Chomsky Normal Form

Course: Formal Languages & Finite Automata

Author: Moraru Gabriel

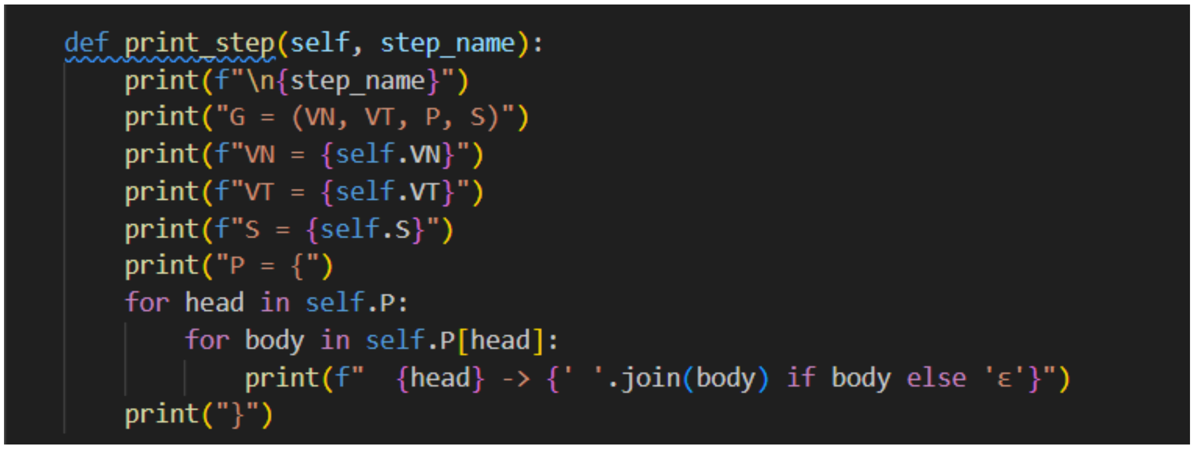
Theory:

Chomsky Normal Form (CNF) is a way of simplifying context-free grammars. In CNF, each rule must be either of the form A → BC (two non-terminals) or A → a (a single terminal). The only exception is the start symbol, which can go to ε (empty string). CNF is useful for parsing and algorithmic analysis, especially in computer science. To convert a grammar to CNF, we eliminate ε-productions, unit rules, and unreachable or useless symbols, then rewrite rules to fit the CNF structure.

Objectives:

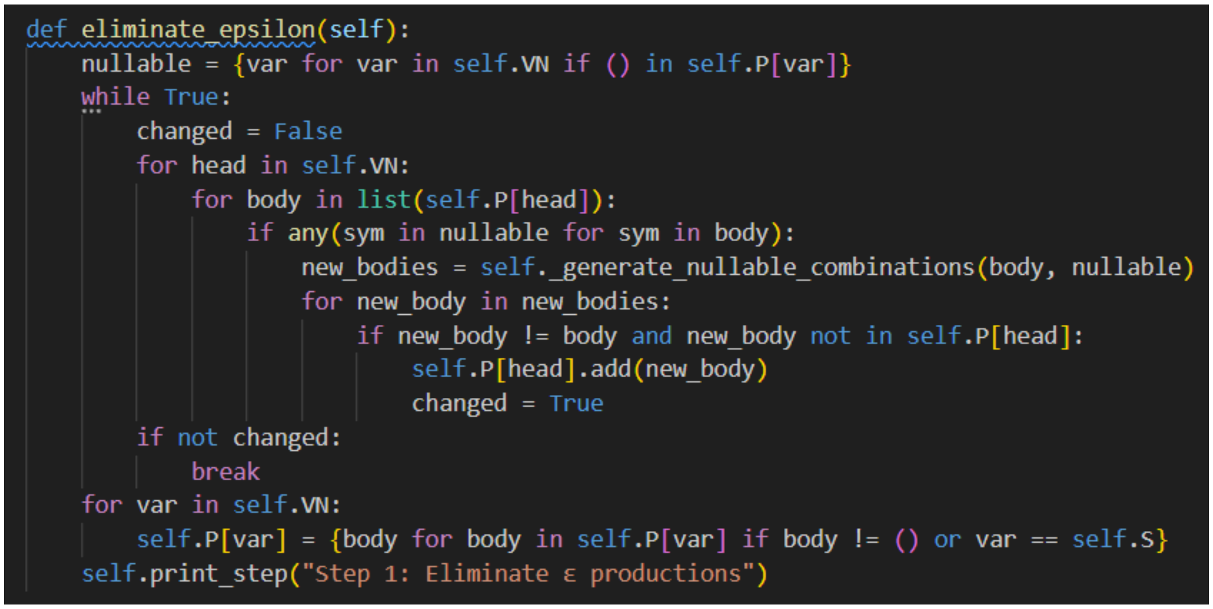
1. Learn about Chomsky Normal Form (CNF) [1].
2. Get familiar with the approaches of normalizing a grammar.
3. Implement a method for normalizing an input grammar by the rules of CNF.
    i. The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).
    ii. The implemented functionality needs executed and tested.
    iii. 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 description:



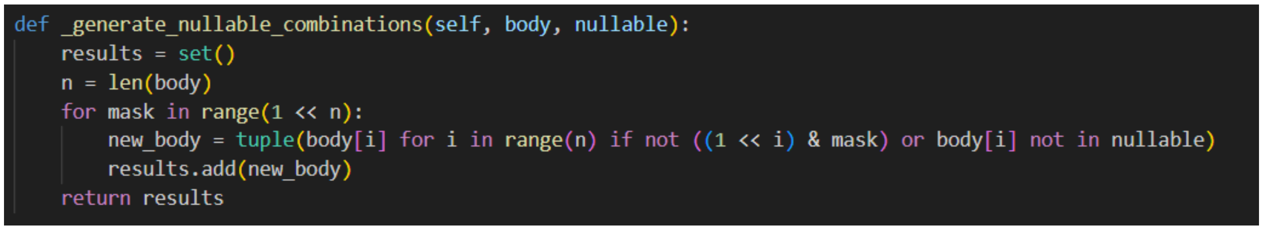
```python
def print_step(self, step_name):
    print(f"\n{step_name}")
    print("G = (VN, VT, P, S)")
    print(f"VN = {self.VN}")
    print(f"VT = {self.VT}")
    print(f"S = {self.S}")
    print("P = {")
    for head in self.P:
        for body in self.P[head]:
            print(f"  {head} -> {' '.join(body) if body else 'ε'}")
    print("}")
```

The print_step method is used to display the current state of the grammar at each step of the conversion process. It prints the grammar components (VN, VT, P, S) in a readable format, including all productions. This helps track changes made during each transformation step.



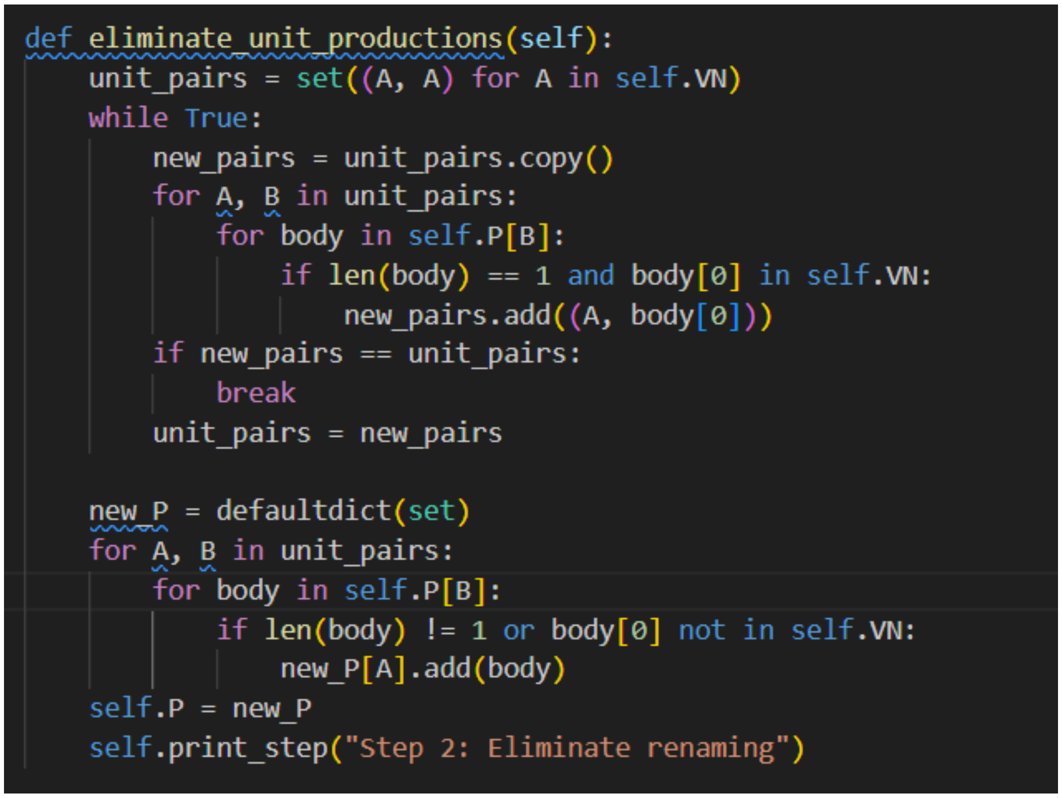
```python
def eliminate_epsilon(self):
    nullable = {var for var in self.VN if () in self.P[var]}
    while True:
        changed = False
        for head in self.VN:
            for body in list(self.P[head]):
                if any(sym in nullable for sym in body):
                    new_bodies = self._generate_nullable_combinations(body, nullable)
                    for new_body in new_bodies:
                        if new_body != body and new_body not in self.P[head]:
                            self.P[head].add(new_body)
                            changed = True
        if not changed:
            break
    for var in self.VN:
        self.P[var] = {body for body in self.P[var] if body != () or var == self.S}
    self.print_step("Step 1: Eliminate ε productions")
```

The eliminate_epsilon method removes ε-productions (nullable productions) from the grammar while preserving its language. It identifies nullable variables, generates all possible combinations of productions without ε, and updates the grammar accordingly.



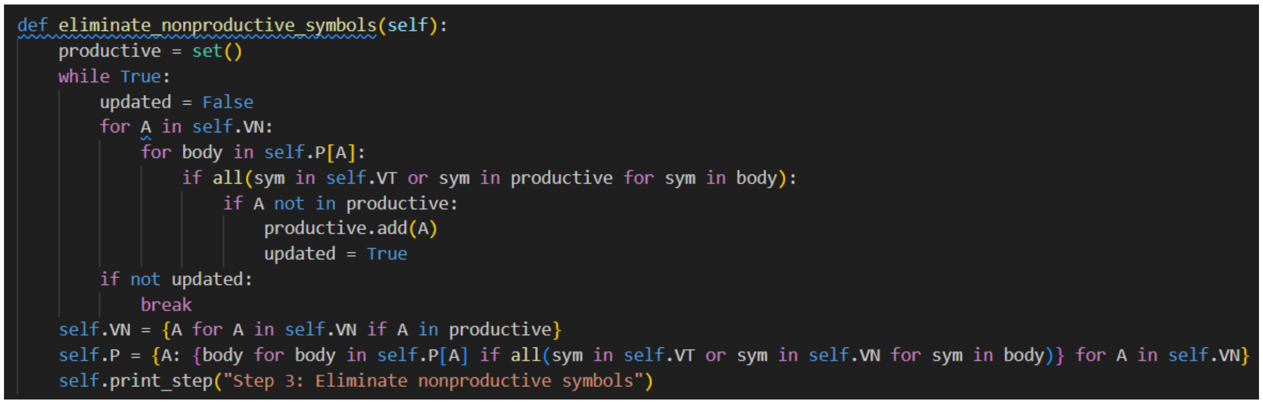
```python
def _generate_nullable_combinations(self, body, nullable):
    results = set()
    n = len(body)
    for mask in range(1 << n):
        new_body = tuple(body[i] for i in range(n) if not ((1 << i) & mask) or body[i] not in nullable)
        results.add(new_body)
    return results
```

The generate_nullable_combinations method generates all possible combinations of a production body by removing nullable variables (variables that can produce ε). It uses bitmasking to systematically include or exclude nullable symbols, ensuring all valid combinations are created.



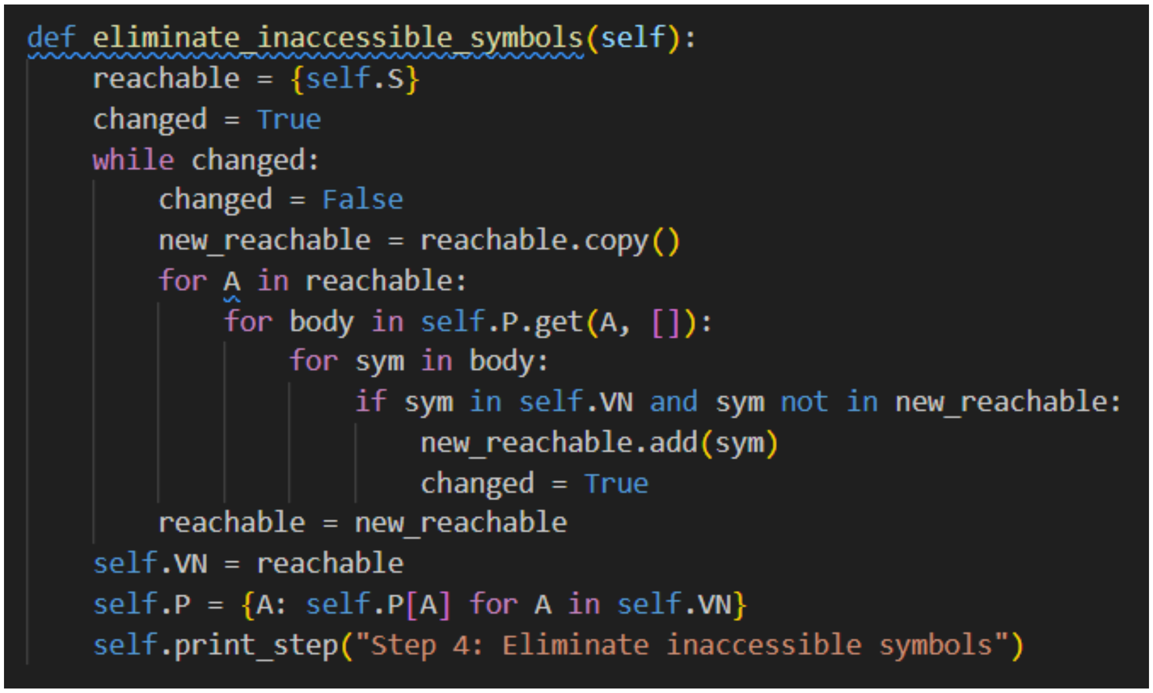
```python
def eliminate_unit_productions(self):
    unit_pairs = set((A, A) for A in self.VN)
    while True:
        new_pairs = unit_pairs.copy()
        for A, B in unit_pairs:
            for body in self.P[B]:
                if len(body) == 1 and body[0] in self.VN:
                    new_pairs.add((A, body[0]))
        if new_pairs == unit_pairs:
            break
        unit_pairs = new_pairs

    new_P = defaultdict(set)
    for A, B in unit_pairs:
        for body in self.P[B]:
            if len(body) != 1 or body[0] not in self.VN:
                new_P[A].add(body)
    self.P = new_P
    self.print_step("Step 2: Eliminate renaming")
```

The eliminate_unit_productions method removes unit productions (e.g., A → B) from the grammar. It identifies all unit pairs (variables that can derive each other) and replaces unit productions with the non-unit productions of the corresponding variables, ensuring the grammar remains equivalent.



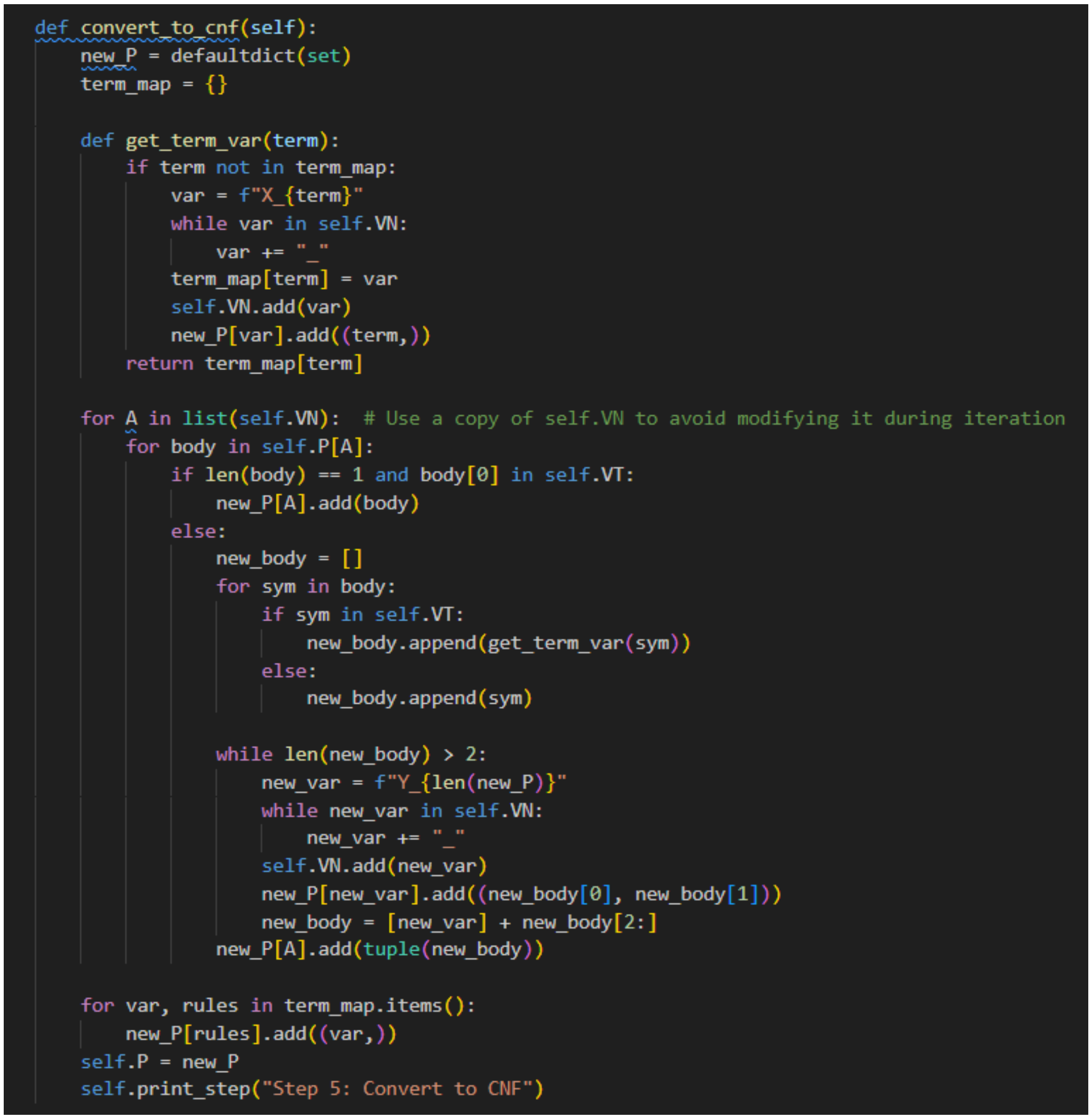
```python
def eliminate_nonproductive_symbols(self):
    productive = set()
    while True:
        updated = False
        for A in self.VN:
            for body in self.P[A]:
                if all(sym in self.VT or sym in productive for sym in body):
                    if A not in productive:
                        productive.add(A)
                        updated = True
        if not updated:
            break
    self.VN = {A for A in self.VN if A in productive}
    self.P = {A: {body for body in self.P[A] if all(sym in self.VT or sym in self.VN for sym in body)} for A in self.VN}
    self.print_step("Step 3: Eliminate nonproductive symbols")
```

The eliminate_nonproductive_symbols method removes nonproductive variables (those that cannot derive terminal strings). It identifies productive variables iteratively and updates the grammar to include only productions involving these variables.



```python
def eliminate_inaccessible_symbols(self):
    reachable = {self.S}
    changed = True
    while changed:
        changed = False
        new_reachable = reachable.copy()
        for A in reachable:
            for body in self.P.get(A, []):
                for sym in body:
                    if sym in self.VN and sym not in new_reachable:
                        new_reachable.add(sym)
                        changed = True
        reachable = new_reachable
    self.VN = reachable
    self.P = {A: self.P[A] for A in self.VN}
    self.print_step("Step 4: Eliminate inaccessible symbols")
```

The eliminate_inaccessible_symbols method removes variables that cannot be reached from the start symbol. It iteratively identifies all reachable variables and updates the grammar to include only those variables and their associated productions.


```python
def convert_to_cnf(self):
    new_P = defaultdict(set)
    term_map = {}

    def get_term_var(term):
        if term not in term_map:
            var = f"X_{term}"
            while var in self.VN:
                var += "_"
            term_map[term] = var
            self.VN.add(var)
            new_P[var].add((term,))
        return term_map[term]

    for A in list(self.VN):  # Use a copy of self.VN to avoid modifying it during iteration
        for body in self.P[A]:
            if len(body) == 1 and body[0] in self.VT:
                new_P[A].add(body)
            else:
                new_body = []
                for sym in body:
                    if sym in self.VT:
                        new_body.append(get_term_var(sym))
                    else:
                        new_body.append(sym)

                while len(new_body) > 2:
                    new_var = f"Y_{len(new_P)}"
                    while new_var in self.VN:
                        new_var += "_"
                    self.VN.add(new_var)
                    new_P[new_var].add((new_body[0], new_body[1]))
                    new_body = [new_var] + new_body[2:]
                new_P[A].add(tuple(new_body))

    for var, rules in term_map.items():
        new_P[rules].add((var,))
    self.P = new_P
    self.print_step("Step 5: Convert to CNF")
```

The convert_to_cnf method transforms the grammar into Chomsky Normal Form (CNF). It replaces terminals in mixed productions with new variables, breaks long productions into binary productions, and ensures all rules conform to CNF requirements.

Conclusion:

This code implements a CFGtoCNFConverter class that systematically converts a context-free grammar (CFG) into Chomsky Normal Form (CNF). It performs key transformations, including eliminating ε-productions, unit productions, nonproductive symbols, and inaccessible symbols, and ensures all productions conform to CNF rules. The process is modular, with each step clearly defined and tested, making the implementation reusable for any grammar.

References:

1. Else Course FAF.LFA21.1

2. Automata Theory, Languages, and Computation: https://www-2.dc.uba.ar/staff/becher/Hopcroft-Motwani-Ullman-2001.pdf

3. Theory of Computation: https://www.geeksforgeeks.org/introduction-of-theory-of-computation/