**COMPILER DESIGN**

**(01CE0714)**

**2024-2025**

**STUDENT LAB MANUAL**

**INDEX**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No.** | **Title** | **Date** | **Grade** | **Sign** |
| 1 | Write a C Program to remove Left Recursion from grammar |  |  |  |
| 2 | Write a C Program to remove Left Factoring from grammar |  |  |  |
| 3 | Write a C program to implement finite automata and string validation. |  |  |  |
| 4 | Prepare report for Lex and install Lex on Linux/Windows |  |  |  |
| 5 | (a) WALEx Program to count words, characters, lines, Vowels and consonants from given input  (b) WALEx Program to generate string which is ending with zeros. |  |  |  |
| 6 | (a) WALex Program to generate Histogram of words  (b) WALex Program to remove single or multi line comments from C program. |  |  |  |
| 7 | WALex Program to check weather given statement is compound or simple. |  |  |  |
| 8 | WALex Program to extract HTML tags from .html file. |  |  |  |
| 9 | Write a C Program to compute FIRST Set of the given grammar |  |  |  |
| 10 | Write a C Program to compute FOLLOW Set of the given grammar |  |  |  |
| 11 | Write a C Program to implement Operator precedence parser |  |  |  |
| 12 | Write a C Program for constructing LL (1) parsing |  |  |  |
| 13 | Write a C program to implement SLR parsing |  |  |  |
| 14 | Prepare a report on YACC and generate Calculator Program using  YACC. |  |  |  |

**Practical 1**

**Title:** **Write a C Program to remove Left Recursion from the grammar.**

**Hint :** The program reads a grammar production, checks for left recursion, extracts `α` and `β`, and then constructs and prints a new grammar without left recursion using the transformations \( A \rightarrow βA' \) and \( A' \rightarrow αA' | \epsilon \).

**Program :**

#include<stdio.h>

#define SIZE 10

void main () {

char non\_terminal;

char beta,alpha[6];

char production[SIZE];

int index=3;

int i=0,j=0; /\* starting of the string following "->" \*/

printf("Enter the grammar:\n");

scanf("%s",&production);

non\_terminal=production[0];

if(non\_terminal==production[index]) {

for(i=index+1;production[i]!='|';i++)

{

alpha[j]=production[i];

j++;

}

alpha[j]='\0';

printf("Grammar is left recursive.\n");

while(production[index]!=0 && production[index]!='|')

index++;

if(production[index]!=0) {

beta=production[index+1];

printf("Grammar without left recursion:\n");

printf("%c->%c%c\'",non\_terminal,beta,non\_terminal);

printf("\n%c\'->%s%c\'|E\n",non\_terminal,alpha,non\_terminal);

}

else

printf("Grammar can't be reduced\n");

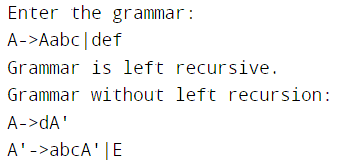
}

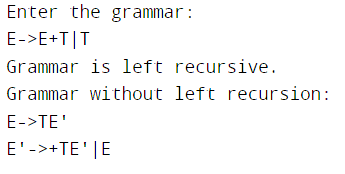
else

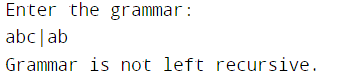
printf("Grammar is not left recursive.\n");

}

**Output:**

****

****

****

**Practical 2**

**Title:** **Write a C Program to remove Left Factoring from the grammar.**

**Hint :** This program reads a production of the form A->part1|part2, finds the common prefix in part1 and part2, and then restructures the grammar to eliminate left factoring.

**Program:**

#include<stdio.h>

#include<string.h>

int main()

{

char gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20];

int i,j=0,k=0,l=0,pos;

printf("Enter Production : A->");

gets(gram);

for(i=0;gram[i]!='|';i++,j++)

part1[j]=gram[i];

part1[j]='\0';

for(j=++i,i=0;gram[j]!='\0';j++,i++)

part2[i]=gram[j];

part2[i]='\0';

for(i=0;i<strlen(part1)||i<strlen(part2);i++){

if(part1[i]==part2[i]){

modifiedGram[k]=part1[i];

k++;

pos=i+1;

}

}

for(i=pos,j=0;part1[i]!='\0';i++,j++){

newGram[j]=part1[i];

}

newGram[j++]='|';

for(i=pos;part2[i]!='\0';i++,j++){

newGram[j]=part2[i];

}

modifiedGram[k]='X';

modifiedGram[++k]='\0';

newGram[j]='\0';

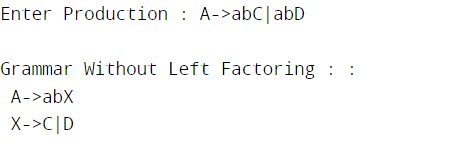
printf("\nGrammar Without Left Factoring : : \n");

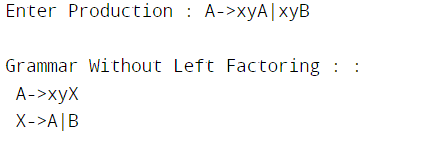
printf(" A->%s",modifiedGram);

printf("\n X->%s\n",newGram);

}

**Output:**

****

****

**//page no 6**

**Practical 5**

**Title: (a) WALEx Program to count words, characters, lines, Vowels and consonants from given input**

**Hint:** This program will take a string input and then count the number of words, characters, lines, vowels, and consonants in that input.

**Program:**

%{

#include <stdio.h>

#include <ctype.h>

int characters = 0, words = 1, lines = 1, vowels = 0, consonants = 0;

int isVowel(char ch) {

ch = tolower(ch);

return (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u');

}

%}

%%

[a-zA-Z] {

characters++; // Count characters

if (isVowel(yytext[0])) {

vowels++; // Count vowels

} else {

consonants++; // Count consonants

}

}

[ \t] {

characters++; // Count characters

}

\n {

lines++; // Count lines

characters++; // Count newline as a character

}

[.] {

characters++; // Period marks the end of input

return 0; // End lexing when a period is found

}

%%

int main() {

printf("Enter text (end input with a period '.'): \n");

yylex(); // Start scanning and tokenizing input

printf("Jay Dalsaniya \n");

printf("92100103336 \n");

printf("\nStatistics:\n");

printf("Characters: %d\n", characters);

printf("Words: %d\n", words);

printf("Lines: %d\n", lines);

printf("Vowels: %d\n", vowels);

printf("Consonants: %d\n", consonants);

return 0;

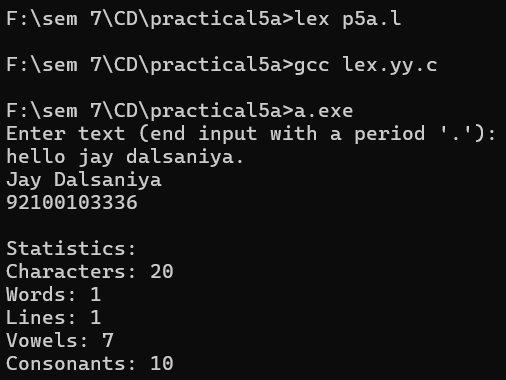
}

int yywrap() {

return 1;

}

**Output:**

****

**(b) WALEx Program to generate string which is ending with zeros.**

**Hint:** This program will take a string input and append a specific number of zeros to it, based on a given condition.

**Program:**

%{

#include <stdio.h>

#include <string.h>

int num\_zeros = 0; // Variable to store the number of zeros to append

%}

%%

[a-zA-Z0-9]+ {

// This pattern matches any alphanumeric string

// Get the length of the string

int len = yyleng;

// Determine the number of zeros to append

num\_zeros = (len % 3); // Example condition: append zeros based on the length modulo 3

printf("Jay Dalsaniya \n");

printf("92100103336 \n");

// Print the original string

printf("Original String: %s\n", yytext);

// Print the string followed by the zeros

printf("Modified String: %s", yytext);

// Variable declaration outside of the loop

int i;

for (i = 0; i < num\_zeros; i++) {

printf("0");

}

printf("\n");

}

.|\n {

// Any other character (including new lines) is ignored

}

%%

int main(int argc, char \*\*argv) {

yylex();

return 0;

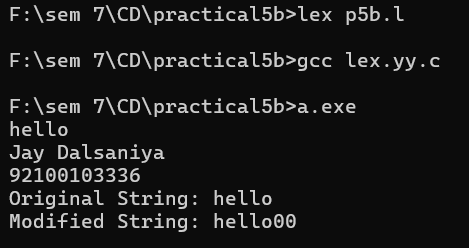
}

int yywrap() {

return 1;

}

**Output:**

****

**Practical 6**

**Title: (a) WALex Program to generate Histogram of words**

**Hint:** This program will take a string input and generate a histogram based on the length of each word.

**Program:**

%{

#include <stdio.h>

#include <string.h>

char words[1000][50];

int counts[1000], n = 0, i;

%}

%%

[a-zA-Z]+ {

for (i = 0; i < n && strcmp(words[i], yytext); i++);

if (i < n)

counts[i]++;

else {

strcpy(words[n], yytext);

counts[n++] = 1;

}

}

.|\n ; // Ignore other characters

%%

int main() {

printf("Jay Dalsaniya \n");

printf("92100103336 \n");

printf("Enter the Sentence:\n");

yylex();

for (i = 0; i < n; i++)

printf("%s: %d\n", words[i], counts[i]);

return 0;

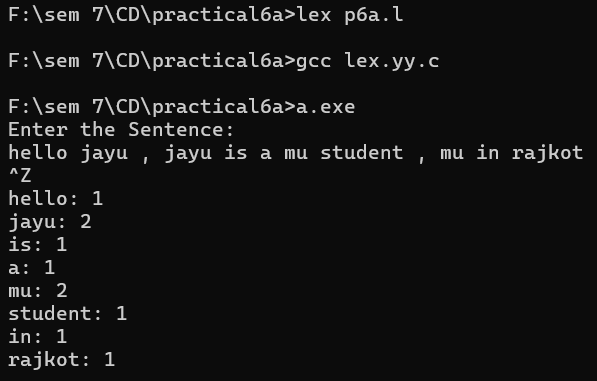
}

int yywrap() {

return 1;

}

**Output:**

****

**(b**) WALex Program to remove single or multi line comments from C program

**Hint:** To remove comments from a C program using Lex

Single-line comments (// ...): Use //[^'\n']\* to ignore everything until the end of the line.

Multi-line comments (/\* ... \*/): Use /\* to start and \*/ to end, employing a custom function to handle the removal of these comments.

**Program:**

%{

#include <stdio.h>

int sl = 0; // Counter for single-line comments

int ml = 0; // Counter for multi-line comments

%}

%%

"//"[^\n]\* { sl++; } // Match single-line comments and increment counter

"/\\*"([^\*]|\\*+[^/\*])\*\\*+"/" { ml++; } // Match multi-line comments and increment counter

%%

int yywrap() {

return 1;

}

int main() {

yyin = fopen("f1.c", "r");

yyout = fopen("f2.c", "w");

if (!yyin) {

perror("Failed to open input file");

return 1;

}

if (!yyout) {

perror("Failed to open output file");

fclose(yyin);

return 1;

}

yylex();

fclose(yyin);

fclose(yyout);

printf("Jay Dalsaniya \n");

printf("92100103336 \n");

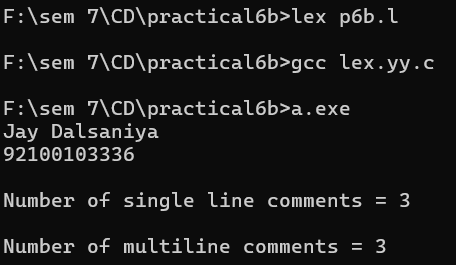
printf("\nNumber of single line comments = %d\n", sl);

printf("\nNumber of multiline comments = %d\n", ml);

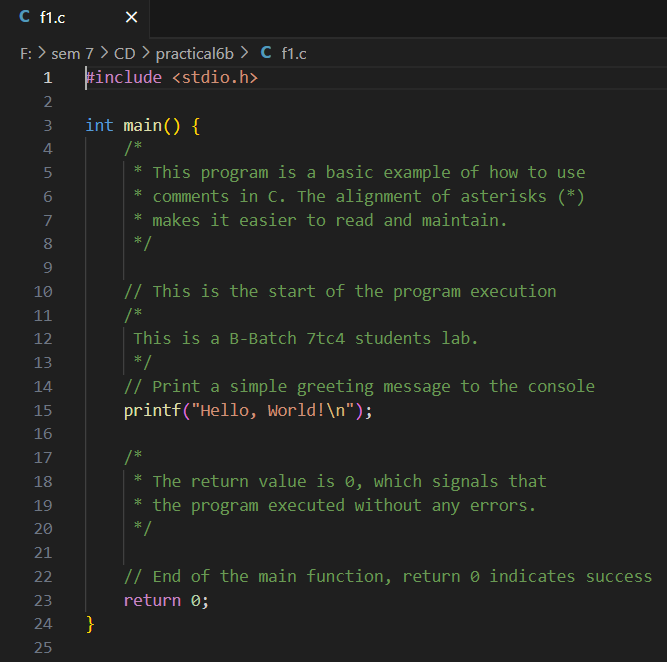
return 0;

}

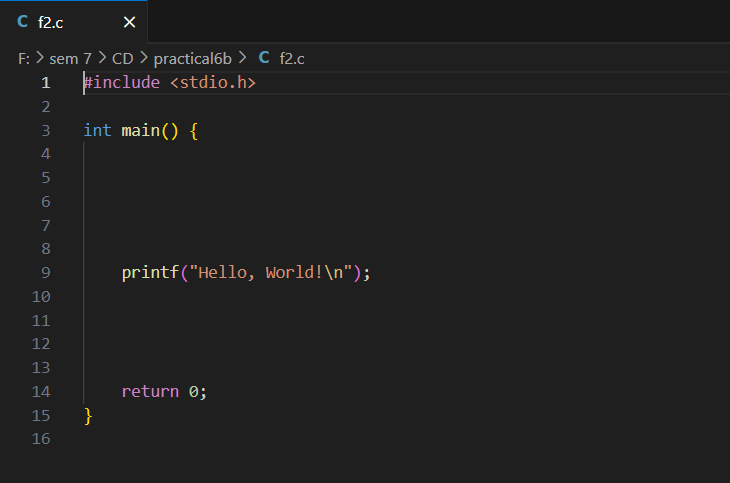
**Output:**

****

**F1 file output:**

****

**F2 file output:**

****

**Practical 7**

**Title: WALex Program to check weather given statement is compound or simple.**

**Hint:** To check whether a given statement is compound or simple using Lex

**Program:**

%{

#include <stdio.h>

int flag = 0; // Flag to determine if the sentence is compound

%}

%%

and|or|but|because|if|then|nevertheless { flag = 1; } // Set flag for compound sentence

[.?!] ; // Match end of sentence punctuation

\n { return 0; } // Return 0 on newline (end of input)

[ \t]+ ; // Ignore whitespace

. ; // Match any other single character (ignore)

%%

int main() {

printf("Jay Dalsaniya \n");

printf("92100103336 \n");

printf("Enter the sentence:\n");

yylex(); // Invoke the lexer

if (flag == 0)

printf("\nThis is a simple sentence.\n");

else

printf("\nThis is a compound sentence.\n");

return 0;

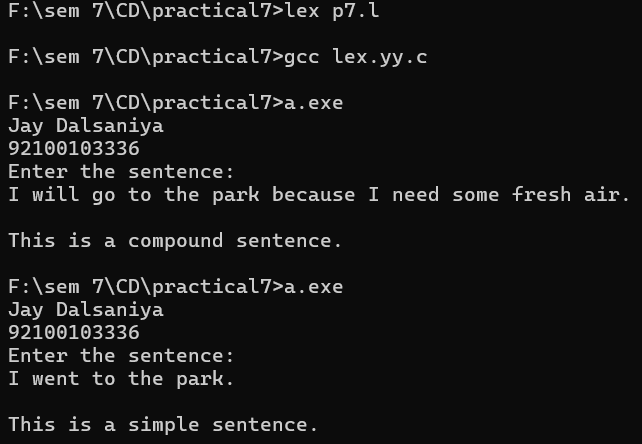
}

int yywrap() {

return 1; // Indicate end of input

}

**Output:**

****

**Practical 8**

**Title: WALex Program to extract HTML tags from .html file.**

**Hint:** In this practical, you will create a Lex program that reads an HTML file and extracts all the HTML tags (elements enclosed within < and >). The extracted tags will be written to an output file.

**Program:**

%{

#include <stdio.h>

%}

%%

\<[^>]\*\> { // Matches anything between '<' and '>' (HTML tags)

printf("%s\n", yytext); // Print the matched HTML tag to the console

fprintf(yyout, "%s\n", yytext); // Write the matched HTML tag to output.txt

}

.|\n; // Matches any other character or newline (ignored)

%%

int yywrap() {

return 1;

}

int main() {

printf("Jay Dalsaniya \n");

printf("92100103336 \n");

yyin = fopen("index.html", "r"); // Input file (HTML file)

yyout = fopen("output.txt", "w"); // Output file (to store HTML tags)

yylex(); // Start lexical analysis

fclose(yyin); // Close input file

fclose(yyout); // Close output file

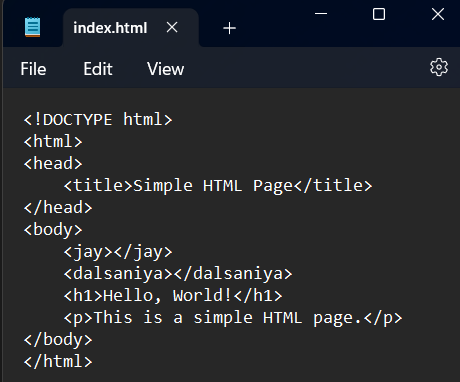
return 0;

}

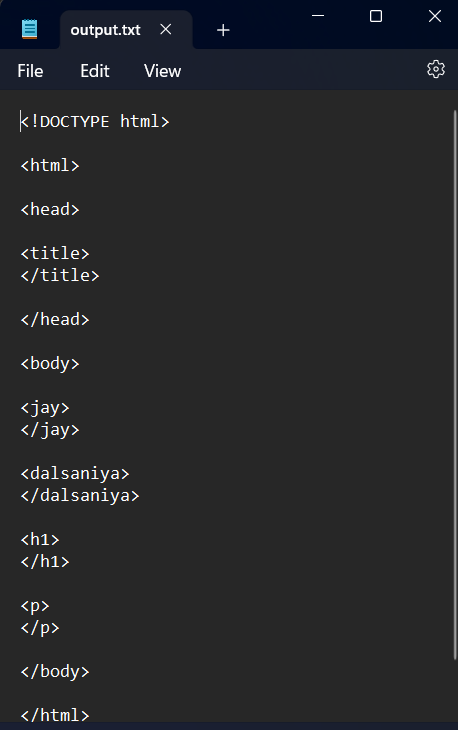
**Output:**

****

**Index.html :**

****

**Output.txt :**

****

**Practical 9**

**Title: Write a C Program to compute FIRST Set of the given grammar.**

**Hint:** Compute the FIRST set of a grammar by recursively determining the first terminals of productions, adding terminals and epsilon (if applicable) for each non-terminal, while avoiding duplicates in the resulting set.

**Program:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX\_PRODUCTIONS 10

#define MAX\_FIRST\_SET 20 // Increased size for FIRST set to handle more elements

int n; // Number of productions

char productions[MAX\_PRODUCTIONS][MAX\_PRODUCTIONS]; // Array to hold productions

char firstSet[MAX\_FIRST\_SET]; // Array to hold FIRST set

void first(char symbol);

int main() {

int i;

char c;

printf("Jay Dalsaniya\nEnroll: 92100103336\n");

printf("Enter the number of productions: ");

scanf("%d", &n);

printf("Enter the productions (epsilon = $):\n");

for (i = 0; i < n; i++) {

scanf("%s", productions[i]);

}

do {

// Clear the FIRST set for each query

memset(firstSet, 0, sizeof(firstSet));

printf("Enter the non-terminal whose FIRST set is to be found: ");

scanf(" %c", &c); // Notice space before %c to consume newline

first(c); // Compute FIRST set

printf("FIRST(%c) = { ", c);

for (i = 0; i < strlen(firstSet); i++) {

if (firstSet[i] != 0) {

printf("%c ", firstSet[i]);

}

}

printf("}\n");

printf("Do you want to continue (0/1)? ");

scanf("%d", &i);

} while (i == 1);

return 0;

}

// Function to compute the FIRST set

void first(char symbol) {

int i, j;

// Check if the symbol is terminal

if (!isupper(symbol)) { // Terminal

if (strchr(firstSet, symbol) == NULL) { // Avoid duplicates

strncat(firstSet, &symbol, 1); // Add to FIRST set

}

} else { // Non-terminal

for (i = 0; i < n; i++) {

if (productions[i][0] == symbol) { // Check productions

if (productions[i][2] == '$') { // Epsilon production

if (strchr(firstSet, '$') == NULL) {

strncat(firstSet, "$", sizeof(firstSet) - strlen(firstSet) - 1); // Add epsilon to FIRST set

}

} else {

for (j = 2; j < strlen(productions[i]); j++) {

if (!isupper(productions[i][j])) { // Terminal

if (strchr(firstSet, productions[i][j]) == NULL) {

strncat(firstSet, &productions[i][j], 1);

}

break; // Stop after first terminal

} else {

first(productions[i][j]); // Recursive call for non-terminal

if (strchr(firstSet, '$') == NULL) {

break; // Stop if there's no epsilon

}

}

}

}

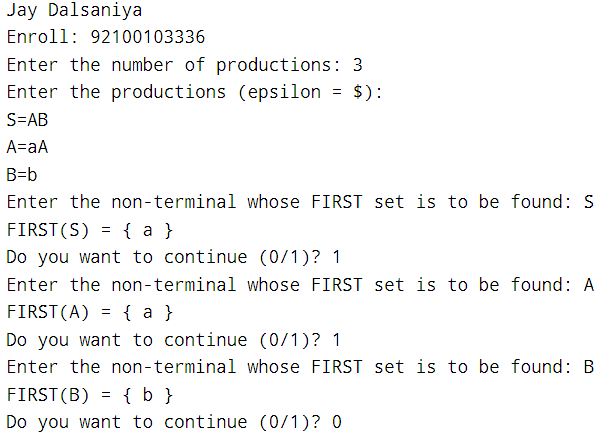
}

}

}

}

**Output:**

****

**Practical 10**

**Title: Write a C Program to compute FOLLOW Set of the given grammar.**

**Hint:** To fix the warning about strncat, ensure that the destination buffer has enough space to accommodate the new character and the null terminator.

**Program:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX\_PRODUCTIONS 10

#define MAX\_FOLLOW\_SET 20 // Increased size for FOLLOW set

int n; // Number of productions

int m = 0; // Index for FOLLOW set

char productions[MAX\_PRODUCTIONS][MAX\_PRODUCTIONS]; // Array to hold productions

char followSet[MAX\_FOLLOW\_SET]; // Array to hold FOLLOW set

void follow(char c);

void first(char c);

int main() {

int i, z;

char c, ch;

printf("Jay Dalsaniya\nEnrollment No.: 92100103336\n");

printf("Enter the number of productions: ");

scanf("%d", &n);

printf("Enter the productions (epsilon = $):\n");

for (i = 0; i < n; i++) {

scanf("%s%c", productions[i], &ch); // Read production rules

}

do {

m = 0; // Reset FOLLOW set index

printf("Enter the element whose FOLLOW is to be found: ");

scanf(" %c", &c); // Space before %c to consume newline

follow(c); // Compute FOLLOW set

printf("FOLLOW(%c) = { ", c);

for (i = 0; i < m; i++) {

printf("%c ", followSet[i]);

}

printf("}\n");

printf("Do you want to continue (0/1)? ");

scanf("%d%c", &z, &ch);

} while (z == 1);

return 0;

}

// Function to compute the FOLLOW set

void follow(char c) {

if (productions[0][0] == c) {

followSet[m++] = '$'; // Add $ to FOLLOW set if c is the start symbol

}

for (int i = 0; i < n; i++) {

for (int j = 2; j < strlen(productions[i]); j++) {

if (productions[i][j] == c) {

if (productions[i][j + 1] != '\0') {

first(productions[i][j + 1]); // Call first for next symbol

}

if (productions[i][j + 1] == '\0' && c != productions[i][0]) {

follow(productions[i][0]); // Follow the left side non-terminal

}

}

}

}

}

// Function to compute the FIRST set

void first(char c) {

int k;

if (!isupper(c)) { // If terminal, add to FOLLOW set

followSet[m++] = c;

} else {

for (k = 0; k < n; k++) {

if (productions[k][0] == c) {

if (productions[k][2] == '$') {

follow(productions[k][0]); // Epsilon production

} else if (islower(productions[k][2])) {

followSet[m++] = productions[k][2];

} else {

first(productions[k][2]); // Recursive call for non-terminal

}

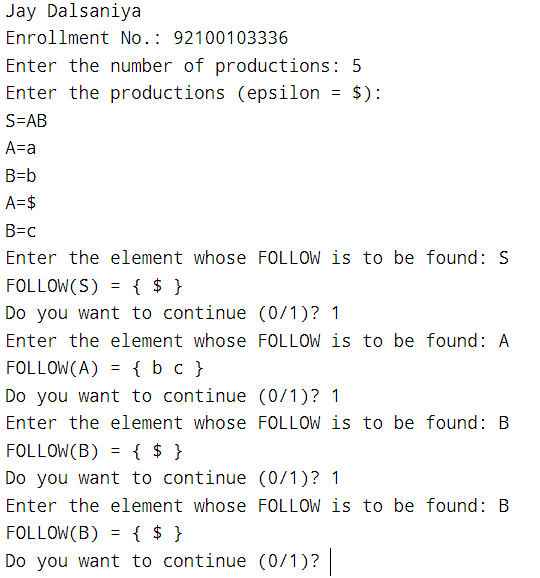
}

}

}

}

**Output:**

****

**Practical 11**

**Title: Write a C Program to implement Operator precedence parser.**

**Hint:** This code implements a shift-reduce parser using a stack to analyze expressions based on predefined grammar rules and operator precedence.

**Program:**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

char \*input;

int i = 0;

char lasthandle[6], stack[50], handles[][5] = {")E(", "E\*E", "E+E", "i", "E^E"};

int top = 0, l;

char prec[9][9] = {

/\*input\*/

/\*stack + - \* / ^ i ( ) $ \*/

/\* + \*/ '>', '>', '<', '<', '<', '<', '<', '>', '>',

/\* - \*/ '>', '>', '<', '<', '<', '<', '<', '>', '>',

/\* \* \*/ '>', '>', '>', '>', '<', '<', '<', '>', '>',

/\* / \*/ '>', '>', '>', '>', '<', '<', '<', '>', '>',

/\* ^ \*/ '>', '>', '>', '>', '<', '<', '<', '>', '>',

/\* i \*/ '>', '>', '>', '>', '>', 'e', 'e', '>', '>',

/\* ( \*/ '<', '<', '<', '<', '<', '<', '<', '>', 'e',

/\* ) \*/ '>', '>', '>', '>', '>', 'e', 'e', '>', '>',

/\* $ \*/ '<', '<', '<', '<', '<', '<', '<', '<', '>',

};

int getindex(char c) {

switch (c) {

case '+': return 0;

case '-': return 1;

case '\*': return 2;

case '/': return 3;

case '^': return 4;

case 'i': return 5;

case '(': return 6;

case ')': return 7;

case '$': return 8;

}

}

int shift() {

stack[++top] = \*(input + i++);

stack[top + 1] = '\0';

}

int reduce() {

int i, len, found, t;

for (i = 0; i < 5; i++) { // selecting handles

len = strlen(handles[i]);

if (stack[top] == handles[i][0] && top + 1 >= len) {

found = 1;

for (t = 0; t < len; t++) {

if (stack[top - t] != handles[i][t]) {

found = 0;

break;

}

}

if (found == 1) {

stack[top - t + 1] = 'E';

top = top - t + 1;

strcpy(lasthandle, handles[i]);

stack[top + 1] = '\0';

return 1; // successful reduction

}

}

}

return 0;

}

void dispstack() {

for (int j = 0; j <= top; j++)

printf("%c", stack[j]);

}

void dispinput() {

for (int j = i; j < l; j++)

printf("%c", \*(input + j));

}

int main() {

int j;

input = (char \*)malloc(50 \* sizeof(char));

// Print name and enrollment number

printf("Jay Dalsaniya\nEnrollment No.: 92100103336\n");

printf("\nEnter the string\n");

scanf("%s", input);

input = strcat(input, "$");

l = strlen(input);

strcpy(stack, "$");

printf("\nSTACK\tINPUT\tACTION");

while (i <= l) {

shift();

printf("\n");

dispstack();

printf("\t");

dispinput();

printf("\tShift");

if (prec[getindex(stack[top])][getindex(input[i])] == '>') {

while (reduce()) {

printf("\n");

dispstack();

printf("\t");

dispinput();

printf("\tReduced: E->%s", lasthandle);

}

}

}

if (strcmp(stack, "$E$") == 0)

printf("\nAccepted;");

else

printf("\nNot Accepted;");

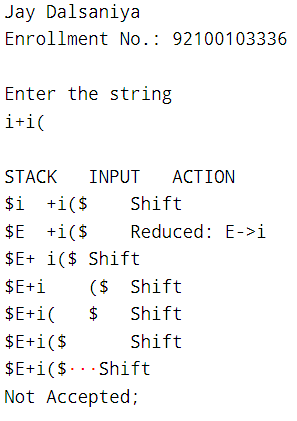
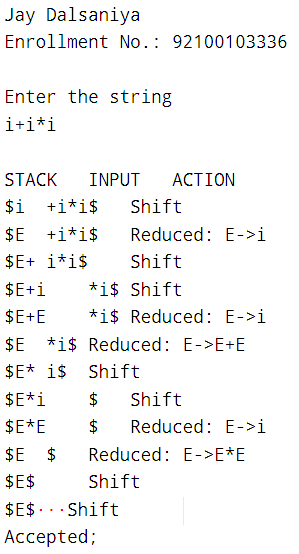
// Free allocated memory

free(input);

return 0;

}

**Output:**

****

**Practical 12**

**Title: Write a C Program for constructing LL (1) parsing.**

**Hint:** To parse arithmetic expressions using a shift-reduce parser, ensure that your parsing table correctly maps non-terminals to productions based on the current input and stack contents.

**Program:**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

char s[20], stack[20];

void main() {

char m[5][6][3] = {

"tb", " ", " ", "tb", " ", " ",

" ", "+tb", " ", "n", "n", "fc", " ", " ", "fc", " ", " ", " ", "n", "\*fc", " a", "n", "n", "i", " ", "", "(e)", " ", " "

};

int size[5][6] = {

2, 0, 0, 2, 0, 0,

0, 3, 0, 0, 1, 1, 2, 0, 0,

2, 0, 0, 0, 1, 3, 0, 1, 1, 1, 0, 0, 3, 0, 0

};

int i, j, k, n, str1, str2;

printf("Jay Dalsaniya\nEnrollment No.: 92100103336\n");

printf("\nEnter the input string: ");

scanf("%s", s);

strcat(s, "$"); // Append '$' to mark the end of input

n = strlen(s);

stack[0] = '$'; // Stack initialization

stack[1] = 'e'; // Start symbol

i = 1; // Stack pointer

j = 0; // Input pointer

printf("\nStack\tInput\n");

printf("\n");

// Parsing loop

while ((stack[i] != '$') && (s[j] != '$')) {

// Check if the top of the stack matches the current input character

if (stack[i] == s[j]) {

i--;

j++;

}

// Determine the current production rules

switch (stack[i]) {

case 'e': str1 = 0; break; // e -> tb | b

case 'b': str1 = 1; break; // b -> +tb | fc

case 't': str1 = 2; break; // t -> i

case 'c': str1 = 3; break; // c -> (e)

case 'f': str1 = 4; break; // f -> i

default: str1 = -1; // Invalid stack character

}

switch (s[j]) {

case 'i': str2 = 0; break; // Input character 'i'

case '+': str2 = 1; break; // Input character '+'

case '\*': str2 = 2; break; // Input character '\*'

case '(': str2 = 3; break; // Input character '('

case ')': str2 = 4; break; // Input character ')'

case '$': str2 = 5; break; // End of input

default: str2 = -1; // Invalid input character

}

// Error handling

if (str1 == -1 || str2 == -1 || m[str1][str2][0] == ' ') {

printf("\nERROR: Invalid input or production rule.\n");

exit(0);

} else if (m[str1][str2][0] == 'n') {

i--; // Do nothing, just pop the stack

} else if (m[str1][str2][0] == 'i') {

stack[i] = 'i'; // Push 'i' onto the stack

} else {

// Expand the stack based on the production rules

for (k = size[str1][str2] - 1; k >= 0; k--) {

stack[i] = m[str1][str2][k];

i++;

}

i--; // Move back to the last pushed item

}

// Print current stack and input status

for (k = 0; k <= i; k++) {

printf("%c", stack[k]); // Print the stack

}

printf("\t");

for (k = j; k <= n; k++) {

printf("%c", s[k]); // Print the remaining input

}

printf("\n");

}

printf("\nSUCCESS\n");

}

**Output:**

|  |  |
| --- | --- |
|  |  |

**Practical 13**

**Title: Write a C program to implement SLR parsing**

**Hint:** The code implements a simple shift-reduce parser using a finite state machine (FSM) and a syntax analysis table (axn). Ensure that the input string adheres to the expected grammar for successful acceptance.

**Program:**

#include<stdio.h>

#include<string.h>

int axn[][6][2]={

{{'S',5},{-1,-1},{-1,-1},{'S',4},{-1,-1},{-1,-1}},

{{-1,-1},{'S',6},{-1,-1},{-1,-1},{-1,-1},{'R',102}},

{{-1,-1},{'R',2},{'S',7},{-1,-1},{'R',2},{'R',2}},

{{-1,-1},{'R',4},{'R',4},{-1,-1},{'R',4},{'R',4}},

{{'S',5},{-1,-1},{-1,-1},{'S',4},{-1,-1},{-1,-1}},

{{-1,-1},{'R',6},{'R',6},{-1,-1},{'R',6},{'R',6}},

{{'S',5},{-1,-1},{-1,-1},{'S',4},{-1,-1},{-1,-1}},

{{'S',5},{-1,-1},{-1,-1},{'S',4},{-1,-1},{-1,-1}},

{{-1,-1},{'S',6},{-1,-1},{-1,-1},{'S',1},{-1,-1}},

{{-1,-1},{'R',1},{'S',7},{-1,-1},{'R',1},{'R',1}},

{{-1,-1},{'R',3},{'R',3},{-1,-1},{'R',3},{'R',3}},

{{-1,-1},{'R',5},{'R',5},{-1,-1},{'R',5},{'R',5}}

};

int gotot[12][3]={1,2,3,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,2,3,-1,-1,-1,

-1,9,3,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}; // GoTo table

int a[10];

char b[10];

int top=-1, btop=-1, i;

void push(int k) {

if(top < 9)

a[++top] = k;

}

void pushb(char k) {

if(btop < 9)

b[++btop] = k;

}

char TOS() {

return a[top];

}

void pop() {

if(top >= 0)

top--;

}

void popb() {

if(btop >= 0)

b[btop--] = '\0';

}

void display() {

for(i = 0; i <= top; i++)

printf("%d%c", a[i], b[i]);

}

void display1(char p[], int m) { // Displays The Present Input String

int l;

printf("\t\t");

for(l = m; p[l] != '\0'; l++)

printf("%c", p[l]);

printf("\n");

}

void error() {

printf("Syntax Error\n");

printf("Given String is rejected\n");

}

void reduce(int p) {

int len, k, ad;

char src, \*dest;

switch(p) {

case 1: dest = "E+T"; src = 'E'; break;

case 2: dest = "T"; src = 'E'; break;

case 3: dest = "T\*F"; src = 'T'; break;

case 4: dest = "F"; src = 'T'; break;

case 5: dest = "(E)"; src = 'F'; break;

case 6: dest = "i"; src = 'F'; break;

default: dest = "\0"; src = '\0'; break;

}

for(k = 0; k < strlen(dest); k++) {

pop();

popb();

}

pushb(src);

switch(src) {

case 'E': ad = 0; break;

case 'T': ad = 1; break;

case 'F': ad = 2; break;

default: ad = -1; break;

}

push(gotot[TOS()][ad]);

}

int main() {

int j, st, ic;

char ip[20] = "\0", an;

int accepted = 0;

printf("Jay Dalsaniya\nEnrollment No.: 92100103336\n");

printf("Enter any String\n");

scanf("%s", ip);

push(0);

display();

printf("\t%s\n", ip);

for(j = 0; ip[j] != '\0';) {

st = TOS();

an = ip[j];

if(an >= 'a' && an <= 'z') ic = 0;

else if(an == '+') ic = 1;

else if(an == '\*') ic = 2;

else if(an == '(') ic = 3;

else if(an == ')') ic = 4;

else if(an == '$') ic = 5;

else {

error();

accepted = 0;

break;

}

if(axn[st][ic][0] == 'S') {

pushb(an);

push(axn[st][ic][1]);

display();

j++;

display1(ip, j);

}

if(axn[st][ic][0] == 'R') {

reduce(axn[st][ic][1]);

display();

display1(ip, j);

}

if(axn[st][ic][1] == 102) {

printf("Given String is accepted \n");

accepted = 1;

break;

}

}

if (!accepted) {

printf("Given String is rejected\n");

}

return 0;

}

**Output:**

|  |  |
| --- | --- |
|  |  |

# **Practical 14**

**Title: Prepare a report on YACC and generate** **Calculator Program using**

**YACC.**

# **Report on YACC**

**Introduction to YACC**

YACC, or Yet Another Compiler Compiler, is a powerful tool used for generating parsers based on a predefined grammar. Developed in the 1970s for the UNIX operating system, YACC plays a crucial role in compiler design, language processing, and syntax analysis tasks. It converts context-free grammar (CFG) rules into C code, helping ensure input or code is syntactically correct.

**Key Features of YACC**

**Context-Free Grammar (CFG):** YACC relies on CFG to define the structure of input, specifying the valid sequences of tokens the parser can recognize.

**Integration with Lexical Analyzer:** YACC works with a lexical analyzer (like Lex or Flex) that converts input into tokens, which are then passed to YACC for parsing.

**C Actions for Grammar Rules:** For each grammar rule, developers can define actions written in C, executed when the rule is matched. These actions enable tasks like building syntax trees or performing calculations.

**How YACC Works**

**Grammar Specification:** Developers define a set of grammar rules that describe the valid language structure. YACC then uses these rules to generate a parser.

**Parsing Process:** The parser reads tokens produced by the lexical analyzer and attempts to match them to the grammar rules. If the input adheres to the rules, YACC executes the corresponding actions.

**Action Execution:** For each matched rule, custom actions are executed, enabling the parser to perform tasks like evaluating expressions or building abstract syntax trees.

**Components of YACC**

**Tokens:** Basic units like numbers, operators, or identifiers recognized by the lexical analyzer and used by YACC during parsing.

**Non-Terminals:** Higher-level structures, such as expressions or statements, defined by combinations of tokens and non-terminals.

**Precedence and Associativity:** YACC allows specifying precedence and associativity rules to resolve grammar ambiguities, such as operator precedence.

**Applications of YACC**

**Compiler Development:** YACC is used to build parsers for compilers, converting source code into syntax trees or intermediate code.

**Interpreters:** It helps create interpreters that parse and directly execute commands or evaluate expressions.

**Expression Evaluators:** YACC is commonly used in calculators or expression evaluators, parsing and computing arithmetic expressions.

**Configuration Processing:** It can also be applied to process configuration files or command-line arguments, ensuring correct syntax.

**Advantages of YACC**

**Efficiency:** YACC generates efficient parsers that handle complex input structures with speed.

**Ease of Use:** YACC’s syntax for defining grammar rules is simple, and integrating the parser into C programs is straightforward.

**C Integration:** YACC’s output is C code, making it easily embeddable in C-based systems.

**Limitations of YACC**

**Context-Free Grammar Only:** YACC is limited to context-free grammars and cannot handle more complex, context-sensitive languages.

**Manual Error Recovery:** While basic error handling is provided, more advanced error recovery requires custom code.

**Left Recursion Issues:** YACC doesn’t handle left-recursive rules well, requiring developers to convert left-recursive grammars to right-recursive formats.

**Conclusion**

YACC remains a valuable tool in compiler construction and language processing. Despite some limitations, such as handling only context-free grammars and requiring manual error recovery, its efficiency, ease of integration with C, and ability to generate fast parsers make it indispensable in many programming and language design contexts.

# **Calculator Program**

**Code**

**Parser.y**

%{

#include <stdio.h>

#include <stdlib.h>

int flag = 0;

%}

%token NUMBER

%left '+' '-'

%left '\*' '/' '%'

%left '(' ')'

/\* Rule Section \*/

%%

ArithmeticExpression:

E {

printf("\nResult = %d\n", $$);

return 0;

}

;

E: E '+' E { $$ = $1 + $3; }

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

| E '\*' E { $$ = $1 \* $3; }

| E '/' E { $$ = $1 / $3; }

| E '%' E { $$ = $1 % $3; }

| '(' E ')' { $$ = $2; }

| NUMBER { $$ = $1; }

;

%%

// Driver code

int main() {

printf("\nEnter any arithmetic expression (supports +, -, \*, /, % and parentheses):\n");

yyparse();

if (flag == 0) {

printf("\nEntered arithmetic expression is valid\n\n");

}

return 0;

}

void yyerror() {

printf("\nEntered arithmetic expression is invalid\n\n");

flag = 1;

}

**Scan.l**

%{

#include <stdio.h>

#include <stdlib.h>

#include "parser.tab.h"

extern int yylval;

%}

/\* Rule Section \*/

%%

[0-9]+ {

yylval = atoi(yytext);

return NUMBER;

}

[\t] ; /\* Ignore tabs \*/

[\n] return 0; /\* End of line \*/

. return yytext[0]; /\* Return any other character \*/

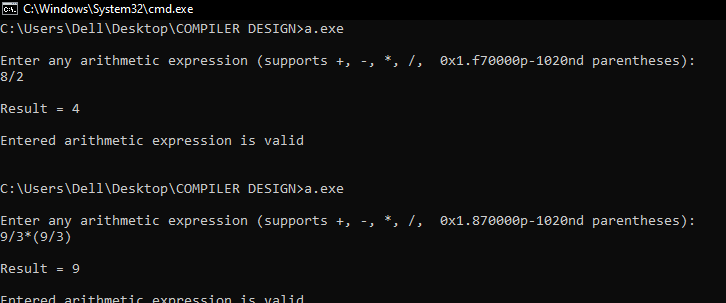
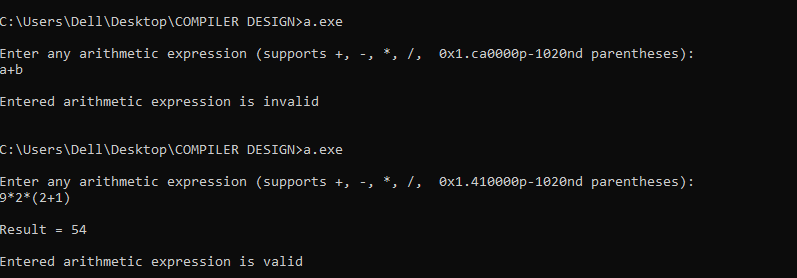
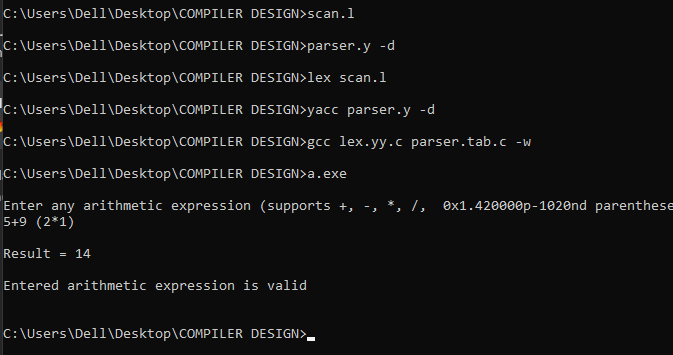
%%

int yywrap() {

return 1;

}

**Output:**

****