Introduction to Functional Grammars

Abhinav Aggarwal

Abstract— Formal grammars are extensively used in Computer Science and related fields to study the rules which govern production of a language. The use of these grammars can be extended beyond mere language production. One possibility is to view these grammars as logical machines, similar to automata, which can be modified to compute or help in computation, while also performing the basic task of language production. The difference between such a modified grammar and an automaton will then lie in the semantics of computation performed. It is even possible for such a grammar to appear non-functional (when no language is produced as a result of its productions), but in reality, it might be carrying out important tasks. Such grammars have been named Functional Grammars (including a special sub-category, called Virtual Grammars), and their properties are studied in the paper.

Keywords - Grammar, Terminals, Productions, Automata, Functionality, Language

V. INTRODUCTION

Let us start by taking an example. Consider the grammar, G_{[S]}, given below:

\[ S \rightarrow aAS \mid bBS \mid T \]
\[ Aa \rightarrow aA \]
\[ Ba \rightarrow aB \]
\[ Ab \rightarrow bA \]
\[ Bb \rightarrow bB \]
\[ BT \rightarrow Tb \]
\[ AT \rightarrow Ta \]
\[ T \rightarrow \varepsilon \]

This grammar generates the context-sensitive language,

\[ C = \{ x \mid x \in \{a,b\}^* \} \]

If we carefully observe the sentential forms which arise while deriving any string in C from S, we would notice that the grammar allows us flexibility only in the first ‘x’, keeping the generation of the latter one to itself. For example, to derive the string ‘aabbaa’, we only need to go up to the production of ‘aah’ in front. The non-terminals in the grammar are “designed” to automatically generate the latter half. This will be clear from a possible derivation given below:

\[ S \Rightarrow aAS \Rightarrow aAaAS \Rightarrow aAaAbBS \Rightarrow aAaAbBT \]
\[ \Rightarrow aAaAbTb \Rightarrow aAabATb \Rightarrow aAabTab \Rightarrow aaAbTab \Rightarrow aabATab \Rightarrow aabTaab \Rightarrow aabaab \]

Looking at the derivation, it is clear that S performs the job of “injecting” terminal symbols corresponding to the first half of the string to be produced (maintaining the relative order of these symbols), after which it is transformed to T, which then takes over the job of producing the exactly similar latter half. Effectively, if one knows the mid-point of such a string, the only requirement is to find the sentential forms leading to the first half of the string, and then let the grammar do its job. It is a trivial, but an excellent example of how grammars can be made to compute as well as produce part or whole of a language.

Let us now look at another interesting grammar, G_{[S]}:

\[ S \rightarrow PQ\varepsilon \]
\[ P \rightarrow Pa \]
\[ aQ \rightarrow Q \]
\[ PQ \rightarrow \varepsilon \]

The most trivial fact about this grammar is that the only string this grammar can produce is the empty string, \( \varepsilon \). Let this triviality not demotivate us to at least analyze the grammar once. It is true that all the non-terminals, including the start symbol S, in this grammar are nullable (they eventually are nullified by producing \( \varepsilon \)). But, the story doesn’t end here. Even this empty string has infinite derivations in this grammar, rendering infinite ambiguity to the process. Consider the following:

\[ S \Rightarrow PQ \Rightarrow PaQ \Rightarrow PaQ \Rightarrow \]
\[ PaaQ \Rightarrow PaaQ \Rightarrow PaQ \Rightarrow P \Rightarrow \varepsilon \]

As we can see, the grammar allows addition of any number of ‘a’s through P, and deletion of any number of ‘a’s through Q. This process can go on as far as all ‘a’s are finally deleted by Q. So, the sentential forms can be described by a regular expression Pa*Q. Two points are important at this stage: one is the nature of these regular expressions and the second is the behavior of this grammar between the productions of \( \varepsilon \) from S.

For the first, we notice that even though the grammar is not a regular grammar (not even context-free), these sentential forms can be written as a regular expression. The important point here is that one can add arbitrary number of ‘a’s in these forms and may even choose to never stop. This endless process will never take the sentential forms to \( \varepsilon \), even when the corresponding automaton is in non-stop motion. Try linking this to the halting criteria for this automaton. Thus, once set in motion, this grammar is capable of entering into an infinite computation despite
being absolutely deterministic in the language it produces. We know that the final result, if the automaton halts, will be $\in$. I hope the triviality of this grammar is now forgotten. I will rest this issue here for the moment and discuss it later in this paper. Nonetheless, it is a point worth giving a thought.

For the second issue, let us assume that we have some "special power" to halt the machine after a finite, however small or large, number of steps. In cases of non-determinism, one seeks for such possibilities, and they indeed, exist. In such a case, it can be noticed that the addition of new ‘a’s happens only through P (or to be precise, to the left side of the sentential forms), while the deletion is near the right end, through Q. Try to visualize some similarity with a queue that adds elements from one end, and deletes from the other, to maintain the FIFO property. This property is, undoubtedly, followed here too. The ‘a’ which arrives first, leaves first too. So, in a way, every sentential form is analogous to implementing a queue, of which the contents can apparently, be only ‘a’s. Even if we ignore the halting criteria for this automaton, it still acts like an infinite-buffer queue. Addition and deletion of ‘a’s is always possible. Another special property of this queue is that after all the operations have been done (even infinite, theoretically at least), the queue eventually vanishes ($PQ$ becomes $\in$). This interpretation might be difficult to digest in the first go, but as I mentioned earlier in the abstract, this paper is based on a semantic analysis of grammars. So, coming to the point, I think it will not be wrong to call such a "system" a virtual queue grammar. If needed, it will do the job we want and vanish thereby, or cease to exist in the first go itself ($S$ also goes to $\in$ in the productions). I call it virtual to highlight the hidden property of storing ‘a’s. This interpretation is the base of this paper, and all further observations and results are related to it.

VI. NULL GRAMMARS vs. PURELY FUNCTIONAL GRAMMARS

We saw above an example of a grammar, which imitates the functioning of a queue and eventually, reaches $\in$. The obvious question arises now. Can all grammars be given such an interpretation? If yes, how? For this section, let us focus our attention to a ‘subset’ (in pure semantics) of this question. Can all grammars, whose language is the empty string $\in$, be given an interpretation similar to the one given above? The answer is exactly what is expected; NO. Consider the grammar, $G_2[S]$ below:

$$S \rightarrow A$$
$$A \rightarrow \epsilon$$

Introducing more and more (rather useless) non-terminals is not exactly an interesting job, as far as this paper is concerned. So, we have a class of grammars, called the Null grammars. An interesting subclass is of the grammars like $G_2$. Even though the grammar produces $\in$ only, we do have ‘a’ as a terminal symbol in the grammar definition, for if it did not exist, then the productions won’t be valid. We can always consider ‘a’ as a non-terminal too, but as I said earlier, since we are not interested in introducing non-terminals to study grammars, let us not consider that domain. Also, had ‘a’ been a non-terminal, we would have to give it a production rule to reach some terminal, for otherwise, it would be a useless non-terminal in the very beginning. Thus, this grammar ‘performs some function’ besides producing $\in$. Hence, we call such grammars Purely Functional Grammars. The word “purely” suffices to the ultimate production of $\in$ from $S$. Note that these grammars are, in principle, similar to Null grammars, which is why they form a subclass of the latter; still, this aspect of functionality or rather virtual functionality imparts it great importance.

Following are three more examples of purely functional grammars.

1. Virtual infinite capacity stack grammar
   $$S \rightarrow PQ\epsilon$$
   $$P \rightarrow Pa$$
   $$Pa \rightarrow P$$
   $$PQ \rightarrow \epsilon$$

2. Virtual infinite capacity double sided queue grammar
   $$S \rightarrow PQ\epsilon$$
   $$P \rightarrow Pa$$
   $$Q \rightarrow aQ$$
   $$Pa \rightarrow P$$
   $$aQ \rightarrow Q$$
   $$PQ \rightarrow \epsilon$$

3. Virtual infinite capacity double sided priority queue grammar (‘b’ has higher priority than ’a’)
In all the examples shown above, the language of each of the grammars is the empty string \( \varepsilon \). But the sentential forms exhibit patterns which resemble the functioning of stacks, queues, etc. Hence, these are purely functional. The word “virtual” can now be aptly added while naming these grammars. The utility and analysis of this class of grammars follows later in this paper.

VII. FUNCTIONAL GRAMMARS

So far, we saw the concept of purely functional grammars. But, how can we be mischievous enough to neglect the grammars that are commonly used? They also perform computations, although the way we see them is different. It is true that all grammars carry out some planned business of its non-terminal symbols to produce the strings in its language. It is exactly this business that the paper focuses on. Very often, the construction of an acceptor automaton for a language is independent of the grammar underlying the language. In other words, some automata are easier to create knowing the kind of language produced, than the grammar to be realized. As a simple example, if we wish to construct a Turing machine to accept the language 

\[
\{ | \}
\]

It is much easier to construct it by exploiting the symmetry in the language. Writing down the grammar is not a requirement at all, because once this machine has been constructed, all the computations on these strings can be performed by using it. But does that imply no utility of writing grammars? That’s surely not the case. This is exactly where functional grammars come into picture. By carefully observing the role of this Turing machine, we are actually accepting a string of the above form, rather than create one. If a string, not of this form, lies on the input tape of the machine, the acceptor will not accept the given input. But, if one constructs a Turing machine, which uses the grammar for such strings to generate strings in the language, then this machine can guarantee to produce acceptable strings. This is slightly different from parsing, as the latter looks for a derivation of a particular string, while we are talking of producing arbitrary strings belonging to \( L \). Grammar now seems to play its card.

Let us take another example. Consider the following two grammars.

\[
\begin{align*}
G_4[S]: & \quad S \rightarrow PQ \\
& \quad P \rightarrow PA | PB \\
& \quad Q \rightarrow AQ | BQ \\
& \quad PA \rightarrow P \\
& \quad PB \rightarrow P \\
& \quad AQ \rightarrow Q \\
& \quad BQ \rightarrow B \\
& \quad AB \rightarrow BA \\
& \quad PQ \rightarrow \varepsilon \\
& \quad A \rightarrow a \\
& \quad B \rightarrow b
\end{align*}
\]

\[
\begin{align*}
G_5[S]: & \quad S \rightarrow PNQR \\
& \quad N \rightarrow Na | Nb \varepsilon \\
& \quad aQ \rightarrow QA \\
& \quad bQ \rightarrow QB \\
& \quad PQ \rightarrow P \\
& \quad PB \rightarrow bP \\
& \quad S \rightarrow bB \mid aC \mid a \mid b \varepsilon \\
& \quad B \rightarrow bB \mid aC \mid a \mid b \\
& \quad AB \rightarrow BA \\
& \quad C \rightarrow aC \mid a \\
& \quad AR \rightarrow Ra \\
& \quad PR \rightarrow \varepsilon
\end{align*}
\]

The language of both the grammars is \( b^*a^* \). Then, how do they differ? Or do they differ at all? My answer is yes, they do. Perhaps one might say that while \( G_4 \) is strictly a regular grammar, \( G_5 \) is context-sensitive (Not important). For us, both are context-sensitive as far as the classification-based difference is concerned. To see a possible difference, let us focus on the derivation of a string, say ‘bbaaa’ for the two grammars.

For \( G_4 \), we have,

\[
\begin{align*}
S & \Rightarrow bB \\
& \Rightarrow bbB \\
& \Rightarrow bbaC \\
& \Rightarrow bbaaC \\
& \Rightarrow bbaaa
\end{align*}
\]

For \( G_5 \), we have (one possible derivation),

\[
\begin{align*}
S & \Rightarrow bB \mid aC \mid a \mid b \varepsilon \\
& \Rightarrow bB \mid aC \mid a \mid b \\
& \Rightarrow AB \rightarrow BA
\end{align*}
\]

The derivations above show some remarkable features. The first derivation is a pretty simple and straightforward derivation, injecting ‘a’s and ‘b’s in exactly the right order and eventually halting by nullifying the non-terminal C. If however, we try to inject ‘b’s before ‘a’s, there is no way that this derivation, or the grammar, can fix this to produce ‘bbaaa’. Thus, the order of introduction of terminals is important. Easy and comfortable.

Let’s now look at the second derivation. Although the grammar for this is ambiguous, I choose this particular derivation for a reason. If you notice, the non-terminal responsible for injecting terminals in the sentential forms is N. No other non-terminal can do that. For the given derivation, the order in which these terminals are introduced is ‘aabba’, which is quite different from the string to be eventually produced. Nonetheless, we get ‘bbaaa’ at the end.
It is also worth noting that no matter what order we choose, as long as we inject two ‘b’s and three ‘a’s with this N, in any order, we will land up with ‘bbaaa’ as our final product of the derivation. Here lies the strength of this grammar over the previous one. The freedom of introducing terminals at will, and the ability to compute the correct permutation for any such order provided, is a strong functional attribute of this grammar. One might question the need of this grammar, if the sole purpose is to produce \(b^*a^*\). My answer is the same as it always was, we concern ourselves with more than just the language production. Another important feature of any derivation though \(G_1\) is that while new terminals are injected by \(N\), the already existing terminals have been continuously arranged in the right order by the grammar productions (replacing ‘b’ by B though, but then that’s just not important as far as semantics are considered). If \(N\) is assumed to be analogous to an external input source, which can expand into a production at will, the time gap between two consecutive expansions of \(N\) into its productions, is utilized by the grammar to arrange the already existing symbols in the right order (‘b’s followed by ‘a’s). Hence, this grammar also optimizes its derivation at each step. This optimization is with respect to the flexibility discussed before. Hence, a Turing machine simulating this grammar, is capable of reading a string of ‘a’s and ‘b’s from one input tape and producing the sorted output (‘b’s followed by ‘a’s) on another tape, with the first input tape acting as a an analogy to \(N\). The grammar eventually helps us sort, apart from just produce something. This is exactly the functional aspect we are concerned with. Since, this grammar does not eventually produce the empty string, we call it a Functional Grammar, rather than a ‘purely’ functional or ‘virtual’ grammar. This does not mean that it isn’t a context-sensitive grammar anymore. It surely is, but it is also a Functional grammar. The above discussion is similar to Programming with grammars.

Now that we have an idea of what functional grammars are, let us look into some properties of these grammars.

1. We start by stating the position of such grammars in the hierarchy provided by Noam Chomsky. It is clear from the above examples that if not higher, these grammars are at least context-sensitive, which inherently includes context-free and regular grammars. Since even the higher grammars (in the classification) can exhibit functionality, I can safely say that for every such grammar, a Turing machine can be built, which may be deterministic or non-deterministic.

2. It is also apparent that what we are concerned with are the aspects of grammars other than just the language production. Thus, we can assign various attributes to the study of these grammars, one of which is the language produced by them. Others can include functionalities similar to examples taken above. Another possibility is considered with grammars whose corresponding automata use external data structures, like stacks. Consider, for example, a simple Pushdown Automata, which uses only one stack with a single input tape. We are already aware of two types of languages that can be considered to be accepted by this machine, based on the definition of “acceptance”. One is the set of all strings, the acceptance of each of which is defined by the existence of machine in a pre-defined accepting state at the end of input. The second one is the set of all strings which are considered accepted if their exhaustion also exhausts the stack contents, irrespective of what state the machine is in. Let us now consider the third variety. Let \(L\) be the set of all possible configurations of the stack, while traversing any string on the input tape. Since the stack contents are grammar terminals, this set \(L\) contains strings made up of these terminals only. Hence, \(L\) is also a variant of the language that can be associated with this grammar. But, since \(L\) will generally not contains strings that the grammar was originally intended to produce, it represents a secondary function performed while the automaton simulates that grammar. For example, Consider the following grammar (for balanced parenthesis):

\[
S \rightarrow (S) | ( ) | S( )
\]

Here, ‘(’ and ‘)’ form the terminals of the grammar. If we simulate this grammar using a PDA, by inserting in the stack every ‘(‘ we encounter and popping one out as we get ‘)’, what we observe is that the language corresponding to pre-defined accepting state based acceptance is the same as that corresponding to stack exhaustion. But the set \(L\), as described above, for this PDA will be \(*\) (the Kleene closure of ‘(‘). The stack can never contain ‘)’, and allows any number of occurrences of ‘(‘, including zero. This set \(L\) is very different from the two languages we would normally study. It is a strictly regular language, obtained as a byproduct of this simulation. We can view this as a kind of secondary function performed by the grammar. Such functions derive from the creative use of grammars. Hence, not underestimating the power of human mind, as well as these grammars, an important property of functional grammars is that they are creative as hell.

3. We can see that every grammar, which is not a Null Grammar (Purely Null, to be precise. Don’t include purely functional grammars in this set, for now), from which all useless non-terminals and productions have been removed (the criterion for useless does not involve nullification of non-terminals) essentially contains a combination of the following types of non-terminals:
   i) Producer Non-Terminals (PNT)
   ii) Consumer Non-Terminals (CNT)
   iii) Modifier Non-Terminals (MNT)

The interplay between these types (and their combinations; some non-terminals can have more than one quality at a time) of non-terminals opens an important door while studying the functional attributes of a grammar. But, one has to be extremely careful while using these terms. The details of each are subsequently provided.
IV. PRODUCER NON TERMINALS

As the name suggests, Producer non-terminals are those non-terminals which are capable of injecting new terminals in the sentential forms. These terminals are new in the sense that they were not present in all sentential forms preceding their introduction. However, two points are worth noting here. One is that the newly injected terminals may not appear in the final string obtained. We are only concerned with their injection in the sentential forms. Second important point to note is that the “newness” has nothing to do with the face-value of the terminals. That is, an ‘a’ introduced in the existence of previously injected ‘a’s is still “new” as far as this concept is concerned. Thus, we can define a non-terminal, V, to be a PNT if there is a derivation,  

*  

\[ V \Rightarrow \alpha, \]  

where \( \alpha \) contains at least one terminal. For example, the following production guarantees N, a non-terminal, to be a PNT:  

\[ N \rightarrow aN \mid Na \varepsilon \]  

Note that a non-terminal may not directly inject a terminal in a single step, but may do it over a sequence of steps. Hence, we use ‘derivation’ instead of ‘production’ in the definition. An example of such a case is:  

\[ B \rightarrow NM \]  

\[ N \rightarrow aN \mid Na \varepsilon \]  

Here, B is also a PNT, as it eventually is capable of injecting terminals into the strings being derived from the start symbol of the grammar. N, undoubtedly, is again, a producer non-terminal too.

Producer non-terminals are of two broad types.

- Context-free Producer Non-Terminals
- Context-Sensitive Producer Non-Terminals

Context-free PNTs are the ones defined above. Their capability to inject terminals is independent of what precedes or follows them in the sentential forms. Context-sensitive PNTs are, however, more clever. Their injection ability is dependent on their “surroundings” or “context”. For example, consider the following productions:  

\[ aA \rightarrow bbA \]  

\[ bA \rightarrow A \]  

As we can see, A, when preceded by ‘a’, produces two ‘b’ s in the production, but when the same A is preceded by a ‘b’, it eats up that ‘b’ and produces no new terminal. Hence, A, in this case, is a context-sensitive PNT. For such cases, we can replace them by suitable non-terminals (may have to introduce new ones) to make them context-free PNTs, as they are easier to deal with. For the above example, this can be done as:  

\[ aA \rightarrow C \]  

\[ C \rightarrow bbA \]  

\[ bA \rightarrow A \]  

In this case, A is no longer a PNT, but now, C becomes a context-free PNT.

We still haven’t seen the reason to study Producer non-terminals. Consider a grammar having no PNTs (context-free or context-sensitive). Can this grammar be anything other than a Pure Null Grammar? We assume here that the utility of terminals can be exploited over non-terminals, in the fact that if we replace ‘a’ in our virtual queue grammar by the non-terminal A, and argue that the function performed is same, doing this is merely equivalent to writing \( G_1 \). The only fact that distinguishes ‘a’ from ‘A’ is that even though ‘a’ is a terminal, it never appears in the language generated by the grammar. Thus, for any grammar to be at least a functional grammar, if nothing else, it ought to have at least one PNT. A grammar void of PNTs is a Pure Null Grammar. However, as we will see in the next section, this is not the complete requirement for a grammar to be a Pure Null Grammar.

V. CONSUMER NON TERMINALS

Similar to PNTs, Consumer non-terminals are those non-terminals which “eat-up” terminals from sentential forms and reduce the number of terminals from subsequent derivations. For such a non-terminal, every sentential form following the one in which it exists, contains lesser number of terminals than the previous sentential forms, unless some PNT inserts a new terminal for the first time. Thus, a non-terminal, V, is a CNT if there is a production of the type  

\[ \alpha V \beta \rightarrow \phi \]  

, where \( \alpha \) and \( \beta \) are strings of grammar symbols (may be empty also, but not simultaneously) with at least one terminal adjacent to V, and \( \phi \) is a string of grammar symbols (may be empty) in which the number of terminals is less than that in \( \alpha \beta \). Needless to say, CNTs are always context-sensitive. For example,  

\[ aA \rightarrow A \]  

Here, A is a CNT, as it eats up any ‘a’ that dares to precede it.

An interesting property that connects PNTs with CNTs is the analogy with a communication system, modeled as a simple producer-consumer problem. We have a producer who injects items it produces into a buffer, from which a consumer takes up items at its will. If the buffer size is greater than the maximum consumption possible, all items produced after this limit has been reached remain in the buffer forever. If, however, the buffer size is the same as maximum consumption as well as the maximum production, it eventually becomes empty when the consumer consumes all that the producer has to offer. Otherwise, the consumer starves. Using this line of thought, consider all PNTs to be producers (similar to Siphons in a Petri net) and all CNTs to be consumers (similar to Traps in a Petri net). The sentential forms are analogous to the buffer that connects these two. If the PNTs produce more terminals than the CNTs can consume, the language of the grammar will be strings of finite length. Each of these strings will contain exactly
those terminals, in exactly that order, which could not be consumed by any of the CNTs, due to some reason. We say that a grammar has a language, which does not contain the empty string, if the PNTs are apparently more “powerful” than the CNTs. This “power” is measured in terms of Production Index of the PNTs, Consumption Index of the CNTs, and the Transient Index of the grammar.

If however, the terminals produced by PNTs are all consumed by CNTs at some stage or the other, this ultimate exhaustion of all terminals lead to a grammar whose language is the empty string. This grammar is a Purely Functional Grammar. For all such cases, the Production Index of the PNTs is the same as Consumption Index of the CNTs and the Transient Index of the grammar is zero.

In the worst case, if CNTs form a more powerful group, then we have a Pure Null Grammar, as consumers will starve after the production has exhausted. These non-terminals will then become useless, or nullify themselves.

VI. MODIFIER NON TERMINALS

Non-terminals, which are neither PNTs nor CNTs are called MNTs. Their job is to carry out the various computations in a grammar, like permuting the substrings of sentential forms, deletion/addition of non-terminals etc. They impart the real utility to grammars. Without them, a grammar will be nothing better than a blind-give-and-take construct. In simpler words, even PNTs and CNTs are essentially modifying the sentential forms, but we talk of more precise definitions here. MNTs control the Transient Index of a grammar, and impart a “lexical outlook” to the strings in the language. It is the job of an MNT to see the order of various terminals in the final strings produced. For purely functional grammars, the secondary functions of the grammar are completely defined by these MNTs. Had there been no MNTs in a grammar whose language is the empty string, this paper would have no meaning at all. The essence of Pure Functionality is the study of these MNTs of the grammar.

VII. SOME POSSIBLE APPLICATIONS OF FUNCTIONAL GRAMMARS

Functional Grammars are just normal grammars put to more use than mere language-production. As demonstrated before, the virtual stack and queue grammars can be used to enhance the computing power of a Turing machine, by introducing them in their control unit, as required. Since they produce nothing better than an empty string as their final language, the input tape remains unaffected. No terminal from the purely functional grammars, enter the input tape, unattended. This will greatly help in designing better and more sophisticated automata.

Functional grammars can also be viewed as real functions. Purely functional grammars are functions with a “void” return type, as they produce no effective language. Others can be functions with a return type, the output being the language of the grammar. In either of the two cases, what I seek to emphasize is the body of the function, and the sole fact of perceiving these grammars as being analogous to functions, or in other words, have a computing ability.

REFERENCES

[1] J. E. Hopcroft, R. Motwani, J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Pearson Education, 2001.
[2] S. P. Eugene Xavier, *Theory of Automata, Formal Languages and Computation*, New Age International Publishers, 2008.

Abhinav Aggarwal is an undergraduate student at Indian Institute of Technology (IIT) Roorkee in Computer Science and Engineering, with specialisation in Information Technology, to Graduate with a B. Tech. and M. Tech. Dual Degree in 2014.