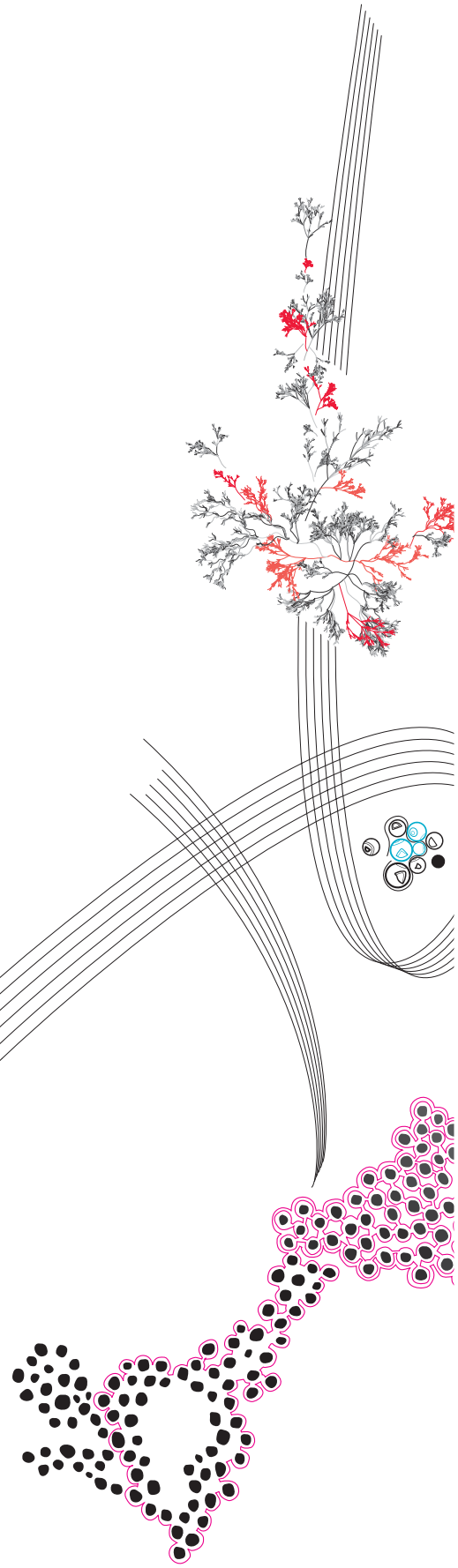


MSc Computer Science
Final Project

Building a Grammar: Generating samples for Weighted Attribute Grammars.

Nick Wolters

Supervisor: Vadim Zaytsev, Marcus Gerhold

April, 2025

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

Grammars are vital in the development of grammarware tooling, where a large set of generated grammars can aid in the development and testing of their applications. However, the existing methodologies of generating new grammars are insufficient to generate a high amount and variety of these grammars. Where either a high number of language samples, positive and negative, are needed to generate a similar grammar in the form of grammar inference, or the implementation of non-CFG features is too hard to achieve. This thesis will research four state-of-the-art generative techniques: Grammarinator, XSmith, transformer networks, and evolutionary grammars. A comparison will be made on both the quality and the quantity of the artefacts, and using two different meta-grammars to differentiate results. The conclusion is that both transformer networks and evolutionary grammar are the most interesting techniques to perform further research, providing either some semantic similarity to meta-grammar or a high number/level of grammatically correct artefacts.

Keywords: Computer Science, Neural Networks, Weighted Attribute Grammar, Attribute Grammar, Weighted Grammar, Context-Free Grammar, Transformer Networks, Genetic Algorithms, Genetic Programming, Evolutionary Grammar

Chapter 1

Introduction

The use of grammar occurs daily, during conversation, writing, and is prone to error in both speech and writing. Grammar is used to convey information, but most importantly, it provides structure to either sentences or lines of text. There are a variety of ways to express grammar $\mathscr{G}$, which can be in the form of natural language, such as English, programming languages, like Java, or Domain-Specific Language(DSL) like SQL. It is the rule that for each grammar there is a related language $\mathscr{L}$, for which a structure is formulated. Each language is structured by rules declared as grammar, where a set of rules creates a sentence, and multiple sentences create an artefact. To illustrate the difference between grammar and language, compare the same two sentences in natural language: "Are you good?" and "You are good!". In the case of language, both sentences are identical since all the tokens(unique words) match, however, the sentence can have two different interpretations, which can be seen in grammar. The first is a question of whether someone is good(i.e. if everything is well), while the other is a definitive statement that they feel well. Similar cases can also be found in both programming languages (different grammar structure of constructors and functions), and DSL(the different grammar rules of MySQL and PostgreSQL).

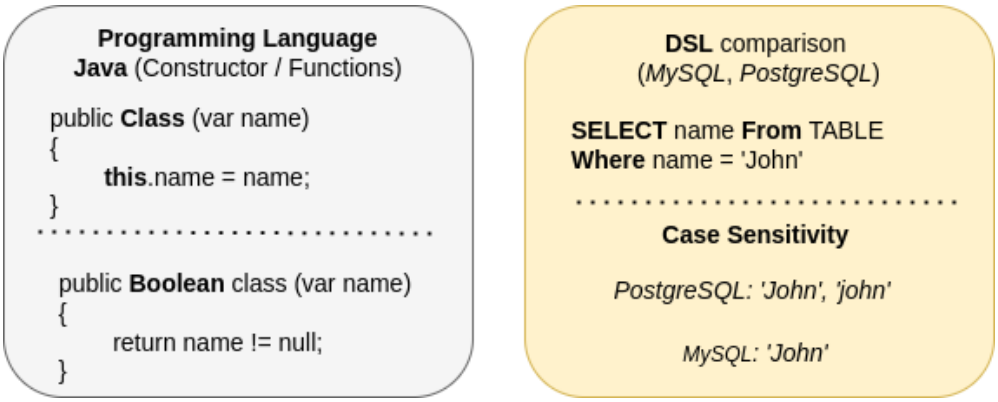
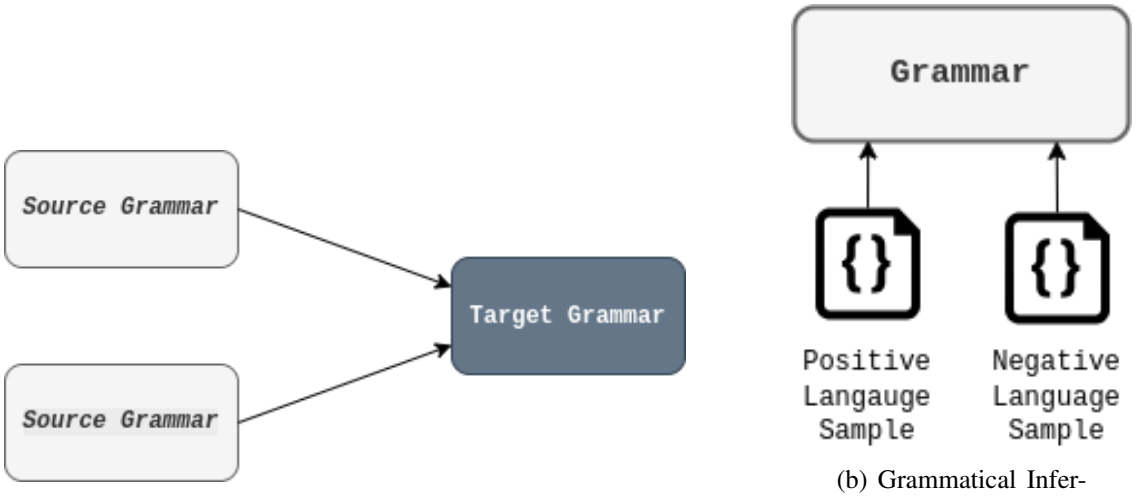


Figure 1.1: Programming and domain-specific language, grammar rule comparison.

How grammar is declared prevents the misinterpretation or incorrectness of information. In most spoken languages, the structure of each grammar is different, but still contains similar elements. Since each natural language will have nouns, verbs, adjectives, etcetera. However, the arrangement through grammar is what distinguishes them. Each individual spoken language or natural language is defined by linguistics, Natural Language Processing [62] and Formal Language / Grammar [20].

This thesis, however, will not focus on grammar for Natural Language, but will focus on



(a) Grammar Convergence

(b) Grammatical Inference

Figure 1.2: Type of Grammar Evolution

grammar for both programming and domain-specific language. The reason is that the content of this thesis focuses on the topic of Computer Science, and specifically, the application of grammar in computing languages. Each programming language and DSL generally has a related grammar, where each language sample s is a combination of a subset of the language corpus $\mathscr{L}$, which has been put into a grammar $\mathscr{G}$, i.e. $s \in \mathscr{G}\ (\mathscr{L})$. One problem arises, however, in cases where the language sample is of a language that is either unsupported or threatened with extinction, they are often referred to as legacy languages.

There are several cases of both public and private institutions that still rely on software systems written in legacy languages. COBOL was a commonly found language in the financial sector. The risk COBOL brings is that these programs handle financial traffic with an estimated value of three billion dollars a day [17]. However, though Cobol was the most commonly used legacy language with 42% still on mainframes, there are also other contenders such as Assembler (37%), and PL/I (22%) [44]. The issue with legacy languages is that they are a fundamental part of the mainframe infrastructure, and it is therefore necessary to perform maintenance for optimal performance. However, there are modern architectures in both software/hardware which are faster and more maintainable, and evolving to a more state-of-the-art language is labour-intensive.

The process of modernising these systems is called software evolution [18], which categorises different types of activities to do so. These range from updating documentation, translating user functionality, and, most importantly for this research, the study of evolving legacy languages into more modern iterations. Other options to run these older languages are to develop a new compiler/parser using compiler design [34], to support more modern architectures.

When a language has to be modernised, it is implied that the current grammar has to be transferred from the language of origin to the modern implementation. To achieve this, one of the solutions would be to look for grammar conversion models and tools, which could be found in the realm of grammarware[57, 112]. The term refers to the modelling and formalisation of grammars, and the advancement of grammar implementations/techniques in topics such as parsing, interpretation and others.

Several grammarware techniques are interesting to research, where two grammatical techniques can be taken as examples: grammar inference [98, 93] and grammar convergence [112]. Figure 1.2 shows two models visualising both of these grammar techniques. Grammar convergence is a methodology to generate an intermediate representation of the grammar, based on the grammar rules from two or potentially more grammars. This is performed by liberally adding

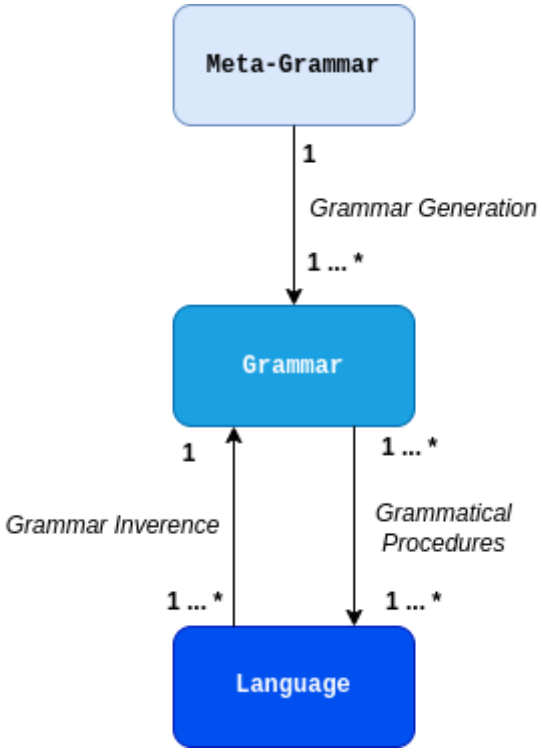


Figure 1.3: Grammar generational model.

transformations between the grammars such that they converge to a shared limit. This process is mostly done by the human-guided system, where a domain expert infers these transitions from multiple grammars to one. Grammatical inference, on the other hand, tries to infer the meaning of a grammar by generalising the resulting languages. This is done either in state space, such as finite automata, or using non-terminal merging in a grammar. An example of this is by Sayilir [93], where the Grammatical Inference of fourth-generational languages(4GLs) is used to generate new positive and negative language samples after inferring the grammar. However, there are potential issues when trying to convert two of the same language but of a different implementation. An example of this is the difference between Mysql and Postgresql, or different versions of the same language.

Both grammars can generate new grammars based on either positive/negative language samples or grammar samples. The downside of both techniques, though, is that the resulting grammar is only one grammar. So what happens in the case where it is necessary to test grammarware tooling, using a set of multiple different, but similar grammar implementations?

To potentially improve grammarware, it is a good option to train a tool/model with more than one specific implementation. An issue that arises when it comes to grammar is that, in comparison to language samples, there is a lack of samples you can test. One of the reasons for this is that once a grammar is written down, it only goes through revisions of the original grammar. In the case of languages, there are multiple different examples since the content of a language can be written in a million different ways, depending on the size of the corpus and the grammar rules. Therefore, this thesis proposes to define a meta-grammar of the original grammar and, through the use of generational techniques, to generate these test samples for grammarware. Both the concept for Grammar Convergence and our proposed meta-grammar result in the structure that is shown in Figure 1.3

Within the realm of Formal Grammar, there are multiple ways to denote all the different grammars, about which specific types will be elaborated upon in Chapter 2. For the research, it was

necessary to choose one grammar for the generation of these grammar samples. A decision was made to use Weighted Attribute Grammar(WAG) [114], as the main form of grammar notation. The reasons for WAG are that adding weights to your grammar makes it possible to specify the probability of different choices of a subsequent grammar rule/terminal state one ends up in. Secondly, attributes increase the expressiveness of the grammar by using additional values to define and declare specific rules, which helps in making grammar decisions. And the last advantage of using WAG is that since it is an extension of Context-Free Grammars(CFG) [22], all possible solutions should be achievable within $O(n^3)$ [102]. More on the particular choice for WAG is also given in Chapter 2.

1.1 Content of the document

The thesis starts with a Background chapter, going into the selection for WAG and its formalisms. Starting by describing the domain through Chomsky's hierarchy of grammar, continuing with the choice for Context-Free Grammar, and explaining both Attribute Grammar(AG) [58] and Weighted Grammar(WG) [21], as each of these is integral to WAG. A brief section will be provided on machine learning(ML) and neural networks(NN) with a few specified techniques. This will contain the different techniques that are part of some generational techniques that are used to create grammar samples.

After providing context on grammar, the research questions will be formulated to specify the specific questions around generating grammar. Based on the research questions, four different approaches to grammar generations have been found and will be given further explanation and results. These four approaches are: Grammarinator [41], transformer networks [104], XSmith [36], and lastly evolutionary grammars(EG) [90]. These techniques were researched, and an explanation will be given on the principles of each technique. Furthermore, there are additional techniques for both transformers and evolutionary grammars, where the different neural networks and the architecture are important to understand transformers. For evolutionary grammars, the fundamentals of their parent subjects, genetic algorithms(GA) [42] and genetic programming(GP) [3], are fundamental to understanding how the generator functions.

The results chapter will provide the implementation of each of the four different techniques. Each of the different techniques will be compared to the criteria on grammatical correctness and a set of other predefined criteria, such as the generation of multiple samples, and generational speed(artefacts per second). Additionally, there are complementary results, such as a syntax highlighter for WAG.

After the given results, some related work in different fields of generation and grammar will be shown, to give a broader context of both domains and where they intersect. This includes potential improvement to EG by adding probability variables called Probabilistic Grammar Evolution [74], ending with a combination of techniques; i.e. transformer with grammar specified Large Language Models(LLM) [31].

This document concludes by answering the research questions and discussing the results. And ending with a section on potential future work on the topic of grammar generation.

Chapter 2

Background

This chapter provides the necessary information to give a basic level of understanding of the topic of grammar. Starting with Chomsky's hierarchy, to show the different levels of grammar and the choice based on complexity and expressiveness. After that, the following section will dive deeper into the chosen level: Context-Free Grammar(CFG), providing the formal specification. Following CFG, different additions to the grammar in the form of Attribute Grammars(AG), Weighted Grammars(WG), and culminating in Weighted Attribute Grammars(WAG) will be formalized hereafter. Concluding by explaining the choice of WAG as the output grammar the generational techniques should produce, and in some case the input grammar to generate the grammars from.

After the definitions of grammar, there is some background research that is vital to understanding the generation techniques that are discussed in the Research chapter 4. This includes the formalization of both Machine Learning and Neural Networks(NN), and a summary of specific NNs such as Convolutional Neural Networks(CNN), Recurrent Neural Networks(RNN), Long Short-Term Memory(LSTM), Residual Neural Networks(ResNet), and Encoder/Decoder Networks.

2.1 Grammars

2.1.1 Chomsky's Hierarchy

To reiterate on the definition of both the meaning of language and grammar, the definition of language is a finite collection of sentences, based on a finite alphabet of symbols, let's call it $\mathscr{L}$. For each language $\mathscr{L}$ the length could be potentially infinite, and therefore the interest is in finite devices (i.e. grammars $\mathscr{G}$) to make an enumeration of possible sentences. To make a grammar $\mathscr{G}$ viable you'll need a class of functions, to give a structure to a given language. Defining these functions can be done in multiple ways, e.g. using unrestricted Turing Machines, or strongly limited condition Markovian sources, e.g. finite automata.

There are several different functions $\mathscr{F}$ to a language $\mathscr{L}$, which is specified by the grammar type. Chomsky in another paper called, "Three Models for the Description of Language" [20], goes over the models that make building a grammar possible, of which transitional an phrase structure are applicable in this case. An example of the phrasal structure is commonly found in NLP, where the relation between linguistical concept are structure inside a tree structure. However, this research is mainly interested in transitional structure where you go from one rule in grammar to others without explicitly referring to a phrasal structure. Each grammar rule consists of two different concepts, non-terminals(NT) and terminals, more on the definitions will be given in the next section. But to summarize an NT is a state in grammar form which it can go to other states or a value; referred to as a terminal. Each non-terminal in the grammar should have at least one function, which declares what the NT does, e.g. go to other NTs or terminals

Most commonly the definition of each of the grammar functions is classified using types, of which Chomsky Hierarchy is commonly cited ad defines four different types [22]. The properties that distinguish the difference between each type of grammar is given as follows:

- Type 3, Regular grammar: Grammar where each of the non-terminal states has a direct terminal, which can be in combination with a maximum of one non-terminal state. Some examples are A $\rightarrow$ a, A $\rightarrow$ aB, and A $\rightarrow \varnothing$, where capital letters are the non-terminal states and non-capitalized letters represent the terminal state.
- Type 2, Context Free Grammar(CFG): A grammar that starts in a non-terminal state and results in a combination of terminal and/or non-terminal state. The syntax of context-free grammar is written as: A $\rightarrow \alpha$, where A is a non-terminal state and $\alpha \in (N \cup T)$, a set of terminal and non-terminal states.
- Type 1, Context Sensitive Grammars(CSG): Similar to Context-free grammar, however, this type of grammar has a context variable, before and after the non-terminal; increasing the level of complexity. The notation for CSG is as follows: $\alpha A \beta \rightarrow \alpha \gamma \beta$, where A represents the non-terminal, α / β represents the context used in the function, and γ all possible terminal and non-terminal states that have been influenced by the rest of the context.
- Type 0, Recursive Enumerable(RE) Grammars: RE can express the set of all possible grammar in a given alphabet of a given language $\mathscr{L}$, such that any form of potential output from a corpus is possible. This also can include recursive series, such as the Fibonacci sequence, though RE can break them of at some point This type is the highest in complexity since it can contain all of the different potential states with the grammar. This makes it also impossible for any size/complexity formalization since the results can be defined as literally everything or a combination of every state, to any other possible combinations of states.

Each of the given types increases in complexity, where regular grammar is restrictive, up until RE where every combination of words within sentences is potentially possible. If you want to parse each of the different types, regular grammar would be the fastest and simplest to parse. Since you only go from one non-terminal state to an empty set, a terminal, or a combination between the terminal state with one non-terminal state. CFG becomes a bit harder but is still solvable since there should always be an eventual terminal state within the grammar for it to be viable. CSG however is as of the writing quite implausibly solvable, because of the sensitivity component. Since the added context can completely change the values when going from one state to another, meaning that there are multiple different resulting solutions to a grammar rule depending on the context.

An example in the English language can be seen through the combination of specific nouns with certain verbs. Take a set of Verb {going, leaving, being} and Noun {home, bridge, John}, with the grammatical rule Verb Phrase -> Verb Noun. Some samples work fine going home, leaving John. However, being bridge, or going John are examples that don't make sense without the proper usage of the English language. The context itself would describe these particular exceptions to prevent wrong usage in not only English but also in other languages, which extends to programming language and DSL as well. In the case of RE, you can go from one word to potentially any other word. This makes it impossible to parse since literally, a sentence can have potentially close to infinite meanings, depending on the size of the subset of $\mathscr{L}$ and the set of all $\mathscr{G}$.

The choice was made to use Context-Free Grammar since the grammar isn't as complex as the lower-order grammar types, but also expressive enough to describe most grammar. However, the CFG may suffer from not having the contextual component of CSG to make the grammar more

expressive, such as the English grammar example. To make the grammar more expressive, there are specialised implementation of CFG. More on one of these grammars will be given in the next section, together with a broader explanation of CFG, and its variables.

2.1.2 CFG and Attribute Grammars

Context Free Grammar

There multiple implementation of CFG, each having their own rules and different extensions that might be interesting for implementation. The first definition is from the one used in the Chomsky Hierarchy. The formal definition of CFG can be described with a 4-tuple, a set of different sets, noted as G = (T, N, P, S_0). Within different research papers, there are many different letters given to these variables, so keep in mind that these might change depending on their use. The variable G is the complete grammar defined as a CFG. Each of the properties can be described as follows:

- T, is the set terminal variables within the given grammar. These terminals mean that there are no further elements to explore and to return a value(s). Each value is defined though typing such as strings, integers, or any other type that has been predefined.
- N, is the set of non-terminal states within a given grammar, and the rule on the set applies $N \cap T = \emptyset$, since there should not be an overlap. Non-terminal, as the name implies is a state that doesn't terminate, but leads to other states, that should terminate. Each non-terminal can either lead to terminal(s) when they are used in function/procedures, into multiple other non-terminals, or a combination of both.
- S_0, is the initial state, or start symbol, which is a part of the non-terminal states, e.q. $S \in N$, where $N \neq \emptyset$. This is because the grammar would directly end in the empty set, which would represent a grammar that contains potentially only values and no grammar rules. The initial state is the beginning point of the grammar from which multiple different terminals and non-terminals will be used during parsing. In the case of a tree structure, this variable will represented by the root.
- P, is the last element and stands for the procedural/production steps, these are steps that are taken to go from one non-terminal state to either a different non-terminal state(s) or a terminal state(s) or any combination. The production rules are strictly defined as relation $\alpha \rightarrow \beta$, where $\alpha \in N$ and $\beta \in N \cup T$. It is always the case that α is a non-empty singular non-terminal, where β can be any combination of terminals and/or non-terminal states. β can even be a reference to an empty string(ε), also called the ε-production rule.

There is a particular way to write down grammar. Take for example some grammar rules to concatenate words together, where G = ($\{S\}, \{a,b\}, P, S_0$). Given the following production rules:

- $S \rightarrow aS$
- $S \rightarrow bSb$
- $S \rightarrow \varepsilon$, or empty set

Example output of the production rule can be $\varepsilon, aa, bb, aaaa, bbbb, aabb, bbaa, etc$. Not only grammars can be context-free, but the same goes for a language, referred to as Context-Free Languages(CFL). For a language to be context-free there should be a CFG(G) where $L = L(G)$ and $L(G) = \{w \in T : P \Rightarrow w\}$. From the example you can say that for CFG there is a CFL, since $L(G) = \{ww^R : w \in \{a,b\}\}$.

Context-free grammars come in many forms, Chomsky created his variation which is more restrictive than base CFG, called the Chomsky Normal Form(CNF) [22]. This grammar is particularity interest since the procedure can be transposed into a binary tree. The grammar has the same 4-tuple but has additional restrictions, such as:

- All production rules start with a singular non-terminal and either return a singular terminal statement or a maximum of two non-terminals. The rule where the ε-production rule applies as terminal.
- A combination of both terminals and non-terminals is not allowed in CNF on the right side of the production rule i.e. $\beta \in N \subsetneq T$
- Lastly each production rule van cannot end with a singular non-terminal on the right side of the equations. An example of this is $Nom \rightarrow Noun$, which states that all noun words in the text can be written as nominal variables. What can be done is that the noun terminals values can directly be referenced by Nom, e.g. $Nom \rightarrow$ '*window*', '*door*'.

The advantage to create these binary trees, is that it has predictable properties for iterations on the tree, such as insertion, shifting, and deletion. It is also used within the real of natural language processing for example in the CYK(Cocke–Younger–Kasami) [33] algorithm. This is an algorithm that parses sentence and give multiple different natural language parse to interpret sentences.

The downside of context-free grammar, that was mention in the Chomsky hierarchy section is that has issues expressing context. However, since context sensitive grammar has a high level of complexity, an alternative approach should be chosen. A solution to question of context could be resolved by Attribute Grammar, which will be elaborated upon in the next section.

$\mathscr{L}_1$ **Grammar**	$\mathscr{L}_1$ **in CNF**
$S \rightarrow NP\ VP$	$S \rightarrow NP\ VP$
$S \rightarrow Aux\ NP\ VP$	$S \rightarrow X1\ VP$
	$X1 \rightarrow Aux\ NP$
$S \rightarrow VP$	$S \rightarrow book \mid include \mid prefer$
	$S \rightarrow Verb\ NP$
	$S \rightarrow X2\ PP$
	$S \rightarrow Verb\ PP$
	$S \rightarrow VP\ PP$
$NP \rightarrow Pronoun$	$NP \rightarrow I \mid she \mid me$
$NP \rightarrow Proper\text{-}Noun$	$NP \rightarrow TWA \mid Houston$
$NP \rightarrow Det\ Nominal$	$NP \rightarrow Det\ Nominal$
$Nominal \rightarrow Noun$	$Nominal \rightarrow book \mid flight \mid meal \mid money$
$Nominal \rightarrow Nominal\ Noun$	$Nominal \rightarrow Nominal\ Noun$
$Nominal \rightarrow Nominal\ PP$	$Nominal \rightarrow Nominal\ PP$
$VP \rightarrow Verb$	$VP \rightarrow book \mid include \mid prefer$
$VP \rightarrow Verb\ NP$	$VP \rightarrow Verb\ NP$
$VP \rightarrow Verb\ NP\ PP$	$VP \rightarrow X2\ PP$
	$X2 \rightarrow Verb\ NP$
$VP \rightarrow Verb\ PP$	$VP \rightarrow Verb\ PP$
$VP \rightarrow VP\ PP$	$VP \rightarrow VP\ PP$
$PP \rightarrow Preposition\ NP$	$PP \rightarrow Preposition\ NP$

Figure 2.1: CFG Vs CNF.
[51]

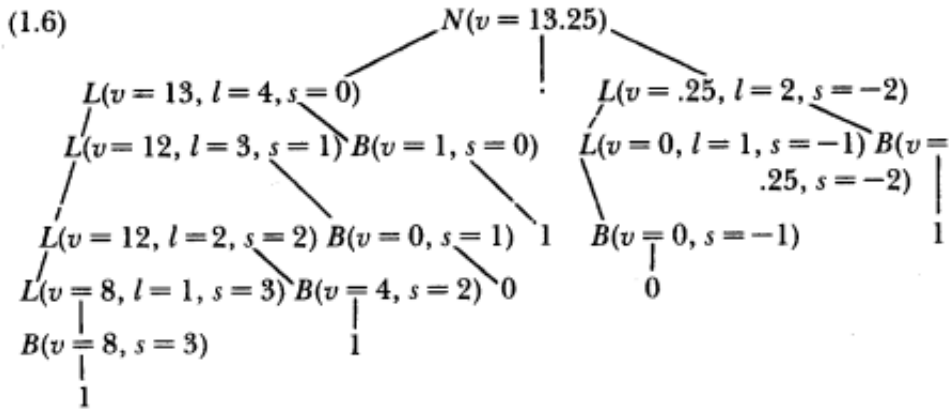


Figure 2.2: Context Free Language examples.
[58]

Attribute Grammars

The formalisation of attribute grammar is written in the paper "Semantics of Context-Free Languages" [58]. The purpose of attributes is to potentially provide attributes for non-terminals in a CFG, which means that they play a part during the parsing of a grammar. To better illustrate the grammar, let's take an example from the paper to convert binary numbers to decimal. Where the binary value 1101.01 is transferred to the decimal value of 13.25, and vice versa.

In this example, there are three symbols: B(bit), L(list), and the number(N). Besides these symbols, there are a few declared rules:

- Each bit has a value attribute v(B), which is an integer
- Each list has a length attribute l(L), which is an integer
- Each list of bits has a combined value attribute of v(L), which is an integer.
- Each L has a "scale" attribute s(L) which is an integer.
- Each number has a value attribute v(N), which is declared as a rational number $\mathbb{Q}$.

In these examples, you can see that each of the non-terminals has a separate attribute(s) that is declared. In this grammar, each Binary and number has a direct value(v) in decimal, while L has both the decimal value and the length of the list, e.g. how many items it can still traverse. For example, the B(v=1), this means that a value v with integer one is attributed to non-terminal B. While the length and value attributes of a L(v=6, l=3), the value at this point in parsing is six, and the height level of the tree is three. Each of the different relations and assignments to both values and lengths can be found in Figure 2.3.

Formally, for each attribute grammar $\mathscr{G} = (V, N, S, \mathscr{P})$, there is a symbol X $\in$ V, the set of terminals and non-terminals. Where $N \subseteq V$, are the non-terminal states of the grammar. Such that there is an associative finite set A(X), which are the related attributes of each of the non-terminal symbols. The attributes can be separated into two different categories: *synthesised* A_0 and *inherited* attributes A_1, where there are no overlapping attributes $A_0 \cap A_1 \in \emptyset$. The difference between the two attributes is the direction the value is propagated. In the case of synthesised attributes, values transfer is from child to parent, and is inherited from parent to child. The grammar starts by defining the attributes for the initial value S, where there are no inherited attributes $A_1(S) \in \emptyset$,

	Syntactic rules	*Semantic rules*
	$B \rightarrow 0$	$v(B) = 0$
	$B \rightarrow 1$	$v(B) = 2^{s(B)}$
	$L \rightarrow B$	$v(L) = v(B),\ s(B) = s(L),\quad l(L) = 1$
(1.5)	$L_1 \rightarrow L_2 B$	$v(L_1) = v(L_2) + v(B),\quad s(B) = s(L_1),$
		$s(L_2) = s(L_1) + 1,\quad l(L_1) = l(L_2) + 1$
	$N \rightarrow L$	$v(N) = v(L),\quad s(L) = 0$
	$N \rightarrow L_1 \cdot L_2$	$v(N) = v(L_1) + v(L_2),\quad s(L_1) = 0,$
		$s(L_2) = -l(L_2)$

Figure 2.3: Semantics and Syntactic Rules.
[58]

since the initial state has no parent. And the synthesised attribute set is empty when the initial state starts with a terminal symbol. Also, each attribute $\alpha \in A(X)$ has a potentially infinite amount of possible values V_α, for which eventually one value will be selected in the derivation tree.

Within the sample attribute grammar in Figure 2.3, two variables show an example of a synthesised and inherited attribute. To start, the attributes v and l are synthesised, since the value and length are calculated/synthesised in each of the leaf nodes and will be done for each node, till the root element is reached. The calculation of the length is independent and increased by one for each parent node visited. However, to calculate the value of a binary at a specific index, you need to scale. This is where the inherited attributes come, in which they take the opposite path from the root to the leaves. In this case, it takes the synthesised length variable that is used on the right side of the equation to calculate the fractional values. On the right side, starting with 0 and increasing each left child, reverse the value of s with 1. Which scale attribute gets used in the calculation, e.g. the fourth element binary value is 1, and s = 3, giving the value of $v(B) = 2^{s(B)} = 2^3 = 8$. For an attribute grammar to be valid, it has to adhere to the following properties:

- $A \rightarrow \alpha \in P$, meaning that each attribute in the list of attributes is linked to at least one element in the set of procedures
- $a = a_1 \ldots a_n, \forall_i, 1 \leq i \leq n : a_i \in (V_n \bigcup V_t)$, implying that for every symbol in the rules, there is either a terminal or non-terminal symbol.
- $A, a = f(a_{j1} \times a_1, \ldots, a_{jm} \times a_m)$, where $\{a_{j1}, \ldots a_{jm}\} \subseteq \{a_1, \ldots, a_n\}$. This means that for every value of a, there is a function that will be applied to the attribute value based upon the chosen non-terminal A.

Furthermore for the set of productions rule $\mathscr{P}$ has for each p-th production the following property: $X_{p0} \rightarrow X_{p1} X_{p2} \ldots X_{pnp}$, where $n_p \geq 0$, $X_{p0} \in N$ and $(X_{pj}) \in V$ for $1 \leq j \leq n_p$. This means that for each p-th production, there is a combination of non-terminal and terminal variables, each potentially has its related attributes.

An example of the use of attribute grammar can be found in Henriques et al. [38], to create a compiled piece of Java code, based on a class diagram. Firstly, the class diagram is converted

(6) [$_S$[$_{NP}$ they] [$_{VP}$[$_{Verb}$ are flying] [$_{NP}$[$_N$ planes]]]].
(7) [$_S$[$_{NP}$ they] [$_{VP}$ are [$_{NP}$ [$_{Adj}$ flying] [$_N$ planes]]]]

Figure 2.4: Chomsky Ambiguous sentences. [22]

through the use of context-free grammar, defining both the terminal and non-terminal states. After the states are defined, particular attributes are assigned to keep track of certain states.

CFG can also be more explicit by other means than providing attributes. One of these methods is to use probability to make decisions in parsing trees, called Weighted Grammar. The next chapter will expand more on the topic.

2.1.3 Weighted/Stochastic Grammar

Grammar can be written in different ways, but when they are executed, there might be cases in which you have overlapping grammar rules, see the case in Figure 2.4. In this case, the non-terminal and terminal are the same, however, other grammar rules were implemented in both cases. A potential solution would be to add some value to indicate the likelihood of a parse between grammar rules. One of these implementations goes by many names: Stochastic Grammar, Weighted Grammar(WG), or Weighted Context-Free Grammar(WCFG). The concept was introduced by Chomsky and Schützenberger [21], in which they added so-called algebraic theory to context-free grammar. The basic idea is to take grammars($\mathbb{G}$), and vocabulary ($\mathbb{V}$) and combine this with weights. The vocabulary itself is a finite set of words belonging to a specific language, be it linguistic or programmatic. Each grammar contains its procedures (P), where for each P there is a pair ($sigma, \mathbb{G}$), where *sigma* is a string that will be transformed by the grammar. The main reason for adding probability to grammar relates to the topic of ambiguous parsing.

The solution to making these grammatical choices can be performed by adding weights to parsing, such that the parser can make decisions based on probable parsing rules. In the example, we can add a higher weight for the combination of Adjectives(Adj) and Nouns(N) as a Noun-phrase(NP) and then upgrade an N directly to an NP. In this case, option (7) will be taken as the parse and not (6). To prevent the potential of randomised parsing, the variable n is introduced, which is a numerical value representing ambiguity. This number is paired up with the string *sigma*, giving a pair of (σ, n). The combination of *sigma* and n means that the string generated through the grammar has n levels of ambiguity. The term ambiguity can express the value as a weight value, which implies that it can also be expressed in the form of probability. Let's imagine two different potential parses one 'A' with a weight of 5 and the other 'B' of 10. From this, we can deduce that the parse from B is more likely to occur, and if we would treat both probabilities that $A = \frac{5}{(10+5)} = \frac{1}{3}$ and $B = \frac{10}{(10+5)} = \frac{2}{3}$.

Chomsky and Schützenberger defined multiple algebraic functions, of which a few of these definitions are vital for understanding the formal basis for WCFG. Starting of finite V contains a terminal vocabulary(V_t) and a non-terminal vocabulary(V_n). Presuppose there is a set of all terminals $F(V_t)$, then $\mathscr{L} \subset F(V_t)$. Now let's consider that there is a potential string $f \in F(V_t)$, which is mapped to some value r, such that $< r, f_i >$. The mapping onto r then represents some power series, in the non-commutative variables x of V_t. Giving:

- $r = \sum_i < r, f_i > f_i = < r, f_1 > f1 + < r, f_2 > f_2 + \ldots$

For $f_1, f_2, \ldots$ is an enumeration of all strings that are available in V_t. To check if these strings are supported, it makes use of the function $Sup(r)$, where the output should not be zero. Therefore,

- $Sup(r) = f_i \in F(V_t) | < r, f_i > \neq 0$

It is explicitly a non-zero value, which means that the value of $Sup(r)$ can also be negative if they have a negative correlation. However, a positive power series can be made by changing the comparison parameter to $< r, f_i > \geq 0$.

A weighted context-free grammar works by weighting the edges of each procedure in the grammar. Let's say that for each procedure, we have a string $f \in F(V_t)$, and there also exists a weight parameter, which is a numerical value $N(\mathbb{G}, f)$. The value of the weight itself is dependent on the implementation of the grammar itself and always is a positive value in case that $f \in L(\mathbb{G})$. We can also express it in the power series where for $r(\mathbb{G})$, provides $< r(\mathbb{G}), f >= N(\mathbb{G}, f)$, where $< r(\mathbb{G}), f >$ is the coefficient for the power series of F in $r(\mathbb{G})$. The coefficient itself is only zero if the grammar can't generate f, and one if only one possible generating sample in $\mathbb{G}$, two for two different interpretations, etcetera.

The first implementation of WCFG can be found in a paper written a decade later by Salomaa [92]. In this paper, Salooma took Chomsky's CFG and combined it with probability theory. It still uses the CFG four-tuple; however, it added new variables as well, written as $G_w = (G, \delta, \varphi)$. Where G is the CFG four-tuple, δ is the initial distribution, which contains all non-negative variables. While φ is a vector of probabilities given to each of the different procedures, where the value of each is $\varphi : P \to [0,1]^{[P]}$

The weighted grammar G_w can also be expressed as a type of language where $L = L_s(G_w, \eta)$, where L_s is a stochastic type grammar. Also η is used as a cut-point where it goes from the initial state to some singular terminal state, where the probability of at least one path has the property $\eta > 1$. Another relevant property is that for every probabilistic grammar where $\eta > 0$, it follows that the language $L_s(G_w, \eta)$ is finite.

There is, however, a much smaller implementation written by Rabin [84], which is related to the subject, but is called probabilistic automata. Instead of using the probability as extended grammar, on top of CFG, it is based on non-deterministic finite automate a 5-tuple $(Q, \sum, \delta, q_o, F)$, where $Q, \sum$ are finite states and input symbols, δ are transition functions $\delta : Q \times \sum \to \rho(Q)$, Q_o the initial state, and F a list of the final states. In the case of probabilistic automata, they add to the transition function, and the initial state is replaced with a stochastic vector giving the probability in a given state. These are mentioned since they helped to influence the creation of some similar implementations in grammar, written by Ellis [28]. The weighted grammar given in the paper is still a four-tuple CFG, however, it directly assigns the stochastic value to an already existing variable. The tuple is $\mathbb{G} =< N, \sum, P, \Delta >$, where $N, \sum$ are the sets of non-terminal and terminal, $\Delta \in \mathbb{R}^N$ is a vector of stochastic variables combining both the δ and s_0 into one set, and P is the procedures, with each given a weight variable. An example of a procedure in this grammar is the following:

- $n_i \xrightarrow{w_{ij}} \sigma_j$, where $n_i \in N$; $\sigma \in (N \cup T)^*$ and $w_{ij} \in \mathbb{R}$
- Implying, $P \subseteq N \times (N \cup T)^* \times \mathbb{R}$

Weighted grammars are also an integral part of weighted attribute grammar, which is combined with attribute grammars and will be elaborated upon in the next section.

2.1.4 Weighted Attribute Grammar

Both attribute grammar and weighted grammar give more expressiveness to context-free grammar, which aids in the parsing of this grammar into languages. However, joining both would create an even more expressive grammar, which leads us to the topic of Weighted Attribute Grammar[114].

An example of trying to model using WAG is the "binary exponential back-off" algorithm, which is an algorithm to prevent collision in a network where two or more entities are using the same resource. The simplified version has the following properties: the first step is that two communicating entities are connected on one channel, with messages from each being sent one at a time. Secondly, it chooses a frequency band and starts sending messages, and is successful if the other system is not using the same frequency. The message gets corrupted if it is used by both entities, meaning that each entity will back off. Lastly, after backing off, each entity doubles its range and goes for the next attempt. A visual representation of the model can be seen in Figure 2.5

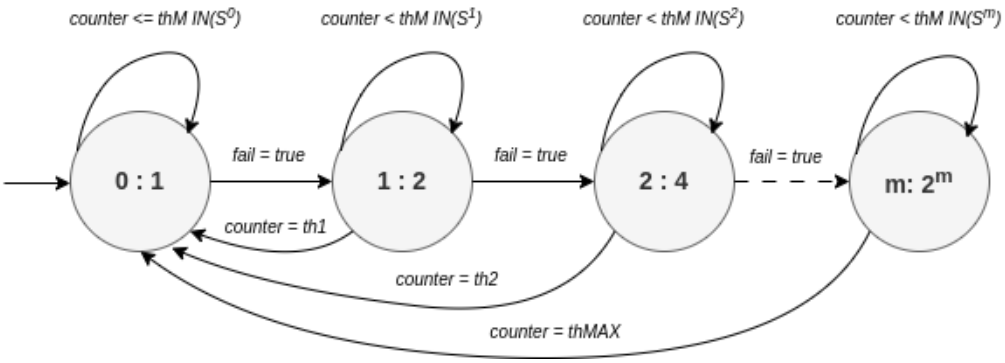


Figure 2.5: Model of binary exponential back-off.

This system can be implemented through an adaptable model since the changes need to be made based on the collision at run-time. To model this, some probabilistic methodology is used to validate the choice and calculate the probability of a sequence of messages being sent intact. The expression of the system in language can be done by creating two instances γ and γ_β where γ is a correct sentence and β is its probability. Where [1,2](25%) describes entity one as on channel one and entity two as the second, which has a 0.25 change, and 2,1 as the reverse with an equal amount of chance. Where a collision is [1,1];[1,2](6,25%), meaning collision on channel one and the second cycle choosing channel 2 for the second entity. Making a grammar for all cases is impossible since, in the end case [5,2], you need to provide all possible back-offs that preceded, which, after each of these, increase in complexity. Only when bounding the model to a condition size n, will it become possible to encode having $\sum_{i=0}^{n} 2^{2i}$ production rules. A better grammar to handle the productions would be to use attribute grammars, however, this will need to take into account that the probabilities have to be calculated separately. Using semantic predicates, e.g. formulae that can fail production rules that potentially create undesirable results after computation, will reduce the production rules to $4+n$. When combining both attribute grammar and probability, i.e. weighted grammar, it has the best of both worlds. The combined version of it makes WAG.

The formalised definition of WAG is given in both Latin and Greek letters, where order tuples are represented by Greek capital letters, mappings of lowercase Greek, and Latin letters are unordered sets. Otherwise, there is $\mathbb{T}$ which is a set of a non-specifically defined type and can be of integer, real, string, lists, and objects. WAG is defined as a tuple of $\mathfrak{G} = \prec \Gamma, \Omega, \Lambda, \Phi \succ$. Where Γ represents the CFG, Ω the weight-related component, Λ the attribute component, Φ representing computational formulae. Each of these has its tuples, except Φ gives computations, like $\langle x := 0 \rangle$.

The first set in the tuple context-free grammar is denoted as $\Gamma = \langle N, T, P, s \rangle$. This includes the non-terminals(N), terminals(T), production rules(P), and the initial state (s). WAG includes colour to describe specific sets as either being of terminal (red) values, or non-terminal (blue), such as the initial state.

The weighted component tuple is $\Omega = \langle \omega, \beta \rangle$ where:

- $\omega : P \to \mathbb{T}$, which is a set of the types for each of the weighted components, which have to be of a comparable type.
- β is the assignment of a weight to each of the production rules (P); however, the co-domain is dependent on input where $\beta(p) \in \omega(p) \cup A$

The last tuple is the attribute component defined as $\Lambda = \langle A, \tau, \kappa, \pi, \sigma \rangle$, where:

- A is a set of attributes.
- $\tau A \to \mathbb{T}$ assign type to each of the attributes.
- $\kappa : N \to A^*$ specifies each inherited attribute for all non-terminals, is passed by a parent; these attributes are denoted with $x\downarrow$
- $\pi : N \to A^*$, these are synthesised attributes, for instantiating nodes to their parent; attribute donated as upward arrow $x\uparrow$.
- $\sigma : N \to A^*$, refers to attributes shared amongst multiple nodes of the same Non-terminal. It is not a part of the initial attribute grammar, and these attributes are also referred to as static attributes.

Another property WAG has is that every CFG can be transferred into WAG, with the following properties:

- N, T, P and S are similarly formatted as the base implementation of context-free grammar.
- ω and β are empty having no weights or associations.
- A = $\varnothing$ is empty, trivialising all components of Λ.

Proving that all context-free grammars can be made into WAG, but not in the opposite direction. Previously, the example of binary back-off was mentioned as a system to model. Implementing this using WAG leads to the following result:

- $S \xrightarrow{1} P\{[1,1]\}$
- $P\{\downarrow v\} \xrightarrow{1} E\{\downarrow v, \uparrow l\} E\{\downarrow v, \uparrow r\} C\{\downarrow v, \uparrow l, \uparrow r\}$
- $P\{\downarrow v, \uparrow r\} \xrightarrow{v_i/|v|} \varepsilon\langle r := i \rangle$
- $C\{\downarrow v, \downarrow x, \downarrow y\} \xrightarrow{x=y} \downarrow x', ' \downarrow y'; ' P\{dup(v)\}$
- $C\{\downarrow v, \downarrow x, \downarrow y\} \xrightarrow{x \neq y} \downarrow x', ' \downarrow y'. '$

In this model, there are four non-terminal states: the initial state (S), protocol (P), entity (E), and channel (C). The first line defines two different entities that need to communicate with one another. The protocol itself is defined by inheriting v results in two different entities E, and a channel between these two entities in C. Each entity inherits the value v from the protocol, and the value for the communication channel is inherited and chooses one of the channels in the channels list. As for choosing the channel, there are two options:

- The channels of both different entities have the same value, e.g. $(x = y)$, then double the range and let both x and y select a new potential channel.
- If both channel x and channel y are completely different channels, e.g. $(x \neq y)$, then the values can be passed to each of the different entities.

There are several implementations of WAG, one generator of Rust code based on WAG called Weighted Attribute Grammar-Oriented Notation or WAGON [27]. A second implementation for an IOT application which uses WAG to implement a chatbot, called WAGIOT [97].

WAGON is a Domain Specific Language, which is specially made based on the WAG grammar. The purpose of WAGON is to convert WAG Grammar and parse it into executable Rust code. One of the examples given in the thesis is a Pokémon grammar, which represents elements of the trading card game such as stats, name, and typing. WAGON itself is also the base for WAGIOT thesis, which defines an IOT Grammar, such that a conversational bot can facilitate the interaction between IOT devices and their user, use human speech and make decisions, such as turn a light on or off. An example of the communication of WAGIOT is seen in 2.6

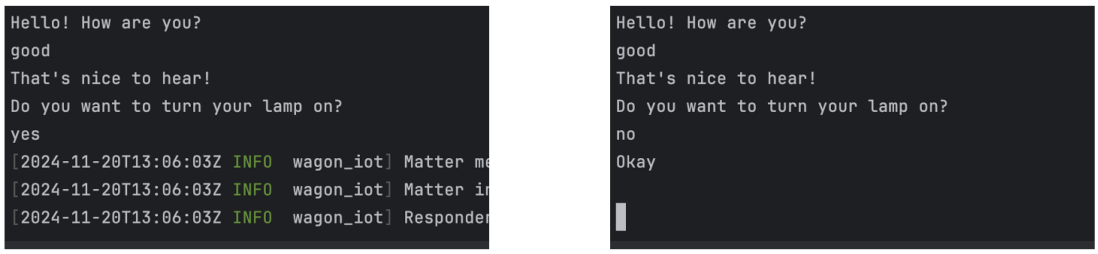


Figure 2.6: Two WAGIOT examples.

To conclude the section on Grammar, WAG will be used as the syntax for are meta-grammar input, to generate new grammar samples. The next chapter will go into the research question and will specify WAG as the chosen grammar, but will also ask questions about what generational techniques/implementations are needed. Some of these techniques are found in neural networks and are included in the research. To give more context to these research techniques, the following section provides a summarised explanation of the basics of Neural Networks and includes specific techniques that are used in the chosen generators.

2.2 Artificial Intelligence, Machine Learning & Neural Networks

This section is on the use of Artificial Intelligence(AI), Machine Learning(ML), and Neural Networks (NN). The topic comes up when researching different techniques to generate artefacts, which in our case would be grammar. This section is used to provide summarised information on different techniques, most of which are neural networks, that are part of the generational algorithms. Start with an introduction on the topics of machine learning and neural networks, by giving them a formal definition, and in the case of machine learning, define the three different types. Then will go over several neural networks that are used in the generational algorithms.

2.2.1 Formalising Machine Learning and Neural Networks

The topic of machine learning is quite broad, and to concretise the application to the thesis, we will provide some fundamental concepts. Machine learning itself is a subset of AI [69], which is defined as the automation of different processes using computers. The term doesn't necessarily mean that a computer has the same capacity as the human brain, but that it can mimic human tasks through calculations. The term Machine Learning has been attributed to Arthur Samuel [105], where it is described as the following:

> "Field of study that gives computers the ability to learn without being explicitly programmed."

The particular difference between the two is found and the end of the quote: "without being explicitly programmed". This means that the inference on how the process should be performed is explicitly learned, and not programmed in advance. Examples of the implementation of machine learning are computer vision [116], routing [59], and speech recognition [85]. The implementation of machine learning can be achieved by selecting one out of three different types. These types consist of learning: supervised learning [89], unsupervised learning [95], and reinforcement learning [50]. These types of machine learning are summarised as follows:

- Supervised Learning: This type of machine learning is used by guiding the algorithm to some preferred outcome of the system, with the use of a set of predetermined inputs and outputs. There are a total of five steps to supervised learning that are used in order:

  1. Define a data set that has both predefined inputs and outputs, which makes both testing and training possible.
  2. Separate the original datasets into two, one training set to train the machine learning algorithm and one to test.
  3. Optional step, encoding the input for the machine learning algorithm by transposing it, in the case of an image, take the pixel matrix of the image as the format.
  4. Select a type of machine learning algorithm, i.e. support vector machine.
  5. Run the algorithm, and depending on the size, run all possible inputs. In the case of sizeable input, define a cut-off point, or in the first step, take a smaller subset of the input.

- Unsupervised Learning: The opposite of supervised learning, meaning the system is agnostic about the outcome of the system, and may provide all types of different results. However, the result still depends on the amount/values of the input and the model/algorithm. The advantage of using unsupervised learning is that the results are unfiltered and unlabelled, meaning that the raw data can be used to analyse trends or patterns after a model has been executed.

- Reinforced Learning: The last form of learning is Reinforcement learning, which is a technique of learning to promote the greatest cumulative result. This is done by training over multiple iterations and encouraging certain decisions in the model/algorithm. The best way to describe reinforcement learning is to show how reinforcement learning is through the use of the following visual representation of the model, see Figure 2.7. Though it sounds similar to supervised learning, the difference is that reinforcement doesn't have a defined output to reach, but uses some predefined reward value to nudge the result to some preferred outcome. Examples of implementing reinforcement learning are IBM Deep Blue [15].

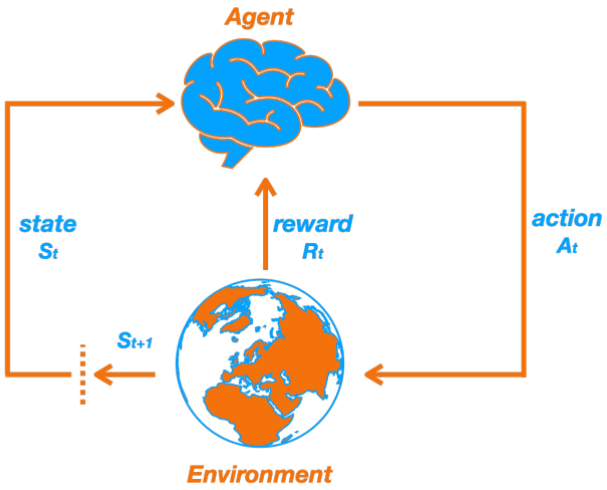


Figure 2.7: Example selecting training and test sets [13]

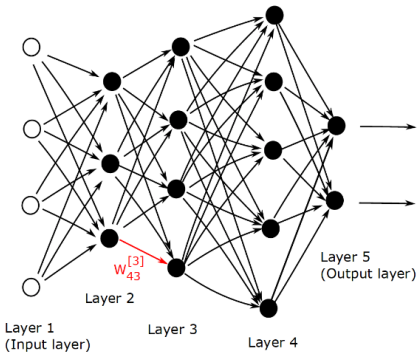
The generation of the grammar will be done through unsupervised learning. Not because it is the best method, but because the generative networks that are used make use of unsupervised learning, such as encoders and decoders [66]. The reason is that unsupervised learning doesn't use predefined labels, meaning that these results are not trained for a specific outcome. There are compromises of these types called semi-supervised training [103], but it is not relevant for this research.

Neural networks Neural Networks [61, 115], also referred to as Deep Learning (DL) [64, 47], is a subset of machine learning which uses layered networks to perform probabilistic computations on a set of inputs. A common example of drawing these neural networks can be seen in 2.8a.

Neural networks can be found in multiple applications, with classification being their most common use case. The reason that neural networks are used for classification is that in a neural network, you can generalise characteristics by embedding the attribute in each node within the network, in the form of weights. All of these layers of nodes together are an amalgamation of these characteristics and are commonly referred to as perceptrons. After training the network, it will be tested and an accuracy value will be calculated by: $\frac{C_g}{C} = accuracy$, where C is the set of all classifications and C_g contains all good classifications of the set. Common examples of classification are ImageNet [61] and CIFAR [60], used to classify images.

The values of each node in a layer are based on the sum of input weights $\sum_{i=1}^{n} x_i$, plus some bias, which is a fixed value. The value itself gets added to an activation function [46], which is are mathematical function. Activation functions introduce non-linearity into the network, providing the ability to learn and represent complex patterns in the data. Common activation functions

(a) Weight relations in neural networks [13]



(b) Output calculations in neural networks

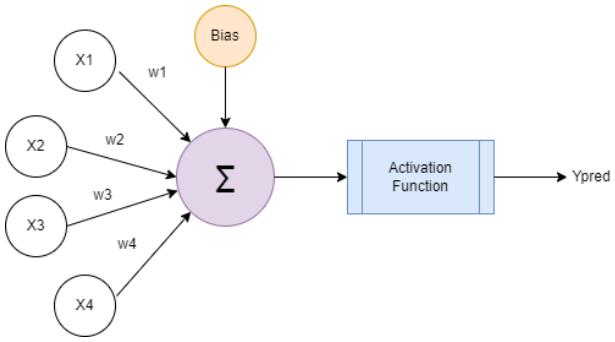


Figure 2.8: Basic Neural Networks

and their mathematical equations/functions are Sigmoid, Tanh, Rectified Linear Unit (ReLU), and Leaky ReLU.

Each of these nodes will propagate its variables to the next layer until they reach the output layer. How many layers and how deep the network goes depend on the architecture of the network. This architecture will hinge on the use case of each of the neural works. When the architecture of a network has a high number of layers, it is called a Deep Neural Network(DNN) [9]. The depth of a neural network correlates with the complexity of the classification, in the sense that more abstract concepts would contain more 'features'. For example, the topic of image processing, where a low layer count can identify the contours of an object, but by increasing the layer count, it can generate better classification on what type of object something is, e.g. vehicle, food, or computer. However, while adding more layers to a neural network helps increase potential accuracy for classification, it also increases the complexity of calculations and makes proper classification slower. Another issue that is known to occur is overfitting. This is a problem of training so well on the initial train, that it overgeneralizes, such that some characteristics get removed and correct classifications get rejected. Which will eventually result in a decrease in the prediction in the test set. The training of the neural network itself is done using the weights and values of the nodes, and these values with backpropagation [64, 40]

Backpropagation works by training the networks to an expected output and updating the perceptrons to conform better to the output. Let's take Figure 2.8a as an example. Within the figure, the updating of neurons is done by calculating the cost function, which compares the output layer of the network after the activation function $\alpha(L)$ with the expected output y. The cost function of a singular node in the output layer L is: $C = (\alpha(L) - y)^2$. The larger the difference between the generated output and the expected, the higher the cost; it will increase the value quadratically. To lower the cost function, the values of the weights and biases need to change to reflect such that the result in $\alpha(L)$ equals or is close to that of the expected value y. To find the most optimal point, it needs to calculate the derivative of the cost function concerning the weights. This is denoted as $\frac{\partial C}{\partial w(L)}$.

In most cases, there is a network with a depth of size two or more, and the output can be multiple. In that case, the cost function is divided into multiple smaller costs per output node. The reason for gathering the derivative of the cost function using weights is that you want to find the gradient of the cost function. This process is called gradient descent. To calculate the gradient of the cost function ∇C, we need the values of all features of the output such that $\nabla C =$

$[\nabla C_0, \nabla C_1, \ldots, \nabla C_n]$. to reflect this we need to sum over all the input neurons. Which results in $z^{(L)}$ being rewritten as $z_j^{(L)} = \sum_{n=0}^{D(L-1)} w_{jn}^{(L)} a_n^{(L-1)} + b_j^{(L)}$, where is D is a function to get the depth of the previous layer. The activation function of the layer itself does not change only by adding the j-th output neuron such that $a_j^{(L)} = \sigma(z_j^{(L)})$. Lastly, the cost function will need to take into account that all the other outputs will be influenced by the other outputs that will be changed as well by updating the weights and activation of the previous nodes. Therefore, the first cost function of the output is written as: $C_0 = \sum_{j=0}^{n_{L-1}} (a_j^{(L)} - y_j)^2$.

The cost function calculation through backpropagation is needed to update the weights, using a technique called stochastic gradient descent [11](SGD). In mathematics, gradients are used to find the fastest rate of growth in functions, which in a graph would be the local or global maxima. However, when taking a negative gradient, you will find the values for the greatest descent in the function to the local or global minima. With each gradient, you will descend the slope until a minimum has been found. After calculating and applying the gradient, the weights get closer to the minimum, however are not there yet. So this process has to be repeated until the gradient is zero or close to zero. The formula for the new weight update is the following:

- $\Delta w = w - \eta \nabla Q(w) = w - \frac{\eta}{n} \sum_{i=1}^{n} \nabla Q_i(w)$

The w is the set of weights, where old are the values before backpropagation and new are after SGD has been applied to the set of old weights. $\nabla Q(w)$ represents the derivative of the cost function, which is calculated by the aforementioned mathematical functions. The last variable is η, which represents the variable called learning rate. This is a value is a rate which can either increase/decrease the strength of the calculated cost function. The best approach for learning rate would be to make the learning rate, sometimes referred to as gain, adaptive [96]. This means that initially, the gain can be higher; however, after multiple iterations by decreasing the learning rat,e you can prevent the issue of overshooting the targeted minimum.

There are three types of gradient descent: batch, stochastic(SGD) [52], and mini-batch [45]. Batch gradient descent goes over each of the elements in the training set and averages out the values to update the weights. This method is good at finding gradients and quickly and accurately decreases the cost function since it takes all possible weights in the network into account. However, in the case of a large neural network, such as LLM [82], this type would be slow in execution, because of its size. Another method that works better with larger neural networks is SGD. The term SGD is sometimes conflated with the concept of batch gradient descent, mainly because most literature around Machine learning for neural networks. SGD differs from batch gradient descent in that it selects a random sample each time and calculates the gradient for that specific instance. The notation given for SGD is the following:

- $\Delta w = w - \eta \nabla Q_i(w)$, where $\nabla Q_i(w)$ is a randomly chosen example in the training set.

The last type of mini-batch takes an intermediate approach in which you take a fraction of the test set and average over a preselected batch of training data. This has the advantage that the convergence after each iteration is done more directly into the minima, but it is also much faster than a normal batch since only a portion of all data is selected. Mini-batch's mathematical notation is similar to that of batch gradient descent; however, instead of going over all n samples, n represents a select random fraction of the training set.

Neural Networks however are not only used for classification but also generate content, this is dependent on the network that has been selected. It can furthermore be combined with classification, such that the generated result reflects the characteristics of some pre-defined output. In the next view section will discuss these specific neural network for both classification and generation.

2.2.2 Convolutional Neural Networks

CNN [67, 68, 49] is a type of neural network that uses kernel functions and convolution techniques to extract features from a training dataset into the set of inputs that will be fed to a neural network. The most common examples of CNN are in image processing, such as image classification. An example of a convolutional network can be seen in Figure 2.9

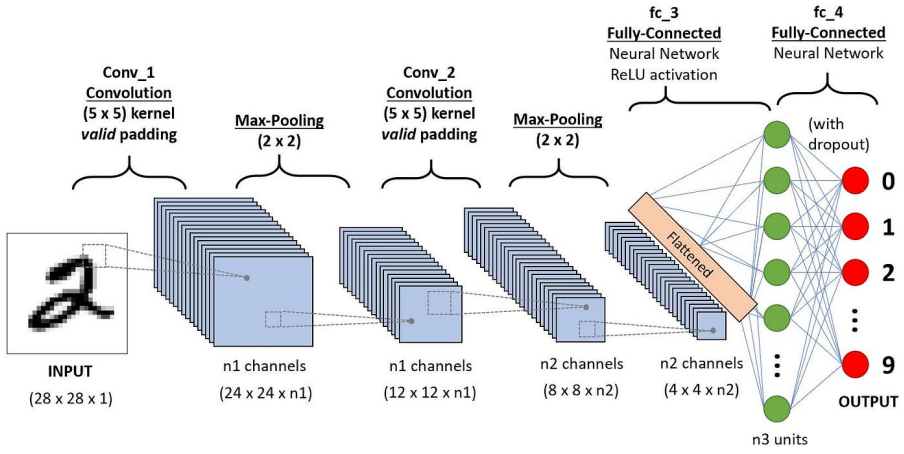


Figure 2.9: Convolutional Neural Network Example. [91]

The stated example takes an image of a written number as input, and based on a fraction of the image, tries to predict the number by first compressing and assigning features to the data through kernels and using the flattened network to train on numbers between zero and nine. The network itself works through the use of convolution, which is a process that adds padding to a matrix of numbers, adds a kernel function, and then executes max pooling on the data sets.

First, there is the defined input, defined as a matrix of numbers. To this input, padding could be added to the edges of the matrix. It is used to provide a way to make the spatial size more malleable to get to some expected output size. However, this is optional and not mandatory if the output size already satisfies the network. The next step which is used in every network is the execution of convolution kernels on the network to both shrink the size of the input matrix and increase the number of inputs. But also to add some function to strengthen certain values and decrease others. Examples of these kernel functions for the CNN example for image processing are in the form of filters, such as grey-scaling, stretching, sharpening, blurring, and many others. Thereafter, are the pooling layers, of which max pooling is the most common, but others can be used like average pooling. This function takes the size of an N X N matrix over the convoluted matrix and searches for the maximal value within a piece of the matrix. The mathematical calculation to find the maximal value is:

$$f_{X,Y}(S) = \max_{a,b=0}^{1} S_{2x+2,a}^{2Y+b}.$$

As you can see in 2.9 these kernel functions and pooling can be executed plenty of time after one another, however, to be useful, at the end of the series of convolutions the network will flatten in a set of outputs, instead of a matrix. The flatting of the network is done so the matrix is represented as an N X 1 matrix. The last step on the part of the Neural Network is that each of the flattened input nodes is fully connected since each of the nodes in the network depends on the context of the nodes in the input.

There are many different applications for CNNs, for topics such as image/video processing, natural language processing(NLP), games and puzzle solving(e.g. chess), and many others. Well-known implementations of CNN are found in the realm of image processing such as AlexNet [61].

The main interest for WAG and CNNs is in the NLP and generation portion of applications. There are already some examples in semantic parsing [63, 32], sentence classifying [55] and prediction [23] can be found. The main interest would be to potentially train a Neural Network for either parsing to WAG, through the use of weights or to use prediction models to train a network to create WAG, from the predefined meta-grammar.

However, CNN is not the only type of Neural Network that is applicable, Recurrent Neural Network(RNN) can also be used. These network are found in generative network, since uses continuous training on the previous iterations, different from CNN which generates based on a train-set. The next chapter will go into this topic with an explanation of how it works.

2.2.3 Recurrent Neural Networks

RNN [40, 94] is a form of architecture that trains a neural network by using both the input and the output of the previous layer to train the next layer or generate the output. Visually, an RNN looks like the model in Figure 2.10. RNNs function by taking input and the previous iterated values and calculating the new values for one iteration.

The main difference between RNN and other neural networks like CNN is that these network can change their forward propagation, by continuously reusing the 'memory' function to embed previous output within the other nodes within the network. One downside of RNN, however, is that for big networks, such as language models, it will discount each long-term memory less and less since the propagated value h gets updated with other values. Of which the latest values have a potentially higher influence than those that were propagated from the start.

One of the iterations of RNN takes the output of the previous one and continuously creates new iterations on the previous executions. These types of networks will potentially go on indefinitely if the input is really big, such as LLMs. One problem that occurs with RNN is that it has an impact on weights during gradient descent, called the vanishing gradient problem. What happens is that the weight update is influenced either in the direction of a slower-decreasing gradient or one that is increasingly uncontrollable. This happens in most cases only when encountering a large network, however, it does still affect the network. To solve this issue, there is a network that solves this problem called Long Short Term Memory(LSTM) [40] (Figure 2.11) or an alternative called Gated Recurrent Unit(GRU) [19].

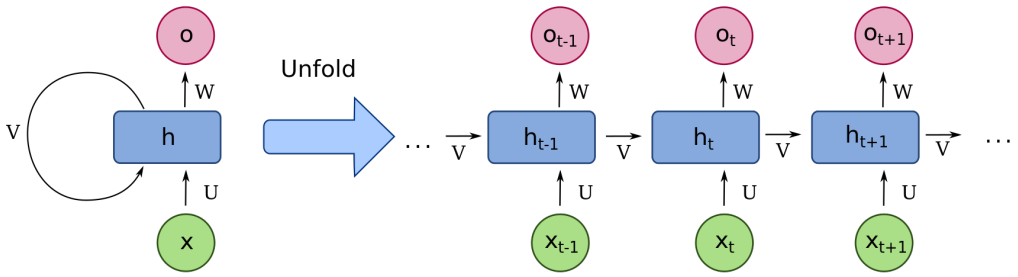


Figure 2.10: Recurrent Neural Network example.

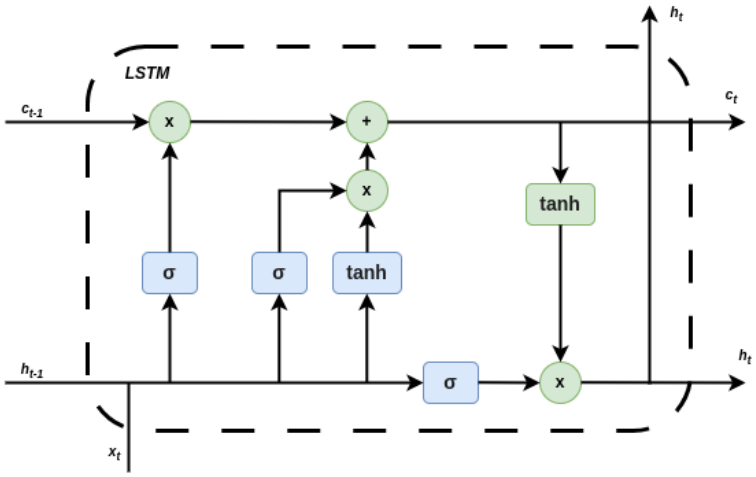


Figure 2.11: Long Short Term Memory(LSTM).

Long Short-Term Memory LSTM solves the problem of gradient vanishing by adding a second parameter called a forget gate, which takes into account whether the network should 'remember' a value and use it in the recurrence. Each LSTM contains a set of values which is dependent on this gate. The variables mentioned are weights W_q and U_q, input i, output o, forget gate f, and memory cell c. The equations are as follows:

- $f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$
- $i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$
- $o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$
- $\tilde{c}_t = tanh(W_c x_t + U_c h_{t-1} + b_c)$
- $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$
- $h_t = o_t \odot tanh(c_t)$

The initial value of c_0 and h_0 are zero and the $\odot$ symbol is the element-wise multiplication (i.e. A $\odot$ B = C). All of the variables depend on time t, where one till t-1 is the previous iterations that occurred before this time slot. The first is the calculation of the forget gate, by taking the sigmoid over the W_f(weights of f) multiplied with the input x at time t, plus the previous forget gate at $t-1$ with weights U, and at last the bias for the forget gate(not time-sensitive). The other three equations are similar, the only difference being the calculated value for input i, output o, and cell input $\tilde{c}_t$. All these four equations are activation functions, with a value between $(0,1)$, and are activated through either the sigmoid function σ or the hyperbolic tangent(tanh). The last two functions are the calculation for both the long-term memory c_t and short-term memory h_t. The first does the matrix multiplication of the forget gate with the long-term memory of the previous calculation, and adds the newly calculated cell activation input with the matrix product of the input activation function. The new short-term memory multiplies the output with the hyperbolic tangent of the long-term memory. The new value of the short-term memory is also the value that gets propagated as the output of the LSTM at t.

As for the range of each of the variables, we have the following listing:

- $x_t \in \mathbb{R}^d$: is the input vector from the previous LSTM layer.
- $(f_t, i_t, o_t) \in (0,1)$: Activation value for the forget/input/output gate.
- $(h_t, \tilde{c}_t) \in (-1,1)$: The long and short-term memory values.
- $c_t = \in \mathbb{R}^h$: both the output and the new short-term input.
- $W \in \mathbb{R}^{h \times d}, W \in \mathbb{U}^{h \times h}, b \in \mathbb{R}^h$: are the weights and biases used to train the activation function needed to calculate both short and long term memory.

The advantage of this construction is that long-term memory helps to prevent the vanishing gradient problem, the network does this through the calculations of the activation values for long-term memory, if a value is important enough to be propagated for the long term the activation functions used to will end up being altogether close to one (f_t, i_t, respectively). While the short-term memory uses the output gate activation function o_t, using the values of the previous states, together with the value of long-term memory, will generate this value. Both the values balance each other out, meaning that the gradient calculation stabilises and doesn't vanish or go in the ascending direction.

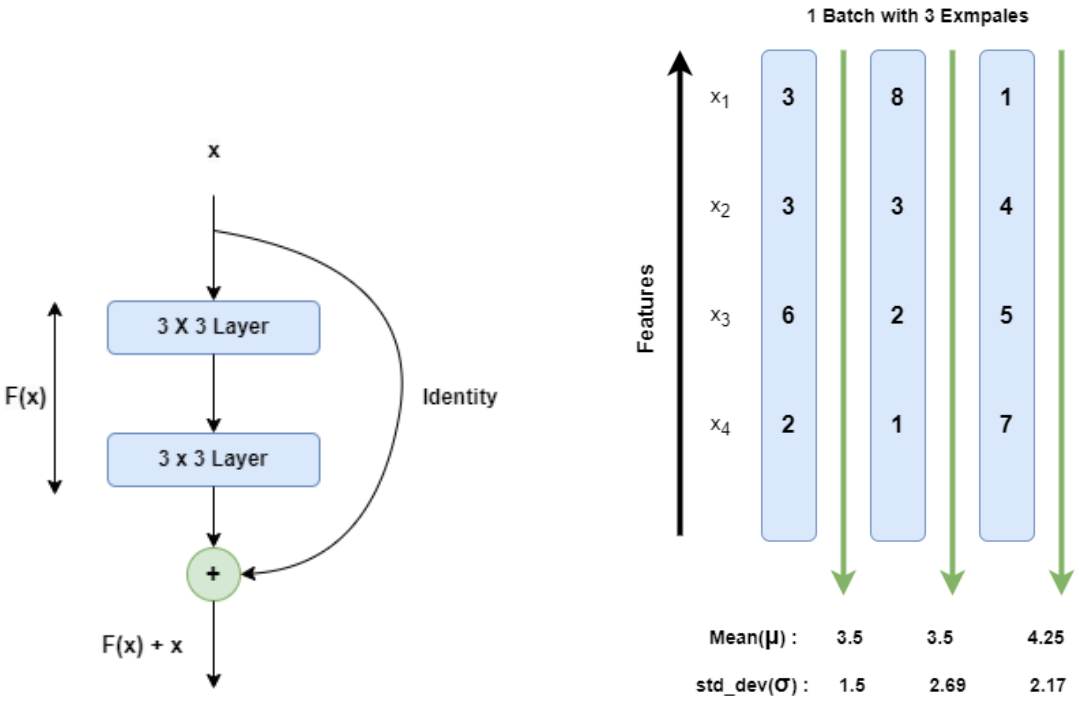
The advantage of RNN in comparison to CNN is that train on the sequential changes which are better suited to textual implementation like grammars. Another example to use this memory component is to use a different, but similar network called Residual Neural Networks or ResNet.

2.2.4 Residual Neural Networks

Residual Neural Networks(ResNet) [107, 71, 100] is a type of neural network that uses skip connection, which uses the value of input or previous set of layers to update the calculated weights. The principle is similar to LSTM, however is different in the sense that the residual step is done on a block of multiple layers and not on each layer continuously. A basic version of ResNet can be seen in 2.12a, which has a connection, called skip-connection, which combines the input of the layers$x$, with the variable after the function performed on the set input $F(x)$.

ResNet comes in many variations, but each variation has the principle that after each block of layers, the input weight is added to the output. Such that $y = F(x) + x$, which is similar to LSTM in the sense that you use the previous output, with short and long-term memory $y_t = F(y_{t-1}) + y_{t-1}$ to prevent accuracy loss. An example of accuracy loss in CNN, is AlexNet [61], which has an issue of decreasing accuracy when adding more layers to the architecture, referred to as the "degradation" problem [37]. This is resolved by a process to normalise the layers (e.g. batch/layer normalization [4]) and backpropagation and gradient descent.

The advantage is that they are less complex and therefore, potentially faster than RNN. As for all the discussed networks, they are mostly used for classification. However, this research is mostly interested in generational techniques. One of the most common generational techniques is the encoder and decoder network, otherwise referred to as auto-encoders which will be discussed in the next section.



(a) Residual Neural Network example

(b) Layer Normalization Example

Figure 2.12: Layer Normalization

2.2.5 Autoencoders Networks

Autoencoders [66, 10, 39] are a type of network that uses both encoding and decoding techniques. Encoding is used to generalise the input to a set of features, which afterwards gets decoded to generate samples based on these generalisations. Such a network can be seen in Figure 2.13

From the figure, you can see that the encoder dimension is reduced to fewer features, and decoding takes these reduced features and generates samples. As for the functional definition of auto-encoders, there are 2 rules:

- There are two sets, an input set used to encode data, let's call it Φ, and an output set to save the decoded data ζ. In most cases, both can be found in the Euclidean space being of dimension m,n such that: $\Phi = \mathbb{R}^m$ and $\zeta = \mathbb{R}^n$
- There are two function families, one encoding function where $E_\phi = \zeta \rightarrow \Phi$ with a parameter ϕ. But also a decoding function where $D_\sigma = \Phi \rightarrow \zeta$ with parameter σ.

With these rules for a feature $x \in \zeta$, there is an encoded variable $z = E_\phi(x)$; this variable has many names, such as latent variable, code, and latent representation. And for $z \in \Phi$ there is also a decoded variable $x' \in \zeta$ that is retrieved by $x' = D_\sigma(z)$, which is commonly referred to as decoded messages. These encoders and decoders are mostly defined in the form of multi-perceptrons, for example, a one-decoder layer, which would be defined as: $D_\sigma(x) = \varepsilon(Wx+b)$. For where ε are activation functions such as sigmoid or ReLu, W is the weight given to the layer, and b is the bias of the layer.

Just like other neural networks, the training is performed by backpropagation. The function to calculate the loss function is done through tasks. A task is defined by using a probability distribution, with attribute $\mu_r ef$ over a set of size ζ. The function called "reconstruction quality"

is defined as $d : \zeta \times \zeta \rightarrow [0,\infty]$, where for variable x there is a $d(x,x'))$, which measure the difference with x'. The loss function itself is defined as the following:

- $L(\sigma,\phi) := E_{x\ \mu_{ref}}[d(x,D_\sigma(E_\phi(x)))]$

The reference distribution μ_{ref} itself is a dataset of $x_1,\ldots,x_n \subset \zeta$, such that $\mu_{ref} = \frac{1}{N}\sum_{i=1}^{N}\delta_{x_i}$. In this case δ_{x_i} is the Dirac measure, which is similar to a step function that only occurs at x_i. Another way to write the loss function is the L2 function, where: $d(x,x') = ||x-x'||^2$, which can be trained by using least squares optimisation, where:

- min $L(\sigma,\phi)$, where $L(\sigma,\phi) = \frac{1}{N}\sum_{i=1}^{N}||x_i - D_\theta(E_\sigma(x))||^2$

There are a few different types of auto-encoders, such as sparse auto-encoders(SAE) [101], denoising auto-encoders(DAE) [66], and variational auto-encoders (VAE) [56]. If there is a high degree of noise, the DAE will find a solution, but even after execution, the general classification becomes harder. These types of noise are sometimes used in so-called adversarial attacks [76]. These are attacks against neural networks to make classification harder on purpose. Defending against these types of attacks will only improve the ability of the neural network to classify.

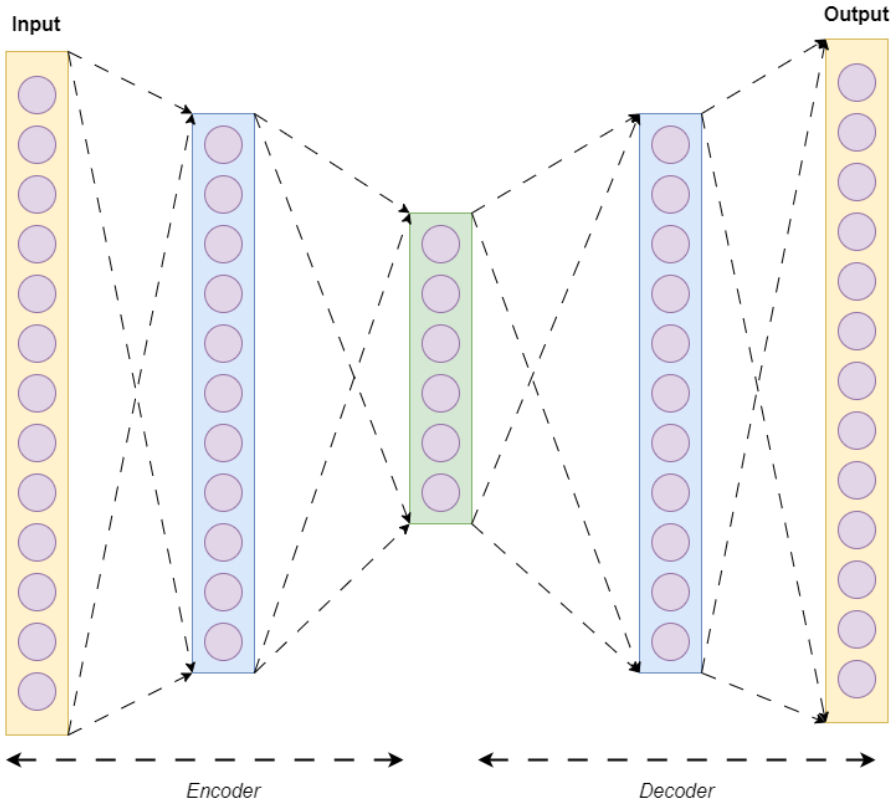


Figure 2.13: Auto-Encoder Architecture

Concluding Neural Networks and Background. To conclude the section on Neural Networks, several interesting techniques aid in both classification and generation. Some of the researched techniques make use of these techniques and provide some sense of the technical formulation. In particular, the Transformer Networks in the form of GPT use an LSTM, an encoder/decoder network in both pre-training and fine-tuning. This will be continued in the Research chapter.

As for the conclusion of the background, WAG is an interesting grammar to apply to the use case of grammarware. The question arises, how can it be applied to generate grammar samples, and what meta-grammar are we using the generate these samples? These questions will be formulated in the next chapter, which describes each of the research questions and which should be answered in the subsequent chapters.

Chapter 3

Research Questions

In the research on how generative techniques can develop new grammar from the meta-grammar, it is necessary to state the main question that has to be answered. This is done by specifying what is being researched and what results are expected when the main research question is answered. The main research is:

- Which generational techniques can be implemented to generate grammar samples from a meta-grammar, using WAG as the output grammar?

The context of the question relates to the introduction, where the addition of new grammars can provide samples to test grammarware-based applications. The question itself will be answered in two sections: research and results. The former will answer the questions about which techniques and meta-grammar will be used, and why these have been selected. In the Results chapter, these generative techniques will be implemented, and each of the characteristics of the different techniques will be compared. Furthermore, a comparison is made between different meta-grammars used as input for generation. In the introduction, the four different techniques of Grammarinator, transformers, XSmith, and evolutionary grammars(EG) have already been provided. The research section will go on and describe how each of them works and why these are potentially interesting to generate grammar samples.

The answer to the main research questions is that you can generate results with these techniques. However, what about the quality of the generated results? Are the generated results reflective of the meta-grammar, and are the generated results applicable to grammarware applications? To answer these follow-up questions, sub-questions were developed, which are to be answered at the end of the research. These research questions are:

- What are the results when the generated samples are parsed, and how does each of these techniques compare?
- Does each meta-grammar generate differing results, and what constitutes that difference?
- Are the results of generation applicable to grammarware, and if not, what can be done about it?

The first question is quite straightforward: can we parse the results that have been generated? This can be tested by trying to generate an AST based on the meta-grammar and comparing it to the generated results. One requirement is that each of the generated results should be predictable. Since it is the case that some generative networks create parses that are inspired by the grammar but are not restricted by it. This means that the resulting samples are quite unpredictable, either generating samples that are completely unparsable. The result will also count the partial parses

of the grammar and will compare the average percentage of correct parses. The interest in this percentage is that there are cases where the grammar is not completely parsable; only partially. In such a case, it is advantageous to know to what degree that is the case, and which technique has a higher percentage chance of parsing.

The second question depends on the chosen grammar in the Research section, but is asked since grammar can be expressed in many ways. In some cases, such as CNF, the grammar is more restrictive than in CFG, and if one takes a meta-grammar of both and generates a new grammar, the newly generated CNF would probably also be a more restrictive expression than CFG. And potentially the opposite with WAG, since it is more expressive, containing both the weighted and attribute components. To test this hypothesis, different meta-grammars will be used, and a comparison made between their expression and generated results.

The last question is to conclude the generated samples by assessing whether the results might be applicable as a grammar. There is the possibility that, though new grammar is generated successfully from the meta-grammar, the results are insufficiently applicable to both cases, or the generated grammar rules conflict with the principles of a language. In that case will go over the results and describe its flaws and which generator(s) show the most potential based on the needed characteristics.

Chapter 4

Research

This section is about the research performed before the implementation of each of the different generational techniques, and comparing them in the results. The section starts by defining the meta-grammars that will be used to generate new grammar samples. Then, a description of each of the four different generational techniques will be given, starting with Grammarinator. Thereafter, transformer networks will be discussed, which use the neural network techniques from the Background chapter. After that will go into the framework called XSmith and provide context on the purpose, origin, and use. Lastly, the topic of Genetic Algorithms(GA) will be given. This section will contain information on crossover/mutation, and the specific subset of both Genetic Programming(GP) and Evolutionary Grammars(EG) will be discussed.

4.1 Input for grammar generation.

To be able to generate new grammar, we need to define the meta-grammar, which is used to generate each of these results. As mentioned in the introduction, WAG syntax will need to be generated, which means that each of these meta-grammars should also reflect both aspects of attributes and weighted grammar. This research came up with two different grammars that were tested. The first is a subset of English grammar, containing basic grammar rules like: sentences, noun-phrase and verb phrases. The second one is based upon the meta-grammar for the WAGON thesis of Rafael Dulfer [27].

The English Grammar has been written to generate simple parses of the English grammar, which contain some basic procedures. A portion of this Grammar containing most procedures can be seen in Listing 4.1. It can be seen that the grammar has few rules for generation, but still contains the most basic assignment of the WAG grammar, such as weight and attributes. The weight variables will either be a static variable or have a reference to a numerical fraction. As for the attributes, these have been predefined and refer to natural language concepts, such as active verbs, tenses, and acronym definitions. All of the concepts are implemented as terminal values to restrict the expression of the grammar, such that the generated samples should have predictable outcomes. This is to say that the sentence“I am going home”, could be one of the potential parses where $S \rightarrow \ [\$p]\ Pronoun\ \{\$active = true\} Verb\ Noun$, where $\$p$ is an attribute reflecting probability.

```
grammar SmallEnglishGrammar;

//Root element
small_english_grammar : rule*;

// Production Rules
rule                : nTAssigment PROPESITION_SIGN rhs
    STATEMENT_END;
rhs                 : weight? chunk*;
weight              : BRACKET_L weightVariable BRACKET_R;
weightVariable      : FRACTION | attrIdentifier;
chunk               : nTAssigment | TERMINAL;

attrIdentifier      : ATTRSPEC ATTR_IDENTIFIER;
attrIdentifierList  : attrIdentifier DELIMITER attrIdentifierList
                    |  attrIdentifier;

nTAssigment         : NON_TERMINALS attributeAssignment?;
attributeAssignment: ATTR_ASSIGN_L (INVERSE_SIGN? attrIdentifier EQ
     atom STATEMENT_END?)* ATTR_ASSIGN_R;
atom                : attrIdentifier | BOOL | FLOAT | INT | STRING |
     FRACTION;

// Lexer
// English Terminals
NON_TERMINALS       : 'S' | 'NP' | 'VP' | 'PP' | 'DET' | 'Verb' | '
    Noun' | 'Nominal' |
                      'ProperNoun' | 'Pronoun' | 'Preposition' ;
ATTR_IDENTIFIER     : 'tense' | 'active' | 'acronym' | 'literal' | '
    proper';

TERMINAL: NOUN | PROPER_NOUN | VERB | PRONOUN | DET | PREPOSITION;
NOUN                : 'door' | 'house' | 'apple';
PROPER_NOUN         : 'Tom' | 'Fred' | 'Enschede';
VERB                : 'visiting' | 'going' | 'eating' | 'visited' |
    'went' | 'ate';
PRONOUN             : 'I' | 'you' | 'him' | 'they' ;
DET                 : 'the' | 'a' | 'this' | 'there' | 'his' | '
    their';
PREPOSITION         : 'to' | 'with' | 'for' | 'at';

// Grammar Variables
PROPESITION_SIGN    : '->';
ATTRSPEC            : '$' | '*';

BOOL                : 'true' | 'false';
INT                 : NUM+;
NUM                 : [0-9];
FLOAT               : [0-9]*'.'[0-9]*;
FRACTION            : '0.'[0-9]*;
STRING              : '"' ~('\r' | '\n' | '"')* '"'
                    | '\'' ~('\r' | '\n' | '"')* '\'';

STATEMENT_END       : ';';
EQ                  : '=';
```

```
49 INVERSE_SIGN        : '!';
50 BRACKET_L           : '[';
51 BRACKET_R           : ']';
52 ATTR_ASSIGN_L       : '{';
53 ATTR_ASSIGN_R       : '}';
54 LPAREN              : '(';
55 RPAREN              : ')';
56 DELIMITER           : ',';
```

Listing 4.1: English Meta Grammar.

The WAGON generation takes the opposite approach by generating non-terminal, terminal, and attributes in grammars based on randomly generated strings. This means that the names of each of these are randomised and depend on the generational techniques that are generated by them. The specifics can be found in the cited document. For this particular research, some tweaking to the grammar has been done to make it possible to be used in ANTLR. The reason is that when using ANTLR, the parser had certain grammar rules that came first, which resulted in parses which are grammatically correct but were not reflective of the meaning of the grammar. As an example to test whether the grammar works, I used the Pokémon WAG example from WAGON.

Each of the grammars, however, is modelling behaviours from some concept, where the small English grammar reflects on the linguistic aspect of the English language, and WAGON, which reflects on potential new grammars generated by the WAGON definitions. To assess whether the results are contextual to each of these concepts, it is necessary to mention semantics, which will is discussed in the next paragraph.

Semantics Some definitions are fundamental to be able to explain the results that will be generated by the different techniques. This section is about the definition of the words 'meaning' and 'semantics'. The reason for providing these specified definitions is that both have a deep linguistic and philosophical underpinning. Furthermore, the word's meaning has different meanings/definitions based on the context it is in. The first step was to look at both dictionary definitions of meaning and semantics. This was done through Encyclopedia Britannica and was taken as the source. Meaning [77], is said to be the sense of linguistic expression that can be understood in contrast to its referent. Examples are referents but no contrast (they/that), and contrast without reference (the current prime minister of the Netherlands). In the case of generating new grammar, this would imply that the generated results are made with the meaning of the grammar by being referent, with the domain of the grammar as the context. Another word for giving meaning to language is the scientific study of Semantics [78]. The most foundational paper on the meaning of semantics was written by Gottlob Frege, named "Über Sinn und Bedeutung" [83], translated into English as Sense and Reference. Where the Reference is the proper naming given to an object, sense is the expression of the reference in language. An example of this is made between two Greek Gods, Hesperus and Phosphorus, which both can refer to the planet Venus, where the following statement is true: "Hesperus is the same planet as Phosphorus". In this case, the referent is the same for both, which means that the sense is specified for one mode of representation. This is to say that only one aspect of both Gods is taken, which is their relation to the planet Venus. This is only one representation of the word sense. Another representation of sense is giving inherent meaning to words, even when not referring to a specific object. Frege's example is the use of the name Odysseus, though the Greek hero is not an object, his name still has sense, i.e. as the main character in the Odyssey. As for reference, there is some debate on the specific translation. In the Germanic language family, the words Bedeutung(German) and betekenis(Dutch) are translated as meaning, purpose, or significance. However, the translation in English uses the word reference. The proper word is referent, which is the most accurate translation of Frege's work [8]. From this

point we can also go into different definitions of sense from both Bertrand Russell [88], or others such as John McDowell [70], however, this would go into hard into the philosophical discussion, about the definition of sense and language.

The way it is used in this thesis is that each of the non-terminals, terminals and attributes is a representation of the semantic meaning of a grammar. Where the references are concepts within the grammar, like "Noun" in the English language, representing the building block of sentences. Where the sense of a noun is the type of values the noun refers to, i.e. "House" or "bed". For a grammar to be semantically correct, not only should it be able to generate references which reflect the grammar, but each of the values and their types should be correct as well.

4.2 Grammarinator

The first application of generating grammar was to use a tool called Grammarinator [41]. Grammarinator is a fuzzing tool to generate samples for testing the implementation of grammar, based on an ANTLR definition. Grammarinator and XSmith are both used to generate samples by using fuzzing techniques. These types of techniques are used to generate samples to test code for its robustness. The upside of using fuzzing techniques is that each generated sample conforms to the grammar specification. However, with some fuzzers, the generated result does not reuse the identifiers within the grammar and replaces them with randomised characters. An example of these fuzzer types is similarly mentioned in XSmith section 4.4 called AFL. The main purpose of these types of fuzzer tools is to find bugs or vulnerabilities in code, not to generate readable grammar. As for the structure of the generator, there is a block diagram, which shows each of the components needed to generate results, see Appendix A.3.

To be able to generate the sample with Grammarinator, it uses the defined grammar from the previous section, where it is then separated into two parts: a parser containing all the grammar procedures from NT to other NT, and a lexer containing all NT to terminal procedures. Both these files will be added together to generate a generation file. The file itself is a Python file used by Grammarinator and contains each of the different grammar rules and variables. An example of a section of the generation file can be seen in Listing 4.2.

The generation of these new files is done, by taking the code from Grammarinator's own GitHub repository. Thereafter, in a terminal of choice, two commands will need to be entered to create the grammar generator file, from the lexer and parser. The second is using this generated file to generate new samples. These two commands that were used in the case of generating the WAG samples can be seen in Listing 4.3

For the command to generate the samples using the Python generator file, some parameters can be taken. The first of which is the root element, i.e. non-terminal in the grammar from which the rest of the grammar will be generated. In the case of the WAG grammar, this is "wag" non-terminal. This is done by adding the non-terminal after the `-r <start-rule>` terminal argument. After providing the starting element, the user can provide how far Grammarinator can go through the generated AST to generate new elements, which is given by setting `-d <max-depth>`. The other variables are mainly what, how many, and where to store the generated results. For the output format/directory you can provide `-o <output-directory>`, the amount of samples `-n <number-samples>` and the path to run the program `-sys-path <path>`

```
import itertools

from math import inf
from grammarinator.runtime import *

class SmallEnglishGenerator(Generator):


    def EOF(self, parent=None):
        pass
    EOF.min_depth = 0

    def NON_TERMINALS(self, parent=None):
        with RuleContext(self, UnlexerRule(name='NON_TERMINALS',
            parent=parent)) as current:
            with AlternationContext(self, [0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) as
                weights0:
                choice0 = self._model.choice(current, 0, weights0)
                UnlexerRule(src=['S', 'NP', 'VP', 'PP', 'DET', '
                    Verb', 'Noun', 'Nominal', 'ProperNoun', 'Pronoun
                    ', 'Preposition'][choice0], parent=current)
            return current
    NON_TERMINALS.min_depth = 0

    def ATTR_IDENTIFIER(self, parent=None):
        with RuleContext(self, UnlexerRule(name='ATTR_IDENTIFIER',
            parent=parent)) as current:
            with AlternationContext(self, [0, 0, 0, 0, 0], [1, 1,
                1, 1, 1]) as weights0:
                choice0 = self._model.choice(current, 0, weights0)
                UnlexerRule(src=['tense', 'active', 'acronym', '
                    literal', 'proper'][choice0], parent=current)
            return current
    ATTR_IDENTIFIER.min_depth = 0

    def TERMINAL(self, parent=None):
        with RuleContext(self, UnlexerRule(name='TERMINAL', parent=
            parent)) as current:
            with AlternationContext(self, [1, 1, 1, 1, 1, 1], [1,
                1, 1, 1, 1, 1]) as weights0:
                choice0 = self._model.choice(current, 0, weights0)
                [self.NOUN, self.PROPER_NOUN, self.VERB, self.
                    PRONOUN, self.DET, self.PREPOSITION][choice0](
                    parent=current)
            return current
    TERMINAL.min_depth = 1
```

Listing 4.2: Grammarinator generated class of WAG grammar.

```
/bin/bash

grammarinator-process WagLexer.g4 WagParser.g4 -o codegen/ --no-
    actions
grammarinator-generate WagGenerator.WagGenerator -r wag -d 10 -o
    samples/sample_%d.wag -n 100 --sys-path codegen
```

Listing 4.3: Commands to generate WAG samples

4.3 Transformer Networks

Machine Learning and, particularly, neural networks are commonly used in different fields of development, ranging from classifiers to decision-making and learning. One of these topics is generation, which ranges from generating text [66] to images [60], or even videos [108]. One of the most well-known iterations of these generation networks is ChatGPT [80], which uses prompts to generate output to the question from the user. ChatGPT is based on Generative pre-trained transformers(GPT) [12], which is based on a technique called Transformer Networks [104, 26, 35] to generate these results. The question is can we use the transform networks, and add a WAG example as input, to create a similar but different grammar?

The transformer networks generate results based on an encoder/decoder network, in combination with an attention network. The main difference between transformers and other neural networks is the use of attention layers. The definition of attention is best described in the initial paper on transformers("Attention is all you need") [104]

> An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

The advantage of using transformer networks is that computationally can be equal if not better, than both recurrent and convolutional networks. An example of the complexity is given in the same paper.

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Figure 4.1: Complexity comparison between RNN/CNN/Transformer Networks.

When performing sequential operations, self-attention layers are instant operations similar to convolutional networks, while recurrent networks have a dependency on the length of the sequence. As for the maximum path, the self-attention layer is instant while both recurrent and

convolutional are dependent on the size of the sequence; an exception is the restricted version, which is similarly dependent on the sequence length. As for the complexity per layer, it is less complex when the size of the sequence is smaller, while the size of the representation layer is larger. The reason why transformer networks might be applicable for the generation of WAG code is based on the properties of the networks to generate results based on previous examples.

The upside of this approach is that the generation of the network will look at input samples and try to generate a similar output. This potentially implies that the result would be closer to the real application of the grammar, that is to say, reflect on the use of grammar, i.e. semantics, instead of grammatical correctness. This might be a potential downside since parsing the grammar would lead to potential errors or misinterpretations.

The architecture for the transformer network with attention layer consists of the mentioned technique, the background: encoding/decoding networks used in the form of unsupervised learning. An overview of the overall architecture of these types of networks can be seen in Figure 4.2

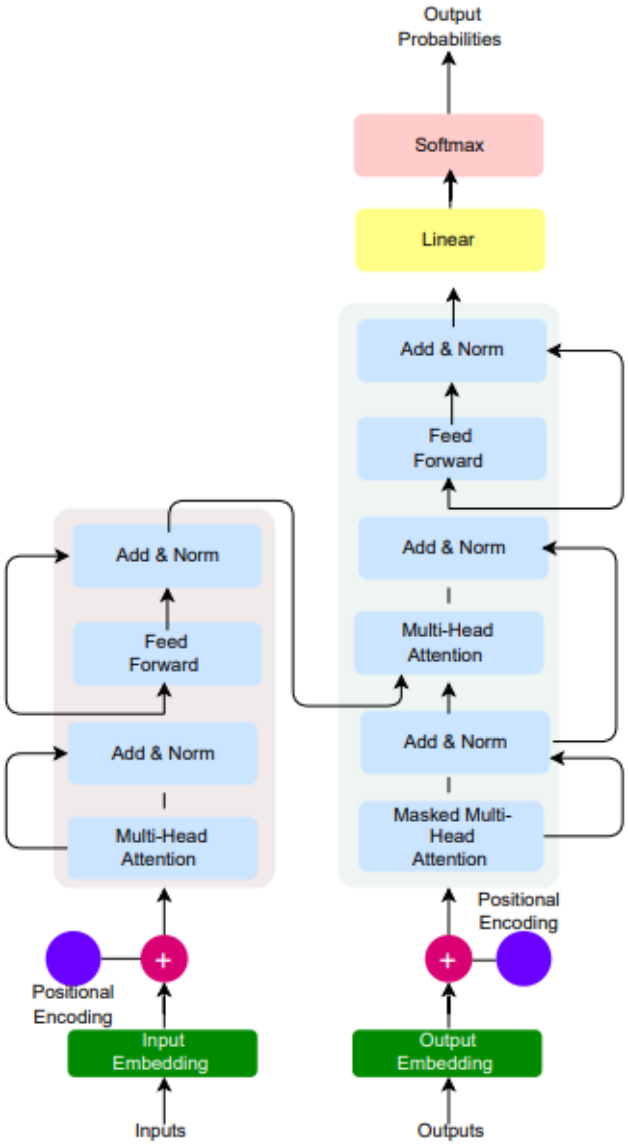


Figure 4.2: Transformer Networks Architecture of GPT.[110]

It might be mitigated by filtering out results using the output and putting it through a grammar tooling, i.e. ANTLR. If it doesn't create a partial AST Tree based on the predefined definition of WAG, it would reject the output sample, and the result would be more semantically correct samples, and would also be grammatically correct for the most part.

GPT can potentially take up the role of grammar generation results based on different corpora. The cited paper [2] is the first release of what would become GPT and mentions the two different stages the framework goes through. These are unsupervised pre-training and supervised fine-tuning, and were defined as such:

- Unsupervised Pre-training: The layer is specifically used to explore a given dataset to find minima, based upon the distribution of data [29]. There are different versions of unsupervised pre-training, such as the use of LSTM to perform the pre-training [43], however, this does not apply to GPT, since, as the name suggests uses transformer networks to perform the pre-training. The network is given a corpus of tokens $\mathbb{U} = \{u_1, \ldots, u_n\}$. Furthermore, in this stage, the log-likelihood will be calculated, which is needed in the supervised fine-tuning, by the following equation:
  - $L_1(u) = \sum_i \log P(u_i | w_{u-k}, \ldots, w_{u-1}; \Theta)$, where k is the size of the context window, P the conditional probability and Θ are the parameters of a neural network, and all is trained using a stochastic NN.
  - $h_0 = UW_e + W_p$
  - $h_l = transformerBlock(h_{l-1}) \forall i \in [1, n]$
  - $P(w) = softmax(h_n W_e^T)$
  - Where $U = (u_{-k}, \ldots, u_{-1})$ contains a vector of tokens, and n is the number of layers in the NN.
- Supervised fine-tuning: After the model has been trained in the previous step, we can use the generated parameters to target a specific task. Assume there is a given dataset which is labelled (C). Each instance in the set contains a sequence of input tokens $(x^1, \ldots, x^m)$ and an output variable y. The inputs x are put into the pre-trained model, with a last transformer block action variable h_l^m, combined with a linear output layer with a parameter W_y to predict the output as: $P(y|x^1, \ldots, x^m) = softmax(h_l^m W_y)$. To maximise the result it combines the log-likelihood of x and y: $L_2(\mathbb{C}) = \sum_{(x,y)} \log P(y|x^1, \ldots, x^m)$

After both steps, you can combine both results into one, by the following equation: $L_3(\mathbb{C}) = L_2(\mathbb{C}) + \lambda * L_1(\mathbb{C})$, where λ is the learning rate given to the pre-trainer.

The most popular/mainstream implementation is ChatGPT [80], which is a chat wrapper around the GPT LLM, which latest version runs on GPT-4 with different versions such as (GPT-4o, GPT-4 mini, and o1 preview and 01 mini). However, these models are behind a paywall and not publicly accessible. A model that is, however, is GPT-2, through either GitHub or the HuggingFace website [86], specifically the Open-AI GPT library.

4.4 XSmith

The next iteration in the generation of WAG will be XSmith, a tool written by the University of Utah, Flux Research Group. The tool itself is based on a different project from the same university called CSmith, which is a generator suite used to create C generate programs [109]. The main use of generating C code was to try and find bugs in the GCC compiler, using a fuzzing algorithm. The idea behind fuzzing is to generate a large number of samples and use them as inputs for

compilers, programs, API, or any piece of software that takes inputs. To test if all functionalities of a program work, it has to have a known output to verify the results. If there are failures within the program, there will be a case that gives an output that is not expected. If this is the case, a bug has been found. An example of such a bug can be seen in Figure 4.3. In this particular case, a loop optimisation caused a wrong assignment, where the answer should have been eleven. Since &y (reference) is given to *p (pointer), and then the value to which p points is set to x, where x = 4. The last assignment is to add 7 in the next iteration of the for loop, so 7 + 4 = 11. This will go through once and afterwards leaves since $11 \not< 8$. Instead, the link through p was skipped by GCC, so y remained 1, and the next cycle 7 was added, and the loop was exited the next cycle, leading to 8.

GCC Bug #4: wrong analysis[7] A version of GCC miscompiled this function:

```
1 int x = 4;
2 int y;
3
4 void foo (void) {
5   for (y = 1; y < 8; y += 7) {
6     int *p = &y;
7     *p = x;
8   }
9 }
```

Figure 4.3: Example of the use of CSmith to use fuzzing to find bugs. [109]

In the paper, there are plenty of examples of generated fuzzing examples that are useful to spot potential bugs in code or vulnerabilities in the code base. CSmith is not the only tool that generates fuzzing results, there are also other examples, such as American Fuzzy Lop(AFL) [65], which is a C++ fuzzer using genetic algorithms to generate fuzzing results to find vulnerabilities in a program. This, in turn, can lead to security breaches. The question is why and how these fuzzers can be applied to the generation of WAG. The answer is that fuzzers work by generating large samples of inputs based on a specific language specification. These samples are used in several ways of testing, unit/integration/performance, to see whether the specified language/framework is sturdy enough to handle all forms of inputs and provide expected outputs. This would make the fuzzer useful for generating specific samples for languages, but could also be implemented on a grammar specification. The use of CSmith does not apply to WAG, since it is written specifically for the C implementations, and wouldn't apply to WAG's grammar.

This is where XSmith comes into the picture, with its main difference being that this fuzzer is not written for a specific language, but generates results based on the language specification itself. XSmith itself is a DSL that is written in the Racket language [30]. Racket is a language initially developed to aid in teaching Computer Science at several American universities in the Scheme Language, which is part of the LISP [99] family. The purpose of Racket is to develop other programming languages by using its language specification and being able to do so quickly. To show how XSmith works, we take the example from one of the cited papers [36] and quickly go over it. In this example, a simple calculator grammar will be written in which you either have expressions or values. The calculator itself only has two possible operations: addition and division. Also, the values of each of the integers must be between 0 and 99. An example of the grammar can be seen in Figure 4.4. Each of these functions can be translated into Racket and will give the code in Figure 4.5.

From both of the examples, it can be seen that the structure of both grammar and code is quite similar; this does not mean that this is the case for all possible grammars within the language. It can be because of the level of complexity within the grammar, or the size of the grammar/language

$$
\begin{aligned}
\langle int \rangle &::= z \mid 0 < z < 100 \\
\langle exp \rangle &::= \langle exp \rangle \quad + \quad \langle exp \rangle \\
&\mid \quad \langle exp \rangle \quad / \quad \langle exp \rangle \\
&\mid \quad \langle int \rangle
\end{aligned}
$$

Figure 4.4: Simple grammar calculation example using Racket DSL. [36]

```
;; Fuzzers may use multiple add-to-grammar forms to
;; declare a grammar in a modular fashion.
(add-to-grammar calc
 [Exp #f ()
      #:prop may-be-generated #f]
 [Add Exp ([lhs : Exp]
           [rhs : Exp])]
 [Div Exp ([lhs : Exp]
           [rhs : Exp])]
 [Int Exp ([val = (random 1 100)])])
```

Figure 4.5: XSmith code example of the calculator. [36]

is too big to potentially model all aspects. The generation itself works by using elements called 'holes', in which any expression can be placed. The idea is that you have a tree with holes, which can be filled by a pre-defined set of expressions, each can have other potential sub-expressions, which are also holes that can be filled. An example of how this works can be seen in the abstract under Figure A.5.1. From the example in Figure 4.5 it can be deduced that the generation of a nearly infinite number of differences since the method can recursively call the ADD or DIV expression in each of the left or right-hand side of the tree. However, a potential issue might be that the grammar has different amounts of occurrence of certain expressions, and you might want similar results to WAG by adding weight to profess importance. XSmith has implemented this feature by being able to add an integer value to the specific expression and set the weight; the higher the weight value, the higher the likelihood the specific expression will be part of one of the resulting outputs. An example of the notation with the calculator would be:

```
1 [Div 20]
2 [Int 3]
```

Since XSmith is agnostic, in that it doesn't conform to a pre-defined language, it performs some basic validation on the definition. However, to validate the language itself, XSmith can use systems under test(SUT), which are generated to prevent errors. In the case of both syntactical and semantic cases, SUT will reject the generation when SUT detects errors and often provides related error messaging.

Otherwise, XSmith also takes both language and syntax into account when trying to generate the results. The generation of these languages themselves depends on an attribute grammar called RACR [14]. Attribute grammar is used to keep track of certain variables through the AST that are made through each of the generations and by the generation of each of the AST trees. The use of these grammars is crucial to the syntactical correctness of the generated result since it needs to convert the AST into a proper compilable/interpretable definition. Besides the grammar to improve the speed of generation, XSmith also makes use of different types. The type declaration supports inheritance, and you can also define whether the child element has inherited the parent type. An example of the declaration of these types can be seen in Figure 4.6.

To compare how well XSmith performs to other similar fussers, we have an example of the number of lines needed to be able to generate samples for a specific coding program; see Figure 4.7 for reference. From it, it can be deduced that most of the grammar needed to generate samples needs a much larger code base than XSmith itself. This should also simplify the WAG implementation since the potential amount of LOC to write will be smaller than other fuzzer variations.

```
(define no-child-types (lambda (n t) (hash)))
;; The `hash` function constructs a hash table
(add-property
 js-component
 type-info
 [LiteralString [string no-child-types]]
 [StringAppend [string (lambda (n t) (hash 'l t 'r t))]]
 [VariableReference [(fresh-type-variable)
                     no-child-types]])
```

Figure 4.6: Example of the use of CSmith to use fuzzing to find bugs. [36]

Generator	LOC	Language
Csmith	38,988	C++
Verismith	10,139	Haskell
SQLsmith	3,909	C++
Xsmith Racket Fuzzer	1,265	Racket
Xsmith Dafny Fuzzer	1,666	Racket
Xsmith Standard ML Fuzzer	1,151	Racket
Xsmith WebAssembly Fuzzer	1,433	Racket
Xsmith Python Fuzzer *	1,800	Racket
Xsmith Lua Fuzzer *	450	Racket
Xsmith Javascript Fuzzer *	412	Racket

Figure 4.7: XSmith code comparison other fuzzers. [36]

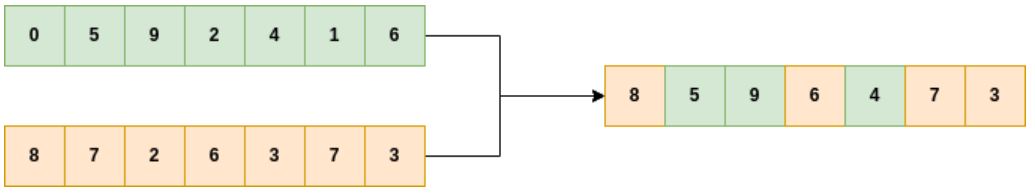


Figure 4.8: Visual example of GA crossover

4.5 Genetic Algorithms

The last variant to research is Genetic Algorithms(GA) [42, 72, 53], of which there are many different combinations and variations. The principle of GA is based on evolutionary theory and took inspiration from Charles Darwin's book: "On the Origins of Species" [25]. The idea is that you have an initial population of data that will be assessed on its fitness. Fitness is a set of competitive criteria which will test parts of a population by competing for the best-fitting resolution to a specific case. Each of these parts of the population is called a chromosome, which is represented as a set of variables. Each of these individual variables is called a gene. New chromosomes are created by using a technique called crossover to create a new chromosome. Crossover works by taking two chromosomes and creating a new one by randomly choosing genes alternating between the genes of each of the chromosomes. An example of this can be seen in the Figure 4.8. Besides crossing genes, in the process of evolution, there is also the phenomenon of gene mutation. In the case of genetic algorithms, it means that genes in a chromosome will have their value altered to make a more unique chromosome. A similar sketch to crossover has been sketched, see Figure 4.9.

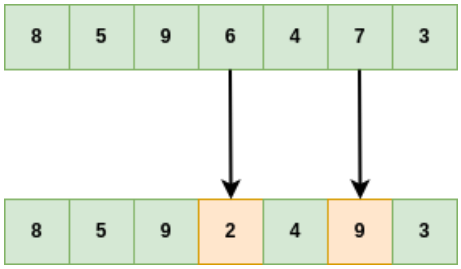


Figure 4.9: Visual example of GA mutation

Combining both techniques on a population will generate a new population, on which testing is performed to check the fitness variable. The goal is to find the best-fitting result, and the algorithm will continue until it converges. In most cases, this means retrieving some optimal population after

the maximum number of performed iterations. When going through the whole procedure of GA in pseudo-code, it will get the following result:

```
1   BEGIN
2
3   Provide an initial population of chromosomes,
4   i.e. populaiton = { [1, 2, ...], [...], etc. }
5   fitness_variable = fitness(population)
6   WHILE (!population_converged)
7       crossover_population = crossover(population)
8       mutated_population = mutation(crossover_population)
9       fitness_variable = fitness(mutated_population)
10      population = mutated_population
11      population_converged = calculate_conversion(fitness_variable)
12
13  END
```

There is a part missing from this pseudo-code that is needed as a precondition to be able to execute genetic algorithms, which is encoding each of the genes, and the selection of the chromosome to create a new iteration through crossover and mutation. The encoding is used to format data such that it has an input to perform both tasks. After the data has been encoded, a selection is made, using different tactics to gather the sets of chromosomes for crossover and mutation. Within one of the papers on GA [53], there is a diagram that gives a list of the more common variants of each of the encodings, selections, crossovers, and mutations.

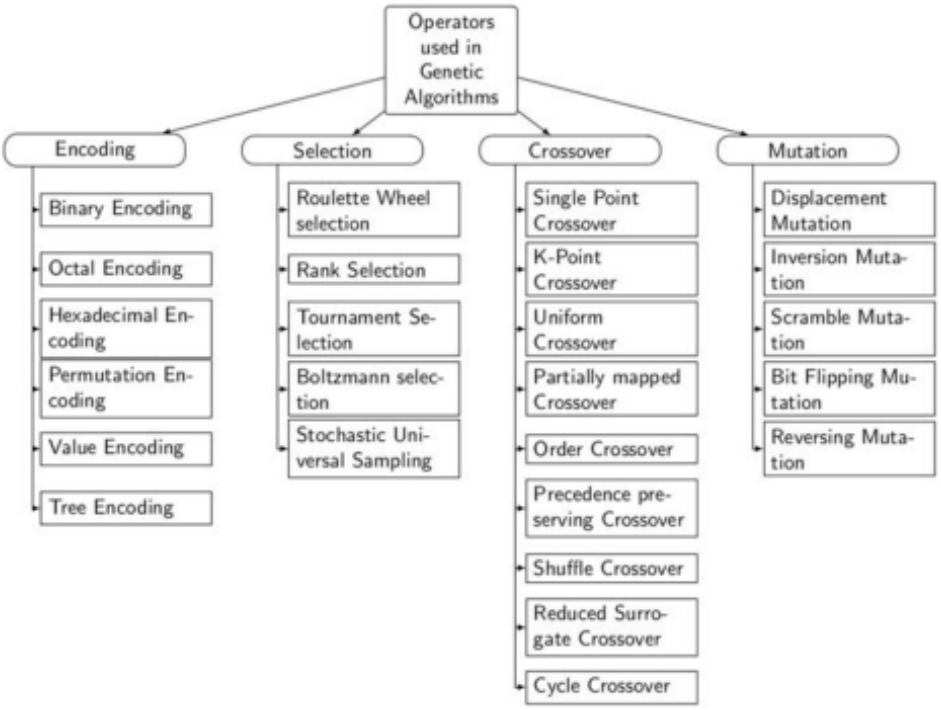


Figure 4.10: Genetic Algorithm overview [53].

For encoding, it depends on what each of the different genes should refer to. In the case of binary encoding, it might be a binary choice (true/false), or a numerical value which can be in octal(0-7) or hexadecimal(0-F). Besides these numerical variants, there are three other types of

Chromsome	Fitness
A	16
B	6
C	3
D	2
E	8
F	1

Figure 4.11: Fitness variables

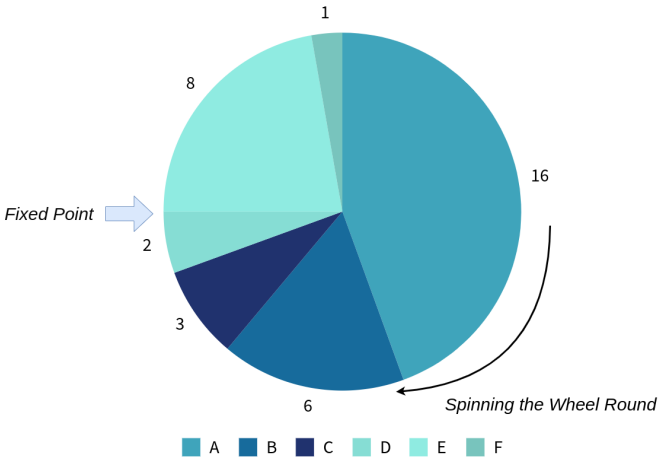


Figure 4.12: Roulette Wheel Selection Example

encoding: permutation, value, and tree encoding. The permutation variant is represented by a string of numbers that represent the positions of the sequence in a gene or chromosome. In the case of value encoding, each of the different variables can be of a range of types, which includes real / integer numbers or characters. The main use case of this type of encoding scheme is in neural networks, since it has a more varied range of possible values than the different numerical encodings. The last potential encoding is tree encoding, which uses a tree structure to encode the chromosomes, where each gene is a node in the tree. This technique is also common in LISP, and the structure looks nearly identical to XSmith hole filling of trees, Figure A.7. Where each node can contain similar values to those of value encoding. The downside of these trees is that not all solutions of genetic algorithms can be expressed in a tree structure.

These encodings will then be put through a selection process, by taking the parent chromosome and which "mates" and create offspring through crossover and mutation. To perform selection, we need to have a defined set of fitness variables for each of the chromosomes in the population. For the explanation will use the table in Figure 4.11.

There are several ways to select the chromosomes, one of which will take a look at the Roulette Wheel first. To better illustrate the working of the Roulette Wheel selection, an example sketch of a pie chart in Figure 4.12, which will represent the population and the percentage of fitness in the entire population from the example table. The first step is to select a random point on the chart and fix it, which is done by generating a random number between zero and the sum of all fitnesses. Then, similar to the game of roulette, the value will land somewhere in a predefined quadrant of a variable that has been assigned to a particular chromosome. The chance that a chromosome will be selected is $p_x = \frac{f_x}{\sum_{j=1}^{N} f_j}$. Each of the different steps the selection algorithm goes through can be described as follows:

- Calculate the sum of all fitness values $f_x \in F$, where F is the set of all fitness values. The result will be calculated as $S = \sum_{x}^{F} f_x$.
- Generate a random variable a, where $0 \leq a \leq S$.
- Start from the origin of the population till the fixed point.
- Then, if you reached the fixed point, take the related chromosome that the fixed point points to, and set it as the first parent. Do the same for the second parent.

From the example, you can see that a high fitness level means that it has a higher chance of choosing that particular chromosome for the next generation. The upside is that the technique is quite simple to implement and is free of biases. The downside is that you have a dependence on the variance of the fitness results, and convergence might finish prematurely.

A solution to the issues with Roulette Wheel where developed by Baker et al. [7], and is called Rank Selection. The difference with Roulette is that Ranked mainly focuses on the ranking between all different chromosomes, and doesn't focus on the variance in the population. Ranked makes use of a variable called selection pressure, which is between 1 and 2. The possibility for a rank position of a chromosome (R_i), is calculated with the following equation:

$$P(R_i) = \frac{1}{n}(sp - (2sp-2)\frac{i-1}{n-1}),$$
$$where\ 1 \leq i \leq n, 1 \leq sp \leq 2 \text{ and } P(R_i) \geq 0, \sum_{i=1}^{n} P(R_i) = 1$$

The mathematical function will try to spread the probability, not based on the fitness value, but on where the fitness criteria rank between the rest of the fitness criteria and setting a fixed probability for each ranking. The rule is still that the combined probability of all ranks is 1, and that each chance for a chromosome is still possible. The probability distribution depends on the selection pressure sp variable, when increased, creates a smaller disparity between each of the different rank probabilities.

Crossover and Mutation Both of these are common examples of how selection can be applied, which will then be passed over to crossover. One of the simplest examples has already been shown in Figure 4.8, which is called uniform crossover. This takes random genes from both and swaps them interchangeably to generate a new child with chromosomes of both parents. There are also other forms, including the following, with each given a concise description:

- Single/two point/k-point crossover: Each of these is in the same category of crossover. The commonality between each is that the cross-over divides two parents into fragments and alternates between each parent, to select which of the genes goes into the offspring. Single crossover divides into two fragments, two into three, and k into k+1 fragments to use. An example can be seen in Figure 4.13.

- Partially Mapped(PMX): This crossover iteration is used by taking two parents and using a specific part from index n to m. For each of these items, we make it so that the values of one parent in that particular segment are replaced with those of the other parent. Take, for example, parent #1 is 1 | 23 | 3 and parent #2 is 5 | 67 | 7. In this case, the 2 in parent #1 will be interchanged into 6 in the first child of parent #1 with parent #2, and vice versa for the second child. This is the same for values of 3 and 7. This leads to two children with the numbers 1 | 67 | 7 and 5 | 23 | 3.

There are also plenty of other variations of crossover, such as Order Crossover(OX) [73], Precedence Preservative(PPX) [87], Shuffle/Cycle Crossover [79]. These three iterations are both quite simple to implement, like k-point. The last choice to make to finish a working GA is to define the mutation component.

Three common types of mutations are used in GA: displacement mutation, Simple-Inversion Mutation(SIM), and scramble mutation. Displacement is taking a subset of the crossover child data and placing it somewhere randomly in the data array. Let's suppose there is a substring from index x to y, for displacement and transform it to index z. Then the substring indexes will be from z to z+(y-x) and the other chromosome after will be from z+(y-x)+1 up till the last chromosome. This technique is quite simple in execution, but depending on the size of the data set might suffer

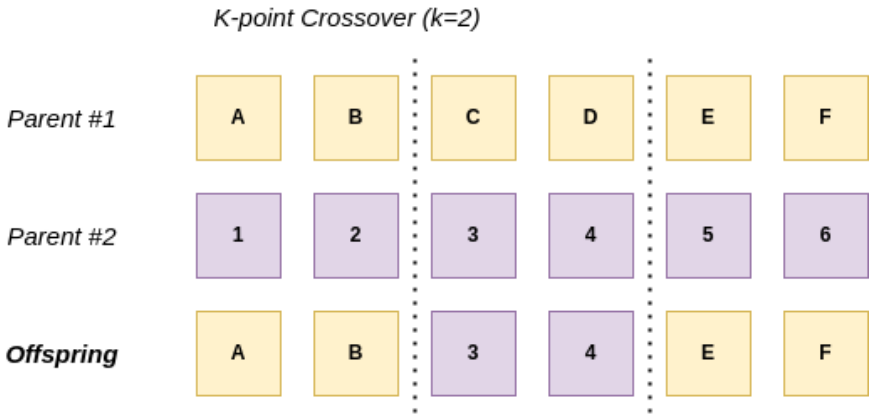


Figure 4.13: K-point Crossover Example

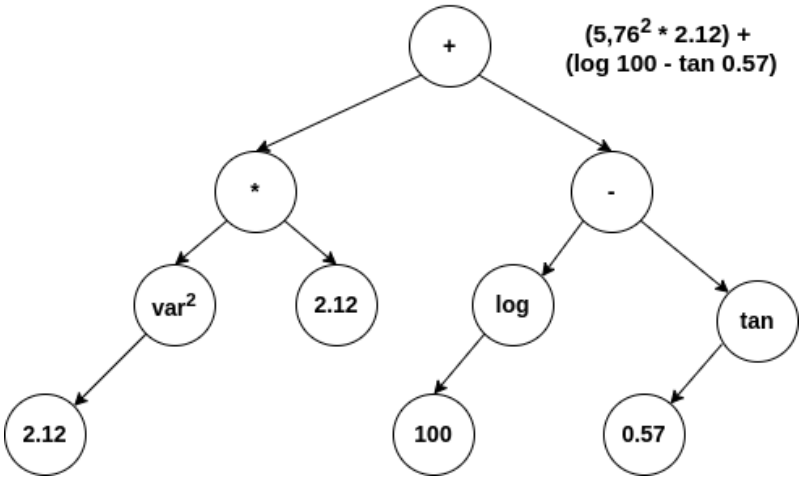


Figure 4.14: Genetic Programming example.

from premature convergence [48]. When using SIM, a subset is taken from the child from two indices x and y, where all variables in between will be in reverse order. This technique is really simple to execute but suffers from the issue of premature convergence [48]. Both of the techniques can also be combined and are known as Displacement Inversion Mutation, where the substring is first inverted and then placed somewhere else in the gene. Lastly, the scramble method takes a gene from crossover, swaps the chromosome in a random order, and checks whether the generated result has improved or not. The upside is that this technique can be used for large-sized problem instances; however suffers from long-term deterioration of results during longer-term generation.

Depending on a particular problem statement, one can fine-tune the specific encoding crossover and mutation to develop the optimal solution. These solutions can be found in a range of examples image/video processing, medical imaging, gaming, localisation, and many more. However, for this project, the main interest lies in the use of GA for grammar. This means that either the grammar or a sample of WAG is needed as input to execute the GA algorithms. Some of these are examples of papers in the neighbouring realm of NLP [6, 16]. These types of algorithms are part of a specialised version of GA called Genetic Programming(GP) [3]. Genetic programming is a subset of GA where the population is made up of programming-related data. An example of these can be best sketched with an example found in Figure 4.14.

The figure represents a tree structure commonly found in programming languages, in which either an interpreter or a compiler will interpret the node to perform some form of manipulation.

The example might represent for example a scientific calculator which uses symbols($\sqrt{}$, log, e), variables (x, y, z), or numerical values ($\mathbb{R}, \mathbb{I}, \mathbb{Z}$). In the example the equation $(5.67^2 * 2.12) + (log(100) - tan(0,57))$ is expressed in tree form. What makes the GP so special is that these trees can be manipulated into different forms and can be made to have different formulations. This is not exclusively mathematical, but the input for selection is just like other genetic algorithms. The selection process involves selecting part of the tree to perform the crossover on. For crossover, it might take two different trees, which both take a sub-tree, and swapping both trees to create two new offspring(trees). These offspring will be potentially altered during mutation. The form of mutation can differ from mutating leaves or the mutation sub-tree of the offspring. Both mutation and crossover can have different types of implementation, such as PMX and SIM.

Evolutionary Grammar There are also a lot of different variants of implementations of genetic programming, of which one is very interesting for the current problem statement. This particular technique is called Evolutionary Grammar [90]. Instead of using trees, it generates coding samples based on the definition of a specified grammar. This makes evolutionary grammar very interesting, for this project, since we can use the WAG's grammar definition to generate samples. The original form in which the evolutionary grammar is written is the Backus-Naur Form(BNF) [5], which has been used as a meta-language to define ALGOL-60. BNF uses a few signs to define the grammar, and they are as follows:

- <>: These triangular brackets are used to define each of the rules of the grammar, similar to Non-Terminal values in WAG/CFG.
- ::= : These combinations of symbols are the separator for the left-hand-side(LHS) and right-hand-side(RHS) of the grammar/production rule and are similar to the $\rightarrow$ in WAG/CFG.
- | : The 'OR' sign is used to define different options within the grammar. And example of this would be <Noun> :== 'desk' | 'lamp'.
- { and } : These curly braces are used to indicate repetitive parts of a grammar rule, for example <array_content> :== [<identifier> { , <identifier> }]

```
1 <expr> ::= a = 0; b = 0; c = 0; <e>
2 <e> ::= <op> | <op>;<e> | <sub> | <sub>;<e>
3 <op> ::= a += 1 | b += 1 | c += 1
4 <sub> ::= a -= 1 | b -= 1 | c -= 1
```

Listing 4.4: Example of BNF.

To develop a grammar for a specific program, each of the grammar rules can be defined by adding new rules. An example of such a grammar can be seen in Listing 4.4. These small examples generate a different set of additions and subtractions on the variables a, b and c. A possible result from the grammar is the following:

```
1     a = 0; b = 0; c = 0; a += 1; c -= 1; c += 1
2     a =  1
3     b =  0
4     c =  0
```

The advantage of BNF is that the grammar notation doesn't have many rules, meaning it can describe several grammars without the problem of mandatory declarations. However, this also means that quite some types that are common in most grammars and languages have to be declared themselves. Examples of these types can be variables such as identifiers, characters/letters, Doubles, Strings, etcetera. Some extensions define some of these common types, of which there are Extended-BNF(EBNF) [1] and Augmented-BNF(ABNF) [24]. ABNF, as the name implies, is an augmented version of BNF, which has specifically been developed for network communication protocols, such as TCP/IP. It contains functionalities such as concatenation, comments, value ranges, and others. EBNF not only add particular functionalities, but also defines the common types referred to before, such as strings, letters, symbols, identifiers, and others. The caveat is that there are different implementations of EBNF, which can lead to different implementations of grammars [113]. Therefore, it is important to define the grammar when applying the evolutionary grammar to avoid misinterpretation.

Chapter 5

Results

This section shows the results of each developed implementation in the Research section. The first step is to compare the results of both types of grammar: basic English and WAGON. After the generation of each of the grammar samples, each of the four researched techniques will be compared based on both context and their applicability to either of the two meta-grammars. Furthermore, a section will take more quantitative analysis, in which we compare variables such as speed, grammatical correctness, and more. This is partially done by using the samples as input for an ANTLR parser, and comparing if the result is a fully or partially parsable. The results chapter ends with complementary results for this thesis, mainly a syntax highlighter and a TUI to generate a sample of grammar not specifically for WAG.

As for all of these examples, there is a GitHub page[106] at the end of the thesis where other researchers can tinker with grammar generation, for each of the researched techniques.

5.1 Comparing Generated Results

To compare each of the different grammars, it would be an advantage to try and be able to parse the grammar based on the overall meta-grammar. More on this in the next subsection. Then will take a sample of each of the generated files and explain how these results were achieved, comparing them to one another and the meta-grammar. Furthermore, a separate sub-section provides the difference between the parses of both restricted and dynamically generated NT, terminals, and attributes.

5.1.1 ANTLR Preview

For Grammarinator, the meta-grammar had to be developed for both the simple English and WAGON meta-grammar variants. An additional benefit to using ANTLR is that it has a plugin called ANTLR Preview. This tool takes the grammar definition written in a *.g*4 file and an input of grammar to try to generate a parse tree from both. The meta-grammars provide the grammar to parse, and the input will be given by the generated WAG files. An example of such generated parses, in this case WAGON, can be seen in Figure 5.1.

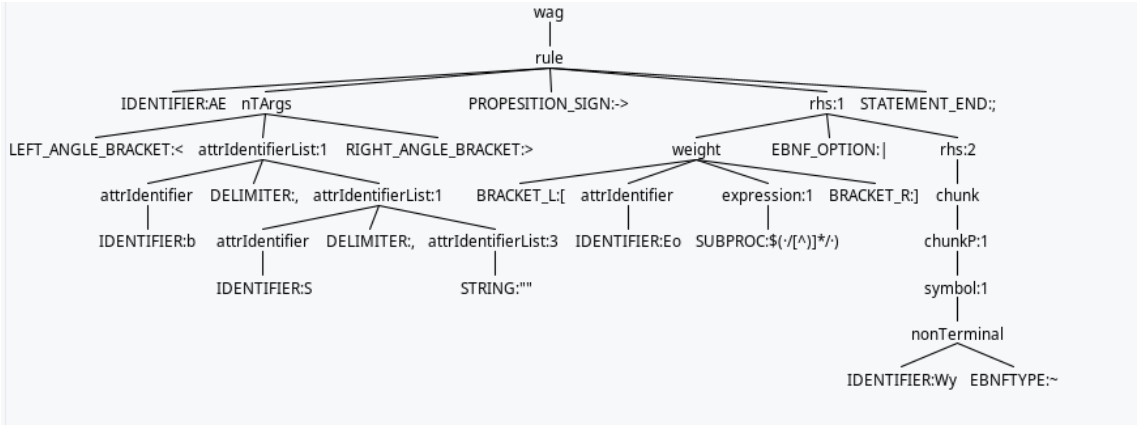


Figure 5.1: Wag Syntax highlighter example.

Some of the generated results might give partial parsing, meaning that only a subset of variables can be parsed. In that case will take a set of samples and aggregate all the percentages of partial parses, giving an average percentage of variables that can be parsed.

One of the advantages of using Antlr Preview is that we found a parsing bug in the WAGON grammar. An example was the use of attribute types in places where they can't be defined. This is based on the specifications in the related thesis paper [27]. This meta-grammar defines a third type of attribute, local, which is only used in the local procedure declaration; however, it was declared in a place of either synthesised or inherited attributes. Also, other changes were made to WAGON to make the results more consistent, such as changing some of the "*" (0 or more NT) with "+"(1 or more NT), to prevent empty generated files or sections in the grammar.

5.1.2 Generated Results.

Each of the generated results will describe the content that has been generated and, at first, will look at the generated parses of WAGON. This is because these results are the most ambiguous of both meta-grammars. This means that it will show the largest variance in the generated results of each of the four generators. Hereafter, a separate section will discuss the difference between the dynamically defined variables in WAGon and the predefined ones in basic English meta-grammar.

Grammarinator Grammarinator was the first to be implemented, using the ANTLR meta-grammar of WAGON. One of the generated WAGON samples is the following:

```
1 Q<*y_,&wv>->[$()]wVPjPuP?<'ac^I'>;
2 VF_9E<&s>->[0.6]('')@;
3 T<&MaQk>->-05.87-282;
4 z<&bmb>->[$()]"FNU,MRB+";
```

Listing 5.1: Grammarinator WAG generated sample.

In this example, two things spring out: the generated values are indeed random strings, and even when using the same grammar file, the parses that Grammarinator generates are not always parsable. However, most of the code is parsable. The downside of these results is that the generated variables have no reference to one another, and don't relate to one another, since each is only used once. A fuzzer would be fine with the results; however, in the case of grammarware, like grammar inference and grammar convergence, it would be important to know the context of each of the non-terminals, terminals, attributes, and weight values.

As for the result of the ANTLR Preview parses, they are quite large, but in the appendix, two sections of the parse are shown. The first in Figure A.4 shows the first rule parsing normally, and the second in Figure A.4 shows the issues with parsing the third procedure.

Transformers The generated results of the Transformers are a bit different from the others in that they do not only use the meta-grammar but where provided with an example grammar to show to which form the results should conform. The reason for doing so is that GPT specifically uses prompting to generate results. When initially running the program, it interprets the result in many different forms. By specifying that it should generate grammar, based on the meta-grammar and with an example, it would generate more targeted parses. The prompt that was used had the following structure:

> Generate a grammar based on this meta-grammar: <Meta-grammar>, where a sample is <WAG sample>

As a sample for the WAG, snippets were taken from a Pokémon grammar formulation in the WAGON project, an example of which is the following:

```
1  S -> { $shiny = false ; $nickname = false ; $gender = " U "; $item
       = false }
2      Pokemon < $shiny, $nickname, $gender, $item >;
3
4  Pokemon <&shiny, &nickname, &gender, &item > ->
5      Info <&nickname, &gender, &item >
6      { $d_ability = false; $d_shiny = false; $d_level = false;
           $d_happy = false;
7           $d_nat = false; $d_ev = false; $d_iv = false; }
8      Optionals < $d_ability, $d_shiny, $d_level, $d_happy, $d_nat,
           $d_ev, $d_iv , &shiny >*
9      { $move_count = 0; } Moves < $move_count >+;
10
11 Info < &nickname, &gender, &item > -> Name
```

Listing 5.2: Pokemon grammar in WAG portion

The sample GPT generated was quite varied. In some cases, the generator would generate samples that seemed to be correct WAG, but in some cases, it interpreted the grammar differently. Take the two examples in Listing 5.3 and 5.4. The first example shows that GPT can generate semi-parsable results, which has an advantage compared to the parses of Grammarinator. This is because GPT uses the grammar input given as an example and generates new non-terminals that are semi-contextual to the grammar. In the example, no Evolution non-terminal was specified, but it is indeed the case that in the game of Pokémon, there are rules for evolution. The inference that GPT takes is a welcome addition to generation, but it can make wrong inferences at times, hence the second example.

The second example shows that GPT also infers the input as code and tries to generate a c code example using the variable. Although it retains these values, it can't be parsed at all since the Context Free Grammar: i.e. $NT \rightarrow NT \cup T$ is not declared anywhere. From this, it can be concluded that Transformers, specifically GPT, focus not on grammatical correctness, but on generating samples that are similar to the input. As for the semi-parses of the partially parsed example, go to Figure A.6 in the Appendix.

```
Evolves <%d_ability, $d_shiney, &nickname, &gender, &item> -> Info
    <&nickname, &gender, &item>
    {$d_ability = false; $d_shiny = false; $d_level = false;
        $d_happy = false; $d_nat = false; $d_iv = false}
    Optional<&d_ability, &d_shiny, $d_level, $d_happy, $d_nat,
        $d_iv>{$move_count = 0;} Moves<$
```

Listing 5.3: Pokemon grammar in WAG portion

```
# include < Pokemon.h >

int main()
{
    # include < u8.h >
    int status = 0;

    int status_size = 20;

    Pokemon pokemon = [[ [ " status " , " status ", " 0   " ]];

    for ( int i = 0 ; i < status_size ; i ++ ) {
        player.status.forEach(Pokomen(name,status)) {
            status_size++;
        }
    }
    return 0;
}
```

Listing 5.4: Randomized C++ generation with GPT2

XSmith The implementation of XSmith had initially some issues in generation, the main reason for this is that XSmith itself is specifically developed to write language samples. It doesn't inherently mean that it is not possible, since there are some grammar samples, but its technique had some conventions that complicated development. An example that is particularly catered to language is the use of code blocks, called "Block", this also goes for base types like "Expressions" and "Lambda" functions.

The foremost struggle was with some of the base types of XSmith, these types are predefined in XSmith, such as String and Boolean. Though this is quite useful, it also caused some issues with some of the defined types. These mainly consist of the implementation collections, such as arrays. Which is fine, since there are examples of CFG using terminals of collections. The downside is that for the array to work you have to implement multiple specific different implementations, such as "MutableStructuralRecordReference" and "MutableStructuralRecordAssignmentStatement". This is, however, mitigated by declaring them in code from the example grammar but never assigning them to any rule.

One advantage that XSmith has compared to the other generational techniques is that it uses a technique to try and preserve semantic validity. This is to say that the generated non-terminal and attribute will be generated in such a way that each consecutive procedure can use these variables in its rules. This differs from Grammarinator and evolutionary grammar, which generate randomised variables in the case of WAGON. The downside of these semantical variables is that the name convention is a combination of a name with a number; i.e. "lift_234" or "attr_756". It is still better than randomised variables, but still less contextual than some of the generated samples from

Transformers. An example of XSmith generation can be seen in Listing 5.5 and a larger one in the Appendix Listing A.3.

```
    Lift_32 -> Lift_33;
    Lift_31 -> Lift_32;
    Lift_30 -> Lift_31;
    Lift_29 -> Lift_30;
    Lift_28 -> Lift_29;
    Lift_27 -> Lift_28 {arg_26= true;};
    Lift_26 -> Lift_27;
    Lift_25 -> -15356745
    Lift_24 -> Lift_26 {args_24 = 152135;, args_25=false;}
```

Listing 5.5: Small XSmith example

The generated results are valid WAG and partially parsable WAG samples. There were some issues with declaring the attributes, specifically the typing and assignment in this case, the only way I could assign a value to an attribute was to use punctuation with the variable and the attribute.

Evolutionary Grammar As discussed in the Research chapter, we use Grape to implement the generator. For this, we change both the definition of English and WAGON grammar to BNF. And generated samples for both. One of the generated parses for EG can be seen in Listing 5.6

```
    FR -> {$zK = if (12 x 1.8 != -2) // (0.956 % 8 / -2 x 0 > 0 +
        0.89 / -5.0 - 0) // (-0 != 1.4 % -7.6) \// false && !false
        && (0.890 / 0.78 - 9.98142 % 6 + 3.37 + 0 != -0.430 + 0 x
        285 / 0) && true then (0.0 % -9 <= 0.6 + 0) // (0.0 - 0 +
        -334.0 > -2.9 + 0.7) // (0 - -50.8 >= 40.2 % 0 x 6.293) //
        (0 <= 0) else if (0.4 - -4.2 % 0 < -0.9 - 0) // (-0 / 0.4 !=
         -54.0 + -9 / 3.5 x 0 x -4.8410 - 0.92 - 2.7) && false then
        (0.36 / -5 < 984.461 - -7) && !true && (0 != 50) // !false
        else (-4.95 == -4.5);}; u<&b, *j> -> [0.8] 7.6 - -51.9 !=
        0.0; Ppo<&zn, *j4> -> (-7 != 0 x -0 / 0.46 / 3)?; R7 ->
        [0.6] 0.7 > -5 {&Kij7 = if (21.8 / 0 < 0 x <Float> <SumP>)
        then <Disjunct>;}; <Rules>
```

Listing 5.6: English Grammar sample from Grape.

The results that Grape generates for Grammarinator look similar to those that are generated for Grammarinator. This is mainly the case that both of these generators are focused on the grammar and use a grammar input, i.e. ANTLR or BNF, to generate the results. Similarly, in this, all the variables(non-terminals, terminals, and attributes) are randomly generated strings. Again, these results are applicable for fuzzing, however, the use case of grammarware, such as Grammar Inference and Convergence, might give issues. For an example of the parse tree, see Figure A.8 A.9 and A.10 in the Appendix.

Concluding sample generation techniques. All of these techniques have their advantage and disadvantages, but based on the generated samples, it can be concluded that Grammarinator, transformers, and evolutionary grammars are better suited for grammar generation for meta-grammar than XSmith. This is mainly the case that most features written for XSmith are written specifically for languages, which forces the user to perform tricks to be able to generate results, even though it has the advantage of using semantical variables.

As for the others, we can separate them into two categories: grammatically correct and contextual representation. In the former case both Grammarinator and EG, are good candidates to

generate grammatically correct samples, however, it has an issue when generating variables that are contextual to some input representation. While Transformer has the opposite problem, it is quite good at interpreting some context from the grammar. It suffers however, in the grammatical correctness department, since the generated samples are not restricted by the grammar rules.

In the next sub-section, a comparison will be made on the level of expression, in which we compare to generate samples of the English and WAGON grammar and explain why the results differ.

5.1.3 Comparing Results between basic English and WAGON.

In the research, we specified two types of meta-grammars: English simple and WAGON. This was mainly done to show the difference in the generated results. For this comparison, I took the example of Evolutionary Grammar, however, the result of the other generators might differ. In the case of Transformer, since it doesn't enforce grammar rules, there is no proper conclusion that can be taken from the results, since it depends on the interpretation of the transformer. Grammarinator itself is built similarly, and the difference there would mainly be in the implementation of the grammar and its perspective syntax. XSmith itself can't implement the English grammar variant, since it only works by defining grammar rules, and then generates semantic variables.

The example of WAGON has already been shown in the previous section as a sample in Figure 5.6. As for the English grammar, we have the following variant.:

```
1    Preposition{$proper = $tense;} -> [0.4]VP door ate DET for;
2    Verb{} -> NP Pronoun ate Preposition
```

Listing 5.7: English Grammar sample from Grape.

In both cases, there are similarities and differences between the given parses. The most obvious difference is that the grammar rules of basic English grammar retain the context in English Grammar. In the case of dynamical variables, it is dependent on the implementation, where random variables for WAGON don't provide reference to any concept, while Transformers infer type through their LLMs.

Where both are similar and also quite consistent through all techniques and meta-grammars, it is a semantics issue. When using a dynamically generated string with no reference, you don't generate similar related context. But even in the case you declare all non-terminals, terminals, and attributes, there are still issues with semantics. Since in the English language, a proposition is not made out of a VP, terminals(door, ate), or some determinant variable.

This can be remedied by specifying more specific grammar rules that already assign some of the known attributes. The example shows that declaring fixed values can make grammar more predictable. However, this process is quite time-consuming, and also increases the potential depth of the parse tree, needing potentially more resources to generate extremely specific cases.

To summarize the generational techniques further, a table is provided with an overview of the characteristics of each of the different techniques and provides some metrics such as speed of generation, and percentage of correct parses for both Grammarinator and EG.

5.2 Generator Characteristics.

To compare all of the generation techniques, a table was made to give an overview of each of the different characteristics that make each of the generations unique. We provided data on the set of characteristics, given the following description:

- *Batch generated samples*: If we want to use grammar for grammarware, it might need a large set of newly generated grammars. This is specified as either a true or false statement.
- *Grammar Input Format*: This is a true or false statement whether the generated results use a grammar definition for the input; i.e. ANTLR or some form of BNF.
- *Identifier references*: This question asks whether each of the attributes of the non-terminal is related to the grammar; i.e. are they semantically valid?
- *Parsing Results*: This variable shows whether the results of the generation are accurate. If this is the case, five samples of both the simple English grammar and WAGon will be put into ANTLR Preview, and the aggregate percentage of correct parses will be given. If this is not the case or is unclear, a reason will be provided.
- *Time to generate samples (s)*: Each of the generations takes time, to compare each on their performance, the generators had the task to generate a hundred, a thousand, and ten thousand samples, each having a timer. The specific type of timer is qualified in the results.

In Figure 5.1, you can see the table containing each of the different results, and we will go through each of these points.

Firstly all except XSmith can generate a multitude of samples at once. The reason for this is that the tool to write XSmith for called DrRacket, generates one sample each run, and doesn't specify anywhere where multiple samples can be generated.

As for the correct parses, two generators are most interesting, mainly Grammarinator and Grape. These results can also be seen in the comparison of the samples, and most likely have to do with them being based on the grammar definition. For GPT, the result, as mentioned before, is randomised. Them being either non-parsable or semi-parsable, depending on the interpretation. Lastly, for XSmith, there were difficulties generating correct parses, since, as mentioned in the previous section, it has an issue specifying the English Grammar, so no clear comparison could be made. In the case of basic English grammar, all five samples of Grammarinator and EG were parsed completely; i.e. hundred percent. However, this is not the case for WAGON, which has two causes: conflict in the parsing decision and the grammar itself. ANTLR Preview sometimes makes preferential parsing decisions, which makes larger grammars sometimes difficult to parse, since some symbols or rules conflict. Secondly, as mentioned before, the grammar of WAGON had to be changed since the original grammar only provided a partial parse. Interestingly, the result from Grammarinator also didn't fully parse, though it is based on the grammar.

	Batch generated samples.	Grammar Input Format.	Identifier References	Parsing Results (English / WAGON)	Time to generate samples(s). (100, 1000, 10.000).
GR	✓	✓	Randomized String / value	(100%, 81,98%)	(0.373, 1.556, 11.540)
GPT	✓	X	Inference from corpus / input	X	(1200.76, 11.397.555, 117025.775)
XSmith	X	X	Identifier_ {number}	X	X
Grape	✓	✓	Randomised String / value	(100%, 77,03%)	(0.075, 0.821, 8.384)

Table 5.1: Generation technique comparison.

The third and fourth points have already been specified in both research and previous results, so they won't be discussed again. The next characteristic is more important, which is the performance of each of the three generative techniques that could produce multiple samples: Grammarinator, GPT, and Grape. For reference, the generation has been run on a laptop with the following specifications:

- **Model**: Lenovo ThinkPad P1 Gen 3.l.
- **Memory**: 16GB.
- **Processor**: Intel Core ™ - i7-10750H x 12.
- **Graphics**: Quadro T1000.
- **Disk**: Samsung 970 EVO M.2 1TB (SSD)

From the results, it can be seen that both Grammarinator and Grape are way faster than generating samples in GPT. The main reason for this is that GPT is an LLM with 1.5 billion parameters, though it has been pre-trained, the network is still really large. Each result for GPT takes about 12 seconds, which isn't that long; however, for extremely large datasets, that is a different case. For the research, the last result couldn't even be executed since when taking the aggregate per sample / s of a hundred and thousand and multiplying by ten thousand will get 117.025,775 seconds of run time which would take about 32.507 hrs, which will take too long and would cost unnecessary power since it is only meant as a comparison between the other generators. As for Grammarinator and Grape, both can generate a lot of samples quite quickly, with Grape having a smaller edge than Grammarinator. The calculation of time for Grammarinator was done through the time command in the Linux CLI(Ubuntu) and used the real-time variable. While for both Grape and GPT calculated the difference was calculated when the algorithm started until all of the generations finished in a for loop with the length of the number of samples using the Python time library.

This concludes both sections by comparing each of the different generational techniques, however during the implementation, additional tools have been created. These are syntax highlighters for WAG, to highlight attributes and variables and a TUI to generate samples using GPT.

5.3 Complimentary Results

5.3.1 Syntax Highlighter

The first step is to create a syntax highlighter to distinguish parts of the grammar. This is done firstly by declaring the file type, such that the syntax highlighter extension can be applied to the correct file. For this, it has been declared that the ".wag" file type has been made specifically for this purpose. The syntax highlighter itself is written for Visual Studio Code, which is a code editor, in combination with the Yeoman [111] framework. The first step when running Yeoman was to add a syntax highlighter by specifying that the highlighter only works on the .wag files. This resulted in the generation of a folder with a Markup Language file for wag specifications and a language configuration file for general language features.

These features are defined in the **language-configuration.json** file and are separated into three sections: *brackets*, *autoClosingPairs*, and *surroundingPairs*. Brackets are signs that surround a value, like characters and strings; these signs surround a value and define the specific type. *AutoClosingPairs* are signs that open and close either statements, arrays, comments, or any other type that has an opening sign and closing sign, f.e, []. The last one is similar; the difference between both is that *autoClosingPairs* generates the closing sign after the opening sign is filled in. The highlighting of the code itself is done through the **wag.tmLanguage.json** file, in which five different types of variables will be highlighted. An example of a file which has the WAG syntax highlighter:

- Keywords: These are the words in the language that are reserved, since they are a vital part of the grammar. In the case of the current implementation of WAG, these are: if, else, then, true, false, and in.
- String: These are quite self-explanatory; every value that starts and ends with a double quote presents a String value.
- Attributes: These highlight the attributes in the Attribute Grammar of WAG. There are three signs which can refer to these attributes: '&', '$', and '*'. The "&" sign represents the synthesised attributes, "$" represents local attributes, and last, the inherited attributes are represented by "*".
- Comments: These are the sections within the code that have been commented on. In the following case, it is a multi-line where '/*' and '*/' are the beginning and end symbols of the comment.

```
/*
    This is a comment section from the syntax highlighter.

    Where you can define the program or general comments

    @author: Nick Wolters
    @version: 1.0
*/

    Sentence {$active = true} -> NP {$proper = false} VP {$active =
        true}
```

Listing 5.8: Code Highlighting example of WAG.

5.3.2 Transformer TUI.

The last result is a small addition to the grammar transformer network in the form of a Textual User Interface(TUI). Since there is a multitude of potential Transformer networks, selecting which one to generate was a challenge. The user has to enter the name they want to give to their file and the number of samples they want to generate. This tool has been used to generate each of the samples used in the performance comparison.

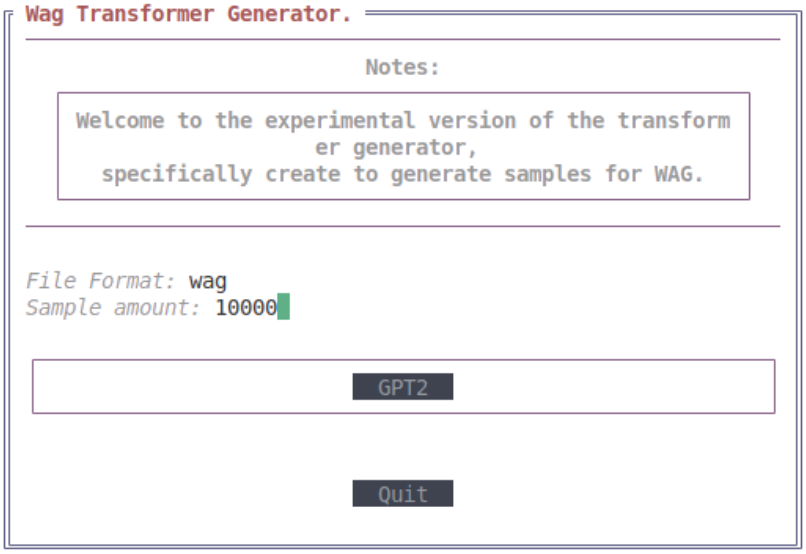


Figure 5.2: Wag Syntax highlight comments.

Chapter 6

Related Work

6.1 Probabilistic Grammar Evolution.

A solution that looks promising is based on the research for the development of EG and combining these with probabilistic context-free grammars, which is named: probabilistic Grammar evolution (PGE) [74, 75, 54]. The idea of PGE is that each of the different alternatives in each of the grammar rules in BNF will be given a probability value and which combined probability should equal to one ($P(x) \in G_R$ and $\sum_x^n P(x) = 1$, where G_r is the grammar rule and n the number of alternatives). This differs from the original selection process of GE, which uses a genotype of randomly generated integers between $(0 \ldots 255)$ and depends on the number of alternatives n. Each of the elements in the genotype array will start from the initial state and go down the tree by using the modulo (%) of the random number with the total number of different paths. Each of the alternatives will be provided with an incremental number from $(0 \ldots n-1)$. An example of how this affects the grammar can be seen in Figure 6.1.

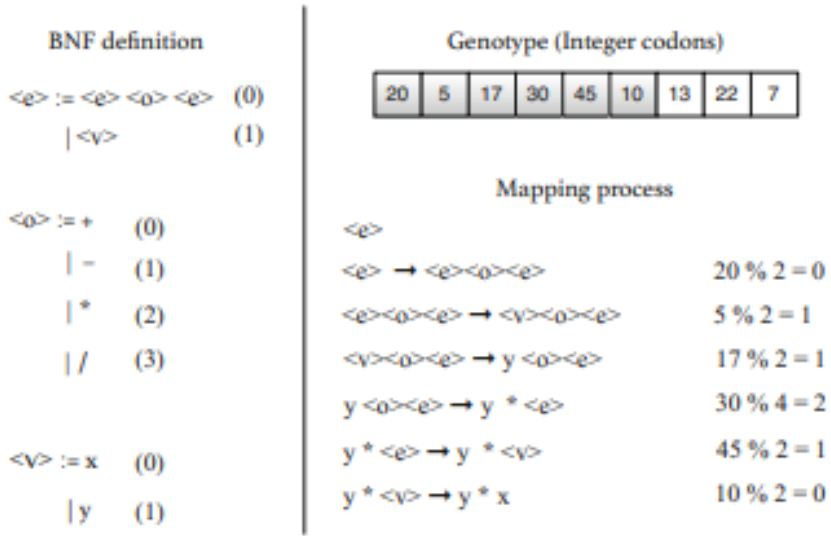


Figure 6.1: Implementation of alternative choosing in standard GE. [54]

The example shows a simple grammar, which generates basic equations using operations(addition, subtraction, multiplication, and division), values (x,y) and elements refer to values or an operation on two elements. The GE generates the genotype of integers and starts at the element rule. The first element <e> in the genotype is 20 and has two different alternatives <e><o><e>(0) and

$< v >(1)$, so n = 2. So the alternate is $20 mod 2 = 0$, which gives that <e><o><e> is chosen, and from this point you continue. If you have multiple NTs as in this step, the algorithm will traverse each from left, until it encounters a terminal and continues to the right. This happens either until all values declared are of the non-terminal variety, or the traversal of the genotype has used the last element.

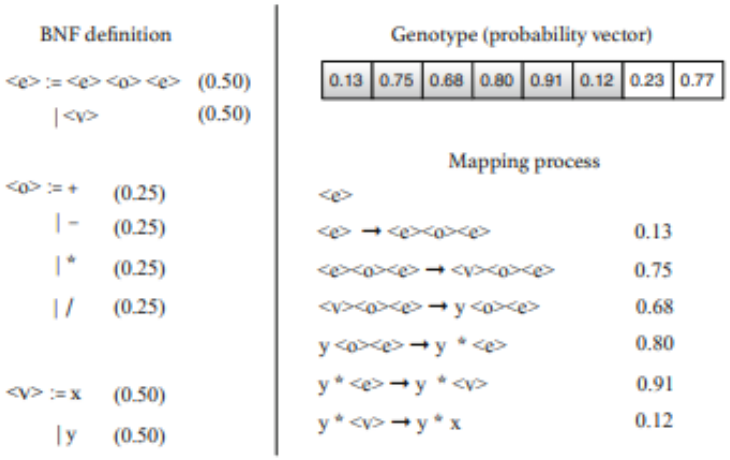


Figure 6.2: Grammar example of PGE [54]

A similar basic structure is used, however, the genotype uses probability at its base instead of doing modules on randomly generated integers. The genotype now contains a value between zero and 1, and as mentioned before, each rule gives is alternative probability, and the sum equals one. The decision process now works by comparing each gene in the genotype to which quadrant of the probability the value is. Take Figure 6.2 as an example, where the same steps are taken by using probability. The first choice for the grammar rule to go from <e> to <e><o><e> is done by checking where 0.13 is on the range of probabilities in the BNF definition. That is to say that both probabilities from are 0.5, which means that <e><o><e> contains all values from 0.01 to 0.50 and <v> 0.51 to 1, which can be seen in the next mapping where 0.75 leads into <e> becoming <v>. When written in mathematical form, where n is the current step, the range of probability P_r gives that $P_r(x,y) = (P_{n-1} + 0.01, P_{n-1} + P_n)$.

This particular way of generating grammar can aid in the development of the WAG generation using EG. By using these probabilities, the results of the current implementation of the Grape algorithm could potentially be improved. The upside is that the code from [74] is publicly available through GitHub, and the software, similarly, is an extension to DEAP. This means that a potential conversion should not be difficult.

6.2 Grammar-Constrained Decoding

The generation of grammar can also be performed by combining different models; an example of such a combined generator is Grammar-Constrained Decoding [31, 81]. Similar to this thesis implementation of GPT both uses an LLM to generate results based on grammar. The difference is that instead of using grammar as an input to generate results, they used grammar to restrict the generation to examples that adhere to the grammar definition. Secondly, their model uses grammar to generate languages; specifically, NLP languages, while ours generates other grammars from the meta-grammar.

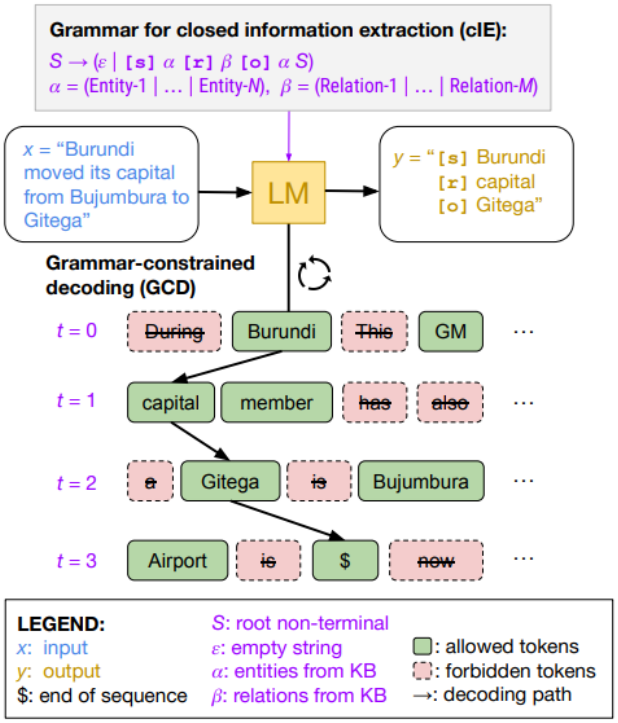


Figure 6.3: Grammar-Constrained Decoding example. [31]

The paper also mentions the use of Input-dependent grammars (IDG), which are grammars that are dependent on the input to retain semantics. The use of IDG could potentially solve a problem that the current generations have, which is the lack of semantic correctness.

Chapter 7

Conclusion

This chapter is the end of the thesis and will answer the research questions in Chapter 3. Furthermore, there is a discussion portion, which discusses some hurdles and changes that were made. And ending with recommendations for future work on the topic of grammar generation. The main research question of the thesis was:

> *Which generational techniques can be implemented to generate grammar samples from a meta-grammar, using WAG as the output grammar?*

The answer is that through the use of Grammarinator, GPT, XSmith, and Grape we can generate samples for both implementations of the simple English and WAGON meta-grammars. A model of the implementation can be seen in Figure in 7.1, and is similar to the figure on the generative model in the introduction.

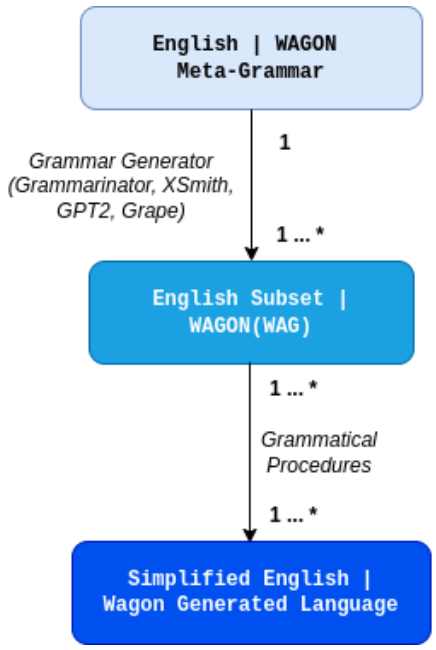


Figure 7.1: Overview of the generative implementation.

The implication is that there a four sets of two meta-grammars as the results; however, the English grammar has not been implemented in XSmith, since it uses semantic generation;i.e. non-terminal_number, and can't therefore implement the predefined attributes and non-terminals.

Otherwise, all of these techniques were successful in generating grammar samples. The resulting generated samples are different in each of the cases, depending on the generative technique and selected meta-grammar. This leads us to the first sub-question:

> *What are the results when the generated samples are parsed, and how does each technique compare?*

To summarise, of the four different generators, Grammarinator and Grape were the most grammatically correct, while the results of GPT related the most to the context of the input. While it was a struggle to implement XSmith, since the tool is mainly meant to generate language samples, and isn't optimised for grammar. Furthermore, XSmith is restricted in that the tool used to develop XSmith code, called DrRacket, only generates a single sample at a time. Only in the case of WAGON could it produce some samples, which seem to be grammatically correct.

As for the characteristics of the other three, it seems that both Grammarinator and Grape are fast in generating new samples, being able to generate ten thousand samples in less than 15 seconds. Grape has a small advantage of barely 3 seconds compared to Grammarinator. While GPT is not slow per se, since it generates a sample round every 12 seconds, however is comparatively extremely slow. When comparing the time to generate 100 samples, it is **3219 times** slower than Grammarinator, and **16010 times** slower than Grape.

The last characteristic to compare is the parsing of the results themselves; in the case of the basic English grammar, both Grammarinator and Grape could fully parse both samples. As for WAGON, both couldn't be completely parsed, but were somewhat comparable in the averaged percentage of partial parses; Grammarinator 81.98%, and Grape 77.03%. The reason is because of the higher level expressiveness of WAGON, and ANTLR Preview preferences for other grammar rules over others.

> *Does each meta-grammar generate differing results, and what constitutes that difference?*

The comparison of the meta-grammars is between the basic English and WAGon meta-grammars, and it was noticeable that the difference can mainly be found in the context of the results. In the case of English grammar, when interpreting the result as grammar rules for a subset of English, it is comparable in structure to similar rules from Figure 2.1, the comparison between CFG and CNF in the Background chapter. While the results in WAGON are mostly parsable for Grammarinator and Grape in structure; however, it is quite dissimilar to the comparison of the Pokémon sample grammar in the WAGON paper. The main difference is that the WAGON meta-grammar has not specified its variables, which leads to random strings that have no relation to contextual variables, like in XSmith, and the grammar has more grammar rules, leading to more potential parsing decisions.

Where both cases are similar is that, though the result of the English meta-grammar was in structure more similar, it created rules that are not possible when applied to the English language. Examples such as Listing 5.7, where a preposition consists of nouns and verbs. This is not only restricted to the non-terminals but also includes attributes and the assignment of attributes, which can, similarly to WAGON, be randomised variables and don't reflect their related typing.

The only potential exception is GPT, since it is not restricted by the grammar, but uses the WAG samples as input, where it can try to retrieve context from. However, since it is not restricted, it can also generate completely other parses. An example of that is the generated C code in Listing 5.4.

The last question from the research question to answer is:

> *Are the results of generation applicable to grammarware, and if not, what can be done about it?*

Regrettably, based on the samples that have been generated, the short answer is **no**, the generated results do not apply to grammarware. The reason is that the generated results are missing semantic significance. Using the definitions from the research section on semantics, it can be concluded that we can generate references, be they predefined(simple English) or dynamically generated(WAGON). However, either the Sense component is missing in WAGON, where the generated non-terminals and attributes are random strings. Or generated simple English grammar that leads to the wrong application of sense, in that each reference has a sense, but the rules don't logically follow. An example previously mentioned is the preposition generation in Listing 5.7.

The second part of the question on whether something can be done about it can be answered with: "Potentially.". One of the results of generation is that Transformers, such as GPT, were able to interpret the concept of each of the meta-grammars by using similar grammar samples, like the Pokémon grammar, to infer potentially related types. In the related section, there was a technique called grammar-constrained decoding, which combines a Transformer and the restriction of grammar to generate results. The difference is that the results are generated in language; however, like this thesis, the use of meta-grammar to generate grammar can be similarly explored.

7.1 Discussion

The thesis went through some iterative changes, which included different formulated research questions. The initial thesis was to use the generated samples to train a neural network to infer the Weighted variables from each of the grammar rules in WAG. This would mean that, based on the weight of each neuron, we can potentially infer the probability of the grammar rule, instead of using fixed probabilities. This would be performed by training a CNNs or ResNet to classify the grammar and, based on the weight of each edge from neuron to neuron, to predict the probability that a grammar rule is chosen. However, since generating the specific grammar from meta-grammars was hard enough, it was decided to change the scope to focus on the grammar generation part itself.

The question then remains: Do the generated results potentially aid in making this kind of network? The answer, like grammarware, is no, since the generated results don't reflect any actual usage of grammar. However, if in future research it is indeed the case that semantics are taken into account during generation, this topic might also be good to revisit.

Lastly, there were some issues when using WAGON in ANTLR, with some parsing issues being found. The WAGON implementation of WAG itself was taken since it was simpler to use an already-defined grammar as a baseline. However, during implementation, there were some flaws in the specification defined in the paper [27], which slowed the thesis down. Furthermore, some of the different types, such as EBNF types, were not particularly useful to generate WAG samples, since only WAGON uses this definition. It might, therefore, have been a better choice to further simplify the grammar and reduce the grammar to a basic implementation, which still conforms to the WAG tuple.

7.2 Future work

As for future work, the first step would be to research a grammar generational algorithm similar to grammar-constrained decoding. This can be a similar version but catering to the generation of grammar from meta-grammar, instead of language from grammar. Otherwise, a combination of Transformer Networks and either Grammarinator or Evolutionary Grammar, like Grape, would also be interesting to research further. Of the two, Evolutionary Grammar is the most interesting since it is a bit faster than Grammarinator, and could also apply Probabilistic Grammar Evolution to specify preferential generation.

Appendix A

WAG Syntax.

A.1 Syntax Highlighter.

This appendix contains the code used to add syntax highlighting for WAG to Visual Studio. This is done using a tool called Yeoman and two JSON files seen below.

A.1.1 Language configuration file(*language-configuration.json*)

```
{
    "brackets": [
        ["'", "'"]
    ],
    "autoClosingPairs": [
        ["{", "}"], ["[", "]"], ["(", ")"], ["\"", "\""],
        ["'", "'"], ["<", ">"], ["==", "=="], ["/*", "*/"]
    ],
    "surroundingPairs": [
        ["{", "}"], ["[", "]"], ["(", ")"], ["\"", "\""],
        ["'", "'"], ["<", ">"], ["=", "="], ["/*", "*/"]
    ]
}
```

Listing A.1: WAG Highlighter language configuration.

A.1.2 TextMate configuration file.(*wag.tmLanguage.json)*

```
{
    "$schema": "https://raw.githubusercontent.com/martinring/
       tmlanguage/master/tmlanguage.json",
    "name": "Weighted Attribute Grammar",
    "patterns": [
  { "include": "#keywords" },
  { "include": "#strings" },
  { "include": "#variables" },
  { "include": "#comments" },
  { "include": "#relations" }
],
"repository": {
  "keywords": {
    "patterns": [
```

```
      {
        "name": "keyword.control.compare",
        "match": "\\b(if|else|then|in)\\b"
      },
      {
        "name": "keyword.control.bool",
        "match": "true|false"
      }
    ]
  },
  "strings": {
    "name": "string.quoted.double.wag",
    "begin": "\"",
    "end": "\"",
    "patterns": [
      {
        "name": "constant.character.escape.wag",
        "match": "\\\\."
      }
    ]
  },
  "variables": {
    "patterns" : [
      {
        "name": "variable.language",
        "match": "[&$*][a-zA-Z_]*"
      }
    ]
  },
  "comments": {
    "begin": "/\\*",
    "end": "\\*/",
    "patterns" : [
      {
        "name": "comment.multiline",
        "match": "[a-zA-z0-9]"
      }
    ]
  },
  "relations": {
    "patterns": [
      {
        "name": "keyword.relation.arrow",
        "match": "->"
      }
    ]
  }
    },
    "scopeName": "source.wag"
}
```

Listing A.2: WAG Highlighter TextMate language configuration.

A.2 WAG Attribute Parses

This section shows two examples of both synthesised and inherited attribute parses in both Attribute Grammar and Weighted Attribute Grammar.

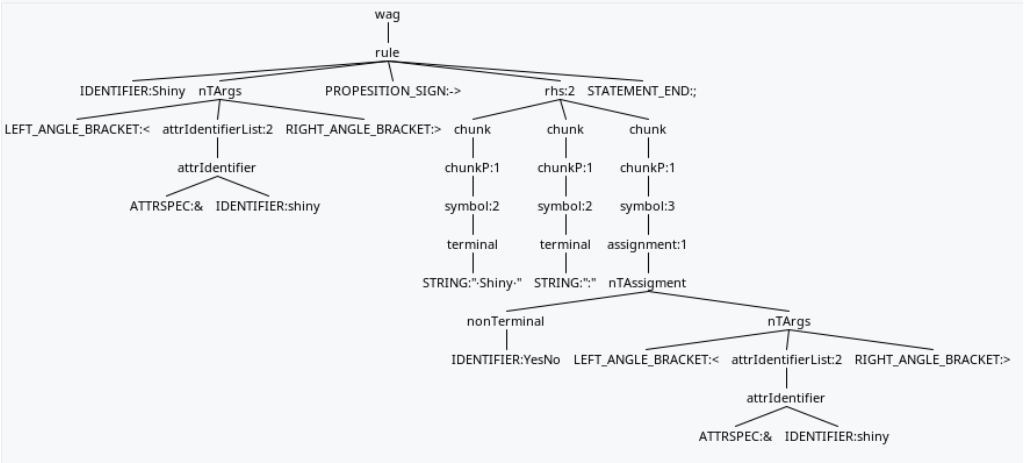


Figure A.1: Wag Synthesised Attributes example

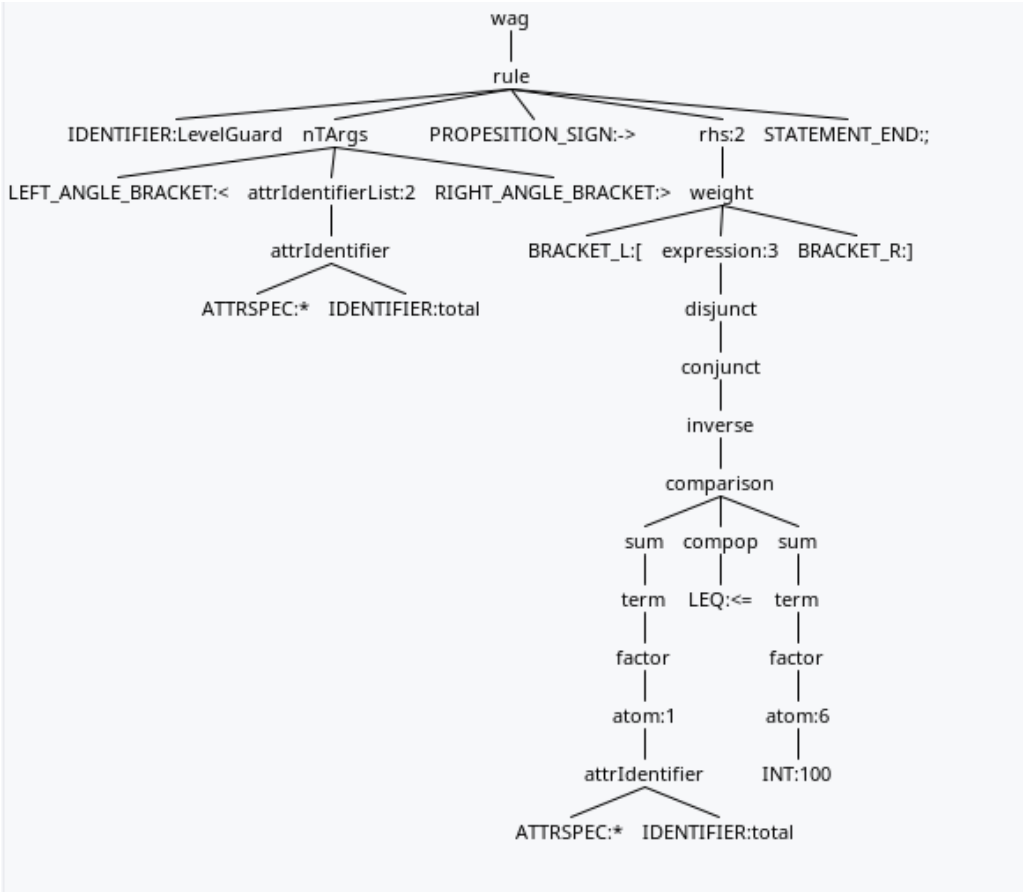


Figure A.2: Wag Inherited Attributes example

A.3 Grammarinator

A.3.1 Grammarinator Structure

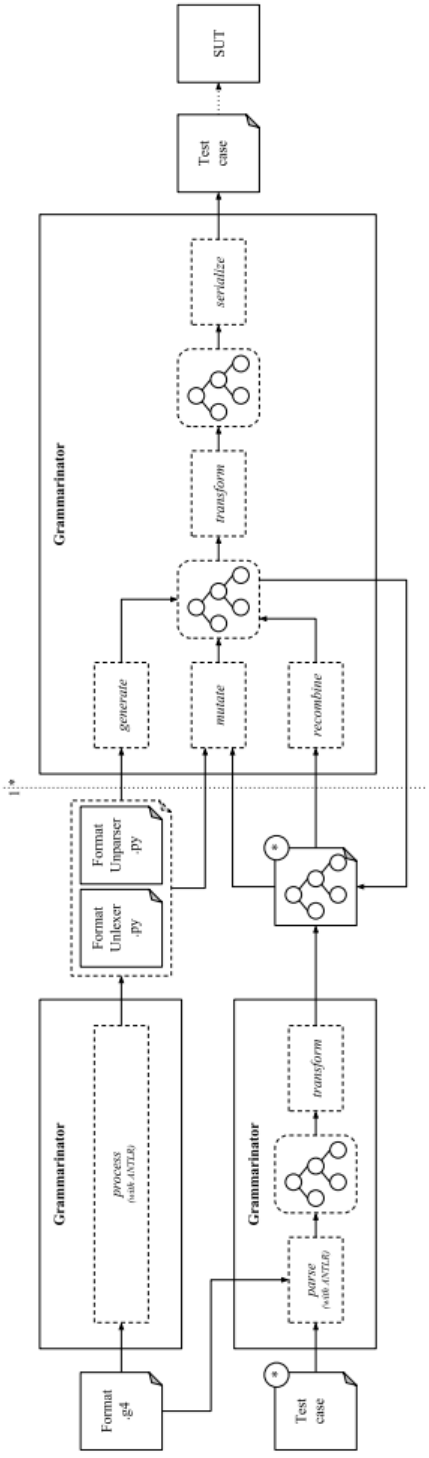


Figure A.3: Grammarinator overview.

A.3.2 Grammarinator Parses

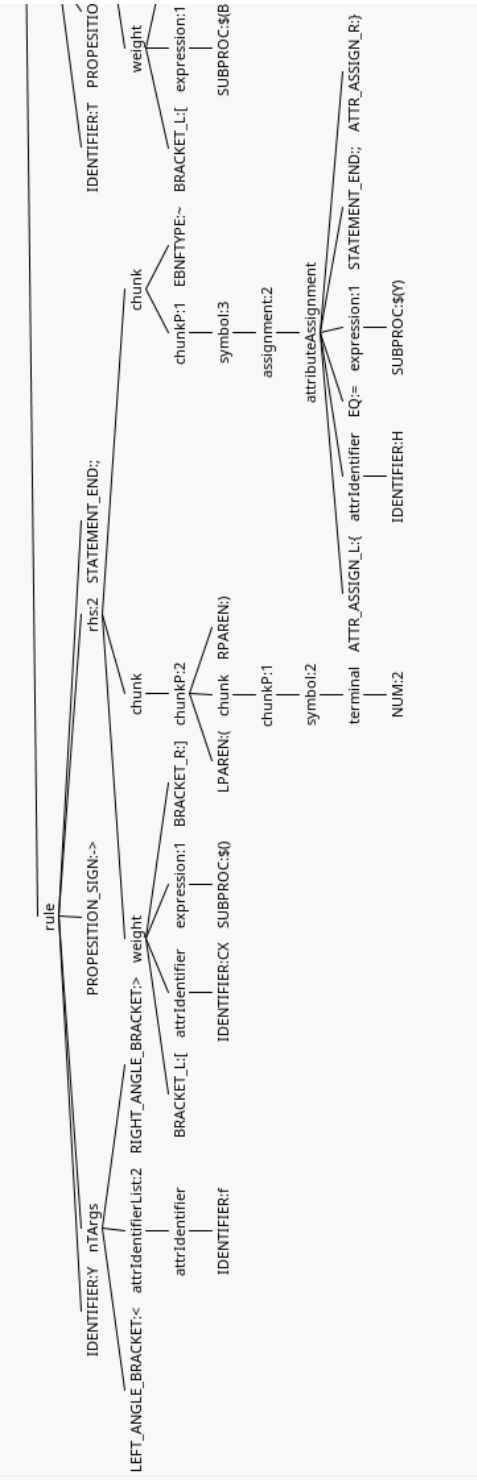


Figure A.4: First grammar rule parse Grammarinator.

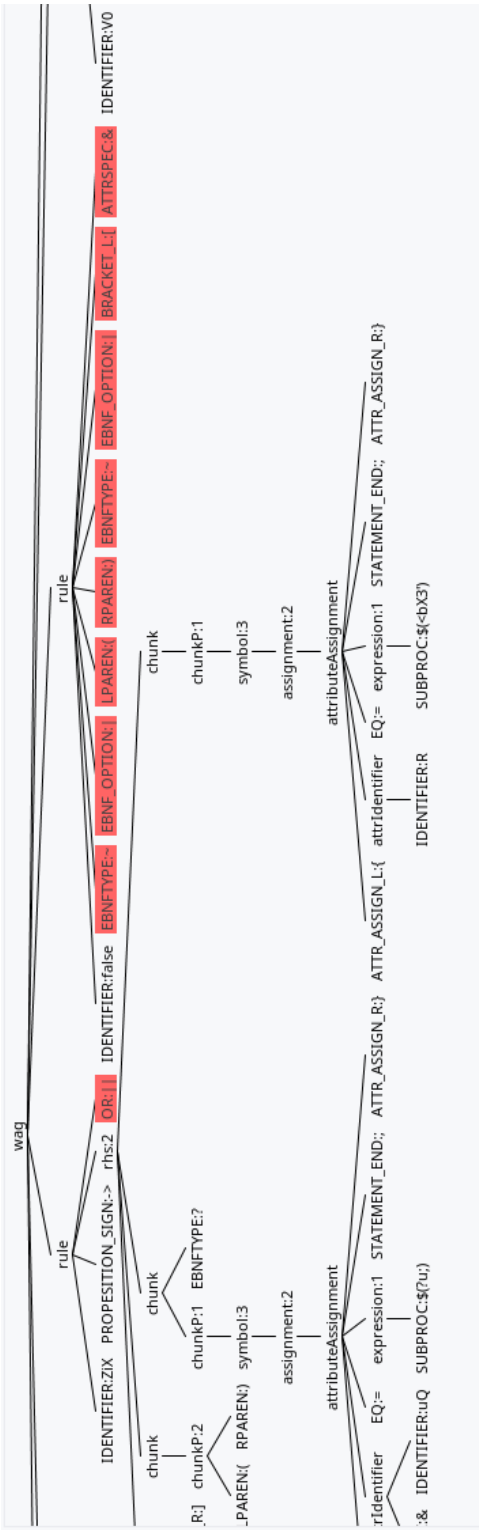


Figure A.5: Second grammar rule parse Grammarinator.

A.4 Transformers

A.4.1 GPT Parsing Results

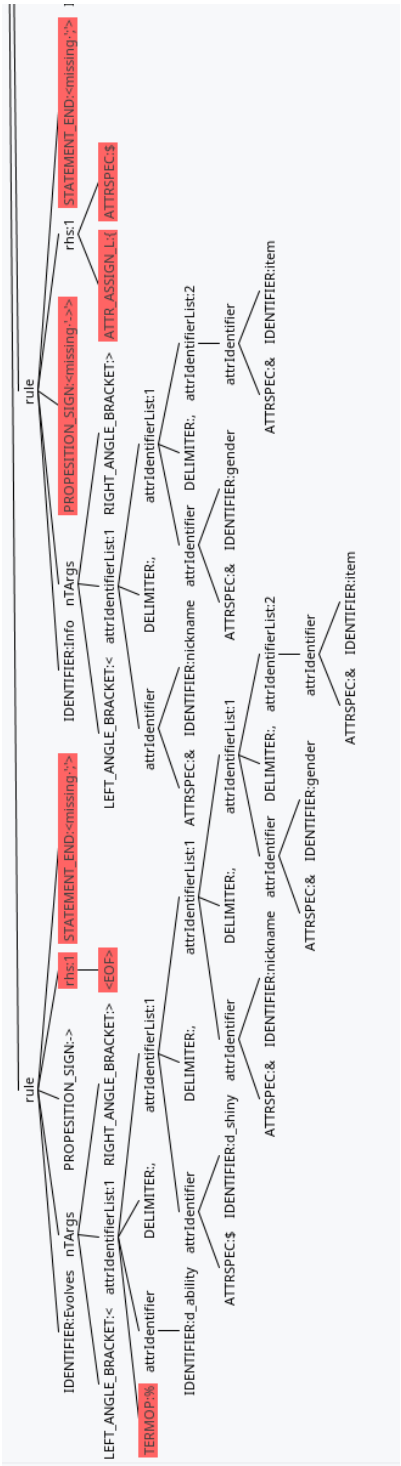


Figure A.6: GPT example of a partially parsable sample.

A.5 XSmith

This section shows all XSmith-related results, the first an example of the XSmith hole filling architecture, the second a large generated sample of XSmith.

A.5.1 XSmith Hole Filling Example

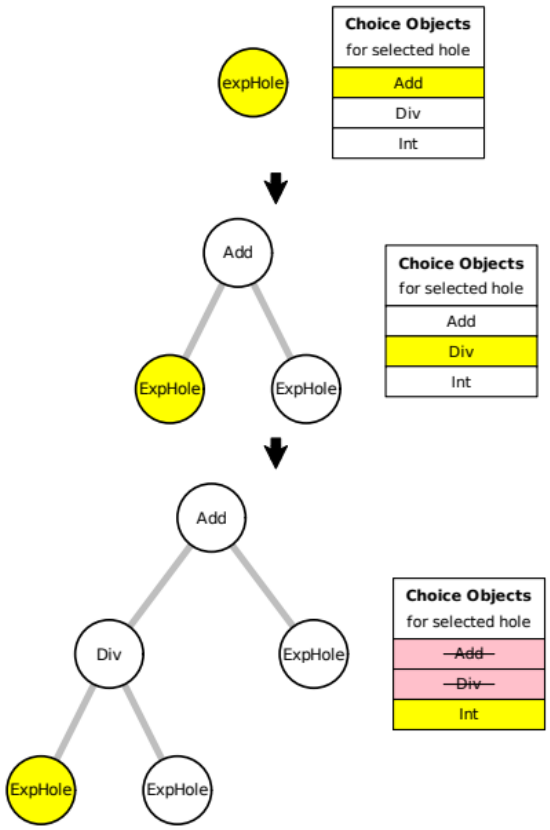


Figure A.7: The process of generating samples through 'holes' [109]

A.5.2 XSmith Larger Sample

```
1  // This is a RANDOMLY GENERATED PROGRAM.
2  // Fuzzer: simple-wag
3  // Version: simple-wag 2.0.7 (ca130f9), xsmith 2.0.7 (ca130f9), in
       Racket 8.10 (vm-type chez-scheme)
4  // Options: --output-file test.txt
5  // Seed: 1893289230
6
7  ****
8  Start of the generated WAG Code.
9  ****
10
11
12 lift_198 ->  1116391911;
```

```
lift_197 ->  ({b: ({g: lift_198})});
lift_192 ->  [];
lift_191 ->  lift_192;
lift_190 ->  lift_191;
lift_189 ->  lift_190;
lift_188 ->  true;
lift_187 ->  false;
lift_186 ->  [true, lift_187, true, lift_188, lift_187];
lift_185 ->  lift_186;
lift_184 ->  [lift_185, lift_189, lift_186, lift_186];
lift_182 ->  182656372;
lift_181 ->  ({a: lift_182});
lift_180 ->  lift_181;
lift_175 ->  false;
lift_174 ->  ({a: lift_175});
lift_173 ->  ({d: lift_174});
lift_172 ->  lift_173.d;
lift_171 ->  1340383588;
lift_170 ->  126865899;
lift_169 ->  1976489753;
lift_168 ->  ({e: [lift_169, 1738946014, lift_170, lift_170,
    lift_171]});
lift_167 ->  lift_168;
lift_165 ->  ({});
lift_164 ->  ({});
lift_163 ->  [({}), ({}), lift_164, ({}), lift_165];
lift_159 ->  1473710288;
lift_158 ->  1797468107;
lift_157 ->  [-335729611, lift_158, lift_158, lift_159];
lift_156 ->  [({e: lift_157})];
lift_155 ->  lift_156;
lift_150 ->  1673550862;
lift_149 ->  lift_150;
lift_148 ->  ({e: lift_149});
lift_147 ->  lift_148;
lift_146 ->  lift_147;
lift_145 ->  -1468587750;
lift_144 ->  lift_145;
lift_143 ->  621607329;
lift_142 ->  ({e: lift_143});
lift_141 ->  lift_142;
lift_140 ->  -359080922;
lift_139 ->  [({e: lift_140}), lift_141, lift_141, lift_141];
lift_138 ->  [lift_139, lift_139];
lift_137 ->  array_safe_reference(lift_138, lift_144, [lift_142,
    lift_146]);
lift_135 ->  -1235459317;
lift_134 ->  1717228235;
lift_130 ->  <arg_131, arg_132, arg_133>;
lift_129 ->  ({f: lift_130});
lift_128 ->  lift_129;
lift_122 ->  <arg_123, arg_124, arg_125, arg_126, arg_127>;
lift_113 ->  <arg_114, arg_115, arg_116>;
lift_112 ->  ({b: ({f: lift_113})});
lift_111 ->  lift_112;
lift_107 ->  <arg_108, arg_109, arg_110>;
```

```
lift_106 ->  ({f: lift_107});
lift_105 ->  lift_106;
lift_104 ->  [({b: lift_105}), lift_111];
lift_98 ->  <arg_99, arg_100, arg_101, arg_102, arg_103>;
lift_97 ->  lift_98;
lift_96 ->  lift_97;
lift_93 ->  -488210641;
lift_92 ->  ({e: lift_93});
lift_89 ->  false;
lift_86 ->  <arg_87>;
lift_85 ->  1542866807;
lift_81 ->  <>;
lift_80 ->  lift_81;
lift_79 ->  ({e: lift_80});
lift_76 ->  "V_6mC7FOm^~?$bjwJ'+|o k1V/KSI]3k4";
lift_75 ->  true;
lift_73 ->  <>;
lift_71 ->  "8puNcLWUa=fP5?c+uUPdW[OO}&oBM)_A'c>QZg`v>%^@U'VV6TPrw1
   -cj9Ex2J|/;+})'a.%Xr_u?(M @y]6=LsBvL";
lift_66 ->  854675232;
lift_63 ->  <arg_64, arg_65>;
lift_60 ->  "JGlneD{jCS6dHC%l\"dk)`yG8C*y)NZOO[AW1JQ<d7Vz?PnfrZ3v,r
   .;6di y ><9mfyw;DWI#Lo";
lift_58 ->  true;
lift_57 ->  false;
lift_56 ->  [lift_57, false, lift_58, lift_58, false];
lift_55 ->  [];
lift_51 ->  <arg_52, arg_53, arg_54>;
lift_39 ->  ({d: "bT*/x-v=<H?hA416]1#90'9y&Pl3:b<y)G9C8eKRzYAzP-g2E
   %,U~>J @H#fs[!^K?Jg^lI-.gRop  ?YWVM}* '"});
lift_38 ->  -1740359514;
lift_33 ->  <arg_34, arg_35, arg_36>;
lift_32 ->  lift_33;
lift_31 ->  lift_32;
lift_30 ->  lift_31;
lift_29 ->  lift_30;
lift_28 ->  lift_29;
lift_27 ->  ({e: lift_28});
lift_26 ->  lift_27;
lift_25 ->  lift_26;
lift_24 ->  ({a: lift_25});
lift_23 ->  -1543044424;
lift_19 ->  <arg_20, arg_21, arg_22>;
lift_18 ->  lift_19;
lift_17 ->  ({e: lift_18});
lift_15 ->  ({});
lift_14 ->  lift_15;
lift_10 ->  <arg_11, arg_12, arg_13>;
lift_9 ->  lift_10;
lift_8 ->  ({e: lift_9});
lift_7 ->  [lift_8, lift_8, lift_17];
lift_6 ->  21492682;
lift_5 ->  -84863781;
lift_4 ->  [lift_5, lift_6, lift_6, lift_6, -363495177];
lift_3 ->  [lift_4, lift_4, lift_4, lift_4];
lift_2 ->  ({c: lift_3});
```

```
lift_1 ->  lift_2.c;
{
    lift_183 ->  lift_184;
    lift_166 ->  ({b: lift_167});
    lift_154 ->  [lift_155];
    lift_151 ->  [];
    lift_121 ->  lift_122;
    lift_120 ->  ["e54_0)[7h2*ngCVeV)cQxE}S iF#Pn_4_C;sR4 TE`Hm!*`1
       iD9_W b*^$IznN}o|~z(O", ".ekp{=Sr_[YS\\cj~gZ<=F`y%m*
       YKojyGJGVL)v`YS?#^F<T;>8Xnq*%*#}$4[", lift_71];
    lift_119 ->  lift_120;
    lift_118 ->  lift_57;
    lift_117 ->  [lift_23, lift_6, lift_66, lift_66, lift_38];
    lift_95 ->  array_safe_reference(lift_96(lift_117, lift_56, [
       true, lift_58, lift_118], lift_119, lift_85), lift_66,
       lift_121(lift_93, lift_5, ({}), lift_56, ({}))).b;
    lift_72 ->  lift_73;
    lift_70 ->  ({});
    lift_69 ->  <>;
    lift_68 ->  [lift_69, lift_69, lift_72, lift_72];
    lift_62 ->  lift_63;
    lift_61 ->  lift_62;
    lift_59 ->  ({c: lift_60});
    lift_50 ->  lift_51;
    lift_49 ->  "| 7K-qZUuOR0gKAnS;fg=YGR<&w4G?hybse7ZI^QV\\a\\N1@
       \\uFJOT-Ns(w^k1&.~6Rw}#rg%,HF5} Ue:UsR";
    lift_48 ->  true;
    lift_40 ->  <arg_41, arg_42, arg_43, arg_44, arg_45>;
    array_safe_reference(array_safe_reference(lift_1,
       array_safe_reference(lift_7, lift_5, lift_24.a).e(lift_39.d,
        array_safe_reference(lift_40(lift_23, lift_56, lift_15,
       ({}), lift_57)(lift_59.c, 488171448, lift_6), string-length
       ({f: lift_60}).f, lift_33(safe_string_append(lift_60,
       lift_60), array_safe_reference(lift_55, 489563983,
       -33008271), lift_39.d)), array_safe_reference(lift_61(
       lift_57, lift_15), lift_38, array_safe_reference(lift_68,
       lift_6, lift_72))()), lift_79.e()), lift_85, lift_86(
       array_safe_reference(array_safe_reference(
       array_safe_reference([], lift_38, [lift_3]), lift_38, lift_3
       ), lift_5, array_safe_reference(<arg_94>(lift_71), lift_19(
       lift_60, lift_38, "qS.H%Q:C>+g?fsk 5c~P=!wg$sjbZCNV;krxP#
       oI1hN7`Vyxsy-/A;A["), lift_55))).e);
    lift_25.e = lift_95.f;
    b_136 = array_safe_reference(lift_137, lift_6,
       array_safe_reference(array_safe_reference(lift_151,
       -1344856040, lift_137), lift_146.e, lift_146)).e
}
```

Listing A.3: Larger XSmith sample of generated grammar.

A.6 Evolutionary Grammars

An overview of some parsed samples of complete, partially complete, and non-parsable samples.

A.6.1 Grape parses

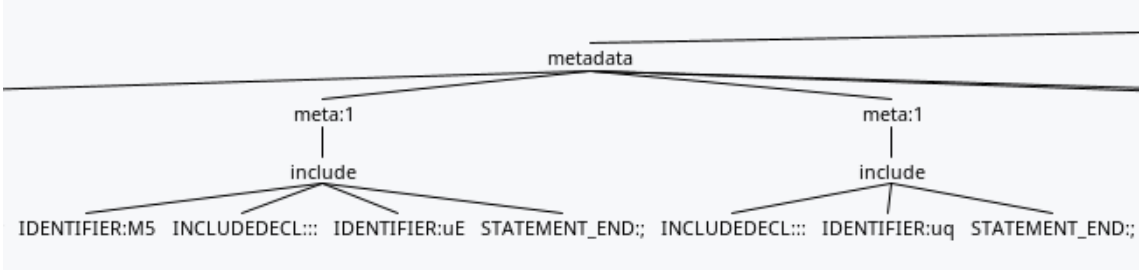


Figure A.8: Complete Grape parse.

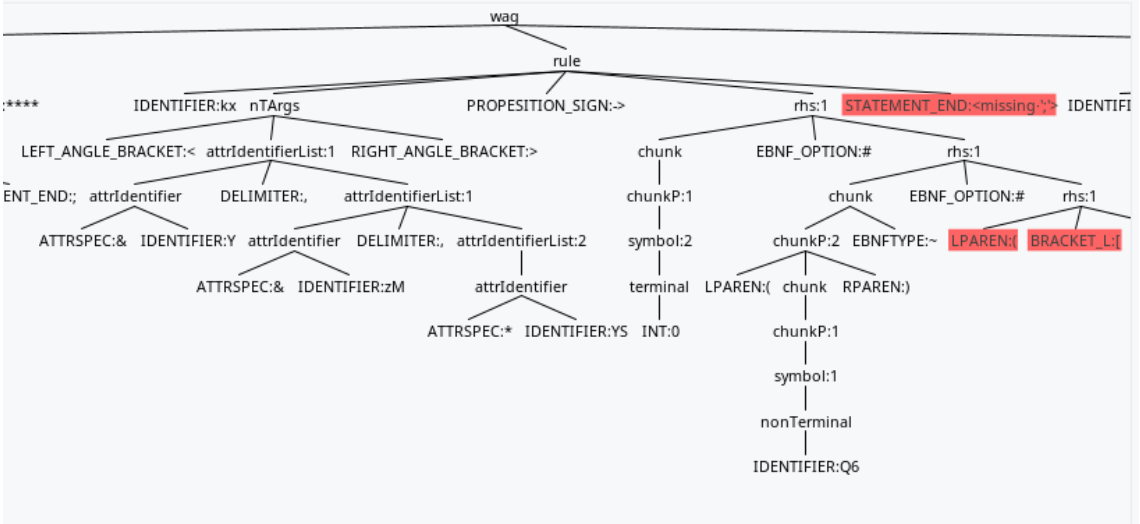


Figure A.9: Mostly Complete Grape parse.

Figure A.10: Parsing issue of Grape.

Bibliography

[1] *ISO/IEC 14977 : 1996(E)*, 1996. URL: https://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf.

[2] Tim Salimans Alec Radford, Karthik Narasimhan and Ilya Sutskever. Improving language understanding by generative pre-training. In *OpenAI*, page 12. OpenAI, June 2019.

[3] Peter J. Angeline. Genetic programming: On the programming of computers by means of natural selection,. *Biosystems*, 33, 1994. doi:10.1016/0303-2647(94)90062-0.

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016. arXiv:1607.06450.

[5] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger, and W. L. van der Poel. Revised report on the algorithmic language algol 60. *Numerische Mathematik*, 4, 1962. doi:10.1007/BF01386340.

[6] Ekagrata Bahadur. Analysis of genetic algorithms in natural language processing, 02 2024. doi:10.21203/rs.3.rs-3969717/v1.

[7] J E Baker. Adaptive selection methods for genetic algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, 1985.

[8] David Bell. On the translation of frege's bedeutung. *Analysis (United Kingdom)*, 40, 1980. doi:10.1093/analys/40.4.191.

[9] Yoshua Bengio. Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, 2, 2009. doi:10.1561/2200000006.

[10] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35, 2013. doi:10.1109/TPAMI.2013.50.

[11] Leon Bottou. Online algorithms and stochastic approximation, 1998.

[12] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 2020-December, 2020.

[13] Christoph Brune. Deep reinforcement learning. https://canvas.utwente.nl/courses/11534/files/3416859?module_item_id=367044, 2023. Accessed: 29/04/24, UTwente.

[14] Christof Bürger. Racr: A scheme library for reference attribute grammar controlled rewriting, 2012.

[15] Murray Campbell, A. Joseph Hoane, and Feng Hsiung Hsu. Deep blue. *Artificial Intelligence*, 134, 2002. doi:10.1016/S0004-3702(01)00129-1.

[16] Florin Capitanescu, Antonino Marvuglia, Enrico Benetto, Aras Ahmadi, and Ligia Tiruta-Barna. Assessing the uses of nlp-based surrogate models for solving expensive multi-objective optimization problems: Application to potable water chains. In *Proceedings of EnviroInfo and ICT for Sustainability 2015*, volume 22, 2015. doi:10.2991/ict4s-env-15.2015.2.

[17] David Cassel. Cobol is everywhere. who will maintain it, 2017. URL: https://thenewstack.io/cobol-everywhere-will-maintain/.

[18] Ned Chapin, Joanne E. Hale, Khaled M. Khan, Juan F. Ramil, and Wui Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution*, 13, 2001. doi:10.1002/smr.220.

[19] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, 2014. doi:10.3115/v1/d14-1179.

[20] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956. doi:10.1109/TIT.1956.1056813.

[21] N. Chomsky and M. P. Schützenberger. The algebraic theory of context-free languages. *Studies in Logic and the Foundations of Mathematics*, 26:118–161, 1 1959. doi:10.1016/S0049-237X(09)70104-1.

[22] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2, 1959. doi:10.1016/S0019-9958(59)90362-6.

[23] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12, 2011.

[24] Dave Crocker and Paul Overell. Augmented BNF for Syntax Specifications: ABNF. RFC 5234, January 2008. URL: https://www.rfc-editor.org/info/rfc5234, doi:10.17487/RFC5234.

[25] Charles Darwin. *On the Origin of Species by Means of Natural Selection*. Murray, London, 1859. or the Preservation of Favored Races in the Struggle for Life.

[26] Jacob Devlin, Ming Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, volume 1, 2019.

[27] R.M. Dulfer. Wagon : A weighted attribute grammar oriented notation, June 2024. URL: http://essay.utwente.nl/99790/.

[28] C. A. Ellis. *Probabilistic Languages and Automata*. PhD thesis, University of Illinois, 1970.

[29] Dumitru Erhan, Aaron Courville, Yoshua Bengio, and Pascal Vincent. Why does unsupervised pre-training help deep learning? In *Journal of Machine Learning Research*, volume 9, 2010.

[30] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay Mccarthy, and Sam Tobin-Hochstadt. The racket manifesto. In *Leibniz International Proceedings in Informatics, LIPIcs*, volume 32, 2015. doi:10.4230/LIPIcs.SNAPL.2015.113.

[31] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. Grammar-constrained decoding for structured nlp tasks without finetuning, 2024. URL: https://arxiv.org/abs/2305.13971, arXiv:2305.13971.

[32] Edward Grefenstette, Phil Blunsom, Nando de Freitas, and Karl Moritz Hermann. A deep architecture for semantic parsing. *CoRR*, 2015. doi:10.3115/v1/w14-2405.

[33] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques*. Springer New York, jan. 2008. doi:10.1007/978-0-387-68954-8.

[34] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J. H. Jacobs, and Koen G. Langendoen. *Modern Compiler Design*. Addison-Wesley, second edition, 2012. URL: https://dickgrune.com/Books/MCD_2nd_Edition/.

[35] Manish Gupta and Puneet Agrawal. Compression of deep learning models for text: A survey. *ACM Trans. Knowl. Discov. Data*, 16(4), jan 2022. doi:10.1145/3487045.

[36] William Hatch, Pierce Darragh, Sorawee Porncharoenwase, Guy Watson, and Eric Eide. Generating conforming programs with xsmith. In *GPCE 2023 - Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, Co-located with: SPLASH 2023*, 2023. doi:10.1145/3624007.3624056.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL: http://arxiv.org/abs/1512.03385, arXiv:1512.03385.

[38] Pedro Rarigcl Henriques, Tomaž Kosar, Marjan Mernik, Maria Joao Varaiida Pereira, and Viljcm Zumer. Grammatical approach to problem solving. *Proceedings of the International Conference on Information Technology Interfaces, ITI*, 2003. doi:10.1109/ITI.2003.1225416.

[39] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313, 2006. doi:10.1126/science.1127647.

[40] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9, 1997. doi:10.1162/neco.1997.9.8.1735.

[41] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, page 45–48, New

York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3278186.3278193.

[42] John H. Holland. *Adaptation in Natural and Artificial Systems*. The MIT Press, 1992. doi:10.7551/mitpress/1090.001.0001.

[43] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In *ACL 2018 - 56th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference (Long Papers)*, volume 1, 2018. doi:10.18653/v1/p18-1031.

[44] Feargus Illingworth et al. Mainframe modernization business barometer report. Technical report, Technical Report. Advanced. https://modernsystems.oneadvanced. com/en ..., 2022. URL: https://modernsystems.oneadvanced.com/globalassets/modern-systems-assets/resources/reports/2022-mainframe-modernization-business-barometer-report.pdf.

[45] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *32nd International Conference on Machine Learning, ICML 2015*, 1, 2015.

[46] Shruti Jado. Introduction to different activation functions for deep learning, March 2018. URL: https://medium.com/@shrutijadon/survey-on-activation-functions-for-deep-learning-9689331ba092.

[47] Christian Janiesch, Patrick Zschech, and Kai Heinrich. Machine learning and deep learning. *Electronic Markets*, 31, 2021. doi:10.1007/s12525-021-00475-2.

[48] Khalid Jebari and Mohammed Madiafi. Selection methods for genetic algorithms. *International Journal of Emerging Sciences*, 3:333–344, 2013.

[49] Yangqing Jia, Thomas K. Leung, Alexander Toshev, and Sergey Ioffe Yunchao Gong. Deep convolutional ranking for multilabel image annotation. In *2nd International Conference on Learning Representations*, pages 1–9, 4 2014.

[50] Deepali J. Joshi, Ishaan Kale, Sadanand Gandewar, Omkar Korate, Divya Patwari, and Shivkumar Patil. Reinforcement learning: A survey. *Advances in Intelligent Systems and Computing*, 1311 AISC, 2021. doi:10.1007/978-981-33-4859-2_29.

[51] Daniel Jurafsky and James Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, volume 2. Prentice Hall, 02 2008.

[52] Divakar Kapil. Stochastic vs batch gradient descent, Januari 2019. URL: https://medium.com/@divakar_239/stochastic-vs-batch-gradient-descent-8820568eada1.

[53] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, 80, 2021. doi:10.1007/s11042-020-10139-6.

[54] Hyun-Tae Kim and Chang Wook Ahn. A new grammatical evolution based on probabilistic context-free grammar. In Hisashi Handa, Hisao Ishibuchi, Yew-Soon Ong, and Kay-Chen Tan, editors, *Proceedings of the 18th Asia Pacific Symposium on Intelligent and Evolutionary Systems - Volume 2*, pages 1–12, Cham, 2015. Springer International Publishing.

[55] Yoon Kim. Convolutional neural networks for sentence classification. *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, 2014. doi:10.3115/v1/d14-1181.

[56] Diederik P. Kingma and Max Welling. An introduction to variational autoencoders, 2019. doi:10.1561/2200000056.

[57] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, July 2005. doi:10.1145/1072997.1073000.

[58] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2, 1968. doi:10.1007/BF01692511.

[59] John R. Koza, Forrest H. Bennett, David Andre, and Martin A. Keane. *Automated Design of Both the Topology and Sizing of Analog Electrical Circuits Using Genetic Programming*, pages 151–170. Springer Netherlands, Dordrecht, 1996. doi:10.1007/978-94-009-0279-4_9.

[60] A Krizhevsky, V Nair, and G Hinton. Cifar-10 and cifar-100 datasets, 2009.

[61] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 2, 2012.

[62] David B. Kronenfeld, Roger C. Schank, and Robert P. Abelson. Scripts, plans, goals, and understanding: An inquiry into human knowledge structures. *Language*, 54, 1978. doi:10.2307/412850.

[63] Robin Kurtz, Daniel Roxbo, and Marco Kuhlmann. Improving semantic dependency parsing with syntactic features. *Proceedings of the First NLPL Workshop on Deep Learning for Natural Language Processing*, 2019.

[64] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning, 5 2015. doi:10.1038/nature14539.

[65] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018. doi:10.1145/3238147.3238176.

[66] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2020. doi:10.18653/v1/2020.acl-main.703.

[67] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: Analysis, applications, and prospects. *IEEE Transactions on Neural Networks and Learning Systems*, 33, 2022. doi:10.1109/TNNLS.2021.3084827.

[68] Shengfei Lyu and Jiaqi Liu. Convolutional recurrent neural networks for text classification. *Journal of Database Management*, 32, 2021. doi:10.4018/JDM.2021100105.

[69] John McCarthy, Marvin Minsky, Nathaniel Rochester, and Claude E. Shannon. A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI Mag.*, 27:12–14, 2006. URL: https://api.semanticscholar.org/CorpusID:1943Javed20059915.

[70] John Mcdowell. On the sense and reference. *Mind*, 86, 1977. doi:10.1093/mind/LXXXVI.342.159.

[71] David McNeely-White, J. Ross Beveridge, and Bruce A. Draper. Inception and resnet features are (almost) equivalent. *Cognitive Systems Research*, 59, 2020. doi:10.1016/j.cogsys.2019.10.004.

[72] Zbigniew Michalewicz and Marc Schoenauer. Evolutionary algorithms for constrained parameter optimization problems. *Evolutionary Computation*, 4(1):1–32, 1996. doi:10.1162/evco.1996.4.1.1.

[73] Melanie Mitchell. L.d. davis, handbook of genetic algorithms. *Artificial Intelligence*, 100, 1998. doi:10.1016/s0004-3702(98)00016-2.

[74] Jessica Mégane, Nuno Lourenço, and Penousal Machado. Probabilistic grammatical evolution. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 12691 LNCS, 2021. doi:10.1007/978-3-030-72812-0_13.

[75] Jessica Mégane, Nuno Lourenço, and Penousal MacHado. Co-evolutionary probabilistic structured grammatical evolution. In *GECCO 2022 - Proceedings of the 2022 Genetic and Evolutionary Computation Conference*, 2022. doi:10.1145/3512290.3528833.

[76] Axi Niu, Kang Zhang, Chaoning Zhang, Chenshuang Zhang, In So Kweon, Chang D. Yoo, and Yanning Zhang. Fast adversarial training with noise augmentation: A unified perspective on randstart and gradalign, 2022. arXiv:2202.05488.

[77] The Editors of Encyclopaedia Britannica. "meaning", 10 2024. URL: https://www.britannica.com/topic/meaning.

[78] The Editors of Encyclopaedia Britannica. semantics, 1 2025. URL: https://www.britannica.com/topic/meaning.

[79] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permutation crossover operators on the traveling salesman problem. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, page 224–230, USA, 1987. L. Erlbaum Associates Inc.

[80] OpenAI. Introducing chatgpt, 2023. URL: https://openai.com/blog/.

[81] Kanghee Park, Timothy Zhou, and Loris D'Antoni. Flexible and efficient grammar-constrained decoding, 2025. URL: https://arxiv.org/abs/2502.05111, arXiv:2502.05111.

[82] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *NAACL HLT 2018 - 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, 1, 2018. doi:10.18653/v1/n18-1202.

[83] Eva Picardi and Annalisa Coliva. 133Über sinn und bedeutung: An elementary exposition. In *Frege on Language, Logic, and Psychology: Selected Essays*. Oxford University Press, 07 2022. arXiv:https://academic.oup.com/book/0/chapter/370291915/chapter-pdf/57893084/oso-9780198862796-chapter-6.pdf, doi:10.1093/oso/9780198862796.003.0006.

[84] Michael O. Rabin. Probabilistic automata. *Information and Control*, 6:230–245, 9 1963. doi:10.1016/S0019-9958(63)90290-0.

[85] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77, 1989. doi:10.1109/5.18626.

[86] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. In *Language Models are Unsupervised Multitask Learners*, 2019. URL: https://api.semanticscholar.org/CorpusID:160025533.

[87] Kazi Shah Nawaz Ripon, Nazmul Siddique, and Jim Torresen. Improved precedence preservation crossover for multi-objective job shop scheduling problem. *Evolving Systems*, 2:119–129, 06 2011. doi:10.1007/s12530-010-9022-x.

[88] Bertrand Russell. On denoting. *Mind*, 14(56):479–493, 1905. doi:10.1093/mind/xiv.4.479.

[89] Stuart Russell and Peter Norivg. Artificial intelligence: A modern approach (global edition). *Artificial Intelligence: A Modern Approach*, 2021.

[90] Conor Ryan, J. J. Collins, and Michael O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1391, 1998. doi:10.1007/BFb0055930.

[91] Sumit Saha. A guide to convolutional neural networks — the eli5 way, December 2018. URL: https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/.

[92] Arto Salomaa. Probabilistic and weighted grammars. *Information and Control*, 15:529–544, 12 1969. doi:10.1016/S0019-9958(69)90554-3.

[93] Ömer Sayilir. Towards grammatical inference of legacy programming languages, May 2024. URL: http://essay.utwente.nl/99151/.

[94] Jürgen Schmidhuber. Deep learning in neural networks: An overview, 2015. doi:10.1016/j.neunet.2014.09.003.

[95] Rituparna Sen and Sourish Das. *Unsupervised Learning*, page pp 305–318. Springer Singapore, 2023. doi:10.1007/978-981-19-2008-0_21.

[96] Dong Shen and Jian Xin Xu. An iterative learning control algorithm with gain adaptation for stochastic systems. *IEEE Transactions on Automatic Control*, 65, 2020. doi:10.1109/TAC.2019.2925495.

[97] L.D. Steenmeijer. Use weighted attribute grammars to formalize human-to-machine communication in internet of things systems, January 2025. URL: http://essay.utwente.nl/104891/.

[98] Andrew Stevenson and James R. Cordy. A survey of grammatical inference in software engineering. *Sci. Comput. Program.*, 96:444–459, 2014. URL: https://doi.org/10.1016/j.scico.2014.05.008, doi:10.1016/J.SCICO.2014.05.008.

[99] Gerald Jay Sussman and Guy Lewis Steele. The art of the interpreter of the modularity complex (parts zero, one, and two), 1978.

[100] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. *31st AAAI Conference on Artificial Intelligence, AAAI 2017*, 2017. doi:10.1609/aaai.v31i1.11231.

[101] Jiexiong Tang, Chenwei Deng, and Guang Bin Huang. Extreme learning machine for multilayer perceptron. *IEEE Transactions on Neural Networks and Learning Systems*, 27, 2016. doi:10.1109/TNNLS.2015.2424995.

[102] Leslie G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10(2):308–315, 1975. URL: https://www.sciencedirect.com/science/article/pii/S0022000075800468, doi:10.1016/S0022-0000(75)80046-8.

[103] Jesper E. van Engelen and Holger H. Hoos. A survey on semi-supervised learning. *Machine Learning*, 109, 2020. doi:10.1007/s10994-019-05855-6.

[104] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 2017-December, 2017.

[105] Gio Wiederhold and John McCarthy. Arthur samuel: Pioneer in machine learning. *IBM Journal of Research and Development*, 36, 2010. doi:10.1147/rd.363.0329.

[106] Nick Wolters. Wagenator, 2025. URL: https://github.com/NickWolters/wagenator.

[107] Zifeng Wu, Chunhua Shen, and Anton van den Hengel. Wider or deeper: Revisiting the resnet model for visual recognition. *Pattern Recognition*, 90:119–133, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0031320319300135, doi:10.1016/j.patcog.2019.01.006.

[108] Jinbo Xing, Menghan Xia, Yuxin Liu, Yuechen Zhang, Yong Zhang, Yingqing He, Hanyuan Liu, Haoxin Chen, Xiaodong Cun, Xintao Wang, Ying Shan, and Tien Tsin Wong. Make-your-video: Customized video generation using textual and structural guidance. *IEEE Transactions on Visualization and Computer Graphics*, 2024. doi:10.1109/TVCG.2024.3365804.

[109] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *ACM SIGPLAN Notices*, 47, 2012. doi:10.1145/2345156.1993532.

[110] Gokul Yenduri, Ramalingam M, Chemmalar Selvi G, Supriya Y, Gautam Srivastava, Praveen Kumar Reddy Maddikunta, Deepti Raj G, Rutvij H Jhaveri, Prabadevi B, Weizheng Wang, Athanasios V. Vasilakos, and Thippa Reddy Gadekallu. Generative pre-trained transformer: A comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions, 2023. URL: https://arxiv.org/abs/2305.10435, arXiv:2305.10435.

[111] Yeoman. Yeoman.io. URL: https://github.com/yeoman/yeoman.io.

[112] Vadim Zaytsev. *Recovery, Convergence, and Documentation of Languages*. Phd thesis, Vrije Universiteit, Amsterdam, The Netherlands, oct 2010.

[113] Vadim Zaytsev. Bnf was here: what have we done about the unnecessary diversity of notation for syntactic definitions. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, page 1910–1915, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2245276.2232090.

[114] Vadim Zaytsev. Speak Well or Be Still: Solving Conversational AI with Weighted Attribute Grammars. In Catherine Dubois and Julien Cohen, editors, *STAF 2022 Workshop Proceedings: Second International Workshop on MDE for Smart IoT Systems (MeSS)*, volume 3250 of *CEUR Workshop Proceedings*, pages 72–74, Enschede, Netherlands, 2022. CEUR-WS.org. URL: http://ceur-ws.org/Vol-3250/messpaper5.pdf.

[115] X. Zhang, S. Ren, J. Sun, and K. He. Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[116] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22, 2000. doi:10.1109/34.888718.