laboratory_5.md

Laboratory Work 5

Course: Formal Languages & Finite Automata

Author: Felicia Novac

Theory

Chomsky Normal Form (CNF) is a specific way of structuring the production rules of a context-free grammar. A grammar in CNF has each of its production rules in one of the two forms: `A->BC` or `A->a`. The main reason of converting a type 2 grammar to CNF is the significant simplification of parsing.

The conversion process of a Context-Free Grammar into CNF includes 5 main steps:

1. *ε-productions removal*
2. *Unit-productions removal*
3. *Inaccessible-symbols removal*
4. *Non-productive states removal*
5. *Conversion to CNF*

Objectives:

- Learn about Chomsky Normal Form (CNF).
- Get familiar with the approaches of normalizing a grammar.
- Implement a method for normalizing an input grammar by the rules of CNF.
- Implement unit tests that validate the functionality of the project
- Make the CNF conversion extendable to any form of grammar (type 2).

Implementation description

In order to implement the functionality *conversion to CNF*, I've initialized a new class named `ChomskyNormalForm` that extends the `Grammar` class, consequently inheriting all its public & protected attributes and methods. When instantiating CNF class, we would do so by passing a `Grammar` object created in advance with corresponding Vn, Vt, P and S attributes.

This conversion, as mentioned before, involves several steps to ensure the grammar meets the specific conditions required by CNF.

ε-productions removal

The `removeEpsilonProductions` method starts with identifying non-terminals that directly produce epsilon by iterating over the productions with the use of the forEach() method for Maps and store the ones that `.contains("ε")` in a `Set<String> epsilonProducingNonTerminals`. Once the states producing epsilon are identified, the method checks for non-terminals that can produce an epsilon indirectly. This is done by iterating over the `Map<String, List<String>> P` entries and for each entry, over the corresponding list of values (productions). Each `char` of every production is being verified for belongness to `epsilonProducingNonTerminals` or even being equal to "ε", and if so it is added to the same Set. After identifying all epsilon producers, we have to adjust the existing production rules that contain those producers. We do so by iterating once again over the productions in the Map and for each production excluding ε in the productions List we generate all possible combinations of symbols in a production string. This is achieved through a for loop that runs from 0 to 2^(production.length()) - 1, representing every possible subset of the production's characters. Each subset is represented by a binary number, where each bit indicates whether the corresponding character is included (1) or excluded (0). For each subset, we iterate over each character in the production combination using another for loop. Then, we perform the bitwise check `(i & (1 << j)) == 0` to find out whether a character in the production string should be excluded from the current subset being generated. Along with bitwise check goes `!epsilonProducingNonTerminals.contains(String.valueOf(production.charAt(j)))`, which ensures that the character, if included, is not an epsilon-producing non-terminal. Each character corresponding to at least one of those conditions is appended to a StringBuilder object, building a new production string.

Unit-productions removal

The `removeUnitProductions()` method starts by identifying and separating productions. This means iterating over the P Map and for each pair of non-terminal with the corresponding List of productions, iterate over those productions and check their length and belongness to Vn, such that a unit production consists only of a single non-terminal. In this way, we separate the productions, and those that are ok we add directly to the final Map. Now we are left with modifying the unit-productions, and to do so we iterate over the previously found unit-productions and for each such production, we search in the P Map and retrieve the leading productions from the correponding single-non terminal and replace it with its productions. If one of the found productions is still an unit production, add it to `toVisit` List if it is not yet there and process until there are no more elements left in that List.

Inaccessible-symbols removal

The `removeInaccessibleSymbols` method starts by initializing a `Set<String> accessibleSymbols` and adding `S` as the first element, as it is automatically accessible. After that, we have to find the over accessible symbols, and we do so just like in real life. Iterate over the entries in P Map and retrieve the current key (non-terminal) being process, to check the belongness to `accessibleSymbols`. Once a belonging non-terminal was found, we iterate over its productions and add the newly found resulting non-terminals. Also, mark `boolean changed` flag as true, to let the program know that there were added new states in the Set and it is needed to be process once again. Now that we have identified the inaccessible symbols, we have to remove all the productions containing them. To do so, we iterate over the original productions Map and verify each key (non-terminal) if it is contained in `accessibleSymbols` Set as well, and if yes add it with the corresponding value to the Map that holds the new productions and only the key to new Vn as well.

Non-productive states removal

The `removeNonProductiveSymbols` method operates similar to the previously described one, but does not initially add S to the Set. It also uses a `boolean changed` flag to keep track of any new States added in `productiveSymbols` Set. We're set to identify all productive symbols, do so by iterating over the P productions, but process only those non-terminal that are not yet added to `productiveSymbols`, and for each non-terminal's production, check each symbol belongness to either `Vt` or `productiveSymbols`. If none of the conditions are met the `productionIsProductive` flag is marked as false and production isn't added to `productiveSymbols`. After identifying all productive symbols, we have to exclude the non-productive ones. Iterate once again over the P Map and process only the productions of the non-terminals that are marked as productive. Same as previously done, check each production's symbol belongness to either `Vt` or `productiveSymbols`. If at least one of the conditions is met, add the corresponding production to the `newProductions` with its corresponding non-terminal.

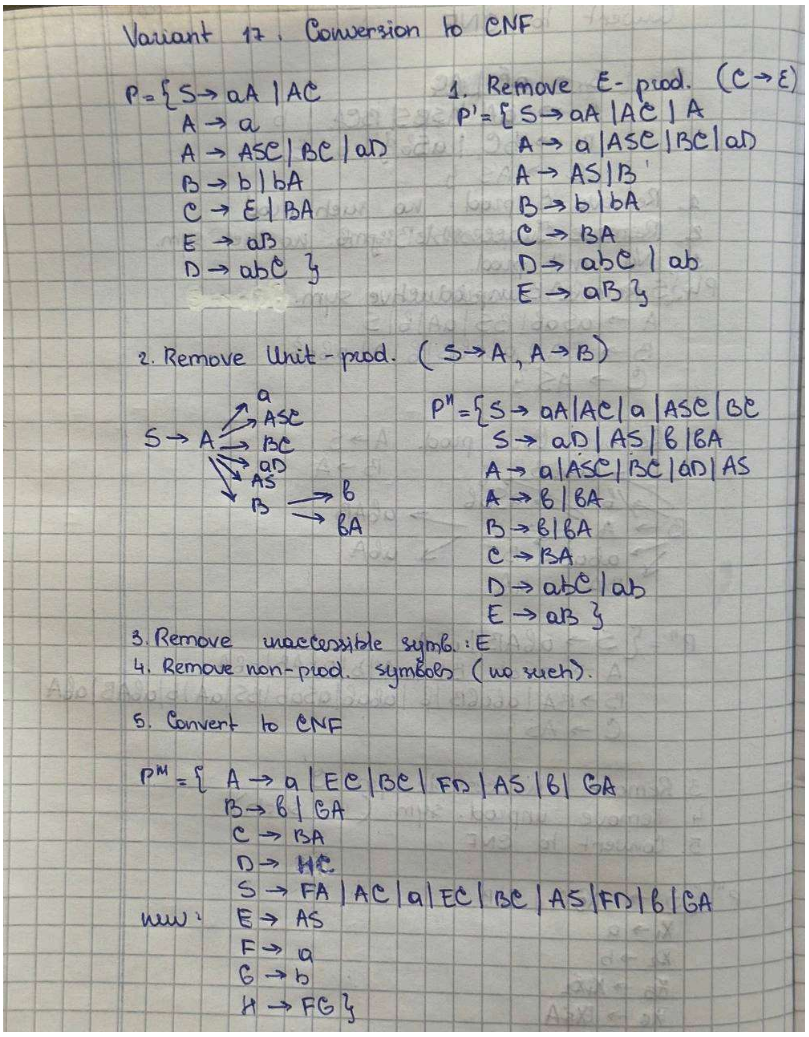
Conversion to CNF

The `toCNF` method starts by initializing some Maps to manage new and existing productions and to track how terminals are replaced by non-terminals, and a Set of non-terminals is maintained to ensure new non-terminals do not duplicate existing ones. Then, it continues with iterating through each production in the grammar and identifies which productions do not require adjustment `rhs.length() == 1 && Vt.contains(rhs.charAt(0))) || (rhs.length()==2 && Vn.contains(rhs.charAt(0)) && Vn.contains(rhs.charAt(1)))`, meaning they are already in the form of either `A->a` or `A->AB` and are directly added to the resulting productions. The next step is to prepare the replacements in the productions that are not in the required form, and especially the terminals' replacements, which are transformed by introducing new non-terminal symbols. Once all terminals are transformed to non-terminals, we can proceed to split the productions to meet the length's requirements. In the same iteration through the P Map, we split these by introducing additional non-terminals for each `subList(0, 2)` from the production. Place the given production in the corresponding Maps and remove the two replaced characters `modifiedProduction = new ArrayList<>(modifiedProduction.subList(2, modifiedProduction.size()))` and add the new non-terminal to the beginning of the list for the next production's iteration in the `while(modifiedProduction.size() > 2)` loop. After the production is fully processed and transformed, we form a String from the given List and add it to the corresponding `newProductions` Map.

Unit Tests

To ensure that the `ChomskyNormalForm` class performs its transformations accurately, I've implemented a comprehensive suite of unit tests using JUnit as the testing framework. This choice is motivated by JUnit's widespread adoption, robust features, and ease of integration with IDEs and build systems, making it a standard for Java unit testing.

I've designed 5 tests for my Variant, where I check the corectness of each method's perfomance, with the use of JUnit's `assertEquals(Object expected, Object actual)`, where `actual` is the method's results and `expected` represents the computed outcomes presented in Fig. 1 and manually introduced within the program.



Variant 17. Conversion to CNF

P = { S → aA | AC
A → a
A → ASC | BC | aD
B → b | bA
C → ε | BA
E → aB
D → abC }

1. Remove ε-prod. (C → ε)
P' = { S → aA | AC | A
A → a | ASC | BC | aD
A → AS | B
B → b | bA
C → BA
D → abC | ab
E → aB }

2. Remove Unit-prod. (S → A, A → B)

$$\begin{tikzcd} S \arrow[r] & A \arrow[r] \arrow[r] & a, ASC, BC, aD, AS \\ & & B \arrow[r] & b, bA \end{tikzcd}$$

P'' = { S → aA | AC | a | ASC | BC
S → aD | AS | b | bA
A → a | ASC | BC | aD | AS
A → b | bA
B → b | bA
C → BA
D → abC | ab
E → aB }

3. Remove inaccessible symb. : E
4. Remove non-prod. symbols (no such).

5. Convert to CNF

P''' = { A → a | EC | BC | FD | AS | b | GA
B → b | GA
C → BA
D → HC
S → FA | AC | a | EC | BC | AS | FD | b | GA
new: E → AS
F → a
G → b
H → FG }

Figure 1. Manual Conversion to CNF V. 17

Additionally, I have designed tests to verify that after executing the transformation methods, there are no epsilon productions, unit productions, and other undesirable elements left at a specific step. These checks are similar to the ones performed within each method described above and the elements' presence is stored in a `boolean` flag, so the success is appreciated according to its value. Also, an important note is that these checks will be performed on a different variant than mine (let's say the 8th one).

All test methods are organized under the `ChomskyNormalFormTest` class in the test/java directory. This conventional placement ensures that tests are easily accessible and distinctly separated from the main application code.

Conclusions / Screenshots / Results

In order to better visualize what happens at each specific steps, I'll use the Grammar's `printGrammar()` method implementend in Laboratory 1, as Grammar class is the ChomskyNormalForm's parent class, and will call it after each method's execution.

In the beggining of the conversion to CNF trip, we are facing the following Grammar (the one presented in V. 17):

```
Non-terminals (Vn): [S, A, B, C, D, E]
Terminals (Vt): [a, b]
Start Symbol (S): S
Production Rules (P):
  A -> a
  A -> ASC
  A -> BC
  A -> aD
  B -> b
  B -> bA
  S -> aA
  S -> AC
  C -> ε
  C -> BA
  D -> abC
  E -> aB
```

After calling the `removeEpsilonProductions()` method, we are left with the Grammar presented below. Observe that the ε-productions did dissapear and some more like `S -> A`, `A -> AS`, `A -> B` and `D -> ab` did appear.

```
Non-terminals (Vn): [S, A, B, C, D, E]
Terminals (Vt): [a, b]
Start Symbol (S): S
Production Rules (P):
  A -> a
  A -> ASC
  A -> BC
  A -> AS
  A -> B
  A -> aD
  B -> b
  B -> bA
  S -> aA
  S -> A
  S -> AC
  C -> BA
  D -> ab
  D -> abC
  E -> aB
```

Then, we apply the `removeUnitProductions()` method on the obtained Grammar from the previous step. Observe the resulting Grammar presented above, the productions `S -> A` and `A -> B` did disappear and some others like `A -> bA`, `A -> b`, `S -> ASC`, `S -> BC`, `S -> AS`, `S -> aD`, `S -> b` and `S -> bA` did appear.

```
Non-terminals (Vn): [S, A, B, C, D, E]
Terminals (Vt): [a, b]
```

```
Start Symbol (S): S
Production Rules (P):
  A -> a
  A -> ASC
  A -> BC
  A -> AS
  A -> aD
  A -> b
  A -> bA
  B -> b
  B -> bA
  C -> BA
  S -> aA
  S -> AC
  S -> a
  S -> ASC
  S -> BC
  S -> AS
  S -> aD
  S -> b
  S -> bA
  D -> ab
  D -> abC
  E -> aB
```

We then apply `removeInaccessibleSymbols()` method and observe that the production with E did dissapeard from Production RUles (P) as well as from Non-terminals (Vn).

```
Non-terminals (Vn): [A, B, C, S, D]
Terminals (Vt): [a, b]
Start Symbol (S): S
Production Rules (P):
  A -> a
  A -> ASC
  A -> BC
  A -> AS
  A -> aD
  A -> b
  A -> bA
  B -> b
  B -> bA
  C -> BA
  S -> aA
  S -> AC
  S -> a
  S -> ASC
  S -> BC
  S -> AS
  S -> aD
  S -> b
  S -> bA
```

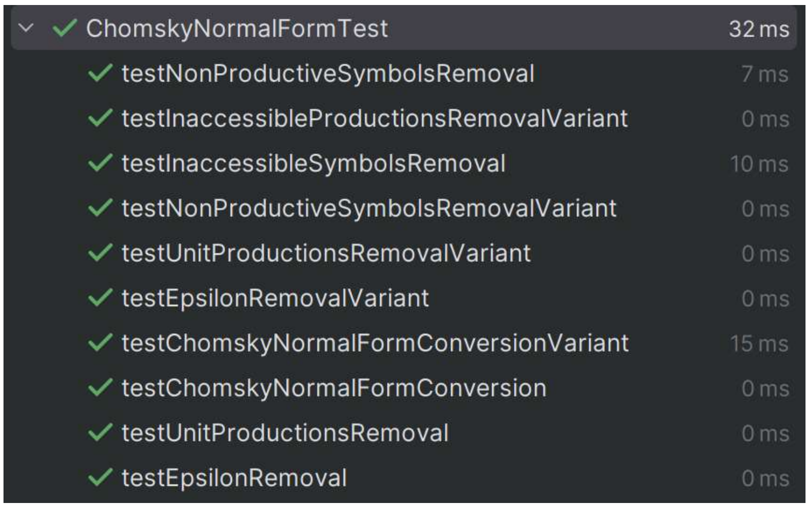
```
    D -> ab
    D -> abC
```

No changes are applied when calling `removeNonProductiveSymbols()` so I won't attach the result, just trust me :D

The last step is to convert the given grammar to CNF, such that all its productions will be in the form `A->BC` and `A->a`. After calling the `toCNF()` method, the resulting Grammar turns out to have new non-terminals that are *E*, *F*, *G*, *H* and the corresponding productions containing them. See below:

```
  Non-terminals (Vn): [A, B, C, S, D, E, F, G, H]
  Terminals (Vt): [a, b]
  Start Symbol (S): S
  Production Rules (P):
    A -> a
    A -> EC
    A -> BC
    A -> AS
    A -> FD
    A -> b
    A -> GA
    B -> b
    B -> GA
    S -> FA
    S -> AC
    S -> a
    S -> EC
    S -> BC
    S -> AS
    S -> FD
    S -> b
    S -> GA
    C -> BA
    D -> FG
    D -> HC
    E -> AS
    F -> a
    G -> b
    H -> FG
```

And of course have a look at the test results from the `ChomskyNormalFormTest` class that are presented in Fig. 2



```
ChomskyNormalFormTest                          32 ms
    testNonProductiveSymbolsRemoval             7 ms
    testInaccessibleProductionsRemovalVariant   0 ms
    testInaccessibleSymbolsRemoval             10 ms
    testNonProductiveSymbolsRemovalVariant      0 ms
    testUnitProductionsRemovalVariant           0 ms
    testEpsilonRemovalVariant                   0 ms
    testChomskyNormalFormConversionVariant     15 ms
    testChomskyNormalFormConversion             0 ms
    testUnitProductionsRemoval                  0 ms
    testEpsilonRemoval                          0 ms
```

Figure 2. Unit Tests' Results

In conclusion, I've managed to understand the essence of Chomsky Normal Form, how to implement it and perform an accurate conversion from a "messy" looking Context-Free Grammar to an "iconic" grammar, a kind of glow up :D. For testing the ChomskyNormalForm in action, I've written 10 unit-tests, 5 of them being variant-specific and the remaining just some general check-up. It was a hard work, but I'm proud that I'm finally writing this conclusion :)

References

[1] **Context Free Grammars. Chomsky Normal Form** - Accessed April 14, 2024. https://drive.google.com/file/d/19muyiabGeGaoNDK-7PeuzYYDe6_c0e-t/view.