



**ALLIANCE**  
**UNIVERSITY**  
*Private University established in Karnataka State by Act No.34 of year 2010*  
*Recognized by the University Grants Commission (UGC), New Delhi*  
**Alliance College of Engineering and Design**

**Department of Computer Science and Engineering**

**CSL 605 – COMPILER DESIGN LAB**

**This Is To Certify Mr MUMUKSH NAYAK Register No 19030141CSE007 Of VI Semester CSE Has Completed The CSL 605 Compiler Design Lab Record For The Even Semester Of The Academic Year 2021-22.**

**Faculty Signature**

## COMPILER DESIGN LAB RECORD

S.NO	Date	Experiment Name	Page No
1	30-12-2021	Token Separation	3-4
2	22-02-2022	Symbol Table Generation	5-6
3	17-02-2022	Simulation Of Deterministic Finite Automaton	7-9
4	26-02-2022	Predictive Parser	10-13
5	10-03-2022	Token Separation Using LEX	14-16
6	17-03-2022	Calculator Using Yacc	17-20
7	24-03-2022	Shift reduce parser	21-25
8	12-05-2022	NLTK Parsing	25-27
9	12-05-2022	INTERMEDIATE CODE GENERATION	28-30
10	12-05-2022	CODE GENERATION	31-33

### Experiment No.1

#### TOKEN SEPARATION

**OBJECTIVE :** To write a program which separates all the lexemes and groups under the corresponding category.

**DESCRIPTION-:** Scanning is the first phase of a compiler in which the source program is read character by character and then grouped into various tokens. Token is defined as sequence of characters with collective meaning. The various tokens could be identifiers, keywords, operators, punctuations, constants, etc. The input is a program written in any high level language and the output is stream of tokens. Regular expressions can be used for implementing this token separation.

During the Lexical analysis phase of the compiler, the source program will be read from left to right and get grouped into TOKENS. Tokens can be any one of the following-:

- 1.Keywords (if, while, do etc.)
- 2.identifiers (num, a, b, c etc.)
- 3.Punctuation Symbols (:,'',.,; etc.)
- 4.operators (+, -, /, \*, >,<,<= etc.)

The Keywords, Operators and Punctuation symbols should be declared and defined in the character array. When the input expression is given each character should be checked and Keywords, Operators, Punctuation symbols, Identifiers should be displayed separately in a table format.

#### **ALGORITHM-:**

1. Start.
2. Get an expression as input.
3. Separate each and every token.
4. Check whether they are of keyword or identifiers or punctuations or symbols.
5. If so save it with necessary information and also with unique identifiers.
6. If the token is already entered in the table avoid repeated entry.
7. Else save it with error information.
8. At the end print the tokens and the category to which they belong in a table format.
9. Stop.

#### **PROGRAM-:**

```
import re
tokens = []
input_code = 'if ( a > b ) { i = j + 2 ; else j = k - 2 ; }'.split()
for word in input_code:
    if word in ["str", "int", "bool", "float", "double", "char", "long"]:
        tokens.append(['DATATYPE', word])

if word
in["auto","break","case","catch","word","class","const","
continue",

"delete","do","if","else","enum","false","for","goto","if","#include","names
pace","not","or","private","protected","public","return","short","signed",
```

## COMPILER DESIGN LAB RECORD

```
"sizeof","static","struct","switch","true","try","unsigned","void",
"while",]:
    tokens.append(['KEYWORD', word])
elif re.match("[a-z]", word) or re.match("[A-Z]", word):
    tokens.append(['IDENTIFIER', word])
elif word in ['_', '~', '!', '@', '#', '$', '^', '&', '"', ':', ';', '<', '>', '?']:
    tokens.append(['NON-IDENTIFIER', word])
elif word in ["+", "-", "*", "/", "%", "+=", "-=", "*=", "/=", "++", "--", "|", "&&"]:
    tokens.append(['OPERATOR', word])
elif word in
    ["\t", "\n", "(", ")", "[", "]", "{", "}", "=", ":", ";;", "<<", ">>"]:
        tokens.append(['DELIMITER', word])
    elif word in ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']:
        tokens.append(['NUMERAL', word]) elif
        re.match("[0-9]", word):
            if word[len(word) - 1] == ';':
                tokens.append(['INTEGER', word[:-1]])
                tokens.append(['END_STATEMENT', ';']) else:
                    tokens.append(['INTEGER', word]) for tkn in
tokens:
    print(tkn)
```

## OUTPUT

```
In [1]: runfile('B:/WAREHOUSE/IMP/6th SEM/CDLab/Lexer/Lexer.py', wdir='B:/WAREHOUSE/IMP/6th SEM/CDLab/
Lexer')
['KEYWORD', 'if']
['DELIMITER', '(']
['IDENTIFIER', 'a']
['NON-IDENTIFIER', '>']
['IDENTIFIER', 'b']
['DELIMITER', ')']
['DELIMITER', '{']
['IDENTIFIER', 'i']
['DELIMITER', '=']
['IDENTIFIER', 'j']
['OPERATOR', '+']
['NUMERAL', '2']
['NON-IDENTIFIER', ';']
['KEYWORD', 'else']
['IDENTIFIER', 'j']
['DELIMITER', '=']
['IDENTIFIER', 'k']
['OPERATOR', '-']
['NUMERAL', '2']
['NON-IDENTIFIER', ';']
['DELIMITER', '}']
```

## COMPILER DESIGN LAB RECORD

### Experiment No.2

#### SYMBOL TABLE GENERATION

**AIM:-** Write a program to scan the input program and generate a symbol table.

**DESCRIPTION:-** Symbol table is a data structure containing a record for each identifier with fields for the attributes of the identifier. The attributes may give the information about the Storage allocated for the identifier, Data type and Scope of the identifier. By using these records each of the identifiers can be retrieved easily and quickly. During the Lexical Analysis the identifiers will be found and stored into the Symbol table but the attributes will not be stored in lexical phase. The Compiler and the Assembler will be having their own Symbol tables.

The expression will be given as the input and the output obtained will be containing the table of information about the identifiers.

#### **ALGORITHM:-**

1. Start.
2. Get an expression as input.
3. Separate each and every token.
4. If the token is id then create an entry in symbol table and enter the identifier name and its corresponding data type.
5. Check the data type of each identifier and assign the corresponding memory location.
6. Check whether the identifier is having a value, if so enter that in the symbol table for that identifier.
7. If the token is already entered in the table, avoid repeated entry.
8. Stop.

#### **PROGRAM-**

```
tokens=[] iden
= []

key_words = ["int", "str", "char", "float", "double", "bool", "long", "short", "do", "if", "else"]

operators = ["+", "-", "*", "/", "=", "<", ">", "%", "+=", "-=", "*=", "/=", "++", "--", "|", "&&"]

punct = ["(", ")", "{", "}", "[", "]", ",", ";", ":"]

with open(r"C:\Users\mumux\Documents\Python\LEXER\sym.txt ") as t:
    reader=t.readlines()

for t in reader:
    tokens=tokens + (t.split(" "))

print("ID          Data_Types          Value Return_Type          N_Parameter
T_Parameter")

for pos, t in enumerate(tokens): for word in
    key_words:
        if(t==word): iden.append(tokens[pos + 1])
        if(tokens[pos + 2 ] == ',,')
```

## COMPILER DESIGN LAB RECORD

```
print(tokens[pos + 1] + " " + tokens[pos] + " " + "NULL") tokens.insert(pos + 3,
tokens[pos])

elif(tokens[pos + 2] == '('):
    end=tokens.index(')')

    para = tokens[pos + 3:end]

    key_count=0;

    pt=[]

    for key in key_words:

        key_count = key_count + para.count(key)

        i=0

        while(i < para.count(key)):

            pt.append(key)

            i = i + 1

print(tokens[pos + 1] + " " + tokens[pos] + " " + str(key_count) + " " +str(pt))

elif (tokens[pos + 1] == '('):

    continue

else:

print(tokens[pos + 1] + " " + tokens[pos] + " " + tokens[pos + 3])
```

## OUTPUT

```
In [1]: runfile('B:/WAREHOUSE/IMP/6th SEM/CDLab/SymTbl/symtbl.py', wdir='B:/WAREHOUSE/IMP/6th
SEM/CDLab/SymTbl')
ID      Data_Types      Value      Return_Type      N_Parameter
T_Parameter
a          int          10;
b          float        20;
c          char         'd';
```

## Experiment No.3

### SIMULATION OF DETERMINISTIC FINITE AUTOMATON

**AIM:-** To write the program, which simulates the working of a Deterministic Finite Automata (DFA).

**DESCRIPTION:-** In Theory of Computation, a branch of theoretical computer science, a deterministic finite automaton (DFA)—also known as deterministic finite state machine—is a finite state machine that accepts/rejects finite strings of symbols and only produces a unique computation (or run) of the automaton for each input string.

A deterministic finite automaton  $M$  is a 5-tuple,  $(Q, \Sigma, \delta, q_0, F)$ , consisting of

- a finite set of states ( $Q$ )
- a finite set of input symbols called the alphabet ( $\Sigma$ )
- a transition function ( $\delta : Q \times \Sigma \rightarrow Q$ )
- an initial or start state ( $q_0 \in Q$ )
- a set of accept states ( $F \subseteq Q$ )

#### ALGORITHM:-

**Input:** An input string  $x$  terminated by an EOF character. A DFA  $D$  with start state  $S_0$  and set of accepting states  $F$ .

**Output:** The answer “yes” if DFA accepts the string; “no” otherwise.

#### Method:

1. Start.
2.  $s = s_0$
3.  $c = \text{nextchar}$
4. while  $c \neq \text{eof}$   $s = \text{move}(s, c)$ ;  $c = \text{nextchar}$ ;
5. end if  $s$  is in  $F$  then return “yes” else return “no”
6. Stop.

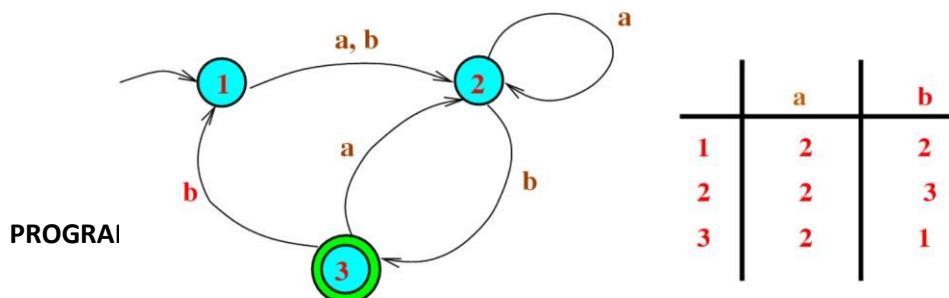


Figure 3: Transition table for a DFA.

## COMPILER DESIGN LAB RECORD

```
n_state=int(input("Enter The Number Of States: ")) state=[input("Enter The States: ") for i
in range(0,n_state)] n_string=int(input("Enter The Number Of Strings: "))
strings=[input("Enter The Strings: ") for i in range(0,n_string)] f_state=input("Enter The
Final State: ")

k=[0 for i in range(len(state))] for i in
range(len(state)):

    k[i]=[0 for j in range(len(strings))]
    for j in range(len(strings)):

        k[i][j]=input('From ' +state[i]+' If String Is 3' +strings[j] +' Then Go To: ')

def _check(x,y): table.append(k[state.index(x)][strings.index(y)])
    return k[state.index(x)][strings.index(y)]

while True:
    table=[]
    start=state[
    0]

    inp_str=input("Enter The String To Check: ") for i in
    inp_str:

        start= _check(start,i) if
    table[-1]==f_state:

        print("Given String Accepted!!") else:

        print("Given String Not Accepted!")
```



## COMPILER DESIGN LAB RECORD

### OUTPUT

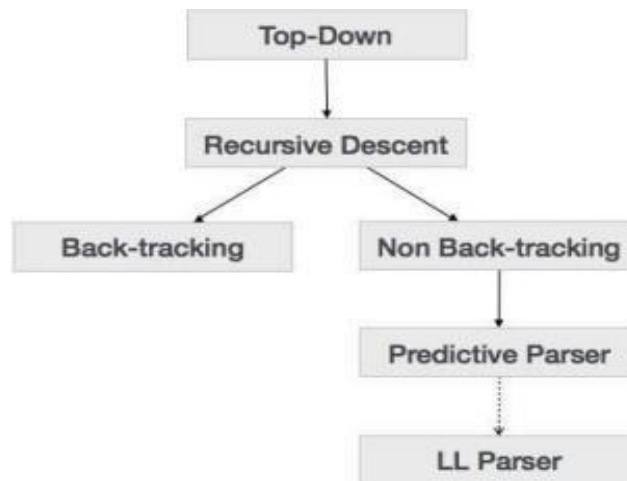
```
PS C:\Users\anshy> & python "b:/WAREHOUSE/IMP/6th SEM/CDLab/DFA/dfa.py"
Enter The Number Of States: 3
Enter The States: 1
Enter The States: 2
Enter The States: 3
Enter The Number Of Strings: 2
Enter The Strings: a
Enter The Strings: b
Enter The Final State: 3
From 1 If String Is a Then Go To: 2
From 1 If String Is b Then Go To: 2
From 2 If String Is a Then Go To: 2
From 2 If String Is b Then Go To: 3
From 3 If String Is a Then Go To: 2
From 3 If String Is b Then Go To: 1
Enter The String To Check: aaabbba
Given String Not Accepted!
Enter The String To Check: aabbaabb
Given String Not Accepted!
Enter The String To Check: aaaabbbb
Given String Accepted!!
```

### Experiment No.4

#### PREDICTIVE PARSER

**AIM-:** Write a program to parse a given string using non recursive predictive parsing for a given grammar.

**DESCRIPTION-:** Recursive Descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non- terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive descent parsing that does not require any backtracking is known as predictive parsing. This parsing technique is regarded recursive as it uses context free grammar which is recursive in nature.



#### ALGORITHM-:

1. Start.
2. set ip to point to the first symbol of w\$  
repeat 1 let X be the top stack symbol and the symbol pointed to by ip.
3. if X is a terminal or 4 then if X = a then pop x from the stack and advance ip else error
4. else if  $m[X,a] = X \rightarrow Y_1 Y_2 \dots Y_k$  then begin pop X from the stack.
5. push  $Y_k, Y_{k-1}, \dots Y_1$  on to the stack, with  $Y_1$  on top.
6. Output the production  $X \rightarrow Y_1 Y_2 \dots Y_k$  end else error until  $x = \$$
7. Stop.

## COMPILER DESIGN LAB RECORD

### PROGRAM

```
def print_iter(Stack,Input,Action,verbose=True):

    if verbose==True:

        print(".".join(Stack).ljust(25)," | ",".".join(Input).ljust(30)," | ",Action)

def predictive_parsing(sentence,parsingtable,terminals,start_state="S",verbose=True):

    status = None

    stack = [start_state,"$"]

    Inp = sentence.split(".")

    if verbose==True:

        print_iter(["Stack"],["Input"],"Action")

    print_iter(stack,Inp,verbose)

    action=[]

    while(len(sentence)>0 and status!=False):

        top_of_input = Inp[0]

        pos = top_of_input

        if stack[0] == "$" and pos == "$" :

            print_iter(stack,Inp,verbose)

            return "Accepted"

        if stack[0] == pos:

            print_iter(stack,Inp,verbose)

            del(stack[0])

            del(Inp[0])

            continue

        if stack[0]=="epsilon":
```

## COMPILER DESIGN LAB RECORD

```
print_iter(stack,Inp,verbose)

del(stack[0])

continue

try:

    production=parsingtable[stack[0]][pos]

    print_iter(stack,Inp,stack[0]+" -> "+production,verbose)

except:

    return "error for "+str(stack[0])+" on "+str(pos),"Not Accepted"

    new = production.split(".")

    stack=new+stack[1:]

return "Not Accepted"

if __name__=="__main__":

    parsingtable = {

        "E" : {"id" : "T.E1", "(" : "T.E1"},

        "E1" : {"+" : ".T.E1", ")" : "epsilon", "$" : "epsilon"},

        "T" : {"id" : "F.T1", "(" : "F.T1" },

        "T1" : {"+" : "epsilon", "*" : ".F.T1", ")" : "epsilon", "$" : "epsilon"},

        "F":{"id":"id", "(" : "(.E.)"}

    }

    terminals = ["id", "(", ")", "+", "*"]

    print(predictive_parsing(sentence="id.+id.+id.$",parsingtable=parsingtable,terminals=terminals,

    start_state="E",verbose=True))
```

## COMPILER DESIGN LAB RECORD

### OUTPUT

PS B:\WAREHOUSE\IMP\6th SEM\CDLab\PredictiveParser> & C:/Users/anshy/AppData/Local/Programs/Python/Python39/python.exe "b:\WAREHOUSE\IMP\6th SEM\CDLab\PredictiveParser/parser.py"

Matched	Stack	Input	Action
	E.\$	id.+(.id.+.id.).\$	Initial
	E.\$	id.+(.id.+.id.).\$	E -> T.E1
	T.E1.\$	id.+(.id.+.id.).\$	T -> F.T1
	F.T1.E1.\$	id.+(.id.+.id.).\$	F -> id
	id.T1.E1.\$	id.+(.id.+.id.).\$	Pop
id	T1.E1.\$	+(.id.+.id.).\$	T1 -> epsilon
id	epsilon.E1.\$	+(.id.+.id.).\$	Poping Epsilon
id	E1.\$	+(.id.+.id.).\$	E1 -> +.T.E1
id	+T.E1.\$	+(.id.+.id.).\$	Pop
id.+	T.E1.\$	(.id.+.id.).\$	T -> F.T1
id.+	F.T1.E1.\$	(.id.+.id.).\$	F -> (.E.)
id.+	(.E.).T1.E1.\$	(.id.+.id.).\$	Pop
id.+(	E.).T1.E1.\$	id.+.id.).\$	E -> T.E1
id.+(	T.E1.).T1.E1.\$	id.+.id.).\$	T -> F.T1
id.+(	F.T1.E1.).T1.E1.\$	id.+.id.).\$	F -> id
id.+(	id.T1.E1.).T1.E1.\$	id.+.id.).\$	Pop
id.+(.id	T1.E1.).T1.E1.\$	+.id.).\$	T1 -> epsilon
id.+(.id	epsilon.E1.).T1.E1.\$	+.id.).\$	Poping Epsilon
id.+(.id	E1.).T1.E1.\$	+.id.).\$	E1 -> +.T.E1
id.+(.id	+T.E1.).T1.E1.\$	+.id.).\$	Pop
id.+(.id.+	T.E1.).T1.E1.\$	id.).\$	T -> F.T1
id.+(.id.+	F.T1.E1.).T1.E1.\$	id.).\$	F -> id
id.+(.id.+	id.T1.E1.).T1.E1.\$	id.).\$	Pop
id.+(.id.+.id	T1.E1.).T1.E1.\$	).\$	T1 -> epsilon
id.+(.id.+.id	epsilon.E1.).T1.E1.\$	).\$	Poping Epsilon
id.+(.id.+.id	E1.).T1.E1.\$	).\$	E1 -> epsilon
id.+(.id.+.id	epsilon.).T1.E1.\$	).\$	Poping Epsilon
id.+(.id.+.id	).T1.E1.\$	).\$	Pop
id.+(.id.+.id.)	T1.E1.\$	\$	T1 -> epsilon
id.+(.id.+.id.)	epsilon.E1.\$	\$	Poping Epsilon
id.+(.id.+.id.)	E1.\$	\$	E1 -> epsilon
id.+(.id.+.id.)	epsilon.\$	\$	Poping Epsilon
id.+(.id.+.id.)	\$	\$	Accepted

Accepted

### Experiment No.5

#### TOKEN SEPARATION USING LEX

**AIM-:** To write a Lex Specification for token separation using regular expressions.

**DESCRIPTION-:** Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine. Lex source is a table of regular expressions and corresponding program fragments.

The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

Lex turns the user's expressions and actions (called source in this memo) into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input in this memo) and perform the specified actions for each expression as it is detected.

```

      +-----+
Source -> | Lex | -> yylex
      +-----+

      +-----+
Input  -> | yylex | -> Output
      +-----+
```

An overview of Lex  
Figure 1

The general format of Lex source is:

{definitions}

%%

{rules}

%%

{user subroutines}

#### ALGORITHM

1. Define the required variables
2. Define the regular expressions
3. Define the required auxiliary definitions

## COMPILER DESIGN LAB RECORD

4. Compile the program by using the following commands vi sample.l

a. lex sample.l ->lex.yy.c

b. cc lex.yy.c ->./a.out

5. Execute the program by the following command \$>> ./a.out

### PROGRAM

```
%{
#include<stdio.h>
%}
%%

[0-9]+[.][0-9]+ printf("%s is a floating point number\n",yytext);

int|float|char|double|void printf("%s is a datatype\n",yytext);

[0-9]+ printf("%s is an integer number\n",yytext);

[a-z]+[()] printf("%s is a function\n",yytext);

[a-z]+ printf("%s is an identifier\n",yytext);

[+*=*-/ ] printf("%s is an operator\n",yytext);

; printf("%s is an delimiter\n",yytext);

, printf("%s is a separator\n",yytext);

#[a-z\.h]+ printf("%s is a preprocessor\n",yytext);

%%

int yywrap(void)
{
return 1;
}

int main(void)
{
Yylex();
Return 0;
}
```

### OUTPUT

## COMPILER DESIGN LAB RECORD

```
anishk07@ANISHK:~$ lex samplefile1.1
anishk07@ANISHK:~$ cc lex.yy.c
anishk07@ANISHK:~$ ./a.out
if i=j+2; else j=k-2;
if is an identifier
  i is an identifier
= is an operator
j is an identifier
+ is an operator
2 is an integer number
; is an delimiter
  else is an identifier
    j is an identifier
= is an operator
k is an identifier
- is an operator
2 is an integer number
; is an delimiter
```



### Experiment No.6

#### BASIC ARITHMETIC OPERATIONS USING YACC

**AIM-:** To write a program which separates all the lexemes and groups under the corresponding category.

**DESCRIPTION-:** YACC is a computer program for the Unix operating system. It is a LALR parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to BNF. YACC is a parser generator. It is to parsers what lex is to scanners. YACC is designed for use with C code and generates a parser written in C. The parser is configured for use in conjunction with a lex- generated scanner and relies on standard shared features (token types, yylval, etc.) and calls the function YYLEX as a scanner coroutine.

You provide a grammar specification file, which is traditionally named using a .y extension. You invoke YACC on the .y file and it creates the y.tab.h and y.tab.c files containing a thousand or so lines of intense C code that implements an efficient LALR(1) parser for your grammar, including the code for the actions you specified.

The file provides an extern function yyparse.y that will attempt to successfully parse a valid sentence. You compile that C file normally, link with the rest of your code, and you have a parser! By default, the parser reads from stdin and writes to stdout, just like a lex-generated scanner does.

```
% yacc myFile.y creates y.tab.c of C code for parser.  
% gcc -c y.tab.c compiles parser code  
% gcc -o parse y.tab.o lex.yy.o -l -ly link parser, scanner, libraries  
% parse invokes parser, reads from stdin
```

The Makefiles we provide for the projects will execute the above compilation steps for you, but it is worthwhile to understand the steps required.

Your input file is organized as follows (note the intentional similarities to lex):

```
%{  
Declarations  
%}  
Definitions  
%%  
Productions  
%%  
User subroutines
```

Another way to set precedence is by using the %prec directive.

When placed at the end of a production with a terminal symbol as its argument, it explicitly sets the precedence of the production to the same precedence as that terminal. This can be used when the right side has no terminals or when you want to overrule the precedence

## COMPILER DESIGN LAB RECORD

given by the rightmost terminal.

Even though it doesn't seem like a precedence problem, the dangling else ambiguity can be resolved using precedence rules. Think carefully about what the conflict is: Identify what the token is that could be shifted and the alternate production that could be reduced. What would be the effect of choosing the shift? What is the effect of choosing to reduce? Which is the one we want? Using **yacc's** precedence rules, you can force the choice you want by setting the precedence of the token being shifted versus the precedence of the rule being reduced. Whichever precedence is higher wins out.

The precedence of the token is set using the ordinary **%left**, **%right**, or **%nonassoc** directives. The precedence of the rule being reduced is determined by the precedence of the rightmost terminal (set the same way) or via an explicit **%prec** directive on the production.

commands for executing YACC

program-: yacc filename.y gcc

y.tab.c \$>> ./a.out

### ALGORITHM-:

1. Start.
2. Write the yacc specification file and name it as <filename>.y For ex. Program.y
3. Compile as yacc program.y
4. Compile the generated file y.tab.c using gcc.
5. Give the input and get the output.
6. Stop.

### PROGRAM

```
%{
```

```
#include<math.h>
```

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

```
#include<ctype.h>
```

```
#define YYSTYPE
```

```
double
```

```
%}
```

```
%%
```

```
input :
```

```
| input line
```

## COMPILER DESIGN LAB RECORD

;

line : '\n'

| expr '\n' {printf("Result is %g", \$1);};

expr : expr '+' term {\$\$=\$1+\$3;} | expr '-' term {\$\$=\$1-\$3;} |

term {\$\$=\$1;} ;

;

term : term '\*' factor {\$\$=\$1\*\$3;} | term '/' factor {\$\$=\$1/\$3;} | factor {\$\$=\$1;} ;

;

factor : NUM {\$\$=\$1;} ;

;

NUM : digit {\$\$=\$1;} |

NUM digit {\$\$=\$1\*10+\$2;} ;

digit : '0' {\$\$=0;} | '1' {\$\$=1;} | '2' {\$\$=2;} | '3' {\$\$=3;} | '4' {\$\$=4;} | '5' {\$\$=5;} | '6' {\$\$=6;} | '7' {\$\$=7;} | '8' {\$\$=8;} | '9' {\$\$=9;} ;

%%

yylex()

{

return getchar();

}

main()

{

return yyparse();

}

void yyerror(char \*s)

{

printf("%s", s);

}

## COMPILER DESIGN LAB RECORD

### OUTPUT

```
./a.out  
[>>> 2+3  
5  
>>> ]
```

### SIMULATION OF SHIFT REDUCED PARSER

**AIM-:** Write a program to implement a shift reduced parser operation. Also, mimic it using NLTK.

**DESCRIPTION-:** Shift Reduce parser attempts for the construction of parse in a similar manner as done in bottom-up parsing i.e. the parse tree is constructed from leaves(bottom) to the root(up). A more general form of the shift-reduce parser is the LR parser. This parser requires some data structures i.e. An input buffer for storing the input string. A stack for storing and accessing the production rules.

Basic Operations –

**Shift:** This involves moving symbols from the input buffer onto the stack.

**Reduce:** If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of a production rule is popped out of a stack and LHS of a production rule is pushed onto the stack.

**Accept:** If only the start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accepted action is obtained, it means successful parsing is done.

**Error:** This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

$E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid \text{digit}$

#### **ALGORITHM-**

1. Start.
2. Push the current symbol in stack at each push action.
3. Replace the symbol by a non-terminal when pop is called.
4. Display and continue for each symbol until only \$ is left, therefore it is accepted.
5. Stop.

#### **PROGRAM**

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

```
int z = 0, i = 0, j = 0, c = 0;
```

## COMPILER DESIGN LAB RECORD

```
char a[16], ac[20], stk[15], act[10]; void  
  
check()  
{  
    strcpy(ac, "REDUCE TO E -> "); for(z  
    = 0; z < c; z++)  
    {  
        if(stk[z] == '4')  
        {  
            printf("%s4", ac);  
            stk[z] = 'E';  
            stk[z + 1] = '\0';  
            printf("\n%s\t%s\t", stk, a);  
        }  
    }  
  
    for(z = 0; z < c - 2; z++)  
    {  
        if(stk[z] == '2' && stk[z + 1] == 'E' && stk[z + 2] == '2')  
        {  
            printf("%s2E2", ac);  
            stk[z] = 'E';  
            stk[z + 1] = '\0';  
            stk[z + 2] = '\0';  
            printf("\n%s\t%s\t", stk, a); i = i - 2;  
        }  
    }  
  
    for(z=0; z<c-2; z++)  
    {  
        if(stk[z] == '3' && stk[z + 1] == 'E' && stk[z + 2] == '3')  
        {  
            printf("%s3E3", ac);
```

## COMPILER DESIGN LAB RECORD

```
        stk[z]='E';

        stk[z + 1]='\0';

        stk[z + 1]='\0'; printf("\n$s%s\t%s$\t",
        stk, a); i = i - 2;

    }

}

return ;

}

int main()

{

    printf("GRAMMAR is -\nE->2E2 \nE->3E3 \nE->4\n");
    strcpy(a,"32423");

    c=strlen(a);
    strcpy(act,"SHIFT");

    printf("\nstack \t input \t action");
    printf("\n$\t%s$\t", a);

    for(i = 0; j < c; i++, j++)

    {

        printf("%s", act);

        stk[i] = a[j];

        stk[i + 1] = '\0';

        a[j]=' ';

        printf("\n$s%s\t%s$\t", stk, a);
        check();

    }

    check();

    if(stk[0] == 'E' && stk[1] == '\0')
        printf("Accept\n");

    else

        printf("Reject\n");

}
```

## COMPILER DESIGN LAB RECORD

### USING NLTK-:

```
import nltk

from nltk.parse import ShiftReduceParser

grammar = nltk.CFG.fromstring("""

S -> NP VP

NP -> PropN | N PP | Det N

VP -> V NP | V S | VP PP | NP PP PP ->
P NP

PropN -> 'He' | 'Utsav'
Det -> 'some' | 'the'

N -> 'compilerdesign' | 'programming' V ->
'eats' | 'studies'

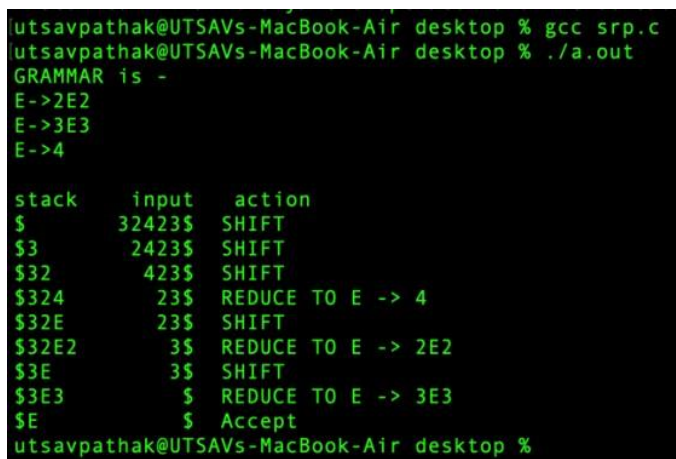
P -> 'and' | 'in' | 'is' """)

sr = ShiftReduceParser(grammar)

sentence1 = 'Utsav studies compilerdesign and someprogramming' tokens =
nltk.word_tokenize(sentence1)

for x in sr.parse(tokens):
    print(x)
```

### OUTPUT



```
utsavpathak@UTSAVs-MacBook-Air desktop % gcc srp.c
utsavpathak@UTSAVs-MacBook-Air desktop % ./a.out
GRAMMAR is -
E->2E2
E->3E3
E->4

stack   input   action
$       32423$  SHIFT
$3      2423$   SHIFT
$32     423$    SHIFT
$324    23$     REDUCE TO E -> 4
$32E    23$     SHIFT
$32E2   3$      REDUCE TO E -> 2E2
$3E     3$      SHIFT
$3E3    $       REDUCE TO E -> 3E3
$E      $       Accept
utsavpathak@UTSAVs-MacBook-Air desktop %
```



## COMPILER DESIGN LAB RECORD

```
[nltk_data] Downloading package punkt to /root/nltk_data...  
[nltk_data] Unzipping tokenizers/punkt.zip.  
(S  
  (NP (PropN Utsav))  
  (VP  
    (V studies)  
    (NP  
      (N compilerdesign)  
      (PP (P and) (NP (Det some) (N programming))))))
```

### NLTK Parsing

**AIM-:** Write a program to implement a NLTK Parsing operation.

**DESCRIPTION -:** Classes and interfaces for producing tree structures that represent the internal organization of a text. This task is known as “parsing” the text, and the resulting tree structures are called the text’s “pareses”. Typically, the text is a single sentence, and the tree structure represents the syntactic structure of the sentence. However, parsers can also be used in other domains.

The parser module defines `Parser I`, a standard interface for parsing texts; and two simple implementations of that interface, `ShiftReduceParser` and `RecursiveDescentParser`. It also contains three sub-modules for specialized kinds of parsing:

- `nltk.parser.chart` defines chart parsing, which uses dynamic programming to efficiently parse texts.
- `nltk.parser.proabilistic` defines probabilistic parsing, which associates a probability with each parse.

### PROGRAM

```
from nltk import CFG

grammar = CFG.fromstring("""

S -> NP VP

PP -> P NP

NP -> 'the' N | N PP | 'the' N PP

VP -> V NP | V PP | V NP PP

N -> 'cat'

N -> 'dog'

N -> 'rug'

V -> 'chased'

V -> 'sat'

P -> 'in'

P -> 'on'

""")
```

## COMPILER DESIGN LAB RECORD

```
from nltk.parse import RecursiveDescentParser

rd = RecursiveDescentParser(grammar)

sentence1 = 'the cat chased the dog'.split()

sentence2 = 'the cat chased the dog on the rug'.split()

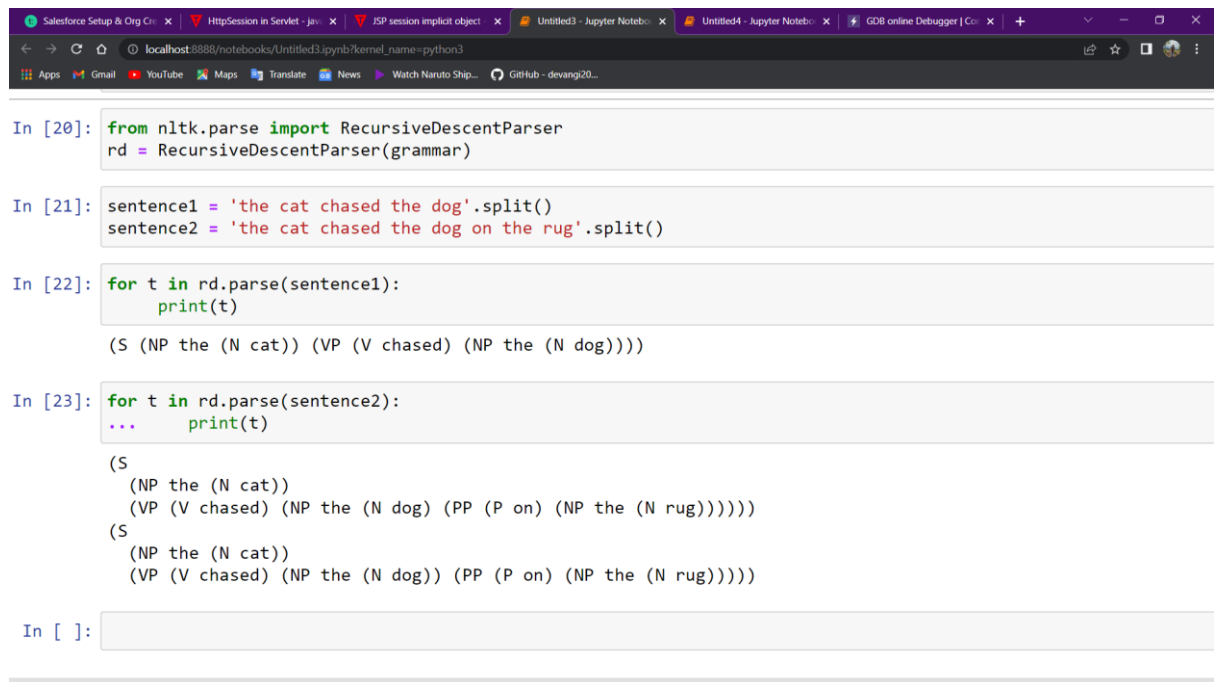
sentence1 = 'the cat chased the dog'.split()

sentence2 = 'the cat chased the dog on the rug'.split()

for t in rd.parse(sentence2):

...    print(t)
```

### OUTPUT



```
In [20]: from nltk.parse import RecursiveDescentParser
rd = RecursiveDescentParser(grammar)

In [21]: sentence1 = 'the cat chased the dog'.split()
sentence2 = 'the cat chased the dog on the rug'.split()

In [22]: for t in rd.parse(sentence1):
          print(t)

(S (NP the (N cat)) (VP (V chased) (NP the (N dog)))))

In [23]: for t in rd.parse(sentence2):
          ...    print(t)

(S
  (NP the (N cat))
  (VP (V chased) (NP the (N dog) (PP (P on) (NP the (N rug))))))
(S
  (NP the (N cat))
  (VP (V chased) (NP the (N dog)) (PP (P on) (NP the (N rug))))))

In [ ]:
```

### INTERMEDIATE CODE GENERATION

**AIM:-** To write a program that converts the high level language code into intermediate format, three address codes.

#### DESCRIPTION

The compilation process can be divided into a number of subtasks called phases. The different phases involved are

- Lexical analysis
- Syntax analysis
- Intermediate code generation
- Code Optimization
- Code generation.

#### Intermediate Code Generation

Once the syntactic constructs are determined, the compiler can generate object code for each construct. But the compiler creates an intermediate form. It helps in code optimization and also to make a clear-cut separation between machine independent phases (lexical, syntax) and machine dependent phases (optimization, code generation).

One form of intermediate code is a parse tree. A parse tree may contain variables as the terminal nodes. A binary operator will be having a left and right branch for operand1 and operand2.

Another form of intermediate code is a three-address code. It has got a general structure of  $A = B \text{ op } C$ , where A, B and C can be names, constants, temporary names etc. op can be any operator. Postfix notation is yet another form of intermediate code.

#### ALGORITHM

1. Get an expression as input.
2. Scan the statement from left to right.
3. Frame new statements such that it performs one operation at a time, either assignment or arithmetic or relational operations.
4. Display the new set of statements.

## COMPILER DESIGN LAB RECORD

### PROGRAM

```
import re

j = 1

y = []

op = ['+', '-', '/', '*']

x = input("ENTER THE EXPRESSION : ")

m = re.split('([+/*])', x)

# Intermediate Code Generation

def comp(m, j):

    for word in m:

        if word in op:

            y.append(''.join(m[0:3]))

            m.pop(0)

            m.pop(0)

            m[0] = 't'+str(j)

            j += 1

            return m, j

    while len(m) > 1:

        m, j = comp(m, j)

        k = len(y)

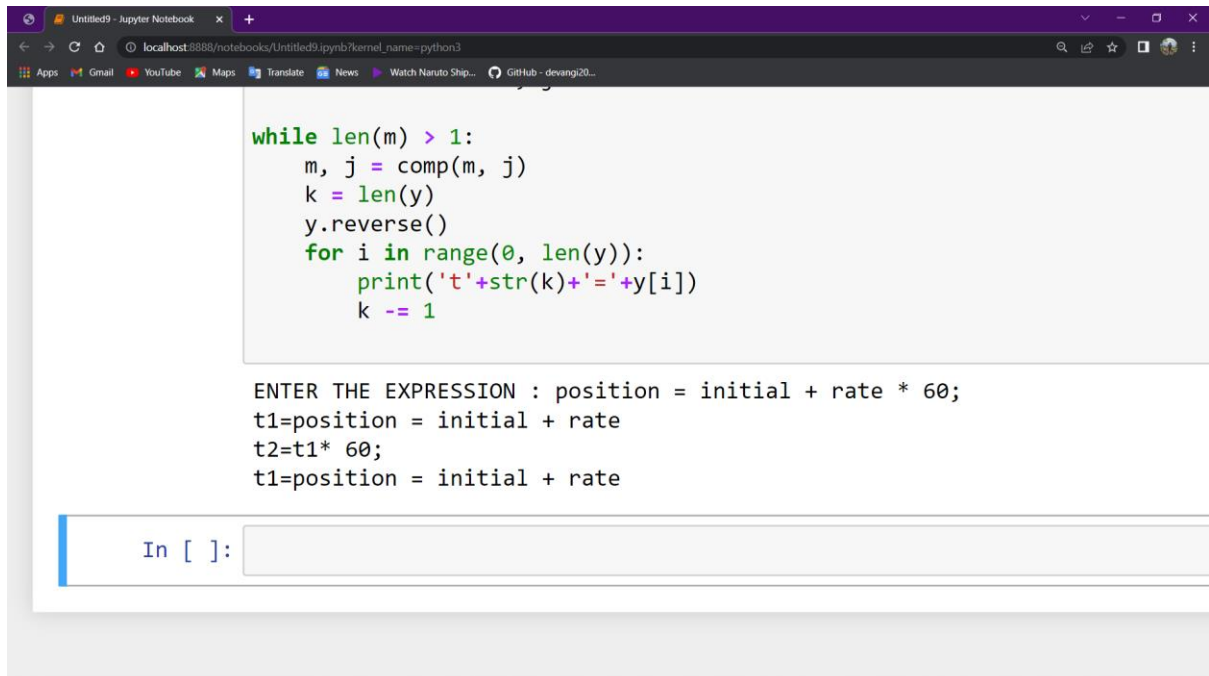
        y.reverse()

        for i in range(0, len(y)):

            print('t'+str(k)+'='+y[i])

            k -= 1
```

### OUTPUT



The screenshot shows a Jupyter Notebook window with a purple header bar. The browser address bar indicates the URL is localhost:8888/notebooks/Untitled9.ipynb?kernel\_name=python3. The notebook contains two code cells. The first cell has Python code for reversing a string and printing its characters. The second cell contains C code for calculating compound interest. Below the code cells is an input prompt 'In [ ]:' followed by an empty text box.

```
while len(m) > 1:
    m, j = comp(m, j)
    k = len(y)
    y.reverse()
    for i in range(0, len(y)):
        print('t'+str(k)+'='+y[i])
        k -= 1
```

```
ENTER THE EXPRESSION : position = initial + rate * 60;
t1=position = initial + rate
t2=t1* 60;
t1=position = initial + rate
```

In [ ]:

### IMPLEMENT CODE GENERATION

**AIM:** To implement the back end of the compiler which takes the three-address code and produces the 8086 assembly language instructions that can be assembled and run using a 8086 assembler. The target assembly instructions can be simple move, add, sub, jump. Also simple addressing modes are used.

**INTRODUCTION:** A compiler is a computer program that implements a programming language specification to “translate” programs, usually as a set of files which constitute the source code written in source language, into their equivalent machine readable instructions (the target language, often having a binary form known as object code).

**Code generation:** The transformed language is translated into the output language, usually the native machine language of the system. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated modes. Debug data may also need to be generated to facilitate debugging.

**ALGORITHM:**

1. Start the program
2. Open the source file and store the contents as quadruples.
3. Check for operators, in quadruples, if it is an arithmetic operator generator it or if assignment operator generates it, else perform unary minus on register C.
4. Write the generated code into output definition of the file in outp.c
5. Print the output.
6. Stop the program.

### **PROGRAM**

```
#include <stdio.h>

#include <string.h>

void main()

{

char icode[10][30], str[20], opr[10]; int i = 0;

printf("\nEnter Set Of Intermediate Code:\n");
```

## COMPILER DESIGN LAB RECORD

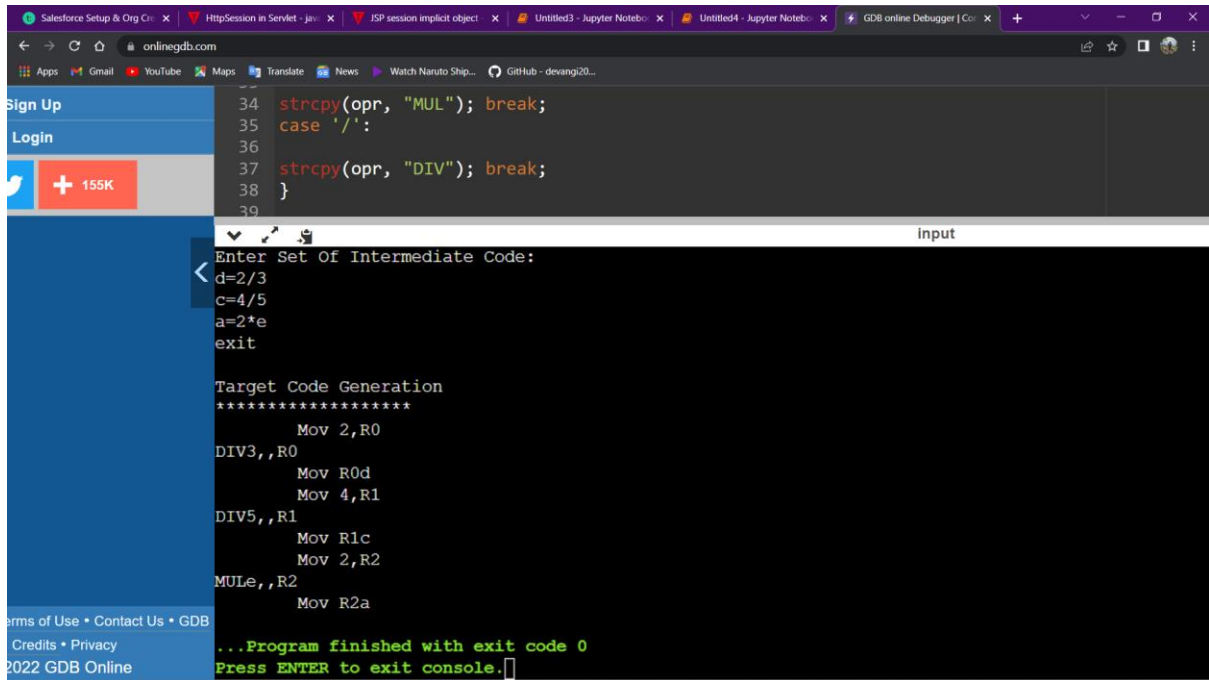
```
do
{
scanf("%s", icode[i]);
}
while (strcmp(icode[i++], "exit") != 0);
printf("\nTarget Code Generation");
printf("\n*****");
i = 0;
do{
strcpy(str, icode[i]); switch (str[3])
{
case '+':
strcpy(opr, "ADD"); break;
case '-':
strcpy(opr, "SUB"); break;
case '*':
strcpy(opr, "MUL"); break;
case '/':
strcpy(opr, "DIV"); break;
}
printf("\n\tMov %c,R%d", str[2], i);
printf("\n\t%s%c,,R%d", opr, str[4], i);
printf("\n\tMov R%d%c", i, str[0]);
}
```



## COMPILER DESIGN LAB RECORD

```
while (strcmp(icode[++i], "exit") != 0);  
}
```

### OUTPUT



The screenshot shows the onlinegdb.com interface. The top bar contains navigation links like 'Sign Up', 'Login', and a '+ 155K' button. The main editor area displays C code for a calculator: 

```
34 strcpy(opr, "MUL"); break;  
35 case '/':  
36  
37 strcpy(opr, "DIV"); break;  
38 }  
39
```

 Below the code editor, the 'input' section shows the user's input: 

```
< Enter Set Of Intermediate Code:  
d=2/3  
c=4/5  
a=2*e  
exit
```

 The 'Target Code Generation' section displays the corresponding assembly code: 

```
*****  
      Mov 2,R0  
DIV3,,R0  
      Mov R0d  
      Mov 4,R1  
DIV5,,R1  
      Mov R1c  
      Mov 2,R2  
MULe,,R2  
      Mov R2a
```

 At the bottom, a green message states: 

```
...Program finished with exit code 0  
Press ENTER to exit console.
```