Experiment 02 : Code optimization Techniques

Learning Objective: Student should be able to Analyse 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 occurence 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

Code :

```
import pandas as pd
print("One thing is noticed before starting of the program is that it is compulsory to exist an 'input.csv' file\n")
print("The formate of 'input.csv' file is that it should contains a 'left' header and a 'right' header with values indicate left and right respectively.\n")
print("Here we can't consider 'equal(=)' sign because it doesn't effect a bit of code with or without its presence.\n")
print("This program is case sensitive this means that 'd*10' and '10*d' is treated in different way\n They are not same. ")
print("The formate and input file for this program is as below..")
a=pd.read_csv("input.csv")
c=a.shape
print(a)
b=[]
for i in range(c[0]):
    for j in range(i+1,c[0]):
        if(a['right'][i]==a['right'][j]):
            for d in range(c[0]):
                b=b+[(len(a['right'][d]))]
                for z in range(b[d]):
                    if(a['right'][d][z]==a['left'][j]):
                        x=list(a['right'][d])
                        x[z]=a['left'][i]
                        l=''.join(x)
                        a['right'][d]=a['right'][d].replace(a['left'][j],a['left'][i])
            a['left'][j]=a['left'][i]
df=pd.DataFrame(a)
df.to_csv('output1.csv',index=False)
p=pd.read_csv("output1.csv")
print("After checking and putting the value of common exepression ")
print(p)
i=0
j=i+1
while(j<c[0]):
    if(p['right'][i]==p['right'][j]):
```

```
        if(p['left'][i]==p['left'][
j]):
            p.drop([j],axis=0,inpla
ce=True)
            i+=2
            j+=1
        else:
            i+=1
    j+=1
print("After elemenating the common
 expression")
df=pd.DataFrame(p)
df.to_csv('output1.csv',index=False
)
p=pd.read_csv("output1.csv")
print(p)
c=p.shape
#print(c)
count=0
i=0
j=0
h=1
while(j<c[0] and i<c[0]):
    b=[]
    b=b+[(len(p['right'][j]))]
    for z in range(b[0]):
        #print(z,j,i)
        if(p['right'][j][z]==p['lef
t'][i]):
            count=1
    j+=1
    #print(j)
    #print(c[0])
    if(j==c[0]):
        #print(count)
        if(count!=1):
            p.drop([i],axis=0,inpla
ce=True)
            df=pd.DataFrame(p)
            df.to_csv('output1.csv'
,index=False)
            p=pd.read_csv("output1.
csv")
            3#print(p)
            c=p.shape
            print(c)
        i+=1
        j=0
print("After dead code elimination"
)
print(p)
df=pd.DataFrame(p)
df.to_csv('output1.csv',index=False
)
p=pd.read_csv("output1.csv")
c=p.shape
print("The final optimized code is.
...")
for i in range(c[0]):
    print(str(p['left'][i])+"="+str
(p['right'][i]))
```

Output

```
  left right
0    a     9
1    b   c+d
2    e   c+d
3    f   b+e
4    r     f
After checking and putting the value of common exepression
left right
0    a     9
1    b   c+d
2    b   c+d
3    f   b+b
4    r     f
After eliminating the common expression
left right
```

```
0    a     9
1    b   c+d
2    f   b+b
3    r     f
(3, 2)
After dead code elimination
left right
0    b   c+d
1    f   b+b
2    r     f
The final optimized code is....
b=c+d
f=b+b
r=f
```

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

Result and Discussion: Code optimization is the process of improving the efficiency and performance of a program by reducing its resource utilization, such as CPU time, memory usage, or disk I/O. There are various techniques used for code optimization, including: Loop unrolling: This technique involves duplicating the loop body several times to reduce the number of iterations, which can improve the program's execution speed. Dead code elimination: This technique involves removing code that is never executed, which can reduce the program's size and improve its performance.

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 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:

To conclude we have successfully completed this experiment and learned about code optimization techniques

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

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:

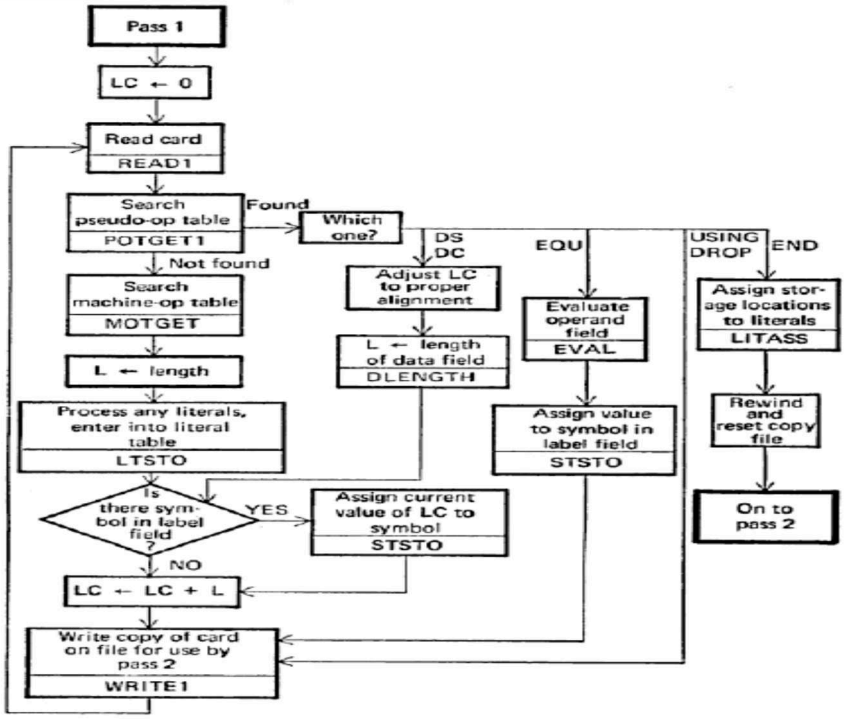
Pass 1

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

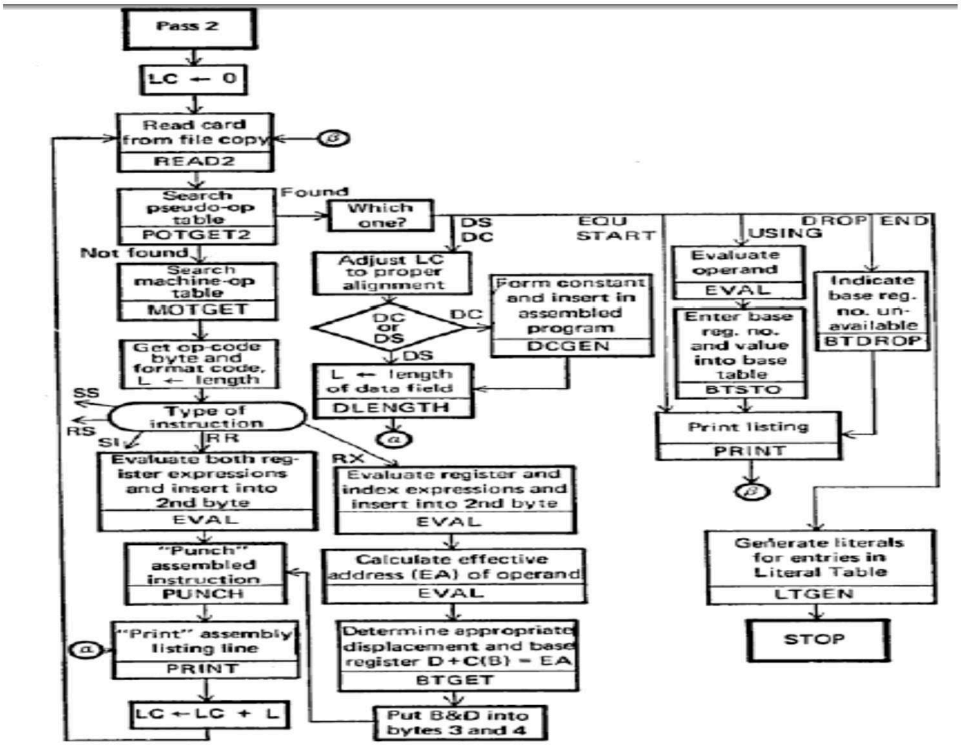
Pass 2

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
```

Output: Display as per above format.
Application: To design 2-pass assembler for X86 processor.

Design:

```
code = []
MOT_ref = {
   'L': ['L', '58', '4', 'RX'],
   'A': ['A', '5A', '4', 'RX'],
   'ST': ['ST', '50', '4', 'RX'],
   'BASR': ['BASR', '0D', '4', 'RX'],
   'BALR': ['BALR', '05', '2', 'RR'],
}
POT_ref = {
   'START': ['START', 'OPCODE FOR START'],
   'USING': ['USING', 'OPCODE FOR USING'],
   'END': ['END', 'OPCODE FOR END'],
   'DC': ['DC', 'OPCODE FOR DC'],
   'DS': ['DS', 'OPCODE FOR DS']
}
lc = 0
length = 0
MOT = []
POT = []
symbolTable = []
operands = []

# take input form file
with open('./textfile.txt') as f:
   code = f.readlines()

for i, line in enumerate(code):
   tokens = line.split(' ')
   print(f"LINE {i+1}:{tokens}")
   for token in tokens:
      if token in MOT_ref:
         length = int(MOT_ref[token][2])
         lc += length
         MOT.append(MOT_ref[token])
      elif token in POT_ref:
         if token == "DC" or token == 'DS':
            lc += 4
         POT.append(POT_ref[token])
      else:
         flag = True
         for char in token:
            if char == ',' or char in '0123456789':
               flag = False
               break
         if flag:
            symbolTable.append(tuple((token, lc, length, 'R')))
         else:
            operands.append(tuple((token[:-1].split(','), f"line: {i+1}")))

print('\n\nMOT:')
print("mnemonic\tbinary_op\tins_length\tins_format")
for x in MOT:
   print(f"{x[0]}\t\t{x[1]}\t\t{x[2]}\t\t{x[3]}")

print('\n\nPOT:')
print("mnemonic\topcode")
for x in POT:
   print(f"{x[0]}\t\t{x[1]}")

print('\n\nSymbols:')
print('symbol\tvalue\tlength\trelocation')
for x in symbolTable:
   print(f"{x[0]}\t{x[1]}\t{x[2]}\t{x[3]}")

print('\n\nOperands:', operands)
print('')
```

Result and Discussion:

```
LINE 1:['JOHN', 'START', '0\n']
LINE 2:['USING', '*,15\n']
LINE 3:['L', '1,FOUR\n']
LINE 4:['A', '1,FIVE\n']
LINE 5:['ST', '1,TEMP\n']
LINE 6:['FOUR', 'DC', "F'4\n"]
LINE 7:['FIVE', 'DC', "F'5\n"]
LINE 8:['TEMP', 'DS', '1F\n']
```

LINE 9:['END']

MOT:

mnemonic	binary_op	ins_length	ins_format
L	58	4	RX
A	5A	4	RX
ST	50	4	RX

POT:

mnemonic	opcode
START	OPCODE FOR START
USING	OPCODE FOR USING
DC	OPCODE FOR DC
DC	OPCODE FOR DC
DS	OPCODE FOR DS
END	OPCODE FOR END

Symbols:

symbol	value	length	relocation
JOHN	0	0	R
FOUR	12	4	R
FIVE	16	4	R
TEMP	20	4	R

Operands: [(['0'], 'line: 1'), (['*', '15'], 'line: 2'), (['1', 'FOUR'], 'line: 3'), (['1', 'FIVE'], 'line: 4'), (['1', 'TEMP'], 'line: 5'), (["F'4"], 'line: 6'), (["F'5"], 'line: 7'), (['1F'], 'line: 8')]

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: In these experiment we perform code is a script for parsing assembly code written in a specific syntax. It reads input from a file, identifies and categorizes different types of tokens (i.e., mnemonics, opcodes, symbols, and operands), and then outputs the results in a formatted manner.

For Faculty Use

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

Experiment 04 : Single Pass MacroProcessor

Learning Objective: Student should be able to Apply single pass Macro Processor.
Tools: Jdk1.8,Turbo C/C++, Python, Notepad++
Theory:
IMPLEMENTATION A SINGLE PASS ALGORITHM Definition of a macro within other macro is possible in case of one pass macro processor. Here the inner macro is defined only after the outer one has been called: in order to provide for any use of the inner macro, we would have to repeat the both the macro definition and the macro call passes. This can be assumed by considering that the macros are never called before they are defined.
Here we make use of additional data structures like macro definition indicator (MDI) and macro definition level counter (MDLC). The MDI and MDLC are the switches used to keep track of macro calls and macro definition.
The MDI has status "ON" during the expansion of macro call and the value "OFF" all the other times. When its value is "ON" the cards are read from the MDT and when it is "OFF" the cards are read from the input source card. The use of MDLC is used keep track of the level of macros while defining the macros. Initially it is zero and it is incremented each time a MACRO code is found within a macro. The reverse process happens in case of MEND i.e. the valued of MDLC is decremented by one each time it encounters a MEND and the process continues till the MDLC is zero i.e. the completion of macro definition.
ALGORITHM
The process of one pass macro process can be clearly understood with the help of a MAIN algorithm that make use of a sub algorithm named READ.
READ: (Macro call expansion or read a next instruction form the source input card)
1. If MDI ="OFF", then
    a. Read next source card from input file.
    b. Return to MAIN algorithm.
2. Else increment MDT pointer to next entry MDTP<-MDTP+1.
3. Get next card from MDT.
4. Substitute arguments from macro call.
5. If MEND pseudo code
    a. Then if MDLC=0, i. Then MDI<-"OFF". ii. GOTO 1.a.
    b. Else return to MAIN algorithm.
6. Else if AIF or AGO present
    a. Then process AIF or AGO and update MDTP.
    b. Return to MAIN algorithm. 7. Else return to MAIN algorithm.

MAIN: (One pass macro processor)
1. Initialize MDTC and MNTC to 1, MDI to "OFF" and MDLC to 0.
2. READ
3. Search MNT for match with operation code.

4. If macro name found
    a. MDI<-“ON”.
    b. MDTP<-MDT index from MNT entry.
    c. Setup macro call ALA.
    d. GOTO 2.
5. Else if MACRO pseudo code
    a. Then READ. //macro name line.
    b. Enter macro name and current value of MDTC in MNT entry number MNTC.
    c. Increment MNTC <- MNTC+1.
    d. Prepare macro definition ALA.
    e. Enter macro name card into MDT.
    f. MDTC<-MDTC+1.
    g. MDLC<-MDLC+1.
    h. READ.
        i. Substitute index notation for arguments in definition.
        j. Enter line into MDT.
        k. MDTC<-MDTC+1
        l. If MACRO pseudo code
            i. MDLC<-MDLC+1
            ii. GOTO 5.h.
        m. Else If MEND pseudo code i. Then MDLC<-MDLC-1
        1. If MDLC=0 the a. Then GOTO 2.
        2. Else GOTO 5.h. n. Else GOTO 5.h.
6. Write into expanded source card file.
7. If END pseudo code
    a. Then Supply expanded source file to assembler processing.
8. Else GOTO 2.

Application: To design Single pass macroprocessor for X86 processor.

Design:

```
import re

f_input = open("macro_input.txt")
inputcode = list(line.strip() for line in f_input)
MDT = []
MNT = {}
ALA_list = {}
input_for_pass_2 = []
iterator = iter(inputcode)

while True:
    try:
        line = next(iterator)
        if line == "MACRO":
            nameline = next(iterator)
            nameline = re.split('[,\s]', nameline)
            macro_name = ""
            for token in nameline:
                if "&" not in token:
                    macro_name = token
                    break
            MNT[macro_name] = len(MDT)
            ALA = {}
            arg_counter = 0
            for token in nameline:
                if token is not macro_name:
                    arg_counter += 1
                    ALA[token] = "#" + str(arg_counter)
                    nameline[nameline.index(token)] =
ALA[token]
            ALA_list[macro_name] = ALA
            MDT.append(nameline)

            while True:
                macroline = next(iterator)
                for argument in ALA.keys():
```

```
                if argument in macroline:
                    macroline = macroline.replace(argument, ALA[argument])
                MDT.append(macroline)
                if macroline == "MEND":
                    break
            else:
                input_for_pass_2.append(line)
        except StopIteration:
            break

print("\nMNT is ")
for line in MNT.items():
    print(line)
print("\nMDT is ")
for line in MDT:
    print(line)
print("\nALAs are ")
for line in ALA_list.items():
    print(line)

iterator = iter(input_for_pass_2)
print("\nFinal Output is ")

while True:
    try:
        line = next(iterator)
        line = re.split('[,\s]', line)
        if any(word in line for word in MNT.keys()):
            macroname = ""
            if line[0] in MNT.keys():
                macroname = line[0]
            else:
                macroname = line[1]
            label = line[0]
            actual_args = []
            for token in line:
                if not token == macroname:
                    actual_args.append(token)

            ALA = ALA_list[macroname]
            ALA = {val: key for key, val in ALA.items()}

            formal_args = sorted(list(ALA.keys()))

            for i in range(len(formal_args)):
                ALA[formal_args[i]] = actual_args[i]

            MDTP = MNT[macroname] + 1

            while "MEND" not in MDT[MDTP]:
                line = MDT[MDTP]

                for formal_arg, actual_arg in ALA.items():
                    line = line.replace(formal_arg, actual_arg)

                print(line)
                MDTP += 1

        else:
            print(" ".join(line))

    except StopIteration:
        break
```

macro_input.txt

```
MACRO
INCR &ARG1
L AX,&ARG1
A AX,1
MEND
MACRO
FOOBAR &ARG1,&ARG2
L AX,&ARG1
L BX,&ARG2
ST AX,BX
MEND
```

```
MACRO
&LAB HARAMBE &ARG1
&LAB SR &ARG1,1
RR &ARG1,2
MEND
START 0
INCR 69
FOOBAR 69,96
LOOP HARAMBE 69
DC F'69'
END
```

Result and Discussion:

MNT is
('INCR', 0)
('FOOBAR', 4)
('HARAMBE', 9)

MDT is
['INCR', '#1']
L AX,#1
A AX,1
MEND
['FOOBAR', '#1', '#2']
L AX,#1
L BX,#2
ST AX,BX
MEND
['#1', 'HARAMBE', '#2']
#1 SR #2,1
RR #2,2
MEND

ALAs are
('INCR', {'&ARG1': '#1'})
('FOOBAR', {'&ARG1': '#1', '&ARG2': '#2'})
('HARAMBE', {'&LAB': '#1', '&ARG1': '#2'})
input pass 2 ['START 0', 'INCR 69', 'FOOBAR 69,96', 'LOOP HARAMBE 69', "DC F'69'", 'END']

Final Output is
START 0
L AX,69
A AX,1
L AX,69
L BX,96
ST AX,BX
LOOP SR 69,1
RR 69,2
DC F'69'
END

Learning Outcomes: The student should have the ability to

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

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

Conclusion:

The code is implementing a macro processor using two-pass algorithm in Python. It reads macro definitions, stores them in MDT and MNT, and replaces macro calls with their expanded version using the ALA.

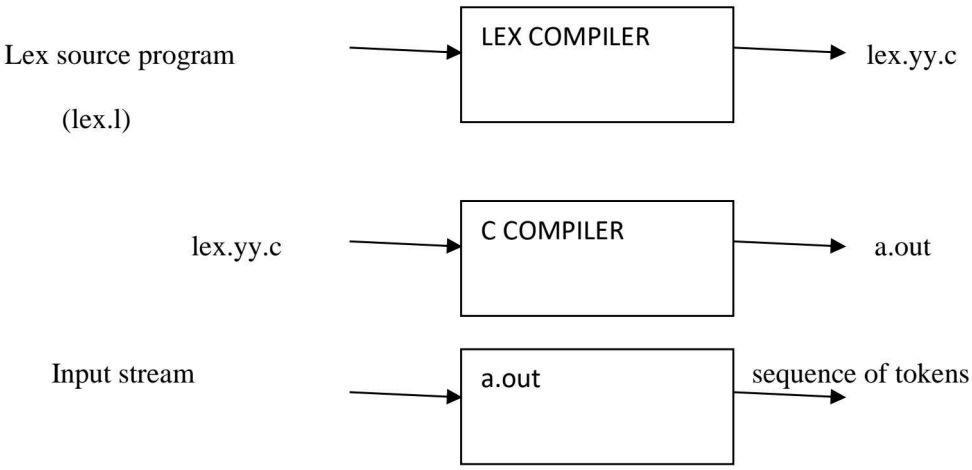
For Faculty Use

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

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 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
}
```

Code:

```
%{
#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;
default:
printf("Invalid choice");
break;
}
}
}
int yywrap(){return(1);
```

Output:

```
Command Prompt - program

float - Identifier
a - Identifier
; - Symbol
int b,c

int - Keyword
b - Identifier
, - Symbol
c - Identifier
; - symbol

; - Symbol
- - Symbol
symbol - Identifier
c = a/b;

c - Identifier
= - Symbol
a - Identifier
/ - Symbol
b - Identifier
; - Symbol
getch();

getch - Identifier
( - Symbol
) - Symbol
; - Symbol
```

Result and Discussion:

- Lex tool itself is a lex compiler. Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- Lex is a tool in lexical analysis phase to recognize tokens using regular expression.
- The function yylex() is the main flex function which runs the Rule Section and extension (.l) is the extension used to save the programs.
- The tokens like – Keywords, Identifiers, Symbols, Digits, Comments, Header File are identified using FLEX.

A Lex program consists of three parts: Declarations, Translation Rules, Auxiliary Procedures.

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 analyze the analysis and synthesis phase of compiler for writhing application programs and construct different parsers for given context free grammars.

Conclusion:

Flex tool was used in the experiment to implement Lexical Analyzer. A .l file is run using flex which in turn converts it into C program, in a file that is always named lex.yy.c. The C compiler compiles lex.yy.c file into an executable file called a.out. The output is obtained stating all the tokens present in the given input.

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:

Cal.l file:

```
%{
#include &amp;quot;y.tab.h&amp;quot;
%}
%%
[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.y File:

```
%{
#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);
}
```

Result and Discussion:

```
>>> 2+3*5/2
9.5
>>> 2+3
5
>>> 2-3
-1
>>> 2*3
6
```

Learning Outcomes: The student should have the ability to

LO1 **Define** Context Free Grammar.
LO2: ***Describ*e** 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: In conclusion, YACC is a powerful tool for generating parsers. It uses a grammar specification file to generate code that can parse input according to the specified grammar. With YACC, you can easily create complex parsers for programming languages, data formats, and more.

For Faculty Use

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