A Report on the Language of TraceryPlusPlus

Theodore Barnes - 1825139
Vakaris Paulavičius - 20062023

March 2023

0.1 Introduction

0.1.1 What is Tracery? A brief overview

Tracery is 'An Author-Focused Generative Text Tool' developed by Kate Compton, an open-source tool that allows both novice and expert authors from many walks of life to produce JSON objects which are then subject to expansion under Tracery's structure to form 'a wide range of stories, poems, dialogue, and even images and code.' (Compton et al. 154-161)[2]. Tracery aims to provide a haven for generative-text authors to be able to produce and extend interactive fiction through the inclusion of data portability, modular design, and additive authoring. Written predominantly using JavaScript, Tracery supports the concept of adding or modifying features during runtime, and the base grammar is intended to support the work of many, and allow for creativity to shine through when extended.

0.1.2 Why develop TraceryPlusPlus?

We chose to develop TraceryPlusPlus to make it simpler for those naive to technology to generate text from easily producible programs more closely related to English. JSON objects are taken as input for the Tracery JavaScript library; for the inexperienced, this format of specifying the text they want to generate may not be so straightforward. We decided to subvert this, creating an environment in which the user may enter their stories in a way that is more closely related to the text output they are looking to generate. This allows the user to define their text through more readable and structured means and keeps them from getting lost in defining many objects in an unfamiliar medium.

0.1.3 The importance of Tracery

With the potential of being overlooked at first glance, Tracery serves a purpose in a modern world. Specialising in short-form generative text, Tracery brings the ability to create additive, creative and structured texts en masse to the many users. Twitterbots, particularly in the context of the modern political landscape[1], can influence public opinion on social media with the sorts of conspiracies, harmful opinions, and fake news peddled by those who benefit from the spread of misinformation. However, an open-source application such as Tracery ensures this ability reaches new clientele, essentially people who wish to share gripping, correct, and essential information. Tracery may also be applied to many fields, for example, weather reporting, providing prompts for presenters to use when avoiding dead air, and generating real-time emergency notifications.

0.1.4 An MDD Approach

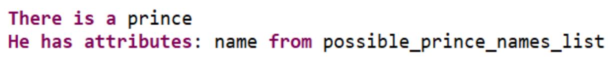
We take an MDD approach to extend Tracery as it is, by definition, 'Author-Focused'. Due to this definitive characteristic, we feel to be truly author-focused is to write scripts more akin to the literary model an author would be used to, abstracting to a higher level model than programming paradigms. Taking a model-driven approach fosters the core values held by Tracery, to be collaborative, understood, and replicable without being overly complex. In the pursuit of serving the users' needs, a model-driven approach is a logical step forward for Tracery. Moreover, Tracery has always encouraged building extensions for the language, something that is more easily done in a domain specific language, and users may choose to extend with ease in TraceryPlusPlus. Overall, taking a model-driven approach to this domain increases the intuitiveness of the user experience, while also encouraging more creative freedom of expression through the language in a larger potential client base.

0.2 Notes on Syntax for a Textual Language

We implemented syntax for a textual language to align with Tracery and the production of text outputs. Due to the nature of Tracery being ‘Author-Focused’, it would only be sensible for the concrete syntax to be textual.

0.2.1 Basic Building Blocks

Our language uses basic building blocks, rules to define types used throughout the rest of the code. Perhaps the most simple of these being a word, a string which may be specified once or more as part of a list through the WordList rule. Cohesively, the user may enter a list with “Cars has values: “Ford”, “Skoda”, “Mercedes””. Another important factor in our language is considered to be objects which have attributes. Object attributes are captured by the Attribute rule, as either a reference to an existing list, or the assignment of a word as the attribute’s value. To accommodate for the former, the type NameExistingListAttribute uses cross-referencing to allow an object attribute to take its value from a ListDeclaration. This allows the user to define an attribute as perhaps, “vehicle from Cars” where the attribute name would be vehicle while the value would contain a reference to the ListDeclaration ‘Cars’. The inclusion of ‘from’ is optional in the case of defining an attribute from a list, an author may choose to include this or an equally optional AssignmentOperator, to enhance readability; they may also be omitted so as to reduce errors at this stage. When declaring an attribute not pointing to an existing list, the user may write, for example, “favouriteFood = ‘Pizza’”, where the left-hand side is the name of the attribute (favouriteFood) and the value is of type Word (“Pizza”). Pronouns are also taken care of in TraceryPlusPlus when an author creates an object, and they specify their choice of: ”He”, ”She”, ”It”, or ”They” (Figure 1).

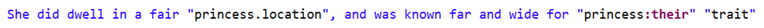


```
There is a prince
He has attributes: name from possible_prince_names_list
```

Figure 1: Specification of an object, prince.

0.2.2 Creating Stories

Stories in TraceryPlusPlus begin with the phrase ‘The story’, and from there, are built with a mixture of words and references to variables. References may point to predefined lists, object attributes, object pronouns, or substories. Variable usage is ruled similarly to how variables are specified, with the ability to use lists, objects, or further substories through ListUse, ObjectUse, and SubstoryUse, each falling under the higher level type VariableUse. When referencing an item from a pre-existing list, ListUse contains a cross-reference to ListDeclaration as the variable being passed, along with a list of modifiers[1]. The ObjectUse rule allows for object attributes to be referenced by the story or substory, alongside object pronouns. To properly use an object attribute in the story, cross-referencing allows for the syntax ‘ObjectDeclaration.Attribute’ and then again a list of modifiers. Moreover, when it comes time to use pronouns within a story, identifiers ‘:they’, ‘:them’, ‘:their’, and ‘:theirs’ indicate the use of subject, object, and possessive forms of object’s pronoun (Figure 2).



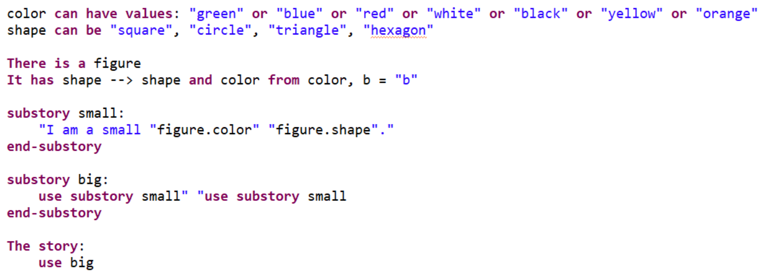
```
She did dwell in a fair "princess.location", and was known far and wide for "princess:their" "trait"
```

Figure 2: Example use of a princess’ possessive pronoun

[1]This list of modifiers may be empty

0.2.3 Substories

Substories may also be specified in TraceryPlusPlus. These are constructed in the same way as the main story, but are bookended by “substory” and “end-substory”. As detailed before, a reference to a SubstoryDeclaration is passed via the SubstoryUse rule. This way, an author may specify multiple substories and place them throughout the main story and within other substories also. This use of substories is applied by an author specifying ‘use’ or ‘use substory/sub’ followed by the substory name (Figure 3).



```
color can have values: "green" or "blue" or "red" or "white" or "black" or "yellow" or "orange"
shape can be "square", "circle", "triangle", "hexagon"

There is a figure
It has shape --> shape and color from color, b = "b"

substory small:
    "I am a small "figure.color" "figure.shape"."
end-substory

substory big:
    use substory small" "use substory small
end-substory

The story:
    use big
```

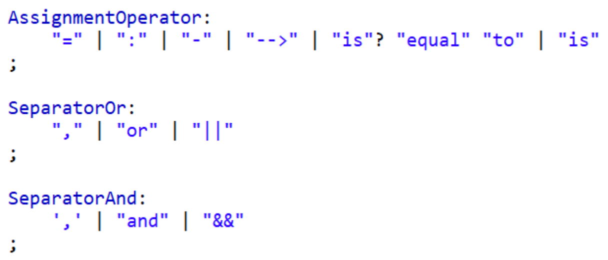
Figure 3: Specification and use of a substory

0.2.4 Modifiers

TraceryPlusPlus offers a multitude of modifiers, appended to allow the full customisation of words and attributes used in a story. These include the ability to capitalize the first letter of a word with ‘-capitalize’, to fully capitalise a word with ‘-CAPITALIZE’, to pluralize with the suffix ‘-s’, to indicate an article with ‘-a’, and to use the past tense of a verb with ‘-ed’.

0.2.5 Operators

Helper functions AssignmentOperator, SeparatorOr, and SeparatorAnd (Figure 4) fulfil the ease of use in TraceryPlusPlus. The litany of assignment operators included make it easy for those who are used to many different assignment operators to denote comfortably in TraceryPlusPlus. AssignmentOperators are consistently optional throughout the TraceryPlusPlus grammar, while SeparatorOr and SeparatorAnd operators distinguish between elements in a list and multiple object attributes respectively.

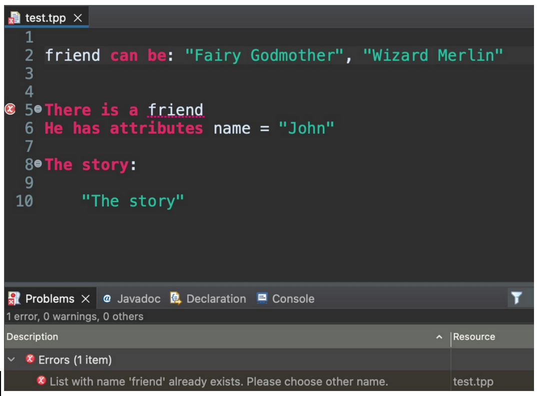


```
AssignmentOperator:
    "=" | ":" | "-" | "-->" | "is"? "equal" "to" | "is"
;

SeparatorOr:
    "," | "or" | "||"
;

SeparatorAnd:
    ',' | "and" | "&&"
;
```

Figure 4: Operators available in TraceryPlusPlus

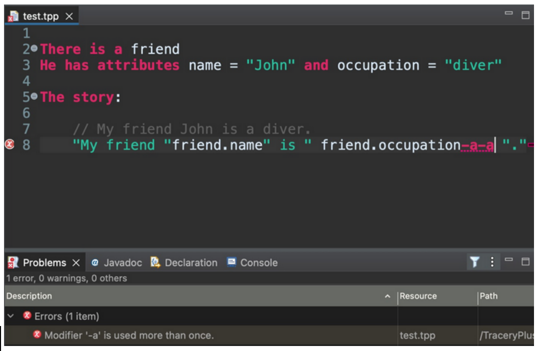
0.3 Validations in TraceryPlusPlus

Custom validators are implemented in TraceryPlusPlus to check for invalid structures in the code. Importantly, the keyword ‘story’ is reserved in TraceryPlusPlus. To ensure the user cannot use this as a variable name, when a variable is found to be named ‘story’ an error is given reading

[a] 

```

friend can be: "Fairy Godmother", "Wizard Merlin"


There is a friend
He has attributes name = "John"

The story:

    "The story"
```

```
Problems | Javadoc | Declaration | Console
1 error, 0 warnings, 0 others
Description                                                Resource
Errors (1 item)
  List with name 'friend' already exists. Please choose other name.   test.tpp
```

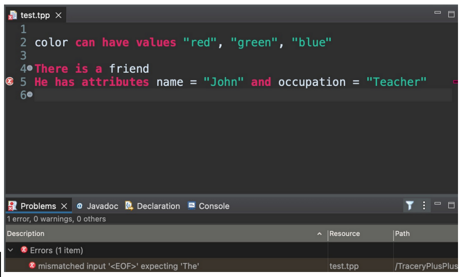
[b] 

```

There is a friend
He has attributes name = "John" and occupation = "diver"

The story:

    // My friend John is a diver.
    "My friend "friend.name" is " friend.occupation-a-a "."
```

```
Problems | Javadoc | Declaration | Console
1 error, 0 warnings, 0 others
Description                              Resource    Path
Errors (1 item)
  Modifier '-a' is used more than once.  test.tpp    /TraceryPlu
```

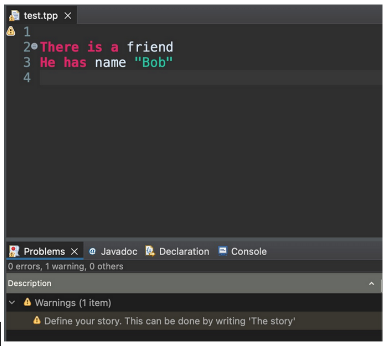
Figure 5: (a) An error, informing the user that a List with name 'friend' already exists (b) An error, informing the user that a modifier '-a' is used more than once

"The name 'story' is reserved and cannot be used as a variable name.". This check is default and therefore marked fast, meaning it will run every time the file is edited. Another essential structural validation implemented is to ensure that all variables have unique names. This check, performed after every edit, is in the form of nested loops which traverse the variables specified in the program. For each variable (i) the method checks the variables that follow in the code (j); if their names match, an error is generated, asking the user to choose another name for this variable (i)(Figure 5(a)). Similarly, it would be erroneous for object attributes to not have uniquely identifiable names when parsing. A default check is included to search for these errors upon every edit, again using a nested loop to check each attribute contained within an object against all others in the same object. When this is found, an error is generated informing the user that the attribute is used more than once. Furthermore, duplications of modifiers, that is using the same modifier on a reference to a list or object attribute more than once, are avoided and produce an error if this occurs during editing of the code. In the grammar, modifiers are created as a ModifierList for each ListDeclaration reference or ObjectAttribute they are imprinted on, therefore the validator goes through these lists, and within each, checks for duplications. When duplications are found, an error is generated informing the user that they have used that particular modifier more than once (Figure 5(b)). This check needs to occur after every edit, as structures such as this may crop up often due to human error when typing. Although stories are the essential component of TraceryPlusPlus, they are left optional in the grammar, as not doing so creates an unappealing and uninformative error when writing code (Figure 6(a)). To circumvent this, a check marked normal was included, run whenever the file is saved, which produces an informative warning if either the program as a whole or the story is undefined or null. We felt that this would be less intimidating to the author and that the warning message, "Define your story. This can be done by writing 'The story'" (Figure 6(b)), was much more informative in directing the author on the next steps they need to take in writing their TraceryPlusPlus code. Private methods to help with these validations are also included. For instance, when checking the names of object attributes for duplications, this process must be handled for both subtypes of an Attribute, namely NameExistingListAttribute and NameValueAttribute; the method getAttributeName in the validator class achieves this by retrieving the attribute name for either potential subtype of the Attribute under inspection. Additionally, the switch expression getType retrieves the type name of a variable reference when needed in a generated error message. This method returns the string "Object" when getType is called with the parameter of a Variable of type ObjectDeclaration, "List" for one of ListDeclaration, "Substory" for one of SubstoryDeclaration, with a default return string "Variable" to ensure completeness.

[a] 

```
color can have values "red", "green", "blue"

There is a friend
He has attributes name = "John" and occupation = "Teacher"

```

```
1 error, 0 warnings, 0 others
Description                                        Resource   Path
Errors (1 item)
  mismatched input '<EOF>' expecting 'The'          test.tpp   /TraceryPlusPlus
```

[b] 

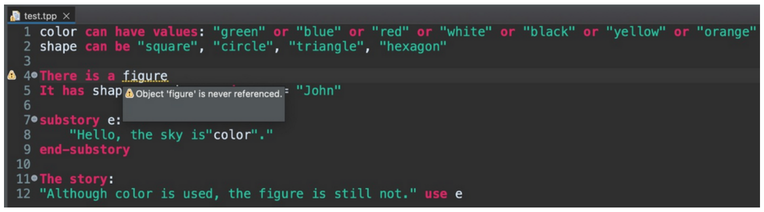
```

There is a friend
He has name "Bob"

```

```
0 errors, 1 warning, 0 others
Description
Warnings (1 item)
  Define your story. This can be done by writing 'The story'
```

Figure 6: (a) An ugly error, caused by the user not defining their story (b) A helpful warning, instructing the user on the next steps they must take

0.3.1 An Advanced Semantic Check

TraceryPlusPlus contains an advanced semantic check which gives warnings to the user if declared variables are never referenced in the story. This default check is prefaced by the named method checkForUnusedVariables, which initially takes the high-level model object TraceryPlusPlusProgram and filters the Variable objects contained in that program to be stored in the local value variables. This allows a search for uses of each defined variable in the rest of the code, at which point dispatch methods checkIfUsed pick up a substantial heavy lifting. These methods are developed using the extension method principle fundamental to Xtend, working in conjunction with dispatch techniques to allow for polymorphic extension methods. In this context, within the EType hierarchy Variable, the correct method to use is inferred from the receiver with which the method was called. Therefore three dispatch methods exist whose first arguments are of type ObjectDeclaration, ListDeclaration and SubstoryDeclaration respectively. All instances also take a second argument, program, passed through parenthesis in the method call. When called, these methods first inspect the overarching story by extracting the containment reference list from the program into a value 'story' to be iterated through. Upon iteration, an if statement containing an instanceof cascade inspects whether each element of the story list is a reference to an instance of ObjectUse/ListUse/SubstoryUse. If true, a second if statement performs an equality operation on the element name and the name of the variable, the initial receiver with which the method was called. If this equality operation returns true, then the method returns true, effectively evidencing that an element used in the story contains a reference to a preexisting Variable. During iteration and inspection of the story, if either of the if statements contained return false, then the method goes through the same iteration and inspections using containment reference lists from all SubstoryDeclaration objects in the program. In the same way, if any element of the substory is found to have the same name as the variable with which the method was called, the method returns true. As in TraceryPlusPlus grammar, an attribute of an object may be specified to be a reference to an element of an existing ListDeclaration; the dispatch method for handling variables of type ListDeclaration must also inspect the name of the value of all object attributes which are of type NameExistingListAttribute. Again, if this is found to be equal to the name of the receiver with which the method was called, then a reference to this variable has been found and so the method returns true. However, this code segment is only entered if there has been no reference to the ListDeclaration found in either the story or any substories. In all cases, if no part of the method code leads to true, then the method returns false, meaning that no reference in the code to the variable has been found in the story, any substory, and in the case of a ListDeclaration, any object either. Here, the initial method, checkForUnusedVariables, again takes over. For each variable found as an existing statement in the code, if a call for checkIfUsed returns false, a warning must be generated. This warning uses the familiar switch expression getType, to begin with the type of variable which the warning is being generated for, followed by the name of the variable for clarity, trailed with the words "is never referenced" (Figure 7).



```
test.tpp
color can have values: "green" or "blue" or "red" or "white" or "black" or "yellow" or "orange"
shape can be "square", "circle", "triangle", "hexagon"

There is a figure
It has shap                        = "John"

substory e:
    "Hello, the sky is"color"."
end-substory

The story:
"Although color is used, the figure is still not." use e
```

Object 'figure' is never referenced.

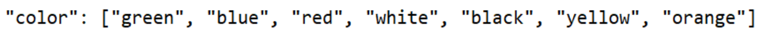
Figure 7: Our Advanced Semantic Check warning that Object 'figure' is never referenced.

0.4 The Execution of TraceryPlusPlus

The Tracery language takes JSON objects as input, and in turn, refactors these into a readable story. As discussed at the beginning of this report, it was felt that, for novice authors, it would be much easier to specify their programs using TraceryPlusPlus syntax, and for this to generate the necessary JSON objects.

0.4.1 Techniques Implemented in our Execution

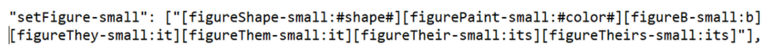
Essentially, our execution of TraceryPlusPlus is to generate JSON objects, stored in a JSON file named 'translated_tracerypp_grammar.json'. The contents of this file are generated starting with the implementation of the method 'generate', which generates a CharSequence from scripting code, alongside target code to ensure that notation is consistent with JSON. Providing the overarching story is defined in the code, this method first extracts the ListDeclaration objects from the program, looping through these, method generateJsonListDeclaration then affixes the name of the ListDeclaration to the list of words contained, each separated by a comma and all bookended by square brackets. This produces lists encased in JSON notation (Figure 8). Object



```
"color": ["green", "blue", "red", "white", "black", "yellow", "orange"]
```

Figure 8: A JSON object generated from a list of colors

use, however, is more complicated in TraceryPlusPlus, and to handle these complexities several interlinking methods must be invoked. Objects are referenced as elemental parts in a specific substory, or overarching story that they belong to, and are therefore generated accordingly within methods getSubstoryObjectDeclarations and getStoryObjectDeclarations. For the substories case, this is achieved by iteration through ObjectUse variables referenced in their respective substories. Method findTheRightObjectDeclaration returns a reference to the ObjectDeclaration whose name matches the name extracted from the ObjectUse variable under inspection. Provided this reference is not null and is not already initialised as an object, code generation may then take place. A setter value is used, which affixes the word "set" to the object name, before appending this to the name of the substory it is referenced in. Then in a for-loop within scripting code, the method getStringForAttribute takes attributes extracted from the correct ObjectDeclaration and returns a string to represent the value of that attribute. Depending on whether this string references a NameValueAttribute or a NameExistingListAttribute, for the latter, it should be interpreted by Tracery as a reference to the use of a list rather than a string literal, and therefore the name of the list is encased in "#list.name#" notation (Figure 9); in the former case, a string is used as the variable name. Our method matchPronouns efficiently returns the subject, object, and possessive forms of the pronoun with which an object was defined, as the prefix identification They/Them/Their/Theirs indicates. In the previous example (Figure 3), for a figure defined with the pronoun 'It', its JSON object would contain the generated full set it/it/its/its (Figure 8). With objects from substories now generated, this process is implemented on objects referenced in the main story in the same way using method getStoryObjectDeclarations. For each substory, and the subsequent main story, a list of each element (that is: words, objects, lists, substories) is



```
"setFigure-small": ["[figureShape-small:#shape#][figurePaint-small:#color#][figureB-small:b]
[figureThey-small:it][figureThem-small:it][figureTheir-small:its][figureTheirs-small:its]"],
```

Figure 9: A JSON object generated for object 'figure' used in substory 'small'

mapped through dispatch methods generateJsonStoryEntry, and separated by quotation marks. These individually ensure the correct notation for each element of the story is emplaced, while also appending any modifiers to an object or list attribute for generation. The origin of a story effectively sets the scene for what lies ahead; entirely contained in '["#... story#"] notation, this introduces objects from within substories and the overarching story, retrieved as a list of strings of the object names in both scenarios. It's important to note that if no story is found within the code, the contents of the generated file will be the string "warning": "To get Tracery code, create the Story element", in keeping with the informativeness of validations in TraceryPlusPlus this clearly instructs the user of the steps they must take towards executing their code.

0.4.2 Executing TraceryPlusPlus

To execute TraceryPlusPlus models, one must first import the TraceryPlusPlus project into eclipse, from here, Xtext artifacts of the grammar file, TraceryPlusPlus.xtext should be generated. Once this is done, the user must launch a runtime eclipse with the correct dependencies, and create a new Java project and in that, a new file in the src folder with the .tpp extension. Here is where the TraceryPlusPlus model should be specified. In the Test directory of the TraceryPlusPlus project there exists a test suite, for performing Xpect tests on code, moreover the extra_files/tpp directory houses .tpp files for additional examples of stories specified in tpp syntax. Once finished, upon saving this file, a new file will be generated in the src-gen folder of the Java project, named 'translated_tracerypp_grammar.json', this file should be placed in the tracery folder within the tracery-plusplus directory created at the import of the project. After navigating to this tracery folder in the command line, and ensuring node.js is installed, entering 'node call-trecery.js' will execute the JSON file generated from the TraceryPlusPlus model specification, printing the generated story to the terminal.

0.5 Proposing Potential Changes to Syntax and Semantics

One potential change to TraceryPlusPlus lies in the advanced semantic check. Each dispatch method implements a nested for loop, which carries time complexity $O(n^2)$, refactoring these methods to ensure they run in linear or constant time would improve the semantic quality and assurance of the language. A key factor in creating stories in TraceryPlusPlus is the use of substories, these work in the same way as stories, being able to take references to lists and objects and effectively randomise the output of this story. However, to enhance the author's experience, there should be a way to specify whether separate instances of a substory should be identical to one another, or whether they should be unique in their randomness. Additionally, references to ListDeclarations and ObjectDeclarations are not distinguished until runtime, this means that when an author begins referencing an object name, an error is thrown stating that the reference could not be resolved to a reference of type ListDeclaration. Furthermore, code completion suggests the use of a modifier following a reference to an ObjectDeclaration, although these are only correctly used following a reference to a ListDeclaration or an ObjectAttribute. The inclusion of this incorrect code completion should be addressed in future updates to TraceryPlusPlus. One potential way of achieving this would be to add the keyword "List" in the ListUse reference rule, however, we chose not to include this, or fixes similar, since the addition of keywords would make authorship cumbersome, and distract from the intended ease of use which TraceryPlusPlus strives for.

Bibliography

[1] Chris Baraniuk. How twitter bots help fuel political feuds, Mar 2018.

[2] Kate Compton, Ben Kybartas, and Michael Mateas. Tracery: an author-focused generative text tool. In *Interactive Storytelling: 8th International Conference on Interactive Digital Storytelling, ICIDS 2015, Copenhagen, Denmark, November 30-December 4, 2015, Proceedings 8*, pages 154–161. Springer, 2015.