Green University of Bangladesh
Department of Computer Science and Engineering (CSE)

Faculty of Sciences and Engineering
Semester: (Fall, Year:2021), B.Sc. in CSE (Day)

Course Title: Data Communication Lab
Course Code: CSE 308 Section: 193D

Lab Project Name: IPv4 to IPv6 Converter

Student Details

Name		ID
1.	Mohammad Nazmul hossain	193902031
2.	Jonaed Hasan	193902029

Submission Date: 05. 01. 2022
Course Teacher's Name: Amena Zahan

[For Teachers use only: Don't Write Anything inside this box]

Lab Project Status

Marks: …… **Signature: .....**

Comments: ..... **Date: .....**

Code file

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible"
content="IE=edge">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <!-- *Bootstrap* - 5 -->
  <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dis
t/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94Wr
HftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">
  <link rel="stylesheet" href="./style.css">
  <title>IPv4 to IPv6</title>

</head>
<body>
  <div class="container d-flex justify-content-center
align-items-center " style="height: 900px;">
    <div class="rounded-3 shadow p-3 mb-5 bg-body
rounded " >
      <div class="row">
        <div class="col-md-6  d-flex
justify-content-center  align-items-center">
          <div>
           <img class="img-fluid"
src="./undraw_Programming_re_kg9v.png" alt=""
srcset="">
```

```
          </div>
        </div>
        <div class="col-md-6 justify-content-center
d-flex align-items-center">
          <form class="container-fluid"
onsubmit="handlesubmit(event)">
            <div class=" mb-3">
              <label for="exampleInputText1"
class="form-label">Input any IPv4 address</label>
              <input type="Text" class="form-control"
name="ip" id="exampleInputText1"
aria-describedby="TextHelp">
              <div id="TextHelp"
class="form-text">We'll convert it to IPv6</div>
            </div>
            <div class="text-start mb-3">
              <button type="submit" class="btn
btn-primary">Submit</button>

            </div>

            <div class="form-floating mb-3">
              <textarea class="form-control"
placeholder="Output as IPv-6 address"
id="floatingTextarea"></textarea>
              <label for="floatingTextarea">IPv4 to
IPv6</label>
            </div>
          </form>
        </div>
```

```
      </div>
    </div>
  </div>
  <script src="./app.js"></script>
  <script>
    const handlesubmit = (e) => {
      e.preventDefault();
      const ip =
document.querySelector('input[name="ip"]').value;
      const output =
document.querySelector('#floatingTextarea');
      const ipv4 = ipaddr.parse(ip);
      const ipv6 = ipv4.toIPv4MappedAddress();
      output.value = ipv6.toString();
    }

  </script>
  <!-- ?bootstrap5 -->
  <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist
/js/bootstrap.bundle.min.js"
integrity="sha384-MrcW6ZMFYlzcLA8Nl+NtUVF0sA7MsXsP1UyJ
oMp4YLEuNSfAP+JcXn/tWtIaxVXM"
crossorigin="anonymous"></script>
</body>
</html>
```

```
(function (root) {
    "use strict";
    // A list of regular expressions that match
arbitrary IPv4 addresses,
    // for which a number of weird notations exist.
    // Note that an address like 0010.0xa5.1.1 is
considered legal.
    const ipv4Part = "(0?\\d+|0x[a-f0-9]+)";
    const ipv4Regexes = {
      fourOctet: new RegExp(

`^${ipv4Part}\\.${ipv4Part}\\.${ipv4Part}\\.${ipv4Part
}$`,
        "i"
      ),
      threeOctet: new
RegExp(`^${ipv4Part}\\.${ipv4Part}\\.${ipv4Part}$`,
"i"),
      twoOctet: new
RegExp(`^${ipv4Part}\\.${ipv4Part}$`, "i"),
      longValue: new RegExp(`^${ipv4Part}$`, "i"),
    };

    // Regular Expression for checking Octal numbers
    const octalRegex = new RegExp(`^0[0-7]+$`, "i");
    const hexRegex = new RegExp(`^0x[a-f0-9]+$`, "i");

    const zoneIndex = "%[0-9a-z]{1,}";

    // IPv6-matching regular expressions.
```

```
    // For IPv6, the task is simpler: it is enough to
match the colon-delimited
    // hexadecimal IPv6 and a transitional variant
with dotted-decimal IPv4 at
    // the end.
    const ipv6Part = "(?:[0-9a-f]+::?)+";
    const ipv6Regexes = {
      zoneIndex: new RegExp(zoneIndex, "i"),
      native: new RegExp(

`^(::)?(${ipv6Part})?([0-9a-f]+)?(::)?(${zoneIndex})?$
`,
        "i"
      ),
      deprecatedTransitional: new RegExp(

`^(?:::)(${ipv4Part}\\.${ipv4Part}\\.${ipv4Part}\\.${i
pv4Part}(${zoneIndex})?)$`,
        "i"
      ),
      transitional: new RegExp(

`^((?:${ipv6Part})|(?:::)(?:${ipv6Part})?)${ipv4Part}\
\.${ipv4Part}\\.${ipv4Part}\\.${ipv4Part}(${zoneIndex}
)?$`,
        "i"
      ),
    };

    // Expand :: in an IPv6 address or address part
consisting of `parts` groups.
```

```
    function expandIPv6(string, parts) {
      // More than one '::' means invalid adddress
      if (string.indexOf("::") !==
string.lastIndexOf("::")) {
        return null;
      }

      let colonCount = 0;
      let lastColon = -1;
      let zoneId =
(string.match(ipv6Regexes.zoneIndex) || [])[0];
      let replacement, replacementCount;

      // Remove zone index and save it for later
      if (zoneId) {
        zoneId = zoneId.substring(1);
        string = string.replace(/%.+$/, "");
      }

      // How many parts do we already have?
      while ((lastColon = string.indexOf(":",
lastColon + 1)) >= 0) {
        colonCount++;
      }

      // 0::0 is two parts more than ::
      if (string.substr(0, 2) === "::") {
        colonCount--;
      }

      if (string.substr(-2, 2) === "::") {
```

```
        colonCount--;
      }

      // The following loop would hang if colonCount >
parts
      if (colonCount > parts) {
        return null;
      }

      // replacement = ':' + '0:' * (parts -
colonCount)
      replacementCount = parts - colonCount;
      replacement = ":";
      while (replacementCount--) {
        replacement += "0:";
      }

      // Insert the missing zeroes
      string = string.replace("::", replacement);

      // Trim any garbage which may be hanging around
if :: was at the edge in
      // the source strin
      if (string[0] === ":") {
        string = string.slice(1);
      }

      if (string[string.length - 1] === ":") {
        string = string.slice(0, -1);
      }
```

```
      parts = (function () {
        const ref = string.split(":");
        const results = [];

        for (let i = 0; i < ref.length; i++) {
          results.push(parseInt(ref[i], 16));
        }

        return results;
      })();

      return {
        parts: parts,
        zoneId: zoneId,
      };
    }

    function parseIntAuto(string) {
      // Hexadedimal base 16 (0x#)
      if (hexRegex.test(string)) {
        return parseInt(string, 16);
      }
      // While octal representation is discouraged by
ECMAScript 3
      // and forbidden by ECMAScript 5, we silently
allow it to
      // work only if the rest of the string has
numbers less than 8.
      if (string[0] === "0" &&
!isNaN(parseInt(string[1], 10))) {
        if (octalRegex.test(string)) {
```

```
          return parseInt(string, 8);
        }
        throw new Error(`ipaddr: cannot parse
${string} as octal`);
      }
      // Always include the base 10 radix!
      return parseInt(string, 10);
    }

    const ipaddr = {};

    // An IPv4 address (RFC791).
    ipaddr.IPv4 = (function () {
      // Constructs a new IPv4 address from an array
of four octets
      // in network order (MSB first)
      // Verifies the input.
      function IPv4(octets) {
        if (octets.length !== 4) {
          throw new Error("ipaddr: ipv4 octet count
should be 4");
        }

        let i, octet;

        for (i = 0; i < octets.length; i++) {
          octet = octets[i];
          if (!(0 <= octet && octet <= 255)) {
            throw new Error("ipaddr: ipv4 octet should
fit in 8 bits");
          }
```

```
        }

        this.octets = octets;
      }
      // Returns the address in convenient,
decimal-dotted format.
      IPv4.prototype.toString = function () {
        return this.octets.join(".");
      };

      // Converts this IPv4 address to an IPv4-mapped
IPv6 address.
      IPv4.prototype.toIPv4MappedAddress = function ()
{
        return
ipaddr.IPv6.parse(`::ffff:${this.toString()}`);
      };

      return IPv4;
    })();

    // Checks if a given string is a valid IPv4
address.
    ipaddr.IPv4.isValid = function (string) {
      try {
        new this(this.parser(string));
        return true;
      } catch (e) {
        return false;
      }
    };
```

```
    // Tries to parse and validate a string with IPv4
address.
    // Throws an error if it fails.
    ipaddr.IPv4.parse = function (string) {
      const parts = this.parser(string);

      if (parts === null) {
        throw new Error("ipaddr: string is not
formatted like an IPv4 Address");
      }

      return new this(parts);
    };

    // Class-full variants (like a.b, where a is an
octet, and b is a 24-bit
    // value representing last three octets; this
corresponds to a class C
    // address) are omitted due to classless nature of
modern Internet.
    ipaddr.IPv4.parser = function (string) {
      let match, part, value;
      // parseInt recognizes all that octal &
hexadecimal weirdness for us
      if ((match =
string.match(ipv4Regexes.fourOctet))) {
        return (function () {
          const ref = match.slice(1, 6);
          const results = [];
```

```
          for (let i = 0; i < ref.length; i++) {
            part = ref[i];
            results.push(parseIntAuto(part));
          }

          return results;
        })();
      } else if ((match =
string.match(ipv4Regexes.longValue))) {
        value = parseIntAuto(match[1]);
        if (value > 0xffffffff || value < 0) {
          throw new Error("ipaddr: address outside
defined range");
        }

        return (function () {
          const results = [];
          let shift;

          for (shift = 0; shift <= 24; shift += 8) {
            results.push((value >> shift) & 0xff);
          }

          return results;
        })().reverse();
      } else if ((match =
string.match(ipv4Regexes.twoOctet))) {
        return (function () {
          const ref = match.slice(1, 4);
          const results = [];
```

```
          value = parseIntAuto(ref[1]);
          if (value > 0xffffff || value < 0) {
            throw new Error("ipaddr: address outside
defined range");
          }

          results.push(parseIntAuto(ref[0]));
          results.push((value >> 16) & 0xff);
          results.push((value >> 8) & 0xff);
          results.push(value & 0xff);

          return results;
        })();
      } else if ((match =
string.match(ipv4Regexes.threeOctet))) {
        return (function () {
          const ref = match.slice(1, 5);
          const results = [];

          value = parseIntAuto(ref[2]);
          if (value > 0xffff || value < 0) {
            throw new Error("ipaddr: address outside
defined range");
          }

          results.push(parseIntAuto(ref[0]));
          results.push(parseIntAuto(ref[1]));
          results.push((value >> 8) & 0xff);
          results.push(value & 0xff);

          return results;
```

```
        })();
      } else {
        return null;
      }
    };

    // An IPv6 address (RFC2460)
    ipaddr.IPv6 = (function () {
      // Constructs an IPv6 address from an array of
eight 16 - bit parts
      // or sixteen 8 - bit parts in network order(MSB
first).
      // Throws an error if the input is invalid.
      function IPv6(parts, zoneId) {
        let i, part;

        if (parts.length === 16) {
          this.parts = [];
          for (i = 0; i <= 14; i += 2) {
            this.parts.push((parts[i] << 8) | parts[i
+ 1]);
          }
        } else if (parts.length === 8) {
          this.parts = parts;
        } else {
          throw new Error("ipaddr: ipv6 part count
should be 8 or 16");
        }

        for (i = 0; i < this.parts.length; i++) {
          part = this.parts[i];
```

```
          if (!(0 <= part && part <= 0xffff)) {
            throw new Error("ipaddr: ipv6 part should
fit in 16 bits");
          }
        }

        if (zoneId) {
          this.zoneId = zoneId;
        }
      }

      // Returns the address in expanded format with
all zeroes included, like
      // 2001:db8:8:66:0:0:0:1
      IPv6.prototype.toNormalizedString = function ()
{
        const addr = function () {
          const results = [];

          for (let i = 0; i < this.parts.length; i++)
{
            results.push(this.parts[i].toString(16));
          }

          return results;
        }
          .call(this)
          .join(":");

        let suffix = "";
```

```
        if (this.zoneId) {
          suffix = `%${this.zoneId}`;
        }

        return addr + suffix;
      };

      // Returns the address in compact,
human-readable format like
      // 2001:db8:8:66::1
      IPv6.prototype.toString = function () {
        // Replace the first sequence of 1 or more '0'
parts with '::'
        return
this.toNormalizedString().replace(/((^|:)(0(:|$))+)/,
"::");
      };
      // Converts this address to IPv4 address if it
is an IPv4-mapped IPv6 address.
      // Throws an error otherwise.
      IPv6.prototype.toIPv4Address = function () {
        if (!this.isIPv4MappedAddress()) {
          throw new Error(
            "ipaddr: trying to convert a generic ipv6
address to ipv4"
          );
        }

        const ref = this.parts.slice(-2);
        const high = ref[0];
        const low = ref[1];
```

```
        return new ipaddr.IPv4([high >> 8, high &
0xff, low >> 8, low & 0xff]);
      };

      return IPv6;
    })();

    // Checks to see if string is a valid IPv6 Address
    ipaddr.IPv6.isValid = function (string) {
      // Since IPv6.isValid is always called first,
this shortcut
      // provides a substantial performance gain.
      if (typeof string === "string" &&
string.indexOf(":") === -1) {
        return false;
      }

      try {
        const addr = this.parser(string);
        new this(addr.parts, addr.zoneId);
        return true;
      } catch (e) {
        return false;
      }
    };

    // Tries to parse and validate a string with IPv6
address.
    // Throws an error if it fails.
    ipaddr.IPv6.parse = function (string) {
```

```
      const addr = this.parser(string);

      if (addr.parts === null) {
        throw new Error("ipaddr: string is not
formatted like an IPv6 Address");
      }

      return new this(addr.parts, addr.zoneId);
    };

    // Parse an IPv6 address.
    ipaddr.IPv6.parser = function (string) {
      let addr, i, match, octet, octets, zoneId;

      if ((match =
string.match(ipv6Regexes.deprecatedTransitional))) {
        return this.parser(`::ffff:${match[1]}`);
      }
      if (ipv6Regexes.native.test(string)) {
        return expandIPv6(string, 8);
      }
      if ((match =
string.match(ipv6Regexes.transitional))) {
        zoneId = match[6] || "";
        addr = expandIPv6(match[1].slice(0, -1) +
zoneId, 6);
        if (addr.parts) {
          octets = [
            parseInt(match[2]),
            parseInt(match[3]),
            parseInt(match[4]),
```

```
            parseInt(match[5]),
          ];
          for (i = 0; i < octets.length; i++) {
            octet = octets[i];
            if (!(0 <= octet && octet <= 255)) {
              return null;
            }
          }

          addr.parts.push((octets[0] << 8) |
octets[1]);
          addr.parts.push((octets[2] << 8) |
octets[3]);
          return {
            parts: addr.parts,
            zoneId: addr.zoneId,
          };
        }
      }

      return null;
    };

    // Attempts to parse an IP Address.
    // Throws an error if it could not be parsed.
    ipaddr.parse = function (string) {
      if (ipaddr.IPv4.isValid(string)) {
        return ipaddr.IPv4.parse(string);
      } else {
        throw new Error("ipaddr: the address has
neither IPv6 nor IPv4 format");
```

```
      }
    };

    // Export for both the CommonJS and browser-like
environment
    if (typeof module !== "undefined" &&
module.exports) {
      module.exports = ipaddr;
    } else {
      root.ipaddr = ipaddr;
    }
  })(this);
```