36

Stochastic Grammars

Not Just for Words!

Mike Lewis

36.1 Introduction
36.2 Formal Grammars
36.3 Generating Sequences from a Grammar
36.4 A Data Structure for Grammars
36.5 Streaming Sequences
36.6 Grammars as Analogues to Behavior Trees
36.7 Grammars as Scripting Engine
36.8 Tuning a Stochastic Grammar
36.9 Feeding a Grammar with Utility Theory
36.10 Conclusion
References

36.1 Introduction

Randomness, when carefully and judiciously applied, can be a powerful tool for augmenting the behavior of game AI agents and systems. Of course, purely random behavior is rarely compelling, which is why having some semblance of pattern is important. One way to do this is to simulate *intent*, making AI decisions based on some kind of deliberate design. Many excellent resources exist for creating the illusion of intentional behavior in game AI—and this very volume is not least among them.

However, there are times when it is useful to tap directly into the realm of randomness. Even if it is purely for the sake of variety, a little bit of random fuzz can do a lot of good for an AI character (Rabin et al. 2014). Occasionally, though, there is a sort of confluence of requirements that makes both purely intentional decision-making and heavily randomized decision-making problematic. It is at those times that *structured randomness* comes into play.

Imagine a mechanism for generating sequences of actions. This mechanism can be tuned and adjusted, either at design-time, or on-the-fly at runtime. It creates a controlled blend between predictable patterns and random chaos. That is to say, the mechanism can be tweaked to create arbitrarily "random-feeling" sequences based on completely customizable factors. Again, this can be done statically or dynamically. Moreover, it can generate any kind of structured data. Sound appealing?

Welcome to the world of *stochastic grammars*.

36.2 Formal Grammars

A *formal grammar* is a tool for describing and potentially manipulating a sequence or stream of data. Ordinarily, grammars are used for textual inputs, and operate on sequences known as *strings*. Although historically envisioned by Pāṇini, circa the 4th century BCE, as a tool for rewriting sequences, grammars are also instrumental in recognizing whether or not a string is *valid* according to the rules of the grammar itself (Hunter 1881).

This recognition of validity, along with *parsing*—extracting syntactic structure from a string—is a key component in both natural language processing and computer languages. A compiler, for example, typically uses a tool called a *parser generator* to convert a grammar into code that can parse strings written by those grammatical rules—that is, programs (Brooker et al. 1963).

As an example, Table 36.1 shows a simple grammar that describes the natural numbers.

This grammar uses Extended Backus-Naur Form, or EBNF (Backus 1959, Wirth 1977). Each row of the table describes a *rule*. The third rule, "Natural Number," can be thought of as the starting point for recognizing or generating a natural number. It specifies a *sequence* of symbols, beginning with a nonzero digit. The comma indicates that the following portion of the rule is concatenated with the leading portion. Next, the braces indicate *repetition*. In this case, any digit may appear, and that sub-rule is allowed to apply zero or more times.

Looking at the actual rule for nonzero digits, there is one additional important symbol, the pipe. This indicates *alternations*, that is, that a choice must be made from several alternatives. Each rule also ends in a semicolon, denoting the termination of the rule.

The net result is that the "Natural Number" rule specifies a nonzero digit followed by any number of digits (including zeros). This matches perfectly with the expectation for what a natural number looks like.

However, grammars need not be relegated to use with pure text or numbers. If a string is defined as a set of data *symbols* with some particular set of meanings, virtually any structured data can be defined with a suitable grammar. Allowing these symbols to carry meanings like "punch" or "attack on the left flank" opens the door for much richer applications than mere operations on words.

Table 36.1 Grammar Describing the Natural Numbers in EBNF

Name of Rule	Rule Matches Strings of this Form
Nonzero Digit	1|2|3|4|5|6|7|8|9;
Any Digit	Nonzero Digit|0;
Natural Number	Nonzero Digit, {Any Digit};

36.3 Generating Sequences from a Grammar

Grammars typically have a selected set of *starting symbols* which control the first rule(s) used when creating a new string. How does one choose which rules of a grammar to follow in order to generate a sequence? If the goal is to exhaustively generate as many strings as possible, then the answer is simply "all of them." Sadly, this is not useful for most nontrivial grammars, because they are likely to contain recursive rules that just never stop expanding.

Suppose that each rule of the grammar is augmented with a weight value. As the output is progressively accumulated, there will (probably) be points where more than one rule from the grammar can be applied. In these cases, the next rule can be chosen via a simple weighted random number generation, using the weights from each available rule. This structure is known as a *stochastic grammar* or *probabilistic grammar*.

Supplementing a grammar with this random selection process is akin to describing how "likely" a given generated string might be. If a sequence describes actions taken by an AI agent, the probability weights control that agent's "personality." Some actions—and even subsequences of actions—can be modeled as more "like" or "unlike" that character, and in this way, a sort of preferential action generation process can be constructed.

Take, for example, a ghost wandering a *Pac-Man*-like maze. At each intersection, the ghost can turn left, turn right, continue forward, or reverse directions. Table 36.2 illustrates a simple grammar that describes these possibilities; note the addition of weights to the Direction Decision rule's alternation pattern.

The ghost AI simply needs to generate a string of decisions and pop a decision from the front of the queue each time it enters an intersection. If an "L" decision is retrieved, the ghost turns left; correspondingly, the ghost turns right for an "R." The "F" decision translates to moving forward in the same direction as before, and "B" indicates moving backward. Obviously in some cases a particular decision may not be applicable, so the ghost can simply pop decisions until one is possible. Should the queue become empty, just generate a new sequence and continue as before.

As described in the table, the stochastic grammar will have an equal chance of making each possible selection. However, the "personality" of the ghost can be adjusted to bias toward (or against) any of the options available, merely by tuning the weights of the grammar.

What other sorts of things can be done with a decision-making grammar? Consider a raid boss in a multiplayer RPG. This boss has two basic attack spells: one that makes enemies in a small region vulnerable to being set on fire, and a separate fireball which capitalizes on this vulnerability. Moreover, the boss has a third, more powerful spell that does huge bonus damage to any foe who is currently ablaze.

The naïve approach is to simply cast the vulnerability spell on as many players as possible, then fireball them, and lastly close with the finishing move. Although this is a workable design, it lacks character and can easily be predicted and countered by attentive players.

Table 36.2 Stochastic Grammar to Control a Ghost in a Maze

Name of Rule	Rule Generates these Symbols
Direction Decision	0.25 L|0.25 R|0.25 F|0.25 B;
Navigation Route	{Direction Decision};

Table 36.3 Grammar Producing Attack Sequences for a Raid Boss

Name of Rule	Rule Matches Strings of This Form
Vulnerability Combo	Vulnerability, {Fireball};
Basic Sequence	Fireball, {Vulnerability Combo}, Finisher;
Attack Sequence	{Fireball}, {Basic Sequence};

Instead, describe the available moves for the boss in the form of a grammar, such as that in Table 36.3. One possible sequence generated by this grammar might look like "Fireball × 3, Vulnerability × 5, Fireball × 4, Finisher, Fireball, Finisher." More importantly, given appropriate probability weights for the rules, this grammar will produce different sequences each time the raid is attempted. Although the overall *mechanics* of the fight are intact, the *specifics* vary wildly. That is to say, players know they must come equipped to defend against fire, but the actual progression of the fight is largely unpredictable.

Although arguably the same results could be had with a more sophisticated AI system, it is hard to compete with the raw simplicity and easy configurability of a stochastic grammar. Certainly, there will be applications for which grammars are not the best choice. However, when used appropriately, the ability to generate a structured, semi-random sequence is a compelling tool to have in one's arsenal.

36.4 A Data Structure for Grammars

The actual implementation of code for generating (and parsing) strings is a rich subject. However, it is quite possible to work with simple grammars using basic, naïve approaches to both parsing and generation. Given a suitable data structure for representing the grammar itself, it is easy to start with the trivial implementation and upgrade to more sophisticated algorithms as needed.

Based on the examples so far, some kind of tree-type structure seems well suited to the task of representing the grammar itself. Each rule can be represented as a node in the tree. A "nested" rule can be pointed to as a child node. The nodes themselves can contain a list of parts, with each part being either a sequence of nodes or a list of weighted alternatives to choose from. Within this structure, nodes can be represented using an abstract base class, with derivative classes for sequences and alternations. The actual generated sequence (or input for parsing) can be represented with a container class such as std::vector or equivalent. Each node should have an interface function for generating (or parsing) the substrings for which it is responsible.

Leaf nodes are the simplest case; they will merely append an element to the sequence and return. These nodes represent the *terminals* of the grammar. Next up, sequencer nodes contain a set of node pointers. When they are asked to generate an element, these nodes traverse the container of child nodes in order, asking each one to recursively generate an element. This process can optionally be repeated randomly, so that the sequence itself appears some random number of times in the final output, in keeping with the rules of the grammar.

Alternations, or choices, are where the magic of a stochastic grammar really happens. These nodes store a set of child nodes, like before, but this time each child has an associated

weight value. As the node is traversed, *one* child node is selected to be traversed recursively, based on a weighted random selection from the available children. (See the accompanying demo code at http://www.gameaipro.com/ for an example implementation.)

36.5 Streaming Sequences

The approach to generation thus far has focused on creating a finite-length sequence, using random weights to control the length and content of each generated string. However, it can sometimes be useful to generate *infinitely long* sequences as well.

Superficially, this might be denoted by setting a sequence node's weight such that it never chooses to stop repeating. However, there is more to the problem—with the methods described so far, generating an infinite sequence in this way will just exhaust available memory and fail to return anything.

There are two basic approaches to "streaming" an infinite sequence based on a generative grammar. On the more sophisticated side, one might allow *any* sequence node to be infinite, regardless of its position in the grammar/tree. This requires some careful gymnastics to preserve the state of a pass when the output is retrieved midway through.

A hackier alternative is to simply allow only the root node to be infinite. Instead of configuring it as a truly infinitely-repeating node, however, it should be wired up to run exactly once. Then, the grammar simply invokes the root node some random number of times in order to generate a "window" into the current sequence. The resulting output is buffered and can be consumed at any arbitrary rate. Conceptually, the sequence behaves as if it were infinitely long, but the process of generating new subsequences is easily accomplished in finite time.

It should be pointed out that grammars are hardly the only tool for generating such infinite sequences. In fact, if the characteristics of the sequence are *context sensitive*, that is, the upcoming output depends on the value of previous output; an approach like *n*-grams is probably much more useful (Vasquez 2014).

36.6 Grammars as Analogues to Behavior Trees

When considering the design and application of a stochastic grammar, it can be helpful to think of them as limited behavior trees (Isla 2005). As seen earlier, a grammar can often be represented as a tree (although a directed graph is needed in the case where rules form cycles). Each rule in the tree can be thought of as a node in a behavior tree, by loose analogy.

Sequences and alternations map directly to sequence and selection nodes in BTs. The primary distinction is that, for a stochastic grammar, the logic for choosing how to proceed is not based on examining game state, but simply rolling a random number. So a stochastic grammar provides a tool for mimicking more complex AI decisions using a weighted random behavioral pattern rather than something more intentional.

The process for designing a stochastic grammar can closely parallel the process of designing a simple behavior tree. Clearly, the stochastic grammar will make far less deliberate and reactive actions in general, but with careful weight tuning, a fuzzy behavior model can look remarkably similar to a more intentional model.

Choosing to use a grammar over a BT is primarily a matter of considering two factors. First, if the behavior tree is designed to carefully handle contingencies or special cases of world state, it is probably not suitable for replacement with a grammar. Second, the design of the agent being controlled may lend itself to a more randomized behavioral pattern, in which case grammars are an excellent choice. A simple way to recognize this is to check for heavy use of random selector nodes in the behavior tree.

One last potential benefit of the grammar model is that it is cheap to evaluate, since it does not require many world state queries and can operate purely on a stream of random numbers. This makes grammars an excellent tool for simulating large numbers of agents with low-fidelity "intelligence."

36.7 Grammars as Scripting Engine

One of the more powerful ways of looking at grammars is as a tool for generating tiny scripts. If the generated symbols are miniature commands for a scripting engine, grammars define the rules by which those commands can be combined into meaningful programs. Working with this analogy, the goal is to first specify a set of commands that are useful for handling a given AI problem, and then specify a grammar that will produce effective sequences of those commands.

The advantage of using a grammar to generate such script programs is that the scripts themselves need not be static. New scripts can be created on-the-fly as gameplay unfolds. Since the rules for creating a script are defined during the game's implementation, any generated script has a reasonable chance of doing the "right thing"—assuming the rules are suitably constrained.

One way of looking at this is that grammars are a key part of dynamically reprogramming a game's behavior as it is played. As long as a given grammar is well-designed, it will produce new behavioral scripts that are effective within the game simulation itself. Controlling the weights of rules in the grammar yields the power to adjust the "success rate" of any given script on-the-fly. Clearly, the possibilities for this are endless.

Generally speaking, any time an AI system (or other game system!) expresses behavior in terms of sequences of actions, a grammar can be deployed in place of handcrafted scripts. Moreover, given the relationship between grammars and *deterministic finite automatons*, it is possible for any behavior generated by a finite-state machine to also be expressed by a grammar (Zhang and Qian 2013).

There is clearly ample material in a typical game's AI—and, again, other game systems—that could be supplanted by the crafty use of grammars. Grammar-like methods known as *Lindenmayer-systems* (or *L-systems*) are already in popular use for procedural generation of certain kinds of geometry, ranging from trees and rivers to buildings and even entire cities (Rozenberg and Salomaa 1992). Some creative users of L-systems have even explored creating gameplay mechanics based on the technique (Fornander 2013).

36.8 Tuning a Stochastic Grammar

One of the classic methods for computing appropriate weights for a stochastic grammar given a preexisting corpus of sequences is the *inside-outside algorithm* (Baker 1979). This

approach computes probabilities of various strings appearing in a given grammar, starting from an initial estimate of each probability. It can then be iteratively applied until a training string's probability reaches some desired point. Indeed, if a corpus of training data is available, this method is the definitive starting point for tuning a grammar.

But what if training data is not available? The primary difficulty of tuning the grammar then becomes generating enough sample outputs to know whether or not the overall distribution of outputs is desirable. Strictly speaking, most stochastic grammar approaches use a *normalized probability* of each rule being selected in the output generation process. This is mathematically elegant but can make it difficult to estimate the overall traits of the grammar, since the human mind is notoriously bad at probabilistic reasoning.

As a compromise, the accompanying demo code does not adhere to a strictly normalized probability model for all of the rules. Some shortcuts have been taken to simplify the tuning process. Namely, *subsequences* have a probability of repeating, which is independently applied after each successful generation of that subsequence. If the random roll fails, the subsequence ends. Further, *alternations* (selections from among several options) employ a simple weighted random scheme to allow the grammar creator to control the relative "importance" of each option.

Although not strictly compliant with the preexisting work on the subject of stochastic grammars, this approach is arguably far simpler to reason about intuitively. More importantly, the tuning process is as simple as generating a large number of outputs, and hand-editing the weights and probabilities of various elements of the grammar to suit.

On an opinionated note, the transparency of the modified stochastic grammar concept is tremendously important. Although probabilistic grammars are typically viewed as a machine learning technique, they need not provoke the negative reaction to machine learning that is so common in game AI circles—because they do not inherently carry the need to give up fine-grained control and intuitive results. Compared with other approaches, the lack of explicit training can actually be a huge boon, since it eschews the "black box" nature of many other learning tools. Designers can rest assured that the grammar will produce *comprehensible* if not directly predictable results.

Moreover, it is trivial to dynamically exploit the direct relationship between weights in a stochastic grammar and the frequency of output patterns. If a grammar produces too many occurrences of some subsequence, the weight for that sequence can simply be decreased at runtime. Of course, the tricky part here is attaching sufficient metadata to the final sequence such that the rules responsible for a particular excessive subsequence can be identified easily. This flexibility (and transparency) is far more cumbersome to build into a system like an artificial neural network or a genetic algorithm.

36.9 Feeding a Grammar with Utility Theory

Another approach to generating weights for a stochastic grammar is to measure them using *utility theory* (Graham 2014). In this technique, the weight of a given node is computed through a scoring mechanism that evaluates how "useful" that node is in a given context. For instance, suppose a turn-based strategy AI has three basic options: attack, reinforce defenses, or expand to new territory. This AI can be given a stochastic grammar for deciding its moves for the next several turns.

Table 36.4 Stochastic Grammar Decides How a Strategic AI Plays the Game

Name of Rule	Rule Generates these Symbols
Smart Turn	0.4 Attack|0.3 Reinforce|0.3 Expand;
Offensive	Attack, Attack, {Smart Turn};
Turtling	{Reinforce}, {Smart Turn};
Conquest	Attack, Expand, {Smart Turn}, Expand;

When moves are needed, the AI recalibrates the weights of each option based on evaluating the current battlefield. Depending on the grammar it uses, the AI can express various "personality" differences. Consider the example grammar in Table 36.4.

In this model, the AI has three basic personalities to choose from. The Offensive personality will attack two times followed by a series of "smart" choices based on utility. AIs that prefer to "turtle" will reinforce their defenses for an arbitrary period, then make a few "smart" moves. Lastly, expansionistic AIs will attack and expand heavily, sprinkling in a few "smart" turns as well.

The default calibration for "smart" moves has Attack turns slightly preferred to Reinforce and Expand selections—but imagine if the AI could calculate *new* weights for these options on-the-fly. If the utility score for a particular move is exceptionally high, that strategy will dominate the AI's play for several turns. Conversely, if the utility score is low, the AI is less likely to favor that selection.

Ultimately, the result is that AIs will tend to play according to a particular style, but also mix things up periodically with sensible moves based on situational reasoning. A moderate style could even be used which simply does the "smart" thing all the time. More sophisticated play styles can be constructed with almost arbitrary power and flexibility, just by expanding the grammar.

36.10 Conclusion

Stochastic grammars are a widely used tool from natural language processing. They have seen limited use outside that field, despite being applicable to a number of interesting problems, when applied creatively.

By generating sequences of data in a controlled—but still random—fashion, stochastic grammars enable the creation of highly structured—but not perfectly predictable—outputs. Such outputs can be suitable for many game AI and game logic tasks, ranging from design-time procedural content creation to actual on-the-fly behavior controllers.

Although slightly unorthodox in the realm of game AI, grammars offer a much higher degree of designer control than many other machine learning techniques. As such, they are a promising tool for the inclusion in every game AI professional's toolbox.

For those interested in further research, the author highly recommends (Collins).

References

Backus, J. W. 1959. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. *Proceedings of the International Conference on Information Processing, UNESCO.* 125–132.

Baker, J. K. 1979. Trainable grammars for speech recognition. *Proceedings of the Spring Conference of the Acoustical Society of America.* 547–550.

Brooker, R.A.; MacCallum, I. R.; Morris, D.; Rohl, J. S. 1963. The compiler-compiler. *Annual Review in Automatic Programming* 3:229–275.

Collins, M. Probabilistic Context-Free Grammars http://www.cs.columbia.edu/~mcollins/courses/nlp2011/notes/pcfgs.pdf (accessed July 10, 2016).

Fornander, Per. 2013. *Game Mechanics Integrated with a Lindenmayer System.* Bachelor's Thesis, Blekinge Institute of Technology. http://www.diva-portal.se/smash/get/diva2:832913/FULLTEXT01.pdf (accessed July 10, 2016).

Graham, Rez. 2014. An introduction to utility theory. In *Game AI Pro,* ed. S. Rabin. Boca Raton, FL: CRC Press, pp. 113–126.

Hunter, Sir William Wilson. 1881. *Imperial Gazetteer of India*. Oxford: Clarendon Press.

Isla, Damian. 2005. *Managing Complexity in the Halo 2 AI System.* Lecture, Game Developers Conference 2005. http://gdcvault.com/play/1020270/Managing-Complexity-in-the-Halo (accessed July 10, 2016).

Rabin, Steve; Goldblatt, Jay; and Silva, Fernando. 2014. Advanced Randomness Techniques for Game AI: Gaussian Randomness, Filtered Randomness, and Perlin Noise. In *Game AI Pro*, ed. S. Rabin. Boca Raton, FL: CRC Press, pp. 29–43.

Rozenberg, Grzegorz; Salomaa, A, eds. 1992. *Lindenmayer Systems: Impacts on Theoretical Computer Science, Computer Graphics, and Developmental Biology.* Verlag/Berlin/Heidelberg: Springer.

Vasquez, Joseph II. 2014. Implementing N-Grams for player prediction, procedural generation, and stylized AI. In *Game AI Pro*, ed. S. Rabin. Boca Raton, FL: CRC Press, pp. 567–580.

Wirth, Niklaus. 1977. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, Vol. 20, Issue 11. 822–823.

Zhang, Jielan; and Qian, Zhongsheng. 2013. The equivalent conversion between regular grammar and finite automata. *Journal of Software Engineering and Applications*, 6:33–37.