Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 5:
CFG to CNF

Elaborated: st. gr. FAF-223 Friptu Ludmila

Verified: asist. univ. Cretu Dumitru

THEORY

1. Chomsky Normal Form

CNF stands for Chomsky normal form. A CFG(context free grammar) is in CNF(Chomsky normal form) if all production rules satisfy one of the following conditions:

- Start symbol generating ε. For example, A → ε.
- A non-terminal generating two non-terminals. For example, S → AB.
- A non-terminal generating a terminal symbol. For example, S → a.

The conversion to Chomsky Normal Form has four main steps:

1. Get rid of all ε productions.
2. Get rid of all productions where RHS is one variable.
3. Replace every production that is too long by shorter productions.
4. Move all terminals to productions where RHS is one terminal.

OBJECTIVES

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

First I defined the CFG class, which will manipulate context-free grammars, especially for converting them into Chomsky Normal Form (CNF). The class includes several methods to remove null productions, unit productions, and inaccessible symbols, before finally converting the grammar into CNF.

1. Class Initialization (__init__)

The constructor initializes a CFG object with a starting symbol and a dictionary of productions. The productions are converted into a set of tuples to avoid duplicates and ensure immutability.

```
class CFG:
    def __init__(self, start_symbol, productions):
        self.start_symbol = start_symbol
        self.productions = {nt: set(map(tuple, p)) for nt, p in productions.items()}
```

2. Method: remove_null_productions()

This method removes ε-productions (productions that generate an empty string). It consists of several steps:

- Identifying Nullable Non-Terminals: First, it identifies all non-terminals that directly produce ε and then iteratively adds non-terminals that produce other nullable non-terminals.
- Adjusting Productions: After identifying nullable non-terminals, it modifies the grammar to include possible combinations of productions that could generate non-empty strings, even when some of their symbols are nullable.
- Expanding Productions: For each nullable symbol in the productions, it generates all possible subsets of the production, excluding and including the nullable symbols.

```
def remove_null_productions(self):
    nullable = set()
    for non_terminal, rhs in self.productions.items():
        for symbols in rhs:
            if symbols == ('ε',):
                nullable.add(non_terminal)

    changed = True
    while changed:
        changed = False
        for non_terminal, rhs in self.productions.items():
            for symbols in rhs:
                if all(symbol in nullable for symbol in symbols) and non_terminal not in nullable:
                    nullable.add(non_terminal)
                    changed = True
```

```
new_productions = {}
for non_terminal, rhs in self.productions.items():
    new_set = set()
    for symbols in rhs:
        if symbols != ('ε',) or (non_terminal == self.start_symbol and len(rhs) == 1):
            new_set.add(symbols)
    new_productions[non_terminal] = new_set

for non_terminal, rhs in new_productions.items():
    expanded_productions = set()
    for symbols in rhs:
        subsets = self._generate_subsets(symbols, nullable)
        expanded_productions.update(subsets)
    new_productions[non_terminal] = expanded_productions
self.productions = new_productions
print("After removing null productions:")
self.display_productions()
```

3. Method: remove_unit_productions()

This method eliminates unit productions (productions where a non-terminal directly produces another non-terminal). It performs the following:

- Separate Unit from Non-Unit Productions: First, it distinguishes between unit and non-unit productions.

- Closure of Unit Productions: It then computes the transitive closure of unit productions, meaning if A -> B and B -> C are unit productions, then A -> C should also be a unit production.

- Integrating Non-Unit Productions: Finally, it updates the grammar by replacing each unit production with the non-unit productions of the non-terminals it can reach.

```
def remove_unit_productions(self):
    import collections
    unit_productions = collections.defaultdict(set)
    non_unit_productions = collections.defaultdict(set)

    for non_terminal, prods in self.productions.items():
        for prod in prods:
            if len(prod) == 1 and prod[0].isupper():
                unit_productions[non_terminal].add(prod[0])
            else:
                non_unit_productions[non_terminal].add(prod)

    changes = True
    while changes:
        changes = False
        for non_terminal in list(unit_productions.keys()):
            current_reach = unit_productions[non_terminal]
            for target in list(current_reach):
                new_reach = unit_productions[target]
                if not new_reach.issubset(current_reach):
                    unit_productions[non_terminal].update(new_reach)
                    changes = True
```

4. Method: remove_inaccessible_symbols()

This method removes symbols (non-terminals) that are not reachable from the start symbol, making the grammar smaller and more manageable. It does this by:

- Tracking Reachable Symbols: Starting from the start symbol, it marks all non-terminals that can be reached by traversing the grammar's productions.

```
def remove_inaccessible_symbols(self):
    accessible = set()
    stack = [self.start_symbol]
    accessible.add(self.start_symbol)

    while stack:
        current = stack.pop()
        for production in self.productions.get(current, []):
            for symbol in production:
                if symbol.isupper() and symbol not in accessible:  # Assuming non-terminals are u
                    accessible.add(symbol)
                    stack.append(symbol)

    self.productions = {nt: prods for nt, prods in self.productions.items() if nt in accessible}
    print("After removing inaccessible symbols:")
    self.display_productions()
```

5. Method: convert_to_cnf()

This is the core method for converting the grammar to Chomsky Normal Form, which restricts productions to either two non-terminals or a single terminal. The method:

- Handle Terminals in Productions: Replaces terminals in productions longer than two symbols with new non-terminals that produce those terminals.

- Reduce Productions to Two Symbols: Ensures that all productions consist of either two non-terminals, one non-terminal and one terminal, or a single terminal, by introducing new non-terminals for every pair of symbols in longer productions.

```
def convert_to_cnf(self):
    new_productions = {nt: set() for nt in self.productions}
    new_non_terminals = {}
    terminal_map = {}
    new_nt_counter = 1  # Counter for new non-terminals

    # Step 1: Replace terminals in multi-symbol productions
    for non_terminal, prods in self.productions.items():
        for prod in prods:
            if len(prod) > 1:
                new_prod = []
                for symbol in prod:
                    if symbol.islower():  # Assuming terminals are lowercase
                        if symbol not in terminal_map:
                            new_nt = f'<Z{new_nt_counter}>'
                            terminal_map[symbol] = new_nt
                            new_productions[new_nt] = {(symbol,)}
                            new_nt_counter += 1
                        new_prod.append(terminal_map[symbol])
                    else:
                        new_prod.append(symbol)
                new_productions[non_terminal].add(tuple(new_prod))
            else:
                new_productions[non_terminal].add(prod)
```

```
# Step 2: Reduce RHS to two symbols
pair_map = {}
final_productions = {nt: set() for nt in new_productions}
for non_terminal, prods in new_productions.items():
    for prod in prods:
        while len(prod) > 2:
            A, B, *rest = prod
            pair = (A, B)
            if pair not in pair_map:
                new_nt = f'<Z{new_nt_counter}>'
                pair_map[pair] = new_nt
                final_productions[new_nt] = {pair}
                new_nt_counter += 1
            new_prod_nt = pair_map[pair]
            prod = (new_prod_nt,) + tuple(rest)
        final_productions[non_terminal].add(prod)
self.productions = final_productions
print("After converting to Chomsky Normal Form:")
self.display_productions()
```

Results

```
# Example grammar
productions = {
    'S': [['d', 'B'], ['A', 'B']],
    'A': [['d'], ['d', 'S'], ['ε'], ['a', 'A', 'a', 'A', 'b']],
    'B': [['a'], ['a', 'S'], ['A']],
    'D': [['A', 'b', 'a']]
}
```

```
After removing null productions:
S ->
    d B
    d
    B
    A B
    A
A ->
    a A a b
    d
    a a A b
    a a b
    d S
    a A a A b
B ->
    a S
    a
    A
D ->
    b a
    A b a
After removing unit productions:
S ->
    a a b
```

```
    d S
    a A a b
    d B
    a
    d
    a S
    a a A b
    A B
    a A a A b
A ->
    a a b
    d S
    a A a b
    a a A b
    d
    a A a A b
B ->
    a a b
    d S
    a A a b
    a
    a S
    a a A b
    d
    a A a A b
D ->
    b a
    A b a
After removing inaccessible symbols:
S ->
    a a b
    d S
    a A a b
    d B
```

```
    a
    d
    a S
    a a A b
    A B
    a A a A b
A ->
    a a b
    d S
    a A a b
    a a A b
    d
    a A a A b
B ->
    a a b
    d S
    a A a b
    a
    a S
    a a A b
    d
    a A a A b
After converting to Chomsky Normal Form:
S ->
    <Z1> S
    <Z8> <Z2>
    <Z3> S
    <Z5> <Z2>
    a
    d
    <Z3> B
    <Z7> <Z2>
    <Z4> <Z2>
    A B
```

```
A ->
    <Z8> <Z2>
    <Z3> S
    <Z5> <Z2>
    d
    <Z7> <Z2>
    <Z4> <Z2>
B ->
    <Z1> S
    <Z8> <Z2>
    <Z3> S
    <Z5> <Z2>
    a
    d
    <Z7> <Z2>
    <Z4> <Z2>
<Z1> ->
    a
<Z2> ->
    b
<Z3> ->
    d
<Z4> ->
    <Z1> <Z1>
<Z5> ->
    <Z4> A
<Z6> ->
    <Z1> A
<Z7> ->
    <Z6> <Z1>
<Z8> ->
    <Z7> A
```

CONCLUSION

In this laboratory, I engaged in the transformation of a context-free grammar (CFG) into the Chomsky Normal Form (CNF). This process helped me understand the essential techniques used to simplify CFGs, crucial for both theoretical studies and practical applications like parsing algorithms. I methodically removed null productions, eliminated unit productions, pruned useless symbols, and standardized the grammar to meet CNF criteria. This experience not only deepened my comprehension of grammatical transformations but also emphasized the practical significance of CNF in computational fields such as parsing and automata theory. Through this exercise, I appreciated the seamless integration of theoretical principles with real-world applications in computer science.