MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA

TECHNICAL UNIVERSITY OF MOLDOVA

FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS

Laboratory work 5:
Formal Languages and Finite Automata

Elaborated:

st. gr. FAF-223 **Vieru Mihai**

Verified:

asist. univ. **Cretu Dumitru**

Chișinău, 2024

Content

We start by removing all states which lead to epsilon or states that lead to other states which lead to epsilon. To solve this task I made 2 methods, one for determening which states ultimally lead to epsilon and one to change those states into their non epsilon equivalent.

AllRoadsLeadTo~~Rome~~Epsilon:

```
def getAllEpsilonNonTerminals():
            epsilonNonTerminals = set()
            def dfs(bad_value="ImagineEpsilonIsHereBecauseLatexIsCringe"):
                for non_terminal in self.productions.keys():
                    if non_terminal not in epsilonNonTerminals:
                        for productions in self.productions[non_terminal]:
                            if bad_value in list(productions):
                                epsilonNonTerminals.add(non_terminal)
                                dfs(non_terminal)
            dfs()
            return epsilonNonTerminals
```

This is a recursive function which checks if a production of a non terminal has a character which is related to epsilon. For example if a non terminal leads to epsilon and another non terminal leads to it, then we do a recursive call and now check if any other non terminals have this non terminal. And while we do this we add all of them into a list, and return it.

Now that we have a list of the non terminals we must normalize we can start doying that. And for that we will be using.

ByByeBadGuy:

```
def getEpsilonEmptyProduction(productions):
            res = set()
            def dfs(ignore=set()):
                if len(productions) == 1:
                    return

                curr = ""
                for part_id in range(len(productions)):
                    str_part_id = str(part_id)
                    if str_part_id not in ignore:
                        part = productions[part_id]
                        curr += part
                        if part.isalpha() and part == part.upper():
                            dfs(ignore | set(str_part_id))

                if curr not in ["", productions]:
                    res.add(curr)

            dfs()
            return res
```

Here we do a backtracking solution where generate all combinations of a word, but without a special non terminal(it's id, because, there can be more that one ignore) and terminals. So for AXaD, we will get 'XaD', 'Aa', 'AaD', 'aD', 'a', 'AXa', 'Xa'.

Then we combine this 2 to get rid of all epsilon transitions:

CombinedByeBye:

```
    epsilonNonTerminals = getAllEpsilonNonTerminals()
            for non_terminal in epsilonNonTerminals:
                for production_id in range(len(self.productions[non_terminal])):
                    production = self.productions[non_terminal][production_id]

                    if len(production) > 1 and production != production.lower():
                        self.productions[non_terminal].extend(list(getEpsilonEm
                        ↪ ptyProduction(production)))
```

Now we got rid of the epsilons. We have to get rid of the single non terminals. We do this by movind their actual value to where they are so lets say

C: aB

A: C

then we just move the value of C to A

C: aB

A: aB

EsliGoraNeIdeotKMohamedu...:

```
def moveNonTerminals(self):
        for non_terminal, productions in self.productions.items():
            for production_id in range(len(productions)):
                production = productions[production_id]
                if len(production) == 1 and production !=
                 ↪  production.lower():
                    productions.pop(production_id)
                    productions.extend(self.productions[production])
                    self.moveNonTerminals()
```

Here we just recursivelly replace all of the productions with their values from the dict.

Now after this it's time to get rid of the terminals from the productions if there are more than one:

ReplaceWithCyrilic:

```
class Iterator():
            def __init__(self, data, iterr=0):
                self.data = data
                self.iter = iterr

            def __iter__(self):
                return self

            def __next__(self):
                if self.iter < len(self.data):
                    self.iter += 1
                    return self.data[self.iter]
                else:
                    raise IndexError("Iterator Out Of Bounds.")

            def reset(self):
                self.iter = 0
```

```
def replaceTerminals(self):
    data = ['All', 'Cyrillic', 'Letters', 'Here', '', '', '', '', '',
     ↪  '', '', '', '', '', '', '', '', '', '']
    iterator = self.Iterator(data)
    new_non_terminals = {}

    for non_terminal, productions in self.productions.items():
        for production_id in range(len(productions)):
            production = list(productions[production_id])
            if len(production) > 1:
                for item_id in range(len(production)):
                    item = production[item_id]
                    if item == item.lower():
                        if new_non_terminals.get(item) == None:
                            new_non_terminal = next(iterator)
                            new_non_terminals[item] = new_non_terminal
                        production[item_id] = new_non_terminals[item]
                productions[production_id] = "".join(production)

    for productions, non_terminal in new_non_terminals.items():
        self.productions[non_terminal] = [productions]
```

To solve the problems of terminals inside productions which are bigger than 1. I created a dictionary and create inside a mapping for all the terminals, which are atributed to a non terminal which is created from the Iterator class I made which just gives me the next word from a list based on the list I give, and I chose the old Cyrilic script so I don't have any collisions with my current non terminals.

Now as the last step I just to group 2 non terminals into a new non terminal so I always have only 2 max non terminals next to each other.

ReplaceWithCyrilic:

```
def groupSelfLiterals(self):
    data = ['', '', '', '', '', '', '', '', '', '', '', '', '']
```

```
        iterator = self.Iterator(data)
        new_non_terminals = {}
        def group():
            for non_terminal, productions in self.productions.items():
                for production_id in range(len(productions)):
                    production = list(productions[production_id])
                    if len(production) > 2:
                        for item_id in range(0, len(production), 2):
                            if item_id + 1 < len(production):
                                item = production[item_id] +
                                 ↪ production[item_id+1]
                                if new_non_terminals.get(item) == None:
                                    new_non_terminal = next(iterator)
                                    new_non_terminals[item] =
                                     ↪ new_non_terminal
                                production.pop(item_id+1)
                                production[item_id] =
                                 ↪ new_non_terminals[item]
                        productions[production_id] = "".join(production)

            for productions, non_terminal in new_non_terminals.items():
                self.productions[non_terminal] = [productions]
```

To do this I do same thing I did in the previous method, but now for 2 non terminals, by mapping the values of 2 non terminals to a new non terminal.

And in the end I just chain them togheter and get the last output.

Conclusion:

In this laboratory, I explored the process of converting context-free grammars (CFGs) into Chomsky Normal Form (CNF). CNF is a highly constrained format for CFGs, where productions adhere to specific rules, making them valuable for certain parsing algorithms. My goal was to develop a systematic method to achieve this transformation while gaining insights into the principles of formal grammars.

My approach involved addressing several key challenges in the conversion process. Firstly, I focused on eliminating epsilon productions, or productions that derive the empty string. I implemented a recursive algorithm (getAllEpsilonNonTerminals) to identify non-terminals capable of such derivations. Subsequently, I employed a backtracking solution (getEpsilonEmptyProduction) to generate all possible combinations of productions while excluding epsilon-generating terminals.

The next step was to remove unit productions – productions where a single non-terminal derives another single non-terminal. I implemented a recursive function (moveNonTerminals) that iteratively replaced such productions with the right-hand side of the referenced non-terminal, effectively propagating productions to eliminate unit productions.

To tackle productions exceeding the length restrictions of CNF, I devised a resourceful scheme. I created a mapping between terminals and unique non-terminals using an Iterator class. This substitution allowed me to isolate terminals within long productions and conform to CNF's constraints.

Finally, I ensured that all production rules adhered to CNF's requirement of having at most two non-terminals on the right-hand side. The groupSelfLiterals function achieved this by recursively combining pairs of non-terminals into new non-terminals, maintaining the CNF structure.

Throughout this laboratory, I gained valuable insights into the manipulation of formal grammars, the interplay of data structures and algorithms, and the stepwise process of problem-solving in computational linguistics. While my solution proved successful, there are opportunities for further refinement. Providing formal correctness proofs would augment the theoretical rigor of the work. Furthermore, a comprehensive analysis of computational complexity could pave the way for potential optimizations.

In the future, I plan to incorporate robust error handling mechanisms and investigate alternative CNF conversion algorithms. This exploration promises to deepen my understanding of formal language theory and its practical applications.