Implementation of Procedure to Convert Pushdown Automata to Context-Free Grammar

Mateus A. Fernandes A.
Dept. of Computer Science and Engineering
University of South Florida
Tampa, Florida, USA
Email: mateusf1@usf.edu

***Abstract*—This paper will describe the process of constructing a context-free grammar (CFG) from a Pushdown Automaton (PDA). An algorithm was implemented using a C++ program that will transform a set of PDA transitions rules to the set of productions that describe a CFG. The transformation from PDA to CFG is essential in understanding the relationship each has to one another for representing context-free languages (CFL).**

Keywords—CFL, context-free languages, CFG, context-free grammar, PDA, pushdown automata, conversion.

I. Introduction

In automata theory and formal languages, it is essential to understand the principles of how languages are defined. Context-free languages (CFL) have many applications in computer science, such as describe the nested structure found in programming languages [1]. Similarly to understand a spoken language such as English, a CFL must conform to a set of rules that define the language. These rules are referred to as the grammar of a language, or context-free grammars (CFG) for CFL. Furthermore, for all CFL, there exists a type of automaton that defines them; it is known as pushdown automata (PDA) [2]. Since both CFG and PDA can define the set of all context-free languages, it is not difficult to infer that there exists a path to develop one from the other. This paper explores such a path to convert PDA to CFG.

II. Elementary Definitions

A. Pushdown Automata Elements

A pushdown automaton (PDA) can be represented as a septuple of the following form [1]:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$$

where: Q = finite set of states,
Σ = input alphabet (a finite set of symbols),
Γ = stack alphabet (a finite set of symbols),
δ = transition function from $Q \times (\Sigma \cup \{ \lambda \}) \times \Gamma$ to the set of finite subsets of $Q \times \Gamma^*$.
q_0 = initial state ($q_0 \in Q$),
Z = stack start symbol ($Z \in \Gamma$),
F = finite set of final states ($F \subseteq Q$).

For this algorithm, the set of final states F will not be necessary since the PDA must be configured to accept by empty stack. See the Preliminary step under the "Procedures to Convert PDA to CFG" section.

B. Context-Free Grammar Elements

A context-free grammar (CFG) can be represented as a quadruple of the following form [1]:

$$G = (V, T, S, P)$$

where: V = finite set of variables (disjoint to T),
T = finite set of terminal symbols (often $T = \Sigma$),
S = start variable ($S \in V$),
P = finite set of productions of the form $A \rightarrow x$, where $A \in V$ and $x \in (V \cup T)^*$.

Usually, the variables are represented as a single character. But, to better follow-up and understand the relationship between the PDA and CFG, the variables in the produced CFG will be the form:

[pAq]

This variable represents sequences of moves that take the PDA from state p to q and have the ultimate effect of removing A from the stack [2].

III. Procedures to Convert PDA to CFG

A. Preliminary step

Before starting to convert the PDA to CFG, there is a preliminary step that should be checked or done on the PDA for the algorithm to function correctly. There are two ways the PDA could be configured to accept a particular input string from the language: acceptance by final state or acceptance by empty stack. For the conversion algorithm, it requires to use the "acceptance by empty stack," otherwise it will be challenging to keep track of the final states. However, if a PDA with final state configuration M_1 is given, there is another equivalent PDA with empty stack acceptance configuration M_2 that can be obtained [2]. This can be done so by first adding a new initial state that will push in a new starting stack symbol first in the stack. This prevents the PDA from emptying the stack prematurely before reaching the given final states. Then, M_2 should be provided with another state where all the M_1 final states will *λ–transition* into, from which there are another *λ–transitions* looping into itself to pop all the stack symbols until empty stack automatically [2].

B. Converting PDA transitions to possible CFG productions

To convert a given PDA into a CFG, all the list of transition rules, set of all possible states (Q), starting state (q_0), and starting stack symbol (Z) in the PDA must be identified first. Using the PDA transition rules given, an equivalent list (or family) of possible CFG productions can be generated for each transition rule depending on their format. There are two general formats that each transition rules can be formed, as follows:

1) If $\delta(p, a, A)$ contains (q, λ)

where: p = current state ($p \in Q$),
a = an input symbol ($a \in \Sigma \cup \{\lambda\}$),
A = a symbol to pop from top of stack ($A \in \Gamma$),
q = next state to transition to ($q \in Q$).

This PDA transition rule indicates that to pop the stack symbol A, it will require only one input symbol a with a direct transition from state p to q [2]. Then, it will only have possible equivalent CFG production, which is in the following format:

$$[pAq] \rightarrow a$$

2) If $\delta(p, a, A)$ contains $(q, Y_1Y_2Y_3 \ldots Y_k)$ for $(k \geq 1)$

where: p = current state ($p \in Q$),
a = an input symbol ($a \in \Sigma \cup \{\lambda\}$),
A = a symbol to pop from top of stack ($A \in \Gamma$),
q = next state to transition to ($q \in Q$),
Y_i = a symbol to push on top of stack ($Y_i \in \Gamma$).

Similarly, this PDA transition rule also indicates that to pop the stack symbol A, it will first require an input symbol a and transition from p to r. However, there will be a new series of stack symbols ($Y_1Y_2Y_3 \ldots Y_k$) that got pushed on top of the stack that must get pop with some other transition rule. This will require an unknown set of movement to any other possible state in Q [2]. Therefore, it will generate a family of all possible CFG productions in the following format:

$$[pAs_k] \rightarrow a[qY_1s_1][s_1Y_2s_2][s_2Y_3s_3] \ldots [s_{k-2}Y_{k-1}s_{k-1}][s_{k-1}Y_ks_k]$$

where: s_i = any possible state in Q [2].

C. Getting all possible starting CFG productions

The previous procedure has only created all possible intermediate and terminal CFG productions. However, this CFG is missing the initial productions and declare the initial variable (S). At least, it is known that the PDA starts at some state q_0 and end popping the starting stack symbol Z with some sequence of transitions and input string. But, because the PDA is configured to accept strings by empty stack, its final state can be any state in Q; it is irrelevant [2]. Therefore, the starting production can be a family of all possible productions in the following format:

$$S \rightarrow [q_0Zq]$$

IV. Additional Simplification Steps

The previous section explained an algorithm to get every possible CFG production from a given PDA. Although it is an acceptable conversion, it will produce multiple useless productions that could be eliminated. There are few cases where a production or variable may be useless in the CFG, such as:

1) Productions with undefined variables

These productions will never accept any string because it contains variables that are undefined [1]. In other words, the variable in the production does not match any of the left-hand side variables in the CFG to make the substitution. For example:

$$S \rightarrow aA \mid aAB$$
$$A \rightarrow b$$

This CFG does not have any definition for the B variable that can be substituted by. Therefore, any production with such a variable must be removed entirely, even if other variables in the same production are defined (such as A). Looking as follows:

$$S \rightarrow aA$$
$$A \rightarrow b$$

2) Variables that do not reach to terminal string

These variables will never accept any string because it will never reach a terminal string [1]. In other words, the variable has no productions, or it will infinitely loop itself. For example:

$$S \rightarrow bAa$$
$$A \rightarrow aA$$

The variable A in this CFG does not reach any terminal; it just endlessly loops itself. Therefore, this A variable and any production that contains it must be removed entirely. Looking as follows:

$$S \rightarrow$$

However, now the S variable is empty; it has no productions. Therefore, it should also be removed, getting an empty CFG.

3) Non-reachable variables from starting

Because the CFG begins with the starting variable S, some other variables could potentially never be reached from this point [1]. For example:

$$S \rightarrow aA$$
$$A \rightarrow aC$$
$$B \rightarrow cA \mid \lambda$$
$$C \rightarrow cC \mid \lambda$$

The variable B in this CFG cannot be reached from the variable S; it will never get used. Therefore, this B variable must be removed entirely. Looking as follows:

$$S \rightarrow aA$$
$$A \rightarrow aC$$
$$C \rightarrow cC \mid \lambda$$

Notice, although the variable C is not directly contained in the starting productions, it can still be reached by another reachable variable A.

With all these useless variables and productions eliminated, the CFG will become much simpler and appealing to read.

V. Implementation

With all these concepts, the procedure to convert a PDA to CFG and respective simplifications has been implemented into a single file C++ terminal or console I/O code.

A. Limitations and Conditions

Before describing the code, some additional limitations and input conditions are given to simplify the algorithm implementation. These are defined as follows:

- Each PDA transition rule can only push up to two symbols on top of the stack at the time.
- Each PDA transition rule must be written on separate lines in the terminal/console input, not on the same set.
- The PDA states in Q must be written in the format q_i (where $0 \leq i \leq 9$). No more than ten states (including q_0).

- The PDA starting state is q_0, and the starting stack symbol must be Z (not lowercased).
- Brackets, greater-than, and vertical bars symbols cannot be part of the input nor stack alphabet ([,], >, | ∉ Σ ∪ Γ). These symbols are crucial to find variables and productions.
- Because Greek letters are not in the standard C++ characters, the transition function symbol (δ) and the empty string (λ) must be replaced by the dollar sign ($) and the number or pound sign (#), respectively.

B. Description and Examples

The code begins providing the list of conditions and requesting the list of PDA transition rules and the number of states. "Fig. 1" shows an example PDA input to the respective PDA graph shown on "Fig. 2".

```
Please enter your PDA (enter DONE at
    the end of your PDA):
$(q0, a, Z) = {(q0, AZ)}
$(q0, a, A) = {(q0, AA)}
$(q0, b, A) = {(q0, AB)}
$(q0, b, A) = {(q1, A)}
$(q0, #, Z) = {(q1, Z)}
$(q1, c, A) = {(q1, #)}
$(q1, d, B) = {(q1, #)}
$(q1, #, Z) = {(q2, #)}
$(q1, x, X) = {(q1, #)}
DONE

Enter the total amount of states: 3
```

[a] The last PDA transition rule was added to test the non-reachable elimination algorithm.

Fig. 1. Example PDA input in code.

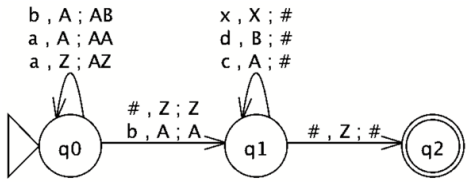

Fig. 2. Equivalent PDA graph to example input in "Fig. 1".

Ones the requested inputs were given, the program proceeds to get a list of all possible CFG productions for each PDA transition rule, as shown in "Fig. 3".

```
S --> [q0Zq0] | [q0Zq1] | [q0Zq2]
[q0Zq0] --> a[q0Aq0][q0Zq0] | a[q0Aq1][q1Zq0] | a[q0Aq2][q2Zq0]
[q0Zq1] --> a[q0Aq0][q0Zq1] | a[q0Aq1][q1Zq1] | a[q0Aq2][q2Zq1]
[q0Zq2] --> a[q0Aq0][q0Zq2] | a[q0Aq1][q1Zq2] | a[q0Aq2][q2Zq2]
[q0Aq0] --> a[q0Aq0][q0Aq0] | a[q0Aq1][q1Aq0] | a[q0Aq2][q2Aq0]
[q0Aq1] --> a[q0Aq0][q0Aq1] | a[q0Aq1][q1Aq1] | a[q0Aq2][q2Aq1]
[q0Aq2] --> a[q0Aq0][q0Aq2] | a[q0Aq1][q1Aq2] | a[q0Aq2][q2Aq2]
[q0Aq0] --> b[q0Aq0][q0Bq0] | b[q0Aq1][q1Bq0] | b[q0Aq2][q2Bq0]
[q0Aq1] --> b[q0Aq0][q0Bq1] | b[q0Aq1][q1Bq1] | b[q0Aq2][q2Bq1]
[q0Aq2] --> b[q0Aq0][q0Bq2] | b[q0Aq1][q1Bq2] | b[q0Aq2][q2Bq2]
[q0Aq0] --> b[q1Aq0]
[q0Aq1] --> b[q1Aq1]
[q0Aq2] --> b[q1Aq2]
[q0Zq0] --> #[q1Zq0]
[q0Zq1] --> #[q1Zq1]
[q0Zq2] --> #[q1Zq2]
[q1Aq1] --> c
[q1Bq1] --> d
[q1Zq2] --> #
[q1Xq1] --> x
```

Fig. 3. Long CFG equivalent to the PDA example input in "Fig. 1".

Finally, the program proceeds to simplify the CFG by combining productions under the same left-hand-side variable and eliminating useless productions. "Fig. 4" shows the end output.

```
CFG:
S --> [q0Zq2]
[q0Zq2] --> a[q0Aq1][q1Zq2] | #[q1Zq2]
[q0Aq1] --> a[q0Aq1][q1Aq1] | b[q0Aq1][q1Bq1] | b[q1Aq1]
[q1Aq1] --> c
[q1Bq1] --> d
[q1Zq2] --> #
```

Fig. 4. Simplified CFG equivalent to CFG on "Fig. 3".

In the case that all productions get removed (the CFG is empty), then the following message gets displayed:

```
CFG:
The language is empty!
There is no input string that will make the PDA stack go empty.
Therefore, no useful CFG productions.
```

Fig. 5. Simplified CFG equivalent to CFG on "Fig. 3".

VI. Conclusion

The algorithm developed was based on the methods described in "Introduction to Languages and the Theory of Computation" [2]. While this algorithm derives all the possible productions of an equivalent CFG, it also produces an extraneous amount of productions. Therefore, many useless productions generated by the algorithm could be eliminated to obtain a much simpler equivalent CFG. Although this algorithm can take any PDA with acceptance by empty stack configuration, additional restrictions were applied to be implemented in a low-level program. Further development and improvements can be performed on the program to reduce or eliminate such restrictions and provide a more graphical user interface (GUI).

Acknowledgment

We would like to thank Dr. Valentina Korzhova, professor in the Department of Computer Science and Engineering, for providing critical guidance, suggestions throughout this project and explaining in great detail the CFG to the PDA algorithm. Likewise, we would also like to thank the teaching assistants, Antonio Laverghetta and Dmytro Vitel, for highlighting the major critical points of the project. Overall, the expert advice and encouragement received was crucial in the completion of this project.

References

[1] P. Linz, An Introduction to Formal Languages and Automata, 6th ed. Burlington, MA, USA: Jones & Barlett Learning, 2017.

[2] J. C. Martin, Introduction to Languages and the Theory of Computation, 4th ed. New York, NY, USA: McGraw-Hill, 2011. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.465.3774&rep=rep1&type=pdf. Accessed: Apr. 19, 2020.