

SMURF

Serial MUsic Represented as Functions

Richard Townsend, Lianne Lairmore, Lindsay Neubauer, Van Bui, Kuangya Zhai
{rtownsend, lairmore, neubauer, vbui, kyzhai}@cs.columbia.edu

September 23, 2013

1 Background: What is Serialism?

In general, serialism is a musical composition technique where a set of values, chosen through some methodical process, generates a sequence of musical elements. Its origins are often attributed to Arnold Schoenberg's twelve-tone technique, which he began to use in the 1920s. In this system, each note in the chromatic scale is assigned an integer value, giving us a set of twelve "pitch classes" (Figure 1). A composer utilizing this method then takes each of these integers, and orders them into a *twelve tone row*, where each number appears exactly once. We refer to this row as the *primeform* of a piece, and conventionally refer to it as P_0 .

The composer can then generate other rows that are derived from P_0 through three types of transformations: transposition, inversion, and retrograde. In each of these transformations, we always use mod 12 arithmetic to preserve the numbering system of our pitch classes. Transposing a row consists of taking each pitch class in the row and adding the same number to each. If we transpose P_0 by four semitones, we add four mod twelve to each pitch class in P_0 and end up with a new row called P_4 . In general, P_x is a transposition of P_0 by x semitones. To invert a row, we flip each interval between two pitch classes in that row. For example, if P_0 starts with pitch classes 0-4-1, then we have an interval of +4 between the first two pitches and -3 between the second two. In the inverse of P_0 (called I_0), the first interval would be -4 and the second would be +3, giving us 0-8-11 as our first three pitch classes. The subscript of I_x refers both to the number of transpositions required to arrive at I_x from I_0 , and to the prime row P_x that would need to be inverted to generate I_x . The final row operation is a retrograde transformation, which merely consists of reading a row backwards. That is, R_x is generated by reading the pitch classes of P_x in their opposite order. One can also have a retrograde inversion; RI_x is generated by reading the pitch classes of I_x backwards.

Once a composer chooses a P_0 , the three transformations outlined above can be applied to varying degrees to generate a *twelve tone matrix*, which will contain each P row as a row in the matrix and each I row as a column. Furthermore, all of the R and RI rows are found by reading the rows in the matrix from right to left or the columns from bottom to top, respectively. An example of a twelve tone matrix from one of Schoenberg's pieces can be found below (Figure 2). Finally, using the twelve tone matrix as a guide, the composer picks various rows and columns to serve as melodic and harmonic elements in their composition, resulting in a piece of serial music.

(Original image: <http://www.music-mind.com/Music/Srm0038.GIF>)

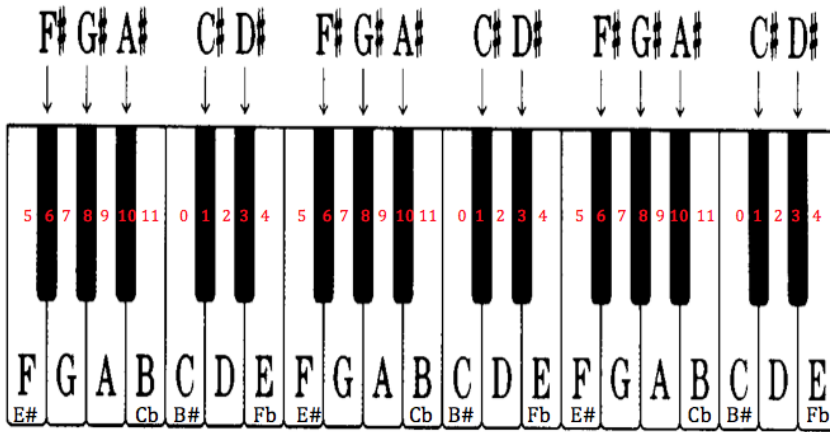


Figure 1: pitch classes

	I_0	I_{11}	I_7	I_8	I_3	I_1	I_2	I_{10}	I_5	I_4	I_9	
P_0	0	11	7	8	3	1	2	10	6	5	4	9
P_1	1	0	8	9	4	2	3	11	7	6	5	10
P_5	5	4	0	1	8	6	7	3	11	10	9	2
P_4	4	3	11	0	7	5	6	2	10	9	8	1
P_9	9	8	4	5	0	10	11	7	3	2	1	6
P_{11}	11	10	6	7	2	0	1	9	5	4	3	8
P_{10}	10	9	5	6	1	11	0	8	4	3	2	7
P_2	2	1	9	10	5	3	4	0	8	7	6	11
P_6	6	5	1	2	9	7	8	4	0	11	10	3
P_7	7	6	2	3	10	8	9	5	1	0	11	4
P_8	8	7	3	4	11	9	10	6	2	1	0	5
P_3	3	2	10	11	6	4	5	1	9	8	7	0

Figure 2: twelve tone matrix

2 Motivation

Our group has decided to create a language that will assist a composer in creating music using the twelve tone method described above. We plan on making this language functional and compile into C. The compilation process will create a C program that will use OpenGL to create an image representing the music created.

Twelve tone serialism is a mathematically intensive method of creating music which involves mapping notes to numbers. It is very natural to work with twelve tone rows using a programming language since the method just treats notes like numbers that can be added and subtracted from. We plan on our language making twelve tone compilation easier using data types and functions specifically for the purpose of creating music in this way. By simplifying the method of inverting and transposing rows composers can focus more on how to exploit new ways to make music in this fashion and worry less about creating matrices.

We chose to implement our language as a functional language because of the clear and succinct programs that functional languages produce. It also makes sense for our language to be functional because functional languages also are well known for their ability to work on lists and most arithmetic done in twelve tone serialism is done on rows or columns. As a group we are also very interested on how a functional language compiler works.

Instead of compiling our language into byte code which can be interpreted and play the notes we thought it would be more interesting to create a language that could be compiled into a score. Scores are made up of just lines and dots and would be easy to create in a graphics library like OpenGL. We decided to compile our language into C since it was a significantly less abstract language which had access to OpenGL library calls. One benefit of compiling into C and not a language to be interpreted as music is that programs in our language could be created that do not actually output any music but instead create functions that could be used in other programs to do the same transformation of rows. That way our language is not just limited to functions defined in our libraries or what is in the current file.

Overall we hope to use the simplicity of a functional language to help composers write complex, new, and interesting music based on twelve tone serialism. These new compositions would then be able to be printed and handed to musicians to play. This simplifies the composers task of converting music in computer format to one which musicians will be able to read easily.

3 Syntax

SMURF is a functional programming language loosely modeled off of Haskell. It has immutable memory, no global variables and no I/O.


3.1 Types

3.1.1 Standard Types

- Integer: `int`
- Boolean: `bool`
- Tuples: elements can have different types
- Lists: elements must have same type

3.1.2 Note (`pc: int, beat: int, register: int`)

- `pc` (pitch class): represented by integers 0-11
- `beat`: represented by powers of 2 up to 32 (assuming $\frac{4}{4}$ time)
- `register`: represented by integers -2 to 2

–  Treble Clef: notes middle C and higher represented by 0-2

–  Bass Clef: notes lower than middle C represented by negative numbers

3.1.3 Chord (`[Note]`)

- Type checks that all notes have same beat count

3.1.4 Figure (`[Chord]`)

3.1.5 Functions

A function is a type whose value can be defined with an expression

- Function declarations must declare type (can declare general type)
- Function declarations must be on own line
- No explicit return
- Pattern matching, guards, and if-then-else clauses used
 - Each pattern matching pattern must be on own line
 - Guards and if-then-else clauses do not have newline restrictions

4 Examples

dummy contents...

Operators	
\n	End of line: Terminates phrases
+	Integer arithmetic: plus
-	Integer arithmetic: minus
%	Integer arithmetic: modulus, ignores negatives
<	Comparison
>	Comparison
<=	Comparison
>=	Comparison
==	Structural comparison
not	Boolean operator
&&	Boolean operator
	Boolean operator
++	Concatenation: concat
:	Construction: cons
/* */	Multiline comments, nesting allowed
//	Single-line comment

Keywords	
let	Specify values and functions
->	Specify arguments and return type of functions
::	Specify types
if, then, else	Specify conditional expression, else compulsory
	Specify boolean expression, used with guards