relativeAddress[i], l[i], machineCode[i], symbolValue[pos]))
            else:
                print("{}          {}          {}".format(
                    relativeAddress[i], l[i], machineCode[i]))

```

Output:

```

Enter the no of instruction lines : 6
Enter instruction line 1 : MOV R
Enter instruction line 2 : Next: ADD R
Enter instruction line 3 : DCH R
Enter instruction line 4 : 2nd Next
Enter instruction line 5 : STORE 2000
Enter instruction line 6 : HALT
Symbol Table :

Symbol      Value(Address)
NEXT        0001

Pass-1 machine code output without reference of the symbolic address :

Relative Address      Instruction      OpCode
0                     MOV R          1
1                     ADD R          8
2                     DCH R          10
3                     2nd Next      15, 0001
4                     STORE 2000    15, 20, 00
5                     HALT          20

Pass-2 output: Machine code output

Relative Address      Instruction      OpCode
0                     MOV R          1
1                     ADD R          8
2                     DCH R          10
3                     2nd Next      15, 0001
4                     STORE 2000    15, 20, 00
5                     HALT          20

```

Result and Discussion: I came to know about how to develop and design 2 pass Assembler for X86 machine.

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: After completion of practical I m able to design 2 pass Assembler successfully.

For Faculty Use

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

Experiment 04 : Two Pass MacroProcessor

NAME: KARTIK KOUNDER

Class: TE COMP B

Roll No: 25

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 *i*th 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

80 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-I 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:

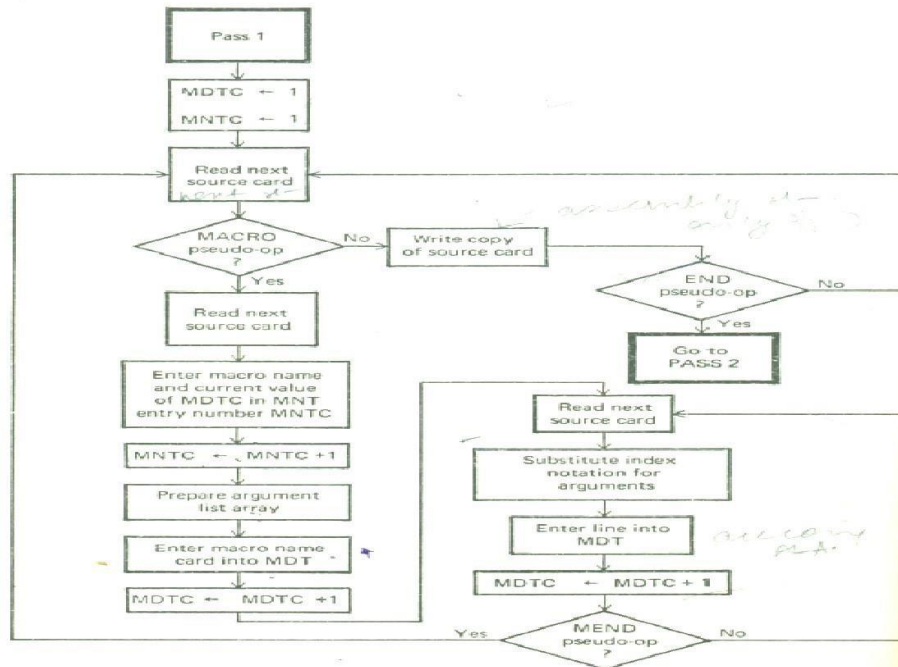
The algorithm for pass 2 tests the operation mnemonic of each input line to see if it is a name in the MNT. When a call is found:-

- 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.

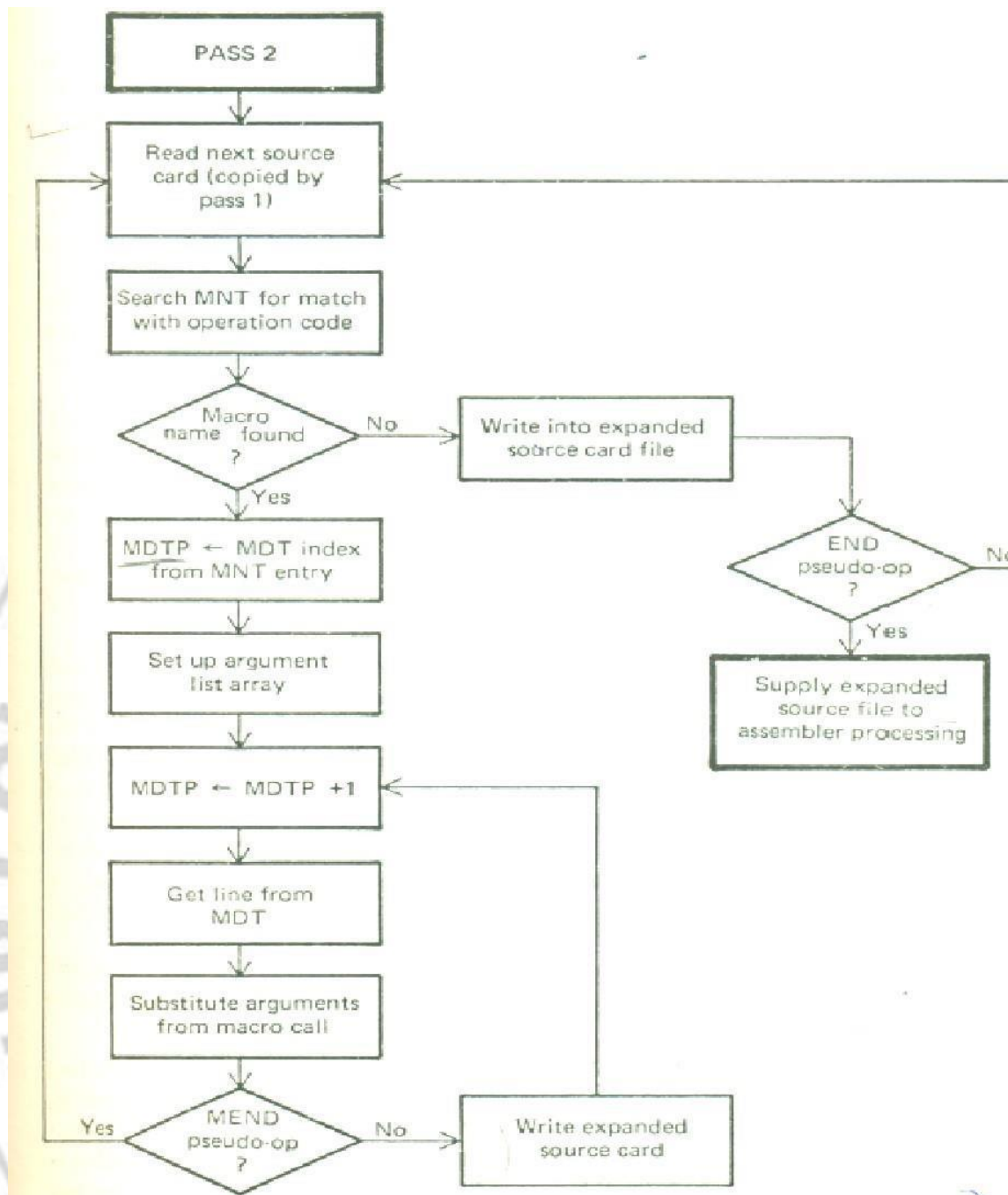
Flowchart of Pass1 & Pass2 Macro processor:

Pass 1: processing macro definitions

ESTD 2001
ISO 9001 : 2015 Certified
NBA and NAAC Accredited



Pass2 : processing macro calls and expansion



Application: To design two-pass macroprocessor.

Design:

```

def loader(ins):
    mdt=[]
    mnt=[]
    ala=[]
    esource=[]
    i=0
    temp=0
    while(i<len(ins)):

```

```

if ins[i]=="MACRO":
    dummy=[]
    i=i+1
    mnt.append(ins[i][:ins[i].index(" ")])
    ala.append(ins[i][ins[i].index(" ")+1:].split(","))
    i=i+1
    while(ins[i!="MEND"):
        dummy.append(ins[i])
        i=i+1
    dummy.append("MEND")
    mdt.append(dummy)
    i=i+1
    temp=i
else:
    i=i+1
print("-- PASS 1 RESULTS ---")
print("-- MDT --")
for a in mdt:
    print(a)
print("-- MNT --")
for b in mnt:
    print(b)
print("--- ALA ---")
print(ala)
print("-- PASS 2--")
j=0
for com in ins[:temp]:
    esource.append(com)
for com in ins[temp:]:
    com1=com.split(" ")
    if com1[0] in mnt:
        j=mnt.index(com1[0])
        for f in mdt[j]:
            esource.append(f)
    else:
        esource.append(com)
for res in esource:
    print(res)

n=int(input("Enter no of ins:"))
ins=[]
i=0
while(i<n):
    ins.append(input("Enter ins:"))
    i=i+1
loader(ins)

```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS E:\Kartik DATA\kartik\SEM 6\spcc> & C:/Python310/python.exe "e:/Kartik DATA/kartik/SEM 6/spcc/spcc4.py"
Enter no of ins:10
Enter ins: MACRO A
Enter ins:ABC A
Enter ins:STORE A
Enter ins:MEND
Enter ins:MACRO
Enter ins:INA P
Enter ins:LOAD P
Enter ins:MEND
Enter ins:ABC A
Enter ins:INA P
-- PASS 1 RESULTS ---
-- MDT --
['LOAD P ', 'MEND']
-- MNT --
INA
--- ALA ---
[['P']]
-- PASS 2--
MACRO A
ABC A
STORE A
MEND
MACRO
INA P
LOAD P
MEND
ABC A
LOAD P
MEND
PS E:\Kartik DATA\kartik\SEM 6\spcc> █
```

Result and Discussion:

- We have successfully implemented the 2 pass macro assembler.
- Also seen how does it expand and evaluate macro calls.

- Successfully understood the algorithm and the databases.

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: We have implemented the code of 2 pass macro assembler and we are able to apply the concept very well.

For Faculty Use

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

ISO 9001 : 2015 Certified
NBA and NAAC Accredited

Experiment 05 : Lexical Analyzer

Learning Objective: Students should be able to design handwritten lexical analysers.

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	$\{ \epsilon \}$
$a \in \Sigma$	$S \{ a \}$
$(r1) \mid (r2)$	$L(r1) \cup L(r2)$
$(r1) (r2)$	$L(r1) L(r2)$
$(r)^*$	$(L(r))^*$
(r)	$L(r)$

- $(r)^+ = (r)(r)^*$
- $(r)? = (r) \mid \epsilon$
- 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 digits ('.' 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 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 automata recognize regular sets.
- Which one?
 - deterministic – faster recognizer, but it may take more space
 - non-deterministic – slower, but it may take less space
 - Deterministic automata 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 \rightarrow NFA \rightarrow DFA (two steps: first to NFA, then to DFA)

Algorithm2: Regular Expression \rightarrow 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 ϵ -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()
{
    int a,b;
    printf("Hello");
    getch();
}
```

Output:

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

Application: To design lexical analyzer.

Design:

lexical.txt → kartik int maximum (x , Y { }) \$.

code :

```
import re

keyword = [
    'break', 'case', 'char', 'const', 'continue', 'default', 'do', 'int', 'else', 'enum', 'extern', 'float', 'for', 'goto', 'if', 'long', 'register', 'return', 'short', 'signed', 'sizeof', 'static', 'switch', 'typedef', 'union', 'unsigned', 'void', 'volatile', 'while'
]

built_in_functions = ['clrscr()', 'printf()', 'scanf()', 'getch()', 'main()']
```

```

operators = ['+', '-', '*', '/', '%', '==', '!=', '>', '<', '>=', '<=', '&&', '||', '!', '&', '|', '^', '~', '>>', '<<', '=', '+=', '-=', '*=']

specialsymbol = ['@', '#', '$', '_', '!']

separator = [',', ':', ';', '\\n', '\\t', '{', '}', '(', ')', '[', ']']

file = open('lexical.txt', 'r+')

contents = file.read()

splitCode = contents.split() #split program in word based on space

length = len(splitCode) # count the number of word in program

for i in range(0, length):

    if splitCode[i] in keyword:

        print("Keyword -->", splitCode[i])

        continue

    if splitCode[i] in operators:

        print("Operators --> ", splitCode[i])

        continue

    if splitCode[i] in specialsymbol:

        print("Special Operator -->", splitCode[i])

        continue

```

```
if splitCode[i] in built_in_functions:

    print("Built_in Function -->",splitCode[i])

    continue

if splitCode[i] in separator:

    print("Separator -->",splitCode[i])

    continue

if re.match(r'(#include*).*', splitCode[i]):

    print ("Header File -->", splitCode[i])

    continue

if re.match(r'^[-+]?[0-9]+$',splitCode[i]):

    print("Numerals --> ",splitCode[i])

    continue

if re.match(r"^[^\d\W]\w*\Z", splitCode[i]):

    print("Identifier --> ",splitCode[i])
```

Result and Discussion:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Windows PowerShell

Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell <https://aka.ms/pscore6>

PS E:\Kartik DATA\kartik\SEM 6\spcc> & C:/python/python.exe "e:/Kartik DATA/kartik/SEM 6/spcc/exp5.py"

Identifier --> kartik

Keyword --> int

Identifier --> maximum

Separator --> (

Identifier --> x

Separator --> ,

Identifier --> y

Separator --> {

Special Operator --> \$

PS E:\Kartik DATA\kartik\SEM 6\spcc> █

Learning Outcomes: The student should have the ability to

LO1: Appreciate the role of lexical analyzer in compiler design

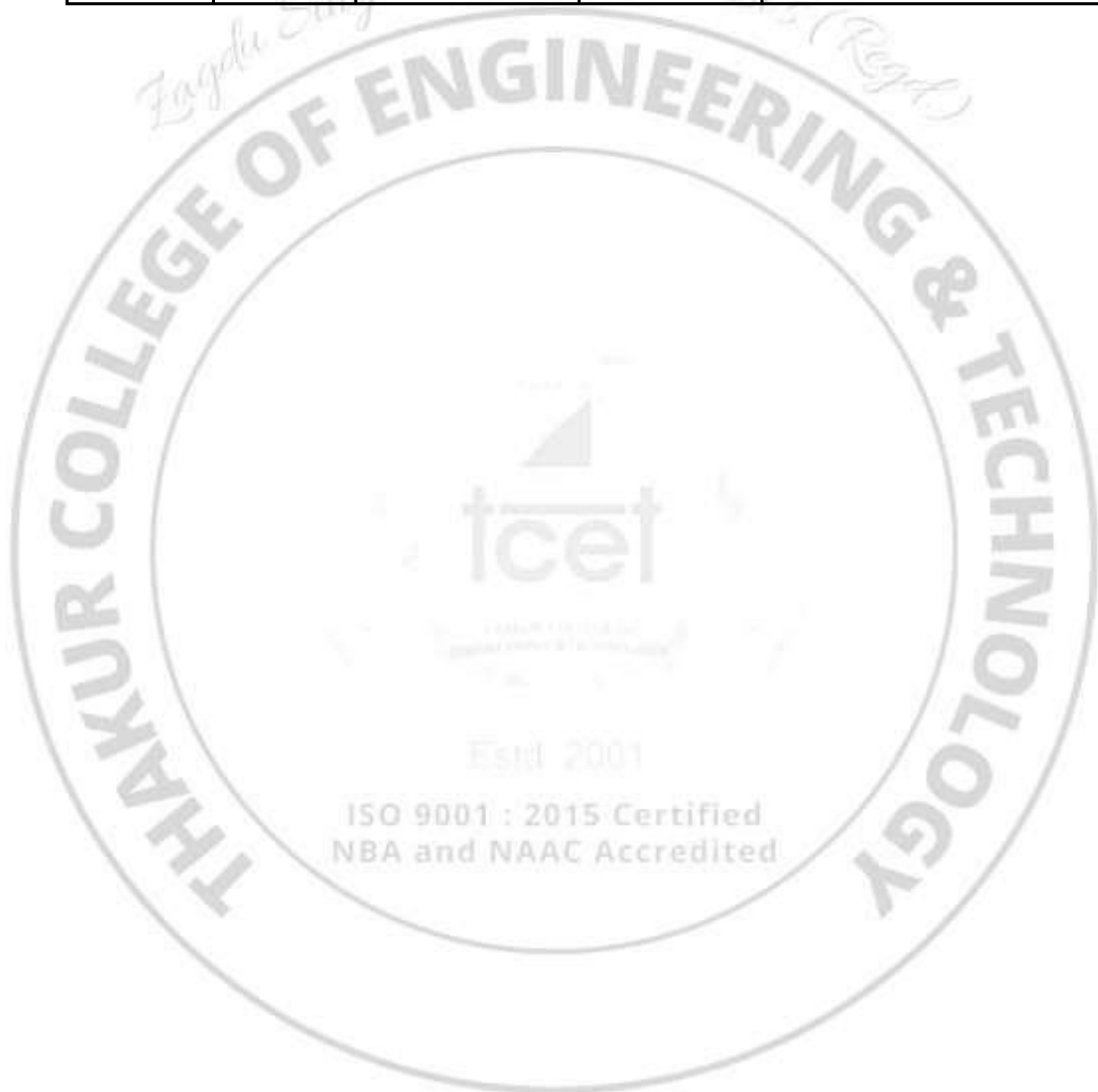
LO2: Define role of lexical analyzer.

Course Outcomes: Upon completion of the course students will be able to design handwritten lexical analyzers using HL programming language.

Conclusion: I will be able to design handwritten lexical analysers. Lexical analyzer practice is performed successfully .Lexical analysis, lexing or tokenization is the process of converting a sequence of characters into a sequence of tokens. A program that performs lexical analysis may be termed a lexer, tokenizer, or scanner, although scanner is also a term for the first stage of a lexer.

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.

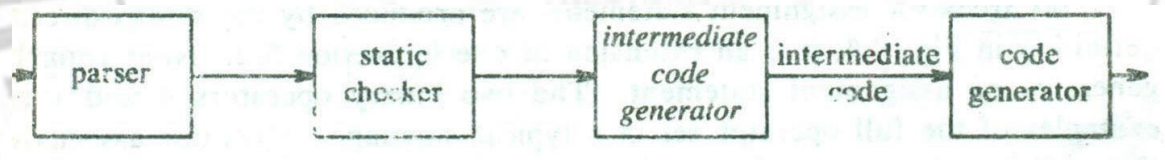
Tools: Jdk1.8, Turbo C/C++, Python, Notepad++

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

$$x := y \text{ op } z$$

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

$$t_1 := y * z$$
$$t_2 := x + t_1$$

where t1 and t2 are compiler-generated temporary names.

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

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

$$t_1 := -c$$
$$t_2 := b * t_1$$
$$t_3 := -c$$
$$t_4 := b * t_3$$
$$t_5 := t_2 + t_4$$
$$a := t_5$$

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:

1. Assignment statements of the form $x = y \text{ op } Z$, where op is a binary arithmetic or logical operation.

2. Assignment instructions of the form $x := \text{op } y$, 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 $\text{goto } L$. The three-address statement with label L is the next to be executed.

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

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

$\text{param } X_1$

$\text{param } X_2$

$\text{param } X_n$

$\text{call } p, n$

generated as part of a call of the procedure $p(X_1, X_2, \dots, X_n)$. The integer n indicating the number of actual parameters in " $\text{call } p, n$ " 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. Threesuch 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 $x := y \text{ op } z$ 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 1, arg 2, and result are normally pointers to thesymbol-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.

	<i>op</i>	<i>arg 1</i>	<i>arg 2</i>	<i>result</i>
(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

	<i>op</i>	<i>arg 1</i>	<i>arg 2</i>
(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).

	<i>op</i>	<i>arg 1</i>	<i>arg 2</i>
(0)	[]=	x	i
(1)	assign	(0)	y

(a) $x[i] := y$

	<i>op</i>	<i>arg 1</i>	<i>arg 2</i>
(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.

	<i>statement</i>		<i>op</i>	<i>arg 1</i>	<i>arg 2</i>
(0)	(14)	(14)	uminus	c	
(1)	(15)	(15)	*	b	(14)
(2)	(16)	(16)	uminus	c	
(3)	(17)	(17)	*	b	(16)
(4)	(18)	(18)	+	(15)	(17)
(5)	(19)	(19)	assign	a	(18)

Indirect triples representation of three-address statements

Input:

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

Output:

	<i>op</i>	<i>arg 1</i>	<i>arg 2</i>	<i>result</i>
(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

	<i>op</i>	<i>arg 1</i>	<i>arg 2</i>
(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.

ISO 9001 : 2015 Certified
NBA and NAAC Accredited

Code :

```
from tabulate import tabulate
```

```
def threeaddr(s):
```

```
    l=s.split(" ")
```

```
    l=l[2:]
```

```
    op=['+', '-', '*', '/', '^']
```

```
    arg1=[]
```

```
    arg2=[]
```

```
    res=[]
```

```
    oper=[]
```

```
    n=(len(l))//2
```

```
    if(l[n] not in op):
```

```
        while(l[n] not in op):
```

```
            n=n-1
```

```
            p1=l[:n]
```

```
            p2=l[n+1:]
```

```
            ind=1
```

```
            '''
```

```
            oper.append('OPERATOR:')
```

```
            arg1.append('ARGUMENT 1:')
```

```
            arg2.append('ARGUMENT2:')
```

```
            res.append('RESULT:')
```

```
            '''
```

```

if(len(l)==3):

    oper.append(l[n])

    arg1.append(l[0])

    arg2.append(l[2])
    res.append("t"+str(ind))
    oper.append("=")
    arg1.append(s[0])
    arg2.append("t"+str(ind))
    res.append("t"+str(ind+1))
    ans=[]

    z1=zip(oper,arg1,arg2,res)

    for a1,a2,a3,a4 in list(z1):

        aq=[]

        aq.append(a1)

        aq.append(a2)

        aq.append(a3)

        aq.append(a4)

        ans.append(aq)

    print("QUADRAPLE TABLE:")

    print(tabulate(ans,headers=["OPERATORS","ARG 1","ARG 2","RESULT"],tablefmt='orgtbl'))

else:

    m=0

    for i in p1:

        if(i[0] in op and len(i)>1):

```

```

oper.append("unary"+i[0])

arg1.append(i[1])

arg2.append("nill")

res.append("t"+str(ind))

ind=ind+1

if(i in op and len(i)==1):

oper.append(i)

arg1.append(p1[m-1])

#print(p1.index(i)+1)

arg2.append(p1[m+1])

res.append("t"+str(ind))

my="t"+str(ind)

ind=ind+1

m=m+1

j=0

for i in p2:

    if(i[0] in op and len(i)>1):

oper.append("unary"+i[0])

arg1.append(i[1])

arg2.append("nill")

res.append("t"+str(ind))

ind=ind+1

if(i in op and len(i)==1):

```

```
oper.append(i)

arg1.append(p2[j-1])

arg2.append(p2[j+1])

res.append("t"+str(ind))

you="t"+str(ind)

ind=ind+1

j=j+1

oper.append(l[n])

arg1.append(my)

arg2.append(you)

res.append("t"+str(ind))

oper.append("=")

arg1.append(s[0])

arg2.append("t"+str(ind))

res.append("t"+str(ind+1))

z=zip(oper,arg1,arg2,res)

ans=[]

for a1,a2,a3,a4 in list(z):

    aq=[]

    aq.append(a1)

    aq.append(a2)

    aq.append(a3)

    aq.append(a4)

    ans.append(aq)

print("QUADRAPLE TABLE:")
```

```
print(tabulate(ans, headers=["OPERATORS", "ARG 1", "ARG 2", "RESULT"], tablefmt='orgtbl'))
```

```
# print(a1,a2,a3,a4)
```

```
s="a = b + c"
```

```
print("\nCode : ", s, "\n")
```

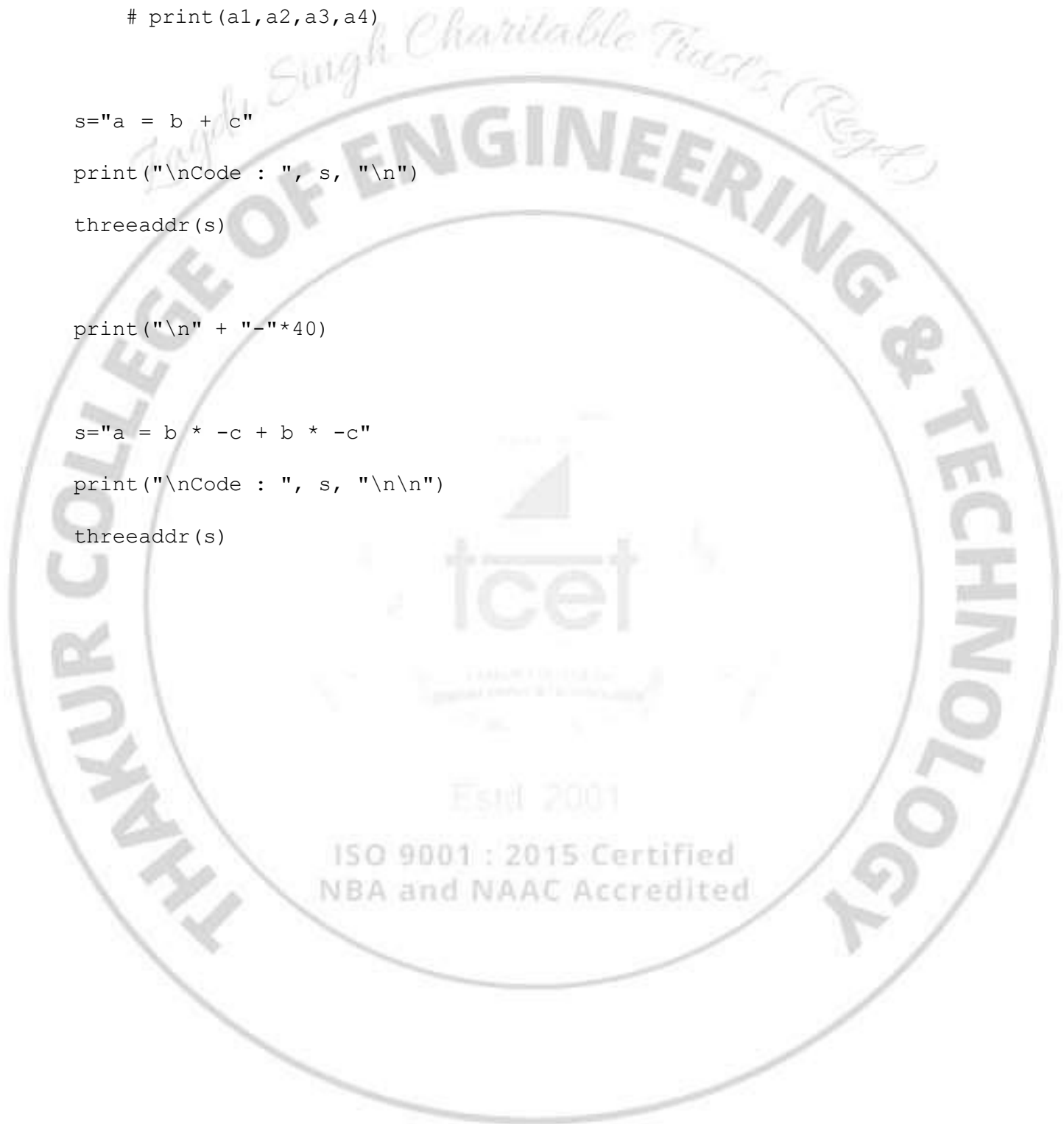
```
threeaddr(s)
```

```
print("\n" + "-"*40)
```

```
s="a = b * -c + b * -c"
```

```
print("\nCode : ", s, "\n\n")
```

```
threeaddr(s)
```



Output :

Code : $a = b + c$

QUADRAPLE TABLE:

OPERATORS	ARG 1	ARG 2	RESULT
+	b	c	t1
=	a	t1	t2

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

QUADRAPLE TABLE:

OPERATORS	ARG 1	ARG 2	RESULT
*	b	-c	t1
unary-	c	nill	t2
*	b	-c	t3
unary-	c	nill	t4
+	t1	t3	t5
=	a	t5	t6

Result and Discussion:

- Understood the role of Intermediate Code Generator in Compiler Design
- Understood the working of Intermediate Code Generator
- Demonstrated the working of Intermediate Code Generator using a python program

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: Successfully understood the role of Intermediate Code Generator in Compiler Design and demonstrated its working using a Python program.

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained		ISO 9001 : 2015 Certified NBA and NAAC Accredited		

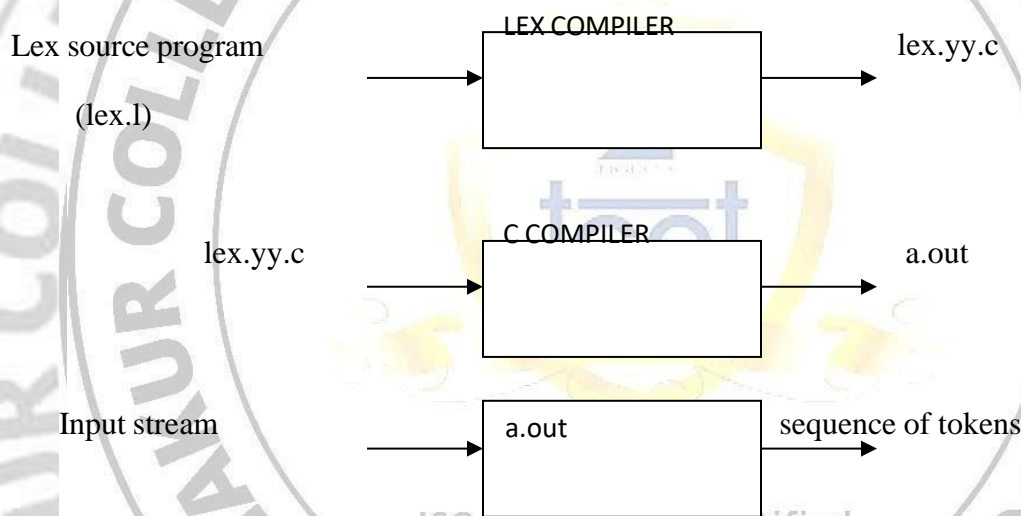
Experiment 07 : Lex Tool

Learning Objective: Student should be able to build 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 proceeds 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 *yylen*

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

1. *yytext* is a variable that is a pointer to the first character of the lexeme.
2. *yylen* 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 yleng

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 lex.yy.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 */
%%

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

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

void yywrap () { }                    /* 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: `int yylex()`

9. User program:

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

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

```
#include "tokens.h"
```

```
int yylex (); // 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  
}
```

Output Screenshots



TCET

DEPARTMENT OF COMPUTER ENGINEERING (COMP)

[Accredited by NBA for 3 years, 3rd Cycle Accreditation w.e.f. 1st July 2019]

Choice Based Credit Grading System with Holistic Student Development (CBCGS - H 2019)

Under TCET Autonomy Scheme - 2019



```
Open Save  
%{  
#define Header 3  
#define Number 1  
#define Key 2  
#define ID 7  
#define NewLine 4  
#define Punc 6  
#define Comment 5  
%}  
  
%%  
[0-9]+|[0-9]+\.[0-9]+ {return Number;}main|int|char|int|void {return key;}  
[a-zA-Z]+[a-zA-Z0-9]* {return ID;}  
\"[^\"]\\n\" {return NewLine;}  
\\<[a-z]\\.[h]> {return Header;}  
[!@#$%^&*(){};\",./<>?+=-] {return Punc;}  
\\/\\/\\/\\/\\/[a-zA-Z]+[a-zA-Z0-9]* {return Comment;}  
%%  
  
#include<stdio.h>  
int main(int argc,char *argv[])  
{  
  int val; while(val=yylex())  
  {  
    switch(val)  
    {  
      case 1:  
        printf(\"\\n%s - Number\",yytext);break;  
      case 2:  
        printf(\"\\n%s - Keyword\",yytext);break;  
      case 3:  
        printf(\"\\n%s - Header File\",yytext);break;  
      case 4:  
        printf(\"\\nNew Line\");break;  
      case 5:  
        printf(\"\\n%s - Comment\",yytext);break;  
      case 6:  
        printf(\"\\n%s - Symbol\",yytext);break;  
      case 7:  
        printf(\"\\n%s - Identifier\",yytext);break;  
    }  
  }  
}
```

Lex Tab Width: 8 Ln 45, Col 2 INS

Experiment 08 : Code Generator

Learning Objective: Student should be able to apply code generator for target machine architecture.

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

Theory:

Code Generator: It takes an input from an intermediate representation of the source program and produces as output an equivalent target program.

Code-Generation Algorithm:

Code-Generation algorithm takes as a input a sequence of three address statements constituting a basic block. For each three address statement of the form $X = Y \text{ op } Z$ we perform the following actions:

- 1 Invoke a function `getreg` to determine the location L where the result of the computation $Y \text{ op } Z$ should be stored. L will be a register or memory location.
- 2 Consult the address descriptor for y to determine y' , the current location of y . Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L , generate the instruction `MOV y' , L` to place a copy of y in L .
- 3 Generate the instruction `OP z' , L` where z' is a current location of z . Prefer register entry of z . update the address descriptor of x to indicate that x is in location L . If L is a register, update its descriptors to indicate that it contains the value of x , and remove x from all other register descriptors.
- 4 If the current values of y and/or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of $x = y \text{ op } z$, those registers no longer will contain y and/or z , respectively.

The Function `getreg`

The function `getreg` returns the location to hold the value of x for the assignment $x = y \text{ op } z$.

1. If the name y is in a register that holds the value of no other names and y is not live and has no next use after execution of $x = y \text{ op } z$, then returns the register of y is no longer in L .
2. failing 1, return an empty register for L if there is one.
3. failing 2, if x has a next use in a block, op is an operator, such as indexing, that requires a register, find an occupied register R . Store the value of R into a memory

location (by MOV R,M) if it is not already in the proper memory location M, update the address descriptor for M, and return R. If R holds the value of several variables, a MOV instruction must be generated for each variable that needs to be stored. A suitable occupied register might be one whose value is also in memory.

4. If x is not used in the block, or no suitable occupied register can be found, select the memory location of x as L.

INPUT:

A=B+C

B=A-D

C=B+C

D=B

OUTPUT:

Mov R1, B

Mov R2, C

Add R1, R2

Mov R2, D

Sub R1, R2

Mov R2, C

Add R1, R2

Mov D, R1

Design:

```
import re
reg, output = {}, []

def allocate_register(op):
    if op in reg.values():
        for key, value in reg.items():
            if value == op:
                return key
    name = 'R{}'.format(len(reg))
    reg[name] = op
    output.append('MOV {}, {}'.format(name, op))
    return name

with open('codeop.txt', 'r') as fd:
    contents = fd.read().split('\n')
input_code = [i.strip() for i in contents]
```

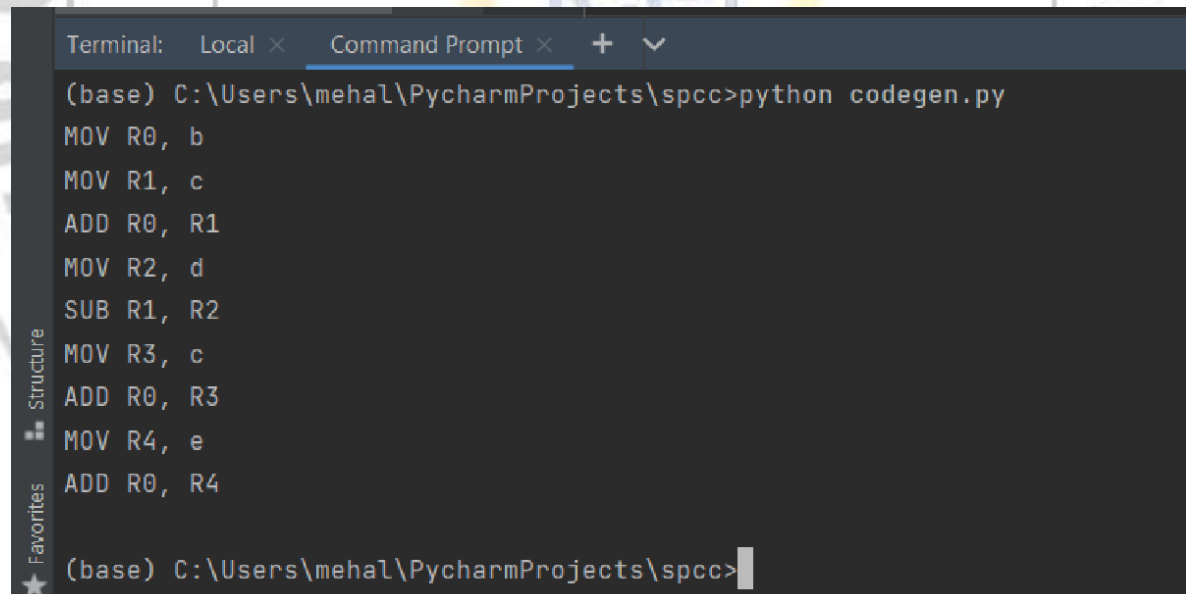
```
for line in input_code:
    line = re.split('([\+\-\*\\/\=])', line)
    lhs, eq, op1, op, op2 = line

    reg1, reg2 = allocate_register(op1), allocate_register(op2)
    if op.strip() == '+':
        op_line = 'ADD {}, {}'.format(reg1, reg2)
```

Input.txt

```
a=b+c
b=a-d
c=b+c
d=b+e
```

Output:



```
Terminal: Local x Command Prompt x + v
(base) C:\Users\mehal\PycharmProjects\spcc>python codegen.py
MOV R0, b
MOV R1, c
ADD R0, R1
MOV R2, d
SUB R1, R2
MOV R3, c
ADD R0, R3
MOV R4, e
ADD R0, R4
(base) C:\Users\mehal\PycharmProjects\spcc>
```

Result and Discussion:

The code generator within a compiler is responsible for converting intermediate code to target code. It is part of the final stages of compilation, within the overall hierarchy of a compiler it is located between the optimisation steps.

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:

We successfully applied the code generator algorithm to generate target code for the optimized code, considering target machines.

For Faculty Use

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

EXPERIMENT 9

CASE STUDY ON YACC TOOL

Aim: To study and understand yacc tool.

Description:

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.

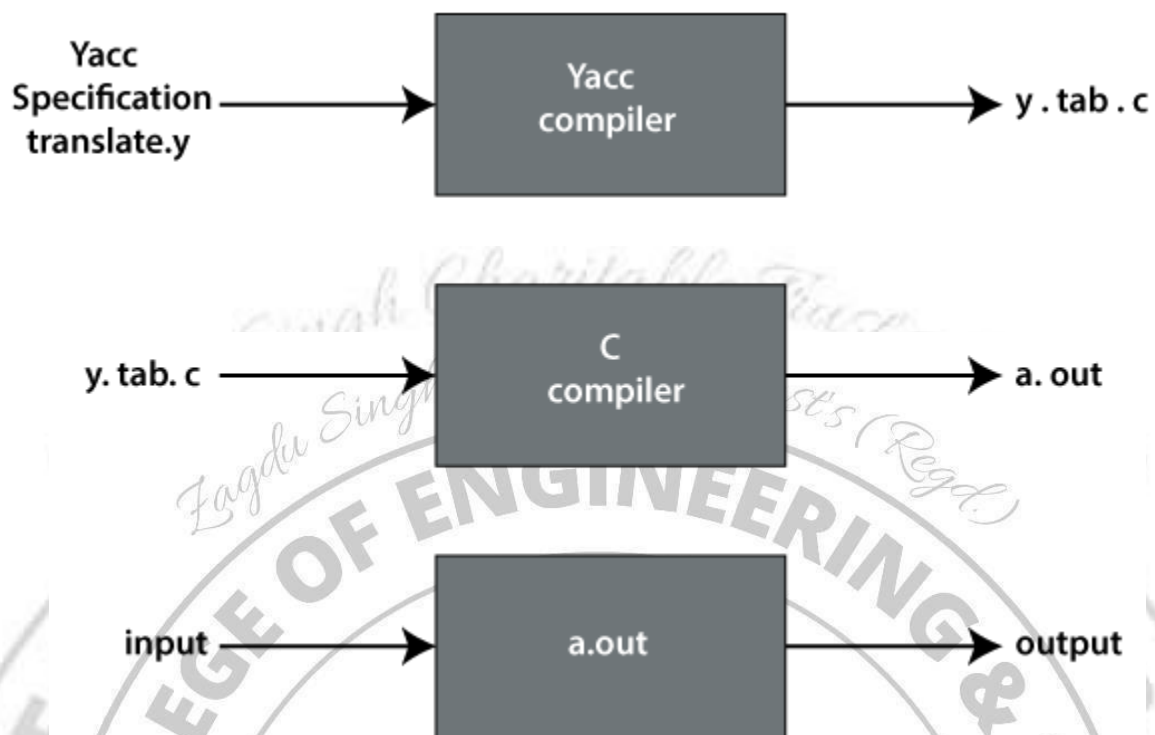
YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

If we have a file **translate.y** that consists of YACC specification, then the UNIX system command is:

YACC translate.y

This command converts the file translate.y into a C file y.tab.c. It represents an LALR parser prepared in C with some other user's prepared C routines. By compiling y.tab.c along with the ly library, we will get the desired object program a.out that performs the operation defined by the original YACC program.

The construction of translation using YACC is illustrated in the figure below:



A YACC source program contains three parts:

- **Declarations**
- **Translation rules**
- **Supporting C rules**

Declarations Part

This part of YACC has two sections; both are optional. The first section has ordinary C declarations, which is delimited by `%{` and `%}`. Any temporary variable used by the second and third sections will be kept in this part. See the below code:

```

%{
#include <ctype.h>

%}

%token DIGIT

%%
  
```

```
line      : expr '\n'      { printf("%d\n", $1); }
```

```
;
```

```
expr      : expr '+' term   { $$ = $1 + $3; }
```

```
| term
```

```
;
```

```
term      : term '*' factor { $$ = $1 * $3; }
```

```
| factor
```

```
;
```

```
factor    : '(' expr ')'    { $$ = $2; }
```

```
| DIGIT
```

```
;
```

```
%%
```

```
yylex() {
```

```
int c;
```

```
c = getchar();
```

```
if (isdigit(c)) {
```

```
    yylval = c-'0';
```

```
    return DIGIT;
```

```
}
```

```
return c;
```

```
}
```

In the above code, the declaration part only contains the include statement.

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

Declaration of grammar tokens also comes in the declaration part.

```
%token DIGIT
```


This part defines the tokens that can be used in the later parts of a YACC specification.

Translation Rule Part

After the first %% pair in the YACC specification part, we place the translation rules. Every rule has a grammar production and the associated semantic action.

A set of productions:

$\langle \text{head} \rangle \Rightarrow \langle \text{body} \rangle_1 \mid \langle \text{body} \rangle_2 \mid \dots \mid \langle \text{body} \rangle_n$

would be written in YACC as

```
 $\langle \text{head} \rangle$  :  $\langle \text{body} \rangle_1$  {<semantic action>1}  
          |  $\langle \text{body} \rangle_2$  {<semantic action>2}  
          ....  
          |  $\langle \text{body} \rangle_n$  {<semantic action>n}  
          ;
```

In a YACC production, an unquoted string of letters and digits that are not considered tokens is treated as non-terminals.

The semantic action of YACC is a set of C statements. In a semantic action, the symbol \$\$ considered to be an attribute value associated with the head's non-terminal. While \$i considered as the value associated with the ith grammar production of the body.

If we have left only with associated production, the semantic action will be performed. The value of \$\$ is computed in terms of \$i's by semantic action.

DIFFERENCE BETWEEN LEX AND YACC

Lex VERSUS Yacc

Lex	Yacc
Computer program that operates as a lexical analyzer	Parser that is used in Unix Operating System
Developed by Mike Lex and Eric Schmidt	Developed by Stephan C. Johnson
Reads the source program one character at a time and converts it into meaningful tokens	Takes the tokens as input and generates a parse tree as output

USES:

Lex and yacc are a pair of programs that help write other programs. Input to lex and yacc describes how you want your final program to work. The output is source code in the C programming language; you can compile this source code to get a program that works the way that you originally described.

You use lex and yacc to produce software that analyzes and interprets input. For example, suppose you want to write a simple desk calculator program. Such a desk calculator is easy to create using lex and yacc, and this tutorial shows how one can be put together.

The C code produced by lex analyzes input and breaks it into *tokens*. In the case of a simple desk calculator, math expressions must be divided into tokens. For example:

178 + 85

would be treated as **178**, **+**, and **85**.

The C code produced by yacc *interprets* the tokens that the lex code has obtained. For example, the yacc code figures out that a number followed by a + followed by another number means that you want to add the two numbers together.

lex and yacc take care of most of the technical details involved in analyzing and interpreting input. You just describe what the input looks like; the code produced by lex and yacc then worries about recognizing the input and matching it to your description. Also, the two programs use a format for describing input that is simple and intuitive; lex and yacc input is much easier to understand than a C program written to do the same work.

You can use the two programs separately if you want. For example, you can use lex to break input into tokens and then write your own routines to work with those tokens. Similarly, you can write your own software to break input into tokens and then use yacc to analyze the

tokens you have obtained. However, the programs work very well together and are often most effective when combined.

RESULTS:

- Thus, yacc is an automated tool that can perform parsing for the given code easily.
- Yacc works well on ubuntu based systems.
- Lex and yacc show a lot of similarities.

CONCLUSION:

Successfully studied yacc tool.

References:

1. <https://www.geeksforgeeks.org/introduction-to-yacc/>
2. <https://www.tutorialandexample.com/yacc>
3. <https://pediaa.com/what-is-the-difference-between-lex-and-yacc/>
4. <https://www.ibm.com/docs/en/zos/2.2.0?topic=yacc-uses-lex-utilities>

