A Web-Based Editor for Multiplayer Choice Games

Ian Holmes
Department of Bioengineering
University of California
California, Berkeley
ihh@berkeley.edu

ABSTRACT

The structure of a choice-driven interactive story can be modeled as a syntax tree, generated by a player from a grammar, which in turn was generated by a game author. Using this grammar metaphor, LetterWriter, a web-based Integrated Development Environment was developed for writing, testing, playing and sharing choice-based interactive fiction. The system includes a domain-specific language, a map overview, a parse-tree debugger, and a network client for multiplayer games.

1. INTRODUCTION

Choice-based interactive fiction came of age in the 1980's: Choose Your Own Adventure[1], Fighting Fantasy[2], and Lone Wolf are well-known imprints. The genre has seen an early 21st-century resurgence with the advent of writing tools—from domain-specific languages like ChoiceScript[3], to interactive editors like Twine[4]—which have lowered the barriers to game creation and increased the diversity of authorial voices.

A simple choice-based story can be compared to a flowchart, or a state machine, or (in the language of grammar theory) a *regular grammar*. The main control-flow construct is the optional `GOTO` statement ("*IF you choose to take the chalice, TURN TO page 400*"). The structure of the story is a linear progression of scenes: the game program is a finite state machine that outputs scenes in a player-driven sequence.

In modeling many kinds of text, it is often useful to allow a richer structure. For example, conversations often make diversions (sometimes nested diversions (and sometimes multiply nested)) before returning to the main theme. Epic adventures can include side-quests; stories often have epilogues.

This sort of structure can be modeled using a *context-free grammar* (CFG). If a typical choice-based story (with a regular grammar) is akin to a state machine, whose main control-flow construct is `GOTO`, then a CFG-based story is like a pushdown automaton (a finite-state machine with a stack) which allows not only `GOTO` but also `GOSUB`.

To extend choice-based stories in this way might indicate an excessive fondness for formalism. However, a well-defined framework provides a robust foundation.

In this abstract we describe LetterWriter, a software system that allows authoring of story CFGs via an interface inspired by graphical editors like Twine and domain-specific languages like ChoiceScript. The created games can be played over the web by multiple players in different locations.

One typical advantage of a formal framework is a rich variety of links to other areas of culture. Grammars have variously served as metaphysical systems[5, 6], tools of social unification and control[7, 8], and models of natural language[9]. They have been used in compiler theory[10], DNA sequence analysis[9], and computer graphics[11]. There is a literature on game-theoretic analysis of grammar-based games [12]. Templates and scripts for social interactions have a rich history in popular social psychology[**?**, **?**]. All of the above influences offer a source of inspiration for single- and multiplayer game design.

2. DESIGN

2.1 Game language

The application is based around a declarative, domain-specific language for specifying an attributed CFG.

The core element in this grammar, syntactically and semantically, is the *transformation rule*.

```
@scene => { hint => expansion }
```

In this, `@scene` is a label denoting the current scene; `hint` is text presented to the player as a choice; and `expansion` is text that will be generated if the player makes this choice (which can include further scene labels).

A block of transformation rules represents a list of choices:

```
@scene => { hint1 => expansion1
          | hint2 => expansion2
          | hint3 => expansion3 }
```

Any number of (`hint`,`expansion`) tuples are allowed. Whitespace can be ignored (although it is preserved in rendered text), and the `hint=>` can be omitted as well. There is also an optional longer form

```
@scene => [ preamble | placeholder | prompt ]
          { hint1 => expansion1
          | hint2 => expansion2
          | hint3 => expansion3 }
```

Here, `preamble` is text describing the scene; `placeholder` is text temporarily displayed at the bottom of the scene (before the list of choices), and deleted when an expansion is selected; and `prompt` is text, also transiently displayed, that is associated with the list of choices. (Typically the prompt takes the form of a question addressed directly to the player.)

As with other parts of the syntax beyond the core idea of a context-free transformation rule, the preamble and/or placeholder can be omitted.

Here is an example:

```
 @unrest =>
[ The peasants are revolting.
  | _So tedious..._
  | What will you do? ]
{ Feed them.
   => You cast grain from the battlements.
      @peace
| Play some music.
   => No lute can drown their hungry groans.
      @revolt }
```

The words "*So tedious...*" will disappear when a selection is made; being wrapped in underscores, they will also appear in italics, as a result of the Markdown-like macro transformation. The prompt "What will you do?" will also disappear when a selection made. (In multiplayer mode, the prompt is only shown to the active player.)

In the terminology of formal grammar theory: the scene label `scene` is a *nonterminal symbol*, whereas other rendered text is composed of *terminal symbols* (ASCII characters). Nonterminal symbols denote points in the text (scenes) where further expansion is possible; terminal symbols denote static endpoints of the text. The preamble, placeholder, prompt and hint are *attributes* added to make the grammar formalism a little more friendly to writing interactive stories.

All nonterminal symbols begin with `@`. Special characters such as `$@[]{}` can be escaped with a backslash if required in the text. Quotation marks and other common punctuation are not special characters and can be used directly, as can HTML tags. Some syntactic sugar for HTML tags (e.g. flanking underscores for italics) is borrowed from Markdown [13].

Starting from an initial designated nonterminal (named `@start` by default), iterated application of these rules generates a

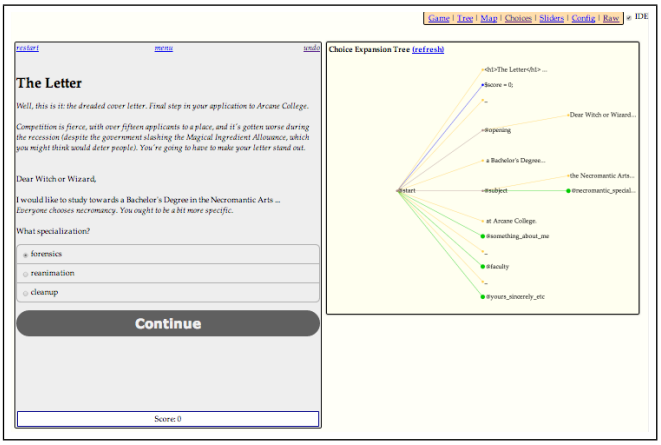

Figure 1: The parse tree visualization (right) helps debug gameplay (left).

parse tree (Figure 1, right), with nonterminal symbols at internal nodes and terminal symbols at the tips. The final text can be read off from the terminals at the leaf nodes (Figure 1, left).

The sequence of terminals is post-processed before being rendered, so the terminals themselves do not necessarily correspond exactly to what is shown on screen. Specifically, the terminal nodes at the leaves of the parse tree represent a *program text*, which can contain embedded commands (such as variable assignments, inputs, modification, and interpolation) that are then interpreted to (deterministically) compute the *story text* presented to the player. As noted above, a Markdown-like macro expansion is further applied to the generated story text, for quick HTML styling [13].

Extra keywords are available; for example, to specify that some variables are directly controlled by the player, (using sliders), or to delineate a variable as the score. Other keywords can specify the game title, or control the behavior of the "undo" button.

Not all transformation rules need be presented to the player at every opportunity. The author can flag individual rules to be hidden from view some fixed number of times before first being shown, or limited to some finite number of uses. Various modifier keywords can be associated with nonterminals to indicate this in the game source file.

More computationally expressive logic for controlling story flow is available via the use of computer players. For nonterminals that are flagged as belonging to the computer player, rules are normally played at random, but this behavior can be manipulated by the author to assign different probabilistic weights to the rules (and these weights can also be boolean logic expressions, allowing for more sophisticated control of flow). The player can also "steer" the computer player by manipulating probabilistic weight parameters directly, by means of slider controls in the HTML page.

2.2 Editor

At the most basic level, the editor includes a textbox for direct editing of game source code. Beyond this, a number

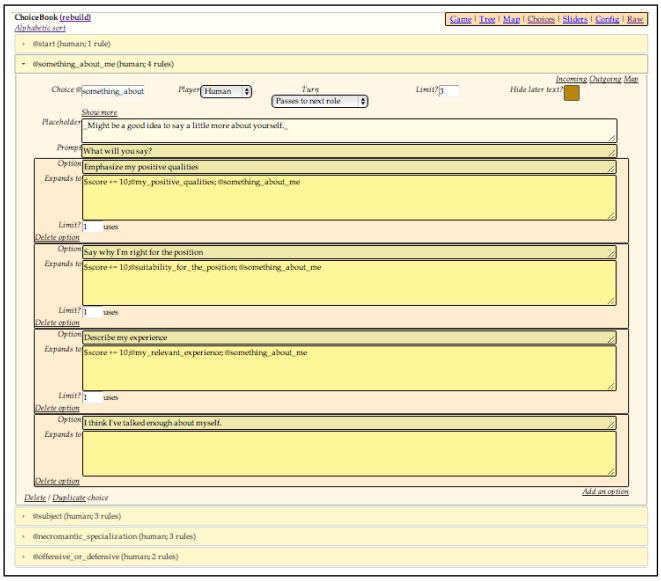

Figure 2: The nonterminal pane allows drag-and-drop ordering and editing of menus.

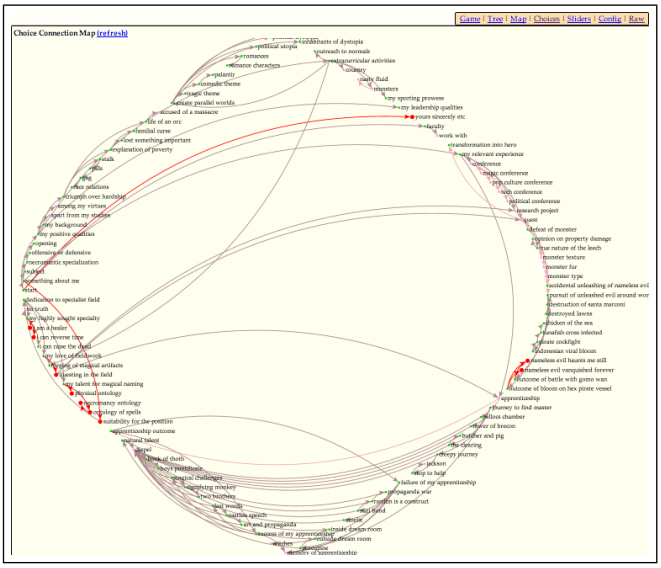

Figure 3: The map provides an overview of connections between nonterminals.

of User Interface (UI) elements are presented together as a basic Integrated Development Environment (IDE).

The main element is a drag-and-drop sortable list of nonterminal editing panes (Figure 2). Each nonterminal pane contains a drag-and-drop-sortable list of transformation rules, together with UI elements for modifying rule behavior (e.g. the number of times a rule can be used by, or should be hidden from, the player). An overview summarizes orphan nonterminals (never generated by any rules), bare nonterminals (no text), and empty rules (loose ends). Hyperlinks are provided for quick navigation to incoming/outgoing nonterminals, and also to the map view.

Another drag-and-drop sortable list specifies the parameters that the player can set directly via slider controls. A few properties of the grammar (such as its title) can be directly edited.

2.3 Map

The map, rendered using a third-party graph visualization library, shows the overall structure of the game (Figure 3). Nodes represent nonterminals (arranged in a circular layout). Edges $a \to b$ imply the existence of at least one rule $a \to \ldots b \ldots$ with a on the left-hand side and b on the right.

Nodes are colored to represent some useful information (e.g. whether the corresponding nonterminal is human- or player-controlled, whether it is a loose end, and so forth). The author can mouse-over a node to highlight incoming & outgoing edges.

2.4 Parse tree debugger

The parse tree is visualized using the same graph library as the map (Figure 1, right). The various types and status of nodes (terminals; expanded & unexpanded nonterminals (player- and computer-controlled); parameter references, assignments & inputs) is indicated via size and coloring. The author can mouseover nodes for more info (e.g. the order and time of expansion). Clicking on a node navigates to the corresponding nonterminal pane in the editor.

2.5 Game client

The parse tree is rendered as text after performing variable substitutions (Figure 1, left). Minimal styling is used to present choice lists and animate key events, e.g. fading-in of newly-rendered text.

An "undo" button with gradually-increasing recharge time is offered as an optional game mechanic for solo play. (The complications of implementing "undo" in multiplayer play were eschewed; we observe in any case that "undo" is rarely a participation-enhancing mechanic for choose-your-own games.)

2.6 Multiplayer operation

Multiplayer mode is enabled by increasing the number of *roles* in the grammar from its default value of 1. The number of roles is the same as the number of (human) players. Each nonterminal node in the parse tree is controlled (i.e. can only be expanded) by the player in one specific role. This prevents race conditions by design: each choice is uniquely controlled by one player. The `@start` nonterminal at the root node is always controlled by the player in the first role. By default, if node X is controlled by role n and there are k roles, then all children of node X are controlled by role $((n+1) \mod k)$. Thus, control passes predictably from one player to the next, although this default behavior can be modified by the author (e.g. to indicate that a particular nonterminal should always be controlled by a given role). For every role, there is one human player (for manual choices) and one computer player (for automatic choices).

Multiplayer mode is implemented using a third-party publish-subscribe (pub-sub) framework. Games are organized on an invitation channel and then take place on an hierarchically-structured play channel, with one channel for each node

of the parse tree. Discreet animations convey (a) successful publication of rules to the pub-sub server (for locally-controlled node expansions), (b) successful subscription of a listener at a particular point in the parse tree (for remotely-controlled nodes).

3. DISCUSSION

Grammars are found throughout computer science, and there are many potential applications of an integrated system for designing grammars and then using them collaboratively to generate texts over a network. Such applications range from serious uses in IT enterprise (e.g. structured chat-rooms for product support), through traditional game tropes (e.g. dungeonmaster-player conversations), through new electronic models of social interaction (e.g. scripted interactive dates).

Below, we discuss some possible extensions.

3.1 Cryptographic signatures

Cryptographically-authenticated play would obviate the need for a server or central scoring authority. A cryptographic extension of the basic pub-sub model should be straightforward. Crucial messages must be signed (and counter-signed when received): these include invitations, applications to join the game, and rule expansions.

3.2 Strategic optimality

Strategically optimal algorithms for playing this kind of game are known to exist when the scoring scheme is a trivial function of the parse tree (e.g. fixed rewards for using certain nonterminals [12]). However, the scoring scheme described here is considerably more flexible, modeling many aspects of context-sensitive grammars as well as CFGs.

The programming language for the computer player does not attempt to model AI in any deep sense: it is very simple, just offering variables, conditional tests, and the in-built facility for looping and recursion that comes for free with the CFG.

A possible extension is to use an ambiguous grammar (multiple parse trees consistent with observed output) and introduce a computer player that can parse the current output (e.g. using Earley-Stolcke parsing [14]) and predict future outcomes probabilistically.

4. IMPLEMENTATION

LetterWriter was implemented in JavaScript using SigmaJS, PegJS, JQuery, OpenPGP, Node, and Faye, with CSS from ChoiceScript. The software has been tested in Google Chrome and Mozilla Firefox.

An early choice-based engine prototype (*Schooz*) was developed in Scheme. A prototype parser-driven AI (*Gertie*) was developed in Perl.

4.1 Availability

Code is freely available at `https://github.com/ihh/funkscene`

4.2 Acknowledgements

Many thanks are due Dan Fabulich, Richard Evans, Michael Mateas, Emily Short, Graham Nelson, Jon Ingold, and Rudy Rucker for help and inspiration.

5. REFERENCES

[1] E. Packard. *The Cave of Time*. Choose Your Own Adventure. Random House, 1982.

[2] S. Jackson, I. Livingstone, and R. Nicholson. *The Warlock of Firetop Mountain*. Fighting fantasy gamebooks. Puffin Books, 1982.

[3] D. Fabulich. *Introduction to ChoiceScript*. `http://www.choiceofgames.com/`.

[4] C. Klimas. *Twine: an Open-Source Tool for Telling Interactive, Nonlinear Stories*. `http://twinery.org/`.

[5] Pāṇini. *Ashtadhyayi*. `http://en.wikipedia.org/wiki/Ashtadhyayi`.

[6] A. Luhtala. *Grammar and Philosophy in Late Antiquity: A Study of Priscian's Sources*. Amsterdam Studies in the Theory and History of Linguistic Science. John Benjamins Pub., 2005.

[7] Académie Française. *L'histoire*, 2013. `http://www.academie-francaise.fr/linstitution/lhistoire`.

[8] R. Lowth. *A Short Introduction to English Grammar: With Critical Notes*. J.J. Tourneisin, 1794.

[9] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge, UK, 1998.

[10] A.V. Aho, M.S. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Educational Publishers, Incorporated, 2007.

[11] G. Rozenberg and A. Salomaa. *The Mathematical Theory of L Systems*. Pure and Applied Mathematics. Elsevier Science, 1980.

[12] K. Etessami, D. Wojtczak, and M. Yannakakis. Recursive stochastic games with positive rewards. In L. Aceto, I. Damgård, L.A. Goldberg, M.M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *ICALP (1)*, volume 5125 of *Lecture Notes in Computer Science*, pages 711–723. Springer, 2008.

[13] J. Gruber. *Markdown*. `http://daringfireball.net/projects/markdown/`.

[14] A. Stolcke. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Comput. Linguist.*, 21(2):165–201, June 1995.