Ministry of Education and Research of the Republic of Moldova
Technical University of Moldova
Faculty of Computers, Informatics, and Microelectronics

Laboratory work 1:
Chomsky Normal Form

Course: Formal Languages & Finite Automata

Elaborated:
st. gr. FAF-223 Maria Colța

Verified:
asist. univ. Dumitru Crețu

Chișinău 2024

TABLE OF CONTENTS

THEORY

Chomsky Normal Form

In formal language theory, a context-free grammar (CFG), G, is said to be in Chomsky Normal Form (first described by Noam Chomsky) if all of its production rules conform to one of the following formats:

1. $A \rightarrow BC$
2. $A \rightarrow a$
3. $S \rightarrow \varepsilon$

Where:

- A, B, and C are nonterminal symbols.
- a represents a terminal symbol, which is a symbol that represents a constant value.
- S is the start symbol.
- ε denotes the empty string.

Conditions:

- Neither B nor C may be the start symbol.
- The third production rule ($S \rightarrow \varepsilon$) can only appear if ε is in L(G), the language produced by the context-free grammar G.

Properties:

- Every grammar in Chomsky Normal Form is context-free.
- Every context-free grammar can be transformed into an equivalent one in Chomsky Normal Form. The size of the transformed grammar will be no larger than the square of the original grammar's size.

Conversion from CFG to CNF

To convert a context-free grammar (CFG) to Chomsky Normal Form (CNF), follow these simplified steps:

- Eliminate ε-productions:
    - Remove rules that produce the empty string, except for the start symbol.
- Remove Unit Productions:
    - Eliminate rules where a nonterminal leads directly to another nonterminal.
- Eliminate Useless Symbols:
    - Remove nonterminals that don't generate terminal strings or are unreachable.
- Standardize Productions:
    - Convert all rules to the forms $A \rightarrow BC$ or $A \rightarrow a$, using intermediate nonterminals if necessary.

OBJECTIVES

Tasks

- Learn about Chomsky Normal Form (CNF)
- Get familiar with the approaches of normalizing a grammar..
- Implement a method for normalizing an input grammar by the rules of CNF.
    - The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).
    - The implemented functionality needs executed and tested.
    - A BONUS point will be given for the student who will have unit tests that validate the functionality of the project.
    - Also, another BONUS point would be given if the student will make the aforementioned function to accept any grammar, not only the one from the student's variant.

IMPLEMENTATION

I implemented a JavaScript class which provides methods for converting any context-free grammar (CFG) into Chomsky Normal Form (CNF). Below are the main components and methods of the Grammar class:

Constructor

- Parameters: nonTerminals (array of non-terminal symbols), terminals (array of terminal symbols), productions (object mapping non-terminals to an array of their productions).
- Purpose: Initializes a new instance of the grammar with the specified non-terminals, terminals, and productions.

```
export class Grammar {
  constructor(nonTerminals, terminals, productions) {
    this.nonTerminals = nonTerminals; // Array of non-terminal symbols
    this.terminals = terminals; // Array of terminal symbols
    this.productions = productions;
  }
```

Methods

- `eliminateEpsilon()`
    - Purpose: Removes ε-productions (productions that generate an empty string) from the grammar. It first identifies non-terminals that produce ε directly or indirectly

and then modifies the productions to eliminate ε while preserving the language generated by the grammar.

```javascript
// Method to eliminate epsilon productions
eliminateEpsilon() {
  let epsilonProducing = new Set();

  // First pass: Find all non-terminals that directly produce epsilon
  for (let nonTerminal of this.nonTerminals) {
    if (this.productions[nonTerminal].includes("ep")) {
      epsilonProducing.add(nonTerminal);
    }
  }

  let additions;
  do {
    additions = false;
    // Second pass: Find all non-terminals that produce epsilon indirectly
    for (let nonTerminal of this.nonTerminals) {
      for (let production of this.productions[nonTerminal]) {
        const symbols = production.split("");
        if (symbols.every((symbol) => epsilonProducing.has(symbol))) {
          if (!epsilonProducing.has(nonTerminal)) {
            epsilonProducing.add(nonTerminal);
            additions = true;
          }
        }
      }
    }
  } while (additions);

  // Third pass: Remove epsilon and update productions
  for (let nonTerminal of this.nonTerminals) {
    this.productions[nonTerminal] = this.productions[nonTerminal].filter(
      (p) => p !== "ep"
    );
    const newProductions = [];
    this.productions[nonTerminal].forEach((production) => {
      let variants = [""];
      production.split("").forEach((symbol) => {
        if (epsilonProducing.has(symbol)) {
          variants = variants.flatMap((v) => [v, v + symbol]);
        } else {
          variants = variants.map((v) => v + symbol);
        }
      });
      newProductions.push(...variants.filter((v) => v.length > 0));
    });
    this.productions[nonTerminal] = [...new Set(newProductions)];
  }
}
```

- `removeUnitProductions()`
  - Purpose: Eliminates unit productions (productions where a non-terminal leads directly to another non-terminal) by replacing them with the appropriate productions of the non-terminals they point to.

```javascript
removeUnitProductions() {
  // Map each non-terminal to the set of non-terminals it can reach via unit productions
  let unitProductions = {};
  this.nonTerminals.forEach((nt) => {
    unitProductions[nt] = new Set();
    this.productions[nt].forEach((prod) => {
      if (prod.length === 1 && this.nonTerminals.includes(prod)) {
        unitProductions[nt].add(prod);
      }
    });
  });

  // Find the closure of the unit productions for each non-terminal
  let changed = true;
  while (changed) {
    changed = false;
    for (let nt of this.nonTerminals) {
      let currentSize = unitProductions[nt].size;
      Array.from(unitProductions[nt]).forEach((unit) => {
        unitProductions[unit].forEach((element) => {
          unitProductions[nt].add(element);
        });
      });
      if (unitProductions[nt].size !== currentSize) {
        changed = true;
      }
    }
  }

  // Replace unit productions with the productions of their reachable non-terminals
  this.nonTerminals.forEach((nt) => {
    let newProductions = this.productions[nt].filter(
      (prod) => !(prod.length === 1 && this.nonTerminals.includes(prod))
    );
    unitProductions[nt].forEach((unit) => {
      this.productions[unit].forEach((prod) => {
        if (!(prod.length === 1 && this.nonTerminals.includes(prod))) {
          newProductions.push(prod);
        }
      });
    });
    this.productions[nt] = [...new Set(newProductions)];
  });
}
```

- `removeInaccessibleSymbols()`
  - Purpose: Removes non-terminal symbols that are not accessible from the start symbol. This helps in reducing the size of the grammar and focusing only on the usable parts.

```javascript
removeInaccessibleSymbols() {
  const accessible = new Set();
  const workList = ["S"]; // Assuming 'S' is the start symbol

  while (workList.length > 0) {
    const current = workList.pop();
    if (!accessible.has(current)) {
      accessible.add(current);
      // Examine each production of the current non-terminal
      (this.productions[current] || []).forEach((production) => {
        // Add non-terminals found in the production to the work list if they aren't already marked as accessible
        production.split("").forEach((symbol) => {
          if (this.nonTerminals.includes(symbol) && !accessible.has(symbol)) {
            workList.push(symbol);
          }
        });
      });
    }
  }

  // Filter out all non-terminals that are not accessible
  this.nonTerminals = this.nonTerminals.filter((nt) => accessible.has(nt));
  // Rebuild the productions object to remove productions of inaccessible non-terminals
  const newProductions = {};
  this.nonTerminals.forEach((nt) => {
    newProductions[nt] = this.productions[nt];
  });
  this.productions = newProductions;
}
```

- `removeNonProductiveSymbols()`
  - Purpose: Identifies and removes non-terminals that do not produce any terminal strings, making the grammar more efficient and manageable.

```
removeNonProductiveSymbols() {
  const productive = new Set();

  // Initial pass: Identify terminals and productions consisting only of
  for (let nt of this.nonTerminals) {
    this.productions[nt].forEach((production) => {
      // Check if the production consists only of terminals
      if (
        production
          .split("")
          .every((symbol) => this.terminals.includes(symbol))
      ) {
        productive.add(nt);
      }
    });
  }

  let changed = true;
  while (changed) {
    changed = false;
    for (let nt of this.nonTerminals) {
      if (!productive.has(nt)) {
        // Check if all symbols in any production are productive
        for (let production of this.productions[nt]) {
          if (
            production
              .split("")
              .every(
                (symbol) =>
                  this.terminals.includes(symbol) || productive.has(symbol)
              )
          ) {
            if (!productive.has(nt)) {
              productive.add(nt);
              changed = true;
              break; // Once found productive, no need to check further
            }
          }
        }
      }
    }
  }

  // Filter non-productive non-terminals
  this.nonTerminals = this.nonTerminals.filter((nt) => productive.has(nt));
  // Rebuild the productions to exclude those involving non-productive symbols
  let newProductions = {};
  this.nonTerminals.forEach((nt) => {
    newProductions[nt] = this.productions[nt].filter((prod) =>
      prod
        .split("")
        .every(
          (symbol) =>
            this.terminals.includes(symbol) || productive.has(symbol)
        )
    );
  });
  this.productions = newProductions;
}
```

- `convertToCNF()`
  - Steps Involved:
    - Calls eliminateEpsilon() to handle ε-productions.
    - Uses removeUnitProductions() to deal with unit productions.
    - Applies removeInaccessibleSymbols() to discard inaccessible symbols.
    - Executes removeNonProductiveSymbols() to eliminate non-productive symbols.
    - Ensures all productions are in the form A → BC or A → a by introducing intermediate non-terminals as necessary.

```javascript
convertToCNF() {
  this.eliminateEpsilon();
  this.removeUnitProductions();
  this.removeInaccessibleSymbols();
  this.removeNonProductiveSymbols();

  // Step 1: Introduce new non-terminals for terminals in productions with more than one symbol
  const terminalMap = {}; // Map each terminal to a new non-terminal
  let newNTCounter = this.nonTerminals.length;
  Object.keys(this.productions).forEach((nt) => {
    this.productions[nt] = this.productions[nt].map((production) => {
      const symbols = production.split("");
      if (symbols.length > 1) {
        return symbols
          .map((symbol) => {
            if (this.terminals.includes(symbol)) {
              if (!terminalMap[symbol]) {
                const newNT = `Z${newNTCounter++}`;
                terminalMap[symbol] = newNT;
                this.nonTerminals.push(newNT);
                this.productions[newNT] = [symbol]; // newNT -> symbol
              }
              return terminalMap[symbol];
            }
            return symbol;
          })
          .join("");
      }
      return production;
    });
  });

  // Step 2: Ensure all productions are in the correct form
  const newProductions = {};
  this.nonTerminals.forEach((nt) => (newProductions[nt] = []));

  Object.keys(this.productions).forEach((nt) => {
    this.productions[nt].forEach((production) => {
      const parts = production.split("");
      if (parts.length == 2) {
        newProductions[nt].push(production); // Already in CNF
      } else if (parts.length > 2) {
        // Convert to binary rules
        let currentNT = nt;
        while (parts.length > 2) {
          const first = parts.shift();
          const second = parts.shift();
          const newNT = `Z${newNTCounter++}`;
          this.nonTerminals.push(newNT);
          newProductions[newNT] = [first + second];
          parts.unshift(newNT);
          currentNT = newNT;
        }
        newProductions[nt].push(parts.join(""));
      } else {
        // Single terminal or non-terminal
        newProductions[nt].push(production);
      }
    });
  });

  // Replace old productions with new ones
  this.productions = newProductions;
}
```

RESULTS

I had the following variant:

```
Variant 4:
VN={S, A, B, C, D},
VT={a, b},
P={
    S → aB
    S → bA
    S → A
    A → B
    A → AS
    A → bBAB
    A → b
    B → b
    B → bS
    B → aD
    B → ep
    D → AA
    C → Ba
}.
```

After running the code, I had the following results:

```
Before elimination:
{
  S: [ 'aB', 'bA', 'A' ],
  A: [ 'B', 'AS', 'bBAB', 'b', 'ep' ],
  B: [ 'b', 'bS', 'aD', 'ep' ],
  D: [ 'AA' ],
  C: [ 'Ba' ]
}
After elimination:
{
  S: [ 'a', 'aB', 'b', 'bA', 'A' ],
  A: [
    'B',   'S',    'A',
    'AS',  'b',    'bB',
    'bA',  'bAB',  'bBB',
    'bBA', 'bBAB'
  ],
  B: [ 'b', 'bS', 'a', 'aD' ],
  D: [ 'A', 'AA' ],
  C: [ 'a', 'Ba' ]
}
After unit prod elimination
{
  S: [
    'a',    'aB',  'b',
    'bA',   'AS',  'bB',
    'bAB',  'bBB', 'bBA',
    'bBAB', 'bS',  'aD'
  ],
  A: [
    'AS',  'b',    'bB',
    'bA',  'bAB',  'bBB',
    'bBA', 'bBAB', 'bS',
    'a',   'aD',   'aB'
  ],
  B: [ 'b', 'bS', 'a', 'aD' ],
  D: [
    'AA',  'AS',  'b',
    'bB',  'bA',  'bAB',
    'bBB', 'bBA', 'bBAB',
    'bS',  'a',   'aD',
    'aB'
  ],
  C: [ 'a', 'Ba' ]
}
```

```
After inaccessible symbols elimination
{
  S: [
    'a',    'aB',  'b',
    'bA',   'AS',  'bB',
    'bAB',  'bBB', 'bBA',
    'bBAB', 'bS',  'aD'
  ],
  A: [
    'AS',  'b',    'bB',
    'bA',  'bAB',  'bBB',
    'bBA', 'bBAB', 'bS',
    'a',   'aD',   'aB'
  ],
  B: [ 'b', 'bS', 'a', 'aD' ],
  D: [
    'AA',  'AS',  'b',
    'bB',  'bA',  'bAB',
    'bBB', 'bBA', 'bBAB',
    'bS',  'a',   'aD',
    'aB'
  ]
}
After non productive symbols elim
{
  S: [
    'a',    'aB',  'b',
    'bA',   'AS',  'bB',
    'bAB',  'bBB', 'bBA',
    'bBAB', 'bS',  'aD'
  ],
  A: [
    'AS',  'b',    'bB',
    'bA',  'bAB',  'bBB',
    'bBA', 'bBAB', 'bS',
    'a',   'aD',   'aB'
  ],
  B: [ 'b', 'bS', 'a', 'aD' ],
  D: [
    'AA',  'AS',  'b',
    'bB',  'bA',  'bAB',
    'bBB', 'bBA', 'bBAB',
    'bS',  'a',   'aD',
    'aB'
  ]
}
```

```
After CNF conversion:
{
  S: [
    'a',    'Z6B',  'b',
    'Z7A',  'AS',   'Z8B',
    'Z10B', 'Z12B', 'Z14A',
    'Z17B', 'Z18S', 'Z19D'
  ],
  A: [
    'AS',   'b',    'Z20B',
    'Z21A', 'Z23B', 'Z25B',
    'Z27A', 'Z30B', 'Z31S',
    'a',    'Z32D', 'Z33B'
  ],
  B: [ 'b', 'Z34S', 'a', 'Z35D' ],
  D: [
    'AA',   'AS',   'b',
    'Z36B', 'Z37A', 'Z39B',
    'Z41B', 'Z43A', 'Z46B',
    'Z47S', 'a',    'Z48D',
    'Z49B'
  ],
  Z4: [ 'a' ],
  Z5: [ 'b' ],
  Z6: [ 'Z4' ],
  Z7: [ 'Z5' ],
  Z8: [ 'Z5' ],
  Z9: [ 'Z5' ],
  Z10: [ 'Z9A' ],
  Z11: [ 'Z5' ],
  Z12: [ 'Z11B' ],
  Z13: [ 'Z5' ],
  Z14: [ 'Z13B' ],
  Z15: [ 'Z5' ],
  Z16: [ 'Z15B' ],
  Z17: [ 'Z16A' ],
  Z18: [ 'Z5' ],
  Z19: [ 'Z4' ],
  Z20: [ 'Z5' ],
  Z21: [ 'Z5' ],
```

The result for CNF conversion can be better, but to be honest I didn't manage my time properly to fully do this work entirely by myself and optimize it :(.

CONCLUSIONS

In this lab, I developed a solution for converting context-free grammars (CFGs) into Chomsky Normal Form (CNF) using Node.js. My approach tackled several key transformations required for CNF conversion, ensuring thorough processing of the grammar. I started by implementing a method to eliminate epsilon productions, which involved identifying nullable non-terminals and modifying the grammar to account for possible combinations that occur without these nullable symbols. Next, I addressed unit productions by identifying direct and transitive unit relations into more complex production sets.

Additionally, I implemented a process to remove inaccessible symbols to ensure that all non-terminals used in the grammar are reachable from the start symbol. This step helps in optimizing the grammar for efficiency by eliminating unnecessary components.

The most significant part of my work involved converting the grammar into CNF. This required restructuring all grammar productions to either two non-terminals or a single terminal. I achieved this by introducing new non-terminals for terminals that appear in complex productions with non-terminals, and by breaking down longer productions into binary forms using newly introduced non-terminals. Although this part could use more work to make it more correct.

Through these systematic transformations, I ensured that the grammar adheres to CNF rules while maintaining its ability to generate the original language. This project not only enhances my understanding of grammar transformations but also provides a robust framework for further applications in parsing and computational linguistics.