Grammatical Methods in Computer Vision : An Overview

Gaurav Chanda and Frank Dellaert
College of Computing
Georgia Institute of Technology
Atlanta, GA
{gchanda, dellaert}@cc.gatech.edu

Technical Report
Nov 29, 2004

Abstract

We review various methods and applications that have used grammars for solving inference problems in computer vision and pattern recognition. Grammars have been useful because they are intuitively simple to understand, and have very elegant representations. Their ability to model semantic interpretations of patterns, both spatial and temporal, have made them extremely popular in the research community. In this paper, we attempt to give an overview of what syntactic methods exist in the literature, and how they have been used as tools for pattern modeling and recognition. We also describe several practical applications, which have used them with great success.

1 Introduction

Grammars have been useful for providing structural descriptions of a variety of objects like plants, buildings, English sentences etc., and associating semantic information with the resulting models. They have been most widely used by the natural language processing community. However, they have also been applied in many pattern recognition and computer vision problems. Some of the areas include character recognition [K.H. Lee and Kashyap, 1988], recognition of handwritten mathematical formulae [Miller and Viola, 1998] and more recently, activity recognition [Minnen et al., 2003].

Grammars consist of a collection of basic primitives and a set of rules which compose patterns out of these primitives. Thus collections of objects which exhibit structural regularity can be modeled using grammars. For example, the set $\{00, 0110, 01110, 011110...\}$ describes the set of sequences over $\{0, 1\}$ which have a sequence of 1s bounded by two 0s. Much more complicated sets, which exhibit regularity in higher dimensions including time, can be modeled using grammars. Roughly speaking, they generate sub-patterns out of basic primitives, and then generate bigger or more complicated patterns from these sub-patterns. This generation is controlled by the grammar rules.

There are several issues in the modeling of objects or events using grammars. The first issue is the choice of basic primitives and the appropriate grammar type. The choice of primitives is determined by how easily basic features can be extracted from the data, while the choice of grammar type is determined

by the complexity of the desired structure. One might want to consider the scenario, wherein extraction of basic primitives and choice of the grammar, be completely independent. However, Narasimhan in Chap 1 of [Kaneff, 1969] argues that, this is not possible. According to him, the inherent complexity of the problem is divided among the basic primitives and the actual grammar and thus the choice cannot be made independently. Related to this point is the *syntax-semantic* tradeoff, which states that one might choose to keep a very simple grammar and imbibe a lot of semantic interpretations in it, or otherwise choose to keep a highly complicated grammar and have simpler and more tractable interpretations. This tradeoff is a crucial decision that researchers have to make before thinking of a solution.

The second important issue is noise handling. Inputs in computer vision applications are very noisy, and formal grammars are too exact theoretical concepts to model these perturbations in data. Thus, people have come up with grammatical models and algorithms which are robust against noise. These two issues are important in defining grammars and using them for performing inference tasks.

In this paper, we survey the various forms of grammars that have been developed and used by the research community. We also attempt to provide scenarios where one form may prove more appropriate than another form, depending on the characteristics of the problem under consideration. The paper is divided into 5 sections. In section 2, we discuss *string grammars*, their different extensions and some applications as to how they have been useful in describing *line drawings* and *activity recognition*. Probabilistic information could be associated with grammars and so we explore *stochastic grammars* in section 3. We then move on to *higher dimensional grammars* and discuss various forms which have been developed over the past 30 years in section 4. We finally conclude the paper in section 5. Most of the mathematical descriptions have been taken from [Bunke and Sanfeliu, 1990].

2 String Grammars

String grammars form the basis of formal language theory. These grammars are defined over a *vocabulary* (finite set of symbols). Each grammar represents a set of (finite) sequences over this *vocabulary*. For example, the sequence $\{01, 0011, 000111, 00001111, ...\}$, comprises of all strings over the *vocabulary* $\{0, 1\}$ which are formed by concatenating a sequence of 0s with a sequence of 1s, such that the length of the two sequences are same. These sequences are generated using a collection of *production rules*. These rules define a set of substring replacement policies which gives the final string certain properties, like the one described in the example above.

The set of symbols belonging to the grammar vocabulary correspond to the basic primitives, while the production rules correspond to the *structural relationships* that exist in the set defined by the grammar. Since a sequence is a one-dimensional concept, the above schema is not very suitable for describing 2D or 3D objects. However, certain extensions of these grammars have led to their use in computer vision and pattern recognition applications. In this section, we first provide some formal definitions and examples of string grammars. We then review the extensions of these grammars and how they have been useful in practical applications.

2.1 Formal definitions

The following definition is taken from [Bunke and Sanfeliu, 1990]. A *formal grammar* is a four-tuple $G = (N, T, P, S)$ where

N is a finite set of *non-terminal symbols,*

T is a finite set of *terminal symbols,*

P is a finite set of *productions,* or *rewriting rules*, and

$S \in N$ is the *initial* or *starting symbol.*

It is required that $N \cap T = \phi$. Let us suppose, $V = N \cup T$. Each production $p \in P$ is of the form $\alpha \rightarrow \beta$ where α and β are called the *left-hand* and *right-hand side* respectively, with $\alpha \in V^+$ (set of finite, non-zero length sequences over V) and $\beta \in V^*$ (set of finite length sequences over V).

To illustrate this definition, consider the grammar $G = (N, T, P, S)$ where $N = \{S, A, B\}$, $T = \{a, b, c\}$, $P = \{S \rightarrow cAb, A \rightarrow aBa, B \rightarrow aBa, B \rightarrow cb\}$.

This grammar generates the language

$$L(G) = \{ca^n cba^n b | n \geq 1\}$$

For example, for generating *caacbaab*, i.e., for $n = 2$, the following sequence of substring replacements are applied.

$$S \rightarrow cAb \rightarrow caBab \rightarrow caaBaab \rightarrow caacbaab$$

2.1.1 *The Chomsky Hierarchy*

Grammars have been divided into four classes or types, based on their degree of expressiveness. This classification has been taken from [Aho et al., 1988]. The classes are as follows :

1. *Unrestricted class* or *type 0* have no constraints other than the definition of grammar described above.
2. *Context-sensitive* or *type 1* have productions of the form $xAy \rightarrow xzy$ where $x, y \in V^*$; $A \in N$; $z \in V^*$. Thus, in a *context-sensitive grammar*, the non-terminal A can be replaced by a string z only if it appears in the context xAy.
3. *Context-free* or *type 2* have productions of the form $A \rightarrow z$, where $A \in N$ and $z \in V^*$. Thus, in a *context-free grammar* no context is necessary for the application of a rule.
4. *Regular* or *type 3* have productions of the form $A \rightarrow aB$, or $A \rightarrow a$ where $A, B \in N$; $a \in T$. These grammars are simpler and can be simulated by finite state automata.

2.2 Attributed Grammars

Grammars are useful to represent structural information of patterns. However, in practice, we need to derive semantic interpretations for those structures. Consider for example, the language $L = \{(0+1)^*\}$ which denotes the set of all strings over 0 and 1. However, one natural semantics that one might want to associate with this set could be the set of whole numbers. For associating appropriate semantics, one needs to associate semantic information with the primitive symbols, non-terminals and the production rules. The semantics associated with symbols are also known as *attributes*, and those associated with productions are *semantic rules*. These *semantic rules* relate the *attribute values* of different symbols involved in a production. Thus, the "meaning" of any string can be obtained by first deriving the string and then extracting the *attribute values* of the start symbol S. Attribute grammars were introduced by Donald Knuth in 1968 [Knuth, 1968].

Consider the grammar $G_{Bin} = (N, T, P, S)$ where $N = \{S\}$, $T = \{0, 1\}$, $P = \{S \rightarrow S0,\ S \rightarrow S1,\ S \rightarrow 0,\ S \rightarrow 1\}$. G_{Bin} generates all sequences of 0s and 1s. Now, if we want to associate the language of G_{Bin} with the set of whole numbers, then the semantics that we choose for *terminals* and *non-terminals* would be their actual value (*attr==value)*. Thus, the symbol 0 will be mapped to the "whole number" *zero*, and the symbol 1 will be mapped to the "whole number" *one*. For further discussions, we will call the actual whole numbers by their English names. Also, for a production $S \rightarrow S0$, the first non-terminal will be referred to as $S[0]$ while the second non-terminal will be referred to as $S[1]$, so that we know which attribute value we are referring to. Thus, we have

$$0.value = zero$$

$$1.value = one$$

We now need to associate semantics to the production rules. The *semantic rules* relate the attribute values of the left-hand side and the right-hand side of a production rule. It is through these rules that information or meaning can propagate from the *start symbol* to the *terminals* and vice versa. For instance, the semantic rule for the first and third productions would be

$$S[0].value = two * S[1].value + zero$$

$$S.value = zero$$

the array index is present to refer to the appropriate non-terminal (S of left-hand side or right-hand side) in the production. Now while deriving the string 10, we have the sequence

$$S \rightarrow S0 \rightarrow 10$$

and the corresponding semantics would be

$$S[0].value = two * S[1].value + zero = two * one + zero = two$$

which was what we expected.

2.3 Parsing

Grammars are used in computer vision and pattern recognition for performing *inference tasks*, i.e., finding out whether the given data satisfies certain structural characteristics. Given the description of a grammar and an input string, determining whether the string belongs to the *language* generated by the grammar is the key question in this *inference* task. This task is performed by a *parser*. A *parser* takes as input a sequence of symbols, and returns a *parse tree* which contains the details of the derivation of that string. If the string is not part of the language, then no such tree can be generated and the parser can detect this. Designing efficient and robust parsers are very important for performing pattern recognition tasks using grammars.

As the degree of expressiveness of the grammar increases, design of the corresponding parser becomes more and more difficult. Most of the parsing methods assume a noise-free stream of input symbols. However, error-correcting parsing methods have also been devised. Much research has also been done in parsing of *higher dimensional structures* like graph grammars and tree grammars [Fu and Shi, 1983]. More detailed description of different parsers and issues in parsing could be found at [Aho et al., 1988] and [Fu, 1974].

2.4 Grammar Learning

A grammar representing a pattern class could be learned from a set of examples from that class. This is the typical *machine learning* problem, where from a huge set of positive and negative examples, one attempts to learn the syntactic structure inherent in this class. Inference of grammars has been a very difficult problem, when applied to practical situations. Since the number of examples is always finite, *generalization* is an important issue in these learning algorithms (otherwise overfitting will lead to inaccurate models). Most of the applications propose grammatical models rather than "learning" those models and then apply them to the test examples.

Much work has been done in developing different inference techniques for various forms of grammar. For surveys of the different methods, see [Fu, 1974], [Angluin and Smith, 1983]. Some of the projects which have used these inference techniques includes fingerprint classification using *tree grammars* [Moayer and Fu, 1975] (discussions on higher dimensional grammars like *tree grammars* will be done in Section 4). In this work, the "interesting" components of the picture were identified which had a lot of repetitive structures. Grammars were inferred for each such component and the different grammars were then merged.

A similar (top-down) approach was used in the work on texture synthesis by [Lu and Fu, 1979], where texture in images were modeled using *stochastic tree grammars*. The problem was broken into simpler problems by decomposing the tree structure into a number of strings. Each such string class corresponded to a *finite state grammar*, and was learnt independently. The resultant grammars were then recomposed to output one tree grammar.

A pattern recognition for use in robotics was developed by [Ouriachi, 1980], where in structure of different machine components were inferred from image features like edges. A polygonal approximation of these features is translated into sentences and a finite state grammar is learnt.

More recent work in this area is based on inferring *bidimensional* objects [Sainz and Sanfeliu, 1996]. A grammar formalism developed by the same authors, **ARE (Augmented Regular Expressions)**, was used to describe two dimensional traffic signals. The grammar was itself learnt and its performance was shown to be tolerant towards noise.

2.5 Syntax Directed Translations

A grammar can be used to model different "domains" having similar structure. Grammars are purely syntactic models and thus using this common interface, patterns in different domains can interact. *Syntax directed translations* allow this interaction.

These translations have been extremely useful in error-correction parsing [Thomason, 1975]. The two domains that we referred to above in this case, are the error prone channel and the actual fact that exists in the world. By "correcting" noise from the data channel, one can infer the ground truth.

2.5.1 Definitions

The following definition has been taken from [Bunke and Sanfeliu, 1990]. A *translation* from language L_1 to language L_2 is a relation M in $L_1 \times L_2$, where xMy means "output string y in L_2 is a translation of input string x from L_1. A *syntax directed translation* is a generalization of context-free grammar, say $G = (N, T, P, S)$ into a *translation schema*, $T = (N, T_i, T_\theta, R, S)$, where N, T_i and T_θ are finite sets of non-terminals, input terminals and output terminals respectively. R is the finite set of translation rules. Each such rule "locks"

together the input generation and the corresponding output generation into one single rule. Thus the rules are of the form $A \rightarrow \alpha, \beta$, where β is a translated version of α. There needs to be some structural similarity (in terms of positions and types of non-terminals), between α and β so that the entire generation process remains essentially the same for both the channels. Moreover, associated non-terminals in the two channels must be rewritten simultaneously.

To illustrate these translations, consider the schema

$S \rightarrow aSc, aSc$

$S \rightarrow b, b$

$S \rightarrow a, b$

The language that is of interest is given by the output channel. This language is $L = \{a^n bc^n | n \geq 1\}$. However, the input channel observes this data and sometimes misreads b to be a. In that case, the translation schemes mentioned in the grammar would do the necessary correction and will restore the correct string.

2.5.2 Some Applications

Error correction is a very important application that uses *syntax directed translations.* One drawback of this error correction model is that, the model assumes knowledge about all kinds of errors. Thus, in the example illustrated above, it was assumed that the only kind of error possible was a b misinterpreted to be a. This need not be true in practice. Models which do not assume this knowledge are more generic, but less efficient. One way in which these two could be combined is by first preprocessing the input string using whatever *error knowledge* is available. Given the "partially corrected" string, one can then apply the more generic approach on the simpler matching problem.

Moreover, this scheme could be extended to multiple dimensions. Order n schemata have been proposed to study images that vary with time [Fan and Fu, 1979]. The structure of the images remain essentially the same. However, the changes are modeled by the translations. Syntax directed models have also been used for performing tree translations [Fu and Fan, 1986] which are useful in *tree-grammars* (discussed later).

2.6 Picture Language

We now discuss practical applications of string grammars in the area of *picture recognition*. Research on syntactic processing of pictures started in the 60s. There was active interest in graphic data-based "conversation programs", and this led to increased work in picture processing. During the same time, important developments were taking place in the field of formal languages. Thus, understanding of the applicability of languages in the domain of pictures was an important area.

One of the first papers [Kirsch, 1964] argued that "*the important fact about natural language text and pictures is that both have a syntactic structure which is capable of being described to a machine and of being used for purposes of interpreting the information within a data processing system*". Much work at that time was being done on analyzing "bubble chamber images" and recognition of hand-written English characters. Syntactic description of these were given in [Narasimhan, 1966]. Existing picture processing techniques at that time were mainly based on "prototypes". However, the nature of the "bubble chamber images" made the application of this approach extremely difficult. Thus, the need for a descriptive scheme for picture analysis and recognition was felt. In an article written by Narasimhan in the book [Kaneff, 1969], a generic *picture language* was described and pointers were provided as to what such a generic language

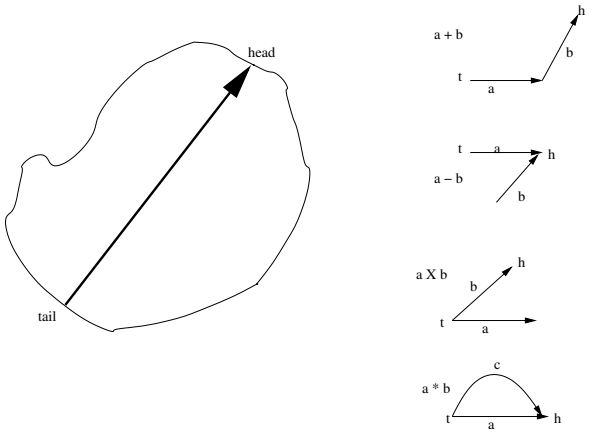

Figure 1: Primitive block and operators of *PDL [Shaw, 1969]*

must comprise of. Based on this work, many languages based on line drawings were introduced. Some of these have been discussed in the next couple of sections of this paper.

2.7 Example : Picture Description Language (PDL)

String grammars generate patterns where the only relation between primitives is concatenation with neighboring symbols. This limits their use for analysis and recognition of 2D or 3D objects. One way to extend the use of string grammars is to generalize the basic primitives and the notion of concatenation. *Picture Description Languages* [Shaw, 1969] do this by representing every primitive part by a directed line segment, and introducing operators which "concatenate" these primitives in a convenient way. An important point to note is that, these operators are also part of the basic primitives. Thus, these languages generalize the notion of *terminals* to relational symbols.

Each primitive part of a pattern has exactly two points, *head* and *tail*, where it can be linked or connected to other patterns. At any point in the generation process, the pattern continues to have two attaching points. Patterns are attached using *operators*. There are four binary operators denoted by $+, -, \times$ and $*$. In addition to these, there are two unary operators $\sim$ and $\backslash$. The first operator simply reverses the direction of the edges in the pattern, while $\backslash$ serves to allow replication of a particular pattern.

Example of a "house" shaped pattern, derived from such a *PDL* is shown. These languages were initially used for analyzing spark and bubble chamber photographs. The pictures describable by *PDLs* are said to be *graph-representabl*e since these languages essentially generate graphs. Depending on the primitive used, arbitrarily complicated designs could be generated and analyzed. These included electrical circuits, text and detailed flowcharts [Shaw, 1969].

PDLs have been used primarily for line drawings. The system MIRABELLE [Masini and Mohr, 1983] was developed in the 80s as a general-purpose system to recognize hand-drawn sketches composed of lines. It was also able to perform labeling of different parts of the drawing. The contribution of this work was the adaptation of Miller's parser [Miller and Shaw, 1968] which was originally developed for speech recognition. Most of the parsing algorithms that exist in formal language literature proceed from left to right of a one dimensional string. Global parsing algorithms proceed in parallel with all the primitives, and can be

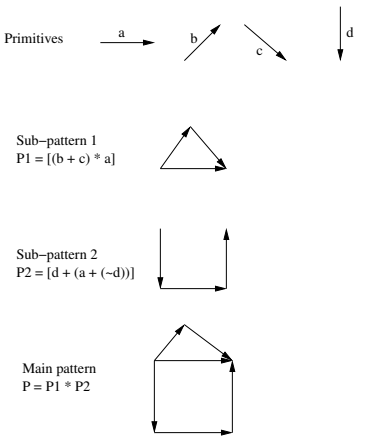

Figure 2: An example of a pattern generated by a *PDL [Bunke and Sanfeliu, 1990]*

applied without the left-to-right restriction. However, these algorithms take into consideration all possible constructions and are space-inefficient. Miller [Miller, 1974] had developed a local parsing scheme as an improvement over this. This scheme was modified and adapted into the MIRABELLE system and an alternating bottom-up and top-down parser was developed. Using this parser, hand drawings of common tools like hammer, and sketches of house drawings were interpreted.

PDL patterns always had only two attaching points which limited their use in defining more complicated drawings. This led to development of *PLEX* structures described in the next section.

2.8 Example : PLEX

Plex structures [Feder, 1971] are similar to *PDLs.* However, they can have any number of concatenation points. Such a *n-attaching point entity* is called a *nape*. Structures formed by combining *napes* are called *plex structures*. A *nape* denoted by *hor(1,2,3)*, denotes a horizontal line segment with 3 attaching points, labeled by numbers. More complicated *napes* are defined by first specifying a list of primitive blocks (*hor,ver*), and then specifying their connections. Some examples have been shown in the figure.

The *nape (ver, hor, hor)(110,210)()* denotes a structure which is composed of one *ver* and two *hor*. This forms the first component of the description. The second component describes how they are connected with each other. *110* denotes that point *1* of *ver* is connected to point *1* of *hor*. *201* denotes that point *2* of *ver* is connected to point *1* of the second *hor*.

A string grammar that generates a set of *plex structures* is called a *plex grammar*. The terminals in these grammars correspond to *napes*. There may also be some non-terminal *napes*, which are described by a set of attaching points. The third component of a *nape* description is used to specify exactly that.

Plex grammars have been used for representing 3D objects [Lin and Fu, 1984]. They were the first grammar concept to come up with description of real world objects. In this work, they also described a Plex parser for recognizing plex structures. A more recent application of these structures has been in the field of engineering drawing. One such application is the analysis of dimensions in a technical drawing [Collin and Colnet, 1994]. In this, the document is first digitized and connected components are extracted. These components are then recognized as segments, arrows or text blocks. The dimension model is defined

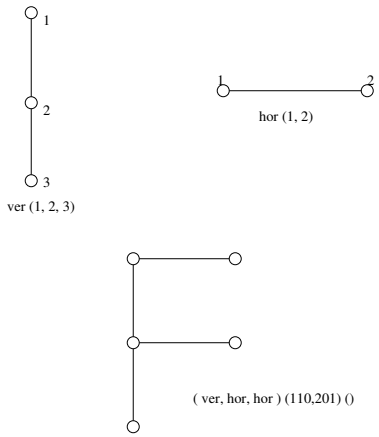

Figure 3: Examples of some *nape* structures [Bunke and Sanfeliu, 1990]

using a *dimension grammar* (first defined by [Dori and Pnueli, 1988]), and the extracted set of primitive components are then parsed by this grammar. The underlying structure is that of the *plex grammar.* The parser used also handles noise, which may be present due to the digitization process. Parsers for *plex languages* are more complex than those corresponding to the strings. These were proposed and developed by [Fu and Shi, 1983], and [Flasinski, 1989].

2.9 Use of grammars in activity recognition

Activity recognition has been an important research topic for the past few years. The problems range from interpretation of basic tasks like sitting or walking to a more complicated problem of action recognition. Some of the first attempts that were made in activity recognition were based on **HMM**s [J. Yamato and Ishii, 1992]. HMMs stand for Hidden Markov Models and they are the stochastic counterpart for finite state machines. In this work, discrete HMMs were used to recognize six tennis strokes performed by three subjects.

One of the first papers that appeared in the use of grammatical approach for activity recognition was [Brand, 1996]. Discrete events were recognized based on blob interactions such as overlapping of objects. The grammar used was non-probabilistic (discussion on probabilistic or *stochastic* grammars is present in Section 3). Recognition of visual activities using stochastic parsing was performed by [Ivanov and Bobick, 2000]. In this paper, the authors describe how activities extended over time (like interactions in a parking lot) were first split into events using probabilistic event detectors. The sequence of events was then parsed to a stochastic context-free grammar for recognition purposes. Applications of this work in gesture recognition was also demonstrated. A more complicated recognition task was performed by [Minnen et al., 2003]. The authors designed *expectation grammars* for modeling the sequence of events. They used stochastic context-sensitive grammars to model the *Tower of Hanoi* task, and were successful in identifying sub-events correctly even when the exact sequence of events were not deterministic.

3 Stochastic Grammars

Grammars can be extended to incorporate probabilistic information. This information allows modeling of the probability distribution over the space of patterns. Given a set of such probability distributions over different grammar classes, it is easy to classify new patterns. We just need to compute for each class, the probability of the pattern belonging to that class (by computing the parse tree), and then choosing the class with the highest value. Even if an exact parse is not possible, one can perform "an approximate" parse and get the probabilities. Probabilistic information allows us to use statistical techniques not only for finding best matches, but also for performing robust feature detection guided by the grammar structure.

3.1 Definitions

A *stochastic grammar* is obtained from the characteristic (base) grammar, by associating a probability with every rule, such that, the sum of the probabilities of rules, with identical left hand side, add up to 1. Intuitively, if we see the generating process of a string, at any stage, the probability of a "rewriting rule" getting fired for a left-hand side sub-string (a non-terminal in case of *context-free grammar*), is based on the discrete distribution of the choices that exist; the distribution for the i^{th} left-hand side (say α) being specified by p_{ij}, where j is the index for the set of productions, whose left-hand side is α.

The following definition is taken from [Bunke and Sanfeliu, 1990]. For simplicity of notation, we will consider *context-free grammars.* Analogous definitions of *stochastic grammars* (used by [Minnen et al., 2003]) could be provided for other grammar forms too. So a generic production $\alpha \rightarrow \beta$ will now have a more specific form $A \rightarrow X_{ij}$, where X_{ij} will be strings over terminals and non-terminals. Consider the grammar $G = (N, T, P, S)$ where for notational convenience, we have

$N = \{A_1, A_2 ... A_m\}$ as the set of non-terminals, and the set of productions as

$A_1 \rightarrow X_{11}, A_1 \rightarrow X_{12}, ..., A_1 \rightarrow X_{1n_1}$ .....

$A_m \rightarrow X_{m1}, A_m \rightarrow X_{m2}, ..., A_m \rightarrow X_{mn_m}$ where $X_{ij} \in (N \cup T)^+$.

A *stochastic* (context-free) *grammar* is a four-tuple $G^s = (N, T, P^s, S)$, where N, T and S are same as the base *context-free grammar* (non-terminals, terminals and initial symbol), and P^s is the set of productions, of the form

$(A_i \rightarrow X_{ij},\ p_{ij})$ such that $A_i \in N$, $X_{ij} \in (N \cup T)^+$ and $\sum_{j=1}^{n_i} p_{ij} = 1$ for $i = 1, ..., m.$, where n_i denotes the number of productions with left hand side as A_i.

3.2 Probabilistic Modeling

We now consider the use of the probabilistic knowledge that is associated with *stochastic grammars*. To motivate this, it is important to understand that a grammar represents a set $L(G)$ of strings, and the information that we might want to associate with this set would be a *probability distribution* over the possible instances of this set $L(G)$. Thus given a grammar G^s, and a string $x \in L(G^s)$, we might want to know the probability of the grammar rules generating this string, denoted by $p(x|G^s)$. This can further be extended to consider a set of pattern classes, say G_i^s, and then asking the question of the most likely class for a given string x.

Let $G^s = (N, T, P^s, S)$ be a *stochastic grammar* and $G = (N, T, P, S)$ be the corresponding base grammar. Then, as specified in [Bunke and Sanfeliu, 1990],

1. If $x \in L(G)$ is unambiguous and has a derivation $S = w_0 \rightarrow^{r_1} w_1 \rightarrow^{r_2} .... \rightarrow^{r_k} w_k = x$, where $r_1, r_2 ... r_k \in P$, then the probability of x with respect to G^s is given by

$$p(x|G^s) = \prod_{i=1}^{k} p(r_i)$$

   This particular definition of probability does not ensure that the function $p(x|G^s)$ is a probability distribution. This problem of *grammar consistency* is described more in [Fu, 1974].

2. If $x \in L(G)$ is ambiguous and has l different derivation trees with corresponding probabilities $p_1(x|G^s)$, $p_2(x|G^s), ..., p_l(x|G^s)$ then the probability of x with respect to G^s is given by

$$p(x|G^s) = \sum_{i=1}^{l} p_i(x|G^s)$$

Thus, to get the probability of an unambiguous string x, we simply multiply the probabilities of the productions that are applied during its derivation. If there are multiple ways of deriving x, then for each derivation, we compute the probability and sum them up.

Example: Consider the grammar $G = (N, T, P, S)$ where $N = \{S, A\}$, $T = \{a\}$, $P = \{S \rightarrow aA,\ S \rightarrow aaA,\ A \rightarrow aa,\ A \rightarrow a\}$ with the probabilities being $\{0.5, 0.5, 0.3, 0.7\}$. Then the probabilities of the strings contained in $L(G) = \{aa, aaa, aaaa\}$ are $\{0.35, 0.5, 0.15\}$.

Now let us consider the problem of classifying a given string x, into one of the n classes (taken from [Bunke and Sanfeliu, 1990]). The classes are each described by a *stochastic grammar* G_i^s. Let the *prior* probability of the pattern classes be denoted by $p(G_i^s)$. Then, if the probability of the string be denoted by $p(x)$, we have by Bayes' formula

$$p(G_i^s|x)p(x) = p(x|G_i^s)p(G_i^s)$$

where $p(x|G_i^s)$ is computed as described above. The expression $p(G_i^s|x)$ is called the *posterior* probability of G_i^s. Thus, if x can be generated by more than one grammar G_j^s, $j = 1..n$, a meaningful classification would be obtained by

$$argmax_{G_j^s}\{p(G_j^s|x)\}$$

3.3 Stochastic Parsing

In this section, we review the different parsing methods that exist for *stochastic grammars*. The problem of parsing for *regular grammars* and *context-free grammars* has been well studied. In the case of *regular grammars*, the characteristic structure of the grammar makes the parsing a simple task. Parsing in *context-free grammars* is non-trivial, and several methods have been proposed to accomplish this task [Aho and Ullman, 1972]. In case of *stochastic grammars*, the task is to find out the most likely "parse" of a given string. In the stochastic version of *regular grammars*, this problem has been well studied in the context of **HMM**s [Rabiner and Juang, 1986]. However, in this paper our focus is on non-regular grammars. For *stochastic grammars*, the *inside* algorithm is most widely used. In all these algorithms, there is a basic assumption about the correctness of the input string. But in practice, this need not be the case. Thus, we go on to discuss parsing when the input string is itself noisy.

3.3.1 Inside algorithm

The *inside algorithm* is a dynamic programming procedure for computing the probability of a string, and the most likely parse of a particular string given a *stochastic context-free grammar*. The number of parse trees for a particular string may be exponential in number and thus, computing all parse trees is not a good idea. Instead, by following a bottom-up approach of parsing smaller strings and storing the results, the computation of the probabilities and *most likely* parse can be done in polynomial time. Material in this section has been taken from [Manning and Schutze, 2001].

We first describe some notations. The string that we are parsing is denoted by $w = w_0 w_1 .....w_n$, and a particular substring $w_p ... w_q$ is denoted by w_{pq}. The grammar, as usual is denoted by the symbol $G = (N, T, P, S)$, where the set of non-terminals is indexed by j, as N^j, with N^1 denoting the initial symbol. The *inside probability* $\beta_j(p,q) = P(w_{pq}|N^j_{pq}, G)$ is the probability of generating $w_{pq} = w_p w_{p+1} .... w_q$ starting from N^j_{pq}, where the subscript is for denoting the positions in the string.

Now, if we want to compute the probability of a string w being generated by a stochastic grammar, we perform the following dynamic programming procedure also known as the *inside algorithm*. We can assume that the grammar is in *Chomsky Normal Form*, i.e., all productions are of the form $A \rightarrow BC$ or $A \rightarrow a$ where A, B and C are non-terminals and a is a terminal. Thus, in the present notation, the productions are of the form $N^i \rightarrow N^j N^k$ or of the form $N^i \rightarrow w_l$.

1. **Base Case:** For each substring w_k of length 1, compute the inside probabilities $\beta_j(k,k)$ of non-terminal N^j generating w_k.

2. **Induction:** Now, for finding $\beta_j(p,q)$ where $p < q$, we see all possibilities of splitting w_{pq}into w_{pd} and $w_{d+1,q}$, and recursively compute the corresponding inside probabilities $\beta(p,d)$ and $\beta(d+1,q)$. Since we are using a bottom-up approach, these probabilities will already be stored in our tables. Also, we have to consider all possible productions that are applicable at N^j for analyzing the possibilities of splitting w_{pq}. More formally, we have to perform

$$\beta_j(p,q) = \sum_{r,s} \sum_{d=p}^{q-1} P(N^j \rightarrow N^r N^s) \beta_r(p,d) \beta_s(d+1,q)$$

By computing $\beta_1(0,n)$, we can obtain the probability of string w being generated by G.

A similar technique could be used for finding the *most likely parse* of a string. In this, we have to store two sets of information for each substring w_{pq} and non-terminal N^j. These are $\delta_j(p,q)$, which contains the highest inside probability parse of a subtree N^j_{pq}. The second is $\psi_i(p,q)$ which stores the rule which led to the most probable parse from non-terminal N^i. Then a similar bottom-up algorithm could be followed :

1. **Base Case:** For each substring of length 1, compute $\delta_i(p,p) = P(N^i \rightarrow w_p)$.

2. **Induction:** For generating w_{pq}, the highest probability parse is computed using

$$\delta_i(p,q) = max_{1 \leq j,k \leq n, p \leq r < q} P(N^i \rightarrow N^j N^k) \delta_j(p,r) \delta_k(r+1,q)$$

and the rule is stored using

$$\psi_i(p,q) = argmax_{(j,k,r)} P(N^i \rightarrow N^j N^k) \delta_j(p,r) \delta_k(r+1,q)$$

3.3.2 Parsing for noisy inputs

When the input stream is noisy, we do not consider the actual string, but a set of *beliefs* about the correctness of a particular symbol at a position. Material in this section has been taken from [Bunke and Sanfeliu, 1990]. Thus we consider a sequence of vectors $[c_1(x_1)\ c_1(x_2)....\ c_1(x_n)]$, ... $[c_m(x_1)\ c_m(x_2).....\ c_m(x_n)]$, where $c_i(x_j)$ is a measure of certainty that the symbol $x_j \in T$ is correct at position i. Thus, for a particular input string $x =< x_{i1}x_{i2}.....x_{im} >$, we define the uncertainty by

$$c(x) = \sum_{j=1}^{m} c_j(x_{ij})$$

The certainties will be defined on the basis of the actual input string observed as well as the belief on the particular string positions. The parsing procedure will then determine the string x which is compatible with the given grammar and has maximum certainty. A fast and simple algorithm for solving this for a *regular grammar* was solved in [Bunke et al., 1984]. Its extension to *context-free grammars* was given in [Bunke and Pasche, 1988].

4 Higher Dimensional Grammars

Grammars can be extended to operate on higher dimensions. As pointed out earlier, *string grammars* are one-dimensional and the only relation that exists between neighboring symbols is that of concatenation. In applications of computer vision and graphics, such one dimensional structures are very restricted. There are two ways to extend this grammar to higher dimensions. The first is to generalize the notion of symbol and concatenation, and map a sequence of symbols to some expression language which can be interpreted as a drawing/model. *PDL*s and *plex structures* used this approach. Another more fundamental approach is to generalize the notion of *strings*, to higher dimensions.

Since we are dealing with images, one obvious extension could be *arrays*. A string of length n could be interpreted to be an array of dimension $n \times 1$. Another way to generalize a *string* is to observe that the symbols are connected by a *neighborhood* relation. Thus, they are essentially *linear graphs*, with every node (except the first and the last) having 2 nodes. *Graph grammars* and their special case *Tree grammars*, have been widely used in pattern recognition applications. Before discussing these various forms of grammar, we first discuss *parallel grammars* and *isometric grammars* which will lead us to these other forms.

4.1 Parallel grammars

In the grammars that we have discussed till now, there was an inherent sequentiality associated with it. We say that the grammar $G = (N, T, P, S)$ accepts a string x if there exists a sequence of derivation steps $S \rightarrow w_0 \rightarrow w_1 \rightarrow ..... \rightarrow w_k = x$, where each derivation step ($\rightarrow$), denotes application of one rule. In the case of *parallel grammars*, this restriction is lifted. Thus, a *parallel grammar* $G^{\|} = (N, T, P, S)$ accepts a string x, if there exists a sequence of *parallel* derivation steps $S \Rightarrow w_0 \Rightarrow w_1 \Rightarrow ..... \Rightarrow w_n = x$, where each derivation step ($\Rightarrow$), denotes application of all possible rewriting rules that are possible for an intermediate string.

Since application of rules takes place in a parallel fashion, except for some simple cases (*regular grammars*), the languages accepted by a *parallel grammar* $G^{\|}$ and the corresponding *sequential grammar*, G are

not the same. Consider for example, the *context-free grammar* denoted by the rules $S \rightarrow SS$, $S \rightarrow a$. Now, the languages accepted by the corresponding grammars are

$$L(G) = \{a^n | n \geq 1\}$$

$$L(G^{\parallel}) = \{a^{2n} | n \geq 1\}$$

In fact, the language that is accepted by $L(G^{\parallel})$ is not context-free even though the underlying grammar is context-free. In the case of *regular grammars*, an intermediate string always consists of a single non-terminal and hence, there is no difference between $G^{\parallel}$ and G.

One problem with *parallel grammars* is that for a rule $\alpha \rightarrow \beta$, instances of α are allowed to overlap. So if there exists a length decreasing rule $XXX \rightarrow ZZ$, then starting from X^m, a *parallel grammar* derives $Z^{2(m-2)}$. A more important problem is that *parsing* and *generation* are no longer inverses of each other. For instance, applying $XX \rightarrow ZZ$ to XXX in parallel yields $ZZZZ$, but applying $ZZ \rightarrow XX$ to $ZZZZ$ in parallel yields $XXXXXX$. Thus parsing is an issue in *parallel grammars*.

Parallel grammars have been widely applied in computer graphics applications in the form of *L-systems*. We discuss some of the applications of *L-systems* in the next section.

4.1.1 L-systems

First introduced as a mathematical theory to explain growth and development of multicellular organisms by Aristid Lindenmayer, L-systems have since then made a big impact in areas of simulation and modeling. These systems were developed by [Prusinkiewicz and Lindenmayer, 1990] into a system which could express plant models. However, now L-systems are being used for various other applications in computer graphics.

L-systems are formal grammars which can generate strings. These grammars are in general *context-sensitive* parallel grammars. The rules that generate these strings and the symbols themselves have particular semantic and visual interpretations based on the problem being modeled. The string of symbols has traditionally been interpreted as a sequence of commands given to a turtle. The type of motion exhibited by the turtle, then describes the end result. Application of rules in L-systems occurs in a parallel fashion, thereby resembling natural sub-division processes in cellular organisms.

Applications of *L-systems* in computer graphics includes modeling of plants [Kurth, 1994]. In this work, parameterized L-systems were used not only for simulating differential plant growth but also for modeling interactions with external environment and other plant structures. Another important application was in the artificial generation of random cities [Parish and Muller, 2001]. In this work, *open L-systems* were used for modeling street maps and building structures, by generating templates and then interacting them with *external functions*.

4.2 Matrix Grammars

Matrix grammars are used for generating two-dimensional strings. The synthesis of the matrices occur by first generating a row of symbols. This row is the top row of the matrix, and every symbol is a start symbol for some grammar. The symbols are then expanded into columns, with the restriction that the lengths of the final strings should remain same for every column. The column languages are *regular* since growth of strings in one-dimension has to be ensured. This also makes parsing easier.

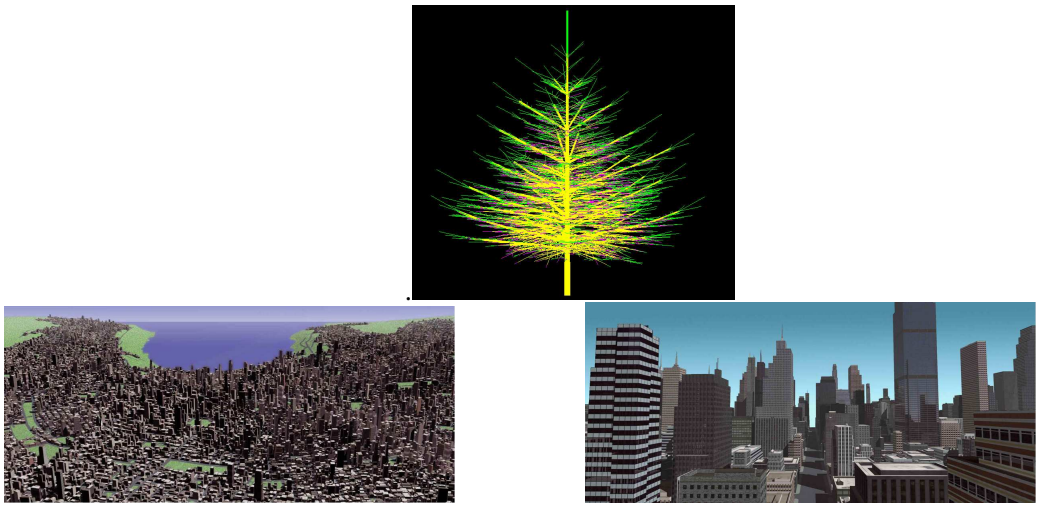

Figure 4: Applications of *L-systems* : Modeling of plants and cities

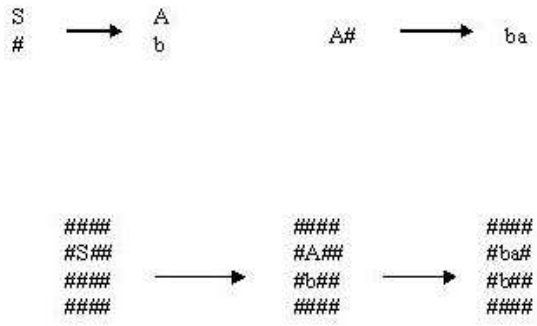

Figure 5: Example of an *array grammar* and a sequence of applications of *array rules* [Bunke and Sanfeliu, 1990]

To put the above ideas more formally we use the definition from [Bunke and Sanfeliu, 1990]. We define a *matrix grammar M* to be an $(n+1)$ tuple $(G, G_1, G_2, ..., G_n)$ where G is a grammar, and $G_1, ..., G_n$ are finite state grammars such that the terminal symbols of G is the set $\{S_1, ..., S_n\}$ of initial symbols of the $G_i's$. M operates by first generating a "horizontal" string of $S_i's$ and then starts expanding every symbol according to the corresponding grammar rules in parallel. By terminating at the same time, one can ensure the "column length" of the individual strings to be the same.

4.3 Array grammars

Array grammars are used to model languages which are connected sequences/arrays of symbols in two (or possible more) dimensions. *Matrix grammars* generate only rectangular arrays, while one might be interested in sequences of vectors of different lengths. Moreover, we also do not want to limit the scope of these grammars to two dimensions. Thus, *array grammars* are useful for generating more complicated

sequences of symbols.

Definitions here are taken from [Bunke and Sanfeliu, 1990]. The entire string is modeled as a function from the set of (integer-coordinate) lattice points to the set of terminal symbols and the blank symbol (#). An example of an array string is shown in the figure. The notion of *connectedness* in these strings is defined on the basis of their coordinates. Two points (i, j) and (h, k) are said to be neighbors (and hence connected), if

$$|i-h|+|j-k|=1$$

An array Σ is called *connected*, if for all non-# symbols A, B in Σ, there exist a sequence $A = A_0, A_1, ..., A_n = B$ of non-# symbols in Σ (a "path"), such that A_i is a neighbor of A_{i-1}. Definition of an *array grammar* keeps all the non-# symbols connected. If this restriction is not imposed, then the distance between disconnected components may become arbitrarily large, and thus modeling the resultant string using local computations will not be feasible. Thus, there are some restrictions on the way production rules are defined. More particularly, the rules follow the structure of an *isometric grammar.* In this grammar, for any rule $\alpha \rightarrow \beta$, the "size" of α and β must be the same. This looks like a big restriction, since then the strings generated will always be of finite length (of the start configuration), and thus the number of strings will always be finite. However, the language of an *isometric string grammar* G is defined to be the set of terminal strings τ such that the infinite string $\#^{\infty}\tau\#^{\infty}$ can be derived in G from the infinite initial string $\#^{\infty}S\#^{\infty}$. Hence, by incorporating rules containing the * symbol, arbitrary length strings could be generated.

Thus, in an *array grammar*, one starts with an infinite two-dimensional array consisting of #'s with the initial symbol located at one place. Each rule of the grammar, replaces one array with another. The rules satisfy the *isometry* constraint since otherwise, it is not clear how one can replace a bigger array, with a smaller array and keep the resulting embedding connected. Similar problems arise when we have a smaller array replacing a bigger array in a production.

4.4 Graph and Tree grammars

Both strings and arrays can be considered to be specific instances of a general graph structure. In this, the relationships between neighbors could be arbitrary in number (rather than 2 in case of string grammars and 4 for array grammars). Modeling of graph grammars is also based on *sub-graph recognition* and *replacement.* Thus for any rule of the form $\alpha \rightarrow \beta$, α and β would represent subgraphs and their replacement would imply removing α from the total graph and replacing it by β.

However, embedding of one graph into another would require more specifications than just the individual graphs. When we replace substring α by substring β, or subarray α by a geometrically identical subarray β, such an issue was not encountered. In this case however, one must specify the new node connections that should be made between the existing graph (with α removed) and the new graph β.

Applicability of graph grammars in the field of vision and pattern recognition has been limited. Only a few applications are well known in the literature [Brayer and Fu, 1975], [Brayer, 1977], [Pavlidis, 1972]. A major issue in working with graph grammars is the *subgraph recognition problem*. Identifying the subgraph were a particular replacement rule should be applied is a non-trivial problem. In a more formal setting, this is referred to as the *subgraph isomorphism problem*. The basic algorithm was given by [Ullmann, 1976]. Since then, much work has been done on finding efficient heuristics for solving this basic problem.

The main difficulty with using graph grammars has been that efficient parsers for graph languages do not exist. One such parser developed by [Fu and Shi, 1983] was based on *attributed expansive graph grammars*.

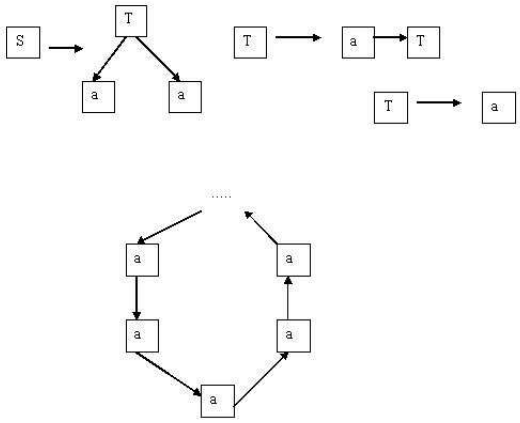

Figure 6: Example of a *graph grammar* and a graph generated from this grammar[Bunke and Sanfeliu, 1990]

These grammars defined graphs which through some re-numbering procedures were directly translated into string grammars. Thus application of string parsing techniques were easily applicable in this case.

An important special case of graph grammars are *tree grammars*. Here in the rules $\alpha \rightarrow \beta$, α and β are labeled sub-trees. We apply the rule to a tree σ by replacing some occurrence of α, as a subtree of σ, by β. In this case, the connections to be made are not ambiguous since β is simply attached to the parent of α. Examples of graph and tree grammars are depicted in the figure. Tree grammars are more tractable than graph grammars since the replacement rule substitutes *entire subtrees*. As a result, parsing of such grammars becomes simpler. Efficient tree parsers based on *tree automaton* exist in the literature. A very good discussion on *matching tree structures* is available in Chap 6 of [Bunke and Sanfeliu, 1990].

4.5 Programmed grammars

Programmed grammars are extensions of our usual notion of string grammars with *success* and *failure* fields associated with every production. These fields provide an implicit context to any production that is applied. So although their representations are simple, one can model these grammars so that they can generate a more expressive language, than what the base grammar can.

More formally, as described in [Bunke and Sanfeliu, 1990], a *programmed grammar* G_p is a five-tuple (N,T,J,P,S) where N, T and P are finite sets of non-terminals, terminals and productions respectively. S is the starting symbol and J is a set of production labels. A production labeled p_i of the form, $\alpha \rightarrow \beta$ has two sets associated with it, namely the *success field* U and the *failure field* W, $U, W \subset J$.

To see how a *programmed grammar* works, consider the following grammar.

(1) $S \rightarrow AB$

(2) $A \rightarrow aA$ (3) $A \rightarrow bA$ (4) $A \rightarrow a$ (5) $A \rightarrow b$

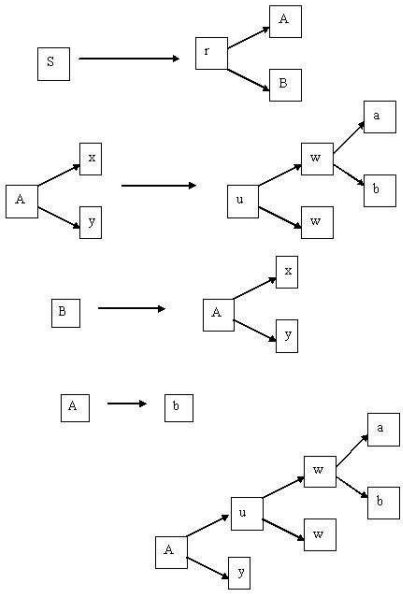

Figure 7: Example of a *tree grammar* and a tree generated from this grammar

(6) $B \rightarrow aB$ (7) $B \rightarrow bB$ (8) $B \rightarrow a$ (9) $B \rightarrow b$
and consider the program table to be

Rule No	U	W
(2)	(6)	None
(3)	(7)	None
(4)	(8)	None
(5)	(9)	None
(6) or (7)	(2), (3), (4) or (5)	None
(8) or (9)	None	None

The language generated by this grammar is $\{\sigma\sigma|\sigma \in (a+b)^+\}$. No context-free language can generate this set. Thus, using *programmed grammars*, one can make use of the simplicity of the language, while also using it to model more complicated features. These languages have been used for *character recognition tasks* [K.H. Lee and Kashyap, 1988]. In this particular work, *attribute dependent regular programmable languages* were used to build a model for *Hangul*, the Korean alphabet scheme.

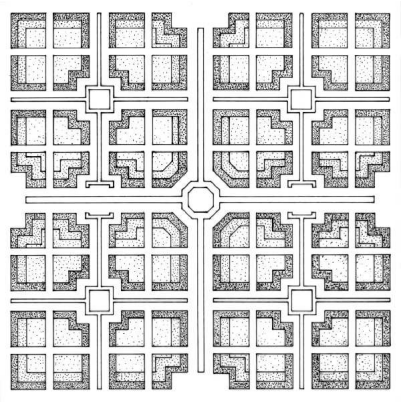

Figure 8: Layout of the historic *Mughal Gardens*

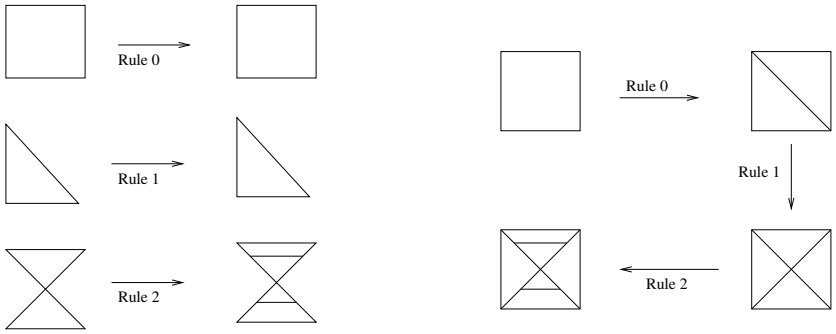

Figure 9: Example of a shape grammar and a shape generated from this grammar

4.6 *Shape Grammars*

Shape grammars have been used mostly by the architectural community to model design patterns of different cultural and historic works of art. For instance, work on understanding the design principles behind the historic Mughal Gardens, has been based on shape grammars. They were introduced by George Stiny in the 1970s [Stiny, 1975]. The basic blocks in these grammars are two-dimensional shapes and some basic generative rules which could range from exact subdivisions and additions of shapes to parametric operations. Thus, issues like how the Mughal Gardens were partitioned into separate areas, what kind of symmetry was exhibited by the different parts etc., could all be modeled using different rules and parameters in these grammars.

A grammar in this category is defined as $G =< S, L, P, I >$ where S is the set of shapes, L is the set of symbols (labels), P is the set of production rules and I is the initial shape. Shape generation follows the *subshape recognition* and *subshape replacement* paradigm. As can be seen in the figure shown, the replacement of shapes in Rule 2 results in a new shape.

Applications of *shape grammars* are mainly in modeling and structure analysis. The *Ice-ray grammar*, developed by [Stiny, 1977] can generate Chinese lattice patterns found in the traditional architecture of China. These patterns resemble the structures that get formed when water solidifies into ice.

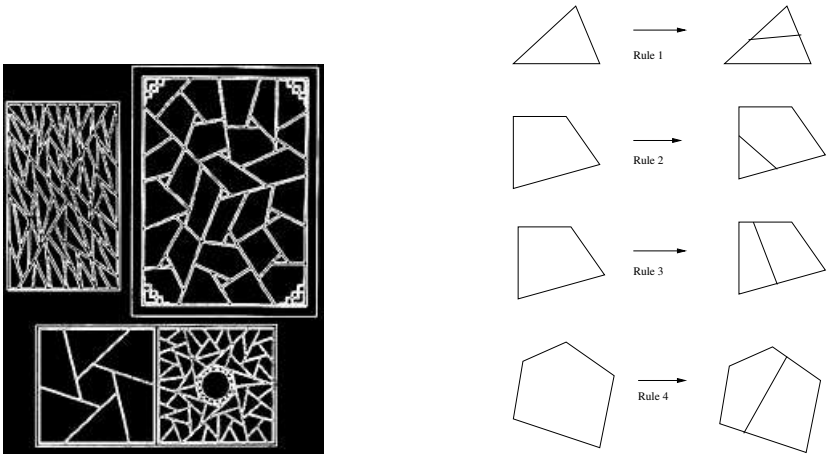

Figure 10: An example of Chinese lattice patterns and the four rules of the *ice-ray grammar*

[Koning and Eizenberg, 1981] defined a parametric shape grammar that could generate the compositional forms and specify the functional zones of *Frank Lloyd Wright's prairie-style houses*. Application of shape grammars in product design was demonstrated by [Agarwal and Cagan, 1998]. Using labeled shape grammars, connections between different parts of the product and their functional implications were considered.

5 Conclusions

In this paper, we have tried to cover most of the important grammar formalisms that have been used in the computer vision and pattern recognition community. If we see the series of developments in the usage of grammars, we do not see a very clear trend. One plausible explanation is, sufficient developments in theoretical aspects of grammatical techniques which might prove suitable for pattern analysis tasks did not take place in parallel. Most of the techniques described were studied in the 70s. Thus in the application domain, these concepts were reused with extensions suitable to the problem in hand.

As aptly pointed out in [Tombre, 1996], syntactic methods lack *generality*. An approach used for one application is not very adaptable to other applications. The inability to handle the primitives and the semantic part of grammatical methods in an independent modular way, further led to application specific algorithms for inference and generation. A clear theory of how a grammar could be learnt from real world data, also exists in a very abstract sense confined mainly to the domain of string and graph grammars. However, the domains where different types of grammars are applied are too varied for such a unified theory to exist.

Thus we conclude that grammatical methods are useful when the problem under consideration has patterns and relationships among sub-patterns. However, recognizing the existence of patterns does not guarantee a direct syntactic solution since the grammatical methods that exist as of today are not generic enough. The robustness and the feasibility of the solution would depend a lot on the actual problem.

References

[Agarwal and Cagan, 1998] Agarwal, M. and Cagan, J. (1998). A blend of different tastes: The language of coffee makers. *Environment and Planning B: Planning and Design*, 25(2):205–226.

[Aho et al., 1988] Aho, A., Sethi, R., and Ullman, J. (1988). *Compilers, principles, techniques and tools*. Addison-Wesley, Reading, MA.

[Aho and Ullman, 1972] Aho, A. and Ullman, J. (1972). *The Theory of Parsing, Translation and Compiling, Vol. 1: Parsing*. Prentice-Hall, Englewood Cliffs, NJ.

[Angluin and Smith, 1983] Angluin, D. and Smith, C. (1983). Introductory inference, theory and methods. *ACM Computing Surveys*, 15:741–765.

[Brand, 1996] Brand, M. (1996). Understanding manipulation in video. In *2nd International Workshop on Automatic Face and Gesture Recognition*, pages 94–99.

[Brayer, 1977] Brayer, J. (1977). Parsing of web grammars. In *IEEE Workshop on data description and management*, Long Beach, CA.

[Brayer and Fu, 1975] Brayer, J. and Fu, K. (1975). Web grammars and their application to pattern recognition. Technical Report TR-EE 75-1, Purdue University.

[Bunke et al., 1984] Bunke, H., Grebner, K., and Sagarer, G. (1984). Syntactic analysis of noisy input strings with an application to the analysis of heart-volume curves. *Proc. 7th ICPR*, pages 1145–1147.

[Bunke and Pasche, 1988] Bunke, H. and Pasche, D. (1988). A new syntactic parsing method and its application to the tracking of noisy contours in images. *Signal Processing IV : Theories and Applications*, pages 1201–1204.

[Bunke and Sanfeliu, 1990] Bunke, H. and Sanfeliu, A. (1990). *Syntactic and Structural Pattern Recognition : Theory and Applications*. World Scientific.

[Collin and Colnet, 1994] Collin, S. and Colnet, D. (1994). Syntactic analysis of technical drawing dimensions. *International Journal of Pattern Recognition and Artificial Intelligence*, 8:1131–1148.

[Dori and Pnueli, 1988] Dori, D. and Pnueli, A. (1988). The grammar of dimensions in machine drawings. *CVGIP:Graphical Models and Image Processing*, 42:1–18.

[Fan and Fu, 1979] Fan, T. and Fu, K. (1979). A syntactic approach to time-varying image analysis. *CVGIP:Graphical Models and Image Processing*, 11:138–149.

[Feder, 1971] Feder, T. (1971). Plex languages. *Info. Sciences*, 3:225–241.

[Flasinski, 1989] Flasinski, M. (1989). Characteristics of ednlc-graph grammar for syntactic pattern recognition. *CVGIP:Graphical Models and Image Processing*, 47:1–21.

[Fu, 1974] Fu, K. (1974). *Syntactic Methods in Pattern Recognition*. Academic Press, New York and London.

[Fu and Fan, 1986] Fu, K. and Fan, T. (1986). Tree translation and its application to time-varying image analysis problem. *IEEE Trans. SMC*, 12:856–867.

[Fu and Shi, 1983] Fu, K. and Shi, Q. (1983). Parsing and translation of attributed expensive graph languages for scene analysis. *IEEE Transactions on PAMI*, 5:472–485.

[Ivanov and Bobick, 2000] Ivanov, Y. and Bobick, A. (2000). Recognition of visual activities and interactions by stochastic parsing. *PAMI*, 22(8):852–872.

[J. Yamato and Ishii, 1992] J. Yamato, J. O. and Ishii, K. (1992). Recognizing human action in time-sequential images using hidden markov model. In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 379–385.

[Kaneff, 1969] Kaneff, S. (1969). *Picture Language Machines*. Academic Press.

[K.H. Lee and Kashyap, 1988] K.H. Lee, K. E. and Kashyap, R. (1988). Character recognition using attributed grammar. *IEEE*.

[Kirsch, 1964] Kirsch, R. (1964). Computer interpretation of english text and picture patterns. *Trans. IEEE EC*, 13:363–376.

[Knuth, 1968] Knuth, D. (1968). Semantics of context-free languages. *Math. Sys. Theory*, 2:127–146.

[Koning and Eizenberg, 1981] Koning, H. and Eizenberg, J. (1981). The language of the prairie: Frank Lloyd Wright's prairie houses. *Environment and Planning B*, 8:295–323.

[Kurth, 1994] Kurth, W. (1994). Growth grammar interpreter grogra 2.4: A software tool for the 3-dimensional interpretation of stochastic, sensitive growth grammars in the context of plant modelling. introduction and reference manual. *Berichte des Forschungszentrums Waldökosysteme der Universität Göttingen, Ser. B*, 38.

[Lin and Fu, 1984] Lin, W. and Fu, K. (1984). A syntactic approach to 3d object representation. *IEEE Trans. PAMI*, 6:351–364.

[Lu and Fu, 1979] Lu, S. and Fu, K. (1979). Stochastic tree grammar inference for texture synthesis and discrimination. *CGIP*, 9:234–245.

[Manning and Schutze, 2001] Manning, C. and Schutze, H. (2001). *Foundations of Statistical Natural Language Processing*. MIT Press.

[Masini and Mohr, 1983] Masini, G. and Mohr, R. (1983). Mirabelle, a system for structural analysis of line drawings. *Pattern Recognition*, 16:363–372.

[Miller and Viola, 1998] Miller, E. and Viola, P. (1998). Ambiguity and constraint in mathematical expression recognition. In *AAAI Nat. Conf. on Artificial Intelligence*.

[Miller, 1974] Miller, P. (1974). A locally organized parser for spoken input. *Comm. ACM*, 17:621–630.

[Miller and Shaw, 1968] Miller, W. and Shaw, A. (1968). Linguistic methods in picture processing - a survey. In *Proc. AFIPS 1968 Fall Joint Computer Conference*, volume 33, pages 279–290.

[Minnen et al., 2003] Minnen, D., Essa, I., and Starner, T. (2003). Expectation grammars: Leveraging high-level expectations for activity recognition. In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*.

[Moayer and Fu, 1975] Moayer, B. and Fu, K. (1975). A tree system approach for fingerprint pattern recognition. *IEEE Trans. on Computers*, 25.

[Narasimhan, 1966] Narasimhan, R. (1966). Syntax-directed interpretation of classes of pictures. *Comm. ACM*, 9:166–173.

[Ouriachi, 1980] Ouriachi, K. (1980). Processus de reconnaissance des formes applicable a un assemblage automatique. *These de 3eme cycle, Universite des Sciences et Techniques de Lille*.

[Parish and Muller, 2001] Parish, Y. and Muller, P. (2001). Procedural modeling of cities. *SIGGRAPH*, pages 301–308.

[Pavlidis, 1972] Pavlidis, T. (1972). Grammatical and graph theoretical analysis of pictures. *Graphic Languages*.

[Prusinkiewicz and Lindenmayer, 1990] Prusinkiewicz, P. and Lindenmayer, A. (1990). *The Algorithmic Beauty of Plants*. Springer-Verlag.

[Rabiner and Juang, 1986] Rabiner, L. and Juang, B. (1986). An introduction to hidden markov models. In *IEEE ASSP Magazine*.

[Sainz and Sanfeliu, 1996] Sainz, M. and Sanfeliu, A. (1996). Learning bidimensional context dependent models using a context-sensitive language. In *13th International Conference on Pattern Recognition*, Viena, Austria.

[Shaw, 1969] Shaw, A. (1969). Parsing of graph-representable pictures. *J. ACM*, 17:453–487.

[Stiny, 1975] Stiny, G. (1975). *Pictorial and Formal Aspects of Shape and Shape Grammars*. Basel: Birkhauser.

[Stiny, 1977] Stiny, G. (1977). Ice-ray: A note on the generation of chinese lattice designs. *Environment and Planning B: Planning and Design*, 4:89–98.

[Thomason, 1975] Thomason, M. (1975). Stochastic sdts for correction of errors in context-free languages. *IEEE Trans. Comp.*, 24:1211–1216.

[Tombre, 1996] Tombre, K. (1996). Structural and syntactic methods in line drawing analysis : To what extent do they work ? In *SSPR*, pages 310–321.

[Ullmann, 1976] Ullmann, J. (1976). An algorithm for subgraph isomorphism. *Journal of the ACM*, 23:31–42.