Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Course: Formal Languages & Finite Automata

Laboratory work 5: Chomsky Normal Form.

Elaborated:

st. gr. FAF-221 Lupan Lucian

Verified:

asist. univ. Dumitru Crețu

Chişinău – 2024

Theory

Regular Expressions

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. For example, A .
- A non-terminal generating two non-terminals. For example, S AB.
- A non-terminal generating a terminal. For example, S a.

For example:

1. G1 = {S AB, S c, A a, B b}
2. G2 = {S aA, A a, B c}

The production rules of Grammar G1 satisfy the rules specified for CNF, so the grammar G1 is in CNF. However, the production rule of Grammar G2 does not satisfy the rules specified for CNF as S aZ contains terminal followed by non-terminal. So the grammar G2 is not in CNF.

Steps for converting CFG into CNF

Step 1: Eliminate epsilon productions from the grammar.

Step 2: In the grammar, eliminate the unit productions, replacing them with the productions of the referenced non-terminal.

Step 3: Remove any inaccessible symbols from the productions.

Step 4: Check and eliminate any unproductive symbols from the grammar.

Step 5: Convert to CNF (Chomsky Normal Form), meaning we can only have a terminal symbol, and at most 2 non-terminal symbols for each of the productions.

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.
    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

Like my previous labs, I used Python to solve the problems. Starting off, I had the 18th variant, which had the following grammar:

- Vn = ["S", "A", "B", "C", "D"]
- Vt = ["a", "b"]
- P = {"S": ["aB", "bA", "B"], "A": ["b", "aD", "AS", "bAB", "ε"], "B": ["a", "bS"], "C": ["AB"], "D": ["BB"]}

I encapsulated all the data for this laboratory work inside a class – named ChomskyNormalForm, which has a simple constructor where the above values are declared:

```
class ChomskyNormalForm:

    Vn = []
    Vt = []
    P = {}

    def __init__(self):
        self.Vn = ["S", "A", "B", "C", "D"]
        self.Vt = ["a", "b"]
        self.P = {
            "S": ["aB", "bA", "B"],
            "A": ["b", "aD", "AS", "bAB", "ε"],
            "B": ["a", "bS"],
            "C": ["AB"],
            "D": ["BB"]
        }
```

In order to get to the Chomsky Normal Form, we first had to eliminate the epsilon productions. To do this, we created a function that checks if the function has any epsilon productions, and if it does, we iterate through the values which do have the epsilon production, making use of a generate

combinations functions, which lists all the possible combinations created from a dictionary entry.

```
def eliminate_epsilon_productions(self):
    if self.check_for_epsilon_productions():
        eps_keys = []
        for key in self.P.keys():
            if "ε" in self.P[key]:
                eps_keys.append(key)
                self.P[key].remove("ε")
        for key in self.P.keys():
            if any([any(value in eps_keys for value in values) for values in self.P[key]]):
                for keys in eps_keys:
                    for value in self.P[key]:
                        if keys in value:
                            if value == keys:
                                self.P[key].append("ε")
                            else:
                                combinations = self.generate_combinations(value, keys)
                                for combination in combinations:
                                    if combination not in self.P[key]:
                                        self.P[key].append(combination)
```

After the epsilon productions have been eliminated, I had to move on to getting rid of unit productions. This was a much simpler process, where I only had to copy the productions of an unit productions to its respective correspondent.

```
def eliminate_unit_productions(self):
    if self.check_for_unit_productions():
        for key in self.P.keys():
            for value in self.P[key]:
                if value in self.Vn and self.P.get(value) is not None:
                    self.P[key].remove(value)

                    for production in self.P[value]:
                        if production not in self.P[key]:
                            self.P[key].append(production)
```

Afterwards, we had to check for inaccessible symbols. To achieve this, we start off from the starting symbol – in our case – "S", and iterate through the entire productions dictionary, mapping all the reachable symbols in a temporary array. At the end, when we iterated through all the keys and values, we remove any keys that are not part of the accessibly_symbols array:

```
def check_for_inaccessible_symbols(self):
    accessible_symbols = [next(iter(self.P.keys()))]
    for values in self.P.values():
        for value in values:
            for non_terminal in self.Vn:
                if non_terminal in value:
                    if non_terminal not in accessible_symbols:
                        accessible_symbols.append(non_terminal)
    return accessible_symbols
```

The second last task before achieving CNF was to remove unproductive symbols. To do this, I used a similar approach to the previous point, by iterating through the dictionary, and constantly updating the productive symbols. If the values of a key are not made entirely out of terminal symbols, or a combination between terminal symbols and productive symbols, then the production is deemed unproductive and is eliminated out of the dictionary.

```
def check_for_inaccessible_symbols(self):
    accessible_symbols = [next(iter(self.P.keys()))]
    for values in self.P.values():
        for value in values:
            for non_terminal in self.Vn:
                if non_terminal in value:
                    if non_terminal not in accessible_symbols:
                        accessible_symbols.append(non_terminal)
    return accessible_symbols
```

Last but not least, we had to create the CNF. As stated in the theory, CNF is composed either out of a single terminal symbol, or at most 2 non-terminal symbols. To create the CNF, we had to make

another dictionary, where we'd store the newly created non-terminal symbols, which either link to 1 terminal, or 2 non-terminals (ex. X1 -> a; X2->AX1). Because this caused a congestion in terms of counting the number of characters in a string, I had to develop a function which would select the first 2 "true" characters of a string, in order to properly group them:

```
def select_first_two_true_characters(self, value):
    true_characters = ""
    true_count = 0
    in_x = False
    for char in value:
        if char == "X" and not in_x:
            in_x = True
            true_characters += char
        elif char == "X" and in_x:
            true_characters += char
            true_count += 1
        elif char.isdigit() and in_x:
            true_characters += char
        elif in_x:
            true_count += 1
            in_x = False
            if true_count == 2:
                break
            true_characters += char
            true_count += 1
```

Lastly, I created a simple function that links all the above functions into 1, creating the CNF:

```
def to_chomsky_normal_form(self):
    self.eliminate_epsilon_productions()
    self.eliminate_unit_productions()
    self.remove_inaccessible_symbols()
    self.remove_unproductive_symbols()
    self.chomsky_normal_form()
```

Results

Input:

```
chomsky = ChomskyNormalForm()
print(chomsky.P)
print(chomsky.Vn)
chomsky.to_chomsky_normal_form()
print(chomsky.P)
print(chomsky.Vn)
```

Output:

```
{'S': ['aB', 'bA', 'B'], 'A': ['b', 'aD', 'AS', 'bAB', 'ε'], 'B': ['a', 'bS'], 'C': ['AB'], 'D': ['BB']}
['S', 'A', 'B', 'C', 'D']
{'S': ['b', 'a', 'X1B', 'X2A', 'X2S'], 'A': ['b', 'AS', 'a', 'X1D', 'X3B', 'X2B', 'X1B', 'X2A', 'X2S'], 'B': ['a', 'X2S'], 'D': ['BB'], 'X1': 'a', 'X2': 'b', 'X3': 'X2A'}
['S', 'A', 'B', 'C', 'D', 'X1', 'X2', 'X3']
```

Conclusions

This laboratory work was relatively challenging, yet also a lot of fun. Working with dictionaries in Python, I learnt about how I could use iterators in order to iterate through the keys of a dictionary, and thus check for any inaccessible/unproductive symbols. I've also learnt some more tricks regarding dictionaries, such as creating copies in order to iterate through them, while also being able to modify them freely, as well as using list comprehension to simplify code and save space.

Regarding Chomsky Normal Form, translating the simple steps from the theory lessons and seminars to code proved to be a bit more challenging than I initially thought. Although I knew what I had to do related to the code, I felt somewhat lost sometimes because I'd miss small things like working with the same dictionary while iterating and modifying it, without creating a copy, which made me waste time debugging. Another challenging aspect was creating the combinations for the epsilon productions. As a number could have a symbol reoccur an infinite number of times, this required the use of a dedicated function, which would iterate through the value, and recursively create all the possible combinations using the symbol which was currently being rid of the epsilon production.

To conclude, I've learnt a lot of interesting things this laboratory, and although it was more challenging than the last one, I've had a lot of fun.

References:

1. Automata Chomsky Normal Form [online]. [accessed 15.04.2024]. Available: https://www.javatpoint.com/automata-chomskys-normal-form