MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA
Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică
Departamentul Inginerie Software și Automatică

Report

Zagorodniuc Anastasia FAF-223

Regular expressions

of Formal Languages & Finite Automata

Checked by:

Cretu Dumitru, *university assistant*

Chișinău – 2023

1. Theory

Chomsky Normal Form (CNF) is a specific way of organizing the rules of a context-free grammar, which is often used in computer science for parsing and working with languages. In CNF, every rule of the grammar must either produce two non-terminal symbols, produce a single terminal symbol, or be the start symbol producing the empty string. This simplification makes algorithms for parsing and analyzing languages more straightforward, especially for tasks like the CYK algorithm, which checks whether a string belongs to a language defined by a grammar in CNF.

2. Objectives

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

3. Implementation Description

The method described below represents - **Elimination of ε-productions**
We start with initialization of a hashset named “nullable” created to store non-terminals that can produce an ε directly or through a series of productions. These non-terminals are considered nullable because they can derive the empty string. We try to find directly nullable non-terminals. We iterate over all productions in the grammar, represented by the productions map, where each non-terminal symbol maps to a list of its production rules. If any production rule for a non-terminal is exactly ε, that non-terminal is added to the “nullable” set. This way we identify all non-terminals that can directly produce ε.

```
var nullable = new HashSet<string>();
foreach (var nonTerminal in productions.Keys)
{
    foreach (var production in productions[nonTerminal])
    {
        if (production == "ε")
        {
            nullable.Add(nonTerminal);
            break;
        }
    }
}
```

Afterwards we try to find indirectly nullable non-terminals by entering a loop that continues as long as changes are made to the “nullable” set. This is to account for non-terminals that might not directly produce ε but can still derive ε through a combination of other nullable non-terminals. For each production rule, if all characters/symbols in the rule are in the nullable set, the non-terminal is added to nullable.

```
bool changes;
do
{
    changes = false;
    foreach (var nonTerminal in productions.Keys)
    {
        foreach (var production in productions[nonTerminal])
        {
            if (production.All(c => nullable.Contains(c.ToString())))
            {
                if (nullable.Add(nonTerminal))
                {
                    changes = true;
                }
            }
        }
    }
} while (changes);
```

After that we try to modify productions to exclude nullable non-terminals. We iterate over all non-terminals to modify their production rules to exclude those that can lead to ε directly through nullable symbols. For each production that does not equal ε, it creates a modified version of the production without the nullable symbols. In the end, the method removes all direct ε-productions from the grammar.

The method described below represents - **Elimination of any renaming**

```
bool changesMade;
do
{
    changesMade = false;
    var newProductions = new Dictionary<string, List<string>>();

    foreach (var nonTerminal in productions.Keys)
    {
        var currentProductions = new List<string>(productions[nonTerminal]);
        var toAdd = new List<string>();

        foreach (var production in currentProductions)
        {
            if (production.Length == 1 && nonTerminals.Contains(production))
            {
                foreach (var redirectedProduction in productions[production])
                {
                    if (!currentProductions.Contains(redirectedProduction) && !toAdd.Contains(redirectedProduction))
                    {
                        toAdd.Add(redirectedProduction);
                        changesMade = true;
                    }
                }
            }
            else
            {
                toAdd.Add(production);
            }
        }
        newProductions[nonTerminal] = toAdd.Distinct().ToList();
    }
    productions = newProductions;
} while (changesMade);
```

Firstly, we initialize a new HashMap named “newProductions”, which will hold the modified set of production rules for each non-terminal, excluding the renaming productions. Iterating over each non-terminal in the production map, we perform several steps to modify its production rules. Create a new list, “currentProductions”, created as a copy of the current production rules for the non-terminal. For each production rule in “currentProductions”, the method checks if the rule is a renaming production. This is done by checking if the production rule's length is 1, indicating it's a

single symbol, and if the symbol is a non-terminal, present in “nonTerminals” set. If a production rule is identified as a renaming production, instead of directly modifying “currentProductions”, the production rules of the non-terminal it points to are added to a temporary list “toAdd”. If a production rule is not a renaming production, it's added directly to “toAdd”.

After processing all non-terminals, the original production map is cleared and then repopulated with the modified productions stored in “newProductions”. This effectively updates the grammar to exclude renaming productions.

The method described below represents - **Elimination of any nonproductive symbols**

```
var productive = new HashSet<string>(terminals);
bool changes;
do
{
    changes = false;
    foreach (var nonTerminal in productions.Keys)
    {
        foreach (var production in productions[nonTerminal])
        {
            if (production.All(c => productive.Contains(c.ToString()) || nonTerminals.Contains(c.ToString())))
            {
                if (productive.Add(nonTerminal))
                {
                    changes = true;
                }
            }
        }
    }
} while (changes);

var newProductions = new Dictionary<string, List<string>>();
foreach (var entry in productions)
{
    if (productive.Contains(entry.Key))
    {
        var newProductionList = entry.Value.Where(production => production.All(c => productive.Contains(c.ToString()))).ToList();
        if (newProductionList.Any())
        {
            newProductions[entry.Key] = newProductionList;
        }
    }
}
productions = newProductions;
```

We initialize a loop that continues until no new productive symbols are found in an iteration. For each non-terminal symbol in the grammar, we iterate over its production rules. We check if every symbol in the rule is either a known productive symbol or a non-terminal symbol. This is done using the `allMatch` method on the character stream of the production string. If the production rule is composed entirely of productive symbols and non-terminal symbols, the non-terminal symbol of the current production rule is considered productive, and it's added to the “productive” set. If it's newly added, the “changes” flag is set to true, indicating that the set of productive symbols has been updated.

After identifying all productive symbols, the method iterates over the “productions” map entries. It checks each entry (each non-terminal and its production rules). If a non-terminal is not in the “productive” set, it means the non-terminal is non-productive, and its entry is removed from the “productions” map. For the remaining non-terminals, it further cleans up their production rules by removing any production that contains symbols not in the “productive” set.

By the end of this method, all non-productive symbols and their associated productions are removed from the grammar.

The method described below represents - **Elimination of any inaccessible symbols**

```
var accessible = new HashSet<string> { startSymbol };
var toProcess = new HashSet<string> { startSymbol };

while (toProcess.Any())
{
    var nextRound = new HashSet<string>();
    foreach (var symbol in toProcess)
    {
        foreach (var production in productions.GetValueOrDefault(symbol, new List<string>()))
        {
            foreach (var c in production)
            {
                var strC = c.ToString();
                if (nonTerminals.Contains(strC) && accessible.Add(strC))
                {
                    nextRound.Add(strC);
                }
            }
        }
    }
    toProcess = nextRound;
}

productions = productions.Where(entry => accessible.Contains(entry.Key)).ToDictionary(entry => entry.Key, entry => entry.Value);
```

We initialize 2 sets, one named “accessible”, that will eventually contain all the non-terminal symbols that are accessible from the start symbol and another “toProcess”, a set of symbols to be processed in the current iteration, initially containing just the start symbol of the grammar. The code enters a loop that continues as long as there are symbols to process in “toProcess”. Another set “nextRound” is initialized to collect symbols found to be accessible in the current iteration, to be processed in the next iteration.

The loop iterates over each symbol in “toProcess” and adds to the “accessible” set as it is now confirmed to be reachable from the start symbol. For each symbol, the code retrieves its productions from the grammar. Each production is a string of symbols. The code iterates over each character in the production. If a character is a non-terminal and not already marked as accessible, it's added to “nextRound” for processing in the next iteration. After processing all symbols in “toProcess”, the set is replaced by “nextRound”, containing the newly found accessible symbols for processing in the next iteration.

Once all accessible symbols have been identified, the “keySet()” of “productions” is filtered to retain only those keys that are in the “accessible” set.

The method described below represents - **The Chomsky Normal Form**

```
var terminalReplacements = new Dictionary<string, string>();
var productionReplacements = new Dictionary<string, string>();
var newNonTerminals = new List<string> { "Ш", "Щ", "Ч", "Ц", "Ж", "Й", "Ъ", "Э", "Ю", "Б", "Ь", "Г", "Ы", "П", "Я", "И" };
int newNonTerminalIndex = 0;
```

```
foreach (var entry in productions)
{
    var modifiedProductions = new List<string>();
    foreach (var production in entry.Value)
    {
        if (production.Length == 1 && terminals.Contains(production))
        {
            modifiedProductions.Add(production);
            continue;
        }

        var newProduction = new StringBuilder();
        foreach (var c in production)
        {
            var symbol = c.ToString();
            if (terminals.Contains(symbol))
            {
                if (!terminalReplacements.ContainsKey(symbol))
                {
                    terminalReplacements[symbol] = newNonTerminals[newNonTerminalIndex++ % newNonTerminals.Count];
                }
                newProduction.Append(terminalReplacements[symbol]);
            }
            else
            {
                newProduction.Append(symbol);
            }
        }
        modifiedProductions.Add(newProduction.ToString());
    }
    newProductions[entry.Key] = modifiedProductions;
}
```

We initialize replacement maps”terminalReplacements”, used to keep track of which terminal symbols have been replaced by which new non-terminal symbols and “productionReplacements” with the intention to track transformations within the productions. As well we initialized a list of “newNonTerminals”, a list of characters that are to be used as new non-terminal symbols for replacement purposes. These symbols are distinct and presumably not used elsewhere in the grammar to avoid conflicts.

The code iterates through a given map “productions”, where each entry consists of a non-terminal symbol and a list of its productions. For each production if a production is a single terminal symbol, it's left unchanged. For longer productions, each symbol is examined. If it's a terminal, a replacement process occurs, if not replaced before, a new non-terminal from “newNonTerminals” is assigned to it, ensuring that each terminal gets a unique replacement until all available non-terminals are used.

This additional step iterates over the “terminalReplacements” map, where terminal symbols have been mapped to their new non-terminal symbols, and adds these mappings as new productions to the “finalNewProductions” map. Specifically, for each terminal-new non-terminal pair, it creates a

production rule where the new non-terminal produces the original terminal symbol, ensuring these replacements are formally integrated into the grammar.

In this segment of code at the beginning the “productions” map is cleared to remove all existing productions and then repopulated with the transformed productions from the first pass, stored in “finalNewProductions”. After that we initialize a new “newProductions” for storing the updated productions from this second transformation pass.

The code iterates over each production in the “productions” map. For productions longer than two symbols, it progressively breaks them down. It creates a new non-terminal symbol for every two symbols at the beginning of the production. This new non-terminal symbol is mapped to the two symbols it replaces in the “productionReplacements” map and added to “newProductions” with a production rule that produces those two symbols. The process ensures unique non-terminal symbols for each unique pair of symbols. Productions that are already unary (single symbol) or binary (two symbols) are added to the “newProductions” map without modification.

Once all productions have been processed and potentially transformed, the original “productions” map is cleared again and repopulated with the updated productions from “newProductions”.

Output:

```
Enter lab:
lab5
Щ -> AS
Ч -> ШA
Ц -> ЧD
Ж -> ШD
Й -> ЖA
Ъ -> ЙD
S -> ШB | a | b | ЩB | BD | ЦB | ЧB | ЪB | DA
A -> a | b | ЩB | BD | ЦB | ЧB | ЪB
B -> b | ЩB
D -> BA
Ш -> a
```

4. Conclusion

In summary, tackling this lab was indeed a formidable task, demanding considerable effort and a deep understanding of the subject matter. The journey of converting a grammar to Chomsky Normal Form (CNF) involved navigating through a complex process comprising multiple intricate steps. This challenge was amplified by the necessity to grasp concepts like epsilon productions, renaming, productive and accessible symbols, and the conversion to CNF itself.

Each stage of the normalization process necessitated meticulous handling of grammar rules to ensure that the resultant grammar strictly adhered to the CNF structure while preserving the language it generates. One of the particularly challenging aspects was managing the indirect consequences of each transformation step. For instance, eliminating certain productions could alter the set of productive or accessible symbols, prompting a reevaluation of the grammar.

Moreover, achieving CNF-compliant productions required a creative approach to decomposing longer productions without introducing redundancy or unwarranted complexity. These intricate considerations underscored the complexity of the task and the importance of precision throughout the conversion process.