Nikit Gokhe

Class : TY-Comp D1

Roll No. 324022

GR No. 21810522

# Assignment 03

# Aim:

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

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

**3.c** -- Write Lexical Analyser without using SYMBOL TABLE for subset of ‘C’ programming language

**3.d** -- Write Lexical Analyser with SYMBOL TABLE for subset of ‘C’ programming language

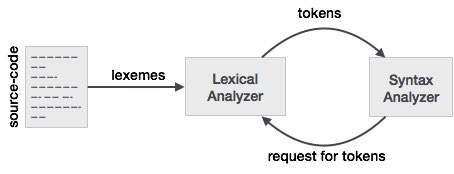
# 

# Theory:

**Lexical Analyser**

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyser breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyser finds a token invalid, it generates an error. The lexical analyser works closely with the syntax analyser. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyser when it demands.



**Language**

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

**Longest Match Rule**

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

For example:

int intvalue;

While scanning both lexemes till ‘int’, the lexical analyzer cannot determine whether it is a keyword int or the initials of identifier int value.

The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

The lexical analyzer also follows rule priority where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

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

# Source Code:

# Output:

%{

%}

%%

is |

am |

are |

were |

was |

be |

being |

been |

do |

does |

did |

will |

would |

should |

can |

could |

walk |

walking |

has |

have |

had |

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

to |

from |

behind |

above |

below |

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

if |

then |

and |

but |

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

their |

my |

My |

bad |

your |

his |

her |

its {printf("%s: is an adjective\n", yytext);}

I |

you |

he |

she |

we |

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

very |

simply |

gently |

quietly |

camly |

slowly |

quickly |

angrily {printf("%s: is an adverb\n", yytext);}

a |

an |

the |

The {printf("%s: is an article\n", yytext);}

[a-zA-Z]+ { printf("%s: don't recognize , might be a noun\n", yytext); }

.|\n {ECHO; /\* normal default anyway \*/ }

%%

int yywrap(void) {

return 1;

}

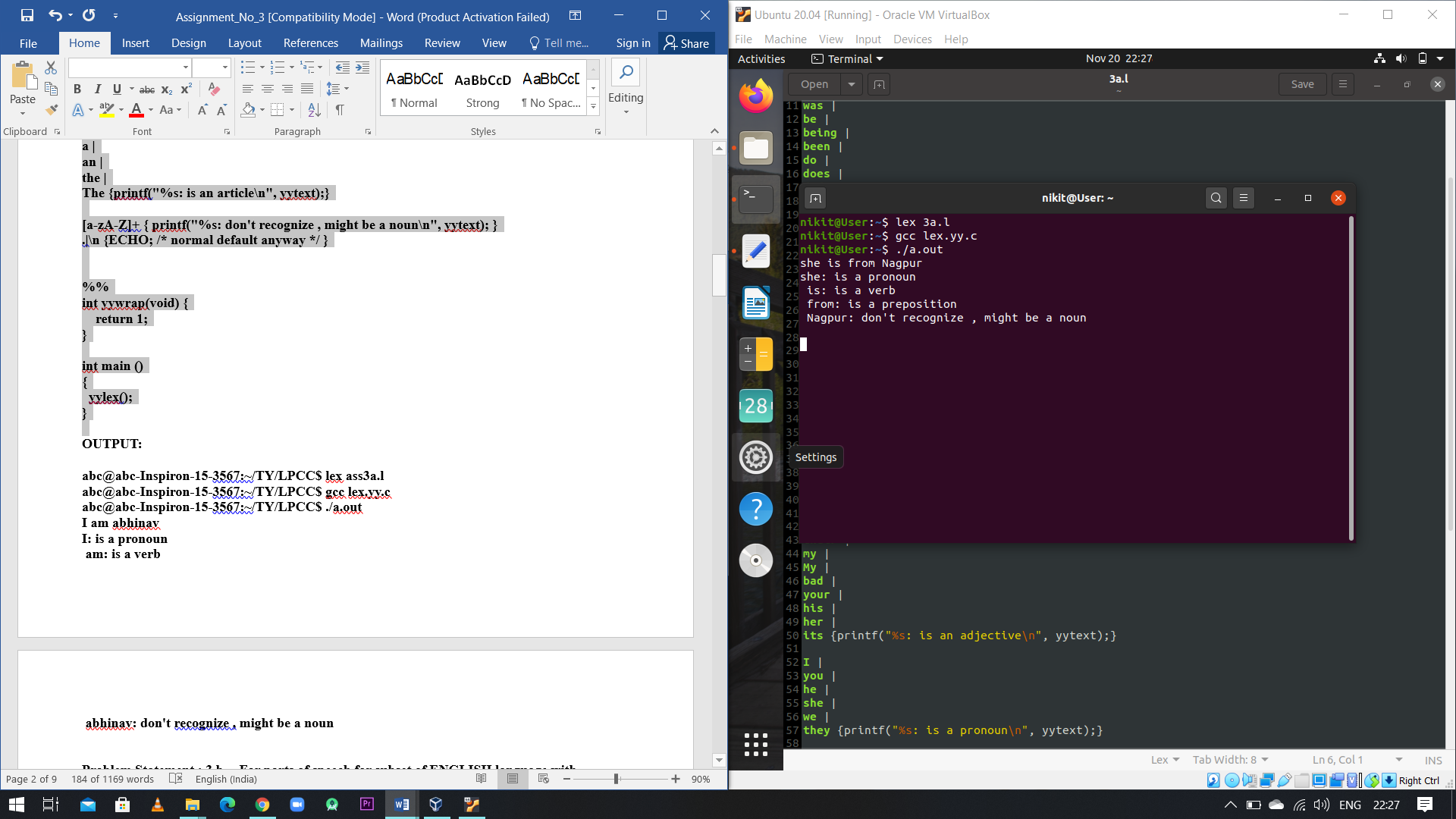
int main ()

{

yylex();

}

# Output:



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

# Source Code:

%{

enum{

LOOKUP = 0 ,

VERB,

ADJ,

ADV,

NOUN,

PREP,

PRON,

CONJ

};

int state ;

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

int look\_up(char \*word);

%}

%%

\n { state = LOOKUP ;}

^verb { state = VERB ;}

^adj { state = ADJ ;}

^adv { state = ADV ;}

^noun { state = NOUN ;}

^prep { state = PREP ;}

^pron { state = PRON ;}

^conj { state = CONJ ;}

[a-zA-Z]+ {

if(state != LOOKUP){

add\_word(state , yytext);

}

else{

printf("---------------------------------------\n");

switch(look\_up(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 recognize\n",yytext);break;

}

printf("\n");

}

}

. ;

%%

int yywrap(void) {

return 1;

}

#include<string.h>

int main()

{ printf("USE:\n VERB for verb\n ADJ for adverb\n ADV for adverb\n NOUN for noun \nPREP for prepositon\n PRON for pronoun\nCONJ for conjunction\n");

yylex();

}

struct word {

char \*word\_name;

int word\_type ;

struct word \*next ;

};

struct word \*word\_list ;

extern void \*malloc();

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

struct word \*wp;

if(look\_up(word)!= LOOKUP)

{

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

return 0 ;

}

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

wp->next = word\_list;

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

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

wp->word\_type=type;

word\_list = wp;

return 1;

}

int look\_up(char \*word){

struct word \*wp = word\_list ;

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

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

return wp->word\_type ;

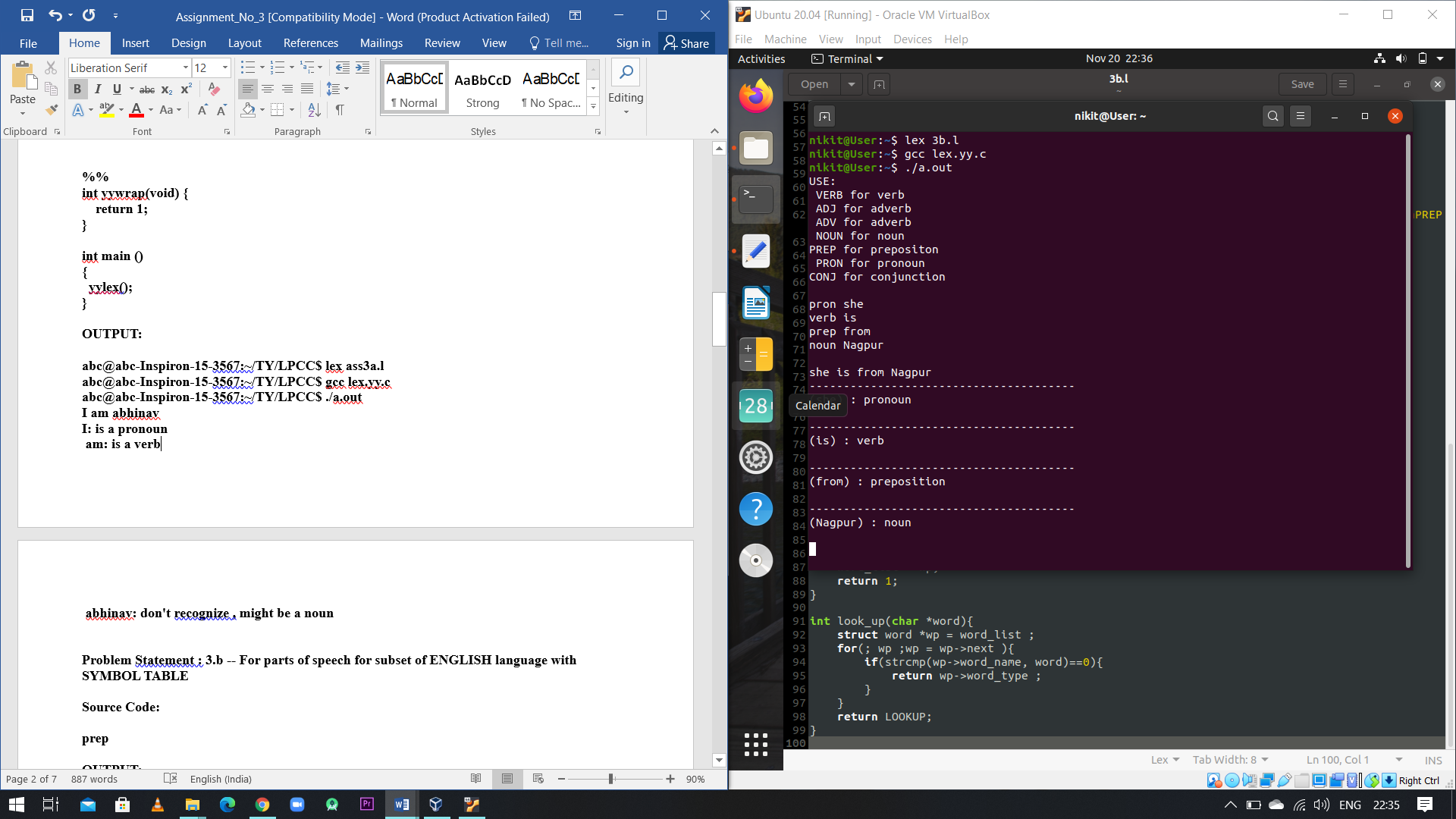
}

}

return LOOKUP;

}

# Output:



# 3.c Write Lexical Analyser without using SYMBOL TABLE for subset of ‘C’ programming language

# Source Code:

%{

FILE \*fr , \*fw;

%}

%%

[a-z] {fprintf(fw, "(%s: VARIABLE)",yytext);}

[0-9] {fprintf(fw , "(%s :DIGIT)",yytext);}

if |

while |

else |

main |

return |

for |

elseif |

printf |

scanf |

include {fprintf(fw , "(%s :KEYWORD)",yytext);}

"+" |

"-" |

"\*" |

"/" |

"%" |

"&&" |

"||" |

"<" |

">" |

"<=" |

">=" |

"==" {fprintf(fw , "(%s :OPERATOR)",yytext);}

"%d" |

"%s" |

"%c" |

"%f" |

"%ls" {fprintf(fw , "(%s :FORMAT SPECIFIER)",yytext);}

"#" {fprintf(fw , "(%s :PREPROCESSING DIRECTIVE)",yytext);}

int |

float |

char |

string {fprintf(fw , "(%s :DATA TYPE)",yytext);}

%%

#include<stdlib.h>

int main(int argc,char\* argv[])

{

fr = fopen("cprogram.c","r");

fw = fopen("Outfile.txt","w");

yyin=fr;

yyout=fw;

yylex();

return 0 ;

}

int yywrap()

{

return 1;

}

# 3.d Write Lexical Analyser with SYMBOL TABLE for subset of ‘C’ programming language

# Source Code:

%{

enum{

LOOKUP = 0 ,

DIGIT,

DTYPE,

KEY,

OPER,

FSPECI,

PREDIR

};

int state ;

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

int look\_up(char \*word);

%}

%%

\n { state = LOOKUP ;}

^digit { state = DIGIT ;}

^dtype { state = DTYPE ;}

^key { state = KEY ;}

^oper { state = OPER ;}

^fspeci { state = FSPECI ;}

^predir { state = PREDIR ;}

[a-zA-Z]+ {

if(state != LOOKUP){

add\_word(state , yytext);

}

else{

printf("---------------------------------------\n");

switch(look\_up(yytext)){

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

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

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

case OPER : printf("(%s) : OPERATOR \n", yytext);break;

case FSPECI : printf("(%s) : FORMAT SPECIFIER \n", yytext);break;

case PREDIR : printf("(%s) : PREPROCESSING DIRECTIVE \n", yytext);break;

default :

printf("(%s) : don't recognize, might be VARIABLE\n",yytext);break;

}

printf("\n");

}

}

. ;

%%

#include<string.h>

int main()

{ printf("USE:\n DIGIT for digits\n DTYPE for datatype\n KEY for keywords\n OPER for operators \nFSPECI for format specifier\n PREDIR for preprocessing directive\n");

yylex();

}

struct word {

char \*word\_name;

int word\_type ;

struct word \*next ;

};

struct word \*word\_list ;

extern void \*malloc();

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

struct word \*wp;

if(look\_up(word)!= LOOKUP)

{

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

return 0 ;

}

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

wp->next = word\_list;

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

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

wp->word\_type=type;

word\_list = wp;

return 1;

}

int look\_up(char \*word){

struct word \*wp = word\_list ;

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

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

return wp->word\_type ;

}

}

return LOOKUP;

}

int yywrap()

{

return 1;

}

# Output:

