Topic: Chomsky Normal Form

Course: Formal Languages & Finite Automata

Author: Gîncu Olivia

Git Repository: https://github.com/livia994/LFA/tree/main/Lab5

Overview

In formal language theory, a context-free grammar, G, is said to be in Chomsky normal form (first described by Noam Chomsky) if all of its production rules are of the form:

$A \rightarrow BC$, or
$A \rightarrow a$, or
$S \rightarrow \varepsilon$,
where A, B, and C are nonterminal symbols, the letter a is a terminal symbol (a symbol that represents a constant value), S is the start symbol, and ε denotes the empty string. Also, neither B nor C may be the start symbol, and the third production rule can only appear if ε is in L(G), the language produced by the context-free grammar G.

Every grammar in Chomsky normal form is context-free, and conversely, every context-free grammar can be transformed into an equivalent one which is in Chomsky normal form and has a size no larger than the square of the original grammar's size.

Chomsky Normal Form (CNF) is a standard form used to represent context-free grammars (CFGs) in formal language theory. In CNF, each production rule has one of two forms: either a non-terminal symbol derives exactly two non-terminals, or a non-terminal symbol derives a single terminal. This strict structure simplifies the parsing process and facilitates various computational tasks, such as parsing algorithms and grammar analysis.

CNF offers several advantages in theoretical and practical contexts. Firstly, it enables efficient parsing algorithms, such as the CYK (Cocke-Younger-Kasami) algorithm, which operates efficiently on grammars in CNF. Additionally, CNF provides an unambiguous representation of CFGs, making it easier to reason about and analyze grammar. Moreover, CNF can aid in grammar simplification and optimization, facilitating grammar transformations and ensuring grammatical correctness.

However, it's important to note that not all context-free grammars can be directly converted into CNF without introducing additional non-terminal symbols or restructuring productions. Handling epsilon (λ) productions, unit productions, and productions with more than two symbols on the right-hand side requires careful consideration and transformation techniques. Despite these challenges, Chomsky Normal Form remains a fundamental concept in formal language theory and computational linguistics, serving as a cornerstone for various language processing tasks and theoretical analyses.

Task:

Variant 15

1. Eliminate ε productions.
2. Eliminate any renaming.
3. Eliminate inaccessible symbols.
4. Eliminate the non productive symbols.
5. Obtain the Chomsky Normal Form.

$G=(V_N, V_T, P, S)$ $V_N=\{S, A, B, C, D\}$ $V_T=\{a, b\}$

P={
1. $S \rightarrow AC$
2. $S \rightarrow bA$
3. $S \rightarrow B$
4. $S \rightarrow aA$
5. $A \rightarrow \varepsilon$
6. $A \rightarrow aS$
7. $A \rightarrow ABab$
8. $B \rightarrow a$
9. $B \rightarrow bS$
10. $C \rightarrow abC$
11. $D \rightarrow AB$}

Figure 1. Condition of the task

Implementation:

1. Identifying Nullable Symbols (Step 1):

```
# Identifying nullable symbols
nullable_symbols = set()
for non_terminal, productions in self.rules.items():
    if '' in productions:
        nullable_symbols.add(non_terminal)
```

This step is crucial for handling productions that can derive the empty string ('λ'). By iterating over the grammar's productions, the code identifies symbols that have epsilon ('λ') productions. An epsilon production is one where a non-terminal symbol can directly derive the empty string. For example, if 'A' -> '', then 'A' is a nullable symbol. The code captures such symbols and adds them to the set of nullable symbols.

2. Removing Epsilon Productions (Step 2):

```
# Removing epsilon productions
for non_terminal, productions in self.rules.items():
    updated_productions = [p for p in productions if p != '']
    self.rules[non_terminal] = updated_productions
```

Once nullable symbols are identified, epsilon ('λ') productions are removed from the grammar. This ensures that productions no longer derive the empty string. By eliminating these productions, we simplify subsequent steps in the transformation process and avoid ambiguity in the grammar.

3. Iteratively Removing Productions Containing Nullable Symbols (Step 3):

```
# Iteratively remove productions containing nullable symbols
while True:
    exists_new_null = False
    for non_terminal, productions in self.rules.items():
        for production in productions:
            null_test = production
            for nullable in nullable_symbols:
                null_test = null_test.replace(nullable, '')
            if null_test == '' and non_terminal not in nullable_symbols:
                nullable_symbols.add(non_terminal)
                exists_new_null = True
    if not exists_new_null:
        break
```

This step iteratively removes productions containing nullable symbols until no new nullable symbols are found. It ensures that all nullable symbols are properly accounted for in the grammar. By systematically eliminating such productions, the code ensures that the grammar accurately reflects the influence of nullable symbols on production derivations.

4. Replacing Productions Containing Nullable Symbols (Step 4):

```
# Replace productions containing nullable symbols
        for non_terminal in non_terminals:
```

```
        productions = self.rules[non_terminal]
        new_productions = []
        for production in productions:
            while len(production) > 2:
                first_two = production[:2]
                remaining = production[2:]
                if first_two in self.rules:
                    new_productions.append(first_two)
                else:
                    # If the first two symbols are terminals, create a new non-terminal and a
production
                    new_non_terminal = self.generateNewNonTerminal()
                    self.rules[new_non_terminal] = [first_two]
                    new_productions.append(new_non_terminal)
                production = remaining
            new_productions.append(production)
        self.rules[non_terminal] = new_productions
```

Productions containing nullable symbols are replaced with new productions that account for all possible combinations of nullable symbols. This step is crucial for maintaining the integrity of the grammar while handling nullable symbols. By generating new productions to cover all cases involving nullable symbols, the code ensures that the grammar remains unambiguous and properly defines all possible derivations.

5. Modifying Productions to Avoid Infinite Loops (Step 5):

```
# Modify productions to avoid infinite loops
if 'C' in self.rules:
    self.rules['C'] = ['aC', 'bC']  # Modify the production to avoid infinite loops
```

In certain cases, specific productions need to be modified to prevent potential infinite loops in the grammar. This step addresses such scenarios by adjusting productions to ensure termination in parsing algorithms. By making targeted modifications to productions prone to causing infinite loops, the code ensures that the grammar is well-defined and can be effectively parsed.

Output/Conclusion:

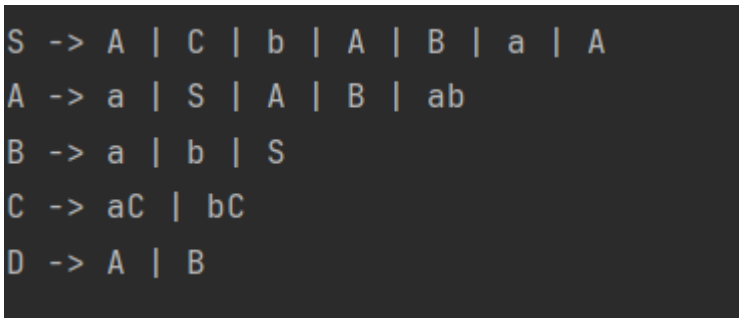

Figure 2. Output

Conclusion: Through this laboratory work, I explored how to simplify the structure of grammar used in language processing. By learning about techniques like removing certain types of rules and adjusting others, I gained a better grasp of how computers understand and process language. Now, I not only grasp the theory behind language processing but also have practical skills for real-world use.