TITLE GOES HERE

By

LUIS F. VIEIRA DAMIANI

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2018

Dedication goes here.

ACKNOWLEDGMENTS

Acknowledgments go here.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

TITLE GOES HERE

By

Luis F. Vieira Damiani

August 2018

Chair: Dr. Beverly Sanders
Major: Computer Science

Abstract goes here.

CHAPTER 1
INTRODUCTION

Formal languages lend precision and flexibility to music specification because they require that musical ideas be turned into abstract symbols and stipulated explicitly.

Herein lies both the advantage and disadvantage of linguistic interaction with a computer music system.

The advantage is that formalized and explicit instructions can yield a high degree of control.

To create an imagined effect, composers need only specify it precisely.

They can easily stipulate music that would be difficult or impossible to perform by human beings.

In some cases, a linguistic specification is much more efficient than gestural input would be.

This is the case when a single command applies to a massive group of events, or when a short list of commands replaces dozens of pointing and selecting gestures.

The shell scripts of Unix operating systems are a typical example of command lists (Thompson and Ritchie 1974).

These advantages turn into a disadvantage when simple things must be coded in the same detail and with the same syntactic overhead as complicated things.

For example, with an alphanumeric language, envelope shapes that could be drawn on a screen in two seconds must be plotted out on paper by hand and transcribed into a list of numerical data to be typed by the composer.

For many tasks, graphical editors and visual programming systems, in which the user selects and interconnects graphical objects, are more effective and easier to use than their textual counterparts (see chapter 16).

Some languages are interactive; one can type individual statements and each of them is interpreted in turn.

This can occur in a concert situation, but the slow information rate of typing not to mention the mundane stage presence of a typist precludes this approach in fastpaced realtime musicmaking.

Gestural control through a musical input device is more efficient and natural.

Hence, languages for music, although important, do not answer all musical needs.

In the ideal, music languages should be available alongside other kinds of musical interaction tools.

[**?** , 785-786].

Textual languages are a precise and flexible means of controlling a computer music system.

In synthesis, score entry, and composition, they can liberate the potential of a music system from the closed world of canned software and preset hardware.

In the mid-1980's it looked as if the Music N synthesis language dynasty might languish, due to the spread of inexpensive synthesizers.

In MIDI systems, however, the use of a score language is less common, since most music can be entered by other means (such as a music keyboard, notation program, or scanner).

For musicological applications that involve score analysis, however, a text-based score representation may be optimum.

New textual languages for procedural composition continue to be developed, but there is a strong parallel trend toward interactive programs with graphical interfaces.

Instead of typing text, one patches together icons, draws envelopes, and fills in templates.

The textual language representation supporting the graphics is hidden from the user.

Early programming languages for music tended to favor machine efficiency over ease of use.

The present trend in programming has shifted from squeezing the last drop of efficiency out of a system to helping the user manage the complexity of layer upon layer of software and hardware.

The most common solution to this problem is object-oriented programming (see chapter 2), and compositional applications are no exception to this trend (Pope 1991b).

[**?** , 817].

CHAPTER 2
LITERATURE REVIEW

Arguably the first notable attempt to design a programming language with an explicit intent of processing sounds and making music was that of *Music I*, created in 1957 by Max Mathews. The language was indented to run on an IBM 704 computer, located at the IBM headquarters in New York City. The programs created there were recorded on digital magnetic tape, then converted to analog at Bell Labs, where Mathews spent most of his career as an electrical engineer. *Music I* was capable of generating a single waveform, namely a triangle, as well as assigning duration, pitch, amplitude, and the same value for decay and release time. *Music II* followed a year later, taking advantage of the much more efficient IBM 7094 to produce up to four independent voices chosen from 16 waveforms. With *Music III*, Mathews introduced in 1960 the concept of a *unit generator*, which consisted of small building blocks of software that allowed composers to make use of the language with a lot less effort and required background. In 1963, *Music IV* introduced the use of macros, which had just been invented, although the programming was still done in assembly language, hence all implementations of the program remained machine-dependent. With the increasing popularity of Fortran, Mathews designed *Music V* with the intent of making it machine-independent, at least in part, since the unit generators' inner loops were still programmed in machine language. The reason for that is the burden these loops imposed on the computer [**?** , 15-17].

2.1 Software Synthesis Languages

Since Mathews' early work, much progress has been made, and a myriad of new programming languages that support sound processing, as well as domain-specific languages whose sole purpose is to process sounds or musical events, have surfaced. In [**?**], we see an attempt to classify these languages according to the specific aspect of sound processing they perform best. The first broad category described is that of *software synthesis languages*, which compute samples in non-real-time, and are implemented by use of a text editor with a general purpose computer. The *Music N* family of languages consist of software synthesis

languages. A characteristic common to all software synthesis languages is that if a toolkit approach to sound synthesis, whereby using the toolkit is straightforward, however customizing it to fulfill particular needs often require knowledge of the programming language in which the toolkit was implemented. This approach provides great flexibility, but at the expense of a much steeper learning curve. Another aspect of software synthesis languages is that they can support an arbitrary number of voices, and the time complexity of the algorithms used only influences the processing time, not the ability to process sound at all, as we see with real-time implementations. As a result of being non-real-time, software synthesis languages usually lack controls that are gestural in nature. Yet, software synthesis languages are capable of processing sounds with a very fine numerical detail, although this usually translates to more detailed, hence verbose code. Software synthesis languages, or non-real-time features of a more general-purpose language, are sometimes required to realize specific musical ideas and sound-processing applications that are impossible to realize in real time [**?** , 783-787].

Within the category of software synthesis languages, we can further classify those that are *unit generator languages*. This is exactly the paradigm originally introduced by *Music III*. In them, we usually have a separation between an orchestra section, and a score section, often given by different files and sub-languages. A unit generator is more often than not a built-in feature of the language. Unit generators can generate or transform buffers of audio data, as well as deal with how the language interacts with the hardware, that is, provide sound input, output, or print statements to the console. Even though one can usually define unit generators in terms of the language itself, the common practice is to define them as part of the language implementation itself. Another characteristic of unit generators is that they are designed to take as input arguments the outputs of other unit generators, thus creating a signal flow. This is implemented by keeping data arrays in memory which are shared by more than one UG procedure by reference. The score sub-language usually consists of a series of statements that call the routines defined by the orchestra sub-language in sequential order, often without making use of control statements. Another important aspect of the score sub-language is that

it defines function lookup tables, which are mainly used to generate waveforms and envelopes. When *Music N* languages became machine-independent, function generating routines remained machine-specific for a period of time, due to performance concerns. On the other hand, the orchestra sub-language is where the signal processing routines are defined. These routines are usually called instruments, and basically consist of new scopes of code where built-in functions are dove-tailed, ultimately to a unit generator that outputs sound or a sound file [**?** , 787-794].

The compilation process in *Music N* languages consists usually of three passes. The first pass is a preprocessor, which optimizes the score that will be fed into the subsequent passes. The second pass simply sorts all function and instrument statements into chronological order. The third pass then executes each statement in order, either by filling up tables, or by calling the instrument routines defined in the orchestra. The third pass used to be the performance bottleneck in these language implementations, and during the transition between assembly and Fortran implementations, these were the parts that remained machine-specific. Initially, the output of the third pass consisted of a sound file, but eventually this part of the compilation process was adapted to generate real-time output. At that point, defining specific times for computing function tables became somewhat irrelevant.

In some software synthesis languages, the compiler offers hooks in the first two passes so that users can define their own sound-processing subroutines. In any cases, these extensions to the language were given in an altogether different language. With *Common Lisp Music*, for example, one could define the data structures and control flow in terms of Lisp itself, whereas *MUS10* supported the same features by accepting Algol code. In *Csound*, one can still define control statements in the score using Python. Until *Music IV* and its derivatives, compilation was sample-oriented. As an optimization, *Music V* introduced the idea of computing samples in blocks, where audio samples maintained their time resolution, but control statements could be computed only once per block. Of course, if the block size is one, than we compute control values for each sample, as in the sample-oriented paradigm. Instead of defining a block size, however, one defines a control rate, which is simply the sampling rate times the reciprocal of

the block size. Hence a control rate that equals the sampling rate would indicate a block size of one. With *Cmusic*, for instance, we specify the block size directly, a notion that is consistent with the current practice of specifying a vector size in real-time implementations. The idea of determining events in the language that could be computed at different rates required some sort of type declaration. In *Csound*, these are given by naming conventions: variables whose names start with the character 'a' are audio-rate variables, 'k' means control rate, and 'i'-variables values are computed only once per statement. *Csound* also utilizes naming conventions to determine scopes, with the character 'g' indicating whether a variable is global [**?** , 799-802].

2.2 Real-Time Synthesis Control Languages

Some of the very first notable attempts to control the real-time synthesis hardware were made at the *Institut de Recherche et Coordination Acoustique/Musique* in the late seventies. Many of these early attempts made use of programming languages to drive the sound synthesis being carried out by a dedicated DSP. At first, most implementations relied on the concept of a *fixed-function* hardware, which required significantly simpler software implementations, as the latter served mostly to control a circuit that had an immutable design and function. An example of such fixed-function implementations would be an early frequency-modulation synthesized, which contained a dedicated DSP for FM-synthesis, and whose software implementation would only go as far as controlling the parameters thereof. Often, the software would control a chain of interconnected dedicated DSP's, which would in turn produce envelopes, filters, and oscillators. The idea of controlling parameters through software, while delegating all signal processing to hardware, soon expanded beyond the control of synthesis parameters, and into the sequencing of musical events, like in the New England Digital Synclavier. Gradually, these commercial products began to offer the possibility of changing how exactly this components were interconnected, what is called a *variable-function* DSP hardware. Interconnecting these components through software became commonly called *patching*, as an analogy to analog synthesizers. The idea of patching brought more flexibility,

but imposed a steeper learning curve to musicians. Eventually, these dedicated DSP's were substituted by general-purpose computers, wherein the entire chain of signal processing would be accomplished via software [**?** , 802-804].

Commonly in a fixed-function implementation there is some sort of front panel with a small LCD, along with buttons and knobs to manage user input. In the case of a keyboard instrument, there is naturally a keyboard to manage this interaction, as well. The purpose of the embedded software is then to communicate user input to an embedded system which contains a microprocessor and does the actual audio signal processing, memory management, and audio input/output. All software is installed in some read-only memory, including the operating system. With the creation of the *Musical Instrument Digital Interface* standard in 1983, which was promptly absorbed my most commercial brands, the issue of controlling sound synthesis hardware transcended the interaction with keys, buttons, and sliders, and became a matter of software programming, as one could easily communicate with dedicated hardware, by means of a serial interface, MIDI messages containing discrete note data, continuous controller messages, discrete program change messages, as well as system-exclusive messages. As a trend, many MIDI libraries were written at the time for general-purpose programming languages such as APL, Basic, C, Pascal, Hypertalk, Forth, and Lisp. In addition, most descendants of the *Music N* family of languages began to also support MIDI messages as a way to control dedicated hardware [**?** , 804-805].

The implementation of a software application to control variable-function DSP hardware is no mundane task, as it requires knowledge of digital signal processing, in addition to programming in a relatively low level language. Dealing with issues of performance, memory management, let alone the mathematics required to process buffers of audio samples, often imposes an unsurmountable burden to musicians. Many solutions were invented in order to work around this difficulties, including the use of graphic elements and controllers, but ultimately it was the concept of a unit generator, borrowed from software synthesis languages, that most influenced the creation of higher-level abstractions that were more suitable for

musicians. This is notably the case of the *4CED* language, which was developed at IRCAM in 1980, and owed greatly to *Music IV* and *Music V*. The resemblance extended as far as to comprise a separate orchestra sub-language for patching unit generators, a score sub-language, and a third command sub-language for controlling effects in real-time, as well as to link both orchestra and score to external input devices such as buttons and potentiometers. The hardware these languages drove was IRCAM's 4C synthesizer. The result of nearly a decade of research at IRCAM culminated in *Max*, a visual programming language that remains to this day one of the most important real-time tools for musicians. *Max*, which will later be discussed in more detail, eventually transcended its hardware DSP and implemented itself in C the sound-generating routines. But that was not until the 2000's, ten years after it became a commercial software application, independent of IRCAM [**?** , 805-806].

Example 2.2.1. *[**?** , 809]* Music 1000 *is a descendant of the* Music N *family of languages that was designed to drive the Digital Music Systems DMX1000 signal processing computer, in which we can clearly observe the unit-generator concept in action: In the code above, a* fnctn

Algorithm 1: *Music 1000* algorithm that produces a sine wave.

```
1 fnctn func1, 512, fourier, normal, 1, 1000
2 instr 1
3     kscale amp, knob1, 0, 10000
4     kscale freq, knob2, 20, 2000
5     oscil x8, #func1, amp, freq
6     out x8
7 endin
```

statement assigns to variable func1 *an array of 512 samples using a* fourier *series of exactly one harmonically-related sine, whose (trivial) sum is* normal-*ized. The amplitude of 1000 is then meaningless, but a required argument. In fact,* func1 *takes a variable number of arguments, where for each harmonic partial, the user specifies a relative amplitude. The block that follows defines an instrument, in which the unit generator* oscil *takes as arguments the output of three other unit generators, which are respectively the wavetable previously computed, as well as amplitude and frequency parameters, whose values are in turn captured by two knobs attached*

to the machine. The knobs produce values between 0 and 1, and the subsequent arguments to kscale *are scaling parameters. Finally,* out *is a unit generator that connects the output of* oscil *to the digital-to-analog converter.*

2.3 Music Composition Languages

Between the 1960's and the 1990's, many programming languages were devised to aid music composition. As a noticeable trend, one can define two categories among those languages, namely those that are *score input languages*, and those that are *procedural languages*. The main difference between the two categories is that, in the former, some representation of a musical composition is already at hand, hence score input languages provide a way to encode that information. This could be a score, a MIDI note list, or even some graphical representation of music. In the latter category, the language provides, or helps define procedures that are used to generate musical material, a practice that is often called *algorithmic* music composition. One outstanding characteristic of score input languages is how verbose and complex they can become, depending on the musical material they are trying to represent. This difficulty influenced the devising of many alternatives to textual programming languages, such as the use of scanners in the late 1990's by Neuratron's *PhotoScore*, an implementation which was predicted by composer Milton Babbitt as early as in 1965. Before the advent of MIDI, however, programming languages were indeed the user interface technology of choice, or lack thereof, to design applications meant for analyzing, synthesizing, and printing musical scores. With the widespread adoption of the MIDI standard in the mid-1980's, whereby one can input note events by performing on a MIDI instrument, combined with the advancements in graphical user interfaces of the mid-1990's, the creation and maintenance of score input languages has faced a huge decline. What is even worse, the paradigm of a musical score is itself inadequate for computer music synthesis, in that a score is more often than not a very incomplete representation of a musical piece, often omitting a great deal of information. It is the job of a musical performer to provide that missing information. In this sense, *procedural languages* are much

better suited for computer performance, but that comes at the cost of replacing the score paradigm altogether [**?** , 811-813].

In 2018, a few *score input languages* remain, despite the vast predominance of graphical user interfaces as a means to input notes to a score. *MusiXTex* is a surviving example that compiles to LaTeX, which in turn compiles to PDF documents. It was created in 1991 by Daniel Taupin. The language has such unwieldy syntax, that often a preprocessor is required for more complex scores. One famous such processors is *PMX*, a *FORTRAN* tool written by Don Simons in the late 1990's. Another was *MPP*, which stands for MusiXTex Preprocessor, created by Han-Wen Nienhuys and Jan Nieuwenhuizen in 1996, and which eventually became *LilyPond*, arguably the most complete surviving score input language today. *LilyPond* has a much simpler syntax than that of *MusiXTex*, however not nearly as simple as *ABC* music notation, a language that much resembles *Musica* and which is traditionally used in music education contexts. A package written by Guido Gonzato is available in LaTeXwhich can produce simple scores in *ABC* notation. Its simplicity comes, however, at the expense of incompleteness. Finally, it is worthwhile to mention a music-notation specific standard that has emerged in the mid-2000's, namely the *MusicXML* standard. Heavily influenced by the industry, it was initially meant as an object model to translate scores between commercial applications where the score input method was primarily graphical, and whose underlying implementation was naturally object-oriented. *MusicXML* is extremely verbose, and borderline human-readable. It is, however, very complete, to the point of dictating what features an object-oriented implementation should comprise in order to be aligned with the industry standards. In recent years, many rumors have surfaced to make *MusicXML* an Internet standard, such as that of *Scalable Vector Graphics*, however nothing concrete has been established.

Example 2.3.1. *[**?** , 812]* Musica *was developed at the Centro di Sonologia Computazionale in Padua, Italy, and is particularly interesting in its interpreter compiles programs into* Music V *note statements.*

Algorithm 2: *Musica* algorithm that creates a simple melodic line.

```
1 4'AGAG / 4.A8G2E / 4DDFD / 2ED
```

In the code above, all numbers indicate note duration, that is, 4 is a quarter-note, 8 is an eighth-note, and 2 is half-note, with dots indicating dotted durations. The letters indicate pitch, and the apostrophe indicates octave such that 'A $= 440Hz$*, and "A* $= 880Hz$*. Finally, the slash indicates a measure. The code below, on the other hand, shows an example of the very same musical material expressed in* MusiXTex*. One can immediately notice the difference in implementation by the sheer amount of code required to express basically the same symbols.*

Algorithm 3: *MusiXTex* algorithm whose output is shown in Fig. **??**.

```
1 \begin{music}
2     \generalmeter{\meterfrac44}
3     \startextract
4     \Notes \qu{h g h g} \en \bar
5     \Notes \qup{h} \cu{g} \hu{e} \en \bar
6     \Notes \qu{d d f d} \en \bar
7     \Notes \hu{e d} \en
8     \endextract
9 \end{music}
```

In the snippet above, many commands, despite verbose, are quite self-explanatory. Some others, however, are not. The \qu *command means a quarter-note with a stem pointing upward, whereas the* \Notes *command actually means how notes should be spaced. The more capital letters, the more spacing between the notes, that is,* \NOTes *is more spaced out than* \NOtes*. Finally, in addition to supporting the same apostrophes as* Musica *for defining octave,* MusiXTex *also supports other letters, as well as capitalizations thereof. In the example above, we have* $h = 440Hz$*, whereas* $a = 220Hz$*.*

One of the greatest contributions of *procedural composition languages* to the field of music composition is arguably the concept of algorithmic composition, in particular when the realization of the musical algorithm is not restricted to human performers. In such circumstances, the composer is capable of exploring the full extent of musical ideas a computer

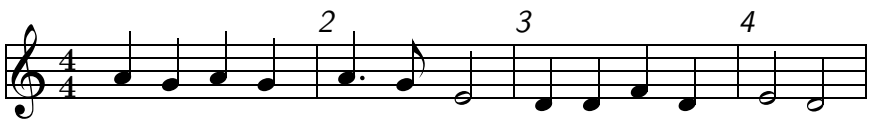

Figure 2-1. Typesetting music with *MusiXTex*.

can reproduce. Naturally, the composer must often trade off the ability to represent those ideas via a score, in which case the algorithm itself becomes the representation. If, on one hand, reading music from an algorithm is somewhat unfamiliar to most musicians, the representation is nonetheless formal, concise, and consistent. Furthermore, it lends itself to be analyzable a much larger apparatus of analytical techniques and visualization tools, hence is equally beneficial a representation to music theorists. A machine is capable of representing all sorts of timbres, metrics, and tunings that humans cannot, but it needs to be told exactly what to do. Unlike a human performer, who interprets the composer's intents, a purely electro-acoustic algorithmic composition must address a human audience without relying on a middle-man. Hence the programming language of choice becomes an invaluable tool for the composer. In addition to all that, another important aspect of algorithmic composition is how it is capable of transforming the decision-making process of a composer. Instead of making firm choices at the onset of a musical idea, a composer can *prototype* many possible outcomes of that idea before deciding. One example is assigning random numbers to certain parameters, this postponing the decision making until more structure has been added to the composition. In fact, this postponing may be final, thus an algorithmic composition may be situated within a whole spectrum of determinism. A fully stochastic piece fixes no parameter, as opposed to a fully deterministic composition. Some of the notable techniques of electro-acoustic music composition also include spatialization, where the emission of sounds through speakers positioned at specific spatial locations constitutes a major musical dimension in a composition; spectralism, where the spectral content of sounds are manipulated by an algorithm; processing sound sources in real time, very often capturing a live performance on stage; and sonification of data nor originally conceived as sound [**?** , 813].

2.4 Libraries

Many domain-specific languages that deal with sound synthesis, processing, and music composition are *extensible* in the sense that they provide a hook for code written in the implementation language to be executed in the context of the DSL. This feature can render a DLS a lot more flexible, at the expense of annulling the very purpose of the DSL, which can be a good trade-off if the latter's implementation is incomplete. An early example would be *Music V*, which could accept user-written subroutines in *Fortran*. *Music 4C* had its instruments written in *C*, and *Cscore* was a *C*-embedding of *Cmusic*. Other examples are *MPL*, which could accept routines written in *APL*, and *Pla*, whose first version was embedded in *Sail*, and whose second version was embedded in *Lisp*. In the particular case of *Lisp*, embeddings include *MIDI-LISP*, *FORMES*, *Esquisse*, *Lisp Kernel*, *Common Music*, *Symbolic Composer*, *Flavors Band*, and *Canon*. *Music Kit* was embedded in the object-oriented *Objective-C* [**?** , 814].

Besides domain-specific languages, a variety of libraries exist for general-purpose programming languages that also deal with aspects of sound synthesis, processing, and music composition. In languages like *Haskell*, these libraries may carry such syntactical weight, with so many specifically-defined symbols, that they do in fact resemble more a DSL that a library, even though such terming would not be technically correct.

APPENDIX A
APPENDIX A

Table A-1. Unit-generator based software synthesis languages [**?** , 789-790].

Application	Year	Authors	Platform	Language
Music III	1960	M. Mathews	IBM 7090	Assembler
Music IV	1963	M. Mathews J. Miller	IBM 7094	Macro assembler
Music IVB	1965	G. Winham H. Howe	IBM 7094	Macro assembler
Music V	1966	M. Mathews J. Miller	GE 645	Fortran IV
MUS10	1966	J. Chowning D. Poole L. Smith	DEC PDP-10	PDP-10 assembler
MUSIGOL	1966	D. MacInnes W. Wulf P. Davis	Burroughs 5500	Burroughs Algol
Music 4BF	1967	H. Howe G. Winham	IBM 360	Fortran II and BAL assembler
Music 360	1969	B. Vercoe	IBM 360	BAL assembler
Music 7	1969	H. Howe	Xerox XDS Sigma 7	Assembler
TEMPO	1970	J. Clough	IBM 360	BAL assembler
B6700 Music V	1973	B. Leibig	Burroughs 6700	Fortran and Algol
Music 11	1973	B. Vercoe S. Haflich R. Hale H. Howe	DEC PDP-11	Macro-11 assembler
MUSCMP	1978	Tovar	Foonly 2 DEC PDP-10	FAIL assembler
Cmusic	1980	F. R. Moore D. G. Loy	DEC VAX-11	C
Cmix	1984	P. Lansky	DEC PDP-11	C
Music 4C	1985	S. Aurenz J. Beauchamp R. Maher C. Goudeseune	DEC VAX-11	C
Csound	1986	B. Vercoe R. Karstens	DEC VAX-11	C
Music 4C	1988	G. Gerrard	Macintosh	C
Common Lisp Music	1991	W. Schottstaedt	NeXT	Common Lisp

Table A-2. Unit-generator based languages for control of real-time DSP [**?** , 807-808].

Application	Year	Authors	Platform	Language	DSP
4B	1978	D. Bayer	DEC LSI-11	Assembler	IRCAM 4B
SYN4B	1978	P . Prevot N. Rolnick	DEC LSI-11	Assembler	IRCAM 4B
4PLAY	1978	C. Abbott	DEC PDP-11	Pascal	IRCAM 4C
Musbox	1979	G. Loy W. Schottstaedt	DEC PDP-10	Sail	Samson Box
4CED	1980	C. Abbott	DEC PDP-11	C	IRCAM 4C
Music 1000	1980	D. Wallraff	DEC LSI-11	Assembler	DMX-1000
4X	1981	J. Kott	DEC PDP-11	Assembler	IRCAM 4X
FMX	1982	C. Abbott	DEC VAX-11	C	Lucasfilm ASP
Music 400	1982	M. Puckette	DEC PDP-11	C	Analogic AP-400
Cleo	1983	C. Abbott	Sun	C	Lucasfilm ASP
Music 320	1983	T. Hegg	MC68000	Assembler	TI TMS 32010
Music 500	1984	M. Puckette	DEC VAX-11	C	Analogic AP-500
4XY	1986	R. Rowe O. Koechlin	DEC VAX-11	C	IRCAM 4X
Csound	1989	N. Bailey A. Purvis I. Bowler P. Manning	Inmos Transputer	Occam	Inmos Transputer
Music 56000	1989	K. Lent R. Pinkston P. Silsbee	IBM PS/2	Assembler	Motorola DSP56001
NeXT Music and Sound Kits	1989	D. Jaffe L. Boynton J. Smith	NeXT	Objective-C	Motorola DSP56001
Digital Signal Patcher	1990	A. Pellecchia	IBM PC	C	TI TMS320C30
MUSIC30	1991	J. Dashow	IBM PC	Prolog	TI TMS320C30
IRCAM Max	1991	M. Puckette	NeXT	C	Intel i860
IRIS Edit20	1992	P . Andenacci E. Favreau N. Larosa A. Prestigiacomo C. Rosati S. Sapir	Atari SM1000	C	MARS
Unison	1992	J. Bate	Apple Macintosh	C C	Motorola DSP56001

Table A-3. Score input languages [**?** , 811].

Language	Reference	Comments
DARMS	Erickson 1975	Score analysis and archival
MUSTRAN	Wenker 1972	Score analysis and archival
Standard Music	Newcomb and	Score analysis and archival
Description Language	Goldfarb 1989	
SCORE	Smith 1972	Score synthesis and printing
Musica	De Poli 1978	Score synthesis and printing
SCRIPT	New England Digital 1981	Score synthesis and printing
Scriptu	Brown 1977	Score synthesis and printing
ASHTON	Ames 1985	Score synthesis and printing
Notepro	Beauchamp, Code and Chen 1990	Score synthesis and printing
Yet Another Music Input Language	Fry 1980	Score synthesis and printing

Table A-4. Procedural music composition languages [**?** , 815-817].

Language	References	Comments
MUSICOMP	Baker 1963, Hiller and Leal 1966	
GROOVE	Mathews and Moore 1970	
TEMPO	Clough 1970	
SCORE	Smith 1972	Preprocessor for Music V and MUS10
PLAY	Chadabe and Meyers 1978	
Tree and Cotree	Roads 1978b	
GGDL	Holtzman 1981	
MPL	Nelson 1977, 1980	Support for MIDI
Pla	Schottstaedt 1983, 1989a	
SAWDUST	Blum 1979, Hamlin and Roads 1985	
PILE	Berg 1979	
Flavors Band	Fry 1984	
FORMES	Rodet and Cointe 1984	
Formula	Andersen and Kuivila 1986, 1991	Forth-based with real-time support
MIDI-LISP	Boynton et al. 1986	Support for MIDI
HMSL	Polansky, Rosenboom, and Burk 1987, Polansky et al. 1988	Forth-based with MIDI support
Personal Composer Lisp	Miller 1985	Integrated MIDI sequencer
Player	Loy 1986	
LOCO	Desain and Honing 1988	Logo-based
COMPOSE	Ames 1989b	
Canon	Dannenberg 1989a	Lisp-based
Hyperscore	Pope 1986a, b	Object-oriented
MODE	Pope 1991a	Object-oriented
Music Kit	Jaffe and Boynton 1989	
Lisp Kernel	Rahn 1990	
Keynote	Thompson 1990	Support for MIDI
Arctic	Dannenberg, McAvinney, and Rubine 1986	Functional
Esquisse	Baisnee et al. 1988	Support for data structures
Moxc	Dannenberg 1989b	
Common Music	Taube 1991	
PatchWork	Laurson and Duthen 1989, Malt 1993, Barriere, Iovino, and Laurson 1991	Graphical patching of Lisp functions
Symbolic Composer	Tonality Systems 1993	Lisp-based with MIDI output

REFERENCES

[1] C. Roads, *The Computer Music Tutorial* (The MIT Press, 1995).

[2] C. Roads, M. Mathews, *Computer Music Journal* **4**, 15 (1980). Available from: http://www.jstor.org/stable/3679463.

BIOGRAPHICAL SKETCH

Bio goes here.