Expressionist: An Authoring Tool for In-Game Text Generation

James Ryan, Ethan Seither, Michael Mateas, and Noah Wardrip-Fruin

Expressive Intelligence Studio
University of California, Santa Cruz
{jor, michaelm, nwf}@soe.ucsc.edu, eseither@ucsc.edu

Abstract. We present Expressionist, an authoring tool for in-game text generation that combines the raw generative power of context-free grammars (CFGs) with the expressive power of free-text markup. Specifically, authors use the tool to define CFGs whose nonterminal symbols may be annotated using arbitrary author-defined tagsets. Any content generated by the CFG comes packaged with explicit metadata in the form of the markup attributed to all the symbols that were expanded to produce the content. Expressionist has already been utilized in two released games and it is currently being used in two ongoing projects. In this paper, we describe the tool and discuss these usage examples in a series of case studies. Expressionist is planned for release in late 2016.

Keywords: text generation · authoring tools · modular content · grammars

1 Introduction

Why have so few completed works of interactive storytelling featured generative text? The dearth of examples is troubling, and especially so given the widely held belief that current authoring practice in interactive storytelling—in which individuals or teams of writers tirelessly produce huge amounts of static content by hand—is limiting the medium [13]. Elsewhere we have argued that previous approaches to this challenge that have employed conventional techniques from *natural language generation* (NLG), though important, have suggested a disturbing reality: full NLG pipelines are not yet ready for use in fully realized works of interactive storytelling [22]. A major reason for this is that NLG pipelines demand NLG expertise, but content authoring in interactive storytelling requires other kinds of expressive expertise—some may have command of all these skills, but NLG practitioners should not be the sole purveyors of generative text.

At the Expressive Intelligence Studio, we are working to democratize the practice of procedural text generation through the development of authoring tools that are intended for use by non-NLG practitioners. A particular such tool is Expressionist, the subject of this paper, which is intended for *in-game* expressive text generation.[1] By this, we mean the generation of text at runtime that

[1] We use the term 'game' here for brevity and convenience, but we more generally mean any work of interactive storytelling or playable media.

depends on the current game state and satisfies authorial goals. This could be in support of procedural dialogue interaction [11] or any other application involving content that is textual in composition (*e.g.*, storyworld artifacts). In designing and developing Expressionist, our core aim has been to produce an authoring environment that maximally utilizes two complementary strengths of humans and computers—humans' deep knowledge of natural-language expressivity (and all its attendant nuances), and a computer's capacity to efficiently operate over probabilities and large treelike control structures—while simultaneously minimizing both entities' huge deficiencies in the converse. In other words, we have strived to produce a configuration that holds humans responsible for things they are good at and not for things they are bad at, and likewise with computers.

Toward these aims, Expressionist combines the raw generative power of *context-free grammars* (CFGs), described in Section 2, with the expressive power of *free-text markup*. Specifically, authors specify CFGs whose symbols may be annotated using arbitrary author-defined tags, and all generated outputs elegantly accumulate all the markup attributed to the symbols that were expanded to produce them. The validity of this approach is demonstrated by the use of Expressionist in two released games, *Snapshot* [26] and *Project Perfect Citizen* [1], as well as two ongoing projects, all of which we discuss in Section 4. While the Expressionist authoring interface is currently being refined, we plan to publicly release the tool in late 2016.

2 Background and Related Work

Though preceded by related concepts in mathematics by Post and Thue, and in linguistics by the ancient Pāṇini, the *context-free grammar* (CFG) formalism was introduced by Noam Chomsky in his seminal work of the mid-1950s (see [2] for this history). CFGs are foundational in theories of natural and computer languages, and the formalism, in the form of *story grammars*, has a colorful history in the field of computational narrative (which is nicely recounted in Section 3.2 of [6]). There are three simple notions that are central to the CFG formalism: terminal symbols, nonterminal symbols, and production rules. A *terminal symbol* is simply a string, and a *nonterminal symbol* (or just *nonterminal*) is a symbol that is associated with one or more *production rules* by which the symbol can be *expanded* into a concatenation of other symbols, which may be either terminal or nonterminal (the latter of which will have their own production rules).

Expressionist's design is meant to instantiate the *modular content* approach put forward in [23], a notion that itself emerged from an earlier project in which we procedurally recombined existing *Prom Week* dialogue by annotating individual lines for how they could be resequenced and what they could express about the underlying system state [21]. While we appropriated an existing annotation interface for this task, the project led us to envision an environment for specifying recombinable content that would be annotated *at the time of authoring*. This line of thinking eventually culminated in the design of Expressionist.

The particular utilization of *context-free grammars* (described in the next section) as Expressionist's resequencing method was influenced by a related tool

called Tracery [5], which also utilizes them. Expressionist differs from Tracery in both its design goals and authoring features. Tracery is aimed at naive procedural authors who wish to generate standalone text by lightweight means—this is demonstrated by the success it has found in fueling generative Twitter bots [5], a task for which Expressionist would be overkill (since metadata is not needed when any generable output will suffice). Expressionist, on the other hand, is intended as a way of supporting the generation of content that satisfies *targeted requests* made by an external application, like a game. This difference manifests in terms of authoring features, where Expressionist diverges from Tracery by affording free-text markup. This allows generated content to accumulate associated metadata, which is necessary for satisfying targeted content requests and for coherently sequencing multiple units of generated content. For example, metadata is used in *Talk of the Town*, discussed below, to specify how generated lines of dialogue may precede and succeed one another in procedural conversations [20]. Using Tracery, such metadata would have to be injected into the symbol names themselves, which would quickly grow unwieldy.

In [19], we review the literature on text generation in games, so we will only summarize here. Early exploration in this young area utilized domain-specific languages [12] and full NLG pipelines [4, 17], with the latter approach notably being employed in the *SpyFeet* prototype [16] and the commercially released *Bot Colony* [10]. As we articulate at length in [19], our approach is a considered reaction to full NLG pipelines that is intended to be more approachable and provide greater authorial control. In the interactive-fiction community, a number of bespoke methods have been employed [24], including *templated dialogue*, which is also used in projects like Curveship [15], Versu [7], and the LabLabLab trilogy [11]. By employing CFGs, Expressionist harnesses the superset generative power of template-based approaches, since a CFG defines a system of arbitrarily nested templated structures. By virtue of its markup affordance, Expressionist also provides more authorial control, for the reasons described above. In addition to Tracery and Expressionist, two other recent systems, *MKULTRA* [9] and Dunyazad [14], have employed grammar-based approaches. Instead of CFGs, these systems use variants of a definite-clause grammar. This allows for constraints (such as variable bindings) to flow downstream during the generation process, which in Expressionist can be handled by markup. A primary difference between our system and these is that Expressionist has a graphical authoring interface. Finally, SimpleNLG is another tool aimed at democratizing text generation [8], but one that is dedicated to facilitating realization details, like agreement and conjugation, rather than supporting broader expressive aims.

3 Expressionist

Expressionist is a tool for authoring CFGs whose generated content comes packaged with explicit metadata; it is designed to support live text generation in games and other interactive media. While its authoring interface is still being refined in terms of aesthetics and usability, Expressionist has already been used in multiple projects, including two completed games, as we discuss in Section 4. In this section, we describe the tool in detail.

3.1 Overview

Using Expressionist, an author specifies a CFG by defining its nonterminal symbols and the production rules that may work to produce terminal expansions. Crucially, nonterminal symbols may be annotated using arbitrary *tagsets* and *tags*, which are defined by the author; this is the core appeal of Expressionist, and the source of its expressive power. When a terminal expansion is produced in a CFG, it will have expanded a set of nonterminal symbols along the way. In Expressionist, such an expansion accumulates all of the markup that an author has attributed to the symbols in this set. This allows an author to modularly specify capsules that contain both symbolic markup (*e.g.*, a speech act, a character personality trait, a specific authorial goal) and the rules for producing variations of the linguistic expression of that markup. Appealingly, because it is inherited from expanded symbols during the generation process, markup does not have to be reduplicated at the level of terminal symbols. In addition to symbol annotation, Expressionist allows production rules to be assigned probabilities of being executed, which more specifically makes the structure authored using the tool a *probabilistic* CFG (which is another difference with Tracery).

3.2 Workflow

Expressionist is implemented as a web application that imports and exports authored CFGs, which are defined as JSON files. To begin an authoring session with the tool, the user navigates to the URL where it is hosted and either uploads an existing grammar or elects to start a new one. From here, the authoring interface, described in the next section, appears. Authoring takes the form of defining nonterminal symbols, production rules (and associated probabilities), tagsets, and tags, and attributing tags to nonterminal symbols. Additionally, the author may utilize a built-in expansion engine to generate terminal expansions of any nonterminal symbol (or the terminal results of executing any production rule) as a way of checking the quality of generable outputs. This *live feedback* feature critically utilizes the probabilities assigned to production rules and accumulates all markup attributed to nonterminal symbols. Finally, to conclude an authoring session, the user exports her work as an updated JSON file. To generate text from this CFG in-game, the author must first implement an expansion engine in the game code, as we discuss below in Section 3.4.

3.3 Authoring Interface

An annotated screenshot of the current version of the Expressionist authoring interface is shown in Figure 1. Briefly, we will describe each numbered element:

1. The *active nonterminal symbol* is the one that the author is currently working on. The asterisk icon on the left may be selected to *favorite* the symbol.
2. All of the authored nonterminal symbols in a CFG appear in a scrolling menu on the right. Symbols for which no production rules have been defined appear in red, while expandable ones are displayed in green. Within the list of expandable symbols, favorited symbols are sorted to the top. The user may click on any symbol to make it the active nonterminal symbol. Additionally, new symbols may be defined by clicking an icon at the bottom of the list.

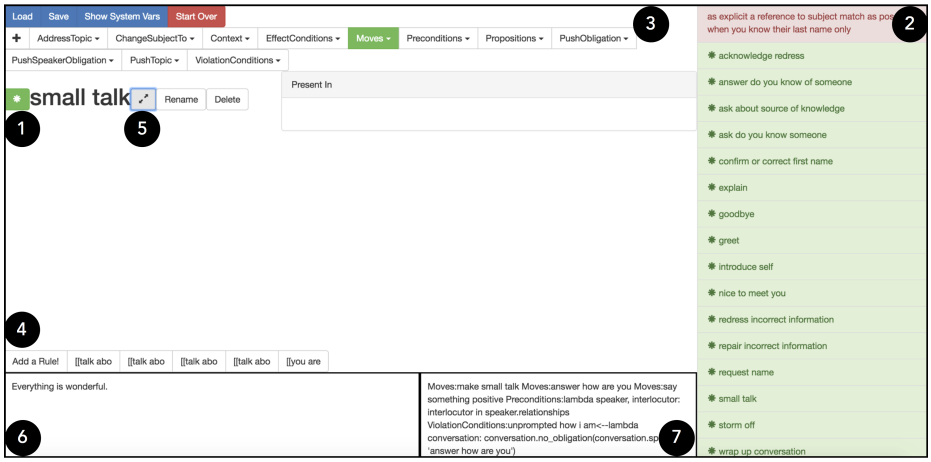

Fig. 1. An annotated screenshot of the current version of the Expressionist authoring interface. See Section 3.3 for descriptions of each of the numbered elements.

3. Each of these dropdown menus represents an individual tagset that the author has defined. When a tagset is clicked on, the menu expands to display the individual tags that the user has defined for the set. The user may then attribute one or more of these tags to the active nonterminal symbol by clicking on them. New tagsets can be defined at any time by clicking on the plus sign at the far left.
4. New production rules for the active nonterminal symbol can be defined at any time, and the user can explore existing rules by clicking on them. When a rule is selected, it becomes the *active production rule*, allowing it be edited, refined, or targeted for live feedback. Production rules may recursively reference other nonterminal symbols using a simple '[[symbol name]]' syntax.
5. At any time, the user can click this expansion icon to request live feedback in the form of terminal expansions (generated text, 6, and the markup accumulated during the production of that text, 7) of the active nonterminal symbol or the terminal results of executing the active production rule.

3.4 Generating In-Game Text

While the Expressionist authoring interface has its own expansion engine, it is only intended for live feedback at authoring time (and does not reason about markup during expansion, except to accumulate it). To generate in-game text at runtime, an expansion engine must be implemented as a module in the game code. The purpose of this module will specifically be to operate over an Expressionist JSON at runtime to generate text on the fly according to requests made by the game's content-selection architecture. In Expressionist parlance, we call such a module a *Productionist*. While each implementation will differ according to the expressive aims of a larger application, there is a general pattern that all

Fig. 2. A *Snapshot* player takes a photograph of a deer and posts it to her photoblog, right, where an evaluative comment generated using Expressionist appears.

Productionist modules follow: a request comes in for generated text with *specific desired markup* (*e.g.*, a speech act), and Productionist operates over an Expressionist JSON to produce a terminal expansion that has that markup. As such, a core part of utilizing Expressionist in a game is making good use of grammar markup by operationalizing it with application-specific semantics, which may guide content generation and drive updates to the game state upon content deployment. In the next section, we discuss a series of Productionist modules that have been implemented in both completed games and prototypes.

4 Case Studies

Expressionist has been used to generate text in two released games, *Snapshot* [26] and *Project Perfect Citizen* [1], and is also being used in two projects that are currently in development. While eventually we plan to conduct user studies to evaluate the usability of its authoring interface, we believe the successful utilization of Expressionist in these works validates it as a tool for in-game text generation. In this section, we will describe this diverse array of usage examples in a series of short case studies.

4.1 Generating Evaluative Photoblog Comments in *Snapshot*

Snapshot is a first-person photography simulator set in a nature reserve that is modeled using a compelling low-poly art style [26]; it is written in C# using the Unity framework. In the game, the player explores the reserve across day and night cycles to take photographs of its wildlife and other natural elements. Between photo sessions in the park, the player can post her favorite shots to an in-game photoblog and view evaluative visitor comments that are generated using Expressionist (as shown in Figure 2).

Generated photoblog comments in *Snapshot* are surface expressions of the core technology underpinning the game: an automated *photo evaluation system*. This system evaluates player photos according to three heuristics: *balance* (how well visual intrigue is distributed across the photo), *spacing* (the rule of thirds), and *subject quality* (the interestingness of the subject of a photo, *e.g.*, a deer or mountain). Whenever a player posts a photo to her blog, the evaluation system scores it using its three heuristics, isolates the heuristic the photo scored

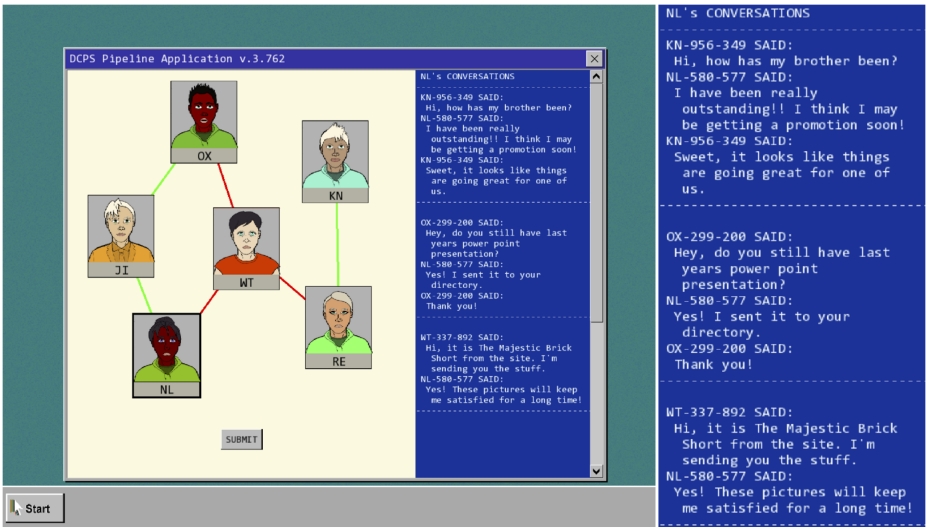

Fig. 3. A *Project Perfect Citizen* player constructs a theory of the connections in a procedurally generated social network using an in-game graph application. To home in on the suspicious person at the center of the network, she studies character SMS exchanges that are generated using Expressionist (detail shown on right).

most highly on, and passes the score for this heuristic (binned to `bad`, `good`, `great`, or `perfect`) as a content request to the game's Productionist module. The Productionist module then operates over an Expressionist JSON with tagsets that pertain to the three heuristics and contain tags corresponding to the four possible binned scores that may appear in a content request. To generate a blog comment, the module targets a top-level symbol with the desired markup (the tag corresponding to the binned heuristic score in the content request, *e.g.*, `balance:great`) and returns its terminal expansion.

The *Snapshot* Expressionist grammar contains 30 nonterminal symbols and 121 production rules, and is capable of generating 17,856 unique comments.

4.2 Generating SMS Exchanges in *Project Perfect Citizen*

Project Perfect Citizen is a "surveillance storytelling" game that tightly combines generative methods, environmental storytelling, and procedural rhetoric [1]; it is written in C++, runs on a custom engine built by its development team, and is beginning to receive press attention [3]. In the game, the player is a low-level employee at a government security agency who is tasked with exploring the private data of individuals to find and report evidence of suspicious activity. This plays out across a series of levels that each require the completion of two related tasks. In the first task (depicted in Figure 3), the player uses an in-game graph application to construct a theory of a social network with a suspicious person at its center; this is done by referencing character SMS exchanges that are generated using Expressionist. Once the ringleader has been

ascertained, the player proceeds to the second task, where she remotely beams into that character's desktop environment to locate and flag suspicious files.

Prior to the generation of SMS exchanges for a given level, a social network is generated by producing a series of characters who have unique personality models and who are linked to one another in specific ways. These network links are expressed as tuples specifying a level of suspiciousness (on a five-point scale) and a relationship type (family, work, or other community). While the representation of the character who is central to the level (the one whose desktop the player will explore) is held constant across playthroughs, the other characters in the network vary. A core authorial goal of the first phase of each level is to prime the player to the type of suspicious activity that she is meant to look out for in the second phase (*e.g.*, hacking, political corruption, sex crimes).

Project Perfect Citizen's Productionist module, written by the team in C++, satisfies targeted content requests for each link in a given network by generating SMS exchanges that evoke the link's suspiciousness level and relationship type, as well as the personalities of its characters. To do this, the module operates over an Expressionist JSON whose terminal expansions are complete SMS exchanges and whose tagsets correspond to link suspiciousness levels, relationship types, and character personality traits. Specifically, their Productionist starts at the single top-level symbol of the grammar and then traverses it in a randomized depth-first search (DFS), expanding symbols along the way. If at any time a symbol is encountered whose markup contradicts the content request, Productionist abandons the path that led to it. The result of this procedure is a terminal expansion of the top-level symbol in the form of a generated SMS exchange whose markup corresponds to (or at least does not contradict) the content request. Most often, this will mean that the generated exchange evokes the very characteristics captured in the content request, though it is possible to generate a default exchange that works for any pair of characters (to ensure that content can be generated for every possible case). The employment of DFS expansion with *backtracking* (*i.e.*, abandoning symbols with incompatible markup) makes this Productionist more complex than the one built for *Snapshot*.

Because each *Project Perfect Citizen* level surrounds a unique central character, each one has its own authored Expressionist grammar. These four grammars feature between 235 and 346 nonterminals, 571 and 956 production rules, and a staggering 1.3 trillion to 65.3 quadrillion generable SMS exchanges.

4.3 Generating Character Dialogue in *Talk of the Town*

Our next case study is the subject of a recent paper [19], so we will only briefly outline it here. It concerns dialogue generation in *Talk of the Town*, a deeply AI-driven game that we are currently developing. Underpinned by a rich simulation of character knowledge phenomena, players in this game will have to home in on the truth of a central mystery by engaging in dialogue interaction with non-player characters (NPCs) who form subjective beliefs about the world (which may be inaccurate). Dialogue interaction will be fully procedural, integrating dialogue management [20], natural language understanding (via Expressionist)

[25], and natural language generation [19]. Expressionist is central to our approach to the latter. Specifically, we rely on a CFG whose tagsets correspond to the concerns of our dialogue manager (enumerated in [20]) and a Productionist module that handles requests for lines of dialogue with desired markup (typically a tag corresponding to a speech act targeted by the dialogue manager).

While the Productionist modules in the previous two examples produced content by terminally expanding top-level symbols—the conventional way of generating from a CFG—this Productionist employs a unique *middle-out* expansion procedure [19]. Specifically, the module targets *mid*-level nonterminal symbols and then attempts to both forward-expand (by terminally expanding the symbol) and backward-chain (by executing production rules that have the symbol on its righthand side) from the targeted symbol. The result is a terminal expansion of a top-level symbol, but one that was in actuality built by chaining bidirectionally from a targeted mid-level symbol. As an example, consider a content request that solicits a line of dialogue that would express certain information to the player, say, details about a relative of the NPC speaker. In our grammar, we can author a nonterminal that expands to produce such dialogue (and is tagged with markup specifying this), but that (as a mid-level symbol) can also be included in larger contexts, *e.g.*, an extended monologue about the NPC's family history. This allows the Productionist to target the particular kind of snippet that it seeks—one with details about the relative—without having to reason about, in this case, the entire family monologue. Alternatively, a content request might only prescribe the gist of a larger structure, in which case Productionist would not have to worry about finer details. As we argue more deeply in [19], this supports a modular, hierarchical authoring scheme that makes it tractable (and less burdensome) to specify huge spaces of generable text. While the Productionist in *Project Perfect Citizen* will not expand a symbol whose markup contradicts the content request (backtracking), in *Talk of the Town* we generalize this to allow authors to attach arbitrary *preconditions* to nonterminal symbols in the grammar. This is simply implemented as a tagset, called `Preconditions`, whose tags are snippets of raw code that can be evaluated at runtime against the game and conversation state. As we assert in [19, 20], this is a very powerful authorial affordance that may be utilized to reduce repetition (*e.g.*, symbol can only be expanded if the last generated line did not have certain lexical content) and express underlying game state (*e.g.*, symbol can only be expanded if speaker has a certain personality trait). Finally, *Talk of the Town* makes use of an Expressionist authoring convention that is not used in *Snapshot* or *Project Perfect Citizen*: runtime variables. Rather than fully realized lines of dialogue, the terminal expansions of the *Talk of the Town* grammar are *templated* lines of dialogue, *e.g.*, "Hello, [interlocutor.name].". Gaps in these templates are called *runtime variables* and are specified as raw code snippets that Productionist can evaluate to a string at runtime (by binding to entities in the game state).

The *Talk of the Town* Expressionist grammar currently features 246 nonterminals and 665 production rules, and is capable of generating 2.8M lines of

dialogue. Examples of the generated dialogue can be found in example procedural conversations included in [20, 25].

4.4 Generating Character Thoughts in *Juke Joint*

Juke Joint is a small work of interactive storytelling that demonstrates an extension to the *Talk of the Town* AI framework by which characters form thoughts, expressed in natural language, that are elicited by environmental stimuli. Because it is the subject of another publication [18], we will remain brief in this case study as well. *Juke Joint* is set in a bar with a jukebox and two patrons who are each facing personal dilemmas, and the player's only action is to select which song from the jukebox will play. As the lyrics of the selected song emanate from the machine, thoughts are elicited in the minds of the characters, constituting streams of consciousness that eventually lead them to resolutions of their respective dilemmas.

Juke Joint's Productionist module utilizes all the same tricks as the *Talk of the Town* one—backtracking, preconditions, runtime variables, middle-out expansion—with an additional rub: heuristic expansion. Song lyrics in the game are annotated for their *themes* (*e.g.*, love, deception), and nonterminal symbols in its Expressionist grammar are annotated for the themes they are associated with. Critically, this allows for *heuristic evaluation* of nonterminal symbols: we have developed a scoring procedure that computes how well the themes of a span of lyrics match the themes associated with a nonterminal. With the themes of the current lyrics functioning as environmental stimuli, Productionist builds an elicited thought by targeting the highest scoring nonterminal symbol in the grammar. From here, it carries out middle-out expansion (as described in Section 4.3), but also *heuristic expansion*: whenever it has to choose between multiple viable production rules, it scores them (as the sum of the scores of the symbols on their left- and righthand sides) and selects a rule probabilistically (according to the scores). While it would not be tractable to compute the maximally optimal expansion trace for a set of stimuli (given the grammar's massive possibility space, noted below), this greedy expansion procedure works to ensure that the elicited thought will be reasonably associated with the stimuli. More generally, it allows us, as procedural authors, to specify expressive aims for generated content—in this case, to maximize the association between song lyrics and the character thoughts they elicit—while relying on Productionist to do the work of actually pursuing these aims (by consulting the explicit metadata attributed during authoring). Appealingly, this advance in system expressivity does not in turn affect the well-formedness of generated content, since Productionist still operates over a fully specified control structure—our Expressionist grammar.

The *Juke Joint* Expressionist grammar currently features 292 nonterminals, 801 production rules, and the largest possibility space of the Expressionist grammars discussed in this paper: 6.3 quintillion unique character thoughts.

5 Discussion and Conclusion

The Expressionist grammars discussed in the above case studies support possibility spaces spanning upward of *quintillions* of generable outputs, each instance of

Table 1. The array of strategies utilized by the four Productionist modules described in our case studies, all of which are made possible by Expressionist's markup affordance.

	Snapshot	*Project Perfect Citizen*	*Talk of the Town*	*Juke Joint*
Top-Down Expansion	✓	✓	✓	✓
Backtracking		✓	✓	✓
Preconditions			✓	✓
Runtime Variables			✓	✓
Middle-Out Expansion			✓	✓
Heuristic Expansion				✓

which comes packaged with metadata that is tailored to the particular expressive aims of a larger application. The tool's simple markup affordance—authors attach free-text markup to symbols that elegantly accumulates during expansion—yields a rich design space supporting an array of application-specific generation approaches, which Table 1 illustrates. As a slightly augmented way of generating from a CFG, *Snapshot* uses markup as a way of representing start symbols by their semantic functions. *Project Perfect Citizen* utilizes markup more deeply, employing a top-down expansion procedure that backtracks from nonterminals whose markup contradicts the content request. This notion is generalized in *Talk of the Town*, where markup may be used to attach arbitrary expansion *preconditions* to nonterminals, formatted as raw code that can be checked against the system state at runtime. Similarly, this Productionist utilizes *runtime variables*: generated outputs may be templates with gaps specified as raw code snippets that evaluate to strings at runtime. More drastically, the *Talk of the Town* Productionist provides additional authorial leverage by employing a unique *middle-out* expansion procedure that utilizes backward chaining in addition to conventional top-down expansion. Further still, the *Juke Joint* productionist carries out *heuristic* middle-out expansion, scoring candidate production rules for their appeal with regard to expressive aims asserted in automated content requests. These examples all work to demonstrate how Expressionist's markup affordance allows authors to move beyond the expressive power that is typical of CFGs, supporting application-specific semantics and diverse expansion approaches.

We believe that the usage of Expressionist in these four applications—and the fact that these projects integrate the tool into different technological ecosystems, including a Unity C# project and a custom C++ game engine—validates it as a powerful new method for in-game text generation. Even more promisingly, the authors who used Expressionist in these projects were not NLG practitioners. We view this as an encouraging indication of the potential of this tool to allay the burden of generative text in interactive storytelling, which we articulated at the beginning of this paper. Expressionist will be publicly released in late 2016, and we hope that others will consider using it in their own future projects.

References

1. Bad Cop Studios: Project Perfect Citizen (2016)

2. Bona, D.J.L.: Recursion in Cognition: A Computational Investigation into the Representation and Processing of Language. Ph.D. thesis, University Rovira i Virgili (2012)
3. Caldwell, B.: Free loaders: This desktop is under investigation. Rock, Paper, Shotgun (June 2016)
4. Cavazza, M., Charles, F.: Dialogue generation in character-based interactive storytelling. In: Proc. AIIDE (2005)
5. Compton, K., Kybartas, B., Mateas, M.: Tracery: An author-focused generative text tool. In: Proc. ICIDS (2015)
6. Elson, D.K.: Modeling narrative discourse. Ph.D. thesis, Columbia University (2012)
7. Evans, R., Short, E.: Versu—a simulationist storytelling system. TCIAIG (2014)
8. Gatt, A., Reiter, E.: SimpleNLG: A realisation engine for practical applications. In: Proc. ENLG (2009)
9. Horswill, I.D.: Architectural issues for compositional dialog in games. In: Proc. GAMNLP (2014)
10. Joseph, E.: Bot colony—a video game featuring intelligent language-based interaction with the characters. Proc. GAMNLP (2012)
11. Lessard, J.: Designing natural-language game conversations. Proc. DiGRA-FDG (2016)
12. Loyall, A.B., Bates, J.: Personality-rich believable agents that use language. In: Proc. AGENTS (1997)
13. Mateas, M.: The authoring bottleneck in creating AI-based interactive stories [panel]. In: Proc. INT (2007)
14. Mawhorter, P.A.: Artificial Intelligence as a Tool for Understanding Narrative Choices. Ph.D. thesis, University of California, Santa Cruz (2016)
15. Montfort, N.: Generating narrative variation in interactive fiction. Ph.D. thesis, University of Pennsylvania (2007)
16. Reed, A.A., et al.: A step towards the future of role-playing games: The SpyFeet mobile RPG project. In: Proc. AIIDE (2011)
17. Rowe, J.P., Ha, E.Y., Lester, J.C.: Archetype-driven character dialogue generation for interactive narrative. In: Proc. IVA (2008)
18. Ryan, J., Brothers, T., Mateas, M., Wardrip-Fruin, N.: Juke Joint: Characters who are moved by music. In: Proc. Experimental AI in Games (2016)
19. Ryan, J., Mateas, M., Wardrip-Fruin, N.: Characters who speak their minds: Dialogue generation in Talk of the Town. In: Proc. AIIDE (2016)
20. Ryan, J., Mateas, M., Wardrip-Fruin, N.: A lightweight videogame dialogue manager. In: Proc. DiGRA–FDG (2016)
21. Ryan, J.O., Barackman, C., Kontje, N., Owen-Milner, T., Walker, M.A., Mateas, M., Wardrip-Fruin, N.: Combinatorial dialogue authoring. In: Interactive Storytelling (2014)
22. Ryan, J.O., Fisher, A.M., Owen-Milner, T., Mateas, M., Wardrip-Fruin, N.: Toward natural language generation by humans. In: Proc. INT-SBG (2015)
23. Ryan, J.O., Mateas, M., Wardrip-Fruin, N.: Open design challenges for interactive emergent narrative. In: Interactive Storytelling (2015)
24. Short, E.: Procedural text generation in IF. https://emshort.wordpress.com/2014/11/18/procedural-text-generation-in-if/ (Nov 2014)
25. Summerville, A.J., Ryan, J., Mateas, M., Wardrip-Fruin, N.: CFGs-2-NLU: Sequence-to-sequence learning for mapping utterances to semantics and pragmatics. Tech. Rep. UCSC-SOE-16-11, UC Santa Cruz (2016)
26. Vacuous Games: Snapshot (2016)