Job No: 09

Job Name: Write a program to eliminate left factoring from a production of a grammar.

Theory:

Left Factoring is a grammar transformation technique. It consists in "factoring out" prefixes which are common to two or more productions.

For example, A -> α β | α γ
After left factoring:
A -> α A'
A' -> β | γ

Code:

```
let nonTerminal = 'S';
let productions = 'iEtS|iEtSeS|a|iES';
console.log(`The given grammer is: ${nonTerminal} --> ${productions}`);

const LeftFactoring = (productions) => {
  let words = productions.split('|').filter(e => e!=='');
  if(words.length>1){
    const counts = words.reduce((acc, w) => {
      prefix = ''
      for (i = 0; i < w.length; i++) {
        prefix += w[i]
        acc[prefix] = (acc[prefix] || 0) + 1;
      }
      return acc;
    }, {});

    try {
      const alpha = Object.entries(counts)
      .filter(([_, v]) => v > words.length / 2.0)
      .sort((a, b) => b[0].length - a[0].length)[0][0];

      let nonTerminalPrime = '';
      words.filter(word => word.includes(alpha))
                    .map(word => word.replace(alpha,''))
                    .filter(e => e!=='')
```

```
                    .forEach(e => nonTerminalPrime += e+'|');
      let gama = words.filter(word => !word.includes(alpha));
      console.log(`${nonTerminal} -->
                  ${alpha}${nonTerminal}'${gama?`|${gama}`:''}`)
      console.log(`${nonTerminal}' --> ${nonTerminalPrime}#`);

      nonTerminal = `${nonTerminal}'`;
      LeftFactoring(nonTerminalPrime);
    } catch {
      console.log('...................');
    }
  }
}

console.log(`After Left Factoring:`);
LeftFactoring(productions);
```

Input/Output:

```
The given grammer is: S --> iEtS|iEtSeS|a|iES
After Left Factoring:
S --> iES'|a
S' --> tS|tSeS|S|#
S' --> tSS''|S
S'' --> eS|#
```

Job No: 09

Job Name: **Write a program to eliminate left factoring from a production of a grammar.**

Theory:

Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers.

For example, A -> α β | α γ
After left factoring:
A -> α A'
A' -> β | γ

Code:

```
let nonTerminal = "S";
let input = "bSSaaS|bSSaSb|a|bSb";
console.log(`The given grammer is: ${nonTerminal} --> ${input}`);

const DoLeftFactor = (input) => {
  let words = input.split("|").filter((e) => e !== "");
  if (words.length > 1) {
    const counts = words.reduce((acc, w) => {
      prefix = "";
      for (i = 0; i < w.length; i++) {
        prefix += w[i];
        acc[prefix] = (acc[prefix] || 0) + 1;
      }
      return acc;
    }, {});
    try {
      const alpha = Object.entries(counts)
        .filter(([_, v]) => v > words.length / 2.0)
        .sort((a, b) => b[0].length - a[0].length)[0][0];
      let nonTerminalPrime = "";
      words
        .filter((word) => word.includes(alpha))
        .map((word) => word.replace(alpha, ""))
        .filter((e) => e !== "")
        .forEach((e) => (nonTerminalPrime += e + "|"));
      let gama = words.filter((word) => !word.includes(alpha));
```

```
      console.log(`${nonTerminal} --> ${alpha}${nonTerminal}'${gama ?
                  `|${gama}` : ""}`);
      console.log(`${nonTerminal}' --> ${nonTerminalPrime}#`);
      nonTerminal = `${nonTerminal}'`;
      DoLeftFactor(nonTerminalPrime);
    } catch {
      console.log("...................");
    }
  }
};

console.log(`After Left Factoring:`);
DoLeftFactor(input);
```

Input/Output:

```
The given grammer is: S --> bSSaaS|bSSaSb|a|bSb
After Left Factoring:
S --> bSS'|a
S' --> SaaS|SaSb|b|#
S' --> SaS''|b
S'' --> aS|Sb|#
...................
```