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 $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X)

   c) if X is a non-terminal and $X \rightarrow Y_1Y_2...Y_n$ is a production, add FIRST(Y_i) to FIRST(X) if the preceding Y_js contain ε in their FIRSTs

2. **Algorithm to Compute FOLLOW as follows:**
   a) FOLLOW(S) contains EOF

   b) For productions $A \rightarrow \alpha B\beta$, everything in FIRST ($\beta$) except ε goes into FOLLOW (B)

   c) For productions $A \rightarrow \alpha B$ or $A \rightarrow \alpha B\beta$ where FIRST (β) contains ε, FOLLOW(B) contains everything that is in FOLLOW(A)

Original grammar:

$E \rightarrow E+E$
$E \rightarrow E*E$
$E \rightarrow (E)$
$E \rightarrow id$

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.

$E \rightarrow E+T$
$T \rightarrow T*F$
$F \rightarrow (E)$

$F \rightarrow id$

Grammar is left recursive hence Remove left recursion:

$E \rightarrow TE'$

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

$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E)$
$F \rightarrow id$

Step 2: Grammar is already left factored.

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

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

FIRST (E') = {**+**, ε}

FIRST (T') = {*, ε}

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

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

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

Example:

E→TE'

E'→+TE'|ε

T→FT'

T'→*FT'|ε

F→(E)

F→id

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

FIRST (E') = {+, ε}

FIRST (T') = {*, ε}

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

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

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

Application: To design Top Down and Bottom up Parsers.

Code:

Output:

Result and Discussion:

Learning Outcomes:

The student should have the ability to

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

Course Outcomes:

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

Conclusion:

For Faculty Use

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

Experiment 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:

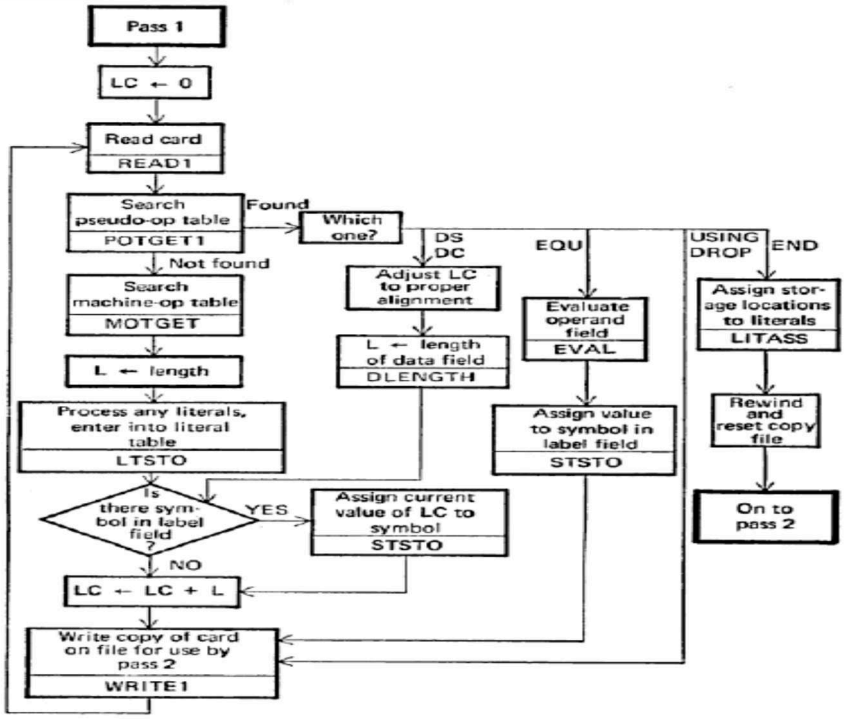
Pass 1

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

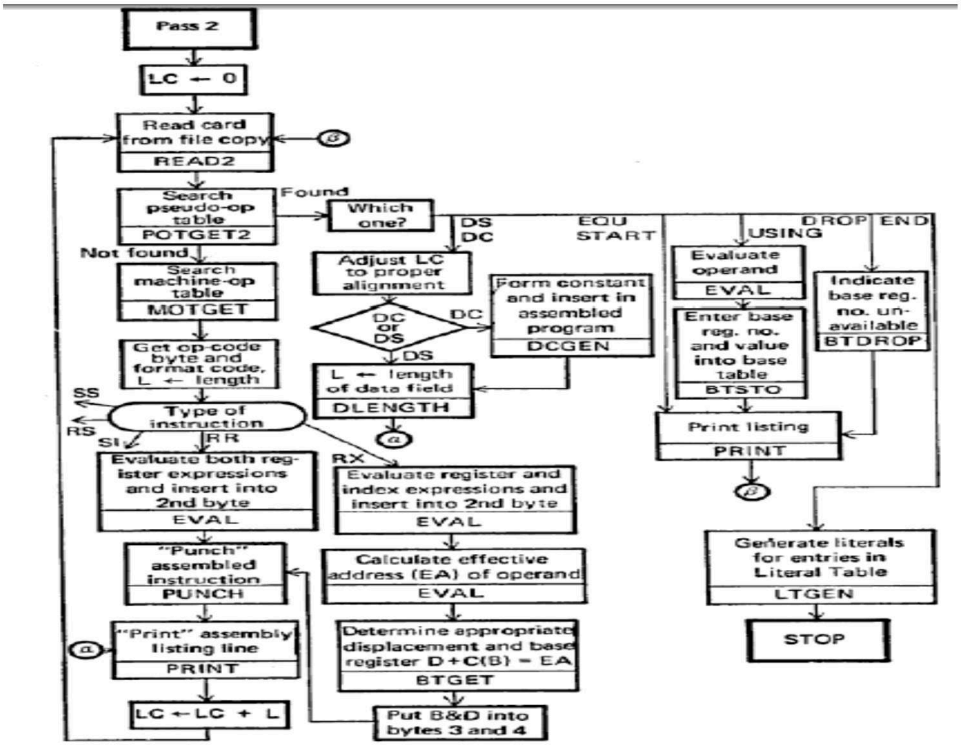
Pass 2

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 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				

Experiment 05 : Intermediate Code Generator

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

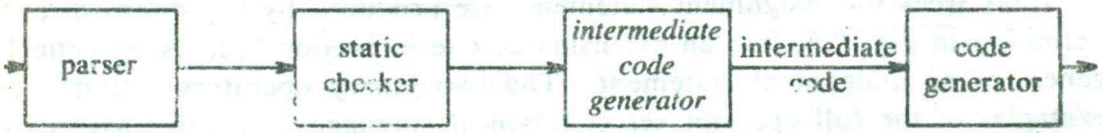
Tools: Jdk1.8,Turbo C/C++, Python, Notepad++

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.



(Intermediate Languages) Intermediate Code Representation:

a) Syntax trees or DAG

b) postfix notation

c) Threeaddress code

Types of Three-Address Statements:

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

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

Here are the common three-address statements:

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

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

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

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

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

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

par am Xl

par am X2

paramXn

callp,n

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

7. Indexed assignments of the form X : = y[i] and x[i] : = y. The first of these sets x to the value in the location i memory units beyond location y. The statement x[i] : = y sets the contents of the

location i units beyond x to the value of y. In both these instructions, x, y, and irefer to data objects.Address and pointer assignments of the form x : = &y, x : = *y, and *x : = y.

Implementations of Three-Address Statements

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

Quadruples:

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

op, arg I, arg 2, and result.

b) The op field contains an internal code for the operator.
c) The three-address statement x : = y op z is represented by placing y in arg1, Z in arg 2, and x in result.
d) Statements with unary operators like x : = -y or x : = y do not use arg 2. Operators like param use neither arg 2 nor result.
e) Conditional and unconditional jumps put the target label in result.
f) The quadruples are for the assignment a : = b * - c + b* - c.
They are obtained from the three-address code in Fig. (a).

g) The contents of fieldsarg I, arg 2, and result are normally pointers to 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 either pointers to the symbol table. Since three fields are used, this intermediate code format is known as triples.

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

(a) Quadruples

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

(b) Triples

Quadruple and triple representations of three-address statements.

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

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

(a) x[i] := y

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

(b) x := y[i]

More triple representations.

Indirect Triples:

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

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

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

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

Indirect triples representation of three-address statements

Input:

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

Output:

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

(a) Quadruples

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

(b) Triples

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

Design:

```
#include<conio.h>

#include<string.h>

int i=1,j=0,no=0,tmpch=90;

char str[100],left[15],right[15];
```

```
void findopr();

void explore();

void fleft(int);

void fright(int);

struct exp

{

int pos;

char op;

}k[15];

void main()

{

printf("\t\t\tINTERMEDIATE CODE GENERATION\n\n");

printf("Enter the Expression :");

scanf("%s",str);

printf("The intermediate code:\n");

findopr();

explore();

}

void findopr()

{

for(i=0;str[i]!='\0';i++)

if(str[i]==':')

{

k[j].pos=i;

k[j++].op=':';
```

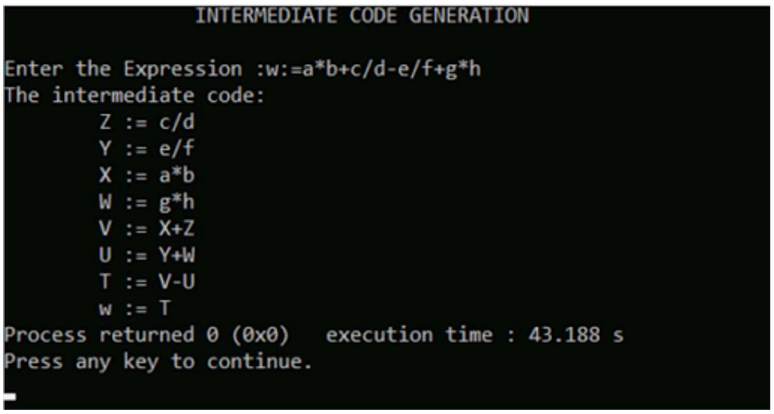
```
}
for(i=0;str[i]!='\0';i++)
if(str[i]=='/')
{
k[j].pos=i;
k[j++].op='/';
}
for(i=0;str[i]!='\0';i++)
if(str[i]=='*')
{
k[j].pos=i;
k[j++].op='*';
}
for(i=0;str[i]!='\0';i++)
if(str[i]=='+')
{
k[j].pos=i;
k[j++].op='+';
}
for(i=0;str[i]!='\0';i++)
if(str[i]=='-')
{
k[j].pos=i;
k[j++].op='-';
}
```

```
}

void explore()

{

i=1;

while(k[i].op!='\0')

{

fleft(k[i].pos);

fright(k[i].pos);

str[k[i].pos]=tmpch--;

printf("\t%c := %s%c%s\t\t",str[k[i].pos],left,k[i].op,right);

printf("\n");

i++;

}

fright(-1);

if(no==0)

{

fleft(strlen(str));

printf("\t%s := %s",right,left);

getch();

exit(0);

}

printf("\t%s := %c",right,str[k[--i].pos]);

getch();

}

void fleft(int x)
```

```
{
int w=0,flag=0;
x--;
while(x!= -1 &&str[x]!= '+' &&str[x]!='*'&&str[x]!='='&&str[x]!='\0'&&str[x]!='-'&&str[x]!='/'&&str[x]!=':')
{
if(str[x]!='$'&& flag==0)
{
left[w++]=str[x];
left[w]='\0';
str[x]='$';
flag=1;
}
x--;
}
}
void fright(int x)
{
int w=0,flag=0;
x++;
while(x!= -1 && str[x]!= '+'&&str[x]!='*'&&str[x]!='\0'&&str[x]!='='&&str[x]!=':'&&str[x]!='-'&&str[x]!='/')
{
if(str[x]!='$'&& flag==0)
{
right[w++]=str[x];
right[w]='\0';
```

```
str[x]='$';

flag=1;

}

x++;

}

}
```

Result and Discussion:



Learning Outcomes: The student should have the ability to

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

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

Conclusion:

For Faculty Use

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

Experiment 02: Lexical Analyzer

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

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

Theory:

Design of lexical analyzer

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

Regular Expressions

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

Regular Expressions (Rules)

Regular expressions over alphabet S

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

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

*	highest
concatenation	next
|	lowest

How to recognize tokens

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

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

Using 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 algorithm to construct an NFA N from R, and the other approach is using a DFA.

Finite Automata

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

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

• A finite automaton can be: *deterministic(DFA)* or *non-deterministic (NFA)*
• This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.
• Both deterministic and non-deterministic finite automaton recognizes regular sets.
• Which one?
– deterministic – faster recognizer, but it may take more space
– non-deterministic – slower, but it may take less space
– Deterministic automatons are widely used lexical analyzers.

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

Algorithm1: Regular Expression → NFA → DFA (two steps: first to NFA, then to DFA)
Algorithm2: Regular Expression → DFA (directly convert a regular expression into a DFA)

Converting Regular Expressions to NFAs

• Create transition diagram or transition table i.e. NFA for every expression
• Create a zero state as start state and with an e-transition connect all the NFAs and prepare a combined NFA.

Algorithm:for lexical analysis

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

Input:

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

Output:

```
Preprocessor Directives: #include
```

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

Application: To design a lexical analyzer.

Design:

Code:-

```
#include <iostream>
#include <cstring>
#include <stdlib.h>
#include <ctype.h>
#include <fstream>
#include <string>
using namespace std;
string arr[] = {"void", "using", "namespace", "int", "include", "iostream", "std", "main",
          "cin", "cout", "return", "float", "double", "string"};
bool isKeyword(string a)
{
  for (int i = 0; i < 14; i++)
  {
    if (arr[i] == a)
    {
      return true;
    }
  }
  return false;
}
bool isOperator(string s)
{
  if (s == "+" || s == "-" || s == "*" || s == "/" || s == "^" || s == "&&" || s == "||" ||
    s == "=" || s == "==" || s == "&" || s == "|" || s == "%" || s == "++" || s == "--" || s == "+=" ||
    s == "-=" || s == "/=" || s == "*=" || s == "%=")
  {
    return true;
  }
  return false;
}
bool isSymbol(string s)
{
  if (s == "(" || s == "{" || s == "[" || s == ")" || s == "}" || s == "]" || s == "<" ||
    s == ">" || s == "()" || s == ";" || s == "<<" || s == ">>" || s == "," || s == "#")
  {
    return true;
  }
  return false;
}
int main()
{
  std::ifstream file("prog.txt");
```

```
std::string x;
string code = "";
while (std::getline(file, x))
{
   code += x;
}
string s = "";
for (int i = 0; i < code.size(); i++)
{
   if (code[i] != ' ')
      s += code[i];
   else
   {
      if (isOperator(s))
      {
         cout << s << " is an operator" << endl;
         s = "";
      }
      else if (isKeyword(s))
      {
         cout << s << " is a keyword" << endl;
         s = "";
      }
      else if (isSymbol(s))
      {
         cout << s << " is a symbol" << endl;
         s = "";
      }
      else if (s == "\n" || s == "" || s == "")
      {
         s = "";
      }

      else if (isdigit(s[0]))
      {
         int x = 0;
         if (!isdigit(s[x++]))
         {
            continue;
         }
         else
         {
            cout << s << " is a constant" << endl;
            s = "";
         }
      }
      else
      {
         cout << s << " is an identifier" << endl;
         s = "";
      }
   }
}
```

}

Output:-

```
PS C:\Users\Anup Jaiswal> cd "c:\Users\Ar
int is a keyword
main is a keyword
( is a symbol
) is a symbol
{ is a symbol
// is an identifier
2 is a constant
variables is an identifier
int is a keyword
a, is an identifier
b is an identifier
; is a symbol
a is an identifier
= is an operator
10 is a constant
; is a symbol
return is a keyword
0 is a constant
; is a symbol
PS C:\Users\Anup Jaiswal\OneDrive\Documer
```

Result and Discussion:

Learning Outcomes: The student should have the ability to

LO1: Appreciate the role of lexical analyzer in compiler design

LO2: Define role of lexical analyzer.

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

Conclusion:

For Faculty Use

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