Generative Text using Classical Nondeterminism

Ian Horswill
Northwestern University, Evanston IL, USA
ian@northwestern.edu

Abstract

Many recent generative text systems combine a context-free grammar base with some set of extensions such as tagging or inline JavaScript. We argue that the restriction to CFGs is unnecessary and that the standard pattern-directed, nondeterministic control structures common to Prolog, definite-clause grammars, and many HTNs, support a superset of these capabilities while still being simple to implement. We describe *Step*, a generative text system for Unity games based on higher-order HTNs that so far as we can determine from published descriptions, generalizes this previous work. We then describe syntactic extensions to make *Step* more natural as a text-authoring language. Finally, we discuss how *Step* and similar systems can be implemented very compactly in modern mainstream programming languages.

Introduction

Most interactive fiction systems, and many video games, use generative text subsystems to dynamically generate dialog and narration. A number of generative text systems have been published in the game AI and IF literatures (Compton, et *al*. 2014; Evans and Short 2014; Garbe et *al*. 2019; Horswill 2014, 2016; Mawhorter 2016; Montfort 2007; Nelson 2006; Reed et *al*. 2011; Ryan, et *al*. 2016).

Although a few of these systems implement full natural language generation pipelines transforming symbolic representations into text, most start from human-authored text and adapt it to context, for example by variable substitution.

These latter systems are the focus of this paper. All implement some generalization of context-free grammars. They can be placed in a rough computational hierarchy, in which each system is a proper superset of those before it:

1. The simplest systems, **template** substitution systems, are less powerful than CFGs. They take a string and substitute the values of variables at specified positions.
2. When a template is allowed to recursively substitute the value of other templates, and alternative versions of templates can be specified, we have a **context-free grammar**.
3. When templates can take arguments and/or return values, we have something closer to Knuth's **attribute grammars** (Aho and Ullman 1977).
4. When the grammar allows equality constraints on those arguments, the result is a **definite-clause grammar** (Pereira and Shieber 1987; Pereira and Warren 1980), or some other unification grammar.
5. When the generator can pass state information from early parts of the sentence to later ones, the result is a first-order, SHOP-style, hierarchical task network **(HTN) planner** (Nau et al. 1999).
6. When templates can take other templates as arguments, one then has a **higher-order HTN planner**.

Most published systems are types 1-3. However, types 4-6 are more general and are all easily implemented using the nondeterministic, pattern-directed, control structure of Prolog (Clocksin and Mellish 2003) and its siblings. Indeed, Prolog was initially invented for natural language processing, and definite-clause grammars are implemented as a Prolog macro. The original SHOP paper (Nau et al. 1999) discusses its similarity to Prolog. SHOP can also be implemented as a Prolog macro.

Unfortunately, the literature on implementing nondeterministic programming languages has few entry points for non-specialists. The Prolog implementation literature from the 80s and 90s (Roy 1993) focuses on complex, high performance systems, using the notoriously opaque Warren Abstract Machine (Ait-Kaci 1991; Warren 1983). The more modern literature from the PL community (Friedman et al. 2018) focuses on using functional programming languages with tail-call optimization and first-class continuations.

Perhaps as a result, many of the text generators in the research literature lack backtracking, equational constraints (unification), or higher-order operators. This paper has 3 goals:

- To argue that nondeterministic programming is the appropriate *internal* representation for CFG-like text generators, regardless of what designer-facing GUI or syntactic sugar is used to aid the authoring process.
- To present *Step*, a text generation language based on higher-order HTNs, which manages to hide much of the Prolog-ishness of the underlying representation from the author.
- To describe how to implement such systems succinctly using in modern, mainstream programming languages.

Step: a Simple Text Planner

Step is a nondeterministic language for writing text generators. It can be used for anything from *Tracery*-style context-free grammars, to very general systems that dynamically inflect nouns and verbs to reflect different cases, voices, tenses, genders, etc. It is written in C# and can be used as a drop-in DLL for Unity projects.

Following Ingold (Ingold 2015), our goal is for the bulk of the text in a script to look like English text with occasional markup, rather than like programming language code with occasional bits of English text. For example:

```
Describe ?who:
  The ?who/Role, ?who, is a ?who/Age+Gender
  with ?who/SalientDetail.  ?who regards you
  intently.
[end]
```

This might expand into text such as:

- The bartender, Olivia Strok, is a 40-something woman with blue eyes. She regards you intently.
- The bartender, Jeff Craig, is an elderly man with crooked teeth. He regards you intently.

This is syntactic sugar for a more overtly code-like form where bracketed expressions represent subroutine calls:

```
Describe ?who:
  The [Role ?who], [Mention ?who], is a
  [Age ?who] [Gender ?who] with
  [SalientDetail ?who]. [Mention ?who]
  regards you intently.
[end]
```

The ultimate internal representation used by the interpreter is roughly all calls:[1]

```
Describe ?who:
  [Write "The"] [Role ?who] [Write ","]
  [Mention ?who] [Write ", is a"]
  [Age ?who] [Gender ?who] [Write "with"]
  [SalientDetail ?who] [Write "."]
  [Mention ?who]
  [Write "regards you intently."]
[end]
```

We will begin by describing the basic language, and then describe the syntactic sugar that allows it to read more like English text.

Core Language

Following the notion of it being an HTN, procedures are named **tasks** rather than predicates (as in Prolog) or non-terminals (as in CFGs). The different ways of achieving a task are called **methods**, rather than clauses (Prolog) or rules (CFGs). Methods can **fail**, in which case the system backtracks and tries the next method. If all methods fail, the task fails, and its caller backtracks. Execution completes when the system finds a series of task calls and methods to implement them such that all methods succeed. **Primitive tasks** can be written directly in C#. **Compound tasks**, are specified by methods decomposing them into subtasks.

Tasks and methods

The simplest *Step* programs implement CFGs:

```
[randomly]
Noun: the cat
Noun: the dog

[randomly]
Verb: bites
Verb: licks
Verb: chases
```

Methods take the form: *task*`:`*text*, and state that a method of achieving the specified *task* is to output the specified *text*. The `[randomly]` annotation states that the subsequent task should try its methods in random order. Thus, `Noun` will randomly generate either "the cat" or "the dog."

Bracketed expressions denote calls whose output are substituted into the output text. Thus:

```
Sentence: [Noun] [Verb] [Noun]
```

produces a random version of "the cat/dog bites/licks/chases the cat/dog".

[1] The true internal representation treats `Write` as a primitive, separate from a call, and also includes a few other non-call primitives such as assignment statements.

Arguments and local variables

Tasks can take arguments, which are pattern-matched to methods using unification. Local variables have names beginning with `?`:

```
Sentence: [Noun animal] ate [Noun ?]

[randomly]
Noun animal: the cat
Noun animal: the dog
Noun vegetable: the carrot
Noun vegetable: the lettuce
Noun mineral: the dirt
Noun ?: nothing
```

The first call to `Noun` will match only its first two methods, while the second will match any, since the variable `?` can match anything, include the `?` variable in the last method. Thus, `Sentence` now generates strings of the form "the dog/cat ate the cat/dog/carrot/lettuce/dirt/nothing".

Predicates

Tasks need not generate textual output. They can be called simply to test whether they succeed, in which case they operate as predicates. They can also be called to bind variables passed in as arguments. For example, if we state:

```
PreferredPronoun bill he:
PreferredPronoun sally she:
PreferredPronoun masha they:
Pronoun ?who:
  [PreferredPronoun ?who ?what]
  [Write ?what]
[end]
```

Then the call `[PreferredPronoun masha ?pro]` will bind `?pro` to "they", while the call `[Pronoun masha]` will output "they" to the text stream.

Generators

Tasks may also generate multiple alternative results, with or without text output. For example, the following generates different fragments of text describing a person, `?who`:

```
[generator]
Detail [BlueEyes ?who] 3: blue eyes
Detail [CrookedTeeth ?who] 2: crooked teeth
Detail ?who 0: nothing remarkable
```

`Detail` takes two arguments: a person (`?who`), and a level of interestingness. Thus, `[Detail ?fred 3]` will attempt to generate a level-3-internestingness detail, while `[Detail ?fred ?i]` will generate the first detail it can, and return its level of internestingnesss by binding it to `?i`.

The bracketed expressions surrounding the first arguments are guards, stating the method doesn't match unless the predicate is true. The first method is equivalent to:

```
Detail ?who 3: [BlueEyes ?who] blue eyes
```

If `?who` doesn't have blue eyes, the call to `BlueEyes` will fail and the system will move on to the next method.

Finally, the `[generator]` annotation states that `Detail` is fully non-deterministic. It will not stop with the first solution it finds, but attempt to generate further solutions should the first be rejected. We will see this used below.

Global state variables

In addition to text output and unification of local variables, tasks can also update state in the form global variables. Unlike `?` variables, which are local to their methods and bind through unification, global variables are Capitalized and are explicitly updated using assignment statements.

For example, the `Mention` task generate names or pronouns, as appropriate, by storing discourse context in the global variables `LatestThey`, `LatestShe`, etc.:

```
Mention LatestThey: they
Mention LatestShe: she
Mention LatestHe: he
Mention ?who:
  [Write ?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]
```

Since mention is not marked with `[randomly]`, it will try its methods in the order listed. The first three match the argument to the current discourse context (`LatestShe`, etc). The fourth is a catch-all that is attempted only when a pronoun is impossible. It prints the name of the character and then updates the discourse context.

Assignments to variables are undone upon backtracking, as are unifications and text output. The final text output and variable bindings show only the choices made in the final, successful execution path.

Equivalent versions of `Mention` can be written to generate object case (him/her/them), possessive (his, her, their) and reflective (himself, herself, themselves) pronouns, respectively. Alternatively, they could be packaged as a single task that takes two arguments, the object to output, and the case in which to inflect it. This can also be extended to support more complex inflections when generating text for languages with more sophisticated case systems than English. For a general discussion of noun/verb agreement and feature marking using matching, see (Covington 1993).

Higher-order tasks

Finally, tasks can take other tasks and task expressions as arguments, allowing the introduction of iteration and aggregation constructs.

For example the built-in primitive, `Max`, allows the author to score different alternative texts and choose the most desirable one. This is how the `SalientDetail` task referred to earlier works:

```
SalientDetail ?who:
  [Max ?salience [Detail ?who ?salience]]
```

Here, `Max` calls `Detail`, but repeatedly forces it to backtrack, generating new solutions, until all possible solutions have been generated. Along the way, it checks the value of `?salience` and remembers the solution with the highest salience. It then continues execution of the rest of the program, using the highest-scoring solution. The program behaves as if that solution, and only that solution, had been run; pronoun bindings and any other state information are appropriate for the text that is displayed.

Syntactic Sugar

Although the core language is powerful, the many bracketed expressions break up the flow of the text, leading to text that looks like code rather than marked-up English. Although this is unavoidable for tasks like `Mention` that require control structure and state manipulation, the bulk of an interactive fiction work would consist of straightforward chunks of text that call into a few complex tasks like `Mention`.

By adding a small amount of syntactic sugar, we can allow this simpler, template-like, code to make simple calls without the use of bracket expressions:

- `?variable`
  Expands into: `[Mention ?variable]`
- `?variable/Task`
  Expands into: `[Task ?variable]`
- `?variable/Task1+Task2`
  Expands into:
  `[Task1 ?variable] [Task2 ?variable]`
  As many tasks as desired can be included.
- `?variable/Relation/Printer`
  Expands into:
  `[Relation ?variable ?t] [Printer ?t]`
  Where `?t` is a temporary variable.

The last form is best explained by an example. Suppose there is a predicate called `Brother` that succeeds whenever its two arguments are brothers. Then the text:

```
?who has a brother, ?who/Brother/Name
```

might generate something like "Sharon has a brother, Bill," the call to `Brother` having bound `?t` to Bill, and the call to `Name` having printed it. If this expression appears inside of a backtracking construct, such as `Max` or `ForAll`, and Sharon has multiple brothers, then `Brother` will generate successive brothers, and they will each be `Named`. Multiple relations can be stacked, as in:

```
?who/Spouse/Brother/Name
```

which would print the name of `?who`'s brother-in-law. Finally, since the "`/Name`" in the construction is awkward, it can be omitted, in which case the system will call `Mention` on the brother. Thus:

```
?who has a brother, ?who/Brother
```

expands into the equivalent code:

```
[Mention ?who] has a brother, [Brother ?who
?t] [Mention ?t]
```

Example

Let's write a more elaborate version of `Mention` that further adapts referring expressions to context. We'll start by changing the final rule for `Mention` to call `FirstMention` the first time `?who` is mentioned, and call `Subsequent` thereafter:

```
Mention ?who:
  [case ?who] Mentioned : [Subsequent ?who]
              [else] [FirstMention ?who]
                 [add ?who Mentioned] [end]
  [PreferredPronoun ?who ?pro]
  [Update ?who ?pro]
[end]
```

`FirstMention` and `Subsequent` can be customized however one needs. In this case, we'll have `FirstMention` generate full name ("Bill Taylor") and `Subsequent` generate something shorter ("Bill", "Mr. Taylor" or "Taylor"):

```
FirstMention ?who: ?who/FirstName+LastName
Subsequent [Familiar ?who]: ?who/FirstName
Subsequent ?who: ?who/Honorific+LastName
Subsequent ?who: ?who/LastName
```

`Subsequent` generates the first name for characters who are on familiar terms with the narrator, otherwise honorific and last name, when possible, or last name if no honorific is known. We might determine the honorific like this:

```
[fallible]
Honorific [PhD ?who]: Dr.
Honorific ?who: ?who/PreferredPronoun/Honor
[fallible]
Honor he: Mr.
Honor she: Ms.
```

Since not all characters have honorifics, the `[fallible]` annotation declares that `Honorific` failing isn't considered an error. If `Honorific` fails, it just triggers backtracking so that `Subsequent` moves on to its next method.

This is an important point. Nondeterministic control frees the author to call fallible tasks *as if* they will work, knowing the system will automatically clean up and try something else should they fail. Without it, callers such as `Subsequent` would need elaborate checks to test whether it was safe to call the fallible task, and those checks would need to be found and updated each time the task was modified.

Implementation

Step is an interpreter written in C#. Each nondeterministic operation (trying a method, calling a task, etc.) takes as an argument a current *state*, which contains everything that can be changed by the operation (text generated, variable bindings). When successful, the task produces a new state, perhaps with additional text or bindings.

This requires that states be represented as immutable data. When we make a new state, we preserve the original one in case we backtrack. Note that nondeterministic languages obey strict stack discipline, so states can be stack allocated when implemented in languages that support it.

Control structure

Since nondeterministic operations can generate zero, one, or many states, they generate candidate states one by one, feeding them to a *continuation*, a callback that approves or rejects the new state. As soon as the continuation returns true for a generated state, the operation returns true. If the operation runs out of new states without the continuation returning true, it returns false, meaning the operation failed.

A simplified version of the core interpreter is as follows. A `CompoundTask` is an object containing a set of `Method` objects and a C# method called `TryTask`:

```
bool TryTask(object[] args, State s,
             Continuation k) {
  foreach (var m in Methods)
    if (m.TryMethod(args, s, k))
       return true;
  return false;
}
```

This tries each method, allowing it to generate new states (new text, bindings) and pass them to the continuation. If the continuation accepts one of the states generated by a method, the method immediately returns true and so does the task. If method fails (returns false), the task tries subsequent methods until some method is successful. If no method is successful, the task fails and returns false.

A `Method` is an argument pattern to match the arguments against, together with a linked list of `Calls` to different tasks. Its `TryMethod` method looks like:

```
bool TryMethod(object[] args, State s,
               Continuation k)
  => s.MakeFrame().Unify(args, Pattern,
                         out newState)
     && FirstCall.TryCall(newState, k));
```

`MakeFrame` makes fresh logic variables for the method's arguments, resulting in a new state. The method then unifies its pattern with its arguments, resulting in a new binding list, and hence another new state. It then tries the first call in its chain, passing it the continuation and new state. If either unification or call fails, the method fails.

A `Call` has fields `Task` (task to call), `Arguments` (to pass to it), and `Next` (the next `Call` in the method). It also has its own `TryCall` method, which will attempt to recursively call `TryTask`, passing a continuation that invokes either the next call in the method's chain of calls, or, if we're at the end of the chain, its own continuation:

```
bool TryCall(State s, Continuation k) {
  => Task.TryTask(
       Arguments, s,
       ns => (Next == null)
             ? k(ns) : Next.Try(ns, k));
```

Finally, the game's C# code can invoke a *Step* task by calling its `TryTask` method and passing it a continuation that saves the final state and returns true:

```
string Call(Task t, object[] args) {
  State result = null;
  t.TryTask(args, EmptyState,
         s => { result = s; return true; })
  return result.Output;
}
```

Alternatively, it could return the entire `State`, including variable bindings, allowing the game to retain discourse context from call to call. It can also be used to get information about the generated utterance, implement a version of *Expressionist*'s content bundles (Ryan, et *al*. 2016).

The code above is highly simplified. The real version of `TryTask` tries `Methods` in random order when the task is

tagged with [*randomly*]. It also passes TryMethod a continuation that counts the number of solutions, since non-[*generator*] tasks are easier to debug if they don't generate multiple solutions, and non-[*fallible*] tasks are easier to debug if they throw exceptions upon failure.

The real version of TryCall includes a check for whether the task to call is a primitive task in which case its C# implementation is called directly. The real system also passes the state as separate arguments for text output and variable bindings rather than as a unified State object.

That said, we hope it is clear from the foregoing that implementing a full nondeterministic system need not be difficult. The three C# methods comprising the simplified core interpreter is 17 lines of code. The most sophisticated programming technique used is λ expressions, which are now quite commonplace.

Related Work

While *Step* is not intended to be a full interactive finction language, there are nonetheless a number of interactive fiction languages that both generate text and also use some kind of declarative 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). *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.

Another IF language, *Curveship* (formerly known as *nn*), uses a full natural language generation pipeline to transform those 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 rather than a full NLG pipeline.

Other games and generative text systems use some variant of CFGs. *Tracery* (Compton et al. 2014; Compton and Mateas 2015) is an extremely successful CFG-based generator designed for casual users. It augments the base CFG with the ability to call arbitrary JavaScript code. *Expressionist* (Osborn et *al*. 2017; Ryan et al. 2015; Ryan et *al*. 2016; Ryan et *al*. 2016) 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, but it appears to be similar to an HTN in which a tag flips a bit in the state output. *Dear Leader's Happy Story Time* (Horswill 2016) used a higher-order HTN similar to *Step*, but was implemented as an embedded language in Prolog, making it considerably less accessible.

Dunyazad (Mawhorter 2016) uses a template/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.

Step supports a strict superset of the functionality of the CFG-based systems, yet it is surprisingly simple. Its source code is 17% smaller than *Expressionist*'s.

Conclusion

Step allows developers to add flexible text generation to any Unity game. It is not a self-contained IF engine such as *Twine* or *Inform*, nor is it a causal creator such as *Tracery*. It is intended for adaptive delivery of hand-authored dialog and narration with a minimum of pain and a maximum of flexibility.

Step allows simple use cases to be written simply; a *Tracery*-style CFG can be written without learning anything Prolog-like. More sophisticated features such as pronominalization or list aggregation can be added easily by writing a small amount of code, or by using a library written by others. Since the features are written in *Step* code, they can be adapted for a given game without changing the interpreter.

Step does have limitations. While it is can be used to inflect words in languages with sophisticated case systems, the resulting scripts look less like story text and more like Prolog or SHOP code. So *Step* is not a solution to the problem of localization to multiple languages.

Finally, *Step* demonstrates that full pattern-directed, nondeterministic programming can be implemented nearly as easily as the simpler CFG systems. We strongly recommend the use of pattern-directed invocation in similar projects.

Acknowledgements

I would like to thank the reviewers, Ethan Robison, Rob Zubek and Matt Viglione for advice and comments, and Ethan for being willing to kick the tires on the system.

References

Aho, Alfred V. and Jeffrey D. Ullman. 1977. “Principles of Compiler Design (Addison-Wesley Series in Computer Science and Information Processing).”

Ait-Kaci, Hassan. 1991. *Warren’s Abstract Machine: A Tutorial Reconstruction*. Cambridgte, MA: MIT Press.

Clocksin, William F. and Christopher S. Mellish. 2003. *Programming in Prolog: Using the ISO Standard*. 5th ed. New York, NY: Springer.

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*.

Covington, Michael A. 1993. *Natural Language Processing for Prolog Programmers*. Prentice Hall.

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

Friedman, Daniel P., William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer*. 2nd editio. MIT Press.

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.

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.

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.”

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.

Pereira, Fernando C. N. and Stuart Shieber. 1987. *Prolog and Natural Language Analysis*. Brookline, MA: Microtome Publishing.

Pereira, Fernando C. N. and David H. D. Warren. 1980. “Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks.” *Artificial Intelligence* 13(231–278).

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.

Reed, Aaron A., Ben Samuel, Anne Sullivan, Ricky Grant, April Grow, Justin Lazaro, Jennifer Mahal, Sri Kurniawan, Marilyn Walker, and Noah Wardrip-Fruin. 2011. “A Step Towards the Future of Role-Playing Games: The SpyFeet Mobile RPG Project.” in *Proceedings of the Seventh Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-11)*. Stanford, CA.

Roy, Peter Van. 1993. *1983–1993: The Wonder Years of Sequential Prolog Implementation*. Paris, France.

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, David H. D. 1983. *An Abstract Prolog Instruction Set*. Menlo Park, CA.