Bronco: A Universal Authoring Language for Controllable Text Generation

Jonas P. Knochelmann[1][0000−0001−6314−8617] and Rogelio E. Cardona-Rivera[1,2]

[1] School of Computing
[2] The Entertainment Arts and Engineering Program
University of Utah, Salt Lake City, UT, USA
jonas.p.knochelmann@gmail.com, rogelio@eae.utah.edu

Abstract. We present Bronco: an in-development authoring language for Turing-complete procedural text generation. Our language emerged from a close examination of existing tools. This analysis led to our desire of supporting users in specifying *yielding grammars*, a formalism we invented that is more expressive than what several popular and available solutions offer. With this formalism as our basis, we detail the qualities of Bronco that expose its power in author-focused ways.

Keywords: text generation · authoring tools · programming language · grammars

1 Introduction

We presently enjoy a plethora of grammar-based tools for text-generation [6,8,11, 16,18,23]. Broadly, these tools share the aim of maximizing *authorial leverage*—i.e., "the power a tool gives an author to define a quality interactive experience in line with their goals, relative to the tool's authorial complexity" [2]. Available tools maximize leverage via different means [10] and each one strikes a different balance between four dimensions of *authorability*: (1) the *technical proficiency* expected of the end user, (2) the *complexity* an end user must contend with to author content, (3) the *clarity* the system offers the end user about the system's state and dynamics during authorship, and (4) the *controllability* the system offers the end user for generating content as they intend.

Despite the cornucopia of tools that explore the design space of authorability, these are effectively powered by context-free grammars (CFGs). We contend that, as a result, the tools in this space mostly explore circumscribed variations on procedural text generation. In view of this gap, we designed Bronco (Fig. 1), a generator that offers greater controllability than existing grammar-based tools.

Bronco affords this controllability by letting users author *yielding grammars*, a novel formalism we have defined that is Turing-complete. Bronco offers a large selection of expressive features that permit precise control in text generation. The language currently supports features useful to our design practice and we expect to refine its specification going forward. Although still nascent, we are emboldened by its promise so far.

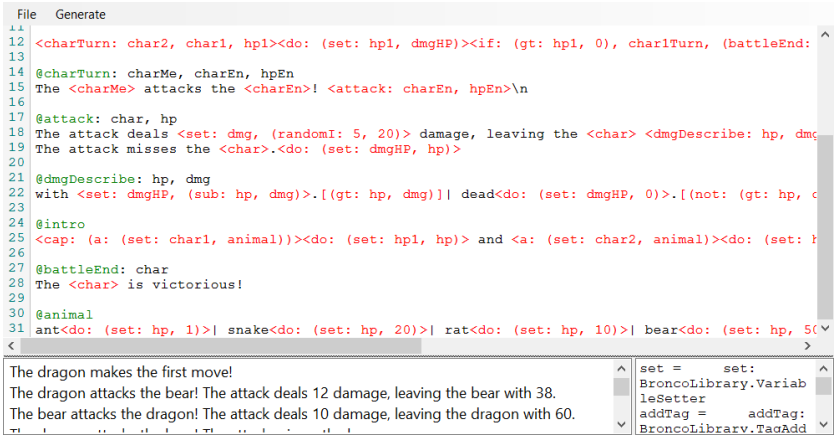

Fig. 1. A screenshot from Bronco IDE with code describing the grammar (top), a sample output (bottom-left) and some debug information (bottom-right).

In this paper, we detail Bronco's design. We focus on its underlying formalism in order to mathematically contrast it against two tools closest to us: Tracery [6] and Expressionist [23]. We also discuss the expressive features the language offers, and present a small case study in our experience writing grammars for both Tracery and Bronco.

2 Literature Review and Background

Bronco is a *universal* text-generation authoring language—a language capable of specifying Turing-complete grammars that can yield finely-controllable templated text; both as a stand-alone tool and as part of a larger system.

Within the design space of authorability, Bronco is Tracery-like in author-focus and Expressionist-like in grammar-specification, but is more flexible than both. To demonstrate how, we rely on the following formalisms and concepts.

2.1 Baseline Formalism: Context Free Grammars

Context-free grammars (CFGs) are a restricted subset of formal grammars notable for being the backbone of many text and story generation systems [1]; they are also a Turing-incomplete formalism [24].

A CFG is a construct that formally describes how to form strings from a (typically finite) set of grammar symbols Σ called an *alphabet*. This formal description is specified as a set of hierarchically-nested rules P called *productions*, which relate a single *non-terminal symbol* (ones in the alphabet that are designated as decomposable) to a finite sequence of *other* grammar symbols; non-terminal symbols $N \subset \Sigma$ in the sequence can be successively decomposed, whereas *terminal symbols* $T \subset \Sigma$ are elementary tokens.

The set of strings that can be formed from a given CFG is called a *language*. All such strings begin with the distinguished *start symbol* $n_{\text{start}} \in N$, which is special in that it never appears in the finite sequence of a production. Formally:

Definition 1 (Context-free Grammar). *A quadruple* $G = \langle N, T, P, n_{\text{start}} \rangle$. N, T, P, *and* n_{start} *are the non-terminal symbols, terminal symbols, productions, and start symbol as described earlier. An element in* $\Sigma = N \cup T$ *is called a* grammar symbol*, and* $N \cap T = \varnothing$*. We refer to the elements of* T *and* N *as* terminals *and* non-terminals*, respectively.*

Each production $p \in P$ *is a pair of the form* $p\colon X \to \alpha$*, where* $X \in N$ *is the* left-hand side *and* $\alpha \in \Sigma^*$ *is totally ordered set of 0-or-more grammar symbols called the* right-hand side*. Applying* p *to substitute the left-hand side for the right-hand side is called a* derivation*.*

Finally, the start symbol never appears on the right-hand side of a production (i.e. $\forall p \in P\colon n_{\text{start}} \notin \alpha$*). The language* $\mathcal{L}(G)$ *is obtained by starting with* n_{start} *and recursively deriving symbols* $\forall p \in P$ *until only terminals remain; this recursive sequence of derivations is a* trace *and can be visualized as a* parse tree*.*

Figures 2 and 3 illustrate an example CFG and a sample trace of it visualized as a parse-tree, respectively.

$$
\begin{aligned}
\mathit{expr} &:= \mathit{expr}\ \texttt{+}\ \mathit{term} \mid \mathit{expr}\ \texttt{-}\ \mathit{term} \mid \mathit{term} \\
\mathit{term} &:= \mathit{term}\ \texttt{*}\ \mathit{factor} \mid \mathit{term}\ \texttt{/}\ \mathit{factor} \mid \mathit{factor} \\
\mathit{factor} &:= \textbf{digit} \mid \texttt{(}\ \mathit{expr}\ \texttt{)}
\end{aligned}
$$

Fig. 2. An example CFG in extended Backus-Naur form for arithmetic expressions. The non-terminals *expr*, *term*, and *factor* help capture precedence and associativity of the operators `+`, `-`, `*`, and `/`.

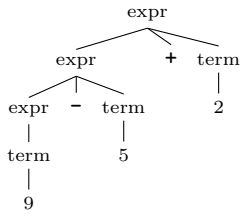

Fig. 3. Parse tree for a trace of the expression `9-5+2` via the CFG in Fig. 2.

2.2 Grammar-based Procedural Text Generators

CFGs are powerful, but by default are too rigid to support fluid text-based authoring within interactive digital narratives. In fact, despite their marketing as CFG-powered tools, Tracery and Expressionist deal with a formalism more powerful than context-free grammars; namely, *attribute grammars* [21].

Attribute Grammars An attribute grammar (AG) augments a CFG by supplying two additional constructs: *attributes* and *semantic rules*.

Informally, an attribute can be thought of as a variable in a programming language. Each attribute a can take on a range of values V; for example, a can be an integer number (making $V = \mathbb{I}$), a rational number ($V = \mathbb{R}$), or a string literal ($V = [\texttt{A-Za-z0-9}]^+$). A semantic rule is what specifies which value in V a given attribute actually takes.

Every grammar symbol $X_i \in \Sigma$ has an associated set of attributes $A(X_i) = \{a_0, \ldots, a_m\}$ ($m \geq 0$) whose values collectively represent what X_i *means*. In turn, every production $p\colon X_0 \to X_1 X_2 \cdots X_n$ ($n \geq 0$) has an associated set of semantic rules R_p that specify how an attribute $X_i.a$ gets its value, for all $a \in A(X_i)$. For example, the CFG from Fig. 2 can be augmented as illustrated in Fig. 4, which defines a *value* attribute for each of the non-terminals. Said *value* is manipulated via the semantic rules next to each production.

$$
\begin{aligned}
expr &:= expr_1 \texttt{ + } term && \{\ expr.value = expr_1.value + term.value;\ \}\\
&\mid expr_1 \texttt{ - } term && \{\ expr.value = expr_1.value - term.value;\ \}\\
&\mid term && \{\ expr.value = term.value;\ \}\\
term &:= term_1 \texttt{ * } factor && \{\ term.value = term_1.value * factor.value;\ \}\\
&\mid term_1 \texttt{ / } factor && \{\ term.value = term_1.value / factor.value;\ \}\\
&\mid factor && \{\ term.value = factor.value;\ \}\\
factor &:= \textbf{digit} && \{\ factor.value = \mathsf{valueOf}(\textbf{digit});\ \}\\
&\mid \texttt{(} expr \texttt{)} && \{\ factor.value = expr.value;\ \}
\end{aligned}
$$

Fig. 4. An AG that augments a CFG in extended Backus-Naur form with semantic rules on the right-hand side. As derivations are applied, the *value* of expressions, terms, and factors gets iteratively computed based on the semantic rules.

Every set of attributes $A(X_i)$ can be partitioned into two disjoint subsets: the *inherited attributes* $I(X_i)$ and the *synthesized attributes* $S(X_i)$. Whether $a \in A(X_i)$ belongs to $I(X_i)$ or $S(X_i)$ depends on how its value is determined. If a is determined from attribute values at X_i or within derivations of X_i (i.e. children of X_i in the parse tree), then it is *synthesized*; $a \in S(X_i)$. For example, the parse tree in Fig. 5 contains *only* synthesized attributes. However, if a is determined from attribute values within derivations that *contain* X_i (i.e. parents of X_i in the parse tree), then it is *inherited*; $a \in I(X_i)$.

We require attribute grammars to be *well-formed*, which means that dependencies between attributes must be acyclic. This constrains each attribute in the grammar to be either synthesized or inherited-with-constraints. The inheritance constraints manifest in the semantic rules, which should not introduce a circular dependency between attributes. For a given production $p\colon X_0 \to X_1 X_2 \cdots X_n$ with inherited attribute $X_i.a$ computed by a rule $r \in R_p$, r is constrained to use only: (a) inherited attributes associated with X_0, or (b) attributes associated with $X_1, X_2, \ldots, X_{i-1}$ (symbols to the left of X_i). Formally, an AG is thus:

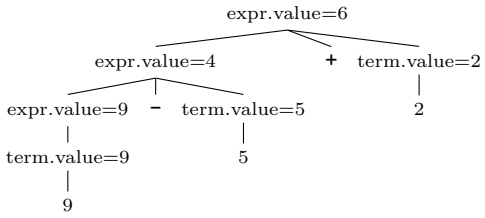

Fig. 5. Parse tree for a trace of the expression `9-5+2` via the AG in Fig. 4. As the trace happens, the *value* of intermediate expressions is recursively computed until the full expression is parsed with a *value* of 6.

Definition 2 (Attribute Grammar). *A triple* $AG = \langle G, A, R \rangle$*. G is a CFG as in Def. 1, A is a set of* attributes*, and R is a set of* semantic rules*.*

The set A is obtained from the attributes associated with each symbol X_i *in the grammar, and the set R is obtained from the semantic rules associated with each production* $p \in P$ *of the grammar. Letting* $\Sigma = N \cup T \in G$*:*

$$A = \bigcup_{X_i \in \Sigma}^{G} \{A(X_i)\} \qquad \text{and} \qquad R = \bigcup_{p \in P}^{G} \{R_p\}$$

Each R_p *is a set of rules of the form* $X_i.a = \mathtt{f}(y_1, \ldots, y_k), k \geq 0$*, such that all the following is true:*

(1) either (a) $i = 0$ *and* $a \in S(X_0)$ *or (b)* $i > 0$ *and* $a \in I(X_j), 1 \leq j < i$*;*
(2) each y in `f` *is an attribute associated with some symbol in production p; and*
(3) `f` *is the* semantic function*—a map from values* $y_1, \ldots, y_k$ *to value* $X_i.a$*.*

Tracery and Expressionist: Attribute Grammar Metalanguages In view of the preceding formalisms, Tracery and Expressionist are *metalanguages* that facilitate authoring a unique kind of attribute grammar: a probabilistic one [15]. In essence, a probabilistic attribute grammar (pAG) indirectly assigns to each element in $\mathcal{L}(AG)$ some probability of being generated as part of a trace. The probabilities are directly specified as attachments to productions: if a derivation involves a choice between one or more productions to apply for the same non-terminal, a production is chosen based on its assigned probability.

Tracery is Effectively CFG-based. Tracery affords authoring a restricted form of pAG's: the choice between productions for the same non-terminal is done randomly, making individual derivations for the non-terminal equally probable.

The Tracery metalanguage can be written in JSON; the one in Fig. 6 defines an attribute grammar where the terminals are defined by symbols in string quotes (e.g. "light") and non-terminals are defined by symbols within number signs (e.g. `#move#`). Productions are compactly written as name/value pairs delimited by commas. For example, the line "`path`":["stream", "brook", "path",

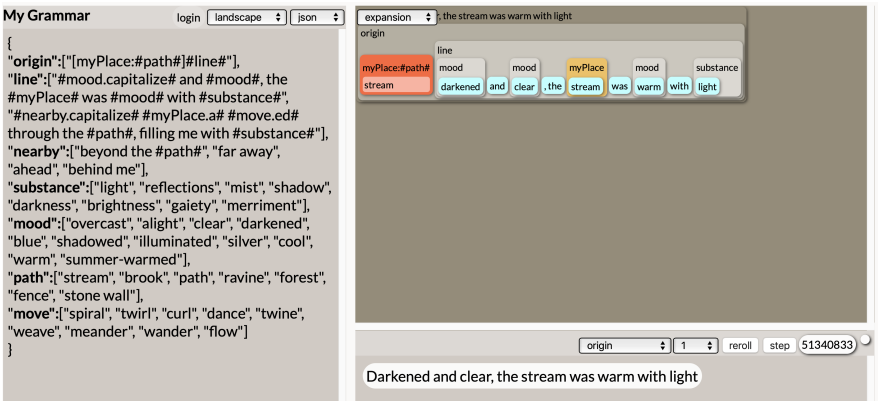

Fig. 6. Tracery's online editor, displaying a JSON-encoded attribute grammar (left) with one inherited attribute (highlighted, right). Productions are delimited by commas, terminals are quoted string literals, and non-terminals are surrounded by the '`#`' character (for example, `#mood#` is a non-terminal in the `line` production).

"ravine", "forest", "fence", "stone wall"] is a specification of 7 productions; i.e. `path`→"stream", `path`→"brook", `path`→"path", and so on. Further, the start symbol is always `origin`. Finally, the example in Fig. 6 defines an inherited attribute called `myPlace`, which is generated from selecting a random production (of the 7 applicable) to `path`. This inherited attribute is then used during the derivation of the `line` non-terminal, as illustrated in in Fig. 7.

Tracery defines several built-in semantic functions beyond assignment of attribute values; e.g. the Fig. 6 grammar uses the `capitalize` function to modify the attribute values it obtains. While there are other such functions in the metalanguage, Tracery does not support defining new functions within its grammars.

As a result, attributes in a Tracery-language grammar are limited to taking values only from finite domains (i.e. productions that randomly generate a value from a list) and whenever the same attribute is used across derivations, its value must also be the same. *This makes Tracery-language grammars more compact to specify than traditional CFGs, but still only as expressive as CFGs* [17].

Expressionist is Potentially Turing-complete. Expressionist affords a finer-degree of control in text generation than Tracery; it requires a user to directly specify the probability distribution between the productions they write. Interestingly, *Expressionist's expressive power depends on whether it is running as a stand-alone authoring tool or as a tool within a host environment capable of generating derivations for grammars written in the metalanguage.*

When running via a stand-alone tool, the Expressionist metalanguage is as expressive as Tracery, albeit with affordances that make its grammars more compact than Tracery's. Formally, all non-terminals $X_i \in N$ in an Expressionist-language grammar have two special attributes: the `tagset` $\in I(X_i)$ and the `tags` $\in S(X_i)$. The value of X_i.`tags` is a string literal set by the author, intended

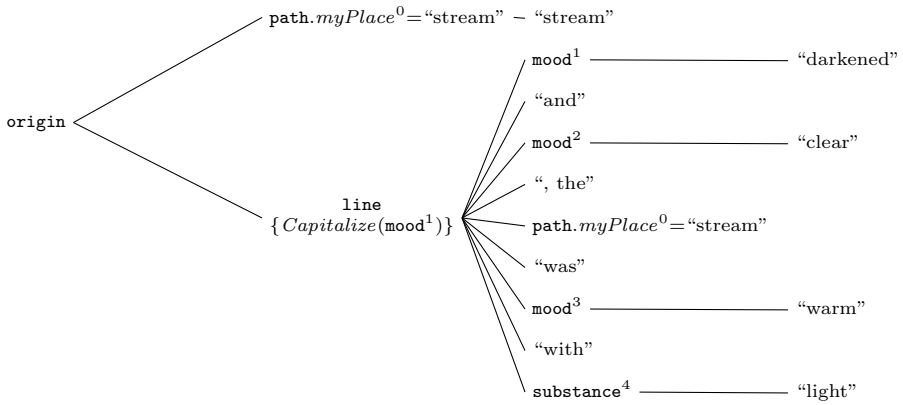

Fig. 7. A parse tree for a trace of the Tracery-language grammar in Fig. 6, read from top to bottom (derivations closer to the top happen earlier). Tracery supports randomly-generated attributes that are inherited elsewhere in the tree (e.g. `path`.*myPlace*, above).

to represent a semantic annotation of interest. The value of X_i.`tagset` is a string literal set by the Expressionist tool during a trace. Every time a production $p\colon X_0 \rightarrow X_1 X_2 \cdots X_n$ is applied, X_0's `tags` are simply concatenated to the `tagset` of $X_1 X_2 \cdots X_n$.

When running within a host environment (e.g. a game world), Expressionist is *potentially* Turing-complete; it depends on whether the host is (1) capable of parsing X_i.`tags` or X_i.`tagset` as programming instructions for the environment, and (2) the environment itself is Turing-complete. If both conditions are met, Expressionist offers a powerful metalanguage akin to a probabilistic programming language [22]. In this mode, an attribute $X_i.a$'s values are non-deterministically dependent on the host's state, and *vice-versa*. This power comes at the cost of needing to engineer a non-trivial amount of machinery to enable it, which presents a considerable trade-off in its use.

Other Related Text Generators The remaining tools we discuss in this section are all less expressive than Bronco and/or further afield in their design than either Tracery or Expressionist. However, the ones we mention next have particular features similar to our system. Bronco is like Ink [13, 14] – a narrative scripting language for game developers – in that it affords quick access to word-by-word variation. Further, Bronco is like Blabbeur [18] – a CFG-powered language for text generation specifically designed for development in Unity – in that it affords easy and explicit limiting of a sub-structures within the grammar.

Finally, MKULTRA [12] and Dunyazad [20] are project-specific tools for text generation. Notably, they rely on *definite-clause grammars* rather than CFGs (making them Prolog-like in nature). As such, they are as expressive as Bronco. However, they are less focused on offering author-focused features that support a person's authoring experience.

2.3 Problem Statement: The Motivation for Bronco

There remains a gap somewhere beyond Expressionist and Tracery in terms of controllability. On the one hand, Expressionist is potentially-costly to implement while also bound in expressivity by its host environment. On the other hand, Tracery is a more-compact alternative but effectively rigid in generative capability. We are left with a motivating question: might there be an alternative?

Bronco emerged in our pursuit of an answer to that question, and we describe its architecture next.

3 Bronco

3.1 Design Considerations and Qualitative Rationale

We propose to explore greater controllability for procedural text generation via a Turing complete formalism. This aim, combined with our observation of the successes and shortcomings of the tools discussed in the previous section, led us to the following list of requirements we believe our tool must fulfill to push the boundaries of authorability toward greater authorial leverage. Based on our analysis, the tool...

1. *...should be easily integratable with other systems in an existing project.* Text generation is, more often than not, a relatively small part of a project's experience. Thus, we cannot expect practitioners to base their code's architecture around the generator; the tool should ideally work in any architecture.
2. *...should provide many ways to easily limit the output of the generator.* The simplest possible CFG derivation system provides no control at all over the output, and most of the unique features in existing tools provide various ways of limiting the random possibility space.
3. *...should have in built, general-purpose logic.* No matter how many pre-built features a tool comes with, there will always be certain things it is unable to do. A tool's general-purpose logic should support users to program their own functionality, for the cases when pre-built functionality falls short.
4. *...should be easily extensible.* Similar to the previous point, but focused on how the general-purpose logic of the language manifests. Ideally, the tool should support the development of features via its own language or an industry-grade programming language.
5. *...should be accessible to non-programmers.* This requirement is one that many existing tools have tried to fulfill. Because writing makes up such a large portion of creating a text generator, it stands to reason that professional writers should be able to use the tool, not just programmers.
6. *...should be easier to use than implementing a generator from scratch.* Arguably the most important, one that all existing tools have succeeded in. However, it must be carefully counter-balanced with the goal of making it author-focused and easy to use.

With this list of requirements laid out, we may construct an image of what such a tool would look like. In order to be easily integrated and extended upon, we decided our tool should exist at a library level in a language that is commonly used in development, and game development in particular. Further, this library should simultaneously exist as a standalone portion of the code base and have many points where programmers are able to inject external code.

In order that users of the tool have fluent access to limiting the generator's output, general-purpose logic, and that the tool be easily accessible to non-programmers, the generators should be authored via a custom-built, text-based programming language. This language could then be stored in separate files such that non-programmers could edit them without entering the code base. The general-purpose logic and output limiting then could be accessed in a similar way to simple general-purpose programming languages. Having access to a miniaturized general-purpose language within the generator language would also make it easier to access features extended from the base language. The aforementioned library then would be responsible for parsing this language and producing a generator runnable in the implementing language.

To meet the final point, both the language and the library need to be designed with the end-user in mind. For the language, this means a minimal syntax that gives very quick access to the design patterns most commonly used in a text generator. For the library, whatever complexity exists under the hood, a limited set of interactions should be exposed to the end programmer.

We now have a fair idea of what a generative text tool for game development should look like: It should be a library in a popular language that parses a custom-built programming language both of which are designed for usability.

3.2 A Formal View Of Bronco

The formalism underlying Bronco is a variant of attribute grammars we're calling *yielding grammars* (YG). Programmable, this system defines *symbols* as objects that provide a function definition for *yield*, which takes zero or more symbols as arguments, must return another symbol. The only other requirement is that successively calling the yield function on the return value of the previous yield will eventually result in a *terminal symbol*, which is a defined as a symbol that yields itself. This process of repeated yielding is called *flattening*.

This formalism is somewhat similar to the Lambda Calculus, in that it is entirely made up of function-like objects [3]. Indeed, it was designed with the hope that it would be used in a functional manner. It differs only in the fact that symbols may not be deterministic, and may have an internal state that can be assessed and modified by other symbols.

This formalism, which was synthesized iteratively during development, has proven invaluable in the construction of Bronco. It benefits both the end-user programming in the language and the developer writings extensions to the language in addition to providing Turing-completeness. For the end-user, it removes the distinction between calling a function and referencing a function, and most of the difficulties that come with a type system: since writing in the language

is effectively constructing a tree, users can rely on the flattening algorithm to collapse the tree into a final output text without considering when exactly each branch of the tree is flattened. If a symbol requires a certain type for one of its arguments, for example, the flattening algorithm will flatten that argument's branch until the specified type is reached, throwing an error if it reaches a terminal first. For developers extending Bronco with custom symbols, they can again lean on the flattening algorithm. They are only required to define an evaluation, and that evaluation only needs to provide a symbol that is one step closer to a terminal.

Formally, yielding grammars are defined as follows:

Definition 3 (Yielding Grammar). *An attribute grammar* $Y = (G, A, R)$ *as defined previously, with the following restrictions:*

Every production $p \in P$ *of the grammar, must have the form* $x \to y$ *or* $x : p_0, p_1, ... \to y$ *where* $x \in N$ *and* $y, p_0, p_1, ... \in N \cup T$. $p_0, p_1, ...$ *are called* parameters*, and are treated by* R_p *as if* $(x, p_0, p_1, ...)$ *is a single symbol. (e.i. the attributes of* $x, p_0, p_1, ...$ *are all defined by* R_p *as well as* y*'s) Within this, all productions must be* inherited*.*

3.3 Authoring in Bronco

As a piece of software, Bronco is a toolset consisting of a custom-built programming language for authoring procedural text generators, and a highly extensible C# library that can parse the Bronco language, and output the generated text. The library portion of the toolset is largely backend that models the internal grammar. It fronts the parser, as well as a range of objects to be derived, and a way to inject derived objects into the parser. In addition to these, a small authoring tool simply titled "Bronco IDE" has been built to make authoring in Bronco as easy as possible (See Fig.1).

Bronco is atomically made up of symbols. All data types in Bronco are symbols including, numbers, text, and functions. The language itself has taken heavy inspiration from Markdown and the minimalist syntax of Ink [7, 13]. Like Tracery, and Blabbeur, Bronco grammars are defined in terms of random production rules, which are called *bags* [6, 18]; a type of symbol. Each bag is made up of a title (denoted '`@`' followed by the bag's title; e.g. `@start`), and a number of items (written as normal text separated by newlines or '`|`'), one of which will be picked at random upon a reference to the bag's title (denoted by the bag's title surrounded by '`<>`'). In addition to user-defined bags, Bronco comes with a large number of other symbols that can be easily referenced in the same format, and provide other functionality. These predefined symbols can also be easily added to via the C# library, which is the primary method of adding custom functionality to the language. Fig. 8 shows basic Bronco syntax.

When defining a bag, you may include arguments that can then be referenced by the bag's item (denoted with '`:`', followed by arguments separated by '`,`'). Similarly, most of the predefined symbols also take one or more arguments. Arguments can then be added to any symbol reference, which modifies the way

```
@start
My favorite color is <color>
I like the color <color>

@color
red| green| blue| yellow| dark <color>
```

My favorite color is blue

I like the color green

I like the color dark red

Fig. 8. Bronco code that generates a description of a favorite color (left), and sample output (right).

that symbol resolves into text. This lets users of Bronco interact with symbols in a similiar way to how they might interact with functions in a traditional programming language.

With symbols that take arguments, comes general-purpose logic. We are able to support conditional statements for example, as a symbol whose first argument must be a Boolean expression, whose second argument will be included in the output if the first argument resolves to true, and whose third argument will be included otherwise. In a similar manner, we are able to support operators for things like arithmetic and comparisons. Fig. 9 showcases the use of arguments and general-purpose logic.

```
@start
My favorite number is <set: favNum, (randomI: 1, 20)>, because it is
<numFact: favNum>!

@numFact: num
<if: (gt: num, 5), `greater than 5`, `less than 6`>
<if: (equal: num, 13), `unlucky`, `isn't unlucky like 13`>
just a good number
```

My favorite number is 7, because it is isn't unlucky like 13!

My favorite number is 15, because it is greater than 5!

My favorite number is 3, because it is just a good number!

Fig. 9. Bronco code that generates a description of a favorite number (left), and sample output (right).

In addition to the features mentioned so far, Bronco gains much expressive power by its inclusion of conditions and weights. A weight is just an easily specifiable constant number that changes the probability of a given item being picked from a bag (a '`%`' followed by a number, written at the end of a bag item). To dynamically change the probability of items being picked, conditions can be attached to an item (a symbol surrounded by '`[]`' at the end of a bag item). Conditions can be any symbol that resolves to a number, which is then multiplied by the base weight for a dynamic weight. Since Boolean type symbols are treated as a type of number in Bronco (with a value of 1 for true and 0 for false), items can be effectively enabled or disabled from a bag by entirely customizable logic. Fig. 10 shows conditions on bag items.

As a means of storing and relating data more abstractly, symbols in Bronco can have tags attached to them (one or more '`#`' followed by the identifier, written at the end of a bag item). A tag is a key-value pair consisting of a text-identifier and a number. Tags do not have any implicit function but can be accessed, compared, and manipulated through various predefined symbols in

```
@start
<draw>The <numToValue: value> of <numToSuit: suit>s

@draw
<do: (set: value, (randomI: 1, 13)), (set: suit, (randomI: 0, 4))>

@numToSuit: num
Spade[(equal: num, 0)]| Heart[(equal: num, 1)]| Diamond[(equal: num, 2)]|
Club[(equal: num, 3)]

@numToValue: num
Ace[(equal: num, 1)]| Jack[(equal: num, 11)]| Queen[(equal: num, 12)]|
King[(equal: num, 13)]
<num>[(and: (gt: num, 1),(lt: num, 11))]
```

```
The 3 of Hearts

The King of Spades

The Queen of Diamonds
```

Fig. 10. Bronco code that generates a description of a playing card (left), and sample output (right).

Bronco. This allows authors to annotate a large list of words with a theme for example, and then increase the probability of a word being picked if it matches a theme specified in advance. Tags might also be accumulated, during one part of the generation, and then used as a filter in a later part of the generation, ensuring consistency. Fig. 11 demonstrates the use of tags for consistent sentences.

```
@start
<do: (set: pro, pronoun)><cap: (their: pro)> favorite color is <set:
favColor, color>. <cap: (they: pro)> <like: pro> it, because it goes so
well with <colorCompliment: favColor>.

@color
red#red| green#green| yellow#yellow| blue#blue| purple#purple|
orange#orange

@colorCompliment: color [(tagOverlap: color, item)]
green#red| red#green| purple#yellow| orange#blue| yellow#purple|
blue#orange

@they: pro [(tagOverlap: pro, item)]
they#they| she#she| he#he

@their: pro [(tagOverlap: pro, item)]
their#they| her#she| his#he

@like: pro [(tagOverlap: pro, item)]
like#they| likes#he #she

@pronoun
they#they| she#she| he#he
```

```
Her favorite color is green. She likes it, because it goes so
well with red.

His favorite color is yellow. He likes it, because it goes so
well with purple.

Their favorite color is purple. They like it, because it goes so
well with yellow.
```

Fig. 11. Bronco code that generates a consistent sentence (left), and sample output (right).

4 Assessment

We will be assessing the performance of Bronco with two distinct methods: Firstly, we will consider the notion of *expressivness* as it applied to Bronco and how it compares to Tracery. The second is similar to that used by Martens and Simmons and their language *Inbox*, where it is directly compared with an existing tool in a similar space [19]. We have chosen to evaluate Bronco with respect to Tracery, because it is the most popular and direct influence for Bronco.

Expressionist, another popular and strong influence, requires its users to write their own expansion engines, which makes it difficult to compare objectively, so we will not be comparing with it.

4.1 Expressiveness

We rely on Felleisen's [9] notion of expressiveness to assess Bronco. Conceptually, we can say that one Turing-complete language is more expressive than another in a particular context, if a local change in the first language would require a global change in the second language in order to achieve the same behavior.

As discussed in Sec. 2.2, Tracery is not strictly context-free but is only capable of referencing branches taken earlier in the expansion by outputting the exact text that branch output. Bronco on the other hand, can reference previous branches by outputting text from any definable function of earlier expansions. Although a great deal of complexity is possible within Tracery's system, in the general case, the number of additional branches needed increases combinatorially with the number of previous branches that need to be taken into account. If branch Q has an outcome dependent on the outcome of an earlier branch P, then Tracery-like systems requires an additional n branches, where n is the number of branches between P and Q. In a Bronco-like system, only 1 extra branch is needed.

As an example, consider a generated story in which a character may or may not find a key at the beginning. At a certain point far into the story, the character can use the key, if they have it, to unlock a door, resulting in a new section of generated story. Though this is certainly possible in Tracery, it would require two entirely separate branches for finding the key or not finding the key up to the point where the key-related part of the story ends. Fig. 12 shows a more general example in which branch C will only be taken if A is taken first, and the same thing for D and B. Even if $N_1, N_2, ..., N_n$ are unaffected by the outcome of P, they must be repeated if there is no general way to reference earlier expansions. The increase in size from the first figure to the second is multiplied with every new branch that needs to be considered.

4.2 Language Comparison With Tracery

Fig. 13 represents the same generator written in Bronco on the left, and Tracery on the right. This example was taken directly from the Tracery website [4]. The figure serves to demonstrate the advantage of a custom-built language for procedural generation by the neatness of the Bronco code when compared with the Tracery code. To put this more quantitatively (admittedly at a very coarse grain of detail), we can count the number of characters that are being used for structural organization versus the number of characters that can actually be output by the generator (this count ignores characters in aesthetic white space and indirect functions like capitalization and variable assignment). For this generator, both versions have 313 characters that can appear in the output, Bronco uses 88 characters for organization (Including the newlines that Bronco

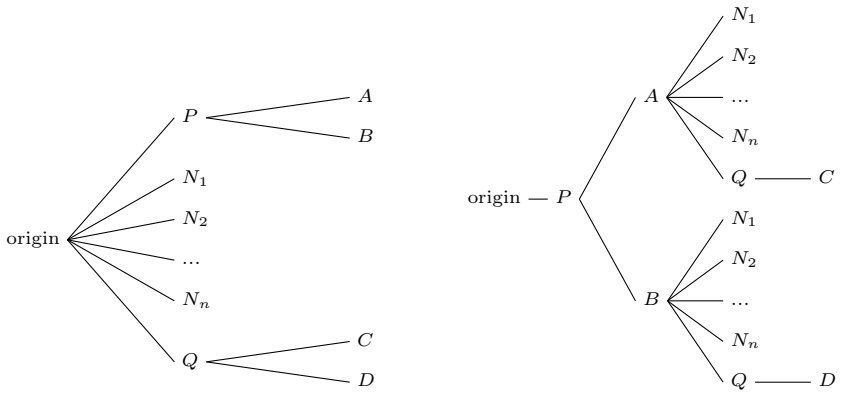

Fig. 12. Trees describing the structure of two grammars, the first without dependendcies between P and Q and the second with dependencies.

```
@start
<do: (set: myPlace, path)><line>

@line
<cap: mood> and <mood>, the <myPlace> was <mood>
with <substance>
<cap: nearby> <a: myPlace> <ed: move> through the
<path>, filling me with <substance>

@nearby
beyond the <path>| far away| ahead| behind me

@substance
light| reflections| mist| shadow| darkness|
brightness| gaiety| merriment

@mood
overcast| alight| clear| darkened| blue|
shadowed| illuminated| silver| cool| warm|
summer-warmed

@path
stream| brook| path| ravine| forest| fence| stone
wall

@move
spiral| twirl| curl| dance| twine| weave|
meander| wander| flow
```

```
{
"origin":
["[myPlace: #path#]#line#"],

"line":
["#mood.capitalize# and #mood#, the #myPlace# was
#mood# with #substance#",
"#nearby.capitalize# #myPlace.a# #move.ed#
through the #path#, filling me with
#substance#"],

"nearby":
["beyond the #path#", "far away", "ahead",
"behind me"],

"substance":
["light", "reflections", "mist", "shadow",
"darkness", "brightness", "gaiety", "merriment"],

"mood":
["overcast", "alight", "clear", "darkened",
"blue", "shadowed", "illuminated", "silver",
"cool", "warm", "summer-warmed"],

"path":
["stream", "brook", "path", "ravine", "forest",
"fence", "stone wall"],

"move":
["spiral", "twirl", "curl", "dance", "twine",
"weave", "meander", "wander", "flow"]
}
```

Fig. 13. The same generator written in Bronco (left), and Tracery (Right)

uses to separate items) and Tracery uses 165. Although it is difficult to conduct this kind of count objectively, these numbers suggest that about twice as much redundant typing is needed to write this generator in Tracery as it is in Bronco. To be sure, writing in JSON has many advantages, such as being relatively understandable by a majority of programmers, and being easily parsed.

```
@start
<intro><battle>

@battle
The <char1> makes the first move! \n<char1Turn>| The <char2> makes the
first move! \n<char2Turn>

@char1Turn
<charTurn: char1, char2, hp2><do: (set: hp2, dmgHP)><if: (gt: hp2, 0),
char2Turn, (battleEnd: char1)>

@char2Turn
<charTurn: char2, char1, hp1><do: (set: hp1, dmgHP)><if: (gt: hp1, 0),
char1Turn, (battleEnd: char2)>

@charTurn: charMe, charEn, hpEn
The <charMe> attacks the <charEn>! <attack: charEn, hpEn>\n

@attack: char, hp
The attack deals <set: dmg, (randomI: 5, 20)> damage, leaving the <char>
<dmgDescribe: hp, dmg>| The attack misses the <char>.<do: (set: dmgHP,
hp)>

@dmgDescribe: hp, dmg
with <set: dmgHP, (sub: hp, dmg)>.[(gt: hp, dmg)]| dead<do: (set: dmgHP,
0)>.[(not: (gt: hp, dmg))]

@intro
<cap: (a: (set: char1, animal))><do: (set: hp1, hp)> and <a: (set: char2,
animal)><do: (set: hp2, hp)> prepare for battle.\n

@battleEnd: char
The <char> is victorious!

@animal
ant<do: (set: hp, 1)>| snake<do: (set: hp, 20)>| rat<do: (set: hp, 10)>|
bear<do: (set: hp, 50)>| dragon<do: (set: hp, 70)>| emu<do: (set: hp,
30)>| lion<do: (set: hp, 40)>
```

```
A snake and a rat prepare for battle.
The rat makes the first move!
The rat attacks the snake! The attack deals 17 damage,
leaving the snake with 3.
The snake attacks the rat! The attack misses the rat.
The rat attacks the snake! The attack misses the snake.
The snake attacks the rat! The attack deals 8 damage,
leaving the rat with 2.
The rat attacks the snake! The attack deals 17 damage,
leaving the snake dead.
The rat is victorious!
```

Fig. 14. Bronco code that generates the description of a battle (left), and sample output (right).

Fig. 14 and Fig. 11 show two simple generators in Bronco that would be difficult to write in the default version Tracery. The code in Fig. 14 generates a simple description of a battle that you might see in a roll playing game. It makes use of basic dynamic elements like numeric variables and conditional recursion. The example in Fig. 11 generates two sentences describing a person's favorite color. It is notable for its use of Bronco's tagging system, which let it use consistent pronouns and to track a color's compliment. Although Tracery is fully integrated with JavaScript making it capable of performing any kind of computation, neither of the two examples given could take meaningful advantage of Tracery's in built features.

5 Future Work

One area in need of improvement is the tagging system. It has already proven very useful for communicating abstract connections such as correct pronouns, an area that is difficult in Tracery [5]. However, the way tags currently work makes it difficult to take full advantage of them. Ideally, they should make it effortless for branches to have prerequisites or be mutually exclusive, but this is currently not the case. Their function will need to be re-worked in the future.

Possibly the most obvious weakness in Bronco right now is its syntax. Generally speaking, there are many areas of the syntax that should be easier to write for users than they are. This is no individual issue, but a broad set of problems that need to be addressed for Bronco to meet its goal of ease of use.

A large portion of this research that has not been discussed in this paper is Bronco's integration into a game that contextualizes Bronco's design. In this game, characters are procedurally generated, and procedural text will be used for their in-game conversations. This project demands a nuanced, dynamic, and realistic output. We expect that this game will continue to yield insights for the design of Bronco. Fully explicating how the game and generator designs are mutually constraining and reinforcing will be a third area of our future effort.

6 Conclusion

In this paper, we (to our knowledge, for the first time) formally described two landmark generative text systems: Tracery and Expressionist. We did so in order to distinguish our novel system, Bronco, as a Turing-complete procedural text generator that is computationally more expressive than either of these. In summary, while Tracery/Expressionist are marketed as though they depend on context-free grammars, the formal properties they rely on makes them instead dependent on restricted attribute grammars. Bronco's expressivity stems from our novel yielding grammar formalism, which carefully relaxes the restrictions that Tracery/Expressionist impose on their underlying attribute grammars. This expressivity was necessary for us; in specific, generating text within a dynamic domain that relies on conditional expansions (as discussed earlier) is too onerous to specify in Tracery relative to Bronco.

At the same time, we offer a necessary caveat: our distinction should not be construed as an implied claim of universal superiority. We look to Tracery, Expressionist, and all systems cited herein with profound respect and admiration, and note that Bronco is pragmatically neither better nor worse than any system we have cited; it is simply *different*. The question of *how* it is different requires we be precise, which explains how our discussion centers on Bronco's unique advantages (as opposed to further explicating how prior systems serve as distinct inspiration for our work). The choice of which procedural text generator to use is highly context-specific and it is entirely possible that—in the words of one our anonymous reviewers—Bronco "is likely to be another tool to be thrown on the cornucopia, only to languish in lack of use." While that comment reflects

an honest assessment of the difficulties in gaining widespread traction with particular tools, we are hopeful that Bronco pushes procedural text generation into a heretofore under-explored corner of the authorability design space: one that affords Turing-complete controllability.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. #2046294. We also wish to thank Monthir Ali, Nancy N. Blackburn, Michael Clemens, and the anonymous reviewers who were tremendously helpful with their comments during peer review.

References

1. Black, J.B., Wilensky, R.: An evaluation of story grammars. Cognitive science **3**(3), 213–229 (1979)
2. Chen, S., Nelson, M., Mateas, M.: Evaluating the authorial leverage of drama management. In: Proceedings of the 5th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment. pp. 136–141 (2009)
3. Church, A.: An unsolvable problem of elementary number theory. American Journal of Mathematics **58** (1936)
4. Compton, K.: Tracery. https://www.tracery.io/ (2015)
5. Compton, K.: Practical Low Effort PCG: Tracery and data-oriented PCG authoring. In: Roguelike Celebration (2016)
6. Compton, K., Kybartas, B., Mateas, M.: Tracery: An author-focused generative text tool. In: Proceedings of the 8th International Conference on Interactive Digital Storytelling (2015)
7. Cone, M.: Basic syntax — markdown guide. https://www.markdownguide.org/basic-syntax/ (2022)
8. Dias, B.: Procedural Storytelling in Game Design, chap. Procedural Descriptions in Voyageur. Taylor and Francis (2019)
9. Felleisen, M.: On the expressive power of programming languages. In: European Symposium on Programming. pp. 134–151. Springer (1990)
10. Garbe, J.: Increasing Authorial Leverage in Generative Narrative Systems. Ph.D. thesis, University of California, Santa Cruz (2020)
11. Grinblat, J.: Procedurally generating history in 'caves of qud'. In: Game Developers Conference (2018)
12. Horswill, I.: Architectural issues for compositional dialog in games. In: Proceedings of the Workshop on Games and Natural Language Processing at the 10th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (2014)
13. Humfrey, J.: Ink: The narrative scripting language behind '80 days' and 'sorcery!'. In: Game Developers Conference (2016)
14. Inkle: Inkle: Ink. https://github.com/inkle/ink/blob/master/Documentation/WritingWithInk.md (2021)
15. Jelinek, F., Lafferty, J.D., Mercer, R.L.: Basic methods of probabilistic context free grammars. In: Speech Recognition and Understanding, pp. 345–360. Springer (1992)

16. Johnson, Z.: Beyond the mad lib (but just barely): an oral history of the ways in which kingdom of loathing uses procedural text generation for flavor and humor. In: Roguelike Celebration (2016)
17. Koster, C.H.: Affix grammars for natural languages. In: International summer school on attribute grammars, applications, and systems. pp. 469–484. Springer (1991)
18. Lessard, J., Kybartas, Q.: Blabbeur - an accessible text generation authoring system for unity. In: Proceedings of the 14th International Conference on Interactive Digital Storytelling (2021)
19. Martens, C., Simmons, R.J.: Inbox games: Poetics and authoring support. In: International Conference on Interactive Digital Storytelling (2021)
20. Mawhorter, P.A.: Artificial Intelligence as a tool for understanding narrative choices. Ph.D. thesis, University of California Santa Cruz (2016)
21. Paakki, J.: Attribute grammar paradigms—a high-level methodology in language implementation. ACM Computing Surveys (CSUR) **27**(2), 196–255 (1995)
22. Ryan, J., Mateas, M., Wardrip-Fruin, N.: Characters who speak their minds: Dialogue generation in talk of the town. In: 12th Artificial Intelligence and Interactive Digital Entertainment Conference (2016)
23. Ryan, J., Seither, E., Mateas, M., Wardrip-Fruin, N.: Expressionist: An authoring tool for in-game text generation. In: International Conference on Interactive Digital Storytelling (2016)
24. Ullman, J., Hopcroft, J.: Introduction to automata theory, languages, and computation, chap. Chapter 4: Context-Free Grammars. Addison-Wesley (1979)