CONTENTS

JAVA PROGRAMS

1. Remove Left Recursion

```java
package com.left_recursion;

import java.util.ArrayList;
import java.util.Arrays;

class LeftRecursion {
   final static String production = "A=Ab|c";
   final static String A = "A'";
   final static String eps = "eps";
   static ArrayList<String> alpha = new ArrayList<>();
   static ArrayList<String> beta = new ArrayList<>();

   static String[] getProductions(String s){
       String arr[] = s.split("\\|=");
       return arr;
   }

   static boolean checkLR(String arr[]){
       String lhs = arr[0];
       boolean bool = false;
       for(int j=1; j<arr.length; j++){
           if(lhs.charAt(0) == arr[j].charAt(0)){
               alpha.add(arr[j].substring(1));
               bool = true;
           }
           else{
               beta.add(arr[j]);
           }
       }
       return bool;
   }

   static void print(String s){
       System.out.println(s);
   }
```

```
    public static void main(String[] args) {
        String arr[] = getProductions(production);
        print(Arrays.toString(arr));
        if(checkLR(arr)){
            print("Yes left recursion exists");
            for(int i=0; i<beta.size(); i++){
                print(arr[0] + "=" + beta.get(i) + A);
            }

            String s = A+"=";
            for(int i=0; i<alpha.size(); i++){
                s = s + alpha.get(i) + A + "|";
            }

            print(s + eps);
        }
        else{
            print("No");
        }
    }

}

/*
Production: A=Ab | c
[A, Ab, c]
Yes left recursion exists
A=cA'
A'=bA'|eps

Production: A = Bb | c
[A, Bb, c]
No
*/
```

2. Three Address Code

```
package com.com.three_address_code;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Stack;

public class ThreeAddressCodeGeneric {
   static String result = "";
   // arranged in increasing order of precedence
   static List<String> operators = new
ArrayList<>(Arrays.asList("+", "-", "*", "/", "^"));

   static int getPrecedence(char c){
       switch (c)
       {
           case '+':
           case '-':
               return 1;

           case '*':
           case '/':
               return 2;

           case '^':
               return 3;
       }
       return -1;
   }
   static String infixToPostfix(String expression){
       Stack<Character> stack = new Stack<>();

       for(int i=0; i<expression.length(); i++){
           char c = expression.charAt(i);

           // if the character is a letter or a digit
           if(Character.isLetterOrDigit(c))
               result += c;

           // if the character is '('
```

```
            else if(c == '(')
                stack.push(c);

            // if the character is ')'
            else if(c == ')'){
                // until the stack is empty or we get '('
                while( !stack.isEmpty() && stack.peek() != '(')
                    result += stack.pop();

                stack.pop(); // popping the '('
            }
            // the character is operator
            else{
                // until the stack is empty or we encounter the
                   operator whose
                // precedence is less than that of current stack top
                while(!stack.isEmpty() && getPrecedence(c) <
getPrecedence(stack.peek()))
                    result += stack.pop();
                stack.push(c);
            }
        }

        // pop all the remaining operators from the stack
        while (!stack.isEmpty())
            result += stack.pop();

        return result;
    }

    static List<String> getOperators(List<String> exp){
        // returns the list of operators in order in which
        // they arrive in the list of postfix string.
        // eg: [a, b, *, c, +]
        // for this list it will return [*, +]
        List<String> op = new ArrayList<>();
        for(String s: exp){
            if(operators.contains(s))
                op.add(s);
        }
        return op;
    }
```

```java
    public void generate3AC(String expression){
        String postfix = infixToPostfix(expression);
        String [] expr = postfix.split("");
        List<String> exp = new ArrayList<>(Arrays.asList(expr));
        List<String> operatorsInPostfix = getOperators(exp);

        int i = 0;
        for(String operator : operatorsInPostfix){
            if(exp.contains(operator)){
                for(int j=0; j<exp.size(); j++){
                    if(exp.get(j).equals(operator)){
                        // for every operator encountered the arg2 is
                           at the i-2 index
                        // and arg1 is at the i-1 index
                        String r = exp.get(j-2);

                        // we print the statement t = arg1 + op +
arg2
                        System.out.printf("t%d = %s %s %s\n", i,
                                exp.get(j-1),
                                operator, exp.get(j-2));

                        // we set the arg1 to the result t
                        exp.set(j-1, "t"+i);

                        // remove the operator
                        exp.remove(operator);

                        // remove the arg2
                        exp.remove(r);
                        i++;

                        /*
                        eg: [a, b, *, c, +]
           for the above list first we will print t0 = a * b
           then we will set a = t0....hence [t0, b, *, c, +]
           now we will remove the operator... hence [t0, b, c, +]
           now we remove the arg1... hence [t0, c, +]
           now in the next iteration we evaluate the updated list.
                        */
                    }
```

```
                }
            }
        }
    }

    public static void main(String[] args) {
        String input = "a*b-a*b+a*b";
 // calling function for generic 3AC code (works for brackets as
well)
        System.out.println("Expression: " + input);
        ThreeAddressCodeGeneric threeAddressCodeGeneric = new
ThreeAddressCodeGeneric();
        threeAddressCodeGeneric.generate3AC(input);

    }
}
/*
Expression: a*b-a*b+a*b
t0 = b * a
t1 = b * a
t2 = b * a
t3 = t2 + t1
t4 = t3 - t0

Expression: (a-b)*(b-a)
t0 = b - a
t1 = a - b
t2 = t1 * t0
*/
```

3. Code Generation

```
import java.util.ArrayList;
import java.util.List;

class Statement{
    private String result;
    private String arg1;
    private String op;
    private String arg2;

    @Override
    public String toString() {
        return "Statement{" +
                "result='" + result + '\'' +
                ", arg1='" + arg1 + '\'' +
                ", op='" + op + '\'' +
                ", arg2='" + arg2 + '\'' +
                '}';
    }

    Statement(String result, String arg1, String op, String arg2) {
        this.arg1 = arg1;
        this.op = op;
        this.arg2 = arg2;
        this.result = result;
    }

    String getArg1() {
        return arg1;
    }

    String getOp() {
        return op;
    }

    String getArg2() {
        return arg2;
    }

    String getResult() {
        return result;
```

```
    }

}

public class CodeGeneration {
    static List<Statement> statements = new ArrayList<>();

    public static void parseInput(String input){
        String arr[] = input.split("\n");
        for(String s : arr){
            String [] st = s.split("");
            String result = st[0];
            if(st.length == 3){
                String op = st[1];
                String arg1 = st[2];
                String arg2 = "null";
                statements.add(new Statement(result, arg1, op,
arg2));
            }
            else{
                String op = st[3];
                String arg1 = st[2];
                String arg2 = st[4];
                statements.add(new Statement(result, arg1, op,
arg2));
            }
        }
    }

    public void generateCode(){
        for(Statement st : statements){
            System.out.printf("MOV R%d, %s\n",
statements.indexOf(st), st.getArg1());
            switch (st.getOp()){
                case "+":
                    System.out.printf("ADD R%d, %s\n",
statements.indexOf(st), st.getArg2());
                    break;

                case "-":
                    System.out.printf("SUB R%d, %s\n",
statements.indexOf(st), st.getArg2());
```

```
                    break;

                case "/":
                    System.out.printf("DIV R%d, %s\n",
statements.indexOf(st), st.getArg2());
                    break;

                case "*":
                    System.out.printf("MUL R%d, %s\n",
statements.indexOf(st), st.getArg2());
                    break;
            }
            System.out.printf("MOV %s, R%d\n", st.getResult(),
statements.indexOf(st));
        }
    }

    public static void main(String[] args) {
        CodeGeneration codeGeneration = new CodeGeneration();
        String input = "a=b\n" +
                "f=c+d\n" +
                "e=a-f\n" +
                "g=b*c";
        System.out.println("Original Code: ");
        System.out.println(input);
        codeGeneration.parseInput(input);
        System.out.println("\nMachine Code: ");
        codeGeneration.generateCode();
    }


}
```

```
/*
Original Code:
a=b
f=c+d
e=a-f
g=b*c

Machine Code:
MOV R0, b
MOV a, R0
MOV R1, c
ADD R1, d
MOV f, R1
MOV R2, a
SUB R2, f
MOV e, R2
MOV R3, b
MUL R3, c
MOV g, R3
*/
```

4. Intermediate Code Generation

```
import java.util.ArrayList;
import java.util.List;

class Quadruples{
   private int location;
   private String op;
   private String arg1;
   private String arg2;
   private String result;

   public Quadruples(int location, String op, String arg1, String
arg2, String result) {
       this.location = location;
       this.op = op;
       this.arg1 = arg1;
       this.arg2 = arg2;
       this.result = result;
   }

   public int getLocation() {
       return location;
   }

   public String getOp() {
       return op;
   }

   public String getArg1() {
       return arg1;
   }

   public String getArg2() {
       return arg2;
   }

   public String getResult() {
       return result;
   }
}
```

```
class Triples{
   private int location;
   private String op;
   private String arg1;
   private String arg2;

   public Triples(int location, String op, String arg1, String arg2)
{
       this.location = location;
       this.op = op;
       this.arg1 = arg1;
       this.arg2 = arg2;
   }

   public int getLocation() {
       return location;
   }

   public String getOp() {
       return op;
   }

   public String getArg1() {
       return arg1;
   }

   public String getArg2() {
       return arg2;
   }
}

class IndirectTriple{
   private int location;
   private String op;
   private String arg1;
   private String arg2;
   private int pointer;
   private int loc;

   public IndirectTriple(int location, String op,
                         String arg1, String arg2,
                         int pointer, int loc) {
```

```
        this.location = location;
        this.op = op;
        this.arg1 = arg1;
        this.arg2 = arg2;
        this.pointer = pointer;
        this.loc = loc;
    }

    public int getLocation() {
        return location;
    }

    public String getOp() {
        return op;
    }

    public String getArg1() {
        return arg1;
    }

    public String getArg2() {
        return arg2;
    }

    public int getPointer() {
        return pointer;
    }

    public int getLoc() {
        return loc;
    }
}


class Statement{
    private String result;
    private String arg1;
    private String op;
    private String arg2;

    @Override
    public String toString() {
```

```
        return "Statement{" +
                "result='" + result + '\'' +
                ", arg1='" + arg1 + '\'' +
                ", op='" + op + '\'' +
                ", arg2='" + arg2 + '\'' +
                '}';
    }

    Statement(String result, String arg1, String op, String arg2) {
        this.arg1 = arg1;
        this.op = op;
        this.arg2 = arg2;
        this.result = result;
    }

    String getArg1() {
        return arg1;
    }

    String getOp() {
        return op;
    }

    String getArg2() {
        return arg2;
    }

    String getResult() {
        return result;
    }

}

public class IntermediateCodeGeneration {
    static List<Quadruples> quadruples = new ArrayList<>();
    static List<Triples> triples = new ArrayList<>();
    static List<IndirectTriple> indirectTriples = new ArrayList<>();
    static List<Statement> statements = new ArrayList<>();

    public void parseInput(String input){
        String arr[] = input.split("\n");
        for(String s : arr){
```

```
            String [] st = s.split("");
            String result = st[0];
            if(st.length == 3){
                String op = st[1];
                String arg1 = st[2];
                String arg2 = "null";
                statements.add(new Statement(result, arg1, op,
arg2));
            }
            else{
                String op = st[3];
                String arg1 = st[2];
                String arg2 = st[4];
                statements.add(new Statement(result, arg1, op,
arg2));
            }
        }
    }

    static String getLocationOfResult(String result){
        // If the result is already present in the table then it will
returns its location
        // else it will return the result.
        for(Quadruples q : quadruples){
            if(q.getResult().equals(result)) {
                return String.valueOf(q.getLocation());
            }
        }
        return result;
    }

    public void generateCode(){
        int location = 100;
        int pointer = 35;
        for(Statement s: statements){
            quadruples.add(new Quadruples(location, s.getOp(),
                            s.getArg1(), s.getArg2(),
s.getResult()));
            triples.add(new Triples(location, s.getOp(),
                        getLocationOfResult(s.getArg1()),
getLocationOfResult(s.getArg2())));
```

```
            indirectTriples.add(new IndirectTriple(location,
s.getOp(),
                                getLocationOfResult(s.getArg1()),
getLocationOfResult(s.getArg2()),
                                pointer, location));
            location++;
            pointer++;

        }
        // Quadruples
        System.out.println("Quadruples: ");
        System.out.println("Location  |  Op  |  Arg1  |  Arg2  |
Result");

System.out.println("-----------------------------------------");
        for(Quadruples q : quadruples){
            System.out.printf("%8d  |  %2s  |  %4s  |  %4s  |  %5s",
                    q.getLocation(), q.getOp(), q.getArg1(),
q.getArg2(), q.getResult());
            System.out.println();
        }

        //Triples
        System.out.println("\nTriples: ");
        System.out.println("Location  |  Op  |  Arg1  |  Arg2");
        System.out.println("--------------------------------");
        for(Triples t : triples){
            System.out.printf("%8d  |  %2s  |  %4s  |  %4s",
                    t.getLocation(), t.getOp(), t.getArg1(),
t.getArg2());
            System.out.println();
        }

        //Indirect Triples
        System.out.println("\nIndirect Triples: ");
        System.out.println("Location  |  Op  |  Arg1  |  Arg2  |
Pointer  |  Loc");

System.out.println("-----------------------------------------------
-----");
        for(IndirectTriple i : indirectTriples){
```

```
            System.out.printf("%8d  |  %2s  |  %4s  |  %4s  |  %7d  |
%3d",
                    i.getLocation(), i.getOp(), i.getArg1(),
                    i.getArg2(), i.getPointer(), i.getLoc());
            System.out.println();
        }
    }

    public static void main(String[] args) {
        IntermediateCodeGeneration intermediateCodeGeneration = new
IntermediateCodeGeneration();
        String input = "a=b\n" +
                "f=c+d\n" +
                "e=a-f\n" +
                "g=b*c";
        System.out.println("Original Code: ");
        System.out.println(input);
        intermediateCodeGeneration.parseInput(input);
        System.out.println("\nIntermediate Code Generation: ");
        intermediateCodeGeneration.generateCode();
    }
}
```

```
/*
Original Code:
a=b
f=c+d
e=a-f
g=b*c

Intermediate Code Generation:
Quadruples:
Location  |  Op  |  Arg1  |  Arg2  |  Result
-------------------------------------------
    100  |   =  |     b  |  null  |      a
    101  |   +  |     c  |     d  |      f
    102  |   -  |     a  |     f  |      e
    103  |   *  |     b  |     c  |      g

Triples:
Location  |  Op  |  Arg1  |  Arg2
---------------------------------
    100  |   =  |     b  |  null
    101  |   +  |     c  |     d
    102  |   -  |   100  |   101
    103  |   *  |     b  |     c

Indirect Triples:
Location  |  Op  |  Arg1  |  Arg2  |  Pointer  |  Loc
-----------------------------------------------------
    100  |   =  |     b  |  null  |       35  |  100
    101  |   +  |     c  |     d  |       36  |  101
    102  |   -  |   100  |   101  |       37  |  102
    103  |   *  |     b  |     c  |       38  |  103
*/
```

5. Code Optimization

```
import java.util.ArrayList;
import java.util.List;

class Statement{
   private String lhs;
   private String rhs;

   public Statement(String lhs, String rhs){
       this.lhs = lhs;
       this.rhs = rhs;
   }

   public String getLhs() {
       return lhs;
   }

   public void setLhs(String lhs) {
       this.lhs = lhs;
   }

   public String getRhs() {
       return rhs;
   }

   public void setRhs(String rhs) {
       this.rhs = rhs;
   }

   @Override
   public String toString() {
       return lhs + " = " + rhs;
   }
}

public class CommonSubExpressions {
   static List<Statement> statements = new ArrayList<>();

   public void remove(String input) {
       System.out.println("\nAfter removing common sub expressions:
");
```

```
        String [] expressions = input.split("\n");

        for(String s : expressions){
            statements.add(new Statement(s.split("=")[0],
s.split("=")[1]));
        }
        for(int i=0; i<statements.size(); i++){
            String rhs = statements.get(i).getRhs();
            String lhs = statements.get(i).getLhs();
            for(int j=i+1; j<statements.size(); j++){
                if(statements.get(j).getRhs().contains(rhs)){

statements.get(j).setRhs(statements.get(j).getRhs().replace(rhs,
lhs));
                }
            }
        }

        for(Statement s : statements){
            System.out.println(s);
        }
    }

    public static void main(String[] args) {
        String input = "a=b+c\n" +
                "d=(b+c)*e\n" +
                "f=(b+c)*(b+c)";

        System.out.println("\nBefore removing common sub expressions:
");
        System.out.println(input);
        CommonSubExpressions commonSubExpressions = new
CommonSubExpressions();
        commonSubExpressions.remove(input);
    }
}
```

```
/*
Before removing common sub expressions:
a=b+c
d=(b+c)*e
f=(b+c)*(b+c)

After removing common sub expressions:
a = b+c
d = (a)*e
f = (a)*(a)
*/
```

6. Design Lexical Analyser to identify tokens

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Lexer{
   Pattern full = Pattern.compile(
           "(?<line>if\\s+" +
                   "+(?<phrase>" +
                   "(?<noun1>[a-z]+)\\s+" +
                   "(?<verb>hate|like)\\s+" +
                   "(?<noun2>[a-z]+)\\s+" +
                   ")"+
                   "then\\s+" +
                   "(?<action>" +
                   "they\\s+" +
                   "(?<noun3>[a-z]+)" +
                   ")" +
                   "\\.)",
           Pattern.CASE_INSENSITIVE);
}

public class LexicalAnalyser {
   public static void main(String[] args) {
       String input = "If dogs hate cats then they chase. " +
                      "If cats like milk then they drink.";
       //region Second approach with roided out single regex
       Lexer lexer2 = new Lexer();
       Matcher matcher_full = lexer2.full.matcher(input);
       while(matcher_full.find()){
           System.out.println("Line:" + matcher_full.group("line"));
           System.out.println("Phrase:" +
                              matcher_full.group("phrase"));
           System.out.println("Noun1:" +
matcher_full.group("noun1"));
           System.out.println("Verb:" + matcher_full.group("verb"));
           System.out.println("Noun2:" +
matcher_full.group("noun2"));
           System.out.println("Action:" +
                              matcher_full.group("action"));
       }
```

```
        //endregion
    }
}

/*
Output:
Line:If dogs hate cats then they chase.
Phrase:dogs hate cats
Noun1:dogs
Verb:hate
Noun2:cats
Action:they chase
Line:If cats like milk then they drink.
Phrase:cats like milk
Noun1:cats
Verb:like
Noun2:milk
Action:they drink
*/
```

LEX PROGRAMS

1. Lex Program for Calculator

```
%option noyywrap

%{
#include <stdio.h>
int op = 0;
float a,b;
%}

dig [0-9]+|([0-9]*)"."([0-9]+)
add "+"
sub "-"
mul "*"
div "/"
ln \n

%%

{dig} {digi();}
{add} {op = 1;}
{sub} {op = 2;}
{mul} {op = 3;}
{div} {op = 4;}
{ln} {printf("\n Result = %2f", a);}

%%
int digi(){
  if (op == 0){
      a = atof(yytext);
  }
  else{
      b = atof(yytext);
      switch(op){
          case 1: a = a + b; break;
          case 2: a = a - b; break;
          case 3: a = a * b; break;
          case 4: a = a / b; break;
```

```
      }
      op = 0;
  }
}

int main(){
  printf("Enter Expression: ");
  yylex();
  return 0;
}
```

2. Count Character, Words, Lines, Spaces

```
%{
#include<stdio.h>
int words = 0;
int space = 0;
int characters = 0;
int lines = 0;
%}

%%
[ ]+ {space++; words++;}
[^\t\n] {characters++;}
[\n] {lines++; words++;printf("Space = %d; Word = %d, Characters =
%d; Lines = %d", space, words, characters, lines);}
%%

int yywrap(){
  return 1;
}

int main(){
  printf("Enter String: ");
  yylex();
  return 0;
}
```

3. Count Special Characters

```
%option noyywrap
%{
   #include<stdio.h>
   int special_characters = 0;
%}

%%
[@#$%&] {special_characters++;}
\n { printf("\nSpecial Characters = %d", special_characters);}
%%

int main(){
   printf("Enter the string of your choice: \n");
   yylex();
   return 0;
}
```

4. Count Vowels and Consonants

```
%option noyywrap
%{
   #include<stdio.h>
   int vowel = 0;
   int consonants = 0;
%}

%%
[aeiouAEIOU] {printf("Vowels\t\n"); vowel++;}
[^aeiouAEIOU\n] {printf("Consonants\t\n"); consonants++;}
\n { printf("Vowels = %d and Consonants = %d",vowel,consonants);}
%%

int main(){
   printf("Enter the string of your choice: \n");
   yylex();
   return 0;
}
```

5. Identify Keywords

```
%option noyywrap
%{
  #include<stdio.h>
  int keywords = 0;
%}

%%
if|else|while|int|switch|for|char {keywords++;}
\n {printf("Total Number of keywords: %d", keywords);}
%%

int main()
{
   printf("Enter the string of your choice: \n");
   yylex();
   return 0;
}
```

6. Identify Tokens

```
%option noyywrap
%{
  #include<stdio.h>
  int keywords = 0;
  int identifiers = 0;
  int operators = 0;
  int constants = 0;
%}

%%
if|else|while|int|switch|for|char {keywords++;}

[A-Za-z][A-Za-z0-9_]* {identifiers++;}

[=+\-*/^] {operators++;}

[0-9] {constants++;}

\n {printf("\nTotal Number Of\n Keywords: %d\nConstants:
   %d\nIdentifiers: %d\nOperators: %d\n", keywords, constants,
   identifiers, operators);}

%%

int main()
{
   printf("Enter the string of your choice: \n");
   yylex();
   return 0;
}
```

7. Show Vowels and Consonants

```
%{
#include<stdio.h>
char vowels[40];
char consos[40];
int n_vowels = 0;
int n_consos = 0;
// yytext will contain the matched characters
%}
%%
[\r\n] {
   vowels[n_vowels] = '\0';
   consos[n_consos] = '\0';

   printf("Vowels: %s\n", vowels);
   printf("Consos: %s\n", consos);

   // Reset the arrays
   n_vowels = n_consos = 0;
}

\\s+ {/*ignore whitespace*/}

[aeiouAEIOU] {
   vowels[n_vowels++] = yytext[0];
}
[^aeiouAEIOU] {
   consos[n_consos++] = yytext[0];
}

%%

int main(){
   printf("1b. Show vowels and consonants\n");
   yylex();
   return 0;
}
```

8. Identify Uppercase and Lowercase Letters

```
%option noyywrap
%{
   #include<stdio.h>
   int lower = 0;
   int upper = 0;
%}

%%
[A-Z] {printf("Upper case\t\n"); upper++;}
[a-z] {printf("Lower case\t\n"); lower++;}
\n {printf("Upper case = %d and lower case = %d", upper, lower);}
%%

int main(){
   printf("Enter the string of your choice: \n");
   yylex();
   return 0;
}
```

THEORY QUESTIONS

1. Code Optimization Techniques

Variable Propagation :

```
//Before Optimization
c = a * b
x = a
till
d = x * b + 4


//After Optimization
c = a * b
x = a
till
d = a * b + 4
```

Dead code elimination :

- Variable propagation often leads to making assignment statement into dead code

```
c = a * b
x = a
till
d = a * b + 4

//After elimination :
c = a * b
till
d = a * b + 4
```

Code Motion :

- Reduce the evaluation frequency of expression.
- Bring loop invariant statements out of the loop.

```
a = 200;
while(a>0){
   b = x + y;
   if (a % b == 0)
       printf("%d", a);
}


//This code can be further optimized as
a = 200;
b = x + y;
while(a>0){
   if (a % b == 0}
   printf("%d", a);
}
```

Induction Variable and Strength Reduction :

- An induction variable is used in loop for the following kind of assignment i = i + constant.
- Strength reduction means replacing the high strength operator by the low strength.

```
i = 1;
while (i<10){
   y = i * 4;
}


//After Reduction
i = 1
t = 4
while(t<40)
   y = t;
   t = t + 4;
```

2. Different Phases Of Compilation

- Lexical Analysis:

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes.

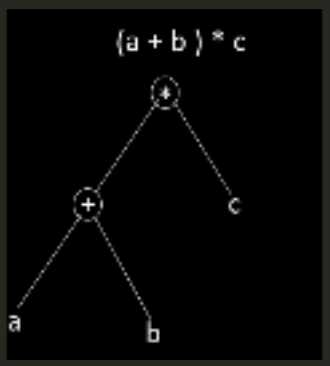
x = y + 10

Tokens

X	identifier
=	Assignment operator
Y	identifier
+	Addition operator
10	Number

- Syntax Analysis:

The next phase is called the syntax analysis or parsing. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.



- Semantic Analysis:

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Example:

```
float x = 20.2;
float y = x*30;
```

In the above code, the semantic analyzer will typecast the integer 30 to float 30.0 before multiplication

- Intermediate Code Generation:

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Example:

```
total = count + rate * 5
```

Intermediate code with the help of address code method is:

```
t1 := int_to_float(5)
t2 := rate * t1
t3 := count + t2
total := t3
```

- Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Example:

Consider the following code

```
a = intofloat(10)
b = c * a
d = e + b
f = d
```

Can become

```
b =c * 10.0
f = e+b
```

- Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) relocatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

Example:

```
     a = b + 60.0
```

Would be possibly translated to registers.

```
     MOVF a, R1
     MULF #60.0, R2
     ADDF R1, R2
```

- Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

3. Direct Linking Loader

Introduction:

The direct linking loader is the most common type of loader. This type of loader is a relocatable loader. The loader cannot have the direct access to the source code. And to place the object code in the memory there are two situations: either the address of the object code could be absolute which then can be directly placed at the specified location or the address can be relative. If at all the address is relative then it is the assembler who informs the loader about the relative addresses.

The assembler generates following types of cards:

ESD - External symbol dictionary contains information about all symbols that are defined in this program but referenced somewhere.

TXT - Text card contains actual object cards.

RLD - Relocation and linkage directory contains information about address dependent instructions of a program.

END - Indicates the end of the program and specifies starting address of for execution

4. Forms of Intermediate Code

Postfix Notation

Example - The postfix representation of the expression (a – b) * (c + d) + (a – b) is : ab – cd + *ab -+.

Three-Address Code

Example - The three address code for the expression a + b * c + d :

T 1 = b * c

T 2 = a + T 1

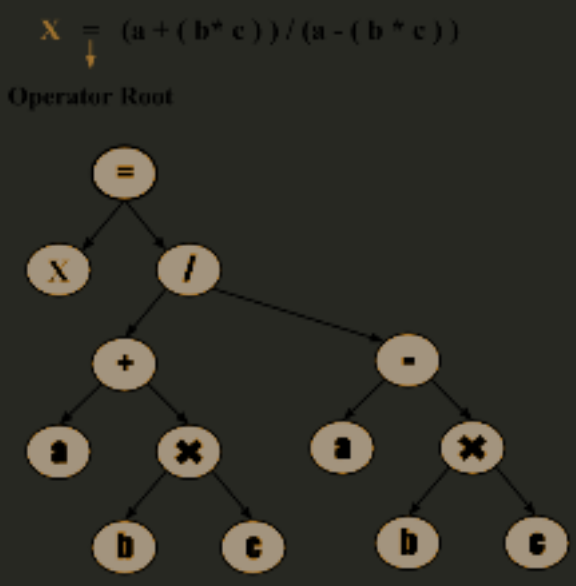
T 3 = T 2 + d

T 1 , T 2 , T 3 are temporary variables.

Syntax Tree

Example:

x = (a + b * c) / (a – b * c)



5. Peephole Optimization

This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand. These methods can be applied on intermediate codes as well as on target codes. A bunch of statements is analyzed and are checked for the following possible optimization:

Redundant instruction elimination:

At source code level, the following can be done by the user:

int add_ten(int x)
{
int y, z;
y = 10;
z = x + y;
return z;
}
	int add_ten(int x)
	{
	int y;
	y = 10;
	y = x + y;
	return y;
	}
		int add_ten(int x)
		{
		int y = 10;
		return x + y;
		}
			int add_ten(int x)
			{
			return x + 10;
			}

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

```
MOV x, R0

MOV R0, R1
```

We can delete the first instruction and rewrite the sentence as:

```
MOV x, R1
```

6. Problems in Top Down Parsing

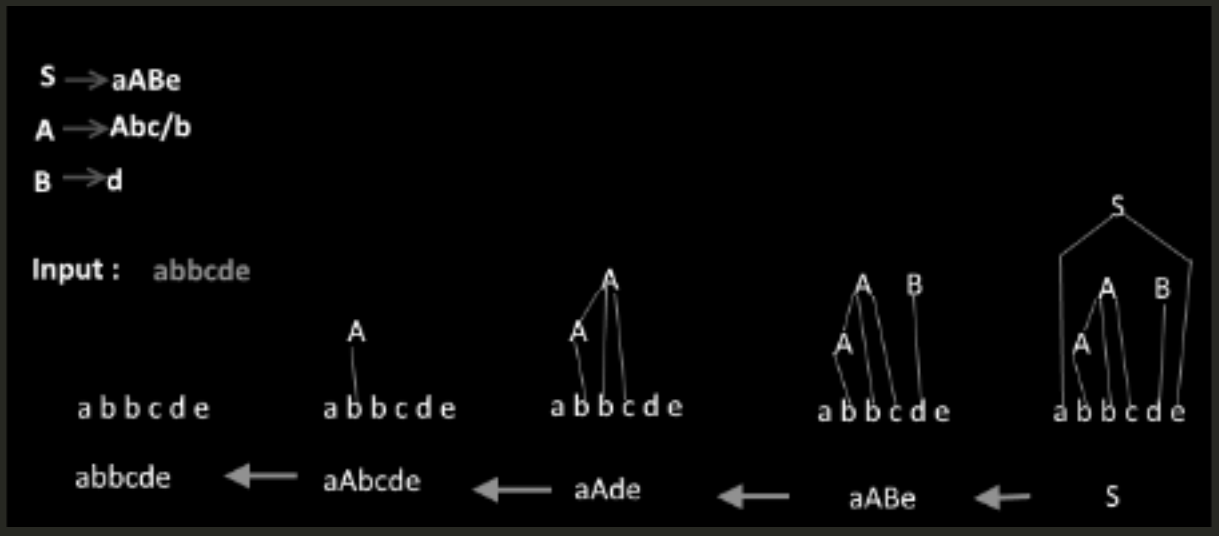
Problems with the Top-Down Parser

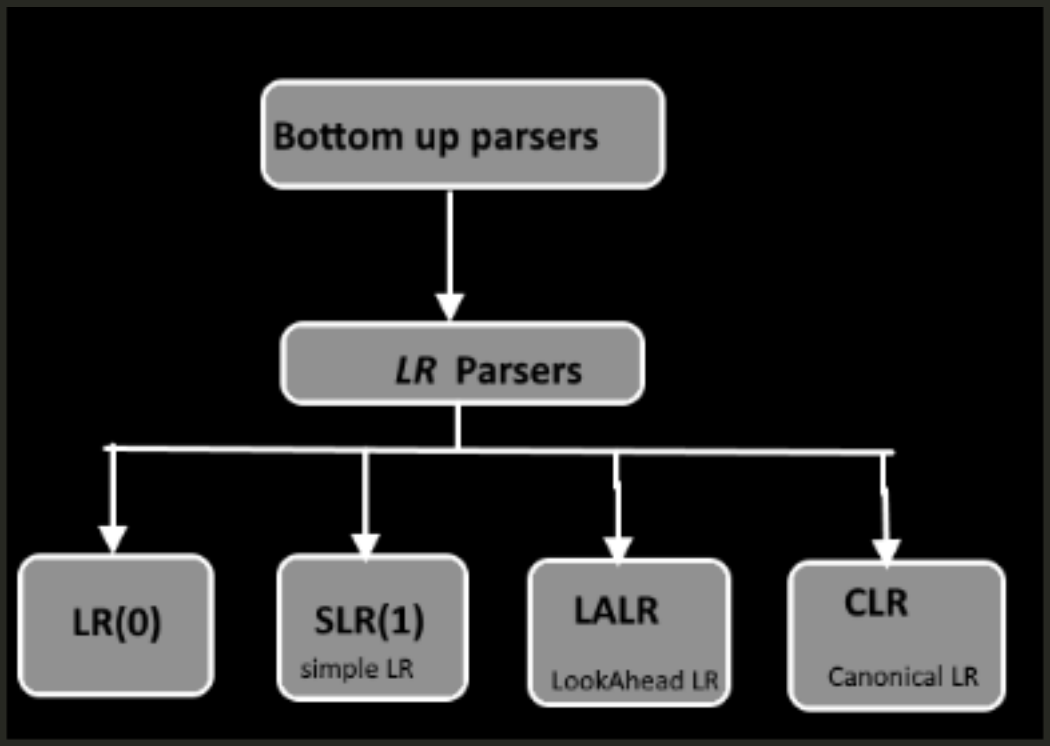
1. Only judges grammatically
2. Stops when it finds a single derivation.
3. No semantic knowledge employed.
4. No way to rank the derivations.
5. Problems with left-recursive rules.
6. Problems with ungrammatical sentences.

7. Bottom Up Parsing

Also called Shift Reduce Parsers

Build the parse tree from leaves to root. Bottom-up parsing can be defined as an attempt to reduce the input string w to the start symbol of grammar by tracing out the rightmost derivations of w in reverse.





8. Top Down Parser

The process of constructing the parse tree which starts from the root and goes down to the leaf is Top-Down Parsing.

- Top-Down Parsers constructs from the Grammar which is free from ambiguity and left recursion.
- Top Down Parsers uses leftmost derivation to construct a parse tree.
- It allows a grammar which is free from Left Factoring.

Brute Force Technique or Recursive Descent Parsing –

1. Whenever a Non-terminal spend first time then go with the first alternative and compare with the given I/P String
2. If matching doesn't occur then go with the second alternative and compare with the given I/P String.
3. If matching again not found then go with the alternative and so on.
4. Moreover, If matching occurs for at least one alternative, then the I/P string is parsed successfully.

LL(1) or Table Driver or Predictive Parser –

1. In LL1, first L stands for Left to Right and second L stands for Left-most Derivation. 1 stands for number of Look Aheads token used by parser while parsing a sentence.
2. LL(1) parsing is constructed from the grammar which is free from left recursion, common prefix, and ambiguity.
3. LL(1) parser depends on 1 look ahead symbol to predict the production to expand the parse tree.
4. This parser is Non-Recursive.

9. Operator Precedence Grammar

- An operator precedence grammar is a kind of grammar for formal languages.
- Technically, an operator precedence grammar is a context-free grammar that has the property (among others) that no production has either an empty right-hand side or two adjacent nonterminals in its right-hand side.
- These properties allow precedence relations to be defined between the terminals of the grammar.
- A parser that exploits these relations is considerably simpler than more general-purpose parsers such as LALR parsers.
- Operator-precedence parsers can be constructed for a large class of context-free grammars.

Operator precedence grammars rely on the following three precedence relations between the terminals:

Relation	Meaning
$a \lessdot b$	a yields precedence to b
$a \doteq b$	a has the same precedence as b
$a \gtrdot b$	a takes precedence over b

10. Algorithm of Pass 1 and Pass 2 Assembler

Pass 1

```
Assembler Pass 1:
begin
    read first input line
        if OPCODE ='START' then
                begin
                save #[OPERAND] as starting address
                initialize LOCCTR to starting address
                write line to intermediate file
                read next input line
                end {if START}
        else
                initialize LOCCTR to 0
        while OCODE != 'END' do
                begin
                        if this is not a comment line then
                          begin
                                if there is a symbol in the LABEL field then
                                  begin
                                        search SYMTAB for LABEL
                                        if found then
                                          set error flag (duplicate symbol)
                                        else
                                          insert (LABEL,LOCCTR) into SYMTAB
                                  end {if symbol}
                                search OPTAB for OPCODE
                                if found then
                                  add 3 {instruction length} to LOCCTR
                                else if OPCODE='WORD' then
                                        add 3 to LOCCTR
                                else if OPCODE = 'RESW' then
                                        add 3 * #[OPERAND] to LOCCTR
                                else if OPCODE = 'RESB' then
                                        add #[OPERAND] to LOCCTR
                                else if OPCODE = 'BYTE' then
                                  begin
                                        find length of constant in bytes
                                        add length to LOCCTR
                                  end {if BYTE}
                                else
                                   set error flag (invalid operation code)
                        end {if not a comment}
                write line to intermediate file
                read next input line
        end {while not END}
  write last line to intermediate file
 save (LOCCTR – starting address) as program length
end  {Pass 1}
```

Pass2

```
Assembler Pass2:
begin
  read first input line (from intermediate file)
  if OPCODE ='START' then
    begin
        write listing line
        read next input line
    end {if START}
  write Header record to object program
  initialize first Text  record
   while OPCODE != 'END' do
     begin
         if this is not a comment line then
        begin
                search OPTAB for OPCODE
                if found then
                      begin
                       if there is a symbol in OPERAND  field then
                            begin
                              search SYMTAB for OPERAND
                              if found then
                                    store symbol value as operand address
                              else
                                    begin
                                      store 0 as operand address
                                       set error flag (undefined symbol)
                                    end
                             end {if symbol}
                        else
                             store 0 as operand address
                        assemble the object code instruction
            end {if opcode found}
               else if OPCODE ='BYTE' or 'WORD' then
                        convert constant to object code
          if object code will not fit into the current Text record then
                    begin
                        write Text record to object program
                        initialize new Text record
                    end
           add object code to Text record
                end {if not comment}
        write listing line
        read next input line
     end(while not END)
  write last Text record to object program
  write End record to object program
  write last listing line
end{Pass 2}
```