Ministry of Education, Culture, and Research of the Republic of MoldovaTechnical University of Moldova
The Faculty of Computers, Informatics, and Microelectronics

REPORT

Laboratory work no.5
Regular expressions

Executed by: Cornievschi Bogdan
gr. FAF-222

Verified by:
univ. assist. Crețu Dumitru

Chişinău - 2024

Introduction

In the field of computer science, context-free grammars (CFGs) are foundational to the understanding and processing of formal languages, particularly in the domain of compilers and natural language processing. The objective of this laboratory exercise is to illuminate the process of converting CFGs into Chomsky Normal Form (CNF), a form where each production rule is restricted to very specific formats. The transformation into CNF is not just a theoretical exercise but a prerequisite for several practical algorithms, including the renowned Cocke-Younger-Kasami (CYK) algorithm, which necessitates grammars in CNF for efficient parsing.

CFGs serve as the backbone for specifying the syntax of programming languages and expressing the rules that govern the structure of sentences in natural languages. As such, converting a CFG into an equivalent CNF is an essential step that aids in the simplification and standardization of grammar, thereby facilitating easier and more efficient parsing and analysis.

Explanation of the code

- **start_removal**: The purpose of this function is to ensure that the start symbol of the grammar does not appear on the right-hand side of any production. This is necessary because, during the CNF conversion process, the start symbol needs to be preserved to maintain the language of the grammar. If the start symbol is found on the right-hand side, we introduce a new start symbol and update the production rules accordingly.
  Example: Consider a grammar where the start symbol S appears in a production like S -> S a | b. The function would introduce a new start symbol X and update the productions to X -> S a | b.
- **rm_null_productions**: This function eliminates productions that can derive an empty string (ε). This is done by removing such productions and updating other productions that contain the nullable non-terminals to include combinations without the nullable parts.
  Example: Given a rule B -> ε, and another rule A -> aBb, the latter would be updated to A -> ab | aBb, removing the possibility of B producing an empty string.
- **rm_unit_productions**: Unit productions are rules where a non-terminal is directly replaced by another non-terminal. This function removes such productions by replacing them with the productions of the non-terminal on the right-hand side of the rule.

Example: For a production A -> B and B -> a, the function will remove A -> B and replace it with A -> a.

- **rm_inaccessible_symb**: Inaccessible symbols are non-terminals that cannot be derived from the start symbol. This function removes any non-terminals and their productions if they are not accessible from the start symbol.
  Example: If C is not accessible from the start symbol in a grammar, and we have C -> c, this rule will be removed from the grammar.
- **repl_tem_with_non**: When a terminal appears in a production with other symbols, this function introduces a new non-terminal for that terminal and updates the productions so that no terminal appears alongside a non-terminal.
  Example: For the production A -> aB, a new non-terminal T_a is introduced with the production T_a -> a, and A's production is updated to A -> T_a B.
- **reduce_pr_length**: The function shortens the length of the productions to comply with CNF, which restricts productions to have at most two non-terminals or a single terminal. Longer productions are broken down by introducing new non-terminals.
  Example: A production like A -> BCDE would be transformed to A -> BX, X -> CY, Y -> DE, introducing new non-terminals X and Y to ensure all productions have at most two symbols.

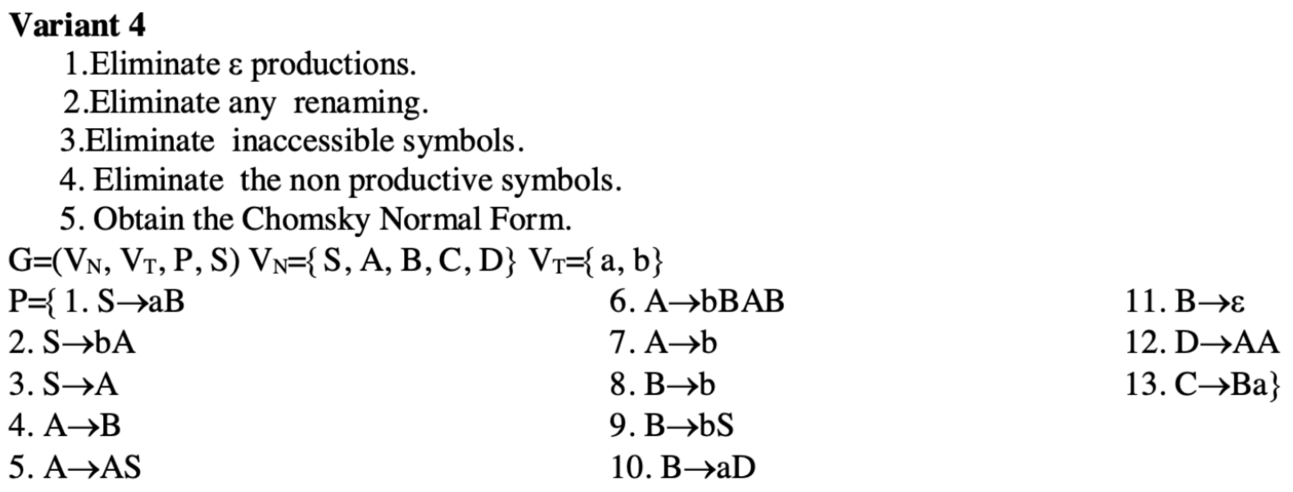
Unit testing

- **test_start_removal**: This test checks that the start symbol does not appear on the right-hand side (RHS) of any production after calling start_symbol_rhs_removal. The assertNotIn method from unittest is used to ensure that the start symbol is not found in the RHS of any production rules, which would violate CNF rules.
- **test_rm_null_productions**: After executing remove_null_productions, this test goes through all the production rules and verifies that none of them produce an epsilon ('ε'). The test uses assertNotIn to ensure that 'ε' is not present, confirming that all null productions have been successfully removed.
- **test_rm_unit_productions**: This test calls remove_unit_productions and then checks each production to make sure there are no unit productions left. A unit production is a rule where a non-terminal produces another single non-terminal. The assertFalse method asserts that no such unit productions exist after the method's execution.

- **test_rm_inaccessible_symb**: This test appends a non-terminal 'Z' that is not reachable from the start symbol and then calls remove_inaccessible_symbols. It uses assertNotIn to confirm that 'Z' is not part of the non-terminals (Vn) after removal, ensuring no inaccessible symbols are left.

- **test_repl_tem_with_non**: To verify that terminals have been correctly replaced with non-terminals in rules with more than one symbol, this test checks after replace_terminals_with_nonterminals that every character in productions longer than one symbol belongs to the set of non-terminals. The assertTrue method asserts the correctness of this replacement.
- **test_reduce_pr_length**: This test confirms that after reduce_production_length, all productions have at most two symbols (for non-terminal productions) or one symbol (for terminal productions), which is a requirement of CNF. The assertTrue method is used to validate that no production exceeds this length.
- **test_grammar_print**: While not a test of functionality, this is included to check the method print_grammar runs after the full conversion to CNF without errors. It doesn't assert anything but is useful for debugging and verifying the output.

My variant



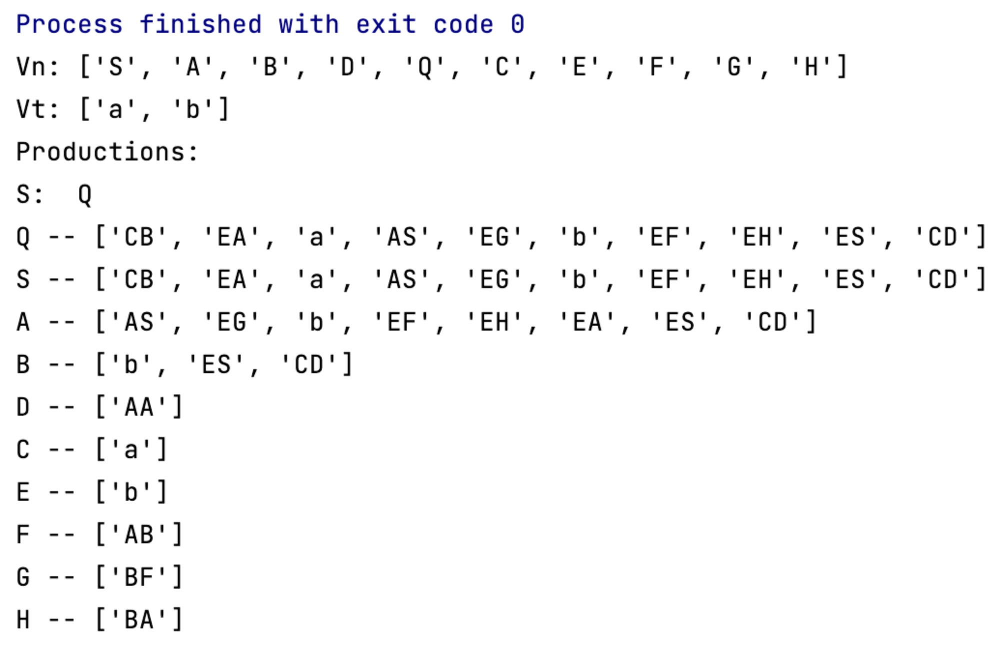
Variant 4

1.Eliminate ε productions.
2.Eliminate any renaming.
3.Eliminate inaccessible symbols.
4. Eliminate the non productive symbols.
5. Obtain the Chomsky Normal Form.

G=(V_N, V_T, P, S) V_N={ S, A, B, C, D} V_T={ a, b}

P={ 1. S→aB	6. A→bBAB	11. B→ε
2. S→bA	7. A→b	12. D→AA
3. S→A	8. B→b	13. C→Ba}
4. A→B	9. B→bS	
5. A→AS	10. B→aD	

Output of the programs



```
Process finished with exit code 0
Vn: ['S', 'A', 'B', 'D', 'Q', 'C', 'E', 'F', 'G', 'H']
Vt: ['a', 'b']
Productions:
S:  Q
Q -- ['CB', 'EA', 'a', 'AS', 'EG', 'b', 'EF', 'EH', 'ES', 'CD']
S -- ['CB', 'EA', 'a', 'AS', 'EG', 'b', 'EF', 'EH', 'ES', 'CD']
A -- ['AS', 'EG', 'b', 'EF', 'EH', 'EA', 'ES', 'CD']
B -- ['b', 'ES', 'CD']
D -- ['AA']
C -- ['a']
E -- ['b']
F -- ['AB']
G -- ['BF']
H -- ['BA']
```

Upon completion of the test suite, we observed a successful execution with all seven tests passing, as reflected by the outcome "OK" and an exit code of 0. This quick execution time of 0.002 seconds underscores the efficiency of our test cases and the underlying functions they evaluate.

This output showcases the newly created non-terminal symbols (Q, C, E, F, G, H) introduced during the terminal replacement and production reduction phases. Each production now adheres to the stringent structural constraints of CNF—every production rule is either a binary expansion of non-terminals or a single terminal generation.

The absence of any inaccessible symbols and the presence of newly introduced non-terminals to encapsulate terminal symbols in larger productions are also notable. These transformations are crucial in ensuring that the grammar is compliant with CNF requirements while maintaining the language it defines.

In conclusion, the successful test results, accompanied by the satisfactory transformed grammar output, confirm the effectiveness of our CFG to CNF conversion process. This validates not only the correctness of each individual function but also their collective operation within the comprehensive conversion algorithm.

Conclusion

The objective of this laboratory exercise—to transform a context-free grammar into Chomsky Normal Form—has been successfully achieved. The suite of tests designed to validate each step of the transformation process ran flawlessly, as confirmed by the all-passing results. This outcome substantiates the correctness of our implementation and the reliability of our transformation logic.

Through meticulous development and thorough testing, we ensured that each function performed its role effectively—from removing null and unit productions to replacing terminals with non-terminals and shortening production lengths. The result is a grammar that strictly adheres to the structural restrictions of CNF, thereby facilitating its use in parsing algorithms and other computational linguistics applications.

Moreover, the rapid execution time of the tests illustrates the potential for this transformation process to be applied efficiently in real-time systems, a testament to both the algorithm's design and the Python language's capabilities.

Despite these successes, it is important to acknowledge that the scope of this lab was limited to the theoretical framework provided by the context-free grammars and did not explore the empirical performance of the CNF on actual parsing tasks. As such, future work could involve applying the transformed grammars to parsing algorithms, such as the CYK algorithm, and evaluating their performance in practical scenarios.