MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII

AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

Copta Adrian | FAF-223

Report

Laboratory work n.5

of Formal Languages & Finite Automata

Checked by:

Cretu Dumitru, *university assistant*

DISA, FCIM, UTM

Chișinău – 2024

1. 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 $S \rightarrow \varepsilon$,

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.

2. Purpose of the task work:

1. Learn about Chomsky Normal Form (CNF).
2. Get familiar with the approaches of normalizing grammar.
3. 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.
    - A BONUS point will be given for the student who will have unit tests that validate the functionality of the project.
    - 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.

3. Implementation description:

To implement a program which transfers a Context Free Grammar to Chomsky Normal Form, we designed a class called CNF where we implemented the steps of the transfer algorithm learned at the seminars. Also we created a helper.py which will help us with the routine of loading and uploading the grammar, and other functions.

In the main.py we create an object of the CNF class and parse it to it, after we call the transform function

```
import sys
from CNF import CNF

if __name__ == '__main__':
    CFG = sys.argv[1] if len(sys.argv) > 1 else 'model.txt'
    cnf_transformer = CNF(CFG)
    cnf_transformer.transform()
```

The transform function calls each functions of the algorithm by order, and then prints the converted grammar to console and oslo to an output.txt

```
    def transform(self):
        self.START()
        self.TERM()
        self.BIN()
        self.DEL()
        self.UNIT()
        print(helper.prettyForm(self.P))
        print(len(self.P))
        open('out.txt', 'w').write(helper.prettyForm(self.P))
```

Let's break down each function one by one:

1. START(self): Adds a new start variable and adjusts productions accordingly.

```
    def START(self):
        self.V_n.append('S0')
        self.P = [('S0', [self.V_n[0]])] + self.P
```

2. TERM(self): Transforms non-simple productions into simple ones by introducing new variables for terminals.

```
    def TERM(self):
        newProductions = []
        dictionary = helper.setupDict(self.P, self.V_n, terms=self.V_t)
        for production in self.P:
            if self.isSimple(production):
                newProductions.append(production)
            else:
                for term in self.V_t:
                                                    for index, value in
enumerate(production[self.right]):
                        if term == value and not term in dictionary:
                            dictionary[term] = self.V_nariablesJar.pop()
                            self.V_n.append(dictionary[term])
                                newProductions.append((dictionary[term],
[term]))
                                                production[self.right][index] =
dictionary[term]
                        elif term == value:
                                                production[self.right][index] =
dictionary[term]
                            newProductions.append((production[self.left],
production[self.right]))
        self.P = newProductions
```

The TERM method, the transformation of non-simple productions into simple ones is handled. It initializes an empty list newProductions to store the transformed productions. This functions works with the function isSimple in parallel:

```
    def isSimple(self, rule):
            if rule[self.left] in self.V_n and rule[self.right][0] in
self.V_t and len(rule[self.right]) == 1:
            return True
        return False
```

Next, it iterates over each production in the grammar. If a production is simple according to the isSimple method, it directly adds it to the newProductions list. Otherwise, it iterates over each terminal in the set of terminals (V_t). For each terminal encountered in the production's right-hand side, it checks if it's already present in the dictionary. If not, it assigns a new variable from VariablesJar to represent the terminal, updates the dictionary, and adds a new production to newProductions where

the terminal is replaced by its corresponding variable. Otherwise, if the terminal is already in the dictionary, it replaces it with its corresponding variable in the production's right-hand side. After processing all productions, the method updates the grammar (self.P) with the transformed productions stored in newProductions.

3. BIN(self): Converts productions with more than two symbols on the right side into binary productions.

```
    def BIN(self):
        result = []
        for production in self.P:
            k = len(production[self.right])
            if k <= 2:
                result.append(production)
            else:
                newVar = self.V_nariablesJar.pop(0)
                self.V_n.append(newVar + '1')
                                    result.append((production[self.left],
[production[self.right][0]] + [newVar + '1']))
                for i in range(1, k - 2):
                    var, var2 = newVar + str(i), newVar + str(i + 1)
                    self.V_n.append(var2)
                          result.append((var, [production[self.right][i],
var2]))
                                    result.append((newVar + str(k - 2),
production[self.right][k - 2:k]))
        self.P = result
```

The method handles the conversion of productions into binary form. It iterates over each production in the grammar (self.P). For each production, it checks the number of symbols on the right-hand side (k). If the production already has two or fewer symbols, indicating it's already in binary form, it simply adds it to the result list. If the production has more than two symbols on the right-hand side, indicating it's not in binary form, the method proceeds with the conversion. It starts by popping a new variable (newVar) from the VariablesJar, a list of available variables. This variable is used to replace the non-binary part of the production.

Next, it appends a new production to the result list, where the first symbol on the right-hand side remains unchanged, and the rest is replaced by the newly introduced variable newVar + '1'. Then, it iterates over the remaining symbols on the right-hand side of the production (excluding the first symbol). For each symbol, it introduces a new variable (var2) to represent the rest of the symbols. The original symbol and the newly introduced variable are appended as a binary pair to the result list.

Finally, if there are any remaining symbols after processing, the method appends a production for the last pair of symbols, ensuring all symbols are accounted for in binary form.

Once all productions are processed, the method updates the grammar (self.P) with the transformed productions stored in the result list.

4. DEL(self): Handles epsilon productions by removing them and updating affected productions. unit_routine(self, rules): Processes unit productions by eliminating them.

```
    def DEL(self):
        newSet = []
                    outlaws, self.P = helper.seekAndDestroy(target='e',
productions=self.P)
        for outlaw in outlaws:
                for production in self.P + [e for e in newSet if e not in
self.P]:
                    if outlaw in production[self.right]:
                          newSet = newSet + [e for e in helper.rewrite(outlaw,
```

```
production) if e not in newSet]
        self.P = newSet + ([p for p in self.P if p not in newSet])
```

The method initializes an empty list called newSet to store the updated set of productions without epsilon productions. It calls the seekAndDestroy function from the helper module to identify epsilon productions. The seekAndDestroy method from helper.py:

```
def seekAndDestroy(target, productions):
 trash, ereased = [],[]
 for production in productions:
     if target in production[right] and len(production[right]) ==
1:
          trash.append(production[left])
     else:
          ereased.append(production)

 return trash, ereased
```

The seekAndDestroy function iterates over each production in the grammar to identify epsilon productions. If a production contains only epsilon on its right-hand side, it adds the left-hand side variable to the trash list. Otherwise, it adds the production to the ereased list, indicating that it doesn't contain epsilon. Finally, it returns both lists.

It iterates over each outlaw (variable with epsilon productions) in the outlaws list. For each outlaw, it iterates over all productions in the original grammar (self.P) and the new set of productions (newSet).

It checks if the outlaw variable appears on the right-hand side of any production. If the outlaw variable is found in a production's right-hand side, it calls the rewrite function from the helper module to handle the rewriting process. The function generates all possible combinations of the production with and without the outlaw variable. It updates the newSet with the rewritten productions, ensuring no duplicates are added.

Once all productions are processed, it updates the grammar (self.P) with the updated set of productions stored in newSet. It also removes any duplicate productions that might have been added.

5. UNIT(self): Applies unit production elimination until no further units exist.

```
def UNIT(self):
     i = 0
     result = self.unit_routine(self.P)
     tmp = self.unit_routine(result)
     while result != tmp and i < 1000:
         result = self.unit_routine(tmp)
         tmp = self.unit_routine(result)
         i += 1
     self.P = result
```

The unit_routine method is crucial in this process. It identifies and eliminates unit productions by iterating over each production rule and replacing unitary rules with equivalent non-unitary ones. It iterates until no more unit productions can be eliminated or until reaching the maximum iteration limit.

```
def unit_routine(self, rules):
     unitaries, result = [], []
     for aRule in rules:
         if self.isUnitary(aRule, self.V_n):
                                        unitaries.append((aRule[self.left],
aRule[self.right][0]))
         else:
```

```
                result.append(aRule)
        for uni in unitaries:
            for rule in rules:
                 if uni[self.right] == rule[self.left] and uni[self.left]
!= rule[self.left]:
                    result.append((uni[self.left], rule[self.right]))
        return result
```

The main method eliminates unit productions from the grammar through iterative refinement. It starts by setting a counter i to zero. It initializes the result with the initial set of productions after the first round of unit elimination. After, it enters a while loop, refining the grammar until no further changes occur or a maximum iteration limit (1000 iterations) is reached. In each iteration the code updates result by applying the unit_routine method, which eliminates unit productions. If result changes and the iteration limit is not exceeded, it continues the loop. Once all unit productions are eliminated or the iteration limit is reached, it updates the grammar with the refined set of productions stored in result.

4. Program execution:

To be able to test the program easily we read the CFG which is written in a .txt file and then we store the result in a output.txt file, here is the result of 2 CFGs:

input.txt:

```
Terminals:
+ - ( ) ^ number variable
Variables:
Expr Term AddOp MulOp Factor Primary
Productions:
Expr -> Term | Expr AddOp Term | AddOp Term;
Term -> Factor | Term MulOp Factor;
Factor -> Primary | Factor ^ Primary;
Primary -> number | variable;
Primary -> ( Expr );
AddOp -> + | -;
MulOp -> * | /
```

output.txt:

```
Expr -> Expr A1 | AddOp Term | Term B1 | Factor C1 | number | variable | Y D1
A1 -> AddOp Term
Term -> Term B1 | Factor C1 | number | variable | Y D1
B1 -> MulOp Factor
Z -> ^
Factor -> Factor C1 | number | variable | Y D1
C1 -> Z Primary
Primary -> number | variable | Y D1
Y -> (
X -> )
D1 -> Expr X
AddOp -> + | -
MulOp -> * | /
S0 -> Expr A1 | AddOp Term | Term B1 | Factor C1 | number | variable | Y D1
```

input.txt:

```
Terminals:
a b d
Variables:
S A B C D
Productions:
S -> d B | A;
```

```
A -> a B d B | d S | d;
B -> A C | a S | a;
D -> A B
C -> b C
C -> e
```

output.txt:

```
S -> A B | B A1 | A S | d
A -> B A1 | A S | d
A1 -> B A2
A2 -> A B
B -> A C | B S | a
D -> A BC
S0 -> A B | B A1 | A S | d
```

5. Conclusion:

In conclusion, this laboratory work focused on the implementation of a Python program designed to transform Context-Free Grammars (CFGs) into Chomsky Normal Form (CNF). Through systematic coding and algorithmic design, we successfully developed a program capable of efficiently converting grammars into a standardized format suitable for various linguistic analysis and language processing tasks. By leveraging Python's versatility and the principles of formal language theory, we were able to create a robust tool for computational linguistics, contributing to our understanding of CFGs and their application in real-world language processing scenarios. This laboratory work not only provided valuable hands-on experience in programming and algorithm development but also enhanced our knowledge of formal language theory and its practical implications.