Phrase Structure Grammar for Non-Prime Production

Chih Chin Chang
Department of Information and Computer Science
University of Hawaii at Manoa
Honolulu, Hawaii
changcc@hawaii.edu

Abstract

A language is a set of strings that can be rigorously defined by set notation, and a grammar has production rules to generate strings belonging to a particular language. Just like languages have a hierarchy of complexity, so too do the grammars that can generate these languages. Phrase structure grammar is a powerful production tool to generate highly complex languages such as context sensitive languages. Furthermore, prime numbers have unique properties that make them non-trivial for the study of languages. To generate a language defined by its association with prime numbers, specifically only composite numbers, requires a phrase structure grammar. Phrase structure grammar, or type-0 grammar, is not unique, and this paper will investigate one possible grammar to generate strings of a particular language.

Index Terms

composite number; type-0 grammar; phrase structure grammar; language

I. Introduction

A language L can be formally defined as a set of strings defined over an alphabet, Σ. An alphabet is a set of terminal symbols, and a language can have any number of concatenations of the powerset of Σ. A language can also be empty, but that is a trivial language and is not particularly interesting in terms of grammar construction.

Generally, a language can fall under a hierarchy of complexity, which is called the Chomsky's Hierarchy, depending on its definition. And different levels of complexity is associated and defined by a particular complex grammar and automaton. The most powerful automaton is the Turing machine. And the most powerful grammar is the type-0 grammar, also known as the phrase structure grammar. The set of languages associated with the Turing machine is the recursively enumerable language. These languages are defined by its ability to be enumerated by a Turing machine.

A. Phrase Structure Grammar

A grammar is a tuple of four components, and some of them are sets. A grammar G = (T, V, P, S) where T is the set of terminal symbols, V is the set of non-terminal symbols, which are often called variables, P is the set of

production rules, and S is the starting symbol. S is also a non-terminal symbol, which means that S is in V.

The phrase structure grammar is particularly powerful because it is also the unrestricted grammar. Formally, production rules of a phrase structure grammar generally have the form $\alpha \rightarrow \beta$ where $\alpha and \beta$ are arbitrary strings of symbols from the sets T and V such that α is not empty. This is different than context-free grammar because in context-free grammar, α can only be a single symbol from the set of non-terminal symbols.

B. Prime and Composite Numbers

This paper will investigate a possible phrase structure grammar for a particular language L, such that L = $\{0^i|$i is not a prime$\}$. The exponentiation is a count for the number of the terminal symbol, 0, in the string and not an arithmatic calculation. And, obviously, i $\in \mathbb{N}$ such that i > 0. The phrase, "i is not a prime" is equivalent to "i must be a composite", which carries some useful properties.

By the definition of composite numbers, an integer is composite when it is divisible by an integer other than 1 and itself. Divisibility is defined such that an integer a is divisible by an integer b if and only if a$\%$b = 0.

This property suggests that a composite number must be the result of a multiplication of two integers greater than 1. The trivial case is when the composite number is even. An even composite number must be divisible by 2 by the definition of even numbers. The other case, when the composite integer is odd, requires that the integer must be divisible by a number greater than 2. The proof is logical. Because composite numbers are divisible by an integer greater than 1 and odd integers are not divisible by 2, logic requires that an odd integer must be divisible by a number greater than 2. This special property is useful when designing the grammar because the language can be broken down into a multiplication of two integers starting at two.

II. A Particular Type 0 grammar for L

Let G = (T, V, P, S). T = $\{0, \epsilon\}$. V = $\{S, A, M, L, R, \$, \#, \%\}$. P = {S$\rightarrow$ $\$AA\#\#MM\%$, A$\rightarrow$AA, M$\rightarrow$MM, A$\# \rightarrow \#L$, $\#M \rightarrow R\#$, $\$\# \rightarrow \$$, LR$\rightarrow$RL0, $\$R \rightarrow \$$, 0R$\rightarrow$R0, $0\% \rightarrow \%0$, L$\% \rightarrow \%$, $\$\% \rightarrow \epsilon$}. To better refer to the individual production rules, they are labeled by a sequence from 1 to 13, incremented by 1.

Intuitively, this grammar creates a left-hand side and a right-hand side. Both sides start with two symbols each and can grow by one by applying rules 2 and 3 repeatedly until a desired number of left-hand side and right-hand side is reached. By looking at the rules, M counts how many multiples of As the grammar will produce.

After a desired number of As and Ms are reached, the $\#$ symbol will convert As to Ls and Ms to Rs so that they can no longer increase. The left $\#$ symbol can only convert the A symbol and travel left. The right $\#$ symbol can only convert the M symbol and travel right. And eventually, the $\#$ symbols will reach the ending symbols and be converted to ϵ. The R symbol crosses every L symbol, creating a terminal symbol 0 for each L. The R symbol can only traverse left and will eventually reach the left end symbol and be converted to ϵ. The right end symbol $\%$ can only traverse left to erase every L symbol that is to the right of every R symbol. Because R symbol can only traverse left, they will never be used twice to copy L to the terminal symbol. Eventually, the $\%$ and the $\$$ symbols will meet and will terminate. These rules make for a basic multiplication function of the right and left operands.

III. Proof L(G) = L

First, the grammar G must be proven to only generate strings belonging to the language L. That is, the grammar G can produce every composite numbers of the terminal symbol 0. Also, the grammar must be proven to only produce composite numbers of the terminal symbol.

To start, composite numbers must be divisible by 2 or an integer greater than 2. Therefore, production rules 2 and 3 can generate twos or any integer greater than 2. The grammar must start with S and S must produce $\$AA\#\#MM\%$. At this point, there are only two options for both left and right hand side of the hashtag symbols: either apply rules 2 and 3 and increment the numbers of A and M symbols or move the hashtag symbols left or right, converting the A to L and M to R respectively. In either case, there are at least 2 A symbols and 2 M symbols, and they must be converted to at least 2 L symbols and at least 2 R symbols.

When that step is completed, the $\#$ symbols will reach the $\$$ and $\%$ symbols, which can use rules 6 and 7 to elimate the $\#$ symbols. At the same time, rule 8 can be applied for every consecutive LR symbol to produce RL0, which would have the R symbol cross the L symbol to produce a single 0 symbol. The R symbol can also traverse to the left over the 0 symbol by rule 10. This means that the R symbol will always traverse to the left until meeting the $\$$ symbol, where production rule 9 would eliminate the R. It is easy to see, then, that the R symbol simply makes L copies of the terminal symbol, hence the R is the number of times to multiply the L symbol, producing the resulting terminal symbol. The $\%$ symbol cannot traverse left until every R has traversed left. This is the symbol to end the generation because there is no more R symbols left. The $\%$ symbol traverses both L and the terminal symbol towards the left until it meets the $\$$ symbol, using rules 11 and 12. Rule 12 also eliminates every L when it is met by $\%$ symbol. When the two symbols meet, rule 13 eliminates both symbols. This terminates the production, effectively having multiplied two numbers together.

Because this grammar produces the multiplication of 2 integers where each integer must be greater than 1, it can generate every composite number of the symbol 0. And because the rules can only allow for multiplication of two integers, the grammar can only produce composite numbers and never prime numbers. Since a positive integer must either be prime or composite, L(G) = L.

IV. Conclusion

A particular phrase structure grammar has been investigated to produce a language defined by L = $\{0^i | \text{i is not a prime}\}$. Phrase structure grammar is a powerful tool to rigorously define rules to generate the strings of a language. This grammar is the most powerful tool to generate complex languages. By defining composite numbers and finding its special property, the type-0 grammar can be designed to generate only strings containing that property. Although the grammar may have many production rules, the intuition is simple, and the particular phrase structure grammar discovered can easily be modified to be a general multiplication function by starting with single A and M symbols. The proof for L(G) = L must show that the grammar produces every string in L and that the grammar can only produce strings in L. And because the grammar strictly only produce strings that follow the property of a composite number, this particular grammar can be used to easily show that L(G) = L.

REFERENCES

[1] W. S. Brainard and L. H. Landweber, *Theory of Computation*, John Wiley & Sons, 1974.

[2] J. Carroll and D. Long, *Theory of Finite Automata with an Introduction to Formal Languages*, Englewood Cliffs, New Jersey: Prentice-Hall, 1989.

[3] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation* Addison-Wesley, 1979.

[4] K. H. Rosen, *Discrete Mathematics and Its Applications*, 6th ed. New York, United States: McGraw-Hill, 2007.