POORNIMA

ZERO LAB

Ques-1// What is compiler?

⇒ A compiler is a specialized program that translates codes written in a high level language program like C, C#, Java etc. into lower level language usually machine code and intermediate language code like assembly which can then be executed by the computer's CPU.

Ques-2// What type of ~~pgr~~ programmed compiler is?

⇒ A compiler is typically a system software program that manages and control the hardware enabling application software to function compiler Itself is usually written in a programming language that is either low level or high level but efficient and close to hardware. Choice of language for writing a compiler is influenced by need for ~~perfo~~ platfrom portability and control hardware resources.

Ques-3// What is need of compiler?

⇒ Key reasons why compiler is needed.

- Translate from high level to machine level code.
- Error Checking - Syntax and semantic validation and help identifying incorrect error early in development process, improve code reliability and reduce runtime errors.

POORNIMA

- Optimization - Optimize the code make it run faster or use less memory compiler reduce the number of instructions
- Portability : many compiler are platform independent (eg. windows, linux mac)
- Security : In inforce strict type checking and other rules that prevent certain types of bugs and make it more secure.

Ques-4) How compiler was evolved?

Early Programming (1940's - 1950's)

- earlier programmers write instructions directly in machine code or assembly language.

Introduction to High level languages (1950's)

- IBM developed the 1st HLL designed for scientific and engineering applications.

Advancement of compiler Theory (1960's)

- Algorithmic language introduced block structure & scope influencing many future languages.

Virtual Machines (1990's)

- Introduction to Java brought concept of JVM & just in time compilation which compile bytecode to machine code at run time.

POORNIMA

Modern Compilers (2005)

- Low level virtual machine introduce a moduler & reusable compiler infrastructure, enabling development of languages & tools

Futur Direction

- Research is ongoing into using AI & machine learning to improve compiler optimization.

Q - What are various phase of Compiler?

⇒ 1) Lexical Analysis: Compiler read the source code & break it down into tokens.

2) Syntax Analysis: Tokens are arranged into syntax tree based on grammer of programming language

3) Semantic Analysis: Checks the syntax tree for sematic error, ensure program sense logically.

4) Intermediate Code Generation: Compiler transforms syntax tree into intermediate code, which is lower-level. representation but not yet machine code.

5) Code Optimisation: Intermediate code is optimized to improve efficiency.

6) Code Generation: Optimized intermediate code is translated to machine code, which the CPU can execute

Finite Infant:

For 1: int
For 2: Hi
For 3: unsigned
For 4: Hello
For 5: switch

Output

For 1: It is a keyword
For 2: It is not a keyword
For 3: It is a keyword
For 4: It is not a keyword
For 5: It is a keyword.

POORNIMA

EXPERIMENT - 1

Objective: To identify whether the given string is keyword or not.

About the operation: In this program we are taking string as input from user and we are checking that the given string is a keyword or not.

Algorithm / Procedure:
Here we are making keyword() function which will return 1 if string is keyword and it will return 0 if string is not keyword.

- in main() function we are giving a string
- then we are calling the iskeyword() function.
- If iskeyword() is returning 1, then it means the given string is keyword.

But if iskeyword() is returning 0, then it means given keyword string is not a keyword.

code:

```
#include <stdio.h>
#include <string.h>
int iskeyword (char* str) {
        char* keyword [] = {"auto", "break", "case", "char",
            "const", "continue", "default", "do", "int", "long"}
```

POORNIMA

```
int n = sizeof (keyword) ;
for (int i = 0; i<n; i++) {
    if (strcmp (str, keyword [i]) == 0) {
        return 1;
    }
}
return 0;
}

int main () {
    char str [100];
    printf( "Enter a string \n");
    scanf ("%s", str);
    if ( iskeyword (str)) {
        printf ("%s" is a inbuilt keyword"; str);
    }
    else {
        printf ("%s" is not a inbuilt function"; str);
    }
    return 0;
}
```

Finite Input

File text : Hi, int and float are present

Output

Hi is not a keyword
int is a keyword
and is not a keyword
float is a keyword
are is not a keyword
present is not a keyword
Total keyword : 2

POORNIMA

EXPERIMENT -2

Objective: Count total no. of keywords in a file. [Taking file from user].

About the operation: The program reads the content of a file provided by the user break the text into individual words, and check if each word is a C programming language keyword. It maintains a count of how many keywords are found in a file.

Algorithm / procedure:

- define a list of C keywords: create an array that stores all C keywords.
- Prompt the user for a file: ask the user to enter the file name.
- Open the file: use open(). If file can't be opened. display an error message and exit the program.
- read and close the file.
- check if the word is a keyword.
- count the keywords.
- display the result

Code:

```
#include <stdio.h>
# include <string.h>
```

POORNIMA

```
const char* keyword[] = {"auto", "break", "case", "char",
    "const", "continue", "default", "do", "double", "else",
    "enum", "return", "float", "for", "goto", "if", "inline",
    "int", long", "register", "short", "signed", "sizeof",
    "static", "switch", "typeof", "union", "unsigned", "void",
    "while", "Bool"};

int num-keywords = sizeof(keywords) / sizeof(keywords[0]);
int isKeyword (char* token) {
    for (int i=0 ; i<num-keywords ; i++) {
        if(strcmp (token, keywords [i] == 0) {
            return 1;
        }
    }
    return 0;
}
int main () {
    FILE *file;
    char filename [100], token [100];
    int keyword count = 0;
    printf("Enter file name:");
    scanf("%s", filename);
    file = fopen(filename, "r");
    If (file == NULL) {
        printf("Error: could not open file %s\n", filename);
        return 1;
    }
```

POORNIMA

```
while (fscanf(file, "%s", token) != EOF) {
    if (iskeyword(token)) {
        keyword_count++;
    }
}
fclose(file);
printf("Total number of keywords in the file
  %d \n", keyword_count);
return 0;
}
```

5m/
18/10/24

Finite Input

Hello, Himank please check 5+4-3=6

Output

Number of operators = 4

POORNIMA

EXPERIMENT-3

Objective : Count total no. of operators in a file [Taking file from user]

About the operation : Operators are symbols that tell the compiler to perform specific mathematical logic. This program will identify common operators such as +, -, *, %.

Algorithm / procedure :

1) Input: file name is provided by the user.
2) Steps : · define a list of common operators in C lang.
   · Open the file for reading.
   · Read the file content word by word.
   · for each word check whether it matches any operator.
   · maintain the count of all the operators found in the file.
3) Output : · Print the total number of operators found in the file.

Code:

```
#include <stdio.h>
#include <string.h>
```

```
const char* operators[] = {"+", "-", "*", "%", "/", "=",
                    "&&", "==", "!=", "<", ">"};
int num_operators = sizeof(operators)/sizeof(operators[0]);

int is_operators (char* word){
    for(int i=0; i<num_operator; i++) {
        if(strcmp(word, operators[i]==0){
            return 1;
        }
    }
    return 0;
}
int main() {
    FILE* file;
    char filename[100], word[100];
    int operator_count = 0;
    printf("Enter the filename: ");
    scanf("%s", filename);
    if(file == NULL){
        printf("Error: could not open file %s \n",
                    filename);
        return 1;
    }
    while (fscanf(file, "%s", word) {
        if(is_operator (word)){
            operator_count++;
        }
    }
```

POORNIMA

```
fclose(file);
printf("Total number of operators in the file :
            %d \n", operator_count);
return 0;
}
```

17/10/24

Filwrite: Input

hello world!!

Output

Character occure in the file as as:
ASCII [32] occurs 2 times

! character occurs 2 times
d character occurs 1 times
e character occurs 1 times
h character occurs 1 times
l character occurs 3 times
o character occurs 2 times
r character occurs 1 times
w character occurs 1 times.

EXPERIMENT-4

Objective: Write a program to count the total occurance of each character from an input (input is taken from a file)

About the program: The program reads a file and counts the occurance of each character in it. uses an array to count. Then display how many times each character appears distinguishing b/w printable & ascii value for non-printable.

Procedure: 1> The user is prompted to each the name of text file they want to analyze.

2> Program attempts to open specific file in read mode if it fails, it notify user.

3> It read file character by character, using array to tally how many times each character appears.

4> After processing file, it outputs the count of each character, highlighting both printable & Non-printable ASCII values.

5> Finally program closes the file to free resources.

POORNIMA

Program:

```
#include <stdio.h>
#include <stdlib.h>
#define max_line_length 256

void char_occ_count (FILE *file, int count[]) {
        char ch;
        while ((ch=fgetc(file))!= EOF) {
                count[unsigned(ch)]++;
        }
}
void display_cc (int count []) {
        printf("character occurance in the file is as : \n");
        for(int i=0; i<max_line_length ; i++) {
            if (count[i]>0) {
                if(i>32 && i<=126) {
                    printf("%c character occurs %d times\n",
                            i, count [i]);
                }
                else {
                    printf("ASCII [%d] occurs %d times \n", i,
                            count [i]);
                }
            }
        }
}
```

```
int main () {
    char Filename [50];
    File * file;
    int count [max_line_length] = {0};
    printf ("Enter the name of file \n");
    scanf ("%s", filename);
    file = fopen (filename, "r");
    if (file == NULL) {
            printf ("file does not exist \n");
            return 0;
    }
    char_occ_count (file, count);
    fclose (file);
    display_cc (count);
    return 0;
}
```

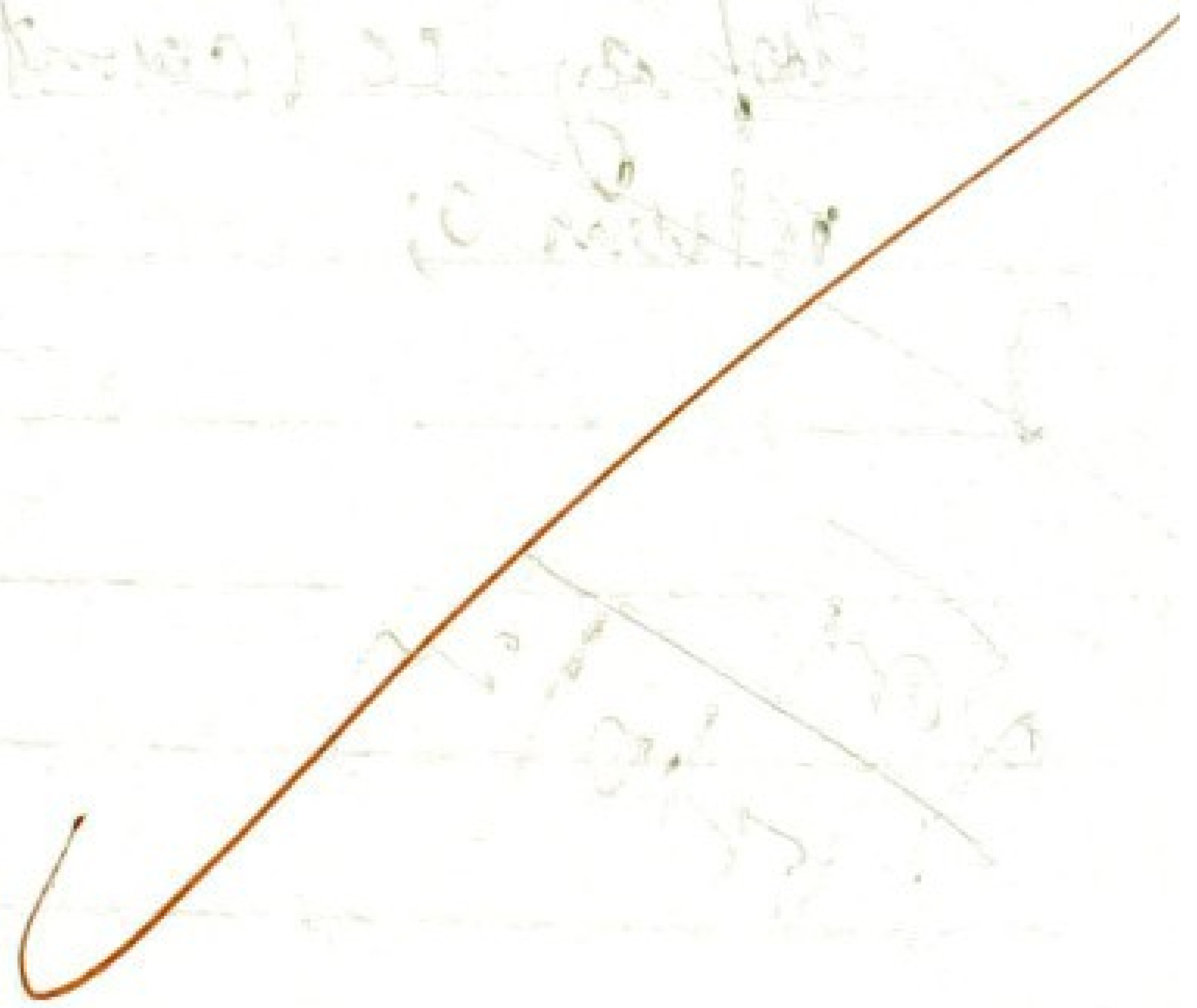
17/10/24

Symbol Added : aman and Type : int
Symbol Added : str and Type : string

Symbol Table is
Symbol : aman and Type : int
Symbol : str and Type : string
Symbol : aman is deleted from Symbol Table

Symbol Table is
Symbol : str and Type : string
Symbol : fit and Type : Float

Symbol Table is
Symbol : and Type Float
Symbol : and Type : string



POORNIMA

EXPERIMENT-5

Objective: Write a C program to implement symbol table operations - insertion, deletion, display.

About the Program: Implements a basic symbol using linked list data structure. It allows for inserting new symbols with their types, deleting existing symbols, and traversing the ~~displ~~ symbol table to display its control.

Procedure:

1> **Creating Symbol:** A new symbol is created using create function, store the symbol & its type & next pointer to NULL.

2> **Insert symbol:** Insert function is used to insert to created node in linked list. Symbol is added to linked list.

3> **Delete Symbol:** Delete function searches for symbol in list & removes of it from the linked list & update pointer to maintain linked list.

4> **Traverse Table:** Traverse function iterates through list displaying all symbols & their types.

Code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct st { char symbol [10];
            char type [10];
            struct st *next;
};
```

POORNIMA

```
struct ST* head= NULL;
struct ST* create (char symbol[], char type[]){
    struct ST* new = (struct ST*) malloc (sizeof(struct ST));
    strcpy(new→symbol, symbol);
    strcpy( new → type, type);
    new → next = NULL;
    return new;
}
void insert( char symbol[], char type[]) {
    struct ST* new = create (symbol, type);
    if (head == NULL) {
        head= new ;
        printf ("Symbol added : %s & Type %s\n", symbol,
              Type);
        return;
    }
    struct ST* temp = head;
    while (temp!= NULL) {
        if (strcmp (temp→symbol, symbol) == 0) {
            printf("ERROR : %s already exist in
                 symbol table \n", symbol);
            free(new);
        }
        temp= temp→next;
    }
    new → next = head; head = new;
    printf("Symbol Added : %s & Type : %s \n", symbol, type);
}
```

POORNIMA

```
void Delete (char symbol [ ]) {
    struct ST* temp = head;
    struct ST* prev = Null;
    if (temp == NULL) {
        printf("Symbol Table is empty \n");
        return;
    }
    if (strcmp (head -> symbol ,symbol) == 0) {
        head = head -> next;
        free (temp);
        printf ("Deleted: %s", symbol);
        return;
    }
    while (temp != NULL) {
        prev = temp;
        temp = temp -> next;
        if (temp == NULL) {
            printf("Symbol: %s does not exist};
            return;
        }
        if (strcmp (temp -> symbol, symbol) == 0) {
            prev -> next = temp -> next;
            free (temp);
            printf("Symbol : %s is deleted from symbol table",
              symbol);
            return;
        }}
}
```

POORNIMA

```
void traverse() {
    printf("\n Symbol Table is:\n");
    struct ST* temp = head;
    while(temp != NULL) {
    printf("Symbol: %s & Type: %s", temp->symbol, temp->type);
    temp = temp->next;
    }
}
int main() {
        char a[] = "aman";
        char b[] = "int";
        char c[] = "str";
        char d[] = "string";
        char e[] = "fit";
        char f[] = "float";
        insert(a, b);
        insert(c, d);
        traverse();
        delete(a);
        traverse();
        insert(e, f);
        traverse();
        return 0;
}
```

17/10/22

Output
Line : 2
Blankspaces : 6
words : 5

POORNIMA

Experiment - 6

Object: write a lex program to count blank spaces, words, lines in a given file.

About the program: The lex program is an example of how you can use regular expression to count specific features (Such as blank spaces) in c text file. The program efficiently processes input file & outputs the desired counts

Procedure:

1) Install lex: first make sure you have lex installed on your system
2) Create a lex file: Write the lex program in .l file.
3) Generate C code: Use lex to generate C code from the lex file.
4) Compile the c program: Use a c compiler to compile the generated c code

Program:

```
%{
#include <stdio.h>
#include <ctype.h>
int. blank_count = 0;
int word_count = 0;
int line_count = 0;
%}
%%
\n {line_count ++;}
```

```
[[:space:]] + {blank.count++;}
[a-z A-Z] + {word-count++;}
%%

int main (int argc, char *argv[]) {
  if (argc != 2) {
     printf("Usage: %s <filename> /n", argv[0]);
     return 1;
  }
  File *file = fopen (argv[1], "r");
     if (file == NULL) {
     perror ("Error opening file");
     return 1;
  }
 yyin = file
 yylex();
 printf ("lines : %d /n", line-count);
 printf (" Blank Spaces %d /n", blank_count);
 printf ("words: %d /n", word-count);
  fclose (file);
  return 0;
}
```

Sample OUTPUT

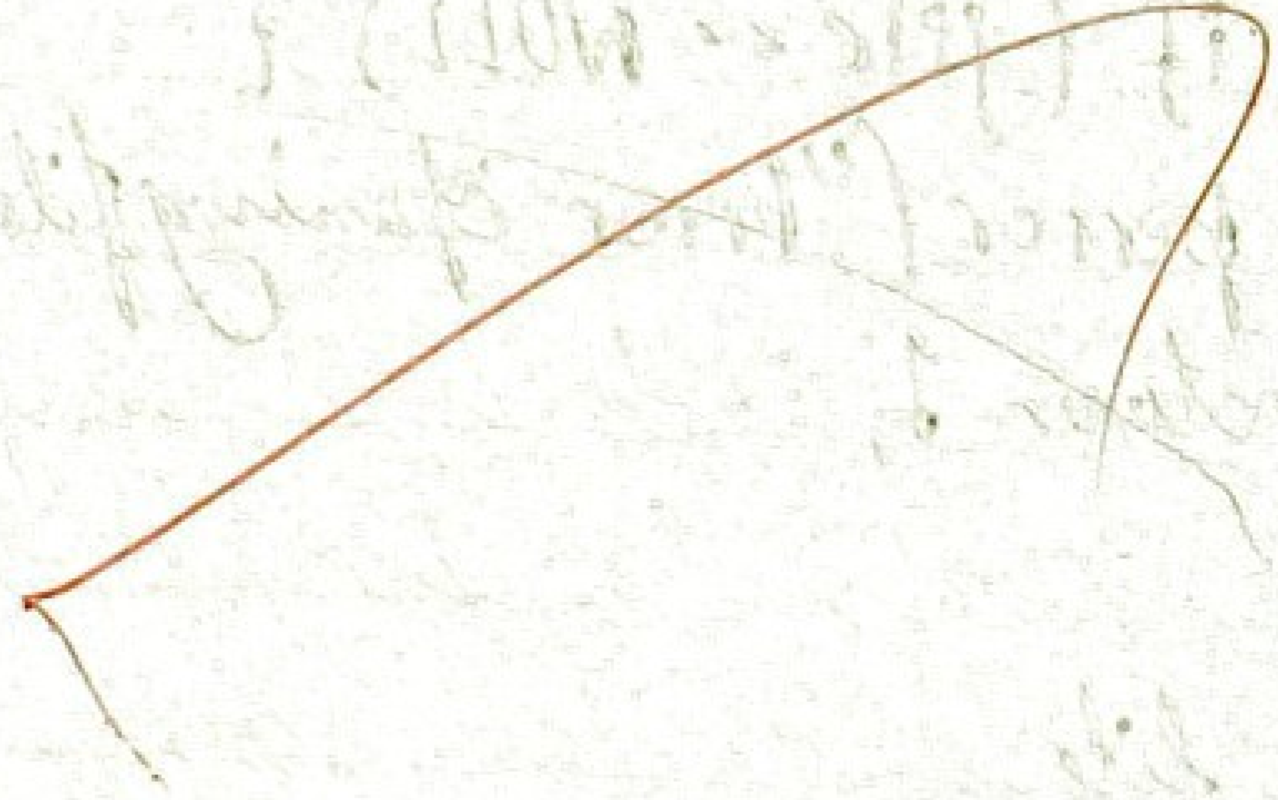
C file (example.c) with content

```
#include <stdio.h>
int main () {
  printf("Hello, world !\n");
  return 0;
```

Output

vowels : 7

consonants : 10



POORNIMA

Experiment-7

Object: Write a lex program to count no. of vowels and constants in a C file.

About the program: The lex program counts the number of vowels and constnants in a C file. It outputs a count of vowels and consonants in the file.

Procedure:

1) Install lex:
2) Write the lex program
3) Generate C code from lex
4) Compile the C program
5) Run the Program: Execute the compiled program with a C file as input to count the vowels and consonants.

Program

```
%{
#include <stdio.h>
#include <ctype.h>
int vowel_count=0;
int consonent_count=0;
%}
%%
[aAeEiIoOuU] {vowel_count++;}
[b-df-hj-np-tv-zB-DF-HJ-NP-TV-Z] {consonent_count++;}
%%
```

POORNIMA

```
int main (int args, char * argv []) {
if (argc != 2) {
  printf ("Usage: %s <filename> \n", argv [0]);
  return 1;
}
File * file = fopen [argv [1], "r");
 if (file == NULL) {
 perror ("Error opening file");
 return 1;
}
yyin = file
yylex ();
printf ("vowels : %d \n", vowel_count);
printf ("consonants : %d \n", consonant_count);
fclose (file);
 return 0;
}
```

3/12/24

Input:

8947900053
https://example.com
- valid identifier 123
15/08/2024
12:45:03

Output:

Valid Mobile Number: 8947900053
Valid URL: https://example.com
Valid Identifier: valid identifier 123
Valid Date: 15/08/2024
Valid Time: 12:45:03

POORNIMA

Experiment 8

Objective: Write a LEX program to identify following.
1) Valid mobile number
2) Valid URL
3) Valid Identifiers
4) Valid date (dd/mm/yyyy)
5) Valid time (hh:mm:ss)

About the program:
1) Valid mobile: A 10-digit number starting using digits 7, 8 or 9.
2) Valid URL: starts with http:// or https:// followed by alphanumeric characters, optional periods, slashes and extensions.
3) Valid identifier: Start with a letter or underscore and followed by letters, digits or underscore.

Procedure:
1) Start.
2) Define rules for
- Mobile numbers.
- URLs
- Identifiers
- dates
- times.

POORNIMA

Program

```
%{
#include <stdio.h>
%}
%%
[7-9][0-9]{9} { printf("Valid mobile Number: %s \n", yytext); }
(http|https)://[a-zA-Z0-9.]+ { printf("Valid URL: %s \n", yytext); }
[a-zA-Z][a-zA-Z0-9]* { printf("Valid Identifier : %s \n", yytext); }
(0[1-9]|[12][0-9]|3[01])/(0[1-9]|1[0-2])/[0-9]{4}
        { printf("Valid Date : %s\n", yytext); }
([01][0-9]|2[0-3]):[0-5][0-9]:[0-5][0-9]
        { printf ("Valid Time : %s \n", yytext); }
%%
int main () {
    printf ("Enter text to validate : \n");
  yylex();
  return 0;
}
```

3/12/2

Input : ⟶
Number of productions : 3
Productions :

E → TR

T → FS

F → a

Output ⟶

FIRST(E) = {a}

FIRST(T) = {a}

FIRST(F) = {a}

POORNIMA

Experiment-9

Objective: Write a program to find first from a given grammar.

About the program: The first set of a grammar helps to determine which terminal symbols can appear at the beginning of strings derived from the non-terminal

Algorithm:

1) Input the grammer.
   - Read the grammer rules, which consist of non terminals and productions.
2) Initialize first set:
   - Create empty FIRST set for each non terminal.

Program

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX 10
char productions [MAX][MAX], first [MAX][MAX];
int n;
void findfirst (char symbol, char result []);
void addToResultSet (char result [], char value []);
int main () {
    int i;
    char result [MAX];
```

POORNIMA

```
printf("Enter the number of productions");
scanf("%d", &n);
printf("Enter the productions (E->AB/a, use / for multiple
     proaction :)\n");
for (i=0; i<n; i++) {
    scanf("%s", production[i]);
}

for(i=0; i<n; i++) {
char non-terminal = production[i][0];
find first (non-terminal, result);
 strcpy (first [i], result);
}
printf(" \n FIRST, ids. \n");
for(i=0; i<n; i++) {
 printf("FIRST (%c) = {%s}\n", production[i])
            [0], first[i]);
  }
 return 0;
}
void findfirst (char symbol, char result []) {
   int i, j, foundEpsilon;
   char, subResult [MAX];
   result [0] = '\0';
if (! isupper (symbol)) {
   addToResultSet (result, symbol);
  return ;}
```

```
for(i=0; i<n; i++){
if(production [i][0]==symbol){
  char current = production [i][j];
  foundEpsilon = 0;
  for (j] = 3; production (i][j]!='\0'; j++){
    char current = production (i][j];
    if (current = '|'){
       continue;
    }

  find_first (current,subresult);
  strcat (result,subresult);
  if(strchr (subresult, 'ε')){
     foundEpsilon = 1;
  } else {
  found Epsilon = 0;
  break;}}
  if(found Epsilon ){
     addToResultSet(result, 'ε');
}}}}

void AddToResult(char result[], char value) {
     if (strchr (result,value) ==NULL) {
        int len=strlen (result);
        result (len]=value;
        result (len+1]='\0';}
}
```

POORNIMA

Experiment-10

Objective: Write a YACC program to recognize string aaab, abbb, using a^n b^n where n >= 0

Lex file:

```
%{
#include "y.tab.h"
%}.
%%
a return 'a';
b return 'b';
\n return 0;
return yytext[0];
%%
```

YACC file:

```
%{
#include <stdio.h>
#include <stdlib.h>
int yylex();
void error (Const char *s);
%}
%token a, b
%%
start:
sequence '\n' { printf ("valid string : Matches a^n b^n \n"); }
  |'\n' { printf ("Empty \n"); }
```

POORNIMA

```
sequence:
    'a' sequence 'b' { /* Ensure matching a^n = b^n */}
    | { /* empty */
;
%%

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

int main() {
    printf("Enter a string (a^n b^n, n>=0): ");
    yyparse();
    return 0;
}
```

POORNIMA

Experiment-11

Objective: Write a YACC program to check validity of a strings abcd, aabbcd using grammer a^nb^nc^md^m, where n,m>0

```
%{
/* definition section */
#include "y.tab.h"
%}
/*rule section*/
%%
[aA] {return A;}
[bB] {return B;}
\n {return NL;}
. {return yytext[0];}
%%
int yywrap() {
    return 1;
}
```

/* parser source code */

```
%{
   /* definition section */
#include <stdio.h>
#include <stdlib.h>
%}
%token A B NL
```

POORNIMA

```
/*Rule section */
%%
stmt : AAAAABNL { printf ("valid string \n"};
         exit(0); }
;
S : SA
|
;
%%
int yyerror (char *msg) {
      printf ("Invalid string \n");
      exit(0);
}
//driver code
main () {
  printf ("Enter the string \n");
  yyparse();
}
```

3/12/24