Procedural Text Generation with Stateful Context-Free Grammars

Aarne Uotila

University of Tampere

Faculty of Natural Sciences

M.Sc.Thesis

Supervisor: Erkki Mäkinen

January 2018

University of Tampere

Faculty of Natural Sciences

Aarne Uotila: Procedural Text Generation with Stateful Context-Free Grammars

M.Sc.Thesis, 31 pages

January 2018

Abstract

Context-free grammars have become an increasingly popular solution to procedural text generation. However, standard context-free grammars have inherent limitations with handling state and continuity. This thesis describes the implementation and evaluation of a domain specific language that aims to combine the simplicity of context-free grammars and the expressiveness of high-level programming languages.

Keywords: Procedural generation, context-free grammar, domain specific language, text generation

Contents

1 Introduction

Procedural text generation has become an increasingly popular field for academic, artistic and entertainment purposes in recent years. As practical applications have increased in both size and quantity, practical considerations such as usability, maintainability and simplicity have become more and more important.

Originally, the only plausible method of computational text generation was using a generic purpose programming language to piece together the text using procedures and variables. This obviously formed a significant barrier to entry for non-programmers. Even those who could write code would be limited by the fact that the implementation languge often wasn't designed with text generation in mind.

As computers eventually became more powerful, it became possible to construct custom domain specific languages for text generation. These languages placed more importance to the structure and presentation of the generated text rather than pure procedurality. Most of these languages have also focused on reducing the workload of the author by using simple structures and markup for common text characteristics.

However, these new languages have inevitably lost a lot of the expressiveness of generic languages as their syntax has become more and more like markup languages. Some of them have solved this shortcoming by adding more and more functionality to the interpreter of language, which poses its own problems. It seems possible that the solution is not to fully embrace the separation of logic and content but to combine the best aspects of both.

This thesis aims to answer whether it is possible to create a text generation system that combines the simplicity and usability of context-free grammars with some of the expressiveness and range of high level languages. In the following chapters we will approach this question by first reviewing the background and previous works related to text generation. We will then cover the description of Orcus, a domain specific language created for this thesis. Finally, we will evaluate whether Orcus has succeeded in answering this question through both a high-level overview and a case study of the language.

2 Background and related works

In this chapter, we will first go through some definitions related to context-free grammars and procedural text generation and then review related works through three different aproaches of text generation; code, template and grammar-based methods.

2.1 Definitions

In this section we introduce some definitions related to context-free grammars and procedural generation. The terms are described as they are relevant to this thesis with relatively little focus on the intricacies of their proper mathematical definitions. We are using *Oxford Dictionary of Computer Science [1]* and *Introduction to Automata Theory, Languages and Computation [2]* as references.

Grammars are a formal language concept which consists of a set of *terminal* and *non-terminal characters* and a set of *production rules*; these production rules describe all possible character strings in the language.

A terminal character is one that is considered to be a part of the final language described by the grammar, i.e., the processing of production rules is terminated by them when there are no more non-terminal characters to process. They are traditionally depicted with lowercase letters.

A non-terminal character is part of a set of characters that define where their respective production rules apply. They are traditionally depicted with uppercase letters.

A production rule traditionally takes the form of S -> a where S is the left and a the right side of the rule, respectively. This rule states that the *non-terminal* S on the left side can be replaced by the expression a on the right side of the rule. The string a itself can consist of both *terminal* and *non-terminal* symbols.

Context-free grammars are grammars that have a single non-terminal character on the left side of all of their production rules. In contrast, a context-sensitive grammar can have both multiple and terminal characters on either side of its production rules.

Procedural generation is a technical and artistic content creation method whe-

re the final assembly of design elements is performed by the computer. This is most commonly implemented by an algorithm that in some way combines human authored pieces at runtime.

Stateful in computer science means a system that has internal data about its current state. In other words, it alters its behavior depending on previous events. For example, a deck is a stateful system, because drawing a card from it changes its state (it no longer has that card) and affects its behavior (it's no longer possible to draw the same card from the deck).

2.2 Related work

Here we examine some previous approaches to text generation. We divide them into three categories: code-, template- and grammar-based methods. The categories are by no means mutually exclusive and individual cases such as Ink [6] might not fit neatly into any single category. However, this classification allows us to consider each high-level approach separately without getting bogged down in the details.

2.2.1 Code-based works

The most intuitive way to generate text is to simply use an existing programming language. This naturally gives the author a lot of expressive power but results in more work as a generic language will never be as efficient as a language built for the sole purpose of text generation. On the other hand, most text generation languages simply do not allow for the range of logic that a generic language does.

In practice, pure code-based works are exceedingly rare and the line between an external data file and a script can be fuzzy. However, as a rule of thumb it can be said that code-based works place more emphasis on the program logic while other methods focus on the content. This usually manifests itself as a background simulation that the generated text is mined from, such as in Curveship [7], which simulates an interactive fiction world, or Talk of the Town [4], which simulates the lives of the population of a small town, complete with life events and information propagation. In each case a separate piece of code then analyzes the simulation and attempts to identify and report interesting series of events.

Even a code-based generator often ends up employing a lightweight language or language-like data structure for formatting the final text. This separation of text generation and presentation keeps the program code relatively short and more maintainable. It also makes adding content easier. However, these benefits are not exclusive to the code-based method and the separated logic and content are naturally harder to integrate than with a combined approach.

The argument of this thesis is that the best solution to this problem is not only to separate the content from the underlying logic but also to elevate the content language and eliminate the need for a separate code layer entirely. As such, we need to take a closer look at the current crop of domain specific languages. We will discuss two general approaches in the following subsections, the template- and grammar-based methods.

2.2.2 Markup-based works

Markup-based works refer to text-generation methods that take a text template (often multiple paragraphs) and replace key parts with algorithmically determined content; for example, injecting the protagonist's name into a plot outline or the current temperature into a weather report. A relatively pure example is Scribe. In Scribe, the system cleanly divides the template text and the injected text into their own separate files. [3]

The key difference between this and a grammar-based approaches is that markup allows for state injection, while a pure grammar has no state at all. Often this state is used to accomodate simple facts about the generated world, such as using the correct pronouns for the protagonist's gender or other grammatical concerns. There is less attention on other potentially mutable aspects of the text, such as plot event ordering, possibly due to the fact that the templates themselves are often relatively large and might not lend themselves to recursion very well.

An important exception to the tendency towards large templates is Ink [6]. In it, each template is small and modular enough that they can be nested and arranged easily. Compared to the dictionary-based approach of Scribe, the inline nature of Ink's templates also allows for much smoother content production. On the other hand, in Ink the resulting text is more tightly integrated to the specific text and not as interchangeable between different texts, while in Scribe the fact that the templated text has its own format, theoretically allows for easier reusability of

its dictionaries and ontologies.

All in all, the markup method seems to be the key to handling continuity and context in text generation. However, it requires a suitably elegant and expressive language to be used efficiently, lest the templates themselves become too rigid and large. One way to approach this problem is to marry it with a grammar-based method, which are described below.

2.2.3 Grammar-based works

Finally, we have grammar-based works, of which Tracery [5] is the overwhelmingly most popular example. Tracery itself is a very simple and pure context-free grammar implemented in JavaScript and using JSON, which makes it easy to modify and expand. It is a perfect example of both the limits and benefits of a pure grammar approach.

Pure grammars are suitable for both top-down and bottom-up design approaches but require the writer to limit the amount of repeated state or eliminate it entirely, if they do not wish for their workload to increase immensely. Each new variable that the system has to maintain continuity of, requires a new version of every rule that includes that variable, a new version of every rule that requires that rule and so on.

For example, if our story outline allows for the protagonist to be a either a bear or a rabbit and we wanted the protagonist's favorite food to reflect that three scenes down the line, we would have to create a new non-terminal for each food scene and probably the whole outline itself. If we then wanted more scenes that took the species into consideration and more variation in general, the whole structure would quickly become extremely convoluted and unmaintainable.

This combinatorial explosion is the reason why most pure context-free grammars are used to generate relatively short or poetic texts, such as shopping lists and very short stories [5]. Creating something longer or less ambiguous requires a modified approach in practice. Fortunately, context-free grammars are themselves so simple that modification is easy and popular.

In addition, the simplicity of the grammars themselves makes them easy to understand, create and expand. Their built-in flexible structure makes them suitable for a variety of purposes and systems. Their major problem is the lack of context

sensitivity, which is often fixed by adding additional markup[4] or creating a new non-terminal on the fly from certain terms [5]. A different approach would be the inclusion of state and algorithmic content as an integral part of the language itself, as detailed in the next chapter.

3 Language structure

In this chapter we go over the implementation of a domain specific language for text generation. This language attempts to combine the best aspects of the three approaches detailed above by augmenting a context-free grammar with functionality usually only available in high level, general purpose programming languages. We will call this language Orcus.

Orcus is the stateful context-free grammar created for this thesis and is so named after the Roman god of the underworld. In addition to being a context-free grammar, Orcus also implements some elements of high-level programming languages. These elements elevate it above other, simpler grammars in terms of expressiveness.

It has been implemented as an interpreted language on top of both C# and TypeScript. The older C# version was created as part of NaNoGenMo 2016, and is available on GitHub [8]. The newer version is unreleased, but adds only a few features when compared to the previous one. The basic structure is the same, however, beginning with rules.

3.1 Rules

The basic unit of Orcus's implementation is the *Rule*. A rule is composed of *a nonterminal symbol* (random_name) and the *contents* ("Alice") of the rule:

```
random_name => "Alice".
```

Multiple rules with the same nonterminal can be written each on their own line:

```
random_name => "Alice"
random_name => "Clarice"
random_name => "Earl Jones Junior".
```

This can be compressed by writing the nonterminal on its own line. The following is equivalent to the previous block of code:

```
random_name
=> "Alice"
```

```
=> "Clarice"
=> "Earl Jones Junior".
```

This saves a lot of trouble for the user when writing large amounts of rules.

Now you may have noticed that the earlier examples don't include other nonterminals at all. This snippet shows an example of invoking a nonterminal:

```
relatives => "[:>random_name] is [:>random_name]'s cousin!".
```

Here the bracketed parts indicate a nonterminal inside the contents of the rule. This bracketed part is replaced by the contents of a random instance of that rule. This works just like you'd expect a context-free grammar to work. In this case, the end result of invoking the "relatives" nonterminal would be *"Clarice is Alice's cousin!"* or some other variation.

Here we are already pushing the limits of a simple context-free grammar. Note how we can't easily replace the word "cousin" in that rule with, e.g., "sister", because we would need to either add different versions of the rule for each gender or remove the concept of gender entirely. This problem can be solved by adding state to the grammar in the form of variables.

3.2 Variables

Variables allow rules to communicate with each other. They are usually set after the contents of a rule, separated by a comma:

```
event
=> "The car broke down.", let broken_car
=> "We ran out of gas", let out_of_gas.
```

Here the "let" keyword tells the grammar to set that variable. We can also set a variable to a specific value:

```
set_weather
=> "It was raining", let weather = "rainy"
=> "It was snowing", let weather = "snowy".
```

A variable is not very useful if we never use it. Here's how a variable can be inserted into the contents of a rule:

```
describe_day => "[:>set_weather]. He hated [weather] days.".
```

This would result in either *"It was snowing. He hated snowy days."* or *"It was raining. He hated rainy days."*. Note how easily we've added some consistency between the rules. However, we're still using the value of the variable verbatim.

What if we wanted to choose a different rule based on the weather but not include it in the contents? Then we would need *conditions*:

```
walk
=> "She jumped over a puddle.", weather == "rainy"
=> "She trodded through the snow.", weather == "snowy"
=> "She walked on.", weather == null.
```

A rule can be chosen only if all of it's conditions are true. The rule is then said to be *valid*. We'll look at other types of conditions later but for now it's only important to know that the == operator is true only if both sides of it are equal and that an unset variable equals null. Therefore, in the previous example the last rule would only be chosen if the set_weather nonterminal hadn't been invoked by that time or if some other rule set the weather variable to null.

The name of a variable (and the nonterminal of a rule) can include both underscores and either lowercase or uppercase alphanumeric characters. However, if the first character of a variable is uppercase, it is considered a local variable. Local variables can only be affected from within the instance of the rule they're set in:

```
set_age => let A = 12
show_age => "[:>set_age][A] [B] old", let B = "years".
```

Note how we left out the unnecessary contents. A rule will default to an empty string if no contents are given.

Invoking the show_age nonterminal above would result in *"null years old"*, because the local variable A hasn't been set in the rule's scope. In turn, locals are also not affected by other rules and are useful for shortening long variable names. This is especially useful when considering *variable tables*, introduced in the next section.

3.3 Variable tables

A variable table is a special type of variable. Instead of a simple numeric or string value, it contains an array of child variables. The child variables can in turn be tables themselves. This means that variables can function like common data structures such as records. The child variables are accessed using dot notation:

```
make_cat => let cat.color = "black", let cat.weight = 100
destroy_cat => let cat = null.
```

If we want to make a list, we can also refer to child variables with a number if we put that number inside parentheses:

```
make_list => "",
   let list.(0) = "Ashley",
   let list.(1) = "David",
   let list.(2) = "Brad".
```

Note that we also split the rule into several lines here. The rule is considered to continue on the next line if the current line ends in a comma.

If we don't want to use plain numbers, we can also use a variable that contains a numeric value:

```
remove_second => let S = 2,  let list.(S) = null.
```

If we want to refer to a child variable that's not a number through a variable, we can use constants:

```
make_city
=> let city.hospital, let city.police_station, let city.sewer
choose_scene
=> city.hospital != null, let scene = @hospital
=> city.police_station != null, let scene = @police_station
=> city.sewer != null, let scene = @sewer
destroy_scene
=> let city.(scene) = null.
```

The != operator specifies that the values on either side of it must not be equal. In this case it means that the rule will never choose a scene that was already destroyed. We will look at @ and the other operators more closely in the next

section.

3.4 Expressions

Simply setting and checking variables have already given our grammar a lot of expressive power. We can get much more if we also include expressions. Expressions are statements that Orcus evaluates and then spits out the result of. They are very similar to common programming expressions. Here is a simple one:

```
(1 + 2) * 4.
```

You can also mix variables with expressions and nest them within each other. These are all expressions:

```
a != b
let a = b + c
let cats.(cats.max - 1) = "last cat".
```

This means that other expressions can be included in the contents of a rule, just like a nonterminal:

```
age => "You are [:>age_difference] than your brother"
age_difference
=> "[you.age - brother.age] years older", you.age >= brother.age
=> "[brother.age - you.age] years younger", you.age < brother.age.
```

The comprehensive list of implemented operators includes:

1. Basic arithmetic operators, including addition, subtraction, division, multiplication and modulo, e.g., $+$.
2. The common assignment versions of the above arithmetic operators, e.g., $+=$.
3. The variable declaration operator *let*.
4. The assignment operator for variables, $=$.
5. Relational operators, e.g., $!=$ and $>$; any expression with a relational operator is considered a condition.
6. Boolean negation operator $!$.

7. Boolean logic operators *or* and *and*.
8. The string capitalization operator, which capitalizes the first character of a string, ^.

In addition, there are some built-in literals:

1. The *null* value.
2. The boolean *true* and *false* values.
3. The rule's *this* value, which is a unique identifier for that specific rule.
4. The *random(X)* value that is equal to a random positive integer from the given range [0, X-1].

Additionally, the *:>* operator we've used in rule invocation is part of a group of rule invocation operators. We go through these in more detail in the following section.

3.5 Rule invocation

Until now we've invoked nonterminals like this:

```
:>name.
```

This is actually only one of the possible rule invocation operators. The *:>* operator tells the program to choose a single rule at random out of all the valid ones. The random choice is weighted according to the number of *conditions* a rule has, so that a rule that has one condition is twice as likely to be chosen than a rule with none. A rule with two conditions is again thrice as likely to be chosen than a rule with one. And so on.

This heuristic assumes that a rule that is rarely valid is more desirable than one that is always available. This results in more appropriate rules being chosen more frequently without restricting the possibility space completely. When we wish to use stricter logic, we can invoke rules with the *ordered invocation operator*:

```
=>weather.
```

The *=>* operator causes the grammar to select the first valid rule. This allows the grammar to imitate the behaviour of if-else -structures:

```
weather
=> "snowy", temperature < 0
=> "rainy", location == @england
=> "cloudy".
```

Here we define that temperature is the primary factor for weather, so that it never rains in sub-zero temperatures, even if we happen to be in England.

The last invocation operator *> applies all valid rules. The contents of the rules are all concatenated and inserted in place of the nonterminal. This is useful when you want to generate lists:

```
inventory => "You are carrying [*>item]"
item
=> "\nmatches", carrying.matches != null
=> "\nflashlight", carrying.flashlight != null
=> "\ncanvas", carrying.canvas != null.
```

Here the items are separated on different lines. We'll look at how to create more natural looking lists in a later section. For now, let's look at how we could make the above inventory handling more elegant with parameters and pattern matching.

3.6 Parameters

Rule nonterminals can also have parameters attached to them:

```
name Object
=> "[Object.name]", Object.name != null
=> "thing", Object.na+,me == null.
```

Here the rule takes a single variable O as a parameter. If that variable contains a variable table with a "name" subvariable, the nonterminal resolves to that value. However, if there is no subvariable, we default to "the thing".

If we want to then invoke that nonterminal with a given variable, we just add the wanted parameter after the nonterminal name:

```
=>name(flashlight).
```

Multiple parameters are separated by a comma and can be any given expression:

```
give_age N A
=> "[N] is [A] years old."
protagonist_age
=> "[=>give_age(\"Tiffany\", 12)]".
```

Note the escaped quotes inside the contents.

We can also use pattern matching inside parameters. This works by giving a parent variable tree from whose children the parameter's value is matched after the parameter name itself, separated by a colon:

```
initialize_family => let family,
    let family.(0).name = "Martha",
    let family.(1).name = "Bertha",
    let family.(2).name = "Ursula"
random_family_member M1:family M2:family
=> "[M1.name] and [M2.name]", M1 != M2.
```

Here both M1 and M2 are parameters that match from the same variable tree. Note the condition excluding matches that have the same family member in both parameters. Invoking *=>random_family_member* after *initialize_family* would then result in, e.g., "Ursula and Bertha".

The different match configurations are considered distinct instances of the rule. This means, among other things, that invoking a pattern matched rule with the *> operator will result in every possible combination of matches to be generated. This makes it easy to iterate on every member of a list, but can cause problems if not handled with care.

3.7 Combined techniques

All the previous sections have detailed a number of techniques available in the grammar. Combining these and using some syntax helpers allows us to write expressive code while reducing the actual text required to a minimum. This is shown most clearly in the *flag operator* and *shared expressions*.

The flag operator *?=* is not an actual operator, but a syntactic shorthand that gets unpacked when the grammar is parsed. The following rules are identical:

```
event
=> "dancing", flags.events.(this) == null, let flags.events.(this) = 1
=> "dancing", flags.events.(this) ?= 1.
```

The latter is unpacked into the former during the parsing of the grammar. The checking and setting of a flag variable is very common during text generation, which makes it ideal for cutting down repetition.

Shared expressions also reduce duplicated code. Consider the following piece of code:

```
event
=> "dancing", flags.events.(this) ?= 1, mood == @happy
=> "singing", flags.events.(this) ?= 1, mood == @happy
=> "karaoke", flags.events.(this) ?= 1, mood == @happy.
```

The rules can be condensed by writing the shared expressions behind the first line, after a *+>* marker:

```
event +> flags.events.(this) ?= 1, mood == @happy
=> "dancing"
=> "singing"
=> "karaoke".
```

The shared expressions are then added to the end of each rule during parsing. This decidedly cuts down on the amount of code that needs to be both written and inevitably rewritten during revisions.

There are also some patterns that don't involve the parser refactoring the code. One involves the fact that a rule invocation is, in fact, just another expression:

```
create_actor => let A,
     let A.name = :>name,
     let A.age = 20 + random(50).
```

Here the actor A's name is the contents of a randomly chosen name rule. This object-oriented aspect can also be used to automate things such as pronoun selection:

```
name A +> let previous_name = A.name
=> "a stranger", A.known == null
=> "[A.name]",  A.name != previous_name
=> "[A.pronoun]",  A.name == previous_name.
```

Here, if we always get the name of an actor by invoking the name nonterminal, we cut down on repetition by referring to them with the appropriate pronoun. We also always substitute the name with "a stranger" if that actor is not yet known. This shows how we can easily standardize and centralize the text generation for different objects without much added code.

Another feature that benefits from this standardization is the handling of lists. Previously, we used line-breaks to crudely separate the items of a list:

```
inventory => "You are carrying [*>item]"
item
=> "\nmatches", carrying.matches != null
=> "\na flashlight", carrying.flashlight != null
=> "\na canvas", carrying.canvas != null.
```

With parameters and the other techniques shown, we can now make it produce more human-readable text with a couple of generic helper rules:

```
next_separator S
=> ".", S == null
=> " and ", S == "."
=> ", ", S == " and " or S == ", "
list_item L I S
=> "[=>list_item(L, I, S)]the [L.(I).name][S]",
    I >= 0, I -= 1, S = =>next_separator(S)
list L => "[=>list_item(L, L.max, null)]".
```

Here the next_separator nonterminal makes sure that after the first item there is a period, an " and " string after the second-last and commas after all the rest. The list nonterminal now works with any variable tree where the children are numbered from 0 to (maximum - 1) and the maximum itself is stored in the max-subvariable. Here's how we would now list our inventory when all items used the object-oriented paradigm:

```
inventory => "You are carrying [=>list(carrying)]".
```

This would then result in, say, "You are carrying the matches, the flashlight and the canvas.". Needless to say this is a much more elegant and extensible solution than writing rules for every possible combination of items. In the next chapter we'll go through some more specific tests of the expressiveness and elegance of this grammar.

4 Language evaluation

In this chapter we evaluate the language based on three use cases, text construction, agent/world simulation and interaction with users or other systems. As per the research question, our main evaluation criteria are the simplicity of implementation and expressiveness of generation.

4.1 Text construction

Text construction here refers to generating text that fits a given structure and presentation. Different types of texts have different inherent structure. For example, a shopping list doesn't have the same form as a newspaper article. Additionally, the same text often has distinct formatting for individual elements, such as the ingredients and instructions in a cooking recipe. Therefore our language should, at the very least, make it possible to have different structures for different texts. However, it should also be simple and flexible enough to support rather than hinder structuring the material.

Here basing our language on a context-free grammar really pays off, as context-free grammars are explicitly designed to describe recursive structures. By varying the density of non-terminal symbols, the author can create as flexible or rigid a structure as they need. In addition, the recursive and reusable nature of the rules makes it easy to create new rules and recombine old ones to support a basic skeleton. The simplest form of an Orcus grammar is, after all, a single rule:

```
description => "It is cold and windy outside.".
```

This is a simpler syntax than even Tracery has, though it lacks the universal support that formats like JSON benefit from. Both can be extended in a similar manner, by simply adding new non-terminals and rules to support those characters. This scalability runs into problems when creating larger and more intricate structures, but this is currently true of most if not all text generation methods. Most developers have to create their own tools for mapping the resulting structure or to make do without.

In any case, we can consider Orcus a success regarding composition. Another aspect that Orcus is specifically suited to compared to other grammar-based methods is data representation. Data representation involves formatting variable

data inside the generated text. Orcus's high level programming language features are useful here. Not only can Orcus grammars include arithmetic calculations, they can also affect rule selection. For example, depicting the temperature in relative terms:

```
weather => "It is [=>temperature] outside."
temperature
=> "hot", temperature > 20
=> "cold", temperature < 10
=> "mild".
```

This is an area that other grammar-based solutions struggle with, requiring either a rigid tagging system or having external functions modify the text. In comparison, the integrated method employed by Orcus does not preclude using external functions while providing powerful, internal and optional tools for performing algorithmic text construction. However, the integration of programming logic really shines when we consider more complex world models in the next section.

4.2 Agent/world simulation

Agent or world simulation is an approach to text generation where the system simulates a physical or social model of a system and reports on the events caused by the system. This is traditionally the domain of code-based approaches, where separate systems are often used for running the system and generating text based on it. In Orcus we can do both (relatively lightweight) simulation and text construction.

Orcus fulfills all the aspects of a programming language. It is Turing complete and contains equivalents of most common algorithmic components such as lists, conditionals and recursion. Orcus is, however, not designed for process heavy systems. This results in poor readability for rules with multiple conditional and effect expressions. This can be alleviated to some extent by cutting large rules into smaller segments, the fact remains that Orcus is not as effective at simulation as more general purpose languages.

Orcus's focus on integration and simplicity does allow for perhaps the simplest method of creating small-scale simulations, to the extent that most larger Orcus grammars start to resemble a simulation of some kind. This is due to the nature

of Orcus's tree-like variable structure and rule selection process. The tree-like variables lend themselves well to object-oriented models of places, objects and people. The tendency to group variables under a parent variable also creates abstractions of state that can be read as parts of an implicit world model. The possibility of creating and processing these models in rules that only handle logic and do not produce output also enables simulation.

More importantly, Orcus's rule selection process is a natural fit for agent action selection. This holds true whether that agent is an actual simulated person or an abstract director persona, though the line is often blurry. One approach is to separate the simulation and presentation rules. This allows the author to clearly delineate the different parts of the system while still benefiting from the shared state and non-terminals between the two halves. It also makes possible to create and expand the simulation organically. Let us consider a simple non-terminal for idle actions in a story generator:

```
idle_action Actor
=> "[^Actor.name] yawns."
=> "[^Actor.name] stares at the floor.".
```

Now, suppose we add some information about actor mood into the simulation. We see that the above non-terminal will seem broken when an angry actor suddenly yawns in the middle of a fight. We can extend the simulation by modifying the rule:

```
idle_action Actor
=> "[^Actor.name] yawns.", Actor.mood != @angry
=> "[^Actor.name] stares at the floor.".
```

Note how similar this is to the event selection in Chapter 3, which could be seen as an example of an author-proxy agent selecting a scene appropriate event. Note also how easy this would be to continue to extend to other aspects until we had a full-fledged simulation. Using Orcus makes for a naturally smooth progression from simple word substitution to computation to simulation, while allowing to separate the model and presentation if the user so wishes.

4.3 Interaction and integration

Orcus can be used for interaction with either other systems or a user. The retention of state between invocations and the rule-based interface makes it easy to have Orcus communicate with either other systems or a user. One possibility is to use Orcus like a text-focused query-based database, saving data from other sources in its variable trees and then invoking non-terminals based on that. This can be done either directly through the data model or through one-off expressions fed into the generator. There are then two alternatives to adding interactivity; the non-terminal based and the selection based approach.

The non-terminal based approach is simpler and consists of creating a set of non-terminals that are only invoked by external components. This could include non-terminals for output, data setup or user choice. In the case of a game, the non-terminal could represent the possible player actions, a list of which the generator can provide to a separate interface that presents them to the player and allows them to choose one. This approach is simple and elegant, but requires that the author specifically design the grammar to support it.

The other approach is to extend the language and allow the user to affect the rule selection process itself. This could apply to all rule selections or involve the creation of a new rule invocation operator, which would mark the selections that the user can influence. An attempt was made to include an operator of this kind to Orcus, but the processing of rules was not implemented with this in mind and it resulted in pernicious bugs. Finally, the conclusion was made that the previous approach was a simpler and less error-prone method of implementing interactivity with no serious disadvantages when compared to a custom rule invoker.

Preliminary tests with a simple kingdom simulator game have proven successful in integrating interactivity with the non-terminal method, with all game logic and state (aside from user interface elements) implemented inside the grammar itself. This is elaborated upon in the next chapter.

5 Prototype case study

5.1 Game overview

In this chapter we examine the case study of an interactive prototype developed with Orcus. This prototype ties together and implements the elements discussed before into a cohesive whole. Although it is not a traditional measurement of text generation tool ability, a game prototype was chosen for this case study precisely because it tests the limits of the state and logic capabilities of Orcus. This case was also chosen for study instead of the more straightforward NaNoGenMo entry mentioned above because it highlights the features not available in the C# version.

In the following sections we will first describe the prototype at a high level and then dive down into the details of its implementation.

The game prototype implemented in this study is a kingdom management simulator. The term "simulator" should not be taken too literally, as the world model of the simulation mainly consists of high level abstractions. For example, the kingdom's population is represented by a single number from 1 to 10.

The player then is tasked with managing these statistics through the use of courtiers and commissioning buildings. In order to accomplis this, during each turn the player is allowed to choose one from a number of actions generated by the grammar. These actions depend on which courtiers the player has in their court and what buildings they have built.

An embedded event generator reacts to the player's actions and the current game state by selecting a random event after every player action, after which the player chooses a new action, followed by an event, etc. This forms the main loop of the game prototype.

To implement this interactive loop with the player the prototype employs the non-terminal method described in the previous chapter. The world model and game content is determined wholly by the script, but select elements such as graphics and player action selection is handled by the game engine.

5.2 Game engine

The game engine mainly deals with the user interface, such as displaying graphics and allowing the player to choose which action (non-terminal invocation) they want to perform next. The interaction with Orcus is executed through two methods: by invocation of ad-hoc Orcus scripts and by a notation generated by Orcus that is then parsed by the engine to know which actions to present to the player, which sounds to play, etc.

When the user interface is created by the engine, it queries for the wanted state by invoking simple scripts such as

```
"Population: [city.population]"
```

to display the city's population. These simple scripts are handled by Orcus by creating a temporary rule with the given string as the rule's right hand side and invoking that rule:

```
temporary_non_terminal => "Population: [city.population]".
```

This method is used to get all the necessary information, such as court members and built buildings, from the world model stored inside Orcus. In addition, there are some specific non-terminals that the engine invokes for specific meta-game actions, such as starting a new game which invokes the "new_game" non-terminal.

However, simple plaintext is not sufficient when the engine needs to know e.g. what text is meant to be the clickable portion of the action and which is the actual non-terminal invoked by clicking on that action. For data such as this a special notation is used. For example, for a list of possible actions, the script generates a string similar to the following:

```
Raise militia#Turns Populus into Military#=>action_raise_militia
Increase patrols#Might lower your Discord#=>action_patrol.
```

In the above example, each action is generated on a separate line and each part of that action's data is separated by a hashtag. The lines are then processed by the engine, with the first part presented to the player as a clickable link, the second part displayed as a tooltip and finally the non-terminal to be invoked when the player clicks on the link is contained in the third part.

Another type of notation is used for text with side effects (such as triggering a change from day to night in the interface) and for triggering dice rolls. In this notation the relevant effect is enclosed between a dollar-sign and a semi-colon like so:

```
next_turn => "[:>random_event][*>give_activity]$advancetime;".
```

In this case the "next_turn" nonterminal which is invoked after every action would case a random event to happen, followed by updating the AI characters' activities and finally resulting in the advancement of night to day or day to night in the user interface.

A similar notation is used to perform dice rolls during actions. A part of the game engine is dedicated to simulating these dice rolls, the results of which are then fed into the generator to deduce the results of that action. This consists of comparing the two sets of dice rolled and whether the sum of the player's set is higher than the sum of their "opponent". A non-terminal specific to that action is then invoked with a parameters to indicate the results of each roll:

```
roll_patrol T1 T2
=> "[^city.captain.name] increases patrols, calming the streets.
[=>stat_change(city,@discord,-1)][=>next_turn]", T1 >= T2
=> "The patrols prove ineffective.[=>next_turn]", T1 < T2.
```

In the above example the parameters T1 and T2 would be bound to the sum of the player's and the opponent's dice, respectively. Thus, if the player's roll was higher than the opponent's the first rule would be triggered, resulting in the player's patrol action to be successful and lowering the city's discord by 1.

Conversely, nothing would happen if the player is unsuccessful as defined in the second rule. In any case, the next turn would be triggered, causing additional changes to the world model.

5.3 World model

The world model consists of variable tables. The main tables are the ones for the city, the map, the player character and the technology tree. These tables are initialized with the following non-terminal "new_game", which is invoked before a new game starts:

```
new_game =>
    =>new_classes,
    =>new_technologies,
    =>new_city, =>new_player,
    =>new_map,
    =>update_build_list.
```

This is a utility rule that does not actually generate any text. All the "behind the scenes" work is performed by rules such as this one. This includes several utilities invoked by the rules invoked by the above; e.g. the "new_player"rule:

```
new_player => let player,
    =>new_actor(:>random_name, @ruler, 1),
    let player = created,
    =>add_to_court(player, city),
    let city.(player.class) = player,
    let player.nature = @enigmatic,
    let player.nature_text = "enigmatic",
    let player.activity = "pondering their rule".
```

The variable table for the player character is initialized by using the utility rule that is the same for all characters, "new_actor". As the player character is special, however, they have additional attributes that other actors lack. They are automatically added to the city's court (another variable table) as its ruler and their nature (a special attribute depicting an actor's personality) is set in stone and not random as it is for other actors.

The utility rule used for all actors defines the scope of their simulated state and takes as arguments the name N, class C and level L of the actor:

```
new_actor N C L =>
    =>create(@actor),
    let created.name = N,
    let created.class = C,
    let created.class_text = classes.(C).name,
    let created.ferocity = 3,
    let created.presence = 3,
    let created.scrutiny = 3,
    let created.age = 15 + random(10) + L * 5,
```

```
    let created.level = 0,
    =>level_up_to(created, L),
    :>give_nature(created),
    :>give_activity(created).
```

Going through the code in order, we first create the variable table using the non-terminal "create", which creates a new variable table of the given type and adds it to the appropriate list:

```
create T => lists.(T) == null,
         let lists.(T).index = -1,
         =>create(T)
create T => lists.(T) != null,
         let L = lists.(T),
         L.index += 1,
         let L.(L.index),
         let created = L.(L.index).
```

This utility makes sure that the list of the given type exists before adding the created variable table into it. Inserting all created entities into their appropriate lists makes it easy to do pattern matching later when you need to find a specific type of actor, for instance.

After creating the variable for the actor we then proceed to set different attributes. Note that we are using the global "created" variable to access the newly created actor as Orcus does not support return values apart from strings. If we wished, for example, to create two entities simultaneously we could bind the first entity to a local variable before creating the second one.

The different attributes are pretty self-explanatory apart from the nature (the actor's personality) and the activity (what the actor is doing). The latter is updated each turn and its main function is to enhance the illusion of life for that character; the player can see each actor's current activity as part of their description which is directly connected to their nature:

```
get_activity_npc A
=> "acting like [:>animal]", A.nature == @insane
=> "ignoring [:>peasant]", A.nature == @arrogant
=> "muttering to themselves", A.nature == @envious.
```

The "animal"and "peasant" non-terminals in turn reference simple lists of things:

```
animal
=> "a squirrel"
=> "a songbird"
=> "a cat"
=> "a dog"
=> "a pig".

peasant
=> "a peasant"
=> "a blacksmith"
=> "a cook"
=> "a farmer".
```

Note that these examples are shortened. The actor creating demonstrates Orcus's flexibility in allowing us to get as technical or simple as we want to and encapsulating the parts into the easy to reuse non-terminals while trickling the generated data down the chain.

The other models are initialized in a similar manner, with utility rules setting default values and creating new variables and variable tables in which to store that state. These range from the extremely technical in the map's case (a 2D grid map generator decides where it is appropriate to create new buildings) to the very data-driven in the technology tree's case (where each building's effects is encoded as data that is interpreted by the building listing rules).

However, they still essentially use the same techniques as the actor model. So, we won't be going through them in detail here. A system that does merit special attention is the event system, which makes the game world seem active.

5.4 Events

As stated before, events represent the world's reactions to the player. In this prototype they are limited to presenting random events tied to specific buildings. These events could just as easily involve specific court members such as an envious captain stealing money from the treasury or a virtuous jailer letting a prisoner go. Using preconditions and pattern matching they could be used in this manner for

simulating artificial intellgence. The limitations of this prototype are due to its incomplete state and should not be taken as a reflection of the expressive power of the language itself.

With that said, there are two general types of events, singular events and event chains. Singular events are simply events that happen based on the game state:

```
random_event
=> "Your harvest has been exceptionally good this year.
         [=>stat_change(city, @populus, 1)]",
         city.built.farms != null.
```

Here we see that one can only get the "good harvest" event when you have built the farms. Additionally, it increases your city's Populus by 1.

An event chain, in contrast looks like this:

```
random_event
=> "An infestation of rats has taken hold in your city.
         [=>stat_change(city, populus, -1)]",
         city.populus > 1, event.rats1 ?= 1, city.built.farms != null
=> "The rats have taken over the sewer as well as the farms.",
        event.rats2 ?= 1, event.rats1 != null, city.built.sewer != null
=> "The cursed rats are now harassing the horses.",
         event.rats3 ?= 1, event.rats2 != null, city.built.stables != null.
```

Here we can see that the events can only be performed once using the *?=* operator. This also makes it easy to have other events as prerequisites for the following ones by using the same flag for subsequent links in the event chain.

One can easily see how this could be expanded to include specific actor traits for AI or add other types of "events" such as a state of war, seasons etc. The prerequisites can be defined as pretty much anything that can be expressed as variables or even other non-terminal invocations.

This results in a possibility space of events that are consistent with minimum effort required on the author's part to make sure they appear in the right order, provided the fidelity of the simulation is not too detailed to keep track of. The author can be as rigid or as freeform with the structure as they wish. In the other extreme they could make each event link back to the previous one in a strictly

linear progression or on the other hand make all events be independent of each other for a more varied experience.

It is this simplicity and elegance that Orcus was developed to express in the first place. This prototype has proven that it can be used to create both simple and relatively complex systems as well as combine them seamlessly into a coherent whole.

6 Summary

In the previous chapters, we have examined previous approaches to text generation and attempted to create a language that could combine the best aspects of each of them. We have hopefully demonstrated how this approach has been successful in creating a language that is both powerful and flexible. We have shown how easily Orcus scales between simple grammars and relatively complex world simulations. We have also seen how this scalability makes Orcus easy to learn, only requiring some knowledge of context-free grammars and basic syntax to get started.

Other similar recent works such as Expressionist [4] have also shown that there is merit to this approach of extending context-free grammars. These other works also demonstrate a failing of Orcus in a sense; the fact that the support for a language is almost as, if not more, important than the characteristics of the language itself. The lack of secondary features such as custom tools and extensive documentation are the main thing limiting Orcus right now.

So, while there are other more minor concerns with the language, for the future it would be most prudent then to either spruce up the supporting material for Orcus or implement its features within a language that has a larger community already surrounding it.

Bibliography

[1] Andrew Butterfield, Gerard Ekembe Ngondi. *A Dictionary of Computer Science* (7 ed.) Oxford University Press, 2016.

[2] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation* (2nd ed.) Addison-Wesley, 2001.

[3] Adam Fedja. Procedural Generation of Text. M.Sc. thesis, Hochschule Darmstadt, 2015.

[4] James Ryan, Ethan Seither, Michael Mateas, Noah Wardrip-Fruin. Expressionist: An Authoring Tool for In-Game Text Generation. *Proc. of the International Conference on Interactive Storytelling*. Springer, 2016, 221-233.

[5] Kate Compton, Ben Kybartas, Michael Mateas. Tracery: an author-focused generative text tool. *Proc. of the International Conference on Interactive Digital Storytelling*. Springer, 2015, 154-161.

[6] Joseph Humfrey, Jon Ingold. Ink. Markup scripting language for interactive fiction, https://www.inklestudios.com/ink/.

[7] Nick Montfort. Curveship's automatic narrative style. *Proc. of the 6th International Conference on Foundations of Digital Games*. Springer, 2015, 154-161.

[8] Aarne Uotila. 72 Cases of the Blackhearts Detective Agency. GitHub repository, https://github.com/Aarneus/blackhearts.