Brian Heim – Department of Music Ph.D. Applicant

Recursive Generation of Rhythmic Structures with PTGGs

By Brian Heim

Final Project for CPSC 431, Spring 2015, Prof. Donya Quick

Abstract

In this paper, I describe an approach for generating passages of rhythm with a probabilistic temporal graph grammar (PTGG) using rulesets that create recursive subdivisions. I first provide an overview of the PTGG implementation written by Donya Quick in Haskell, then present my general approach along with an example rule set and output. Finally, I discuss the possibilities these results suggest for their use in automated composition, and I describe the changes made in my port of the code for Quick's general PTGG and my rhythmic PTGG into the music programming language SuperCollider.

1. Introduction

In this paper, I describe an approach for generating passages of rhythm via the use of a probabilistic temporal graph grammar (PTGG) with rule sets that operate via recursive subdivision. This project is motivated by three factors: (1) the recognition that the literature on computer-generated rhythm is generally not as robust as those for melody and harmony; (2) my desire as a composer of avant-garde classical music to have more tools for computer-assisted composition; and (3) the hope that such an approach, having already produced compelling results in Donya Quick's Kulitta project [4], may provide insight into the structure of rhythms found in existing repertoires.

2. PTGGs

2.1. Definition and Features

A grammar is defined as a 4-tuple: $G = (N, T, R, S)$ where the alphabet $N \cup T$ comprises nonterminals N and terminals T. R is a set of rules of the form $x \in N \rightarrow y^*$ where x is a nonterminal symbol and y^* is any string of symbols; $S \in N$ is the starting symbol. Terminals are symbols that only produce themselves, i.e. cannot be further altered, while nonterminals may produce any combination of terminals and nonterminals. The generative algorithm is similar to that of L-Systems: at each generative iteration of the grammar, starting with the string $X = S$, the appropriate rule $(n \rightarrow y) \in R$ is applied to each nonterminal symbol left to right to produce a new string X'. This process of iteration is called expansion, because the string length is a nondecreasing function of the number of iterations. The condition that production rules operate on individual symbols without knowledge of the surrounding symbols means that this definition also describes a context-free grammar (CFG).

A probabilistic context-free grammar (PCFG) is a CFG with a rule set that may contain multiple production rules per nonterminal. Rules take the tuple form $r = (x \rightarrow y^*, \pi)$ where the additional element $\pi \in [0,1]$ is the probability of r being applied to an instance of symbol x. I denote the probability of rule r as π_r. There is an added constraint that all probabilities for a given nonterminal sum to 1. As noted by Paul Hudak and Donya Quick [-2], PCFGs are a reasonable approach to the problem of automated music composition, as they are efficient, easily defined, and take into account the fact that probabilistic harmonic behavior has been observed in significant repertories of classical music [8, 9].

A probabilistic temporal graph grammar as defined by Quick and Hudak in [5, 6, 7] is distinguished by two features:

(1) Each symbol is accompanied by a parameter p with the notation x^p. The parameter typically has the same data structure for all symbols, and may be arbitrarily operated on by the rules. Also, a starting parameter must be specified for the starting symbol S.

(2) Production rules may convert symbols into let-in constructions of the following form:

$$S \rightarrow \boldsymbol{let}\, x = A\, \boldsymbol{in}\, xBx \tag{1}$$

where $S, A \in N$. This syntax's behavior is similar to that of the **let** and **in** Haskell keywords: after generation, the expression represented by variable x is expanded once, and all instances of x take the value of the expanded string. These structures may nest: e.g., the rule $A \rightarrow \boldsymbol{let}\, x = A\, \boldsymbol{in}\, xBx$ applied to the starting string A twice in succession would result in the strings

$$\boldsymbol{let}\, x = A\, \boldsymbol{in}\, xBx \tag{2}$$

$$\boldsymbol{let}\, x = (\boldsymbol{let}\, y = A\, \boldsymbol{in}\, yBy)\, \boldsymbol{in}\, xBx. \tag{3}$$

In the final phase of PTGG string generation, all variables are resolved to their referenced expressions. For instance, the above two statements would become ABA and $ABABABA$ respectively. Via this modification, PTGGs, unlike plain PCFGs, can express the notion of repetition, making them more useful in many musical contexts.

3. A PTGG for Rhythm

In designing a PTGG for rhythm, I took as a starting point the treatment of rhythm in Quick's Kulitta program. There, the symbol parameter *p* represents a rhythmic duration, and rules operate on this parameter by dividing it, typically by a power of two. Moreover, this division always results in a string that has equal duration to the original symbol. In her notation, a rule such as

$$A^t \rightarrow B^{\frac{t}{2}}\, C^{\frac{t}{4}}\, D^{\frac{t}{4}} \tag{4}$$

indicates that wherever symbol *A* produces string *BCD* the durations of the produced symbols are ½, ¼, and ¼ the durations of A respectively. This implies a general approach of beginning with a long duration and iteratively dividing it into progressively smaller durations; since one would typically like to have control over the exact duration of a generated composition, this is a reasonable approach.

The grammar I propose is based on the observation that many theories of Western counterpoint, including those of Fux, Schoenberg, and Schenker [1, 10], treat rhythm as a hierarchy of nested structures, discuss metrical patterns such as *anapest* (short-short-long) in the same light regardless of the level of duration on which they occur, and regard small note values as elaborations or ornamentations of a less complex metrical pattern. For the sake of focus the grammar is defined only for measures of 4/4 meter, since many counterpoint treatises begin in—and sometimes never leave—this meter.

Durations are treated as both terminals and nonterminals of an infinite parametrized alphabet of duration types in a two-step generative process that first divides a “raw” duration, equal to a power of two, into a sequence of measures; second, these measures are transformed into whole note durations and subdivided further. Throughout the following sections, I use the value 1.0 to represent the duration of a whole note, which is also the duration of a measure.

3.1. Nonterminals

The nonterminals of the grammar are

$$N = \{Measure, Beat\}. \tag{5}$$

Beat represents a note with duration $\frac{2^n}{m}$ where *n* and *m* are integers with $n \geq 0, m \geq 1$; *Measure* is the term used for a raw duration whose value is a positive integer. *Measures* only appear in the first phase of production and are replaced with *Beats* of duration 1.0 in the second phase. The reason for this distinction is twofold: first, it was desired that the second phase of production begin with a series of 1.0 durations; second, it allows the frequency of variable instantiation at the hypermetrical level to be controlled independently of that at the metrical level.

3.2. Terminals

The terminals used are

$$T = \{Dotted, Short\}. \tag{6}$$

Dotted represents a duration that, like a dotted note, lasts for 1.5 times the value of its undotted value. It was chosen to be a terminal for convenience since any subdivision of a dotted value can also be articulated as a more complex subdivision of a longer duration. *Short* stands for a *Beat* which has been subdivided to the point where its duration is less than or equal to $\frac{1}{2^n}$, where *n* is an arbitrary and predetermined integer constant. This behavior is designed to mimic the limits of human performers' rhythmic articulation.

The notations M^d, B^d, S^d, D^d will be used hereafter for *Measures, Beats, Shorts,* and *Dotteds* of duration *d* respectively.

3.3. Sentential Forms

The set K of *sentential forms* of this PTGG, without considering parameter values, is

$$k \in K ::= M \mid k \ldots k \mid \textbf{let}\, x = k\, \textbf{in}\, k \tag{7}$$

for the first phase and

$$l \in L ::= B \mid S \mid Dl \mid lD \mid l \ldots l \mid \textbf{let}\, y = l\, \textbf{in}\, l. \tag{8}$$

for the second, where $x \in V, y \in W$, and V, W are predefined sets of variables.

3.4 Production Phases

Production occurs in two phases of rulesets, and a sentence is not considered well-formed until passing through both phases. Phase 1 begins with the string M^d where $d = 2^a, a \in \mathbb{N}$, and ends with the string $A \in K$. Letting A_V be the set of all M^d in A and all A's variables, Phase 1 only halts when $\forall M^d \in A_V, d = 1$. Phase 2 begins with the string B obtained by replacing all $M^d \in A_V$ with B^d, and halts after a predeterminsed number of iterations. At each iteration of both phases, the sum D of durations in the string remains constant.

3.5. Production Rules

Production rules in Phase 1 are functions from measures to measure groups, $M \rightarrow K$, and in Phase 2 from beats to beat groups, $B \rightarrow L$. The strict definition of PFCG says that a single rule must produce a single string of symbols, as in the case of $B \rightarrow BB$ and $B \rightarrow SS$. Yet *Short*, as noted above, in only produced when its parameter meets certain conditions. Since the decision to output *B, D* or *S* can be made solely on the basis of the parameter value, I instead define rules as functions

$$B^a \rightarrow X \in \mathbb{Q}^p \text{ where } p > 0, \forall x \in X: 0 < x \leq 1, \textstyle\sum_{x \in X} x = 1 \tag{9}$$

and then use the resulting parameters $[d_1, d_2, \dots, d_p] = aX$ to label the produced symbols. This allows me to define patterns of subdivision clearly and succinctly.

For example, let two Phase 2 rules be $B \rightarrow [\frac{3}{4}, \frac{1}{4}]$ and $B \rightarrow \left[\frac{1}{2}, \frac{1}{4}, \frac{1}{4}\right]$. Here are two iterations of production using these rules and a *Short* threshold of $\frac{1}{16}$:

Rule	String
[start]	B^{1}
$B \rightarrow [\frac{3}{4}, \frac{1}{4}]$	$D^{3/4}\, B^{1/4}$
$B \rightarrow \left[\frac{1}{2}, \frac{1}{4}, \frac{1}{4}\right]$	$D^{3/4}\, B^{1/8}\, S^{1/16}\, S^{1/16}$

4. Implementation

My implementation uses the PTGG framework written by Donya Quick in Haskell and the MIDI playback capabilities offered by Paul Hudak's Euterpea [3] to allow for elegant rule definition and parsing. Below are some of the key features of this implementation.

4.1. Algorithm for Two-Phase Generation

The method $fullgen(m, i, pLetM, maxPoT, s)$ generates a complete and playable sentence from five arguments: *m*, the total duration of the passage; *i,* the number of Phase 2 iterations; *pLetM*, the probability of expansion using variable instantiation in Phase 1; *maxPoT*, the shortest allowable duration $\frac{1}{2^n}$ expressed as *n*; and a trivial random seed argument *s* for choosing among rules.

Phase 1 uses only two rules: $M^d \rightarrow M^{\frac{d}{2}}\, M^{\frac{d}{2}}$ and $M^d \rightarrow \boldsymbol{let}\ x = M^{\frac{d}{2}}\ \boldsymbol{in}\ x\ x$, which have the effect of subdividing the input duration into two halves. Since the input duration m is a power of two, Phase 1 is completed in $log_2\, m$ iterations. The parameter $pLetM \in [0,1]$ determines the probability mass distribution between the two rules.

In Phase 2, the algorithm is as follows:

For each symbol *X* in string *A*:

1. From the set of rules applicable to *X*, choose a rule *R* according to their p.m.d.
2. Determine if the subdivision would create values smaller than the limit set by *maxPoT*. If so, choose another rule until this is not true.
3. Produce a substring *X'* using *X* and *R*.
4. Concatenate all *X'* produced from *A* to produce a new string *A'* to use in the next iteration.

Repeat these steps for *i* iterations.

At the end of the process, a single whole note beat is added to the end of the sequence for the sake of stylistic normalcy.

4.2. Parsing of Subdivision Rules and Tuplet Creation

The subroutine $subdivN(p, xs)$ was written to simplify the task of rule declaration. From a duration parameter *p* and array of integers *xs* it creates an array of symbols representing the subdivision of B^p into beats of sizes $xs/\sum_{xs}$ including the decision to classify symbols as *B*, *D*, or *S*. The definition of the production rules in (9) implies that the vector *X* may be expressed as X'/a for some integer *a* and therefore $subdivN$ uses the sum of *xs* to infer this common denominator. As a result, the production rules from section 3.5 can also be written as function calls $subdivN(p, [3, 1])$ and $subdivN(p, [2, 1, 1])$. Since the duration parameters are implemented using the Haskell Rational type, which expresses numerical values as ratios of integer pairs, $subdivN$ is also able to work with arbitrary subdivisions of a beat in the same way tuplets behave in music notation. Thus, there is no conceptual difference between the calls $subdivN(p, [2, 1, 1])$ and $subdivN(p, [3, 1, 1, 4, 2])$: the latter call divides the beat into 11 equal parts, but so long as none of these durations fall below the minimum duration threshold, it is a valid operation. Symbols are still classified as they would be using a power-of-two denominator: the result of the call $subdivN(1, [3, 1, 1, 4, 2])$ is the string $D^{\frac{2}{11}}\ B^{\frac{1}{11}}\ B^{\frac{1}{11}}\ B^{\frac{4}{11}}\ B^{\frac{2}{11}}$.

5. Example

I now present a fully defined instance of the PTGG approach outlined above and use it to generate a substantial rhythmic passage.

5.1. Rule set

The production rules used for Phase 2 are shown in Table 1. As noted earlier, the first production phase always employs the same two rules, and its only free parameter is the probability mass distribution between the rules. The rule set sticks to simple rhythmic templates, never subdividing by more than a factor of 4. The right-hand sides are groupings typically found in Baroque and Classical era music, and were chosen through intuition. A more rigorous approach to learning these groupings and probabilities is a topic for future research.

The right-hand side of each rule is written as a transformation from *Beats* to *Beats* and *Dotteds*. In the actual operation of the algorithm, the right-hand side *Beats* are replaced with *Shorts* when needed. The exception is Rule 7, which always uses a *Short* in its produced string. Although the probability mass distribution among rules applicable to *Beats* remains nominally static, the actual distribution may change while iterating due to the minimum duration constraint, which will disqualify rules that subdivide into smaller values before those that subdivide into larger values.

Some pairs of rules, such as Rules 8 and 9, use the same metrical subdivision in production rules that both do and do not link the shorter values through variable instantiation. There is one rule (#5) to produce a *Dotted* symbol and one rule (#10) that produces a triplet figure. The possibility in Rule 1 for a *Beat* to remain unchanged allows some parts of the passage to develop more complex groupings than others; the potential for rhythmic variety would be reduced if this rule were omitted the potential for rhythmic variety would be reduced.

Rule No.	Probability	Rule
1	0.20	$B^d \rightarrow B^d$
2	0.10	$B^d \rightarrow B^{\frac{d}{2}}\,B^{\frac{d}{2}}$
3	0.10	$B^d \rightarrow \boldsymbol{let}\ x = B^{\frac{d}{2}}\ \boldsymbol{in}\ x\ x$
4	0.15	$B^d \rightarrow B^{\frac{d}{2}}\,B^{\frac{d}{4}}\,B^{\frac{d}{4}}$
5	0.15	$B^d \rightarrow D^{\frac{3d}{4}}\,B^{\frac{d}{4}}$
6	0.10	$B^d \rightarrow B^{\frac{d}{4}}\,B^{\frac{d}{2}}\,B^{\frac{d}{4}}$
7	0.10	$B^d \rightarrow \boldsymbol{let}\ x = B^{\frac{d}{4}}\ \boldsymbol{in}\ x\ S^{\frac{d}{2}}\ x$
8	0.05	$B^d \rightarrow B^{\frac{d}{4}}\,B^{\frac{d}{4}}\,B^{\frac{d}{4}}\,B^{\frac{d}{4}}$
9	0.05	$B^d \rightarrow \boldsymbol{let}\ x = B^{\frac{d}{4}}\ \boldsymbol{in}\ x\ x\ x\ x$
10	0.05	$B^d \rightarrow B^{\frac{d}{3}}\,B^{\frac{d}{3}}\,B^{\frac{d}{3}}$
11	1.00	$D^d \rightarrow D^d$
12	1.00	$S^d \rightarrow S^d$

Table 1. Production rules of Phase 2 of the grammar that generates Figure 1.

5.2. Results

Figure 1 shows an example output of the fully implemented generation algorithm. The *pLetM* argument was set at 0.25, meaning that at each iteration of splitting a *Measure* into two halves there was a 25% chance of variable instantiation. The second production phase was stopped after 3 iterations and had a *maxPoT* argument of 4. This accounts for the lack of subdivisions beyond sixteenth notes in the passage, although they could have easily appeared through repeated applications of Rules 4–9.

Since the goal of this project was not to imitate any style directly, but rather to explore the versatility of this method, the comments I offer now about this particular example are cursory in nature. The passage exhibits rhythmic variety while also maintaining some moments of symmetry at the hypermetrical (mm. 1-2 and 3-4) and metrical (m. 12) levels, which I would expect from the mix of rules which use and do not use variable instantiation. In some places, the rhythms feel quite natural, while at other times (mm. 9-11) there are abrupt changes and uncommon figures. Although the passage does seem to be grouped into four-bar segments, there is a lack of overall phrasing and pacing. The most surprising moment is m. 12, which does not seem likely given my rule set, but could have appeared as a result of applying the rule sequence 9–2–2. Overall, this passage is rough, but promising.

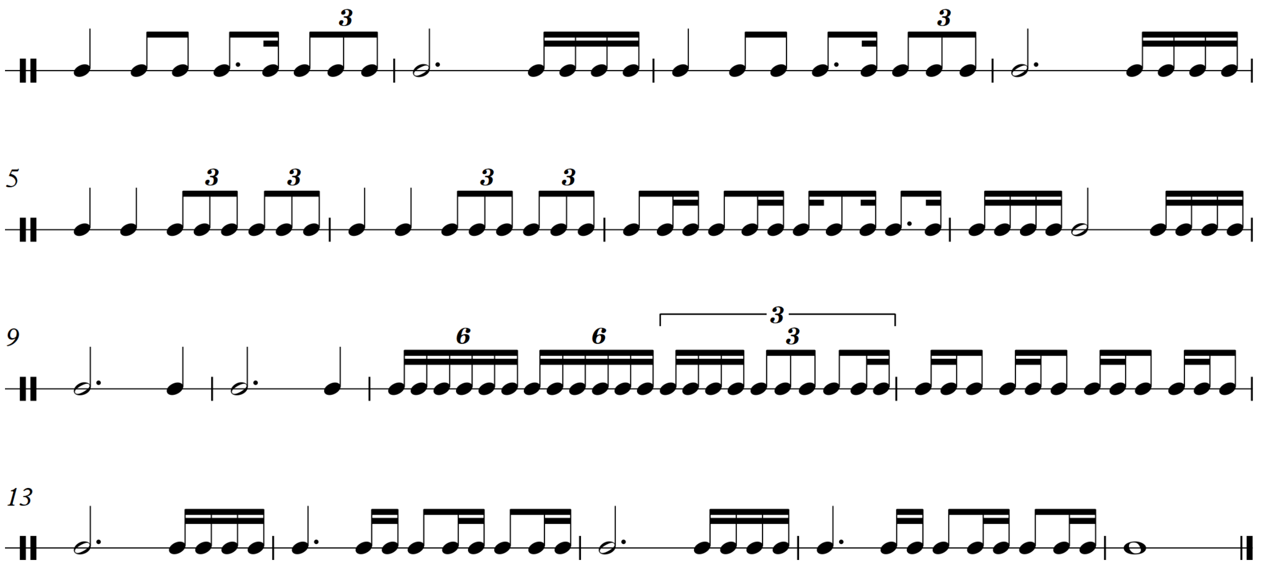

Figure 1. An example of a long rhythmic passage from my implementation.

7. Future Work

I have presented an application of PTGGs to the topic of automated composition along the sole parameter of rhythm. I detailed one implementation of such a grammar and examined a sample of its output. The grammar shown above is a preliminary attempt, and there are some possible routes for its extension.

One clear choice would be to try introducing rhythmic concepts that this grammar cannot currently express. These include ties, rests, and triple (3/4) meter. The former two items suggest the introduction of more alphabet symbols: a *Tied* symbol, produced from a *Beat* or *Short*, could be interpreted as tied over from the previous note during playback. A *Rest* symbol could be similarly used to eliminate some articulated notes from the interpreted string. Adding compatibility for triple meter would imply reworking *Dotted* as a nonterminal symbol.

Another concept that seems to be missing from the current alphabet is that of an embellishment or ornamentation. Such symbols could add short note values before or after more metrically significant beats, and in so doing introduce a third level of metrical hierarchy. A separate way to approach this would be by allowing existing ***let-in*** structures in the string to be resolved during or between iterations, rather than at the end of the entire cycle. This would preserve high-level symmetry while allowing for small-scale variations.

With certain adaptations, the proposed grammar could generate rhythms in more contemporary styles: for instance, the composition of mixed-meter passages would require a reworking of the first, *Measure*-based production phase, and might be better generated additively as opposed to recursively. The grammar already supports the complex and nested tuplets which are often found in classical music of the 20th and 21st centuries, and by slackening the imposed constraint on constant total string duration, it might also be able to generate passages in free meter or “non-geometric” time.

Another direction to explore is context sensitivity. In some cases, it might be helpful to know certain contextual details about a symbol such as: (1) its position within the measure, which typically determines its emphasis (downbeat vs. upbeat), (2) its position within the overall duration of the passage, and (3) whether the passage is increasing or decreasing in rhythmic activity. Otherwise, it is not possible to account for what Schoenberg termed “the tendency of the smallest notes” [10]—a contrapuntal phenomenon wherein shorter durations are used more frequently toward the end of a passage or phrase—without adding other constraints to the rule set.

Finally, these variations and improvements could also be useful for interfacing my grammar with other automated composition algorithms that produce melodies, harmonies, and counterpoint, either by using its output as a blueprint or by including it in a “composition by committee” approach. In that case, it would be important to embed as much contextual and functional knowledge as possible into the alphabet and the symbol parameters so that other grammars and algorithms can make informed decisions. For example, the knowledge that a downbeat is tied over or that two halves of a phrase have some rhythmic symmetry could be interpreted by a harmonic composition algorithm to produce more contextually aware chord sequences.

Bibliography

[1] Johann Joseph Fux, Alfred Mann, and John Edmunds. *Gradus ad Parnassum: The Study of Counterpoint*. WW Norton & Company, 1965.

[2] Robert Giegerich. Introduction to stochastic context free grammars. *RNA Sequence, Structure, and Function: Computational and Bioinformatic Methods*, pages 85-106, 2014.

[3] Paul Hudak. Euterpea, 2014. Software library.

[4] Donya Quick. Kulitta, 2014. Software program.

[5] Donya Quick. *Kulitta: a Framework for Automated Music Composition*, 2014.

[6] Donya Quick and Paul Hudak. A temporal generative graph grammar for harmonic and metrical structure. In *Proceedings of the International Computer Music Conference*, 2013.

[7] Donya Quick and Paul Hudak. Grammar-based automated music composition in haskell. In *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling, and design*, pages 59–70, 2013

[8] Christopher Raphael and Joshua Stoddard. Functional harmonic analysis using probabilistic models. *Computer Music Journal*, 28(3):45–52, 2004.

[9] Martin Rohrmeier and Ian Cross. Statistical properties of tonal harmony in Bach's chorales. In *International Conference on Music Perception and Cognition*, 2010.

[10] Arnold Schoenberg. *Preliminary exercises in counterpoint*. Ed. Leonard Stein. Faber & Faber, 1963.

Appendix: Haskell Code

```
> module Final where
> import Euterpea
> import PTGG
> import System.Random
> import Data.List
> import Data.Fixed

===================
basic definitions:

Maximum power of two used as a subdivision (i.e. maxPoT = 4 => 1/(2^4) = 1/16 is
smallest unit in a measure)

> maxPoT = 4

The main symbol is "Beat", which is any even subdivision of a measure (half, quarter,
and eighth notes, for example)
"Dotted" is a Beat with 3/2 the duration
"Short" is a Beat which has reached the maximum power of two and cannot be further
subdivided (convenience)

> data RTerm = Measure | Beat | Dotted | Short
>     deriving (Eq, Ord, Enum, Read, Show)

> allRTerms = [Measure, Beat, Dotted, Short]

===================
parameters

Three parameters:
      power and ratio: the duration of the beat is 1/2^power*ratio. Ratio is usually
1, unless the Beat is within a tuplet in which case it becomes <written number of
beats in tuplet>/<actual duration of tuplet>. Thus a triplet has ratio 2/3, a
quintuplet ratio 4/5
      measures: determines the length of a Measure object during the initial phase of
generation and is not used for any final calculations (should always end up =1)

> data Param = Param {power :: Int, measures :: Int, ratio :: Rational}
>     deriving (Eq, Show)

Default parameter is a single whole note, of one full measure, with a normal ratio

> defParam = Param 0 1 1

Modifier functions:

> powFcn x p = p{power = power p + x}
> half = powFcn 1
> quarter = powFcn 2
> eighth = powFcn 3

> maxPow :: Param -> Bool
> maxPow p = power p == maxPoT

> toMaxPow :: Param -> Int
> toMaxPow p = maxPoT - power p

> mkRatio r p = p{ratio = ratio p * r}
> halveMeasures p = p{measures = (measures p) `div` 2}
```

Convenience method: returns Short if the subdivision about to be used will make the new notes the shortest possible

```
> shortIfMaxed x p = if toMaxPow p == x then Short else Beat
```

====================

Returns the highest power of two not greater than the argument

```
> potFloor :: Int -> Double
> potFloor x = 2.0^(truncate $ logBase 2 $ fromIntegral x)
```

subdivN is a convenience method that, given a Term's Param and a list of Ints, returns a list of Beats, Shorts, and Dotteds that represent dividing the parent Beat (whose Param is passed in) into smaller units, where a "1" in the list ends up with duration <Beat's duration>/<sum of the list>. In this way, any arbitrary tuplet or non-tuplet subdivision may be easily and concisely written without specifying anything other than the durations of its components. For example, subdivN p [1,1] divides a beat in half; subdivN p [1,1,1] divides it into a triplet; subdivN p [2,3] makes a quintuplet with the first note 2/5 the duration of the Beat and the second note 3/5 the duration. Any subdivisions which would result in a written note smaller than the maxPoT will return (Beat, p). However, actual durations smaller than 1/2^maxPoT (for example, triplet 16ths) may result.

```
> subdivN :: Param -> [Int] -> [Term RTerm Param]
> subdivN p xs = if toMaxPow p < pwr then [NT (Beat, p)] else map f xs where
>                    s = fromIntegral $ sum xs
>                    r = (toRational $ potFloor s) / (fromIntegral s)
>                    pwr = truncate $ logBase 2 $ potFloor s
>                    f n = NT (rterm n, mkRatio r $ powFcn (truncate $ logBase 2
(potFloor s / potFloor n)) p) where
>                        rterm x = case (fromIntegral x)/(potFloor x) of
>                                   1.0 -> shortIfMaxed (pwr `div` n) p
>                                   1.5 -> Dotted
>                                   _ -> error "subdivN: check the array; there is an
invalid value"
```

====================

The rules for "measure generation" determine what measures will be tied together. "letChance" determines how likely this is to happen.

```
> mRules :: Prob -> [Rule RTerm Param]
> mRules letChance = if (letChance < 0.0) || (letChance > 1.0) then error "mRules:
Chance is not within 0.0-1.0" else [
>             (Measure, 1-letChance) :-> \p -> if (measures p > 1) then [NT (Measure,
halveMeasures p), NT (Measure, halveMeasures p)] else [NT (Measure, p)],
>        (Measure, letChance) :-> \p -> if (measures p > 1) then [Let "x" [NT (Measure,
halveMeasures p)] [Var "x", Var "x"]] else [NT (Measure, p)]]
```

Rules for rhythmic subdivision.

```
> rRules :: Bool -> [Rule RTerm Param]
> rRules useLets = normalize ([
>             (Measure, 1.0) :-> \p -> [NT (Beat, p)],

>             (Beat, 0.2) :-> \p -> [NT (Beat, p)],
>             (Beat, 0.15) :-> \p -> subdivide p [2,1,1],
>             (Beat, 0.05) :-> \p -> subdivide p [1,1,1,1],
>             (Beat, 0.15) :-> \p -> subdivide p [3,1],
>             (Beat, 0.1) :-> \p -> subdivide p [1,1],
>             (Beat, 0.05) :-> \p -> subdivide p [1,2,1],
>             (Beat, 0.05) :-> \p -> subdivide p [1,1,1],

>             (Short, 1.0) :-> \p -> [NT (Short, p)],
>             (Dotted, 1.0) :-> \p -> [NT (Dotted, p)]
>             ] ++ if useLets then letRules else []) where
>       letRules = [
>             (Beat, 0.1) :-> \p -> [Let "x" [NT(shortIfMaxed 1 p, half p)] [Var "x",
Var "x"]],
>             (Beat, 0.1) :-> \p -> if toMaxPow p < 2 then [NT (Beat, p)] else
>                   [Let "x" [NT(shortIfMaxed 2 p, quarter p)] [Var "x", NT(Short,
half p), Var "x"]],
>             (Beat, 0.05) :-> \p -> if toMaxPow p < 2 then [NT (Beat, p)] else
>                   [Let "x" [NT(shortIfMaxed 2 p, quarter p)] [Var "x", Var "x", Var
"x", Var "x"]],
>             ]
```

=====================

Generation: fullGen s i m : s is the gen seed, i is the iteration to draw from, m is the length in measures (must be a multiple of two)

mGen / rGen no longer used

```
 mGen :: Int -> Int -> Int -> Sentence RTerm Param
 mGen s i m = snd $ gen (mRules 0.2) (mkStdGen s, [NT (Measure, Param 0 m 1)]) !! i

 rGen :: Int -> Int -> Sentence RTerm Param -> [(RTerm, Param)]
 rGen s i terms = toPairs $ snd $ gen (rRules True) (mkStdGen s, terms) !! i

> fullGen :: Int -> Int -> Int -> [(RTerm, Param)]
> fullGen s i m = toPairs $ snd $ gen (rRules True) ms !! i where
>                   ms = gen (mRules 0.25) (mkStdGen s, [NT (Measure, Param 0 m 1)])
!! (truncate $ logBase 2 $ fromIntegral m)

> addFinalBar :: [(RTerm, Param)] -> [(RTerm, Param)]
> addFinalBar xs = xs ++ [(Beat, Param 0 1 1)]

> transform :: [(RTerm, Param)] -> Music Pitch
> transform [] = rest 0
> transform xs =
>     let f (t, Param pow _ r) = note (0.5^pow*(modifier t)*r) (pitch 60) where
>         modifier x = case x of
>              Dotted -> 1.5
>              _ -> 1.0
>     in instrument Percussion $ foldr (:+:) (rest 0) $ map f xs

> addClick :: Music Pitch -> Music (Pitch, Volume)
> addClick m = addVolume 127 m /=: clicktrack where
>              clicktrack = addVolume 60 $ instrument Percussion $ foldr (:+:) (rest
0) $ map (\x -> note qn (pitch x)) clicks
>              clicks = (concat $ take (truncate $ dur m - 1) $ repeat [64,69,69,69])
++ [64]
```

Convenience: click track + final whole note

```
> transform' = addClick . transform . addFinalBar
```

Demo functions to try out different combinations of seed and iteration

```
> demo2 s i = play $ transform' $ fullGen s i 2
> demo4 s i = play $ transform' $ fullGen s i 4
> demo8 s i = play $ transform' $ fullGen s i 8
```

=========================

Generation experiments: playing around with update probabilities

```
> probs n g = map (\x -> (fromIntegral (x `mod` 1000) / 1000)^4) $ take n $ list n g
where
>                 list 0 _ = []
>                 list x g = let (a,s) = next g in a : list (x-1) s

> randomRules s = let rs = rRules True
>                     ps = probs (length rs) (mkStdGen s) in
>                 normalize $ updateProbs rs ps

> showRules :: [Rule RTerm Param] -> String
> showRules [] = ""
> showRules [r] = show (round $ (*1000) $ prob r)
> showRules (r:rs) = show (round $ (*1000) $ prob r) ++ "," ++ showRules rs
```

This is the most convenient method to use here: s1 and s2 are the seeds for rules and generation respectively.
i, m, and p are the index, length in measures, and Let-probability during measure gen

```
> randomRulesGen s1 s2 i m p = toPairs $ snd $ gen (randomRules s1) ms !! i where
>                           ms = gen (mRules p) (mkStdGen s2, [NT (Measure, Param 0 m
1)]) !! (truncate $ logBase 2 $ fromIntegral m)
```

Cleaner list presentation

```
> present :: [(RTerm, Param)] -> String
> present [] = "";
> present ((t, p):xs) = f t p ++ "  " ++ present xs where
>                           f t p = show $ 2^(maxPoT-power p)*l*(ratio p)
>                           l = case t of
>                               Dotted -> 1.5
>                               _ -> 1.0
```