**SYMBOL TABLE :**

➔Create a user specific data structure called “symbolTab” that is used to contain 2 string

variables representing the data type and variable name respectively along with an integer

variable for the address allocation of the variable.

➔Then we take two variables “first” and “last” to specify the first and last indexes of the

symbol table along with a global variable size to specify the number of variables allotted in

the program.

➔Allocate a variable address to specify the starting address to allocate variables with

respect to particular data types.

➔Create

 two file pointers one for writing and another for reading an error logs file that

were observed during compilation of the code.

 a search function, that will check whether a variable received from the parser is

already present within the symbol table or not.

 a function isKeywordVar to check is the variable is of the mention data types being

either “int”, “float”, “long”, etc.

 a function insert that will check if the variable name is already present in the

symbol table or not. If present, then it will place an error message within the

errorLogs file else will insert the variable name with its data type and address

within the user defined data type symbolTab.

 a function “display” that will display the entire symbol table from the first pointer

till the last created after reading the entire code entered by the user.

 a function isDelimiter(char ch) that takes character input to mark delimiter i.e. end

of statement.

 a function isOperator(char ch) that takes character input to check whether it is an

operator or not.

 a function validIdentifier(char \*str) that takes a string input to check whether it is

a valid identifier or not bsed to criteria that it does not start with a number.

 a function isKeyword(char \*str) that takes string input to check whether it is a

keyword or not.

 a function isInteger(char \*str) that takes string input to check whether it is an integer

or not.

 a function isRealNumber(char \*str) that takes string input to check whether it is a

real number or not.

 a function \*trim(char \*s) that takes a string as input and trims the extra white

spaces from start and end of the string.

 a function isComment(char \*str) that takes string input and checks whether the

string starts with (‘//’) or (‘/\*’ and ends with ‘\*/’).

 a function isHeader(char \*str) that takes string input and checks whether the

string ends with ‘.h’ making it a header file.

 a function isSpecial(char \*str) that takes a string input to check whether it is a

valid identifier or not bsed to criteria that it does not contain a special

character.

 a function isFormat(char \*str) that takes a string input to check whether it is a

format specifier or not based on whether it is of the pattern “%d” or “%ld”, etc.

 a function \*subString(char \*str, int left, int right) that takes a string, and 2 indexes

left and right as input to extract the substring form a larger string str.

 a function parse(char \*str) that takes a large string as input. Then declare a file

pointer \*fcode. Then open a file temporary “codefile.txt” that is pointed by fcode.

➔Then check with the boolear function isComment(str) if the string is a comment or not.

➔Check the conditions:

 If the string is not a comment line, the we start checking each string based on index

extraction of strings from left and right, thereby checking isDelimiter(str[right]), then

increment right by 1.

 if isDelimiter(str[right]) == true && left == right, then we check if

isOperator(str[right]) == true, to store in the file that the character is an

operator.

 else if isDelimiter(str[right]) == true && left != right || (right == len && left != right),

then char \*subStr = subString(str, left, right - 1)

➔Then we check whether the substring is a keyword or header file or integer or real

number or format specifier or contains special character or is a valid identifier based on the

boolean functions we created above and save the corresponding message in the file.

➔If the variable is a keyword, we copy it into a temporary variable.

➔If we find a valid identifier, then check if it is preceded by a data type specified

earlier. If it is specied, then we pass the variable to our insert function else we place a

message into the error log file.

➔Then we make left = right and close the file finally with fclose(fcode).

➔Within the main function, we ask the user to enter the code and terminate entering by

pressing “Ctrl+Z” followed by “Enter”.

➔We create another file pointer FILE \*fptr.

➔As the user enters code line by line, we use a counter to count the number of lines in the

program input by the user. Then we print the lexical analysis of the input code by taking the

first string from the file str = fgetc(fptr).

➔While we do not reach the end of file, we print the content of the file. Then we close

the file with fclose(fptr) and print the total number of lines in the program.

➔We then print the entire symbol table with the display function mentioned above.

➔We start reading the error logs file to print any error (if any) within the code we

passed into our program and print them.

➔Finally, we remove both the temporary files with remove("codefile.txt") and

remove("errorLogs.txt") respectively.

**MISPELLED WORDS:**

 Initialize the file pointers, predefined keywords, operators in separate arrays.

 Read every line of the input file by reading each character and appending it to

another string “line” till it encounters ‘\n’.

 Now check whether line is a comment or not.

 Run a loop through the line string and extract words from it using special

characters as delimiters.

 For each special character encountered, check whether the character is an

operator or not and then send the extracted word into the function that will

decide what category of Lexeme it is.

 If it is one of the keywords, then the output is 1, if it is a valid identifier then

output is 2, if it is a valid function the output is 3, otherwise output is 0.

 Write onto the respective files according to the classified data.

 Print the number of lines as output.

 Build a hash table (array of linked lists) for each bucket of words found.

 Hash function takes the word as input and returns

(sum\_of\_ascii\_values\_of\_letters) % (size\_of\_word).

 Each integer returned by the hash function corresponds to a given bucket and

each bucket has a linked list attached to it.

 This reduces the time to search the dictionary as the programs has to look through

only the correct bucket from the hash table instead of the entire dictionary.

 Load the words from the dictionary and create the hash table.

 If the current word from the input file has a (or < next to it then check for spelling

in the hash table using check().

 Print the misspelled word if check() returns false.

FIRST\_FOLLOW

 Create a function findfirst(char c, int q1, int q2) and check if the character is a

terminal, then store it within the array first[]. Then we check the production matrix

to check if the character is in the first place or in the last place. If the string gets

terminated, then the results are stored within first, else if the query checking is not

for the character in the first place, then a recursive call is made to the function

findfirst(char c, int q1, int q2), else the check for terminal epsilon, which in this case

is ‘#’.

 For each row, check if the First of c has already been calculated or not.

 Then the first for the grammar is printed for each variable present within the

grammar based on the terminals.

PARSE TREE

Declare a function isUpper (char c) that checks if the input character passed as

parameter is a character or not.

 Declare a function substring (char \*str, int left, int right) that extracts the substring

from the input string based on the indexes of left and right

 In the main method, declare the production matrix, the input string, count and

reference string.

 Enter the production rules of grammar in the production matrix in a synchronised

order using strcpy() function.

 Print the production rules of the original grammar from the production matrix

created above.

 Take the language input from the user and store it inside the string s

 Print the updates occurring in the original grammar starting with the original form

from production [0].

 Start a loop for the number of rows of the production rules according to the

production matrix.

 Retrieve the right part of the original production with the substring function.

 Then take a temporary string modStr where the string and characters are

concatenated.

 For each production rules with values in common, corresponding string or character

is applied based on the condition within another nested loop.

 Then check whether the final temporary string generated has uppercase characters

in it or not at each step.

 If the final string has no upper-case letters and the output matches the input

language, then use a flag variable.

 If the flag variable is 1 then print the particular language can be retrieved from the

given grammar.

 Else print the particular language cannot be retrieved from the given grammar.

**GRAMMAR ACCEPTED BY LL1 PARSER**

Declare a matrix table of int type, 2 arrays terminal and nonterminal of character

type.

 Declare a structure product having a string and an integer in it.

 Declare number of productions, first and follow matrix along with another first\_rhs

matrix.

 Create a function that checks if a symbol is a non-terminal or not.

 Declare a function readFile that reads the input from the inputFile.txt.

 Based on the terminal and nonterminal, the production rules are generated on the

string buffer.

 Declare a function add\_FIRST\_A\_to\_FOLLOW\_B(char A, char B) to work on the first

method for production rules.

 Declare another function add\_FOLLOW\_A\_to\_FOLLOW\_B(char A, char B) to add the

follow method for production rules.

 Declare function Follow that generates the follow terminals of the input grammar.

 Declare function add\_FIRST\_A\_to\_FIRST\_B(char A, char B) to add elements to first of

production rules based on need.

 Declare the First function that generates the first terminals of the input grammar.

 Declare function add\_FIRST\_A\_to\_FIRST\_RHS B(char A, int B) to work for the first

rhs method based on production rules.

 Decare function FIRST\_RHS() to generate the first from the right hand side of the

production rules.

 In the main method, print the input grammar first followed by the first and the

follows terminals var each variable present in the production rules.

 The rules for checking are defined considering ‘#’ as the epsilon.

 Check if any of the derived strings for parse table contains null ending due to

multiple production rules satisfying the same condition of each variable.

 In case of null string print, the particular grammar is not accepted by LL (1) parser.

 In case of no null string print, the particular grammar is accepted by LL (1) parse and

a parse table can be generated from it.

**PARSE TABLE LL(1)**

 Declare a matrix table of int type, 2 arrays terminal and nonterminal of character

type.

 Declare a structure product having a string and an integer in it.

 Declare number of productions, first and follow matrix along with another first\_rhs

matrix.

 Create a function that checks if a symbol is a non-terminal or not.

 Declare a function readFile that reads the input from the inputFile.txt.

 Based on the terminal and nonterminal, the production rules are generated on the

string buffer.

 Declare a function add\_FIRST\_A\_to\_FOLLOW\_B(char A, char B) to work on the first

method for production rules.

 Declare another function add\_FOLLOW\_A\_to\_FOLLOW\_B(char A, char B) to add the

follow method for production rules.

 Declare function Follow that generates the follow terminals of the input grammar.

 Declare function add\_FIRST\_A\_to\_FIRST\_B(char A, char B) to add elements to first of

production rules based on need.

 Declare the First function that generates the first terminals of the input grammar.

 Declare function add\_FIRST\_A\_to\_FIRST\_RHS B(char A, int B) to work for the first

rhs method based on production rules.

 Decare function FIRST\_RHS() to generate the first from the right hand side of the

production rules.

 In the main method, print the input grammar first followed by the first and the

follows terminals var each variable present in the production rules.

 The rules for checking are defined considering ‘#’ as the epsilon.

 Finally print the parse table generated within the table matrix via the production

structure indexing of the string and its particular length. This gives the final parse

table via 2 nested loops.

**LL(1) WITH STACK**

 First, check for left recursion in the given grammar. If left recursion is present, remove

it and proceed.

 Calculate the First() and Follow() for all non-terminals.

 Find First(α) and for each terminal in First(α), make entry A –> α in the table.

 If First(α) contains ε (epsilon) as terminal than, find the Follow(A) and for each

terminal in Follow(A), make entry A –> α in the table.

 If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> α

in the table for the $.

 Finally, to check whether an input string can be generated from the parsing table, use

the below mentioned algorithm:

Let given grammar is G = (V, T, S, P)

where V-variable symbol set, T-terminal symbol set, S- start symbol, P- production

set.

while(stack is not empty) {

// initially it is S

A = top symbol of stack;

//initially it is the first symbol in string

r = next input symbol of given string;

if (A∈T or A==$) {

if(A==r){

pop A from stack;

remove r from input;

}

else{

}

ERROR();

else if (A∈V) {

if(PT[A,r]= A⇢B1B2....Bk) { //PT=Parsing Table

pop A from stack;

push Bk,Bk-1.......B1 on stack

}

P a g e | 79

nclude <std o.h>

nclude <ctype.h>

nclude <str ng.h>

vo d followf rst(char,nt, nt);

vo d f ndf rst(char, nt, nt);

vo d follow(char c);

nt count, n = 0;

char calc\_f rst[10][100];

char calc\_follow[10][100];

nt m = 0;

char product on[10][10], f rst[10];

char f[10];

nt k;

char ck;

nt e;

nt ma n( nt argc, char \*\*argv)

{

nt jm = 0;

nt km = 0;

nt , cho ce;

char c, ch;

pr ntf("Enter number of product on rules : ");

scanf("%d", &count);

pr ntf("\nEnter %d product on rules:\n\n", count);

for ( = 0; < count; ++)

{

scanf("%s%c", product on[ ], &ch);

}

nt kay;

char done[count];

nt ptr = -1;

for (k = 0; k < count; k++)

{

for (kay = 0; kay < 100; kay++)

{

calc\_f rst[k][kay] = '!';

else if (PT[A,r] = ERROR())

ERROR();

}

}

**LR(0) AND SLR(1)**

 Include the basic header files for performing string manipulations, including

<stdlio.h>, <stdlib.h> and <string.h>.

 Include the external header files created by us majorly being the closure\_goto.h,

parsingtable.h and first\_follow.h.

 Call the functions compute\_first() and compute\_follow() to calculate the first and

follow of the input grammar non-terminals.

 Call the create\_parsing\_table() function present within the parsingtable.h header file

to form the respective parsing table

 Within the closure\_goto.h header file, declare a function check() that returns 1 if the

production strings encounter a character that is a terminal and return 0 otherwise.

 Then declare the function generate\_terminals() that augments the productions rules

to formulate the final set of rules to generate the terminals of the input productions.

 Then declare another function check2() to return 1 for non-terminals in input

production rules and 0 otherwise.

 Then declare another function generate\_nonterminals() to generate the non-

terminals from the input production rules similar to the way we generated the

terminals above.

 Declare function initialize\_items() to call the generate\_terminals() and

generate\_nonterminals() functions and calculate the number of items in total.

 Declare function generate\_item() to generate the final production rules to iterate

over to find the final results.

 Declare functions item\_found() and isterminal() to check for a particular item within

a production rule and check whether a particular character is a terminal or not

respectively

 Declare a function closure to work with the canonical forms of the input production

rules via working with the ‘.’points added to form the corresponding item states.

 Declare functions Goto1(), state\_found() and transition\_item\_found() to mark the

goto points during canonical analysis, checking for the states in the production rules

along with the transition items to derive the final result.

 Declare the functions compute\_closure\_goto() to compute the goto actions of the

canonical structure for different item states along with print() to print the total

number of states along with each item states derived from the input production

rules.

 Declare a flag variable flUse along with the struct to define a new user defined data

type consisting of char type Action table and an int type Goto table with the struct

being called as table and display the final table produced for the input grammar.