PDEU

PANDIT DEENDAYAL ENERGY UNIVERSITY

Formerly Pandit Deendayal Petroleum University (PDPU)

Laboratory Manual
20CP302P: System Software & Complier Design Lab

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

SCHOOL OF TECHNOLOGY

Name of Student:

Roll No:

Branch:

Sem./Year:

Academic Year:

PANDIT DEENDAYAL ENERGY UNIVERSITY

Raisan, Gandhinagar – 380 007, Gujarat, India

Computer Science and Engineering Department

Certificate

This is to certify that

Mr./Ms. _____ Roll no._____

Exam No. _____ of 5th Semester Degree course in Computer Science and Engineering has satisfactorily completed his/her term work in System Software & Compiler Design Lab (20CP302P) subject during the semester from_____ to _____ at School of Technology, PDEU.

Date of Submission:

Signature:

Faculty In-charge Head of Department

Index

Name:

Roll No:

Exam No:

Sr. No.	Experiment Title	Pages		Date of Completion	Marks (out of 10)	Sign.
		From	To			
1	Write C/C++ program to identify keywords, identifiers (using DFA) and others from the given input file.					
2	a. Write a LEX program to count the number of tokens and display each token with its length in the given statements. b. Write a LEX program to identify keywords, identifiers, numbers and other characters and generate tokens for each.					
3	a. Write a LEX program to eliminate comment lines (single line and multiline) in a high-level program and copy the comments in comments.txt file and copy the resulting program into a separate file input.c. b. Write a LEX program to count the number of characters, words and lines in the given input. c. Write a LEX program to identify HTML tags/SQL tags.					
4	WAP to implement Recursive Decent Parser (RDP) for given					

	grammar using C/C++/Java.					
5	Write a program to implement predictive parser for a given LL (1) grammar using C/C++/Java.					
6	WAP to construct operator precedence parsing table for the given grammar and check the validity of the string using C/C++/Java.					
7	a. Write a YACC program for desktop calculator with ambiguous grammar (evaluate arithmetic expression involving operators: +, -, *, / and ↑). b. Write a YACC program for desktop calculator with ambiguous grammar and additional information. c. Design, develop and implement a YACC program to demonstrate Shift Reduce Parsing technique for the grammar rules: E→ E + T | T T→ T * F | F F → P ↑ F | P P → (E) | id And parse the sentence: id + id * id.					
8	Write a program to implement pass-I and data structures of an assembler.					
9	Implement menu driven program to execute any 2 code optimization techniques on given code.					
10	Implement a toy compiler for any programming language.					

COURSE OUTCOMES

On completion of the course, student will be able to
CO1- Identify token in the given input string using any programming language.
CO2- Apply different parsing algorithms to check whether the given string is valid or not.
CO3- Calculate the value of a mathematical expression using parsing algorithms.
CO4- Analyze pass1 and pass2 assembler algorithms.
CO5- Apply optimization techniques related to target code generation.
CO6- Design demo compiler.

Prerequisite knowledge required:	**New Tools to learn:**
C/C++/Java programming language	-Lex tool for lexical analysis -YACC tool for parsing

Experiment No: 1 Write C/C++ program to identify keywords, identifiers (using DFA) and others from the given input file.

Algorithm:

- **Reads an input file** containing C/C++ source code.

- **Uses a Deterministic Finite Automaton (DFA)** to recognize:

  - **Keywords** (predefined in C/C++).
  - **Identifiers** (valid variable/function names).
  - **Others** (operators, symbols, literals, etc.).

A valid identifier in C/C++ must:

- Begin with a letter (A-Z or a-z) or underscore _.
- Followed by any combination of letters, digits (0-9), or underscores.

A keyword is a predefined identifier that should be matched separately.

High-Level Algorithm:

Algorithm: Identify Tokens from C/C++ Code using DFA

1. Define a list/array of C/C++ keywords.

2. Open the input file containing C/C++ code.

3. Read the file word by word (tokenize it based on space and special characters).

4. For each token:

   a. Apply logic of pattern matching to check if it's in the keyword list → classify as KEYWORD.

   b. Else, use transition table of DFA/loop & control structure to check if it's a valid IDENTIFIER as per rules.

   c. Else → classify as OTHER (numbers, symbols, operators, etc.).

5. Print or store the classification in file.

Example Input:

```
int main() {
    int a = 10;
    float b = 20.5;
    if (a < b) {
        return 1;
    }
}
```

Output:

```
int --> KEYWORD
main --> IDENTIFIER
( --> OTHER
) --> OTHER
{ --> OTHER
int --> KEYWORD
a --> IDENTIFIER
= --> OTHER
10 --> OTHER
; --> OTHER
float --> KEYWORD
b --> IDENTIFIER
... etc.
```

Experiment No: 2 a. Write a LEX program to count the number of tokens and display each token with its length in the given statements.

Step-1: Introduction of Lex tool.

Reference book: John Levine, Doug Brown, Tony Mason, "Lex & yacc", 2nd Edition, O'Reilly Media

Logic:

```
%{
#include <stdio.h>
#include <string.h>
int token_count = 0;
%}

%%
[a-zA-Z_][a-zA-Z0-9_]* {
   printf("Token: %s, Length: %lu\n", yytext, yyleng);
   token_count++;
}

[0-9]+ {
   printf("Token: %s, Length: %lu\n", yytext, yyleng);
   token_count++;
}

"." {
   printf("Token: %s, Length: %lu\n", yytext, yyleng);
```

```
    token_count++;
}

[\+\-\*/=] {
    printf("Token: %s, Length: %lu\n", yytext, yyleng);
    token_count++;
}

[;,] {
    printf("Token: %s, Length: %lu\n", yytext, yyleng);
    token_count++;
}

[ \t\n]+ ;
. {
    printf("Unknown token: %s\n", yytext);
}
%%

int main() {
    printf("Enter the input statements (Ctrl+Z to end):\n");
    yylex();
    printf("\nTotal number of tokens: %d\n", token_count);
    return 0;
}

int yywrap() {
    return 1;
}
```

Input and output:

```
c:\Aadish College\SEM 5\Compiler\CD\session2>a
Enter the input statements (Ctrl+Z to end):
Aadish
Token: Aadish, Length: 6
123
Token: 123, Length: 3
@
Unknown token: @
;
Token: ;, Length: 1
^Z

Total number of tokens: 3
```

Experiment No: 2 b. Write a LEX program to identify keywords, identifiers, numbers and other characters and generate tokens for each.

Logic:

```
%option noyywrap
%{
   #include<stdio.h>
   int kw=0;
   int id=0;
   int num=0;
   int sl=0;
%}

%%
int|for|while|if|else|return|float    {printf("%s is Keyword\n",yytext); kw++;}
[a-zA-z][a-zA-Z0-9]*       {printf("%s is Identifier\n",yytext); id++;}
-?(([0-9+)|([0-9]*\.[0-9]+)([eE][+-]?[0-9]+)?)  {printf("%s is number\n",yytext); num++;}
\"[a-zA-Z0-9 @#$%^&*!]*\"     {printf("%s is string literal\n",yytext); sl++;}
%%
```

```
int main(){
   yylex();
   printf("%d : keywords\n",kw);
   printf("%d : identifier\n",id);
   printf("%d : numbers\n",num);
   printf("%d : string literals\n",sl);
   return 0;
}
```

Input and output:

```
c:\Aadish College\SEM 5\Compiler\CD\session2>a
int
int is Keyword

for
for is Keyword

123.45
123.45 is number

"Hello World"
"Hello World" is string literal

2 ^C
c:\Aadish College\SEM 5\Compiler\CD\session2>
```

Experiment No: 3 a. Write a LEX program to eliminate comment lines (single line and multiline) in a high-level program and copy the comments in comments.txt file and copy the resulting program into a separate file input.c.

Logic:

```
%option noyywrap
%{
   #include<stdio.h>
   FILE *resultFile;
%}

%
\/\/.*    { fprintf(yyout, "%s\n", yytext); } /* Write single-line comments to comments.txt */

\/\*[^\*]*\*+([^\/][^\*]*\*+)*\/      { fprintf(yyout, "%s\n", yytext); } /* Write multi-line
comments to comments.txt */

.|\n { fprintf(resultFile, "%s", yytext); } /* Write the rest of the program to input.c */
%%

int main() {
 yyin = fopen("highlevel.c", "r");
 if (!yyin) {
    perror("Error opening input file highlevel.c");
    return 1;
 }
```

```
    yyout = fopen("comments.txt", "w");
    if (!yyout) {
        perror("Error opening output file comments.txt");
        fclose(yyin);
        return 1;
    }

    resultFile = fopen("input.c", "w");
    if (!resultFile) {
        perror("Error opening output file input.c");
        fclose(yyin);
        fclose(yyout);
        return 1;
    }

    yylex();

    fclose(yyin);
    fclose(yyout);
    fclose(resultFile);
    return 0;
}
```

Input and output:

prog11.l highlevel.c

session4 > highlevel.c > ...

```
1   //Aadish
2   /*
3   Aadish Sheth
4   */
5   int a=3;
6   int b=5;
7   int sum;
```

prog11.l comments.txt

session4 > comments.txt

```
1   //Aadish
2   /*
3   Aadish Sheth
4   */
```

prog11.l input.c

session4 > input.c > sum

```
1
2
3   int a=3;
4   int b=5;
5   int sum;
```

Experiment No: 3 b. Write a LEX program to count the number of characters, words and lines in the given input.

Logic:

```
%option noyywrap
%{
   #include<stdio.h>
   int charCount=0, wordCount=0, lineCount=0;
%}

word [^ \t\n]+
eol \n

%%
{word} {wordCount++; charCount += yyleng; }
{eol}  {charCount++; lineCount++;}
. charCount++;
%%
int main(){
   yyin=fopen("input.c","r");
   yylex();
   printf("Line Count: %d, Word Count: %d, Char Count:
%d\n",lineCount+1,wordCount,charCount);
   return 0;
}
```

Input and output:

```
c:\Aadish College\SEM 5\Compiler\CD\session4>a
Line Count: 5, Word Count: 6, Char Count: 28
```

Experiment No: 3 c. Write a LEX program to identify HTML tags/SQL tags.

Logic:

```
%%

^<[^>]+>    { printf("HTML Tag Found: %s\n", yytext); }

SELECT      { printf("SQL Keyword Found: SELECT\n"); }
INSERT      { printf("SQL Keyword Found: INSERT\n"); }
UPDATE      { printf("SQL Keyword Found: UPDATE\n"); }
DELETE      { printf("SQL Keyword Found: DELETE\n"); }
CREATE      { printf("SQL Keyword Found: CREATE\n"); }
DROP        { printf("SQL Keyword Found: DROP\n"); }
FROM        { printf("SQL Keyword Found: FROM\n"); }
WHERE       { printf("SQL Keyword Found: WHERE\n"); }

[ \t\n]+    ;
. { printf("Unrecognized Token: %s\n", yytext); }
%%
```

```
int main() {
    yyin = fopen("input.txt", "r");
    if (!yyin) {
        perror("Error opening input file input.txt");
        return 1;
    }

    yylex();
  fclose(yyin);
  return 0;
}
```

Input and output:

```
c:\Aadish College\SEM 5\Compiler\CD\session4>a
HTML Tag Found: <h1>
HTML Tag Found: </h1>
SQL Keyword Found: INSERT
SQL Keyword Found: SELECT
```

Experiment No: 4 WAP to implement Recursive Decent Parser (RDP) for given grammar using C/C++/Java.

Grammar:

E -> TE'
E'-> +TE' | -TE' | epsilon
T -> FT'
T' -> *FT' | /FT' | epsilon
F -> (E) | i

Steps:

1. Define the Grammar:

Begin with a context-free grammar that describes the language you want to parse.

2. Create Parsing Functions:

For each non-terminal symbol in the grammar, create a corresponding parsing function.

3. Tokenize the Input:

Use a lexical analyzer (lexer) to break down the source code into a stream of tokens.

4. Implement Parsing Functions:

- **Match Terminals:**

  Within each parsing function, check if the current token matches the expected terminal symbol. If it does, consume the token (advance the input pointer) and continue parsing.

- **Call Other Functions:**

  If the current token is a non-terminal, call the corresponding parsing function recursively.

- **Handle Alternatives:**

  If a non-terminal has multiple production rules, implement logic to try each alternative in order, potentially using lookahead (e.g., checking the next token) to make the correct choice.

- **Backtracking (If Necessary):**

  If a parsing path fails (e.g., a mismatch between expected and actual terminals), the parser may need to backtrack and try an alternative production rule.

- **Error Handling:**
  Include error detection and reporting mechanisms to handle syntax errors in the input.

5. Start with the Start Symbol:

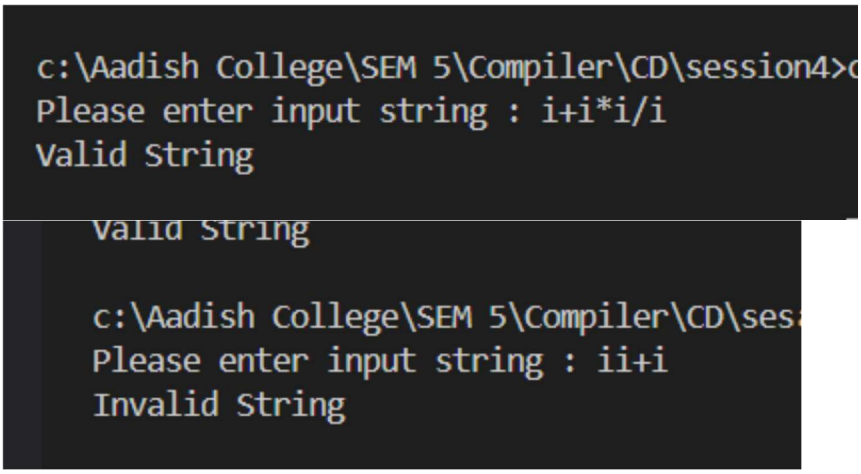
The parsing process begins by calling the parsing function that corresponds to the start symbol of the grammar.

6. Optional Semantic Actions:

While parsing, you can include semantic actions (code to be executed) within the parsing functions to build a parse tree, perform type checking, or generate code.

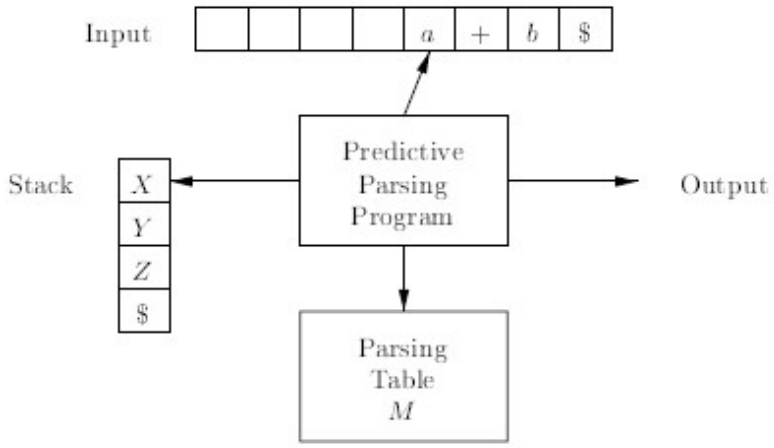
For given examples grammar:
Each rule in the grammar would have a corresponding function in the parser (e.g., E(), E_prime(), T(), T_prime(), F()). The E() function would be called initially to start the parsing process.

Input and output:



Experiment No: 5 Write a program to implement predictive parser for a given LL (1) grammar using C/C++/Java.

Method:



Steps:

1. Preprocessing the Grammar:

- Eliminate ambiguity
- Eliminate Left Recursion:
- Left Factoring:

2. Computing FIRST and FOLLOW Sets:

- **FIRST Sets:**

1. If X is a terminal, then $\text{FIRST}(X) = \{X\}$.
2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $\text{FIRST}(X)$ if for some i, a is in $\text{FIRST}(Y_i)$, and ϵ is in all of $\text{FIRST}(Y_1), \ldots, \text{FIRST}(Y_{i-1})$; that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in $\text{FIRST}(Y_j)$ for all $j = 1, 2, \ldots, k$, then add ϵ to $\text{FIRST}(X)$. For example, everything in $\text{FIRST}(Y_1)$ is surely in $\text{FIRST}(X)$. If Y_1 does not derive ϵ, then we add nothing more to $\text{FIRST}(X)$, but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add $\text{FIRST}(Y_2)$, and so on.
3. If $X \to \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.

- **FOLLOW Sets:**

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.

2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).

3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

3. Constructing the Parsing Table:

The parsing table is a two-dimensional table with non-terminals as rows and terminals (plus \$) as columns.

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(α), add $A \to \alpha$ to $M[A, a]$.

2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). □

4. Parsing the Input String:

The parser uses a stack and an input buffer.

METHOD: Initially, the parser is in a configuration with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above \$. The program in Fig. 4.20 uses the predictive parsing table M to produce a predictive parse for the input. □

```
let a be the first symbol of w;
let X be the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X = a ) pop the stack and let a be the next symbol of w;
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X → Y1Y2 ··· Yk ) {
                output the production X → Y1Y2 ··· Yk;
                pop the stack;
                push Yk, Yk−1, ... , Y1 onto the stack, with Y1 on top;
        }
        let X be the top stack symbol;
}
```

Input and output:

```
c:\Aadish College\SEM 5\Compiler\CD\session5>cd "c:\
sion6\"FirstFollow
Please enter product rules :
Enter q to stop
E
TR
R
+TR
R
^
T
FY
Y
*FY
Y
^
F
(E)
F
i
q
Please enter your start symbol : E
```

```
The first set of all items is displayed below :
E : ( i
F : ( i
R : + ^
T : ( i
Y : * ^

The Follwo set of all items is displayed below :
E -> $ )
F -> $ ) * +
R -> $ )
T -> $ ) +
Y -> $ ) +
```

Experiment No: 6 WAP to construct operator precedence parsing table for the given grammar and check the validity of the string using C/C++/Java.

The input string is w$, the initial stack is $ and a table holds precedence relations between certain terminals

Algorithm:

```
set p to point to the first symbol of w$ ;
repeat forever
   if ( $ is on top of the stack and p points to $ ) then return
   else {
      let a be the topmost terminal symbol on the stack and let b be the symbol
       pointed to by p;
      if ( a <· b or a =· b ) then {          /* SHIFT */
         push b onto the stack;
         advance p to the next input symbol;
      }
      else if ( a ·> b ) then                 /* REDUCE */
         repeat pop stack
         until ( the top of stack terminal is related by <· to the terminal most
                                recently popped );
      else error();
   }
```

Input and output:

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\hat{}E \mid (E) \mid -E \mid id$

The partial operator-precedence table for this grammar

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

- Then the input string id+id*id with the precedence relations inserted will be:

  $ <· id ·> + <· id ·> * <· id ·> $

$	<·

stack	***input***	***action***	
$	id+id*id$	$ <· id	shift
$id	+id*id$	id ·> +	reduceE → id
$	+id*id$	shift	
$+	id*id$	shift	
$+id	*id$	id ·> *	reduce E → id
$+	*id$	shift	
$+*	id$	shift	
$+*id	$	id ·> $	reduce E → id
$+*	$	* ·> $	reduce E → E*E
$+	$	+ ·> $	reduce E → E+E
$	$	accept	

Experiment No: 7 a. Write a YACC program for desktop calculator with ambiguous grammar (evaluate arithmetic expression involving operators: +, -, *, / and ↑).

Logic:

first.l

```
%{
#include "first.tab.h"
#include <stdlib.h>
extern int yylval;
%}

%%
[0-9]+        { yylval = atoi(yytext); return NUMBER; }
[ \t]         ; // Ignore whitespace characters
\n            { return 0; } // Return 0 on newline
.             { return yytext[0]; } // Return single character for anything else
%%

int yywrap() {
    return 1;  // Lex will return 1 when done processing input
}
```

first.y

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
extern int yylex(void);

void yyerror(char *s);
%}

%token NUMBER

%%
statement: expression { printf("= %d\n", $1); }
expression:
   expression '+' expression { $$ = $1 + $3; }

  | expression '-' expression { $$ = $1 - $3; }

  | expression '*' expression { $$ = $1 * $3; }

  | expression '^' expression { $$ = (int)pow($1, $3); }
```

```
| expression '/' expression
  { if($3 == 0)
     yyerror("divide by zero");
    else
     $$ = $1 / $3; }  // Division with zero check

| '-' expression { $$ = -$2; }  // Unary minus

| '(' expression ')' { $$ = $2; }  // Parenthesized expression

| NUMBER { $$ = $1; }  // Number token

;

%%

void yyerror(char *s)
{
   fprintf(stderr, "Error: %s\n", s);  // Error reporting
}

int main(void)
{
   return yyparse();  // Parse input
}
```

Input and output:

```
C:\Aadish College\SEM 5\Compiler\CD\session7\without prec>a
5+5
= 10
```

Experiment No: 7 b. Write a YACC program for desktop calculator with ambiguous grammar and additional information.
Include following code before rules section in 7 a:

```
%token NUMBER

%left '+' '-'

%left '*' '/'

%right '^'
 %nonassoc 'UMINUS'
```

Input and output:

```
C:\Aadish College\SEM 5\Compiler\CD\session7\extra_in_this>a
7+15
Result: 22

C:\Aadish College\SEM 5\Compiler\CD\session7\extra_in_this>
```

Experiment No: 7 c. Design, develop and implement a YACC program to demonstrate Shift Reduce Parsing technique for the grammar rules:

E→ E + T | T
T→ T * F | F
F → P ↑ F | P
P → (E) | id
And parse the sentence: id + id * id.

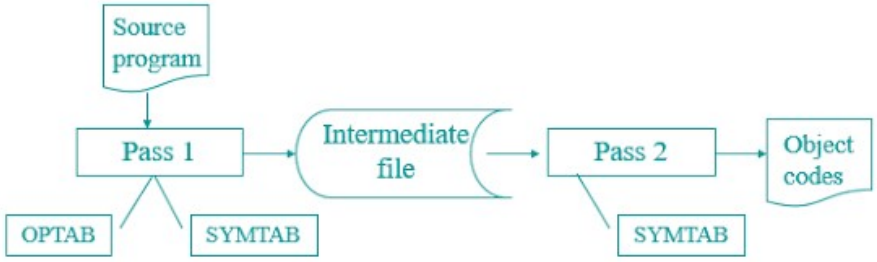
Replace the rules section as follows in 7 a:

```
%token NUMBER
%%
statement: e { printf("= %d\n", $1); }
        ;
e: e '+' t { $$ = $1 + $3; }
        | e '-' t { $$ = $1 - $3; }
            | t ;

t:t '*' f { $$ = $1 * $3; }
            | t '/' f { $$ = $1 / $3; }
        | f ;
f:p '^' f { $$ = pow($1,$3); }
            | p;
p:'-' s { $$ = -$2; }

            | s;

s:'('e')' { $$ = $2; }
            | NUMBER    { $$ = $1; }  ;

%%
```

Input and output:

```
C:\Aadish College\SEM 5\Compiler\CD\session7\unam>a
Enter expressions to evaluate :
5+3*4
= 17
```

Experiment No: 8 Write a program to implement pass-I and data structures of an assembler.



Tasks performed by the passes of a two pass assembler are as follows:

Pass I

- Separate the symbol, mnemonic opcode, operand fields
- Build the symbol table
- Perform LC processing
- Assembler Directive Handling
- Construct intermediate representation

Pass II

- Synthesis the target program

It uses the following data structures:

1. OPTAB –A table of mnemonic opcodesand related information
2. SYMTAB –Symbol Table
3. LITTAB –A table of literals used in the program

Algorithm-Assembler First Pass

1. *loc_cntr* := 0; (default value)
   pooltab_ptr := 1; POOLTAB[1] := 1;
   littab_ptr := 1;
2. While next statement is not an END statement
   (a) If label is present then
   this_label := symbol in label field;
   Enter (*this_label*, *loc_cntr*) in SYMTAB.

(b) If an LTORG statement then

(i) Process literals LITTAB [POOLTAB [*pooltab_ptr*]] ... LITTAB [*littab_ptr*−1] to allocate memory and put the address in the *address* field. Update *loc_cntr* accordingly.

(ii) *pooltab_ptr* := *pooltab_ptr* + 1;

(iii) POOLTAB [*pooltab_ptr*] := *littab_ptr*;

(c) If a START or ORIGIN statement then
loc_cntr := value specified in operand field;

(d) If an EQU statement then

(i) *this_addr* := value of <*address spec*>;

(ii) Correct the symtab entry for *this_label* to (*this_label, this_addr*).

(e) If a declaration statement then

(i) *code* := code of the declaration statement;

(ii) *size* := size of memory area required by DC/DS.

(iii) *loc_cntr* := *loc_cntr* + *size*;

(iv) Generate IC '(DL, *code*) ···'.

(f) If an imperative statement then

(i) *code* := machine opcode from OPTAB;

(ii) *loc_cntr* := *loc_cntr* + instruction length from OPTAB;

(iii) If operand is a literal then
this_literal := literal in operand field;
LITTAB [*littab_ptr*] := *this_literal*;
littab_ptr := *littab_ptr* + 1;
else (i.e. operand is a symbol)
this_entry := SYMTAB entry number of operand;
Generate IC '(IS, *code*)(S, *this_entry*)';

3. (Processing of END statement)

(a) Perform step 2(b).

(b) Generate IC '(AD,02)'.

(c) Go to Pass II.

Input:

```
 1          START   200
 2          MOVER   AREG, ='5'      200)  +04 1 211
 3          MOVEM   AREG, A         201)  +05 1 217
 4   LOOP   MOVER   AREG, A         202)  +04 1 217
 5          MOVER   CREG, B         203)  +05 3 218
 6          ADD     CREG, ='1'      204)  +01 3 212
 7          ...

12          BC      ANY, NEXT       210)  +07 6 214
13          LTORG
                    ='5'            211)  +00 0 005
                    ='1'            212)  +00 0 001
14          ...
15   NEXT   SUB     AREG, ='1'      214)  +02 1 219
16          BC      LT, BACK        215)  +07 1 202
17   LAST   STOP                    216)  +00 0 000
18          ORIGIN  LOOP+2
19          MULT    CREG, B         204)  +03 3 218
20          ORIGIN  LAST+1
21   A      DS      1               217)
22   BACK   EQU     LOOP
23   B      DS      1               218)
24          END
25                  ='1'            219)  +00 0 001
```

Output Code:

```
          START   200             (AD,01)   (C,200)
          READ    A               (IS,09)   (S,01)
LOOP      MOVER   AREG, A         (IS,04)   (1)(S,01)
            :                       :
          SUB     AREG, ='1'      (IS,02)   (1)(L,01)
          BC      GT, LOOP        (IS,07)   (4)(S,02)
          STOP                    (IS,00)
A         DS      1               (DL, 02)  (C,1)
          LTORG                   (DL,05)
          ...                     ...
```

Output data Structures:

mnemonic opcode	*class*	*mnemonic info*
MOVER	IS	(04,1)
DS	DL	R#7
START	AD	R#11
	:	

OPTAB

symbol	*address*	*length*
LOOP	202	1
NEXT	214	1
LAST	216	1
A	217	1
BACK	202	1
B	218	1

SYMTAB

	literal	*address*
1	='5'	
2	='1'	
3	='1'	

LITTAB

literal no
#1
#3
–

POOLTAB

Experiment No: 9 Implement menu driven program to execute any 2 code optimization techniques on given code.

Code optimization Techniques:

1. Constant Propagation:
   - Identify variables with known constant values at specific points in the program
   - Substitute those constants wherever the variables are used

2. Variable Propagation:
   - Identify Variables and Their Values
   - Replace Variables with Values
   - Remove Dead Code
   - Iterate and Optimize

3. Constant Folding:
   - Identifying Constant Expressions: The compiler detects expressions with constant operands.
   - Evaluating at Compile Time: These expressions are evaluated during compilation, not when the program runs.
   - Replacing with Result: The evaluated constant replaces the original expression in the code.

4. Deadcode elimination:
   - Identification of Dead Code
   - Removal of Dead Code
   - Iterative Refinement

5. Common Sub Expression Elimination:
   - Identify Subexpressions
   - Analyze for Redundancy
   - Store Results
   - Replace with Temporary
   - Handle Control Flow
   - Optimize Further

Input and output:

```
Menu: Choose an Optimization Technique:
1. Constant Folding
2. Copy Propagation
3. Apply Both Techniques
4. Exit
Enter your choice: 1
Enter the number of expressions:
1

Enter expression (in format 'var = expression'):
a = 22 / 7
Constant Folding: a = 3.14286
```

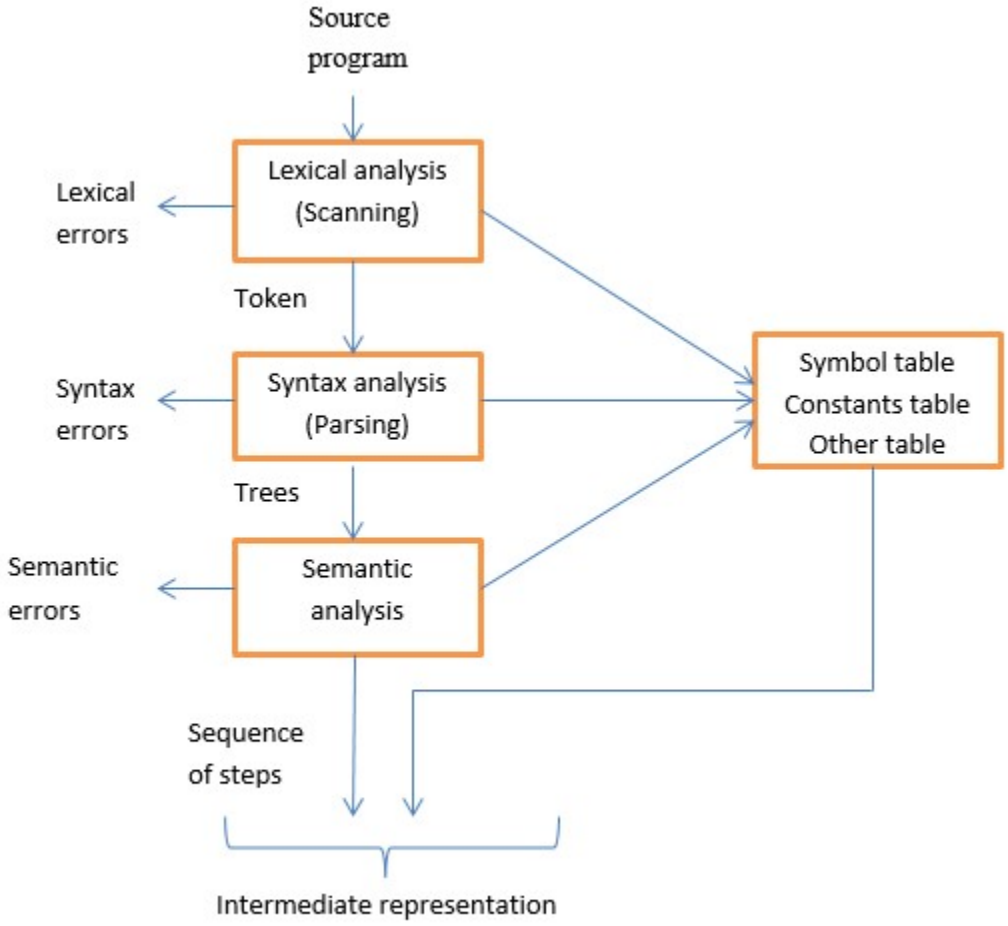
```
Enter your choice: 2
Enter the number of expressions:
2

Enter expression (in format 'var = expression'):
a = 5
After Copy Propagation: a = 5

Enter expression (in format 'var = expression'):
b = a + 5
After Copy Propagation: b = 5.000000 + 5
```

Experiment No: 10 Project based learning

Implement a toy compiler for any programming language.

Logic:

The front end of the Toy compiler performs lexical, syntax, and semantic analysis on source code. It constructs an intermediate code representation and stores analysis information in tables. The back end performs memory allocation, assigning addresses to identifiers, and code generation to produce assembly code.



Front end of toy compiler