Proceedings of the Eighteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment
(AIIDE 2022)

Step: A Highly Expressive Text Generation Language

Ian Horswill
Northwestern University
ian@northwestern.edu

Abstract

Games often generate text from human-authored templates and adapt that text to relevant context. Many systems have been developed to aid this process using techniques such as context-free grammars, randomization, logic programming, global state, and HTN planning.

In this paper, I present *Step*, a novel programming language for text generation. So far as I can determine, previous techniques can all be implemented within *Step* using a few lines of code. This allows designers to mix-and-match features as needed, without having to write an interpreter for a new language.

While extremely expressive as a programming language, *Step* is also intended to allow writers to add new text to an existing system with minimal markup and little knowledge of programming. In a head-to-head comparison, the *Step* implementation was more compact than, and used less markup than, a *Prolog* implementation of the same generator.

Introduction

Generating text from human-authored templates is a common task in games. A number of specialized research systems (Compton, Filstrup, and Mateas 2014; Ryan, Seither, et al. 2016) and at least one commercial middleware system, SpiritAI's *CharacterEngine*, have been developed for text generation. These systems combine grammars with some subset of randomization, preconditions on rules, backtracking, argument passing, global state, report-outs of information about the derivation, and escapes to raw code in the host language.

While the precise algorithms and capabilities of these systems are often unclear from the published descriptions, they all appear to be implementable using non-determinism, unification, and reflection. This isn't surprising since context-free grammar parsing/generation, definite clause grammar parsing/generation, *Prolog*-style logic programming (Warren, Pereira, and Pereira 1977), and *SHOP*-style HTN planning (Nau et al. 1999) all use depth-first search of an AND/OR tree. They vary primarily in what state information they track along the way.

In this paper, I present *Step*, a novel language for text generation that implements a robust generalization of this control structure, together with reflection primitives for reasoning about the derivations of a string. So far as I can determine, it can implement previous techniques with only a few lines of code.

Having one general system rather than many bespoke systems lowers the cost of experimentation. Designers can mix and match techniques without having to modify the core system. If a technique turns out to be unworkable, considerably less effort was wasted. It also allows more engineering effort to be devoted to tooling: *Step* supports syntax highlighting, interactive debugging, stochastic sampling of derivations, and custom design-rule checkers.

While extremely expressive computationally, it is designed to present minimal friction to writers. Non-programmers may use libraries designed by specialist programmers, and add new text to an existing system with minimal markup and little knowledge of programming. *Step* has been used for two years in an AI-based narrative class targeting both CS majors and non-STEM students.

Step is open source, and distributed under the MIT license. The *Step* interpreter and debugger may be found at https://github.com/ianhorswill/{Step,StepRepl}. The syntax highlighting package for Visual Studio Code is available for free in the VS Code Marketplace.

In the remainder of this paper, I describe *Step* and show how it can be used to emulate existing techniques. I will not assume familiarity with *Prolog*, but will assume familiarity with unification and the general notion of non-deterministic algorithms.

Related Work

While *Step* is not an interactive fiction language *per se*, there are several interactive fiction languages that generate text,

some of which use some variant of logic programming. *Inform 7* (Nelson 2006), the oldest and best developed of these, uses an English-like syntax to allow authors to write deterministically pattern-matched rules for game logic. *Versu* (Evans and Short 2014) uses straightforward text templates, but its game logic is implemented in a novel logic programming language. Both systems use template-based NLG, although *Inform*'s appears to be equivalent to a CFG. Martens' *Ceptre* is a logic programming language for interactive narrative based on linear logic; Martens' *Quiescent Theatre* compiles linear-logic narratives to *Twine*, using templated text (Martens 2015).

Another IF language, *Curveship* (formerly known as *nn*), uses a full natural language generation pipeline to transform symbolic structures into English surface text, allowing it to dynamically change grammatical tense and mood (Montfort 2007). More recently, NLG pipelines have been implemented in a few research games. *SpyFeet* (Reed et *al*. 2011a, 2011b) can dynamically vary generation based on parameters such as character personality. *Bot Colony* (Joseph 2012) generates English from logical forms, although the authors' paper focused primarily on NL understanding rather than generation.

MKULTRA (Horswill 2014) also generated surface forms from logical forms, however it did so using definite clause grammars, context-free grammars in which nonterminals can take parameters that bind through unification, rather than a full NLG pipeline. More recently *Lume* (Mason et al. 2019) has also used DCGs. To make *Prolog*-language DCGs more usable by writers, it provided a bespoke tool to heuristically derive DCG rules from plain text.

Many systems use some variant of a context-free grammar (CFG). *Tracery* (Compton et al. 2014; Compton and Mateas 2015) is an extremely successful CFG-based generator designed for casual users. It is by far the most widely used text generation system. It augments the base CFG with the ability to call arbitrary JavaScript code. *Dunyazad* (Mawhorter 2016) uses a CFG scheme to generate text from a logical form. It includes a number of extensions to handle pronominalization and subject/verb agreement, and contains an interesting scheme for imposing a tree structure on the names of non-terminals, with wildcard matching of non-terminals, implementing some of the functionality of DCGs. *Improv* (Dias 2020) is a also CFG-based system, similar to Tracery. However, *Improv* is intended to base its text generation on game state, such as a world model. It provides hooks to allow the programmer specify arbitrary JavaScript code to call when expanding a nonterminal. The code can reorder or replace possible expansions based in game state.

The most widely used text-generation middleware system is *Expressionist* (Osborn et *al*. 2017; Ryan et al. 2015; Ryan et *al*. 2016; Ryan et *al*. 2016). It augments CFGs with a tagging mechanism such that the input is a set of tags that must be generated and the output is a string plus a set of tags that were generated. The exact operation of the system isn't described, and indeed many different versions have been written with different sets of features, including one version used in a commercial middleware tool, *CharacterEngine* (SpiritAI). In this sense, *Expressionist* is perhaps best thought of as a family of related tools united by the combination of CFGs and tagging. The argument of this paper is in part that *Step* would be a more productive implementation language for systems like *Expressionist* than JavaScript or C#.

Dear Leader's Happy Story Time (Horswill 2016) used a higher-order HTN similar to *Step*, but was less general and implemented as an embedded language in *Prolog*, making it considerably less accessible. *StoryAssembler* (Garbe et al. 2019) combines a forward state-space planner with an HTN planner to generate branching choice-driven narratives. A template system is then used for text generation.

Syntax

Step code consists primarily of declarations of task methods. A method declaration has the form:

TaskName arguments … **:** *body*

Multiline methods end with `[end]`. A method with no body ends with a period rather than a colon. Multiple bodiless methods can be loaded in tabular form from CSV files.

Following Ingold's (Ingold 2015) dictum that writer-facing scripting languages should look like text with occasional code markup, not code with quoted strings, the body text is printed verbatim. Hello World is simply:

```
HelloWorld: Hello world!
```

I will adopt the convention of proportionally-spacing text to be printed, and monospacing all other code.

Calls to other *Step* tasks can be placed in the body wrapped in square brackets to escape from text mode:

```
Story: [Beginning] [Middle] [End]
```

Their output is substituted for the bracketed expressions.

Argument expressions can be variables or constants. Local variables of a method are named starting with a `?`: `?x`, `?foo`, or just `?`. They behave as logic variables in logic programming languages: initially unbound, then acquiring values through unification with other expressions.

Global variables are shared by all tasks and methods. Their names start with a capital letter: `HelloWorld`, `Story`, `Beginning`.

Words beginning with a lower-case letter, such as `string` or `this_is_a_string`, are treated as string literals. Strings quoted with double-quotes, such as `"quoted string"`, are represented internally as arrays of tokens (words). True single string constants with embedded spaces can be typed either by escaping the space with a backslash or quoting it with vertical bar characters, as in: `|quoted string|`. Quoted strings are stored in tokenized form to allow the output system to identify word and sentence boundaries during post-processing.

Numeric literals are largely as in other languages.

Bracketed expressions within a call are treated as tuples/lists. The language is thus homoiconic: tuples can be used to represent calls to other code.

Variable Substitution

The body of a task is treated as text to be printed, with two exceptions: bracketed expressions to call subtasks, discussed above, and variables, whose values are substituted into the output. The method:

```
Greet ?who: Hello, ?who!
```

Prints the text "Hello," followed by the value of ?who, and an exclamation point. In particular, the appearance of a variable here is treated as syntactic sugar for a call to the built-in task, `Mention`:

```
Greet ?who: Hello, [Mention ?who]!
```

The default `Mention` implementation prints the value of its argument. However, the user may provide a more sophisticated system implementing pronoun substitution, special casing the first mention of a topic, or do whatever other clever processing is desired. Since this can involve arbitrarily sophisticated coding and grammatical knowledge, users generally use an off-the-shelf implementation of `Mention`.

Although the default is for an interpolation to call the built-in task `Mention`, it can be used to call any other task. The construction *?variable*`/`*Task* is sugar for `[`*Task ?variable*`]`. Thus:

```
Greet ?who: Hello, ?who/FirstName!
```

Calls `FirstName` rather than `Mention`. It's equivalent to:

```
Greet: ?who: Hello, [FirstName ?who]!
```

but hopefully more readable. The `+` operator can be used to call multiple tasks on the same variable:

```
?who/FirstName+LastName
```

will print first the `FirstName` and then `LastName` of `?who`.

Finally, the `/` operator can be repeated to invoke binary relations. The construction *?variable*`/`*Relation*`/`*Task* is sugar for `[`*Relation ?variable ?temp*`] [`*Task ?temp*`]`. Thus, `?who/Brother/FirstName`, which is equivalent to `[Brother ?who ?b] [FirstName ?b]`, i.e. "find `?who's` brother `?b` and then print his first name."

Semantics

Tasks can have multiple methods that handle overlapping cases, with the system trying each at run-time. *Step* is a non-deterministic language: calls and methods are allowed to fail and the system performs a backtracking search to find an execution path in which all selected methods are successful.

For example, the program:

```
Run: [A] [B]
A: [C]
A: [D]
B: [E]
B: [F]
```

defines one `Run` method that executes `A` followed by `B`, each of which have two methods, calling either `C` or `D`, or `E` or `F`, respectively. A call to `Run` thus has four possible execution paths: `Run A C B E`, `Run A C B F`, `Run A D B E`, and `Run A D B F`. If any of the methods in a path fails, the system backtracks and tries another method. If all methods for a call fail, the call itself fails and the method containing it backtracks.

Importantly, any changes to the execution state (output text, variable bindings, cool downs) made by a method are rolled back if that method is backtracked. When execution completes, it appears as though the system had only executed the operations of the successful path. In most cases, the designer can ignore the details of the search and assume the system will do some version of the right thing.

Pattern-Directed Invocation

Like *Prolog*, *Step* uses pattern-directed invocation, meaning methods can require specific values for specific arguments or require specific arguments be identical. Caller arguments are matched to method argument patterns using unification. The method fails and is ignored if they do not match. In the fragment:

```
Greet john: Oh, God. You again!
Greet ?who: Hello, ?who!
```

The system prints "Oh, God. You again!" when the argument is the string `john`, otherwise it prints "Hello, `?who`!", whatever `?who` might be.

Sequencing and Randomization

Also like *Prolog*, methods are tried in the order they appear in the source. This can be changed by tagging a task *[randomized]*, in which case each call uses a different weighted shuffle of the methods. Methods can also be tagged as single-use. The example:

```
[randomized]
Greet: [once] Dude.
Greet: Hi.
[2] Greet: Hello.
```

lets `Greet` try the methods in any order, with stipulation that the first should be used only once per run, and that the last should be chosen twice as often as the others.

Search Control

The default semantics for a *Step* task differ from logic programming languages such as *Prolog*. By default, a *Step* task is required to succeed exactly once. It is an error for it to fail, and once it succeeds, it's done: the system will not rerun the task should it backtrack later. This makes debugging easier, since standard logic programming semantics can unintentionally hide error conditions by silently backtracking over errors or produce infinite loops when a task intended to succeed only once is written in such a way that it can keep generating new solutions.

Standard logic programming semantics are enabled on a task-by-task basis by tagging a task with *[predicate]*:

```
[predicate]
Familiar john sarah.
Familiar sarah shelly.
Greet ?who: [Familiar Speaker ?who] Dude.
Greet ?who: Hi.
```

Here, `Familiar` is a predicate and so calls to it are allowed to fail. `Greet` then uses it as a guard on its first method to ensure it's used only when the speaker, stored in the global variable `Speaker`, and the addressee, `?who`, are on familiar terms. The call `[Greet sarah]` will print "Dude" when `Speaker=john`, In all other cases, the call to `Familiar` will fail, and the first method will also fail. The system will then use the second method, which prints "Hi."

Mutable State

Like most programming languages, *Step* allows the manipulation of mutable state: text output, assignments to variables, *etc*. However, it guarantees that mutations are undone when a call is backtracked: text is "unwritten" and variables "unset." Upon completion of a program, it is as if the system had only ever executed the final, successful path.

In addition to text output and unification of local variables, *Step* supports explicit mutation of global variables:

```
[set Speaker = jane]
```

behaves exactly like assignment statements in C or Python, however the update is undone upon backtracking.

Step also supports mutable predicates, which is useful for symbolic planning. The annotation *[fluent]* declares that a predicate can be dynamically set true or false for particular ground values at runtime using the `now` statement. Rules can still be specified for the predicate, but `now` assertions override them when the arguments match.

For example, the following fragment defines a `Shoot` task that marks a character as dead:

```
# All characters are alive by default
[fluent]
Alive ?.

# Shoot ?character
# Kills ?character
Shoot ?c: You shot ?c. [now [Not [Alive ?c]]
```

The *[function]* annotation marks a predicate as being mutable, but also that its last argument is unique given the values of its previous arguments. Therefore asserting `[On a c]` would implicitly retract `[On a b]`, since the two cannot be true simultaneously.

```
# On is a mutable predicate whose second
# argument is unique given its first
[function]
On a b.

# Move ?x to ?y
# Retracts ?x's old position and
# asserts [On ?x ?y]
Move ?x ?y: [now [On ?x ?y]]
```

As with other mutations, these are automatically undone when backtracked.

Higher-Order Operations

Step supports higher-order tasks – tasks that take code as an argument. Tuple data structures look like calls and so can be treated as code and passed as an argument to a task. The task can use the `Call` primitive to execute it. This can be used to write looping constructs, metadata, design-rule checkers, and so on.

Reflection

Reflection allows a program to query itself as a data object. In static reflection, a program queries its own source text. In dynamic reflection, it queries its execution state.

Step supports static reflection via higher-order primitives such as `TaskSubtask`, which tests whether a task has a method that calls another task. This allows programmers to write bespoke static analysis tools, such as checking that every precondition in an HTN formalization appears in the add list of some task or operator. For example, if `Beat` is a predicate that identifies tasks intended as story beats, then:

```
[Beat ?b] [Not [TaskSubtask ? ?b]]
```

identifies beats that are not called by any other code.

Step supports dynamic reflection through higher-order primitives such as `PreviousCall`, which unifies its argument with a call in the current execution path:

```
CalledBeat ?b: [Beat ?b] [PreviousCall [?b]]
Storybeats ?beats:
   [FindUnique ?t [CalledBeat ?b] ?beats]
```

Here `CalledBeat` finds beats used in the current execution path and `Storybeats` finds a list of all beats generated thus far in the story. This is used in the profiler for *Dear Leader's Happy Story Time* to find beats that are used too frequently, infrequently, or not at all. It samples one million random stories and computes beat and plot-point usage statistics.

The `UniqueCall` higher-order primitive executes its argument but backtracks over any calls that unify with previous calls in the execution path. This is useful for choosing beats or plot points that have not already been used. It's also useful for story casting. The fragment:

```
Cast ?role: [UniqueCall [Character ?role]]
```

might be used to call `Character` to bind `?role` to some character not already cast in the execution path.

Implementation and Tooling

The *Step* interpreter is written in C#. It is a stand-alone DLL that can be used in any C#/.NET/CLR application, including Unity games. The language is designed for versatility over efficiency: while it is more than fast enough for its intended use cases, it would not be an appropriate language for hard search problems.

A great deal of effort has been put into tooling for the language. An interactive debugger, *StepRepl*, supports interactive execution, breakpointing, single-stepping, etc. Combined with syntax highlighting support for Visual Studio Code, it forms an effective development environment. *StepRepl* also provides primitives for developing custom samplers and profilers, and automatically generates a reference manual of built-in primitives.

There is also a tutorial and course materials for targeting CS and non-STEM majors, as well as a simple autograder.

Emulating Existing Systems

Standard text-generation techniques can be written directly in *Step* in a few lines. We survey some examples here.

Logic Programming

Predicates in logic programming are emulated by *Step* tasks marked *`[predicate]`* that output no text. However, *Step* does not implement *Prolog*'s cut operator or its syntax for linked lists.

Context-Free Grammars

Nonterminals in a CFG are modeled as parameterless tasks. The *Tracery* (Compton et al. 2014) grammar:

```
{
    "origin": "#greeting#, #noun#!",
    "greeting": ["Howdy", "Hello"],
    "noun": ["world", "solar system"]
}
```

can be written in *Step* as:

```
Origin: [Greeting], [Noun]!
Greeting: [randomly] Howdy [or] Hello [end]
Noun: [randomly] world [or] solar system [end]
```

Definite-Clause Grammars

DCGs add parameters and unification to CFGs. DCG nonterminals are emulated in *Step* as parameterized tasks with that output text. The `Greet` example from the previous section on pattern-directed invocation was a DCG.

Pronoun Substitution

Discourse context can be tracked by adding state variables. Since *Step* automatically retracts variable updates upon backtracking, the variables always represent the values for the selected execution path.

As one very simple example, we can generate personal pronouns by defining `Mention` to track the most recent referents for each pronoun, substituting the pronoun when the item to print matches, and updating the referents when new items are printed. Here is a schematic example:

```
Mention LatestThey: they
Mention LatestShe: she
```

```
Mention LatestHe: he
Mention ?who:
  [Say ?who] [PreferredPronoun ?who ?pro]
  [Update ?who ?pro]
[end]
Update ?who he: [set LatestHe ?who]
Update ?who she: [set LatestShe ?who]
Update ?who they: [set LatestThey ?who]
Say ?x: [Not [Mentioned ?x]] [SayFirst ?x]
[now [Mentioned ?x]]
Say ?x: [Write ?x]
```

This is admittedly a simple-minded scheme, but it's the scheme most used in games. More complex versions implementing other kinds of anaphors are natural extensions.

Planning

Step is not intended for sophisticated planners, but does make it easy to add simple planning to a system. Since *Step* is a generalization of SHOP-style HTNs, HTN planning is easy. Space precludes a more substantive example of other styles, but the following simple library function can be used to add means/ends analysis to a *Step* program.

We assumes a predicate, `[Achieves` *?goal ?task*`]`, that is true when *?goal* is a postcondition of *?task*. `Achieves` could be written to find the task either by static reflection, or the programmer could manually add `Achieves` methods to tag tasks according to the goals they achieve. For example, a blocks world planner might declare:

```
  Achives [On ?a ?b] [Move ?a ?b].
```

We also assume a fluent predicate, `TryingToAchieve`, that is used to detect looping (a goal leading to itself as a subgoal). We can now write a task, `[Achieve` *?goal*`]`, that does nothing if *?goal* is already true, otherwise it makes it true by looking up a task with its postcondition and executing it:

```
[predicate]
Achieve ?goal: [Call ?goal]
Achieve ?goal:
  [Not [TryingToAchieve ?goal]]
  [now [TryingToAchieve ?goal]]
  [Achieves ?goal ?task] [Call ?task]
  [now [Not [TryingToAchieve ?goal]]]
[end]
```

The first method tests the `?goal` using `Call`. If false, the second method checks that we are not already trying to achieve the goal. If not, it marks the goal as currently being attempted. The it uses `Achieves` to find an `?task` that can achieve it, then executes it.

Tasks can then then written in the form:

```
Task args …:
  [Achieve [Precondition]] …
  [now added [Not deleted]]
[end]
```

Expressionist-style Tagging

Tagging is one of *Expressionist*'s marquee features (Ryan, Seither, et al. 2016). This can be emulated in *Step* using calls to a placeholder, `[Tag` *tag*`]`, to tag a method. We can determine the tags used in the current derivation using:

```
  [FindAll ?t [PreviousCall [Tag ?t]] ?all]
```

This will bind `?all` to a list of all the tags used in the current execution path. We can force derivations to use a particular tag by calling `[PreviousCall [Tag` *tag*`]]` after generation. Tags can be arbitrary data, removing *Expressionist*'s need to encode data as complex strings and parse them after the fact. Tags can also be computed at run-time.

Example Applications

Although space makes it impossible to include source code, we mention here some small but non-trivial applications of *Step*.

Dear Leader's Happy Story Time

We have reimplemented the party game *Dear Leader's Happy Story Time* (Horswill 2016) in *Step*. *Dear Leader* uses a higher-order HTN to generate stories that incorporate narrative devices such as callbacks and montages. The *Step* version is a direct translation of the original, with more fluent text generation, as well as sampling and profiling tools.

This gives us the opportunity to directly compare *Step* with a previous system. The *Step* version is considerably more compact: 845 lines vs. 1319 lines of Prolog code, a 35% reduction despite increased functionality.

While readability is more subjective, it's instructive to compare implementations of the same story beat. The version for the original *Prolog*-based HTN reads:

```
beat(married_life(P, L, bed) :
  { setting: home },
  $text(
    "[P] and [L] read in bed together")).
```

By contrast, the *Step* version is:

```
MarriedLife ?P ?L bed: [Setting home]
    ?P and ?L read in bed together [Beat]
```

The *Step* version is noticeably shorter and, I would argue, easier to read and write. Taking punctuation characters as a proxy for the complexity of code markup, the *Step* version requires 9 punctuation characters vs. the original 21, a 57% reduction.

Story Sifting

Step has been used to implement story sifting (Ryan 2018) in both Ryan's *Talk of the Town* (ibid), and a simple "newspaper" generator for the commercial game *City of Gangsters* (Zubek and Viglione). *City of Gangsters* involves over 1000 NPCs at any given time, so the player can't track events through gameplay alone. Sebastian Perez-Delgato's newspaper generator lets players explore events in their city by mining save files. For example, the first line of the method below detects dead NPCs with surviving children, the rest generates a brief obituary for them:

```
Obituary:
  [Dead ?p ?e] [Child ?p ?m] [WillDie ?m ?]
  In loving memory ?p, ?p/Born/FullDate – ?e/FullDate.
  No one who met ?p/FName forgot their ?p/PTrait personality and infectious passion for ?p/FavActivity.
  They are survived by ?m/Child/FName. Funeral services will be held ?p/FuneralDate/FullDate at noon in Old St. Patrick's Church.
[end]
```

TTRPG Scenario Generation

With permission of the authors, we've implemented the published scenario generator for the queer, anticapitalist, table-top role playing game *iHunt: Killing Monsters in the Gig Economy* (Hill and Young 2019) as a *Step* program. It allows GMs to quickly generate scenarios and choose the one they like best. The code is given in the appendix.

The published generator is 9 pages of tables and English text. The *Step* version includes all the cases of the original, but filters nonsensical combinations and generates narrative descriptions of the scenarios. The authors are now working to develop new content for the game directly in *Step*.

Conclusion

Current text generation systems can be emulated by a small basis set of features: non-determinism, pattern-directed invocation, reflection, controlled randomization, and higher-order procedures. Previous systems can be seen as partial implementations of this basis. A full implementation forms a natural and powerful scripting language on which to build whatever application-specific tricks one needs.

Step demonstrates that full implementations of the basis set do not require more engineering effort: *Step* is 12K lines of C# code + 3.6K for *StepRepl*, compared to 73K lines Python, JavaScript, and JSON for *Expressionist*.

While *Step* is not the only possible implementation, it provides a congenial syntax that allows more compact expression with less markup than *Prolog* in a head-to-head comparison. It has been used successfully by both programmers and non-programmers to write generators for a variety of applications.

Appendix: Extended Example

The following is an example of a non-trivial *Step* program. It generates quests for the *iHunt* tabletop RPG based on the published dice-based quest generator. This differs from the previous examples in the paper in that it is not algorithm-heavy; it gives a sense of a writer's view of the language.

Job generator.step

This is the generator itself. A gig (a quest) consists of a client hiring you to kill a monster (the mark) because of a trouble they caused. The generator presents two suggested hangups (complications) for the GM to confront the players with, and an aftermath (also a complication).

The generator filters nonsensical combinations that were possible in the original dice-based version, such as corporations falling in love, marks being simultaneously good and evil, or characters who are already dead taking actions later in the story. This is done by adding calls to the predicates `Individual`, `GoodGuy`, and `Alive/Dead` to methods that require those properties. They're updated using `now`, when a method narrates a change to them in the story world.

The text uses the following utilities defined in Mention.step; space precludes including their source code:

- `Obj` (invoked using `^`*Variable*`/Obj`)
  Same as `Mention`, but generates pronouns in object case.
- `Mention` (invoked using `^`*Variable*)
  Prints *variable*'s value and generates subject-case pronouns as appropriate. Tracks plurality for conjugating auxiliary verbs. Full phrases are printed for first mention, and a short form is used thereafter.
- `Poss` (invoked using `^`*Variable*`/Poss`)
  Same as `Mention`, but generates in possessive case, i.e. either "*Variable*'s" or a possessive pronoun.
- `Is` (invoked using `[Is]`)
  Generates "is" or "are", depending on the plurality of the subject.
- `Has` (invoked using `[Has]`)
  Same, but for the verb has.

In the code below, lines have been wrapped in order to save space and stay within the two-column format. The

text has also been edited to be less profane than the original game.

```
# Generate a gig
Gig:
  # Choose client and mark
  [PossibleClient ?client]
  [now Client = ?client]
  [PossibleMark ?mark] [now Mark = ?mark]

  You are hired by ^Client to take out ^Mark.

  # Print random trouble, hangups, etc.
  [TroubleCausedByMark]
  [NewParagraph]
  [Hangup] [Hangup2]
  [NewParagraph]
  [Aftermath]
[end]

#
# Storyworld state
# These track facts that may have been
# established in previous plot points
# in order to prevent future ones from
# contradicting them.
#

# Was the character killed?
predicate Dead ?who.
# Does the character want to live?
predicate ResistingDeath ?who.
# Is the character good or evil?
predicate GoodGuy ?who.

[predicate]
Alive ?who: [Not [Dead ?who]]

# Pick a random mark
# 1/8 chance of it being a group.
[randomly]
[7] PossibleMark ?monster:
   [Monster ?monster]
PossibleMark [group ?monster]:
   # Pick a monster to make up the group
   # It must not already be a group
   [Monster ?monster] [Singular ?monster]

#
# Possible troubles for the mark to cause
#
[randomly]
TroubleCausedByMark:
  ^Mark [Is] actually good, but ^Client wants ^Mark
  dead. [now [GoodGuy Mark]]
# The Individual test here blocks this from
# being used when the client is a group or
# corporation.
TroubleCausedByMark: [Individual Client]
  ^Mark hurt someone very dear to ^Client.
TroubleCausedByMark:
  ^Mark [Has] been terrorizing the community.
TroubleCausedByMark:
  ^Mark stole something very important to ^Client.
TroubleCausedByMark:
  ^Client has found evidence that ^Mark [Has] been killing dozens of people.
TroubleCausedByMark:
  ^Mark did something that seriously embarrassed ^Client/Obj.
TroubleCausedByMark:
  [Individual Client] ^Mark stole ^Client/Poss *[randomly]* boyfriend *[else]* girlfriend *[end]*.
TroubleCausedByMark:
  ^Client just really hates ^Mark/Plural.
TroubleCausedByMark:
  ^Mark didn't do anything wrong, ^Client is just a jerk.

#
# Possible first problems for the players
# to encounter
#
[randomly]
Hangup: [Individual Client]
  somebody killed ^Client/Obj. This may complicate getting paid ... [now [Dead Client]]
Hangup:
  when you find ^Mark/Obj, it turns out to actually be [UniqueCall [PossibleMark ?realMark]] ?realMark [now Mark = ?realMark].
Hangup: when you finally find ^Mark/Obj, the area is swarming with cops.
Hangup: ^Mark [Is] in the middle of a crowd.
Hangup: killing ^Mark/Obj requires some very hard to come by supplies.
Hangup: ^Mark [Has] a pretty good argument why you shouldn't kill them.
        [now [ResistingDeath Mark]]
Hangup: ^Client really just want[s] you to kill ^Mark/Obj so ^Client can do ^Client/Poss own evil.
Hangup.      # No hangup!

#
# Possible second problems
# These depend on the first problems
#
```

```
[randomly]
Hangup2: ^Mark offer[s] you a better deal to leave
         them alone.
Hangup2: [Individual Client] Then, ^Mark
         offer[s] you a ton of money to kill ^Client.
Hangup2: [DeeperThreat] now you have a much
         bigger problem.
Hangup2: You're actually being framed; ^Client
         actually just wants to screw you over.
Hangup2:  When you finally find ^Mark/Obj, they're
          already dead. Sucks to be you.
Hangup2: When you find ^Mark/Obj, there's
         [randomly] another hunter [or] an amateur
         [else] a seriously clueless cop [end] also on
         the job.
Hangup2: [PotentiallyHot Mark] When you find
         ^Mark/Obj, they turn out to be extremely
         gorgeous. What will you do?
Hangup2: [Not [ResistingDeath Mark]] ^Mark
         set the whole thing up as an elaborate form
         of suicide.
Hangup2.      # no hangup!

# The mark wasn't the real threat!
DeeperThreat:
  [Not [GoodGuy Mark]] Little does ^Client know,
  the ^Mark is really being directed by an archdemon
  [now Mark = archdemon].

#
# Possible aftermaths
# These create continuing challenges for
# the players, generally economic ones.
#
[randomly]
Aftermath:
  In the end, things seemed to work out okay. That's dis-
  turbing.
Aftermath:
  [Alive Client] [Individual Client] Then
  ^Client underpaid because [randomly] they're actually
  kind of broke [or] because they didn't like the way you
  did the job [else] they're a jerk [end].
Aftermath:
  [Alive Client] [Individual Client]
  [randomly] suddenly ^Client is nowhere to be found
  [else] ^Client has a tantrum and refuses to pay[end].
Aftermath:
  Fighting ^Mark/Plural is expensive. Your expenses
  are going to eat most of your profits.
Aftermath: Someone close to you got caught up in it.
Aftermath:
  ^Mark [Has] friends; they won't forgive you.
Aftermath:
  Now your boss at your day job is pissed at you for
  missing work.
Aftermath:
  [Alive Client] now ^Client has another job for
  you, a really hard one, and they want you to start right
  now.
```

Monsters.csv

This is a spreadsheet file defining three predicates: `Monster`, the set of monsters, `PluralForm`, a binary relation giving the plural form of each monster, and `PotentiallyHot`, a unary predicate defining whether a monster is a potential player love interest.

Monster	@PluralForm	Potentially-Hot?
hungry dead	hungry dead	No
vampire	vampires	Yes
wizard	wizards	Yes
werewolf	werewolves	Yes
demon	demons	Yes

PotentialClient.csv

This also defines a three predicates: `Client`, the set of possible clients, `ShortForm` gives a more compact form for referring to the client (the long form is only used for the first mention of the client), and `Individual`, which indicates a client is a person and not an institution.

Client	@ShortForm	Individual?
corporate client	corporation	no
wealthy individual	client	yes
overwhelmed executor	executor	yes
upwardly mobile professional	yuppy	yes
curious party	client	yes
poor community	community	no
mysterious benefactor	client	yes

Acknowledgements

I would like to thank the reviewers for their patience, testing, and helpful comments. I would also like to thank Rob Zubek, Matt Viglione, Hecate Robison, Olivia Hill, and Filamena Young for feedback and encouragement; and Diana Smith, Andrea Nolla, Sebastian Perez-Delgado, Le Fang, Yiran Zhang, and the students and TAs of CS 295 and CS 396 for beta testing.

References

Compton, Kate, Benjamin Filstrup, and Michael Mateas. 2014. "Tracery : Approachable Story Grammar Authoring for Casual Users." *Papers from the 2014 AIIDE Workshop, Intelligent Narrative Technologies (7th INT, 2014)* 64–67.

Compton, Kate, and Michael Mateas. 2015. "Casual Creators." *Proceedings of the Sixth International Conference on Computational Creativity June*. doi: 10.1074/jbc.M409039200.

Dias, Bruno. 2020. "Improv." github.com/sequitur/improv. Accessed: 8-11-22.

Evans, Richard, and Emily Short. 2014. "Versu - A Simulationist Storytelling System." *IEEE Transactions on Computational Intelligence and AI in Games* 6(2):113–30.

Garbe, Jacob, Max Kreminski, Ben Samuel, Noah Wardrip-fruin, and Michael Mateas. 2019. "StoryAssembler : An Engine for Generating Dynamic Choice-Driven Narratives." P. August in *Foundations of Digital Games (FDG)*. San Luis Obispo, CA, USA: ACM Press.

Hill, Olivia, and Filamena Young. 2019. *IHunt: Killing Monsters in the Gig Economy*. Calumet City, IL: Machine Age Productions.

Horswill, I. D. 2016. "Dear Leader's Happy Story Time: A Party Game Based on Automated Story Generation." in *AAAI Workshop - Technical Report*. Vol. WS-16-21-.

Horswill, Ian. 2014. "Architectural Issues for Compositional Dialog in Games." in *AAAI Workshop - Technical Report*. Vol. WS-14-17.

Ingold, Jon. 2015. "Adventure in Text: Innovating in Interactive Fiction." in *Game Developer's Conference*. San Francisco, CA: UBM Techweb.

Joseph, Eugene. 2012. "Bot Colony – a Video Game Featuring Intelligent Language-Based Interaction with the Characters." in *Workshop on Games and NLP (GAMNLP)*. Raleigh, North Carolina, USA: AAAI Press.

Martens, Chris. 2015. "Programming Interactive Worlds with Linear Logic." Carnegie Mellon University.

Mason, Stacy, Ceri Stagg, Noah Wardrip-fruin, and Michael Mateas. 2019. "Lume: A System for Procedural Story Generation." in *The Fourteenth International Conference on the Foundations of Digital Games (FDG '19)*. San Luis Obispo, CA, USA.

Mawhorter, Peter. 2016. "Artificial Intelligence as a Tool for Understanding Narrative Choices: A Choice-Point Generator and a Theory of Choice Poetics." University of California, Santa Cruz.

Montfort, Nick. 2007. "Generating Narrative Variation in Interactive Fiction." University of Pennsylvania.

Nau, Dana, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. 1999. "SHOP: Simple Hierarchical Ordered Planner." Pp. 968–73 in *Proceedings of the 16th international joint conference on Artificial intelligence*. Stockholm, Sweden: Morgan Kaufmann Publishers Inc.

Nelson, Graham. 2006. "Natural Language, Semantic Analysis, and Interactive Fiction." in IF Theory Reader, www.ifarchive.org/if-archive/books/IFTheoryBook.pdf. Accessed 8-11-22.

Osborn, Joseph C., James Ryan, and Michael Mateas. 2017. "Analyzing Expressionist Grammars by Reduction to Symbolic Visibly Pushdown Automata Analyzing Expressionist Grammars by Reduction to Symbolic Visibly Pushdown Automata." in *Intelligent Narrative Technologies (INT)*. Snowbird, UT: AAAI Press.

Reed, Aaron A., Ben Samuel, Anne Sullivan, Ricky Grant, April Grow, Justin Lazaro, Jennifer Mahal, Sri Kurniawan, Marilyn Walker, and Noah Wardrip-fruin. 2011a. "A Step Towards the Future of Role-Playing Games : The SpyFeet Mobile RPG A Step Towards the Future of Role-Playing Games : The SpyFeet Mobile RPG Project." in *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*.

Reed, Aaron A., Ben Samuel, Anne Sullivan, Ricky Grant, April Grow, Justin Lazaro, Jennifer Mahal, Sri Kurniawan, Marilyn Walker, and Noah Wardrip-fruin. 2011b. "SpyFeet : An Exercise RPG." in *Foundations of Digital Games*. Bordeaux, France.

Ryan, James. 2018. "Curating Simulated Storyworlds." University of California Santa Crus.

Ryan, James, Michael Mateas, and Noah Wardrip-fruin. 2016. "Characters Who Speak Their Minds : Dialogue Generation in Talk of the Town Characters Who Speak Their Minds : Dialogue Generation in Talk of the Town." in *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*. Burlingame, CA: AAAI Press.

Ryan, James Owen, Andrew Max Fisher, Taylor Owen-milner, Michael Mateas, and Noah Wardrip-fruin. 2015. "Toward Natural Language Generation by Humans." in *Intelligent Narrative Technologies (INT)*. Santa Cruz, California: AAAI Press.

Ryan, James, Ethan Seither, Michael Mateas, and Noah Wardrip-fruin. 2016. "Expressionist : An Authoring Tool for In-Game Text Generation Expressionist : An Authoring Tool for In-Game Text Generation." Pp. 221–33 in *International Confedrence on Interactive Digital Storytelling (Lecture Notes in Computer Science)*.

Warren, D. H. D., L. M. Pereira, and F. Pereira. 1977. "PROLOG - The Language and Its Implementation Compared with LISP." Pp. 109–15 in *Symposium on AI and Programming Languages*. Vol. 12. ACM.