Laboratory Work Nr.5 Chomsky Normal Form Converter

Cretu Cristian

April 29, 2024

Theory

In formal language theory, a context-free grammar, G, is said to be in Chomsky normal form (first described by Noam Chomsky) if all of its production rules are of the form:
$A \rightarrow BC$, or
$A \rightarrow a$, or
$S \rightarrow \epsilon$,
where A, B, and C are nonterminal symbols, the letter a is a terminal symbol (a symbol that represents a constant value), S is the start symbol, and ϵ denotes the empty string.
Also, neither B nor C may be the start symbol, and the third production rule can only appear if ϵ is in L(G), the language produced by the context-free grammar G. Every grammar in Chomsky normal form is context-free, and conversely, every context-free grammar can be transformed into an equivalent one which is in Chomsky normal form and has a size no larger than the square of the original grammar's size.

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.
   (a) The implementation needs to be encapsulated in a method with an appropriate signature (also ideally in an appropriate class/type).
   (b) The implemented functionality needs executed and tested.
   (c) A BONUS point will be given for the student who will have unit tests that validate the functionality of the project.

(d) 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 description

To execute the conversion of a context free grammar to the Chomsky Normal Form, there are some operations that help us do that. This is why I chose to implement them separately in each class.

START Operation

START: Eliminate the start symbol from right-hand sides.
Introduce a new start symbol S0, and a new rule
S0 → S,
where S is the previous start symbol. This does not change the grammar's produced language, and S0 will not occur on any rule's right-hand side.
Implementation:

```
class StartOperation:
    @staticmethod
    def do(P: dict = None, S: str = None, Vn: list = None):
        try:
            for value in P.values():
                for production in value:
                    for character in production:
                        if character == S:
                            raise ValueError('S is in a
    production')
        except:
            new_P = {'X': [S]}
            new_P.update(P)
            S = 'X'
            Vn.append(S)
            P = new_P

        return P, S, Vn
```

This code aims to ensure that the start symbol of a grammar is not present in any of its production rules. If the start symbol is found in a production rule, it raises an error. Otherwise, it replaces the start symbol with a new symbol 'X', adds a new production rule for 'X', and updates the start symbol accordingly, ensuring the grammatical correctness of the input.

TERM Operation

To eliminate each rule
A $\rightarrow X_1$ ... a ... X_n
with a terminal symbol a being not the only symbol on the right-hand side, introduce, for every such terminal, a new nonterminal symbol Na, and a new rule
Na $\rightarrow$ a.
Change every rule
A $\rightarrow X_1$... a ... X_n
to
A $\rightarrow X_1$... N_a ... X_n.
If several terminal symbols occur on the right-hand side, simultaneously replace each of them by its associated nonterminal symbol. This does not change the grammar's produced language.

```python
class TermOperation:
    @staticmethod
    def do(P: dict = None, S: str = None, Vn: list[str] = None, Vt: list[str] = None):
        new_P = TermOperation.replace_terminals_with_nonterminals(P, Vn, Vt)
        return new_P

    @staticmethod
    def replace_terminals_with_nonterminals(P: dict, Vn: list[str], Vt: list[str]) -> dict:
        terminal_to_nonterminal = {}
        new_P = P.copy()  # Initialize new_P with a copy of P
        for key, productions in P.items():
            new_productions = []
            for prod in productions:
                new_prod = TermOperation.replace_terminal(prod, Vt, terminal_to_nonterminal, Vn, new_P)
                new_productions.append(new_prod)
            new_P[key] = new_productions
        return new_P
```

The `TermOperation` class provides functionality to replace terminal symbols with non-terminal symbols in the production rules of a grammar. The `do` method orchestrates this operation by invoking the `replace_terminals_with_nonterminals` method. Within `replace_terminals_with_nonterminals`, each terminal symbol in the grammar's production rules is replaced with a corresponding non-terminal symbol, ensuring compliance with the rules of Chomsky Normal Form (CNF). The resulting modified production rules are then returned for further processing or analysis.

BIN Operation

Replace each rule
A $\rightarrow X_1\ X_2\ ...\ X_n$
with more than 2 nonterminals X_1,...,X_n by rules
A $\rightarrow X_1\ A_1$,
A1 $\rightarrow X_2\ A_2$,
... ,
An-2 $\rightarrow X_{n-1}\ X_n$,
where A_i are new nonterminal symbols. Again, this does not change the grammar's produced language.

```
class BinOperation:
    @staticmethod
    def do(P: dict[str, list[str]] = None, Vn: list[str] =
    None) -> dict[str, list[str]]:
        if P is None or Vn is None:
            return {}

        existing_binaries = {}
        new_productions_dict = {}
        for key, productions in P.items():
            new_productions = []
            for production in productions:
                if len(production) > 2:
         ...
```

DEL Operation

An ϵ-rule is a rule of the form $A \rightarrow \varepsilon$, where A is not the start symbol S_0 of the grammar. To eliminate all rules of this form, first determine the set of all nonterminals that derive ϵ, known as nullable nonterminals. Nullable nonterminals are computed as follows:

- If a rule $A \rightarrow \varepsilon$ exists, then A is nullable.
- If a rule $A \rightarrow X_1...X_n$ exists, and every single X_i is nullable, then A is nullable, too.

Obtain an intermediate grammar by replacing each rule $A \rightarrow X_1...X_n$ by all versions with some nullable X_i omitted. Then, delete each ϵ-rule, unless its left-hand side is the start symbol S_0. The resulting transformed grammar is obtained. For example, in the provided grammar, the nonterminal A (and hence B) is nullable, while neither C nor S_0 is. After intermediate steps, the ϵ-rules are removed, yielding a grammar without ϵ-rules.

UNIT Operation

A unit rule is a rule of the form $A \rightarrow B$, where A and B are nonterminal symbols. To remove unit rules, for each rule $B \rightarrow X_1...X_n$, where $X_1...X_n$ is a string of nonterminals and terminals, add a new rule $A \rightarrow X_1...X_n$, unless this is a unit rule which has already been (or is being) removed. The skipping of nonterminal symbol B in the resulting grammar is possible due to B being a member of the unit closure of nonterminal symbol A.

Order of operations

When choosing the order in which the above transformations are to be applied, it has to be considered that some transformations may destroy the result achieved by other ones. For example, START will re-introduce a unit rule if it is applied after UNIT. The table shows which orderings are admitted. Moreover, the worst-case bloat in grammar size depends on the transformation order. Using $|G|$ to denote the size of the original grammar G, the size blow-up in the worst case may range from $|G|^2$ to 2^2 $|G|$, depending on the transformation algorithm used. The blow-up in grammar size depends on the order between DEL and BIN. It may be exponential when DEL is done first, but is linear otherwise. UNIT can incur a quadratic blow-up in the size of the grammar. The orderings START,TERM,BIN,DEL,UNIT and START,BIN,DEL,UNIT,TERM lead to the least (i.e. quadratic) blow-up.

Testing

The testing of correct execution is done by the `TestGrammar` class that executes the operations on the grammar and returns a new set of production rules. By iterating over them, we can see which operations succeeded or failed.

Execution

To showcase the execution of the program, the variant is restructured in a '.json' file and read using a 'Variant' class. Then, with all functionality encapsulated in the `ChomskyNormalFormConverter` class, the operations are executed after testing each operation separately in the `TestGrammar` class.

```
from ChomskyNormalFormConverter import
    ChomskyNormalFormConverter
from Variant import Variant
from Grammar import Grammar
from UnitTests import TestGrammar
variant = Variant(
'/Users/cristiancretu/
Documents/UniCode/
LFA/Chomsky Normal Form/variant.json')
production = variant.getP()
```

```
terminals = variant.getVT()
non_terminals = variant.getVN()
grammar = Grammar(terminals=terminals, non_terminals=
    non_terminals, productions=production)
converter = ChomskyNormalFormConverter(grammar)
print(grammar)
test = TestGrammar(grammar=grammar, converter=converter)
test.run()
print("All tests passed",end="\n\n")
new_grammar = converter.transform()
print(new_grammar)
```

This outputs:

```
Vn=['S', 'A', 'B', 'C', 'E']
Vt=['a', 'b']
Start_symbol=S
Productions=
S -> ['bA', 'B']
A -> ['a', 'AS', 'bAaAb']
B -> ['AC', 'bS', 'aAa']
C -> ['', 'AB']
E -> ['BA']

START test passed
DEL test passed
UNIT test passed
BIN test passed
TERM test passed
All tests passed

Vn=['S', 'A', 'B', 'C', 'X', 'D', 'F', 'G', 'H', 'X', 'K', 'L']
Vt=['a', 'b']
Start_symbol=X
Productions=
X -> ['KA', 'AC', 'KS', 'LD', 'L', 'AS', 'KH']
S -> ['KA', 'AC', 'KS', 'LD', 'L', 'AS', 'KH']
A -> ['L', 'AS', 'KH']
B -> ['AC', 'KS', 'LD', 'L', 'AS', 'KH']
C -> ['AB']
D -> ['AL']
F -> ['AK']
G -> ['LF']
H -> ['AG']
K -> ['b']
L -> ['a']
```

Conclusions

In summary, the CNF converter implementation demonstrates a modular design with each transformation operation encapsulated in its own class, promoting code reusability and readability. Static methods within each class allow for direct invocation of operations, simplifying usage. Error handling mechanisms are incorporated to address potential issues during transformation. Additionally, iterative approaches, like in the UNIT operation, efficiently eliminate unit rules until convergence is achieved. These implementation insights highlight a systematic and robust approach to grammar normalization, ensuring the reliability and versatility of the CNF converter.