Topic: Chomsky Normal Form

Course: Formal Languages & Finite Automata

Author: Cretu Dumitru and cudos to the Vasile Drumea with Irina Cojuhari

Author: Ungureanu Vlad

Overview

Context-free grammars (CFGs) are a fundamental concept in the field of formal languages and automata theory. They are used to describe the syntax of various formal languages, including programming languages and natural languages. A context-free grammar consists of a set of production rules that define how symbols (both terminal and non-terminal) can be combined to form strings in the language.

Chomsky Normal Form (CNF) is a specific form of context-free grammars that simplifies the analysis and manipulation of CFGs. In CNF, each production rule is either of the form A -> BC (where A, B, and C are non-terminal symbols) or A -> a (where A is a non-terminal symbol and a is a terminal symbol). Converting a CFG into CNF involves several steps, including removing epsilon productions, unit productions, and non-productive symbols, as well as ensuring that all productions have the correct form.

In this report, we will explore the process of converting a CFG into CNF. We will discuss each step of the conversion process in detail and provide Python code examples to illustrate how each step can be implemented. By the end of this report, readers will have a thorough understanding of CNF and the steps involved in converting a CFG into CNF, which is essential for anyone studying formal languages, automata theory, or compiler construction.

Introduction:

Context-free grammars (CFGs) are essential tools in the study of formal languages and computational theory. They are used to describe the syntax of programming languages, natural languages, and many other types of formal languages. A context-free grammar consists of a set of production rules that define how symbols (both terminal and non-terminal) can be combined to form strings in the language.

One important concept related to CFGs is Chomsky Normal Form (CNF), a specific form that simplifies the analysis and manipulation of CFGs. In CNF, each production rule is either of the form A -> BC (where A, B, and C are non-terminal symbols) or A -> a (where A is a non-terminal symbol and a is a terminal symbol). Converting a CFG into CNF involves several steps, including removing epsilon productions, unit productions, and non-productive symbols, as well as ensuring that all productions have the correct form.

In this report, we will explore the process of converting a CFG into CNF. We will discuss each step of the conversion process in detail and provide Python code examples to illustrate how each step can be implemented. By the end of this report, readers will have a thorough understanding of CNF and the steps involved in converting a CFG into CNF.

Objectives:

1. Learn about Chomsky Normal Form (CNF) [1].
2. Get familiar with the approaches of normalizing a grammar.
3. Implement a method for normalizing an input grammar by the rules of CNF.
    1. The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).
    2. The implemented functionality needs executed and tested.
    3. A **BONUS point** will be given for the student who will have unit tests that validate the functionality of the project.
    4. 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

Removing Epsilon Productions:

```
def remove_epsilon_productions(self):
    # Find nullable symbols
    nullable = set()
    for prod in self.P:
        if self.epsilon in self.P[prod]:
            nullable.add(prod)

    # Calculate the closure of nullable symbols
    while True:
        new_nullable = set()
        for prod in self.P:
            if all(ch in nullable for ch in self.P[prod]):
                new_nullable.add(prod)
        if new_nullable.issubset(nullable):
            break
        nullable |= new_nullable

    # Remove epsilon productions
    new_P = defaultdict(set)
    for prod in self.P:
        for null_set in self.powerset(self.P[prod]):
            if len(null_set) > 0 and len(null_set) < len(self.P[prod]):
                new_P[prod].add(''.join(null_set))
        if len(new_P[prod]) == 0:
            new_P[prod].add(self.epsilon)

    self.P = new_P
```

Explanation: This method removes epsilon (ε) productions from the grammar. It first identifies nullable symbols (symbols that can produce ε) and then calculates the closure of nullable symbols. It then removes ε productions by generating all possible combinations of nullable symbols in each production and adding them to the new set of productions.

Removing Unit Productions:

```
def remove_unit_productions(self):
    # Find unit productions
    unit_prods = defaultdict(set)
    for var in self.VN:
        for prod in self.P[var]:
            if len(prod) == 1 and prod in self.VN:
                unit_prods[var].add(prod)

    # Calculate the closure of unit productions
    while True:
        new_unit_prods = defaultdict(set)
        for var in unit_prods.copy():
            for unit_prod in unit_prods[var].copy():
                new_unit_prods[var] |= unit_prods[unit_prod]
        if all(new_unit_prods[var].issubset(unit_prods[var]) for var in unit_prods):
            break
        for var in new_unit_prods:
            unit_prods[var] |= new_unit_prods[var]

    # Remove unit productions
    new_P = defaultdict(set)
    for var in self.VN:
        for prod in self.P[var]:
            if len(prod) > 1 or prod not in self.VN:
                new_P[var].add(prod)
        for unit_prod in unit_prods[var]:
            new_P[var] |= self.P[unit_prod]

    self.P = new_P
```

Explanation: This method removes unit productions (productions of the form A -> B) from the grammar. It first identifies all unit productions and then calculates the closure of unit productions. It then removes unit productions by replacing them with the productions of the non-terminal they produce.

Removing Inaccessible Symbols:

```
def remove_inaccessible_symbols(self):
    # Find reachable symbols from the start symbol
    reachable = set()
    reachable.add(self.S)
    while True:
        new_reachable = set()
        for var in self.VN:
            if any(ch in reachable for prod in self.P[var] for ch in prod):
                new_reachable.add(var)
        if new_reachable.issubset(reachable):
            break
        reachable |= new_reachable

    # Remove unreachable symbols
    new_P = defaultdict(set)
    for var in reachable:
        new_P[var] = self.P[var]

    self.VN = reachable
    self.P = new_P
```

Explanation: This method removes symbols that are not reachable from the start symbol. It starts with the start symbol and iteratively adds symbols that can be reached from the current set of reachable symbols. It then removes symbols that are not reachable from the start symbol.

Removing Non-Productive Symbols:

```
def remove_non_productive_symbols(self):
    # Find productive symbols
    productive = set()
    for var in self.VN:
        if any(all(ch in productive or ch in self.VT for ch in prod) for prod in self.P[var]):
            productive.add(var)

    # Remove non-productive symbols
    new_P = defaultdict(set)
    for var in productive:
        new_P[var] = self.P[var]

    self.VN = productive
    self.P = new_P
```

Explanation: This method removes symbols that cannot produce any terminal string. It starts with the set of terminal symbols and iteratively adds symbols that can be derived from the current set of productive symbols. It then removes symbols that cannot produce any terminal string.

Converting to Chomsky Normal Form (CNF):

```
def convert_to_cnf(self):
    # Apply all conversion steps
    self.remove_epsilon_productions()
    self.remove_unit_productions()
    self.remove_inaccessible_symbols()
    self.remove_non_productive_symbols()

    # Convert the grammar to Chomsky Normal Form (CNF)
    new_P = defaultdict(set)
    new_VN = set()
    new_VT = set()
    for var in self.VN:
        for prod in self.P[var]:
            if len(prod) == 1 and prod in self.VT:
                new_P[var].add(prod)
                new_VT.add(prod)
            else:
                new_var = var
                for ch in prod:
                    if ch in self.VT:
                        new_var += ch
                    else:
                        new_non_term = ch
                        new_VN.add(new_non_term)
                        new_P[new_non_term].add(ch)
                        new_var += new_non_term
                new_P[var].add(new_var)

    self.VN = new_VN
    self.VT = new_VT
    self.P = new_P
```

Explanation: This method converts the grammar to Chomsky Normal Form (CNF). It first applies all the previous conversion steps. It then converts each production into a form where each production is either a terminal or a pair of non-terminals. It creates new non-terminals for non-terminal symbols in the original productions.

Conclusion:

The conversion of a CFG into Chomsky Normal Form (CNF) is a fundamental process in the study of formal languages and computational theory. CNF simplifies the analysis and manipulation of CFGs, making it easier to determine various properties of context-free languages. Through the systematic application of conversion steps, we can transform a CFG into CNF while preserving the language defined by the original CFG.

In this report, we have discussed each step of the CNF conversion process in detail and provided Python code examples to illustrate how these steps can be implemented. By following these steps, one can convert a CFG into CNF and gain a deeper understanding of the structure and properties of context-free languages.

Understanding CNF and its conversion process is essential for anyone studying formal languages, automata theory, or compiler construction. It provides a foundation for further study in these areas and enables one to analyze and manipulate CFGs more effectively.