Aishwarya Shukla

Roll no. 324002

Gr no. 22010492

Batch: D1

**Assignment no. 03**

Problem Statement: Write a program using LEX Tool, to implement a lexical analyzer for parts of speech for C programming language and English language with and without Symbol table.

Objective:

1. Students will learn Lex tool and how to implement it with and without symbol table

Theory:

A symbol table is a global data structure that can be used in all stages/phases/passes of a compiler. This means that it can be used/accessed from both the lex and yacc generated components.

It is conventional to access the symbol table entry from the lexical analyser when it finds a token that would be stored in the table, such as an identifier, it can find the entry and update it with information only available to the lexer like line number and character position and it can also store the lexeme value if it is not already there. The symbol table pointer can now be returned in the lval of the token.

3a. For parts of speech for subset of ENGLISH language without using SYMBOL TABLE

**Code:**

%option noyywrap

%{

/\* 3a. Lex program for parts of speech in English \*/

%}

%%

[\t]+

is |

are |

am |

were |

was |

be |

being |

been |

do |

does |

did |

will |

would |

should |

can |

could |

has |

have |

had |

go {printf("%s: is a verb\n", yytext);}

car |

house |

man |

state |

ocean |

country |

city {printf("%s: is a noun\n", yytext);}

she |

he |

we |

they |

it {printf("%s: is pronoun\n", yytext);}

pretty |

old |

blue |

smart {printf("%s: is adjective", yytext);}

abnormally |

accidentally |

actually |

beatuiful |

bitterly |

brightly |

calmly {printf("%s: is adverb", yytext);}

by |

with |

about |

until {printf("%s: is preposition", yytext);}

and |

but |

or |

while |

because {printf("%s: is conjunction", yytext);}

Oh! |

Wow! |

Oops! |

Alas! {printf("%s: is interjection", yytext);}

[a-zA-Z]+ {printf("%s: is not a verb\n", yytext);}

.|\n {ECHO;}

%%

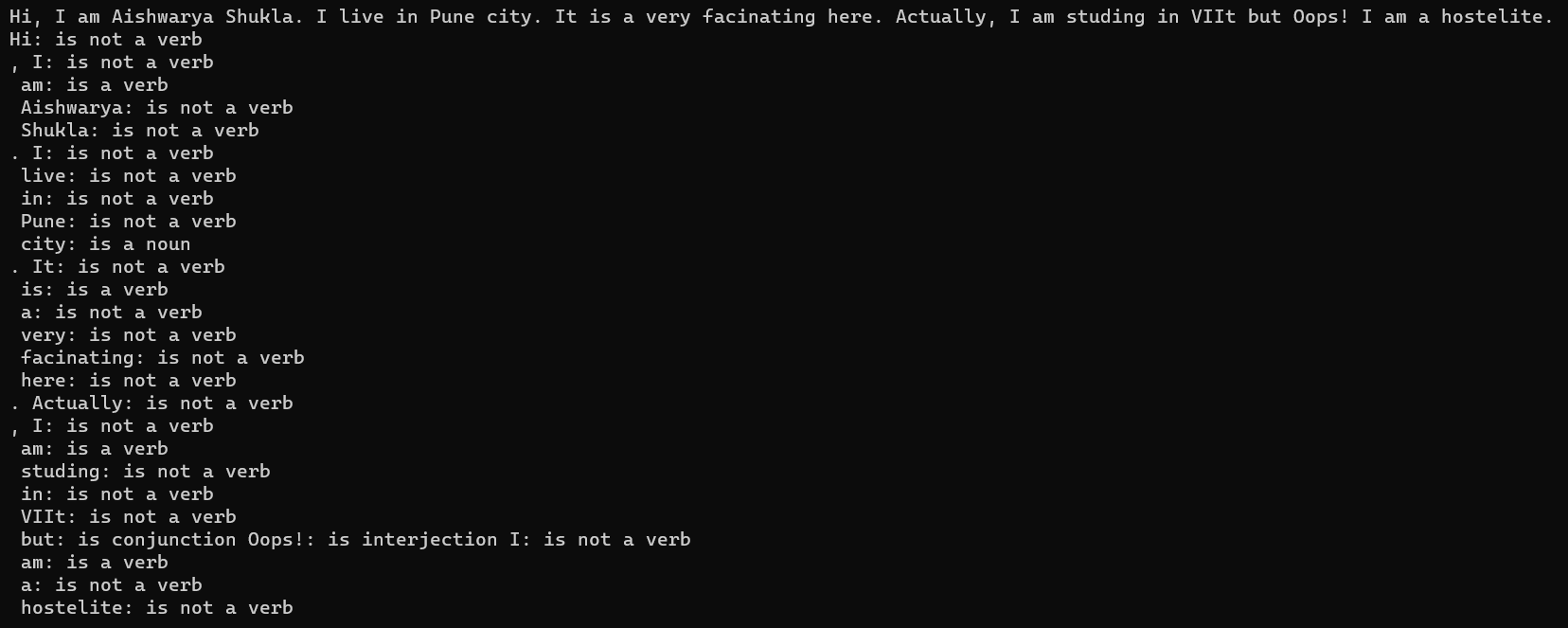
int main()

{

yylex();

}

**Output:**



3b. For parts of speech for subset of ENGLISH language with SYMBOL TABLE

**Code:**

%option noyywrap

%{

#include <string.h>

/\*

3b. Lex code to recognise parts of speech in English using Symbol Table

\*/

enum { /\*We define an enum in order to use in our table to record the types of indi-

vidual words, and to declare a variable state.\*/

LOOKUP = 0,/\*default - looking rather than defining\*/

VERB,

ADJ,

ADV,

NOUN,

PREP,

PRON,

CONJ

};

int state;

int add\_word(int type, char \*word);

int lookup\_word(char \*word);

%}

%%

/\* end of line, return to default state\*/

\n {state = LOOKUP;}

/\* whenever a line starts with a reserved part of speech name \*/

/\* start defining words of that type \*/

^verb {state = VERB;}

^adj {state = ADJ;}

^adv {state = ADV;}

^noun {state = NOUN;}

^prep {state = PREP;}

^pron {state = PRON;}

^conj {state = CONJ;}

/\* a normal word, define it or look it up \*/

[a-zA-Z]+ {

if(state != LOOKUP) {

/\* define the current word \*/

add\_word(state, yytext);

} else {

switch(lookup\_word(yytext)) {

case VERB: printf("%s: verb\n", yytext); break;

case ADJ: printf("%s: adjective\n", yytext); break;

case ADV: printf("%s: adverb\n", yytext); break;

case NOUN: printf("%s: noun\n", yytext); break;

case PREP: printf("%s: preposition\n", yytext); break;

case PRON: printf("%s: pronoun\n", yytext); break;

case CONJ: printf("%s: conjunction\n", yytext); break;

default:

printf("%s: don't recognise\n", yytext);

break;

}

}

}

. /\* ignore anything else \*/;

%%

int main()

{

yylex();

}

/\* define a linked list of words and types \*/

struct word

{

char \*word\_name;

int word\_type;

struct word \*next;

};

struct word \*word\_list; /\*first element in word list\*/

extern void \*malloc();

int

add\_word(int type, char \*word)

{

struct word \*wp;

if (lookup\_word(word) != LOOKUP)

{

printf("Warning: word %s is already defined. \n", word);

return 0;

}

/\* word not there, allocate a new entry and link it on the list\*/

wp = (struct word \*) malloc(sizeof(struct word));

wp -> next = word\_list;

/\* have to copy the word itself as well \*/

wp -> word\_name = (char \*) malloc(strlen(word) + 1);

strcpy(wp -> word\_name, word);

wp->word\_type = type;

word\_list = wp;

return 1;

}

int lookup\_word(char \* word)

{

struct word \*wp = word\_list;

/\* search dowm the kist looking for the word \*/

for (; wp; wp = wp -> next)

{

if (strcmp(wp -> word\_name, word) == 0)

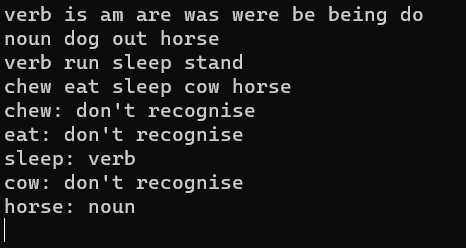
return wp -> word\_type;

}

return LOOKUP;

}

**Output:**



3c. Write lexical analyzer for the subset of C programing language

**Code:**

%{

/\* 3c. Write lexical analyzer for the subset of C programing language \*/

#include<stdio.h>

%}

DIGIT[0-9]

NUMBER {DIGIT}\*

REAL {DIGIT}\*[.]{NUMBER}

TEXT [A-Za-z]

KEYWORDS "auto"|"break"|"enum"|"struct"|"typedef"

DATATYPE "int"|"float"|"double"|"long"|"void"|"unsigned char"

CONDTIONAL "if"|"else"|"else if"|"switch"|"case"

ITERATIVE "for"|"while"|"do"

SEMICOLAN ";"

IDENTIFIER {TEXT}({DIGIT}|{TEXT}|"\_")\*

NONIDENTIFIER {DIGIT}({TEXT}|{DIGIT}|"\_")\*

ARITHMATIC\_OP "+"|"-"|"/"|"%"|"\*";

LOGICAL\_OP "&&"|"||"|"!"|"!="

RELEATIONAL\_OP "<"|">"|"<="|">="|"=="

UNARY "++"|"--"

FUNCTION {ACCESS}{DATATYPE}{IDENTIFER}"("({DATATYPE}{IDENTIFIER})\*")"

%%

[ \n\t]+ ;

{CONDTIONAL} {printf("%s\t==> CONDTIONAL \n",yytext);}

{ITERATIVE} {printf("%s\t==> ITERATIVE \n",yytext);}

{DATATYPE} {printf("%s\t==> DATATYPE\n",yytext);}

{KEYWORDS} {printf("%s\t==> KEYWORDS \n",yytext);}

{IDENTIFIER} {printf("%s\t==> IDENTIFIER \n",yytext);}

{REAL} {printf("%s\t==> REAL \n",yytext);}

{NUMBER} {printf("%s\t==> NUMBER \n",yytext);}

{NONIDENTIFIER} {printf("%S\t==> NONIDENTIFIER \n",yytext);}

{SEMICOLAN} {printf("%s\t==> SEMICOLAN \n",yytext);}

{UNARY} {printf("%s\t==> UNARY \n",yytext);}

{ARITHMATIC\_OP} {printf("%s\t==> ARITHMATIC\_OP \n",yytext);}

{LOGICAL\_OP} {printf("%s\t==> LOGICAL OP \n",yytext);}

{RELEATIONAL\_OP} {printf("%s\t==> RELEATIONAL OP \n",yytext);}

"=" {printf("%s\t==> ASSIGNMENT OP \n",yytext);}

"{" {printf("%s\t==> BLOCK BEGIN \n",yytext);}

"}" {printf("%s\t==> BLOCK END \n",yytext);}

"(" {printf("%s\t==> PARATHRSIS BEGIN \n",yytext);}

")" {printf("%s\t==> PARATHRSIS END \n",yytext);}

. ;

%%

int yywrap(void){}

main ( )

{

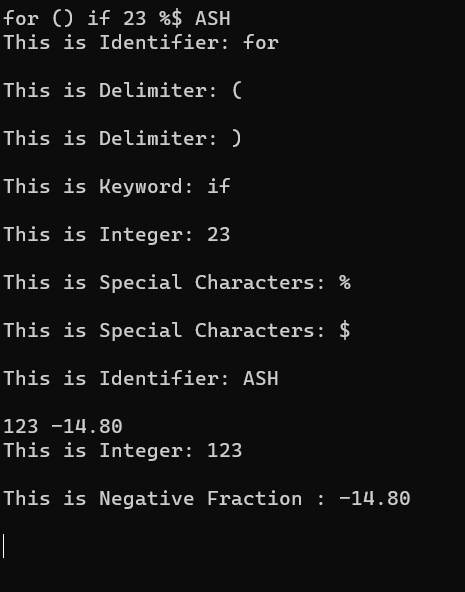
printf("Enter a string: ");

yylex();

return 0;

}

**Output:**



3d. Write lexical analyzer for the subset of C programing language with Symbol table

**Code:**

%{

/\* 3d. Write lexical analyzer for the subset of C programing language with Symbol table \*/

#include<stdio.h>

#include <string.h>

enum {

LOOKUP = 0, /\* default - looking rather than defining. \*/

TEXT,

KEYWORD,

DATATYPE,

CONDTIONAL,

ITERATIVE,

ARITHMATIC\_OP

};

int state;

int add\_word(int type, char \*word) ;

int lookup\_word(char \*word) ;

%}

%%

/\* end of line, return to default state \*/

\n { state = LOOKUP; }

/\* whenever a line starts with a reserved part of speech name \*/

/\* start defining words of that type \*/

^text { state =TEXT; }

^keyword { state = KEYWORD; }

^datatype { state = DATATYPE; }

^condtional { state = CONDTIONAL; }

^iterative { state = ITERATIVE; }

^arithmatic\_op { state = ARITHMATIC\_OP; }

/\* a normal word, define it or look it up \*/

[a-zA-Z]+ {

if (state!=LOOKUP) {

add\_word(state, yytext); /\* define the current word \*/

}

else

{

switch ( lookup\_word(yytext) )

{

case TEXT: printf("%s: text \n", yytext); break;

case KEYWORD: printf("%s: keyword \n", yytext); break;

case DATATYPE: printf("%s: datatype\n", yytext); break;

case CONDTIONAL: printf("%s:condtional\n", yytext); break;

case ITERATIVE: printf("%s:iterative \n", yytext); break;

case ARITHMATIC\_OP: printf("%s: arithmatic\_op \n", yytext); break;

default:

printf ( "%s: Don't recognize\n" , yytext) ;

break;

}

}

}

. /\* ignore anything else \*/ ;

%%

int yywrap(void){}

int main( )

{

printf("Enter a string: \n");

printf("\n");

yylex();

return 0;

}

/\* define a linked list of words and types \*/

struct word {

char \*word\_name;

int word\_type;

struct word \*next;

};

struct word \*word\_list; /\* first element in word list \*/

extern void \*malloc() ;

int

add\_word(int type, char \*word)

{

struct word \*wp;

if(lookup\_word(word) != LOOKUP) {

printf("!!! warning: word %s already defined \n", word);

return 0;

}

/\* word not there, allocate a new entry and link it on the list \*/

wp = (struct word \*) malloc(sizeof(struct word) ) ;

wp->next=word\_list;

/\* have to copy the word itself as well \*/

wp->word\_name = (char \*)malloc(strlen(word)+1);

strcpy (wp->word\_name, word);

wp->word\_type = type;

word\_list = wp;

return 1; /\* itworked \*/

}

int

lookup\_word (char \*word)

{

struct word \*wp = word\_list;

/\* search down the l i s t looking for the word \*/

for(; wp; wp = wp->next) {

if(strcmp(wp->word\_name, word) == 0)

return wp->word\_type;

}

return LOOKUP; /\* not found \*/

}

**Output:**

