Hochschule für Musik und Darstellende Kunst
Frankfurt am Main

Institut für zeitgenössische Musik

Generative Grammars

Autor: Amir Teymuri

Diese Arbeit ist als Teil des Masterabschlusses im Fach Komposition entstanden und ist in englischer Sprache verfasst.

March 30, 2019

Contents

1 Outline

Generative Grammars are powerful tools for algorithmic music composition. Inspired from the pioneering works of Noam Chomsky in the late 1950s in the fields of linguistics, these generative and analytical concepts were soon introduced and adopted into music as a means for composition and analysis. This paper provides an overview about formal languages and some of their usages in algorithmic music. In the last section, a formal language is developed based on a comparison between two pieces which exhibit high structrual resemblance, namely *Spectral Canon* by James Tenney and *Falling Music* by Frederic Rzewski. Some of the examples in the text are accompanied by codes which are included in an appendix at the end of this paper. The codes are implemented in the Hy programming language[1], using it's music notation module Kodou[2].

[1]For more information see the documentation of the language under `http://hylang.org/`

[2]Kodou is a versatile software for algorithmic music notation developed and maintained by the author. More on Kodou under `https://kodou.readthedocs.io/`

2 Generative Grammars

Probably the most controversial claim of Noam Chomsky was that the aquisition of Language takes place through an innate human faculty. According to Chomsky making linguistic communication is as much of an instinctive human act as walking, due to specific cognitive faculties of our brains[3]. Linguists call this innate human facility for learning of languages the *Universal Grammar* (or UG)[4]. Chomsky's survey of language denoted that the study of structures of natural languages and their construction regularities would not only shed light on human languages, but also reveal some remarkable pecularities about human's thought pattern and his cognitive faculty. Chomsky's pioneering work in the field of linguistics soon was adopted for other scientific fields such as cognitive science, logic and computer science.

Study of languages begins with the study of their syntax. A syntax can be defined as a set of principles which regulate the structures of valid outputs producible in a given language. In human languages syntax is the study of the ways the order of words in a sentence affects it's meaning. There are two main contemporary views on the evolution of syntax in natural languages:

"an incremental view which claims that the evolution of syntax involved multiple stages between the communication system of our last common ancestor with chimpanzees and full-blown modern human syntax, and the saltational view which claims that syntax was the result of just a single evolutionary development" [5].

The theory of syntax is a sub-area of linguistics which investigates formal structures of compound sentences in any natural or formal languages[4]. A syntax is thus a constraint which takes care of conformity of the arrangment of generated sentences with the formalities of it's language. This definition indicates that re-applications of the same rules to the same constituent units would result in a (possibly new and) grammatically correct product in the language. The ultimate goal of syntactic theory is to decode and to model the procedures used as part of human's cognitive abilities to produce valid sentences. The study of these subconciouse set of procedures in our brains is a case of particular interest in treating generative grammars in linguistics [4].

Definition of rules as a means for describing and modeling these procedures which are able of producing valid sentences of a specific language are called *generative gram-*

[3]As a secondry object: this assumes a human language as a natural phenomenon, yet it is obvious that language and it's rules of construction for conveying diverse forms of information has evolved and developed through generations and is an artifact of a human instinct and necessity for communication. A child never exposed to some sort of complex communication community will probably have hard times learning any natural languages, whereas someone grown up bilingually will most probably find it easy in his later life to *learn* other systems of communications and/or languages. Thus contrary to an instinctive human act like walking, training is a substantial part of natural language acquisition. Chomsky's views refer to the necessary and proven preconditions of human brain for acquisition of natural and formal languages.

[4]Sentences or more generally expressions are in this context strings of symbols whose formations comply with certain rules.

mars. Informally a generative grammar is a *recursive* rule system for the definition of a language that is capable of producing well-formed expressions in a given context[2]. According to Chomsky:

"*a grammar is based on a finite number of observed sentences (the linguist's corpus) and it 'projects' this set to an infinite set of grammatical sentences by stablishing general 'laws' (grammatical rules) framed in terms of [such] hypothetical constructs as the particular phonemes, words, phrases and so on, of the language under analysis.*"[6]

He defines Language and Grammar as:

"*By a language, [...] we shall mean a set (finite or infinite) of sentences, each of finite length, all constructed from a finite alphabet of symbols. If A is an alphabet, we shall say that anything formed by concatenating the symbols of A is a string in A. By a grammar of the language L we mean a device of some sort that produces all of the strings that are sentences of L and only these.*"[6]

Formally a formal grammar is a tuple consisting of four elements: $G = \{S, P, T, N\}$ where $S =$ start symbol (or a set of initial states), $P =$ a set of production rules, $T =$ a set of terminal smybols and $N =$ a set of non-terminal symbols. A valid and well-formed sentence in a language is a string of symbols in T driven from S by application of one or more rules in P. In terms of notation, the generation of new sentences[5] occures by replacing the symbols on the left-hand side of some rewriting rules in P with symbols on their right-hand side[2].

A formal language L over an alphabet Σ is a subset of Σ^*[6] created by means of a formal grammar G and thus:

$$L(G) \subseteq \Sigma^*$$

At the heart of any formal language lies a set of syntactic rules that specify the set of all and only those strings of symbols that constitute legal or *well-formed* expressions in the language [1]. An essential critertion regarding sequences within a formal language is their *well-formedness*, signifying their correctness in terms of the syntactic rules - this, however, does not automatically imply that these sentences are semantically accurate, i.e. meaningful [2] as will be briefly demonstrated in the next section.

For the generation as well as checking for correctness of output expressions of a given generative grammar, Chomsky introduced four different types of grammars, known as *Chomsky Hierarchy.* Each of these four types of grammar generate formal languages and correspond to a type of automata which can test the membership of any symbol strings to the language in question. Named with their orders *Type-0, Type-1, Type-2* and *Type-3*, higher order types of grammars will carry out more restrictions on

[5] Or more generally sequences

[6] Kleene star notation. It was first introduced by Stephen Kleene in the context of regular expressions where it is understood as a certain automata meaning "zero or more". In the context of grammars it refers to the infinite set of all possible finite-length string concatenations Σ^* over the symbols of the alphabet Σ including the empty string ε.

the applications of their rules which reults in more accurate outputs in terms of the production rules. On the contrary the higher the order of a type, the lower is it's *generative capacity*, which is defined as it's ability of producing several expressions and preventing inaccurate products at the same time. The different types of the *Chomsky Hierarchy* are:

- *Unrestricted Grammar (Type-0)*: As it's name suggests, in this type of grammar there are no restrictions for the rules of generation. Any number of combinations of terminal or non-terminal symbols may appear on both sides of it's production rules. This type of grammar produces *recursively enumerable languages*, which can be defined informally as a language whose sentences can be checked for validity by some Turing-machine in (at best) a finite amount of steps[7]. Due to this property, *Type-0* grammars can exhibit an infinitely high computational complexity.

- *Context-Sensitive Grammar (Type-1)*: In this type of grammar an arbitrary number of terminal and/or nonterminal symbols may appear on both sides with the restriction that the number of symbols on the right-hand side must be equal or greater than the number of symbols on the left-hand side of the production rules. Context-sensitivity refers to the possible comprehension of the context during each replacement procedure. These languages are also known as *decidable languages*, i.e. a language-membership test for any given string can be done in a finite number of steps by it's appropriate automaton. Context-sensitive production rules are of the form $\alpha A\beta \longrightarrow \alpha\gamma\beta$ which means that an occurance of a nonterminal A can be replaced by another nonterminal γ given that A is embedded between the terminals and/or nonterminals α and β.

- *Context-Free Grammar (Type-2)*: Here the left-hand side of a rule consists of one single nonterminal, whereas the right-hand side may consist of zero or an arbitrary number of terminals and/or nonterminals. Formally this grammer consists of $G = (N, \Sigma, P, S)$ where $S \in N$ is the start symbol, N is a finite set of nonterminal symbols, Σ is a finite set of terminal symbols which at the end make up a sentence in the language and P is a set of production rules. Now each rule $p \in P$ is a pair of two symbols $p = (\alpha, \beta)$ where $\alpha \in N$ and $\beta \in (N \cup \Sigma)^*$. Hence the application of this rule $\alpha \longrightarrow \beta$ replaces some nonterminal symbol with a string of zero or more terminals and/or nonterminals.

- *Regular Grammar (Type-3)*: The left-hand side of the rules of this grammar consists of a single nonterminal of the alphabet. If it's right-hand side is a sequence

[7] Formally: let L be a recursive language, M the Turing-machine that accepts that language and w some string of symbols, then if $w \in L$, M recognizes it and halts in some final state, otherwise (if $w \notin L$) M halts in no final states and runs forever.

of a terminal followed by at most one nonterminal this form of production rule is refered to as *right-linear*. If the right-hand side consists of a non-terminal followed by a terminal it is known as *left-linear*. An example language of this grammar is a regular expression. For instance a regular expression of the form $\alpha^*\beta^+\gamma^*\delta$ can be generated by the *right-linear* grammar:

$$
\begin{array}{ccc}
S & \longrightarrow & \alpha A \\
S & \longrightarrow & B \\
A & \longrightarrow & \alpha A \\
A & \longrightarrow & \alpha B \\
B & \longrightarrow & \beta B \\
B & \longrightarrow & \beta C \\
C & \longrightarrow & \gamma C \\
C & \longrightarrow & \delta
\end{array}
$$

whith $N = \{A, B, C, S\}$, $\Sigma = \{\alpha, \beta, \gamma, \delta\}$ and S being the initial symbol. Languages which can be generated by this type of grammar are called *regular languages*[2].

3 Examples

Grammers are tools for analyzing [formal] languages. They are often described as a set of rules capable of generating well-formed and gramatical sentences within a given language. In the context of natural language these grammars are often refered to as *phrase structure grammars*. It can be defined as "a finite vocabulary (alphabet) Σ, a finite set S of initial strings in Σ, and a finite set P of rules of the form $X \longrightarrow Y$, where X and Y are strings in Σ."[6][8]

Here is an example[2] of a simplified phrase structure grammar of English:

$$
\begin{array}{cccccc}
S & \longrightarrow & NP & VP & & \\
VP & \longrightarrow & v & (NP) & (PP) & \\
AP & \longrightarrow & (adv) & a & & \\
PP & \longrightarrow & p & NP & & \\
NP & \longrightarrow & (det) & (AP) & n & (PP) \\
n & \longrightarrow & \text{man} & \text{girl} & \text{john} & \\
det & \longrightarrow & \text{a} & \text{the} & & \\
v & \longrightarrow & \text{met} & \text{saw} & & \\
a & \longrightarrow & \text{nice} & \text{good} & \text{quick} & \\
adv & \longrightarrow & \text{very} & \text{extremely} & & \\
p & \longrightarrow & \text{in} & \text{for} & \text{to} &
\end{array}
$$

[8] X and Y can be terminals, nonterminals or combinations thereof.

Implementing these simple rules we could generate random grammatical sentences which partially come close to correctly articulated english sentences [Programs, 0]. Some of the thus generated sentences are: *'the man met a nice girl ', ' man met a girl ', ' very quick man met the extremely quick girl ', ' quick man met quick girl ', 'the man saw ', 'the nice man to john met the man '* Even a nonsense like *'a good girl met a john '* is still fully acceptabale and well-formed in terms of the grammatical regularities of the english language. Formal grammars are hence a device for syntactical work, not a semantical one! They are tools for analysis of structure, not meaning. In the context of linguistics a sentence is a non-expandable well-ordered set of terminal symbols produced by the rules of production of a certain grammar[9]. Formal grammars can thus be used for creating languages evolving from a basic recursive syntax.

As an example in the following a language for creating simple palindrome sentences L_{pal}[10] will be composed. L_{pal} will have a minimalist alphabet $\Sigma = \{P, \varepsilon, 0, 1\}$ made up of two disjoint sets of terminal[11] alphabets $\{\varepsilon, 0, 1\}$ and a non-terminal alphabet $\{P\}$.

The first three production rules of this mini language are defined as follows:

$$P \longrightarrow \varepsilon \mid 0^{oe} \mid 1^{oe}$$

The above basic rules indicate that a null string ε, 0 or 1 are all in L_{pal} and hence valid and gramatical palindrome sentences, with oe showing whether the initial sentence is of odd ($oe = 1$) or even length ($oe = 2$).

The next two inductive rules of L_{pal} are definied as

$$P \longrightarrow 0P0 \mid 1P1$$

These two production rules indicate that new palindromic strings can be created by placing any valid palindrome sentences in between any two similar symbols recursively to create more complex palindromes.

The third and last rule of L_{pal} says that nothing else can be in L_{pal}!. Deducing the number of recursive steps needed for generating a palindrome sentence from it's desired length can be calculated by:

$$\frac{SentenceLength - oe}{2} \in \mathbb{N}_{\geq 0}$$

Implementing the L_{pal} [Programs, 4] based on the above definition of the language allows generation of grammatical palindromic sentences of arbitrary length such as:

[9] In the context of programming languages it is a syntactically correct formed part of a program.

[10] A palindrome is a string which reads the same forwards and backwards, e.g. OTTO, 0110 etc.

[11] In context-free grammers a terminal symbol is a character of the alphabet that appears in the strings generated by the grammar, whereas a non-terminal symbol is a placeholder for patterns of terminal symbols. Formally the alphabet is designated with Σ, the terminals with T and the nonterminals with N so that $\Sigma = \mathrm{T} \cup \mathrm{N}$ and $\mathrm{T} \cap \mathrm{N} = \varnothing$.

'10101', '0110000110', '11011100011000111011',
'0111010111100110101001010110011110101110' or
'1011000111111100101010101001111111000110 1'.

Another example for composing a formal language for basic arithmetic expressions with the alphabets $\Sigma = \{+, -, *, /, (,), \mathbb{R}, expression\}$ from which only the $expression$ will provide a productive and non-terminal symbol in the language can be carried out in the following way:

$$
\begin{array}{lcccc}
expression & \longrightarrow & \mathbb{R} & & \\
expression & \longrightarrow & (expression) & & \\
expression & \longrightarrow & expression & \times & expression \\
expression & \longrightarrow & expression & \div & expression \\
expression & \longrightarrow & expression & + & expression \\
expression & \longrightarrow & expression & \text{-} & expression
\end{array}
$$

With the first rule ending up in a terminal alphabet and hence the only non-recursive production rule of the language, this simple language depicts a formal grammar of any elementray arithmetic expression[12].

4 In music

Chomsky's important work in the linguistic *Syntactic Structures* from 1957 was the onset for a series of similar attempts in the field of music. Notably Ray Jackendoff's and Fred Lehrdahl's *A Generative Theory of Tonal Music* from 1983 aims at applications of more extended generative grammars for the tasks of musical analysis[2]. Also Mark J. Steedman provided some studies on the usage of generative principles for analyzing and generating musical structures[13].

In the following example a simple formal grammar for real[14] musical sequences L_{R-seq} will be composed. It provides (theoretically) infinite-length sequences of musical parameter by applying it's second production rule to the previously generated member

[12]This context-free grammar could also be expressed in a more dense form as:

$$E \longrightarrow EopE \mid (E) \mid \mathbb{R}$$

$$op \longrightarrow \times \mid \div \mid + \mid -$$

To keep this example simple i will not consider the innate ambiguity of this definition, i.e. that hereby the priority of operations has not been taken into account.

[13]e.g. *A Generative Grammar for Jazz Chord Sequence* from 1984 and *The Perception of Musical Rhythm and Metre* from 1977

[14]Inspired from the two alternative forms of theme answers in the exposition of a fuge, i am making an analogy to the notion of *real* versus *tonal* sequences. L_{R-seq} presumes for generation or recognition of sequences only the context-ignorant data which could be based on pitches, note values, frequencies, onset times etc., whereas for a tonal sequence further context declarations would be necessary. In this sense any tonal sequence could be considered as subset of a real sequence $L_{T-seq} \subset L_{R-seq}$.

of the sequence. L_{R-seq} will be a recursively enumerable[15] language, corresponding in the Chomsky Hierarchy to the superset of all types, namely *Type-0.* This has the formal grammar $L_{R-seq}(G) = (N, \Sigma, S, P)$ with S the start of a sentence, $N = \{X_0, ops_k\}$ set of nonterminals, $\Sigma = \varnothing$ an empty set of terminals and P:

$$\begin{array}{ccc} S & \longrightarrow & X_0 \\ X_i & \longrightarrow & O_{i \bmod k}(X_i) \\ O_{i \bmod k} & \longrightarrow & O_{(i+1) \bmod k} \end{array}$$

set of production rules where $O_{i \bmod k}$ is an element of ops_k, a k-ary tuple of symbol-constructing unary[16] operations with $k \in \mathbb{N}+$[17]. The use of the modulo operator in indexing the O implies the innate recursive nature of L_{R-seq}.[18]

The operations being applied on each X are responsible for giving the outpus of the language their unique shapes and are hence a crucial part of the grammar. They are defined as recursive functions whose output is the next element in L_{R-seq} or formally $\forall\, O : O_{i \bmod k}(X_i) \in L_{R-seq}$. Using this recursive nature of sequences, we can derive any arbitrary element of L_{R-seq} by doing:

$$X_{nth} = \begin{cases} X_0 & nth = 0 \\ O_{nth \bmod k}(X_{nth-1}) & nth > 0 \end{cases}$$

The second production rule of P generates thus all seccessive elements of the sequence:

$$X_i \longrightarrow X_{i+1}.$$

An implementation of the grammar of L_{R-seq} and for finding some nth elements in L_{R-seq} can be found in [Programs, 5].

It is important to mention once again that the coherence and uniformity of a sequence in L_{R-seq} is ensured through the application of it's operations on every sequence element. The operations guarantee that all symbols of a sentence are stringently related to each other and give the language it's conformity[19].

[15] Also known as Turing-recognizable, which suggests that for such a sequence there should exist an algorithm whose output is a list of memebers of the language. If necessary this algorithm can run forever! Hence there are many counterparts like the set of integers $\mathbb{Z}$ etc.

[16] Of course other numbers of operands and operation depths are conceivable, though here i confine myself to using unary functions to keep things simple.

[17] Where $k = 0$ (no operations provided) the result will be a sequence consisting of the initial symbol.

[18] The sentences of L_{R-seq} are theoretically infinitely long as part of their nature. I will use the superscript notation L^m_{R-seq} to confine the examples to the first m elements of a sequence.

[19] In this sense a sequence exists only through a relationship between it's members which is a by-product of abovementioned operations, e.g. (Banana, Toothbrush, Lamp, Boeing) is not a sequence since no evident relationship can be established between it's components, whereas (A red lamp, A turned-on lamp, A broken lamp) with a 3-ary operations tuple $ops_3 = (O_0 = paintred(object), O_1 = turnon(object), O_2 = break(object))$ is! So L_{R-seq} should be closed under the collection of it's operators in $ops_k \in N$. It means that for every element in the sequence there should exist at least one

Another property of this language is that L_{R-seq} with concatenation is a *monoid*, i.e. for all valid sentences in the language also their concatenations will result in a grammatical sentence of the language:

$$\forall\, seq \in L_{R-seq} : seq_0 + seq_1 + ... \in L_{R-seq}$$

This is a given property by the author to put an emphasis on the structures of sequences and relations of their constitutes, and to legitimate observations on micro-levels of more complex constructs for recognizing *well-formed*[20] sequences.
For instance the concatenation of the following three sequences:

$$
\begin{array}{lc}
(1) & L^{5}_{R-seq}(G) = \{N = \{60, (\lambda x.x + 7, \lambda x.x - 2)\}\} \\
(2) & L^{8}_{R-seq}(G) = \{N = \{(70, 72, 77), (\lambda t.(t_0 + 5, t_1 - 2, t_2 + 4), \lambda t.(t_0 - 4, t_1 + 6, t_2 - 7))\}\} \\
(3) & L^{10}_{R-seq}(G) = \{N = 78, (\lambda x.x + 5, \lambda x.x - 7, \lambda x.x + 1, \lambda x.x - 8)\}
\end{array}
$$

which are equal to:

$$
\begin{array}{lc}
(1) & (60, 67, 65, 72, 70, 77) \\
(2) & ((70, 72, 77), (75, 70, 81), (71, 76, 74), (76, 74, 78), \\
 & (72, 80, 71), (77, 78, 75), (73, 84, 68), (78, 82, 72), (74, 88, 65)) \\
(3) & (78, 83, 76, 77, 69, 74, 67, 68, 60, 65, 58)
\end{array}
$$

forms also a L_{R-seq}-compliant sequence:
(60, 67, 65, 72, 70, 77, (70, 72, 77), (75, 70, 81), (71, 76, 74), (76, 74, 78), (72, 80, 71), (77, 78, 75), (73, 84, 68), (78, 82, 72), (74, 88, 65), 78, 83, 76, 77, 69, 74, 67, 68, 60, 65, 58). Interpreting this sequence as a tuple of pitches [Programs, 6] we have:

Such a basic example of L_{R-seq}, a rigid sequential chromatic passage can be found in Chopin's Etude Op.10 No. 3 in E major, bars 38 upto 41. These three measures exhibit an interesting shift in the context of music. In contrast to the exposition and recapitulation sections of this piece and as part of the rather chromatically stamped middle section it exhibits a culmination in decentralization of the tonal language of the piece; a moment where the music seems to be headed nowhere or to an eternity as it proceeds in a pure recursive manner. This *endless moment of music* is backed in it's intervalic structure with Chopin's decision of using tritoni in both hands. The

operator which halts for that element:
$\forall\, X_i \in L_{R-seq}\ \exists\, O_{i \bmod k} \mid O_{i \bmod k}(X_i) \in L_{R-seq}$. Likewise something like the set of prime numbers $\mathbb{P}$ can not be considered as a sequence in this sense since there exist no operators which are capable of producing the next prime number based on the current one.

[20] Well-formedness is judged here merely in compliance with L_{R-seq}.

grammars for the right-hand and the left-hand parts of this section are shown below. Here the part A of the right hand represents the building of the *micro-sequences* which can be grouped in three notes each (a downward movement of a minor second followed by an upward jump of a fourth in the course of the first eight notes), whereas part B builds the *macro-sequences* being the three repetitions of part A (each time with a downward minor second transposition):

$$\text{(Rh-A)} \quad L^7_{R-seq}(G) = \{N = \{(68, 74), (\lambda t.(t_0 - 1, t_1 - 1), \lambda t.(t_0 + 5, t_1 + 5))\}\}$$
$$\text{(Rh-B)} \quad L^3_{R-seq}(G) = \{N = (Rh - A), \lambda T.(\lambda t.(t_0 - 1, t_1 - 1))\}$$

The left hand is accordingly built by nesting two different L_{R-seq}'s[21]:

$$\text{(Lh-A)} \quad L^7_{R-seq}(G) = \{N = \{(53, 59), (\lambda t.(t_0 - 1, t_1 - 1))\}\}$$
$$\text{(Lh-B)} \quad L^3_{R-seq}(G) = \{N = (Lh - A), \lambda T.(\lambda t.(t_0 - 1, t_1 - 1))\}$$

An implementation of these sequences (using the in [Program, 5] defined function 'sequence') can be found in [Program, 7] which outputs:

In the following example the intervalic structure of the opening theme of Piano Phase by Steve Reich is expressed as a sequence of temporal onsets and distances. The 12-notes long theme can be composed using solely it's first five pitches 64, 66, 71, 73 and 74[22]. It can be noticed that the infinite character of this theme is realized through repetitive and ordered onset times of each of these five notes. Here the rhythm of each single tone is a L_{R-seq}. The ops_1 contains the operator:

$$O(o_{a_b}) \mapsto o_{a_b} + (b + 1) \times d_a$$

where $b \in \mathbb{N}+$ is the index of the current repetition, $a \in \mathbb{N}+$ is the index of the pitch starting at it's initial onset time o_{a_b} and $d \in \mathbb{R}$ is the distance between two repetitions of the pitch. Note that since *ops* has only a single operator, the application of O on the last generated symbol of the sequence can also be notated by the lambda notation $\lambda o.o + d$. The grammars for each of the five pitches can thus be formulated as follows:

$$(64) \quad L_{R-seq}(G) = \{N = \{0, (\lambda x.x + 1.5)\}\}$$
$$(66) \quad L_{R-seq}(G) = \{N = \{0.25, (\lambda x.x + 1)\}\}$$
$$(71) \quad L_{R-seq}(G) = \{N = \{0.5, (\lambda x.x + 1.5)\}\}$$
$$(73) \quad L_{R-seq}(G) = \{N = \{0.75, (\lambda x.x + 1)\}\}$$
$$(74) \quad L_{R-seq}(G) = \{N = \{1, (\lambda x.x + 1.5)\}\}$$

[21]Although the grammars of part B's of both hands are virtually identical, i rewrite it for clarity.
[22]Expressed as MIDI key numbers.

The whole theme can be constructed by concatenating the generated sequences as is demonstrated in [Program, 8]:

5 Two musical dialects

"*There are two central problems in the descriptive study of language. One primary concern of the linguist is to discover simple and 'revealing' grammars for natural languages. At the same time, by studying the properties of such successful grammars and clarifying the basic conceptions that underlie them, he hopes to arrive at a general theory of linguistic structure.*"[6]

Frederic Rzewski's *Falling Music* for piano (and amplified sound sculpture ad lib.) and James Tenney's *Spectral Canon for Conlon Nancarrow* (for mechanical piano), though not inspired from eachother, exhibit highly similar structures. Such a coincidental close analogy between two pieces within the field of algorithmic music[23] is interesting if we keep in mind that the freedom of expression since the break-up of old music paradigms should have had created a plurality and divercity in musical languages. In this section i will attempt to compose a formal language based on these two pieces, which shows to what extent they are structurally interrelated and which would be capable of predicting a possibly indefinite number of new related phenomena. Hereby i will focus on the temporal formalities of the pieces and will not consider the methodes used by each composer for the organization of tonal material, as each piece has a rather unique approach to this subject.

Spectral Canon was composed between 1972 and 1974. The first realizations were made by Tenney together with composer Gordon Mumma in Santa Cruz, California, in 1974. The instrument should be retuned to be capable of producing the first 24 harmonics of A_1 which are assigned to 24 voices. Each voice plays 185 repetitions of it's assigned pitch while accelerating *harmonically*, which is followed by another 185 repetitions of the same pitch accompanied by an identical *harmonic* deceleration. The durational sequence is identical between all 24 voices, whereas the voices enter successively (at durational octaves of the first voice)[7]. *Falling Music* was composed a little earlier in May 1971. The piano part consists of 36 voices stretched over an ambitus of three octaves, from B_2 to Bb_5. Every voice is formed by repetitions of one of the pitches of the ambitus assigned to it, whereat the distances between each

[23] Or rather rigid, mechanical music

repetition are shortened by one pulse[24] unitl a duration of one pulse is reached. At this point the retrograde of the durations sequence is played and the voice stops.

Both of these pieces are built in an incremental manner, where new instances of the same layer with some sort of modification will be added to the running music. Also both pieces have a leading voice which is responsible for triggering those instances in due time. The fact that in *Spectral Canon* the time spans between to voice triggers are filled with other tone repetitions will not affect the overall definition of the language.

Each piece consists of an N-ary tuple of attack indices $j = (0, 1, ..., N-1)$ associated with some temporal onset identifiers $a_j \in \mathbb{R}$. Onset units[25] will be defined as functions of these identifiers $o : \mathbb{R} \rightarrow \mathbb{R}$. Voice indices will also be specified on the basis of [manipulated] attack indices of the leading voice notated in the following as j^*. A voice will be notated hereafter in the form $V_{voice-index_{onset-unit}}$.

Obviously the startup of such a machinery is the triggering of the initial leading voice:

$$S \rightarrow V_{0_{o(a_0)}} V_{0_{o(a_1)}} ... V_{0_{o(a_{N-1})}}$$

The attacks (left-hand side tuple) and their corresponding onset identifiers (right-hand side tuple) are:

$$a_j = \begin{cases} (a_0, a_1, ..., a_{366}) = (0, 1, ..., 183, 182, ..., 1, 0) & \textit{Tenney} \\ (a_0, a_1, ..., a_{70}) = (0, 1, ..., 35, 34, ..., 1, 0) & \textit{Rzewski} \end{cases}$$

The onset units for each piece are then computed recursively as follows:

$$o(a_j) = \begin{cases} 0 & j = 0 \\ \begin{cases} k \times \log_2(\frac{9+a_{j-1}}{8+a_{j-1}}) + o(a_{j-1}) & \textit{Tenney} \\ 36 - a_{j-1} + o(a_{j-1}) & \textit{Rzewski} \end{cases} & j > 0 \end{cases}$$

In the case of Tenney's onset times above, k is a constant set by him to hold the duration between first two attacks of each voice equal to 4 seconds[3]. It is equal to:

$$k \times \log_2(\frac{9}{8}) = 4$$

$$k = 4 \times (\log_2(\frac{9}{8}))^{-1}$$

$$k \approx 23.539799$$

The attack identifiers are carrying information for (1) triggering new voices and (2) providing the onset units for each voice. Defining the onsets in dependence on the

[24] An eight-note of tempo 76-80

[25] I call it units to allow a boarder and appropriate interpretion of whatever it should be. In the case of *Spectral Canon* this unit is in seconds, whereas in *Falling Music* eight-note beats are specified.

attack identifiers allows simple rhythmic and formal manipulations of the output pieces. From the above tuples of attack identifiers it can be easily noticed that both *Spectral Canon* and *Falling Music* exhibit a symmetrical construction plan. The only remaining step to complete the temporal definition of this formal language is formulating the production rules for new voice generations. This takes place whenever the leading voice (the voice with $voice - index = 0$) arrives at certain positions along it's roadmap or satisfy particular conditions. In the case of *Spectral Canon* each time the number of hitherto played tones (the very first attack of the leading voice excluded) is a multiple of 8 and is no bigger than 184[26] the next voice will be triggered. In *Falling Music* this is done on each of the first 36 tone-repetitions of the leading voice (again not taking $j = 0$ into account). Being dependent on attack indices we can identify these regeneration spots whenever the following conditions are satisfied:

$$c(j) = \begin{cases} \frac{j}{8} \in \mathbb{N} \wedge j \leq 184 & Tenney \\ 0 < j \leq 35 & Rzewski \end{cases}$$

upon which we are provided with the adjusted voice indices:

$$j^* = \begin{cases} \frac{j}{8} & Tenney \\ j & Rzewski \end{cases}$$

and thus the reproduction rule

$$V_{0_{o(a_j)}} \longrightarrow (V_{j^*_{o(a_j)}}, V_{j^*_{o(a_j)+o(a_1)}}, ..., V_{j^*_{o(a_j)+o(a_{N-1})}}) \mid \forall j : c(j)$$

Summarizing these we obtain our final grammar of a temporal canonical language, also capable of generating both of the above studied pieces:

$$\begin{array}{ccc} S & \longrightarrow & (V_{0_{o(a_0)}}, V_{0_{o(a_1)}}, ..., V_{0_{o(a_{N-1})}}) \\ V_{0_{o(a_j)}} & \longrightarrow & (V_{j^*_{o(a_j)}}, V_{j^*_{o(a_j)+o(a_1)}}, ..., V_{j^*_{o(a_j)+o(a_{N-1})}}) \mid \forall j : c(j) \end{array}$$

Considering also the pitches of *Falling Music* $(V_0, V_1, V_2..., V_{35}) = (82, 81, 80, ..., 47)$ this grammar would more specifically look like:

[26] Which makes up a total number of $(184 \div 8 = 23)$ + the leading voice = 24 voices.

$$
\begin{array}{lll}
 & & (82_{0=0}, 82_{1=36}, 82_{2=71}, 82_{3=105}, \\
 & & 82_{4=138}, 82_{5=170}, 82_{6=201}, 82_{7=231}, \\
 & & 82_{8=260}, 82_{9=288}, 82_{10=315}, 82_{11=341}, \\
 & & 82_{12=366}, 82_{13=390}, 82_{14=413}, 82_{15=435}, \\
 & & 82_{16=456}, 82_{17=476}, 82_{18=495}, 82_{19=513}, \\
 & & 82_{20=530}, 82_{21=546}, 82_{22=561}, 82_{23=575}, \\
 & & 82_{24=588}, 82_{25=600}, 82_{26=611}, 82_{27=621}, \\
 & & 82_{28=630}, 82_{29=638}, 82_{30=645}, 82_{31=651}, \\
S & \longrightarrow & 82_{32=656}, 82_{33=660}, 82_{34=663}, 82_{35=665}, \\
 & & 82_{36=666}, 82_{37=668}, 82_{38=671}, 82_{39=675}, \\
 & & 82_{40=680}, 82_{41=686}, 82_{42=693}, 82_{43=701}, \\
 & & 82_{44=710}, 82_{45=720}, 82_{46=731}, 82_{47=743}, \\
 & & 82_{48=756}, 82_{49=770}, 82_{50=785}, 82_{51=801}, \\
 & & 82_{52=818}, 82_{53=836}, 82_{54=855}, 82_{55=875}, \\
 & & 82_{56=896}, 82_{57=918}, 82_{58=941}, 82_{59=965}, \\
 & & 82_{60=990}, 82_{61=1016}, 82_{62=1043}, 82_{63=1071}, \\
 & & 82_{64=1100}, 82_{65=1130}, 82_{66=1161}, 82_{67=1193}, \\
 & & 82_{68=1226}, 82_{69=1260}, 82_{70=1295}) \\
82_1 & \longrightarrow & (81_{0=36}, 81_{1=72}, 81_{2=107}, 81_{3=141}, ..., 81_{68=1262}, 81_{69=1296}, 81_{70=1331}) \\
82_2 & \longrightarrow & (80_{0=71}, 80_{1=107}, 80_{2=142}, 80_{3=176}, ..., 80_{68=1297}, 80_{69=1331}, 80_{70=1366}) \\
82_3 & \longrightarrow & (79_{0=105}, 79_{1=141}, 79_{2=176}, 79_{3=210}, ..., 79_{68=1331}, 79_{69=1365}, 79_{70=1400}) \\
. & & \\
. & & \\
. & & \\
82_{35} & \longrightarrow & (47_{0=665}, 47_{1=701}, 47_{2=736}, 47_{3=770}, ..., 47_{68=1891}, 47_{69=1925}, 47_{70=1960})
\end{array}
$$

and for *Spectral Canon* with the frequencies:

$$(V_0, V_1, V_2 ..., V_{23}) = (55\text{Hz.}, 110\text{Hz.}, 165\text{Hz.}, ..., 1320\text{Hz.})$$

the grammar would be:

$$
\begin{array}{lcl}
S & \longrightarrow & (55\text{Hz}._{0=0}, 55\text{Hz}._{1=4.0}, 55\text{Hz}._{2=7.57812192802372}, 55\text{Hz}._{3=10.814926932180002}, \\
 & & 55\text{Hz}._{4=13.769898384723438}, 55\text{Hz}._{5=16.48820861428996}, 55\text{Hz}._{6=19.00497078604862}, \\
 & & 55\text{Hz}._{7=21.348020312747153}, 55\text{Hz}._{8=23.53979676944687}, \ldots, \\
 & & 55\text{Hz}._{364=201.91093141635625}, 55\text{Hz}._{365=204.8659028688997}, 55\text{Hz}._{366=208.10270787305598}) \\
55\text{Hz}._{\frac{8}{8}} & \longrightarrow & (110\text{Hz}._{0=23.53979676944687}, 110\text{Hz}._{1=27.53979676944687}, \ldots, 110\text{Hz}._{366=231.64250464250284}) \\
55\text{Hz}._{\frac{16}{8}} & \longrightarrow & (165\text{Hz}._{0=37.309695154170306}, 165\text{Hz}._{1=41.309695154170306}, \ldots, 165\text{Hz}._{366=245.41240302722628}) \\
55\text{Hz}._{\frac{24}{8}} & \longrightarrow & (220\text{Hz}._{0=47.079593538893754}, 220\text{Hz}._{1=51.079593538893754}, \ldots, 220\text{Hz}._{366=255.18230141194974}) \\
. & & \\
. & & \\
. & & \\
55\text{Hz}._{\frac{184}{8}} & \longrightarrow & (1320\text{Hz}._{0=107.92908546251081}, 1320\text{Hz}._{1=111.92908546251081}, \ldots, 1320\text{Hz}._{366=316.0317933355668})
\end{array}
$$

References

[1] Mark J. Steedman *A Generative Grammar for Jazz Chord Sequences* Music Perception, Fall 1984, Vol. 2, No. 1, 52-77

[2] Gerhard Nierhaus *Algorithmic Composition, Paradigms of Automated Music Generation*, SpringerWienNeyYork, 2010, 84

[3] Charles de Paiva Santana, Jean Bresson, Moreno Andreatta *Modeling and Simulation: The Spectral Canon for Conlon Nancarrow by James Tenney* UMR STMS, IRCAM-CNRS-UPMC 1, place I.Stravinsly 75004 Paris, France

[4] Andrew Carnie *Syntax: A Generative Introduction* 3rd Edition, 2013, John Wiley & Sons, Inc.

[5] Brady Clark *Syntactic Theory and the Evolution of Syntax* Northwestern University Department of LinguisticsEvanston, IL 60208-4090USA Biolinguistics7: 169197, 2013ISSN 14503417 http://www.biolinguistics.eu

[6] Noam Chomsky *Three Models for the description of Language* https://chomsky.info/wp-content/uploads/195609-.pdf

[7] Rob Wannamaker *The Spectral Music of James Tenney* Contemporary Music Review, February 2008, Pages 105 ff.

Programs

For this part the following modules are imported beforehand:

```
(import [random [randrange :as rnd
                 random :as r]])
(import [kodou [*]])
```

0 (Phrase structures)

```
;;; An implementation of an example from
;;; Nierhaus' book "Algorithmic Composition", page 86

;;; Join strings with a space
(defn join [&rest items]
  (setv items (->> items (.join " ")))
  items)

;;; Make occurance of s optional
(defn opt [s &optional [weight 0.5]]
  (if (<= (r) weight) s ""))

;;; Phrase structure
(defn ps [unit]
  (cond
    ;; non-terminals
    [(= unit "S") (join (ps "NP") (ps "VP"))]
    [(= unit "VP") (join (ps "v")
                         (ps (opt "NP"))
                         (ps (opt "PP")))]
    [(= unit "AP") (join (ps (opt "adv"))
                         (ps "a"))]
    [(= unit "PP") (join (ps "p") (ps "NP"))]
    [(= unit "NP") (join (ps (opt "det"))
                         (ps (opt "AP"))
                         (ps "n")
                         (ps (opt "PP")))]
    ;; terminals
    [(= unit "n") (rnd-sel (, "man" "girl" "John"))]
    [(= unit "det") (rnd-sel (, "a" "the"))]
    [(= unit "v") (rnd-sel (, "met" "saw"))]
    [(= unit "a") (rnd-sel (, "nice" "good" "quick"))]
    [(= unit "adv") (rnd-sel (, "very" "extremely"))]
    [(= unit "p") (rnd-sel (, "in" "for" "to"))]
    [(= unit "") ""]))

(ps "S") ;; => 'the  man  met a  nice girl  '
(ps "S") ;; => '  man  met a  girl  '
(ps "S") ;; => '  very quick man   met the extremely quick girl  '
(ps "S") ;; => '  quick man  met   quick girl  '
(ps "S") ;; => 'the  man  saw  '
(ps "S") ;; => 'the  nice man to   john  met the  man  '
```

1 (Phrase structures)

Another simplified example of english grammar:

$$
\begin{array}{ccccccc}
sentence & \longrightarrow & subject & verbphrase & object & \\
verbphrase & \longrightarrow & adverb & verb & & \\
object & \longrightarrow & article & noun & & \\
subject & \longrightarrow & These \mid & Computers \mid & We & \\
adverb & \longrightarrow & never \mid & \varepsilon & & \\
verb & \longrightarrow & run \mid & are \mid & tell & \\
article & \longrightarrow & the \mid & a \mid & \varepsilon & \\
noun & \longrightarrow & university. \mid & world. \mid & cheese. \mid & lie.
\end{array}
$$

```
;;; Choose a random element of seq
(defn rnd-choose [seq]
  (get seq (rnd (len seq))))
(defn subject []
  (rnd-choose (, "These" "Computers" "We")))
(defn determiner []
  (rnd-choose (, "the" "a" "")))
(defn noun []
  (rnd-choose (, "university." "world." "cheese." "lie.")))
(defn adverb []
  (rnd-choose (, "never" "")))
(defn verb []
  (rnd-choose (, "run" "are" "tell")))

;;; Generate a sentence
(defn sentence []
  (-> " "
    (.join [(subject)
            (adverb)
            (verb)
            (determiner)
            (noun)])))
```

Some of the thus generated sentences are:

'We run the university.', 'We never are the world.', 'Computers never run the world.', 'Computers never are a world.', 'Computers are a world.', 'Computers run the university.', 'We never tell the university.', 'These never tell the world.', 'These run a university.'

2 (Phrase structure)

```
;;; Recursive definition of the English Phrase Structure from Chomsky's
;;; "Three models ..." pages 117 & 118, examples 20, 21, 23 & 24
(defn make [what
            &optional
            [nps ["the man" "the book"]]
            [verbs ["took"]]]
      (cond [(= what "NP") (rnd-choose nps)]
            [(= what "VP") (+ (make "Verb" nps verbs)
                              " "
                              (make "NP" nps verbs))]
            [(= what "Verb") (rnd-choose verbs)]
            [(= what "Sentence") (+ "#" (make "NP" nps verbs)
                                        " "
                                        (make "VP" nps verbs) "#")]))

(make "Sentence") ;; => '#the man took the book#'
(make "Sentence"
      ["they" "planes" "flying planes"]
      ["are flying" "are"])  ;; => '#they are flying planes#'
```

3 (Phrase structure)

```
;;; Join strings with a space
(defn join [&rest items]
  (setv items (->> items (.join " ")))
  items)

;;; Phrase Structure
(defn ps [unit]
  (cond
    [(= unit "S") (join (ps "NP") (ps "VP"))]
    [(= unit "NP") (join (ps "a") (ps (rnd-sel (, "n" "NP"))))]
    [(= unit "VP") (join (ps "v") (ps "adv"))]
    [(= unit "a") (rnd-sel (, "colorless" "green"))]
    [(= unit "n") "ideas"]
    [(= unit "v") "sleep"]
    [(= unit "adv") "furiously"]))

(ps "S") ;; => 'colorless green ideas sleep furiously'
(ps "S") ;; => 'colorless ideas sleep furiously'
(ps "S") ;; => 'green colorless ideas sleep furiously'
(ps "S") ;; => 'green green green ideas sleep furiously'
(ps "S") ;; => 'green ideas sleep furiously'
(ps "S") ;; => 'colorless green ideas sleep furiously'
```

4 (Palindrome)

```
;;; Deduce the number of required steps
(defn count-steps [init-size sentence-size]
  (// (- sentence-size init-size) 2))

;;; Expand the existing palindromic sentence
(defn expand-sentence [sentence]
  (setv word (rnd-choose ["0" "1"]))
  (+ word sentence word))

;;; Generate a palindromic sentence recursively
(defn sentence [size]
  (setv init-size (if (zero? (% size 2)) 2 1))
  (setv steps (count-steps init-size size))
  (setv init-sentence (* (rnd-choose ["0" "1"]) init-size))
  (loop [[s init-sentence]
         [i steps]]
        (if (zero? i)
            s
            (recur (expand-sentence s) (- i 1)))))

(sentence 5)  ;; => '10101'
(sentence 10) ;; => '0110000110'
(sentence 20)  ;; => '11011100011000111011'
(sentence 40)  ;; => '0111010111100110101001010110011110101110'
(sentence 41)  ;; => '10110001111111001010101010011111110001101'
```

5 (L(R-seq))

```
  ;;; Grammar of L(R-seq)
(defn sequence [X0 ops m]
  (setv k (len ops))
  (setv m (if (zero? k) 0 m))
  (loop [[i 0]
         [seq [X0]]]
        (if (= i m)
            seq
            (recur (+ i 1)
                   (+ seq [((get ops (% i k)) (last seq))])))))

;;; Searching the nth symbol of some sentence of L(R-seq)
(defn seq-elem [X0 ops nth_]
  (setv L (len ops))
  (loop [[i 0]
         [curr X0]]
        (if (= i nth_)
            curr
            (recur (+ i 1)
                   ((get ops (% i L)) curr)))))

(sequence "-" (, (fn [x] (+ x ">"))) 3) ;; => ['-', '->', '->>', '->>>']
(sequence "-" () 3) ;; => ['-']
(sequence "-" (, (fn [x] (+ x ">"))) 0) ;; => ['-']
(seq-elem "-" (, (fn [x] (+ x ">"))) 3) ;; => '->>>'
(sequence 0 (, (fn [x] (+ x 1.5))) 9)
;;; => [0, 1.5, 3.0, 4.5, 6.0, 7.5, 9.0, 10.5, 12.0, 13.5]
(seq-elem 0 (, (fn [x] (+ x 1.5))) 9) ;; => 13.5
```

6 (Sequence Concatenation)

```
  ;;; Concatenation of three sequences results
  ;;; in a new valid sequence.
  (kodou
    (Part
      {"notes"
        (+ (sequence 60
                     (, (fn [x] (+ x 7))
                        (fn [x] (- x 2)))
                     5)
           (sequence [70 72 77]
                     (, (fn [L] [(+ (get L 0) 5)
                                 (- (get L 1) 2)
                                 (+ (get L 2) 4)])
                        (fn [L] [(- (get L 0) 4)
                                 (+ (get L 1) 6)
                                 (- (get L 2) 7)])) 8)
           (sequence 78 (, (fn [x] (+ x 5))
                           (fn [x] (- x 7))
                           (fn [x] (+ x 1))
                           (fn [x] (- x 8)))
                     10))
      "beats" (range 26)}))
```

7 (Chopin)

```
(kodou
  (Part
    {"notes"
       [(reduce +
          (sequence
            (sequence [68 74]
                      (, (fn [chord] (lfor n chord (- n 1)))
                         (fn [chord] (lfor n chord (+ n 5))))
                      7)
            (, (fn [phrase] (lfor chord phrase (lfor n chord (- n 1)))))
            3))
        (reduce +
          (sequence
            (sequence [53 59]
                      (, (fn [chord] (lfor n chord (- n 1))))
                      7)
            (, (fn [phrase] (lfor chord phrase (lfor n chord (- n 1)))))
            3))]
     "beats" (lfor _ (range 2)
                (sequence 0.25
                          (, (fn [beat] (+ beat 0.25)))
                          23))}
    {"staff" {"n" 2 "bind" "piano"}
     "clef" {1 {0 "bass"}}
     "timesig" {0 (, 2 4)}}))
```

8 (Reich)

```
(kodou
  (Part
    {"notes"
      (reduce +
             (lfor pitch (, 64 66 71 73 74)
               (sequence pitch
                         (, (fn [x] x))
                         7)))
     "beats" (reduce +
                     (lfor onset-dist [[0 1.5]
                                       [.25 1]
                                       [.5 1.5]
                                       [.75 1]
                                       [1 1.5]]
                         (sequence (get onset-dist 0)
                                   (, (fn [x] (+ x (get onset-dist 1))))
                                   7)))}
    {"timesig" {0 (, 3 4)}}))
```