9. Diffie-Hellman

AIM: Implement the Diffie-Hellman Key Exchange mechanism using HTML and JavaScript. Consider the end user as one of the parties (Alice) and the JavaScript application as other party (bob).

PROGRAM:

```
import java.math.BigInteger;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom; import
javax.crypto.spec.DHParameterSpec; import
javax.crypto.spec.DHPublicKeySpec; public
class DiffeHellman { public final static int
pValue = 47;
public final static int gValue = 71;
public final static int XaValue = 9;
public final static int XbValue = 14;
public static void main(String[] args) throws
      Exception { // TODO code application logic here
BigInteger p = new BigInteger(Integer.toString(pValue));
BigInteger g = new BigInteger(Integer.toString(gValue));
BigIntegerXa = new
BigInteger(Integer.toString(XaValue)); BigIntegerXb =
new BigInteger(Integer.toString(XbValue)); createKey();
intbitLength = 512; // 512 bits
SecureRandomrnd = new SecureRandom();
      p = BigInteger.probablePrime(bitLength, rnd);
      g = BigInteger.probablePrime(bitLength, rnd);
```



```
createSpecificKey(p, g);
   }
public static void createKey() throws Exception {
KeyPairGeneratorkpg =
KeyPairGenerator.getInstance("DiffieHellman"); kpg.initialize(512);
KeyPairkp = kpg.generateKeyPair();
KeyFactorykfactory = KeyFactory.getInstance("DiffieHellman");
DHPublicKeySpeckspec = (DHPublicKeySpec) kfactory.getKeySpec(kp.getPublic(),
DHPublicKeySpec.class);
System.out.println("Public key is: " +kspec);
   }
public static void createSpecificKey(BigInteger p, BigInteger g) throws
Exception { KeyPairGeneratorkpg =
KeyPairGenerator.getInstance("DiffieHellman"); DHParameterSpecparam = new
DHParameterSpec(p, g); kpg.initialize(param);
KeyPairkp = kpg.generateKeyPair();
KeyFactorykfactory = KeyFactory.getInstance("DiffieHellman");
DHPublicKeySpeckspec = (DHPublicKeySpec) kfactory.getKeySpec(kp.getPublic(),
DHPublicKeySpec.class);
System.out.println("\nPublic key is : " +kspec);
   }
}
```

OUTPUT:

```
Public key is: javax.crypto.spec.DHPublicKeySpec@5afd29
Public key is: javax.crypto.spec.DHPublicKeySpec@9971ad
```

10. SHA-1

AIM: Calculate the message digest of a text using the SHA-1 algorithm in JAVA.

PROGRAM:

```
import java.security.*;
public class SHA1 {
public static void main(String[] a) {
try {
MessageDigest md = MessageDigest.getInstance("SHA1");
System.out.println("Message digest object info: ");
System.out.println(" Algorithm = " +md.getAlgorithm());
System.out.println(" Provider = " +md.getProvider());
System.out.println(" ToString = " +md.toString());

        String input = "";
md.update(input.getBytes());
byte[] output = md.digest();
System.out.println();
System.out.println("SHA1(\""+input+"\") = " +bytesToHex(output));

input = "abc";
md.update(input.getBytes());
output = md.digest();
System.out.println();
System.out.println("SHA1(\""+input+"\") = " +bytesToHex(output));

input = "abcdefghijklmnopqrstuvwxyz";
md.update(input.getBytes());
output = md.digest();
System.out.println();
System.out.println("SHA1(\"" +input+"\") = " +bytesToHex(output));
System.out.println(""); }
catch (Exception e) {
```



```
System.out.println("Exception: " +e);
      }
  }

public static String bytesToHex(byte[] b) {
      char hexDigit[] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'};
StringBufferbuf = new StringBuffer();
for (int j=0; j<b.length; j++) {
buf.append(hexDigit[(b[j] >> 4) & 0x0f]);
buf.append(hexDigit[b[j] & 0x0f]); }
returnbuf.toString(); }
}
```

OUTPUT:

```
Message digest object info:
  Algorithm = SHA1
  Provider = SUN version 1.6
ToString = SHA1 Message Digest from SUN, <initialized> SHA1("") =
DA39A3EE5E6B4B0D3255BFEF95601890AFD80709 SHA1("abc") =
A9993E364706816ABA3E25717850C26C9CD0D89D
SHA1("abcdefghijklmnopqrstuvwxyz")=32D10C7B8CF96570CA04CE37F2A19D8424
0D3A89
```

3. Implementation of lexical analyzer using lex tool.

Flex (Fast Lexical Analyzer) is a tool for generating lexical analyzers in C or C++ based on regular expressions.

Steps for implementing a lexical analyzer using Flex:

1. Install Flex:
   - You can install Flex on Linux with the following command:

     sudo apt install flex

2. Create a Lex Specification File (lexer.l): Write a .l file that defines the regular expressions (patterns) for each token type.

Example of lexer.l file:

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%%
// Regular Expressions (token definitions)
int     [a-zA-Z_][a-zA-Z0-9_]*
float   [0-9]+\.[0-9]+
digit   [0-9]+
keyword int|float|if|else|for|while|return
%%
// Rules and actions
{keyword}       { printf("Token: %s | Type: KEYWORD\n", yytext); }
{int}           { printf("Token: %s | Type: IDENTIFIER\n", yytext); }
{float}         { printf("Token: %s | Type: CONSTANT\n", yytext); }
{digit}         { printf("Token: %s | Type: CONSTANT\n", yytext); }
"+"|"-"|"*"|"/" { printf("Token: %s | Type: OPERATOR\n", yytext); }
"(" | ")" | ";" | "," { printf("Token: %s | Type: PUNCTUATION\n", yytext); }
"//".*          { printf("Token: %s | Type: COMMENT\n", yytext); }
"/*".* "*/"     { printf("Token: %s | Type: COMMENT\n", yytext); }
[ \t\n\r]+      { /* Ignore whitespace */ }

.               { printf("Token: %s | Type: UNKNOWN\n", yytext); }
%%

int main() {
   yylex();  // Invoke the lexical analyzer
   return 0;
}
```

Explanation:

- Regular Expressions: Similar to JLex, we define token patterns such as identifiers, constants, operators, comments, etc.
- Actions: Each pattern has an associated action. When a pattern is matched, it prints the matched token and its type.

3. Generate the Lexer with Flex: After creating the lexer.l file, run Flex to generate the C code for the lexical analyzer:
   flex lexer.l

   This will create a file called lex.yy.c containing the lexer code.

4. Compile the C Code: Compile the generated lex.yy.c file using GCC:
   gcc lex.yy.c -o lexer -lfl

   The -lfl flag is used to link the Flex library.

5. Run the Lexer: After compiling, you can run your lexer on a text input file:
   ./lexer < input.c

   This will read the file input.c, process it, and output the recognized tokens.

Example Output:

For the input:

```
int x = 100;  // This is a comment
float y = 3.14;
x = x + y;
```

5. Convert the bnf rules into yacc form and write code to generate abstract syntax tree.

To convert BNF (Backus-Naur Form) rules into Yacc form and write code to generate an Abstract Syntax Tree (AST), let's break it down into steps:

1. Understanding the BNF Grammar: We'll first take a simple arithmetic expression grammar in BNF form.
2. Convert BNF to Yacc: Convert the BNF rules to Yacc's syntax and add actions to build the AST.
3. Define AST Nodes: Create structs to represent the AST nodes and their relationships.
4. Actions in Yacc: During parsing, we will create AST nodes for each operation.

Let's implement the solution for a simple arithmetic expression that uses the operators +, -, *, / and constructs an Abstract Syntax Tree.

1. BNF Grammar

A simple BNF for arithmetic expressions:

```
<expression> ::= <expression> '+' <term>
           | <expression> '-' <term>
           | <term>
<term>       ::= <term> '*' <factor>
           | <term> '/' <factor>
           | <factor>
<factor>     ::= <number>
           | '(' <expression> ')'
<number>     ::= [0-9]+
```

This is the arithmetic grammar for addition, subtraction, multiplication, division, and parentheses.

2. Convert BNF to Yacc

We will now translate this BNF grammar to Yacc form and include actions to create the Abstract Syntax Tree.

3. Define Abstract Syntax Tree (AST) Structure

An AST is represented using structs. For arithmetic expressions, the nodes can represent operations (like addition or multiplication) and numbers.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// Abstract Syntax Tree (AST) structures
typedef struct node {

   enum { NUM, ADD, SUB, MUL, DIV } type; // Node types: NUM (number), ADD
(addition), SUB (subtraction), MUL (multiplication), DIV (division)
   union {
      int value; // Holds the number for NUM nodes
      struct {   // Holds pointers to child nodes for operator nodes
         struct node *left;
         struct node *right;
      };
   };
} Node;
// Functions to create nodes for the AST
Node *new_number(int value) {
   Node *n = malloc(sizeof(Node));
   n->type = NUM;
   n->value = value;
   return n;
}
Node *new_operator(int type, Node *left, Node *right) {
   Node *n = malloc(sizeof(Node));
   n->type = type;
   n->left = left;
   n->right = right;
   return n;
}

void print_ast(Node *n) {
   if (!n) return;
   if (n->type == NUM) {
      printf("%d", n->value);
```