Preliminary Report On
Design Rationale, Syntax, and Semantics of LSL:
A Specification Language for Program Auralization[*]

David B. Boardman   Aditya P. Mathur

September 21, 1993

Abstract

The need for specification of sound patterns to be played during program execution arises in contexts where program auralization is useful. We present a language named LSL (**L**isten **S**pecification **L**anguage) designed for specifying program auralization. Specifications written in LSL and included in the program to be auralized are preprocessed by an LSL preprocessor. The preprocessed program when compiled and executed generates MIDI or voice data sent through a MIDI interface to a synthesizer module, or via audio channels, to an audio processor, which transforms the notes or voice into audible sound. LSL has the generality to specify auralization of a variety of occurrences during program execution. It derives its broad applicability from its few generic elements that when adapted to any procedural programming language, such as C, C++, or Ada, enable the writing and use of LSL specifications for auralizing sequential, parallel, or object oriented programs in that language. We view LSL as a useful tool for building general purpose multimedia applications and for research in program auralization.

Contents

List of Figures

List of Tables

1 Introduction

The idea of using sound to understand program behavior and to analyze data using sound has been reported by several researchers. By program and data auralization we refer to, respectively, the activity of mapping aspects of program execution or properties of data to sound[1]. A survey of the use of sound in understanding program behavior or analyzing data collected from experiments or generated by a program is reported by Francioni and Jackson[7]. It is believed, and has been demonstrated in a few cases, that the use of sound can enhance program understanding. Yeung[18] proposed the use of sound to recognize multivariate analytical data. Francioni and Jackson[7] used program auralization to understand the runtime behavior of parallel programs. Brown and Hershberger[4] auralized some of their animations generated using the Zeus animation system. In his doctoral dissertation, Edwards[6] built and evaluated a word processor with an audio interface for use by visually handicapped users. Gaver[8, 9] proposed the use of auditory icons for use as a part of Apple's interface on the Macintosh machines.

It has been noted by researchers that in most situations program output is visual. Programming environments and applications developed so far have attempted to use visual media to a great extent[2]. Though research in the use of audio in workstations[11, 14] has been on the rise, audio remains a distant second to visual media. However, with the availability of low cost audio devices such as synthesizer modules and sound digitizers and the provision of sound generators in workstations, audio is within reach of most PC and workstation users.

We have designed a language that simplifies the task of specifying which occurrences during program execution are to be auralized and how. The language is named Listen Specification Language, abbreviated as LSL. Listen is the name of our project to investigate possible uses of sound in programming environments and software based applications. The need for LSL, its syntax, use, and current implementation status are described in the remaining sections. Section 2 outlines the need for specifications of program auralizations and LSL. Language requirements and the underlying rationale are presented in Section 3. Essential features of LSL are presented in Section 4. Examples illustrating the use of LSL in various applications are given in Section 5. Section 6 explains how LSL can mesh with a programming environment. An implementation strategy for LSL is outlined in Section 7. Current status of LSL implementation is reported in Section 8. A summary and preliminary conclusions regarding the utility of LSL appear in Section 8.

2 The need for LSL

Our research is concerned with the investigation of various uses of sound in programming environments. Some generic questions we ask and seek answers to are listed below.

---

[1]We include human voice in this context.

1. How useful is sound in debugging programs?

2. How useful is sound in program understanding?

3. Can program auralization be used to improve the quality of simulations of various kinds such as telephone networks, mechanical systems, and biological systems?

4. How can sound be used to make programming environments usable by, in particular, the visually handicapped with the same ease as by individuals without such a handicap?

As mentioned above, researchers have attempted to obtain answers to similar questions. We believe that significant progress remains to be made before sound is used widely in various programming related tasks.

While investigating answers to the above questions, we encountered a need for a general purpose mechanism to specify the auralization of programs. In the absence of such a mechanism, auralization is done by editing the source code and adding calls to library procedures that generate sound. For example, suppose that a parallel program to be executed on a machine with 1000 processors is to be auralized. The auralization should be such that whenever an even numbered processor sends data, a sound with timbre characteristic T_1 is generated and whenever an odd numbered processor sends data a sound with timbre T_2 is generated. When so auralized, a listener will be able to distinguish between data sends by even and odd numbered processors. Assuming that one has access to a library routine `gen_sound` to generate sound with a specific timbre, one can edit the program and add calls with appropriate parameters to `gen_sound` at all places where a data send occurs. This could become an inconvenient task if communication statements are distributed over many procedures in a large program spread over several source files. The task could be simplified if one could formally specify the auralization mentioned above. Such a specification and the program to be auralized could be preprocessed automatically to generate a source program with code inserted for the desired auralization. Any change in auralization will then require a change only to the specification and not to the source of the parallel program.

As another example, consider the task of debugging a distributed system that controls various functions of an automobile. It is desired to auralize the program so that each call to a control procedure, e.g. to procedure gear_change, is identified by a suitable sound. As above, one may edit the source to achieve the desired effect. Another alternative is to formally specify the above auralization requirement and use preprocessing to add the necessary code.

A more difficult situation arises when the value of one or more variables is to be monitored. Suppose that we want to monitor the value of a variable x in a program. Whenever this variable exceeds a predetermined value a sound should be generated. Again, one may use an editor to add a few lines of code at all places in the program where x has been defined. An easier alternative

appears to be to formalize the above auralization and preprocess it. During further development of the software if one decides to monitor other variables, the specification can be easily modified as compared to editing the code which might require careful passes through various source files.

Examples such as the ones above led us to consider designing a language for specifying auralizations. We note the pioneering work in the design of languages for music [13, 17]. The main purpose of these languages was to specify music. In their present form, these languages are not suited to the auralization tasks mentioned above.

3   Basic definitions and LSL requirements

Based on the perceived need for a specification language, we set forth the following idealized requirements for LSL.

1. *Generality*: It should be possible to specify any auralization using LSL.

2. *Language independence*: It should be possible to use LSL with the commonly used programming languages such as C, C^{++}, Ada, Pascal, and Fortran.

Below we define basic terms and introduce concepts that help us formalize the above goals. Our formalization brings reality to the above requirements. LSL satisfies the requirements with respect to this formalization.

3.1   ASPECs and realizations

To be able to design a language that can specify all possible auralizations, we need a quantification of two domains. Let E be the domain of all those occurrences during the execution of any program that one may wish to auralize. The nature of such occurrences is discussed below. Let S be the domain of all possible sound patterns that may be associated with each element of E. A *mapping* from E to S is an association of sound patterns in S to occurrences in E. Such a mapping is specified as a set of pairs (e, s) where $e \in E$ and $s \in S$. The term *program auralization* for a given program P refers to the set $\{(e_1, s_1), (e_2, s_2), \ldots, (e_n, s_n)\}$, where each $(e_i, s_i), 1 \leq i \leq n$ is an association of an occurrence to a sound pattern. A *language L* for program auralization is a notation to specify any such mapping for any program. A mapping specified using L is referred to as *auralization specification* abbreviated as ASPEC. We assume that specifications are always written with reference to a given, though arbitrary, program in some programming language. Figure 1 illustrates this view of program auralization. Note that an ASPEC is a many-to-many mapping.

Let (e, s) be an element of an ASPEC for program P. During the execution of P if each occurrence e is identified by a sound pattern s, we say that the pair (e, s) has been *realized*. An ASPEC for program P is considered realized if all its elements are realized for all executions of P.

Occurrence space   Sound pattern space
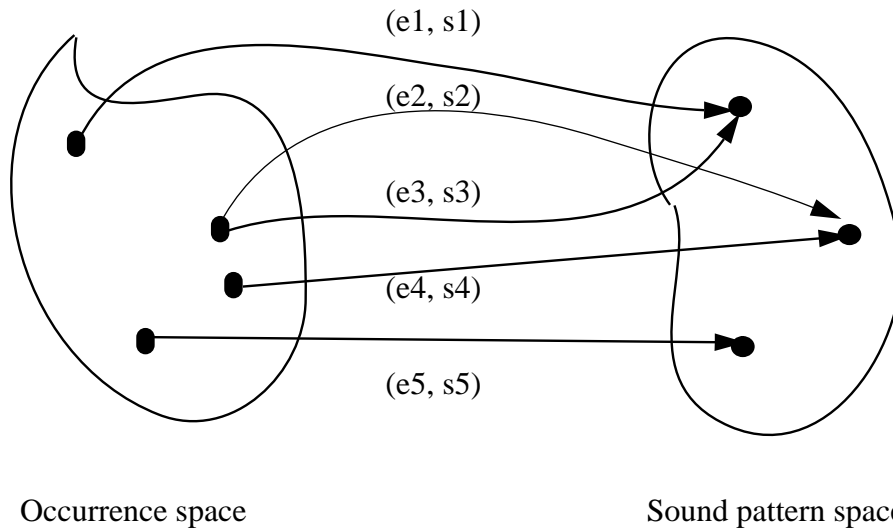
Figure 1: A domain-based view of program auralization. The ASPEC in this example consists of four occurrence and sound pattern pairs as shown. (e_i, s_i) are elements of the specified mapping.

We say that an implementation of L for programs in a given programming language PL is correct if each ASPEC, written in L, for any program P, written in PL is realized.

3.2   Occurrence space characterization

Ideally, we would like to be able to specify any auralization. To do so, we need to characterize the space of all possible occurrences that might arise during program execution. Towards this end we select a three-dimensional space using the orthogonal notions of position, data, and time. *Position* refers to any identifiable point in a program. For example, in a C program, beginning of a function call, end of a function return, start of a while-loop, start of a while-loop body, and start of a condition, are all positions. In general, an identifiable point is any point in the program at which an executable syntactic entity begins or ends. This implies that a position cannot be in the middle of an identifier or a constant. In terms of a parse tree for a given program, any node of the parse tree denotes a position. For example, the subscripted dot ($\bullet_i$) denotes seven possible positions in the following assignment: $\bullet_1 X \bullet_2 = \bullet_3 X \bullet_4 + \bullet_5 3 \bullet_6 / \bullet_7 2$.

Data in a program refers to constants allowed in the language of the program being auralized and the values of program variables. A *data relationship* is an expression consisting of constants, variables, and function calls. Time refers to the execution time of the program. It is measured in units dependent on the system responsible for the execution of the auralized program. In a heterogeneous system, time is measured in units agreed upon by all elements of the system.

As shown in Figure 2, a three dimensional space is used for specifying occurrences in LSL. Two kinds of occurrences are distinguished: events and activities. LSL allows an arbitrary combination of data relationships, positions, and time to specify an event or an activity associated with program
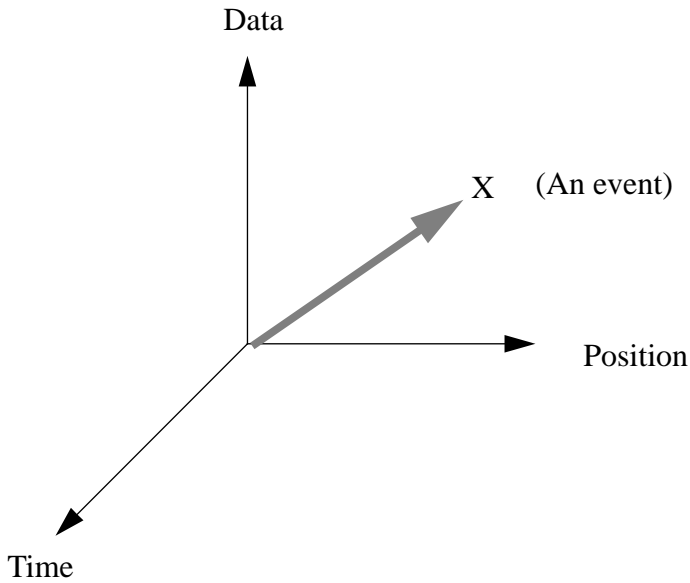
Figure 2: Occurrence space characterization in LSL.

execution.

3.3   Sound space characterization

The sound space is characterized by sound patterns comprised of notes, durations, play styles, and instruments. Notes of arbitrary durations can be combined to form sound patterns. Each note can be associated with one of several play styles and with an arbitrary instrument. For example, a note can be played staccato on a piano with a specified volume. Combining notes in various ways gives rise to a domain consisting of an infinity of sound patterns. Digitized sound, such as human voice, is considered a sound pattern.

3.4   Programming language independence

The second requirement stated above is significant as we want LSL to be usable by programmers regardless of their preference for one or the other programming language. Adherence to this requirement has produced a language which in the strict sense should be considered as a meta-language. One can therefore adapt LSL to specific programming languages as described later. In all examples below, we use a C [12] adaptation of LSL termed LSL/C.

4   Features and syntax of LSL

The salient features of LSL are reviewed next. Details of LSL syntax and semantics appear in the Appendix. An LSL program is known as a specification. Each specification is composed of one or more specification modules. Each specification module is composed of zero or more specification

definitions and one main specification. A specification module, a specification definition, and a main specification are analogous to, respectively, a module, a procedure, and a module body in a Modula-2 [15] program. As an example of LSL specification structure consider the specification listed in Figure 3. It begins with `begin auralspec` and ends with `end auralspec`. Each module begins with a header identified by the `specmodule` keyword followed by the module name such as spec_module_1, spec_module_2, and so on in this example. A module header is followed by applicability constraints which specify parts of the program to which the specifications are to be applied. Then come declarations of variables used in this module followed by zero or more specification definitions such as spec_def_1, spec_def_2, and so on. Global variables are shared between various modules. Variables and specification definitions to be exported (imported) are listed in the `export` (`import`) declaration. Variables declared in the program being auralized can also be used inside LSL specifications. These are known as *external* variables.

4.1   Constants, variables, and types

LSL is a typed language. It contains constants, variables, and types just as several other languages do. An identifier name is a sequence of one or more characters consisting of upper or lower case letters, digits, and the underscore (_). The first character in an identifier must be a letter or an underscore. Upper and lower case letters are treated as being different. Variables and constants can be assigned arbitrary names. Values likely to arise during program auralization are grouped together into primitive types. Table 1 lists the primitive types available in LSL. Values of type `note` and `pattern` are enclosed in quotes to distinguish them from variable names. A note is specified by indicating its pitch e.g. "E4b" indicates E-flat above the middle C on a piano keyboard. Attributes listed in Table 2 can be added to a note separated by a colon (:). A pattern is a sequence of notes and voices[2] played in the specified sequence. A sequence of notes within a pattern can be enclosed in parentheses to indicate a blocked chord also referred to as a chord pattern. A variable name can be used within a pattern by preceding it with a dot. For example, if the identifier *cmajor* denotes a chord pattern, then p:= ".cmajor E5" denotes a pattern consisting of the value of *cmajor* followed by the note E5. Just as values could be printed or displayed in computer programs, we say that values of type `note` and `pattern` could be *played back* during the execution of an auralized program.

   The set of key signatures constitutes the type `ksig`. Pre- or user-defined functions are used to manipulate values of type `ksig`. Constants of type `ksig` are enclosed inside double quotes and can be assigned to variables of the same type. A key signature could be predefined or user defined. A predefined key signature consists of two parts: a key name and a modifier. Examples of key

---

[2]Throughout this work data of type "voice" refers to digitized sound. Thus, for example, both digitized voice and digitized guitar sound are characterized as voice data.

```
begin auralspec
      specmodule myprog_auralize

      /* This module contains specifications to auralize myprog procedure.          */

      /* Applicability constraints, if any, come here.                              */

      /* Declarations for variables global and external to this module .            */

      specdef specdef_1 (parameters);
        /* Declarations of parameters, local variables, and functions. */
        begin specdef_1

          :

        end spec specdef_1;

      spec-def spec_def_2 (parameters);
        /* Declarations of parameters, local variables, and functions. */
        begin specdef_2

          :

        end specdef_2;


          :

      specdef specdef_n (parameters);
        /* Declarations of parameters, local variables, and functions. */
        begin specdef_n

          :

        end specdef_n;

      begin myprog_auralize;
      /* Specifications for module myprog_auralize. */

          :

      end myprog_auralize;

      /* Other module specifications. */

      :
end auralspec.
```

Figure 3: Structure of an LSL specification containing one module..

names are Eb (denoting E flat) and C# (denoting C sharp). Modifiers could be major, minor (same as harmonic minor), lydian, ionian (same as major), mixolydian, dorian, aeolian, phrygian, and locrian. Thus, for example, "C#:minor" and "E:phrygian" are valid key signatures. A user defined key signature is any enumeration of notes. For example, "C D Eb G A" is a key signature of a pentatonic scale.

The set of time signatures constitutes the type `tsig`. Constants of type `tsig` are enclosed within parentheses. A time signature consists of two parts: the beat structure and the note that takes one beat. For example, (4:4) is a simple time signature indicating 4 beats to a measure with a quarter note of one beat in duration. A more complex time signature is (3+2+2:8) which indicates a beat structure of 3+2+2 with an eigth note taking one beat. A beat structure such as 3+2+2 indicates that the first measure is of 3 beats in duration, followed by two measures each of 2 beats duration, followed by a measure of 3 beats and so on. Time signatures can be assigned to variables of the same type and manipulated by functions.

Type `file` is the set of file names. A filename is specified by enclosing the name within double quotes. Thus, "your_name_please.v" can serve as a file name. The use of file names is illustrated through LSL examples below. Note that we use a string of characters enclosed within double quotes in a variety of contexts. It is the context that unambiguously determines the type of a string.

A special type `voice` has been included to play digitized voice during program execution. We have assumed that voice will be digitized using a suitable digitizer, e.g. the Audiomedia II [5] card from Digidesign, and stored as a sample in a file. It is this sample that becomes a constant and can be assigned to a variable of type `voice`. Voice can be used in note patterns by specifying variables of type `voice`.

Variables must be declared before use. The following declaration declares body_begin and body_end to be of type `note`, loop_begin, loop_end, and measure to be of type `pattern`.

```
var
  body_begin, body_end: note;
  loop_begin, loop_end, measure: pattern;
```

Note and rest values

Attributes aid in specifying various properties of notes and patterns. Perhaps the most common attribute of a note or a chord sequence is its duration. For example, "E4:q" denotes a quarter note whose duration will be determined by the time signature and the metronome value. The duration attributes can be multiplied or added to get dotted quarter note, and other fractions of note values. For example, (hq) read as *half of quarter* denotes an eight note, (hhq) read as *half of half of a quarter* denotes a sixteenth note. Table 3 lists sample note values and the corresponding attribute

11

Table 1: Primitive types in LSL.

Keyword	Sample values	Description
int	−20 or 76	Set of integers.
note	"E4b"	Set of notes; not all of these may be played back in a particular implementation. A subset of the notes is labelled starting at A0 and going up to C8 as found on an 88-key piano keyboard. These 88 notes correspond to integer values of 0 to 87. Other notes values may be obtained using the predefined function *inttonote*. Other predefined functions are listed in Table 6.
	R	A rest is treated as a silent note with duration specified by a duration attribute.
tsig	(3:8) or (3+2+2:4)	Set of pairs of values denoting a time signature. The first element in the pair specifies the beat structure i.e. the number of beats per measure. The second element is the note value that corresponds to one beat. The beat structure could be complex as explained in the text.
ksig	"Eb:minor" "(C D E F# G A B)"	Set of k-tuples of pitch values. The set may be specified using abbreviations such as Eb:minor to indicate the key of Eb minor or by enumerating all pitches regardless of their specific position on a keyboard as in the example.
pattern	"G3E3C4"	Set of note and/or chord patterns consisting of zero or more notes or chords.
voice	†	Set of digitized voice patterns. A variable of this type can be set to point to a memory or disk file containing a digitized voice pattern.
file	"done-voice.v"	Set of file names. File extensions are interpreted. .c is for C program files, .v for digitized voice files.

† Any digitized sound in a suitable format, e.g. AIFF [3].

Table 2: Attributes in LSL.

Code	Applicability	Description
f	Note	Indicates a full note .
h	Note	Indicates a half note.
q	Note	Indicates quarter note.
e	Note	Indicates eight note.
s	Note	Indicates sixteenth note.
chan	Note, pattern	Specifies the MIDI[†] channel on which to play.
play	Note	Indicates one or more play styles.
inst	Note, pattern	Specifies which instrument is to play.
mm	Pattern	Metronome setting. This is applicable only to patterns. Notes not part of a pattern are played for a duration determined by global metronome setting. A metronome setting specified for a pattern takes priority over any global setting only while this pattern is played.
ptime	Note, pattern	Specifies the exact time in seconds to play the note or a pattern.

[†] MIDI is an acronym for Musical Instrument Digital Interface.

combinations. Various rests could be obtained using the attribute combinations shown in Table 3 with the letter R. For example, "R:(hq+hhq)" denotes a dotted eight rest.

Duration can be specified for a chord by a single duration attribute. For example, "(C4E4G4):q" denotes a chord consisting of three quarter notes. Notes and chords for which the duration is not specified explicitly, as in "E4", are played for a duration determined by implementation dependent default durations (See Table 4 for various defaults.).

Type constructor

Values of primitive types can be combined together into an array. The following sequence declares an array of measures, each measure being a pattern. Elements of an array can be accessed by subscripting. Thus tclef_staff [k+1] refers to the (k+1)th element of tclef_staff which is of type pattern.

```
const
  scoresize = 25;

var
  tclef_staff: array [1..scoresize] of pattern;
```

Table 3: Sample note values using LSL duration attributes.

Note value	Attribute combinations
Quarter note	q or hh
Eight note	hq
Sixteenth note	hhq or qq
Thirty second note	hhhq
Sixtyfourth note	hhhhq or ss
Dotted half note	h+q
Dotted quarter note	q+hq
Dotted eigth note	hq+hhq

Table 4: Default values of run time parameters.

Item	Default value
Metronome	q=120
Key signature	C major
Time signature	(4:4)
Channel	1
Instrument code	1
Note duration	q
Play mode	discrete for notify
	discrete for dtrack
	continuous for atrack
Pitch	"C4"

4.2   Sound pattern specification

The `play` command is used to specify what sounds are to be generated when some part of a program is executed. The general syntax[3] of `play` is:

    `play` <playlist>

where <playlist> is a list consisting of one or more notes and patterns specified using constants, variables, and function calls. Key and time signatures are some of the parameters that may be specified. Elements of <playlist> can be separated by a comma (,) or a parallel (∥) sign. An example of `play` command appears below.

    `play` (loop_background ∥ (func_call, no_parameters)) `with mm q` =120, `inst` = "piano";

The above `play` when executed will play the sound associated with the variable loop_background together with a sequence of sounds denoted by the variables func_call and no_parameters. Default key and time signatures will be used. The metronome will be set to play 120 quarter notes per minute and the notes will be played using a piano sound.

4.3   Event notification

A useful characteristic of LSL is its ability to specify events to be auralized. A programmer may formulate an event to be auralized in terms of the application. However, such a specification is translated in terms of program position, data, and time as described earlier. For example, in an automobile simulator, events such as *gear change, speed set, resume cruise*, and *oil check* may be candidates for auralization. Suppose that the occurrence of these events is indicated by calls to procedures that correspond to the simulation of an activity such as *gear change*. It is these procedure calls that serve as event indicators to LSL. Thus, for example, such a call to the *gear_change* procedure could be mapped to sound using an LSL specification.

Event specification is achieved by the `notify` command. `notify` is a generic command and can be adapted to a variety of procedural languages. In examples below we assume that programs being auralized have been coded in C. The syntax of `notify` appears below:

    `notify` {<all-selective>} {<label-parameter>} <event-specifier> {<sound_specifier>}
    {<scope-specifier>}

<all-selective> specifies which subset of events selected by a `notify` are to be auralized. Possible event codes are `all` and `selective`. If `selective` is used, one or more labels must be specified to indicate which events are to be selected. <event-specifier> specifies one or more events to be notified aurally.

There are five ways to specify an event. One may specify a general syntactic entity, a special

---

[3]Syntactic entities are enclosed in < and >. Optional entities are enclosed in { and }. For a complete syntax of LSL see Appendix.

syntactic entity, an assertion, a relative timed event, and any combination of the above four. Relative timed events are discussed in Section 4.6; other methods are described below. Table 5 lists all event codes in LSL/C. For example, *while-statement-enter* is an event specifier; the corresponding event occurs once each time a while statement is executed. The start and termination of program execution serve as events.

The expression $(x < y)$ serves as a special syntactic entity. The associated event occurs whenever the expression (x<y) is executed. An assertion such as $(x + y) > (p + q)$ also specifies an event which occurs whenever the assertion evaluates to false. If e_1 and e_2 are two events specified using any of the above approaches, then (e_1 **and** e_2) and (e_1 **or** e_2) are also events.

The scope of a **notify** may be restricted using the <scope-specifier>. In LSL/C, the scope can be restricted to one or more functions or files. For example, if an assertion is to be checked only inside function *sort*, one may suitably restrict the scope to that function. Labels can be used in conjunction with scope restrictions to specify arbitrarily small regions in a program.

The sound specifier is a variable name, constant, or a function call that specifies the intended auralization of the selected events. Sample **notify** commands appear below.

1. **notify all rule**=while_loop_body_begin **using** body_begin;

   notify all rule= while_loop_body_end **using** body_end;

2. **notify selective label** = special_loop **rule**=while_loop_body_begin **using** body_begin;

   notify selective label = special_loop **rule**=while_loop_body_end **using** body_end;

3. **notify all instance**= "++count" **and** "search(x)" **using** count_or_search **in func** = "search", "report";

4. **notify all assertion** = (x<y || p≥q) **using** assertion_failed;

5. **notify all rule** = conditional_expression **and assertion** = odd(x) **using** cond_sound **in filename** = "myfile.c";

Example 1 above specifies two event types, namely the beginning and end of a while-loop body using two general purpose syntactic specifiers. It also indicates that all positions in the program where such events could occur are to be auralized. Thus, a C program auralized using the above **notify** will generate the sound corresponding to the variables body_begin and body_end, respectively, whenever the beginning and end of a while-loop body are executed.

Example 2 is the same as Example 1 except that the event selection is selective. Thus, any loop body labelled by *special_loop* will be auralized. Any syntactic entity can be labelled in the program being auralized by placing an LSL **label** command in front of that entity as described in Section 4.11.

16

Example 3 specifies the execution of the statements *++count* and *search(x)* as the events. When any of these two events occur, *count_or_search* is played. However, these events are to be recognized only inside functions *search* and *report*.

Example 4 above specifies an event which occurs whenever the condition (x<y || p≥q) is not satisfied. Note that this condition is based on variables in the program being auralized. When this condition is not satisfied, *assertion_failed* is to be played. Example 5 shows how to specify the auralization of all conditional expressions that occur in file *myfile.c* only when condition *odd(x)* is not satisfied.

The `all` and `selective` tags can restrict any event selection. Multiple labels are used within one `notify` command as in the following.

> `notify selective label` = loop_1, loop_2 `rule`=while_loop_body_begin `using` body_begin;
> `notify selective label` = special_loop `rule`= while_loop_body_end `using` body_end;

The above `notify` commands specify the same type of events as in Example 2 except that loop body begins and ends that contain any one of the two labels *loop_1* and *loop_2* will be selected for auralization.

4.4   Data tracking

Event notification consists of specifying one or more events and reporting them aurally during program execution. There are applications wherein changes to values of variables need to be monitored. It is certainly possible to specify assignments to such variables as events and then report the execution of these assignments aurally. Such reporting is, however, independent of the data being assigned. To obtain data dependent auralization, LSL provides the `dtrack` command. The syntax of `dtrack` appears below.

> `dtrack` <track-id-list> <sound-specifier> {<mode-specifier>}
>   {<start-event-spec>} {<term-event-spec>}

Using `dtrack`, one or more variables can be tracked. For the variable to be tracked, an initial value can optionally be specified using the `init` keyword. The type of the initial value must match that of the variable to be tracked. The initial value may also be captured immediately after the execution of an assignment labelled using an LSL label.

As in `notify`, a <sound-specifier> specifies the sound to be used while tracking the variables. Here we introduce another method for specifying sounds which is particularly useful in conjunction with the `dtrack` command. A sound pattern whose characteristics depend on program generated data will be referred to as a *Value Dependent Aural Pattern* and abbreviated as VDAP. The `using` clause in the <sound-specifier> specifies the name of the function, say f, that emits a VDAP based on variables being tracked. f is a language dependent function containing LSL commands

Table 5: Keywords and codes for LSL event specifiers in C.

Category	Event specifier	Code†	Event specifier	Code†
Program	start	start	end	end
Expression	variable	var	assignment_expression	aex
	conditional_expression	cex		
Iteration	iteration_statement	ist	iteration_body_begin	ibb
	iteration_body_end	ibe	while_statement_enter	wse
	while_statement_exit	wsx	do_while	dow
	for_statement_enter	fre	for_statement_exit	frx
	while_body_begin	wbb	while_body_end	wbe
	for_body_begin	fbb	for_body_end	fbe
	do_while_body_begin	dbb	do_while_body_end	dbe
Jump	jump_statement	jmp	continue_statement	cst
	break_statement	bst	return_statement	rst
	goto_statement	gst		
Selection	selection_statement	sst	if_statement	ist
	if_then_part	itp	if_else_part	iep
	switch_statement	sst	switch_body_begin	sbb
	switch_body_end	sbe		
Functions	function_call	fnc	function_entry	fne
	function_return	fnr		

† Event specifiers and their abbreviated codes can be used interchangeably to specify a rule in a notify statement.

Table 6: Predefined functions in LSL.

Function name	Mapping	Description
intton	int → note	Converts an integer to a note. Integers in the inclusive range 0 to 87 get converted to notes A0 to C8.
ntoint	note → int	Converts a note to an integer.
nabove	note × ksig → note	Returns the note above the input in the given scale.
nbelow	note × ksig → note	Returns the note below the input in the given scale.
naboveh	note → note	Returns the note one half step above the input.
nbelowh	note → note	Returns the note one half step below the input.
circlen	ksig → ksig	Returns the next key signature in the circle of fifths. Valid only for predefined key signatures.
circlep	ksig → ksig	Returns the previous key signature in the circle of fifths. Valid only for predefined key signatures.
sectotick	int → int	Converts seconds to system dependent ticks.

for auralization. Thus, in LSL/C, f is a valid C function interspersed with LSL commands. f is executed after each assignment to the variable being tracked.

Tracking may be carried out in continuous or discrete mode. In continuous mode, tracking begins at the start of program execution, unless specified otherwise. A note pattern is emitted continuously until there is a change in the value of the variable being monitored. When the value changes, a newly computed note pattern is emitted continuously. In discrete mode, a note pattern is emitted once whenever the tracked variable changes its value. In discrete mode tracking begins the first time the tracked variable changes its value after program execution.

Tracking can also be controlled using <start-event> and <term-event>. Start and terminating events are specified, respectively, using the when and until clauses. A few examples of dtrack use appear below.

1. dtrack *speed*; will track variable *speed* using an initial value of 0 and default sound parameters such as note pitch and volume.

2. dtrack *crash* init=false; will track *crash* assuming an initial value of false.

3. dtrack *x* capture=x_reset; will track *x* after capturing its initial value at the assignment labelled by the LSL label *x_reset*

4. dtrack *mouse* and *color* using *color_mouse_melody* (&mouseval, &colorval) ; will track variables *mouse* and *color* using a user defined function named *color_mouse_melody* with two parameters.

5. dtrack *speed* `when` speed>65 `until` x≤65 `mode`=`continuous`; will begin tracking *speed* whenever its value exceeds 65 and will stop tracking it immediately after its value becomes equal to or less than 65. Tracking will resume if the start event occurs again. The discreet mode can be used to avoid resumption of tracking of *speed*.

4.5   Activity monitoring

An activity is a sequence of actions between two events. An activity begins at the occurrence of an event and ends at occurrence of a later event. As mentioned earlier, start and termination of program execution are considered as events. LSL allows specification of tracking arbitrary activities using the `atrack` command given below.

> `atrack` { `when` <event-specifier> } { `until` <event-specifier> }
> <sound-specifier> {<mode-specifier>}

<event-specifier>, <sound-specifier>, and <mode-specifier> have the same meaning as in the `dtrack` command. Tracking begins when the event specified immediately following `when` occurs (start event) and stops when the event specified following `until` occurs (terminating event). If the start event is omitted, tracking begins at the start of program execution. Tracking ends at program termination if the terminating event is omitted. If both the start and terminating events are omitted then the entire program execution is tracked. In `continuous` mode, an activity begins whenever the starting event occurs and terminates at the terminating event. In the `discrete` mode, an activity occurs as above but does not resume. Using the start and terminating events one may specify a variety of activity tracking patterns as shown in Figure 4.

4.6   Timed events

LSL provides a powerful mechanism to auralize timed events. `time` is a special variable in LSL which denotes the time spent from the start of program execution. Time is measured in system dependent *ticks*; each tick being the smallest unit by which `time` could be incremented. Thus, any expression using `time` can be used as a timed event. As an example, suppose that the *gear_change* function must be invoked in a program in less than 60 seconds after the program execution begins. It is desired to playback variable *bad_program* if this condition is not satisfied. The following `notify` illustrates how to write this specification in LSL.

> `notify rule` = function_call: gear_change `and assertion`=`time` ≤ sectotick(60)
> `using` bad_program `mode` = `discrete`;

In the above example, *sectotick* is an LSL predefined function to convert seconds to ticks. Notice that the expression `time` > sectotick(60) is a valid way to specify an event as described earlier while discussing the syntax of `notify`.
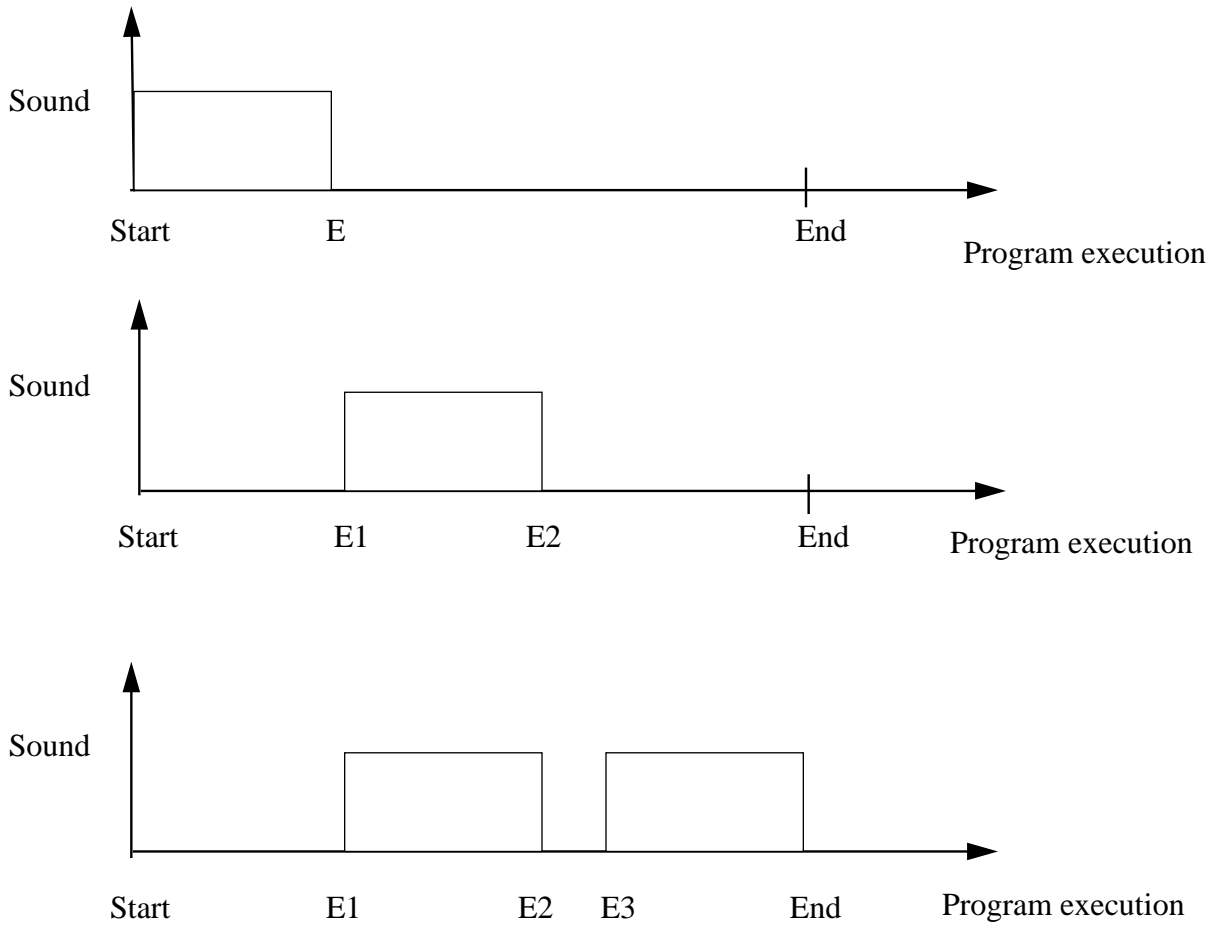
Figure 4: Sample activity patterns specifiable in LSL. E1, E2, and E3 denote events. Start and End denote the start and end of program execution.

21

It is often required to specify time relative to the occurrence of some event. This can be done in LSL using relative timed events as shown below.

   rtime = <expression> **after** <event-specifier>

Consider the use of this mechanism in the following example for tracking an event.

   dtrack **when** (rtime = sectotick(30)) **after rule** = function_call: missile_launch
      until rule= function_return: target_hit **using** missile_in_motion;

The above **dtrack** can be read as "Begin tracking 30 seconds after the function *missile_launch* has been called and terminate tracking when the function *target_hit* returns. The tracking sound is defined by the LSL variable *missile_in_motion*. Thus, using a combination of **time** and **rtime**, one may specify a variety of timed events for auralization.

4.7   Playback synchronization

Synchronization mode controls the playback of notes during program execution. There are two such modes: *program* or *metronome*. In the program mode, playback is synchronized to the program. In the metronome mode it is synchronized to a global metronome. The **syncto** command is used for setting the synchronization mode. The syntax of **syncto** is:

   syncto <sync-to>

The <sync-to> parameter can be **program** or **mm** for synchronization with, respectively, program execution or a global metronome. Multiple **syncto** commands may be placed in an LSL specification to alter the synchronization mode.

In the metronome mode, a buffer holds the notes generated by the executing program. When this buffer is full and the program attempts to send a note for playback, the playback routine does not return control to the program until the received note can be buffered. This may slow down program execution. To avoid this situation in metronome mode, one may use the **noslow** parameter such as in the command **syncto mm q=120, noslow**. When the **noslow** parameter has been specified, playback routine discards notes that are received when the buffer is full. This could cause some events or data tracking to pass by unauralized. The size of the playback buffer can be controlled by setting the **bufsize** parameter such as in **syncto mm=120, bufsize=1000** which specifies a buffer size that will hold at least 1000 notes.[4]

4.8   Assignments, loops, and conditionals

An assignment command has the general syntax shown below.

   identifier {<subscript_list>}:= <expression>;

---
[4]Each note belonging to a chord counts as one note.

where identifier is the name of a variable. Expression is any valid expression that evaluates to the type of the identifier on the left of the assignment. <subscript_list> is a list of subscripts used for selecting array elements if the identifier denotes an array. Loops can be formulated in an LSL specification using the `for` and `while` constructs. Syntax of these two constructs is given below.

> `for` <for_index> := <init_expression> `to` <final_expression> {`step` <step_expression>}
>   <spec_sequence>
> `while` <condition> `do` <spec_sequence>;

The semantics of each of the above commands are similar to that of the `for` and `while` statements in Pascal. All expressions in a `for` command must evaluate to integers. A <spec_sequence> is a sequence of zero or more LSL specification commands.

Conditional commands are provided in LSL for selectively specifying an auralization. The syntax of a conditional command appears below. Its semantics are similar to that of the `if` statement in Pascal.

> `if` <condition> `then` <spec_sequence> {`else` <spec_sequence>}

4.9   Controlling auralization state

During execution, an auralized program can be in one of two auralization states: ON or OFF. In the ON state any sound data resulting from the occurrence of an auralized event is sent to the sound processor. In the OFF state any such sound data is suppressed. LSL provides two commands to dynamically alter the auralization state. These are the `turn` and the `toggle` commands. These commands have no effect when placed inside an LSL specification. They may affect the auralization state when placed inside the auralized program.

Using `turn` is one way to switch sounds on or off. `turn on` switches the sound on and `turn off` switches it off. The command may be placed anywhere inside the auralized program. Upon the start of program execution, the auralization state is ON. The `turn` command takes effect immediately after it is executed. Sound channels can be switched off selectively by specifying the channel number as in `turn off chan`=4; switches off any sound on channel 4.

Another way to turn the sound on or off is with the `toggle` command. The syntax of `toggle` is given below.

> `toggle` {id} <toggle-source> = constant

where <toggle-source> could be the MIDI or computer keyboard indicated, respectively, by the keywords `midi` and `keysig`. The constant is a string containing the toggle note from the MIDI keyboard and the toggle key from the computer keyboard. When specified, `id` denotes the name of a class (defined below) of events, activities, and data items to be affected by this command.

During program execution, the auralization state can be toggled using the source specified in the command. For example, if the middle C on a MIDI keyboard is the toggle source, tapping the middle C once, after program execution begins, turns the sound off. Tapping it again turns it on. Input from the toggle source is processed only when an auralized event occurs. When such an event occurs, an LSL library routine is invoked to check for a pending toggle request. If a request is pending, the auralization state is switched to OFF if it is ON, or to ON if it is OFF.

A program may contain both `turn` and `toggle` commands. A `turn` might change the auralization state to off only to be switched back to on by a toggle. This is certainly one useful scenario. Note that whereas `turn` commands are placed into the code prior to compilation and do not provide the user any control *after* compilation, the `toggle` command permits dynamic changes to the auralization state. The toggle default in LSL is the space bar on the computer keyboard. Thus, even when no `toggle` is specified in a program, auralization state may be toggled using the space bar.

Regardless of the auralization state, note values are generated and sent to the library routine responsible for playback. It is this library routine that decides, based on the current auralization state, if the received notes are to be played or not. In the metronome sync mode, all notes emitted are buffered in a special playback buffer maintained by the library routine. The buffered notes are removed from the buffer when their turn comes for playback. This is determined by the current metronome setting. When playback resumes due to a `toggle` or a `turn` changing the auralization state to `on`, the notes are played back in accordance with the metronome setting. In program sync mode, notes received by the library routine are discarded if playback is turned off.

4.10   Event, data, and activity classes

An event class[5] consists of one or more events. A `notify` command specifies one or more events which may occur at several positions inside a program and several times during program execution. Events specified in one or more `notify` commands constitute an event class. Similarly, a *data class* is a collection of one or more variables. A `dtrack` command specifies one or more variables to be tracked. Variables specified in one or more `dtrack` commands constitute a data class. An *activity class* is defined similarly with respect to activities specified in one or more `atrack` commands. A class that consists of at least two elements of different types, e.g. event and activity, or event and data, or data and activity, is known as a *mixed class*.

It is possible for a user to define each of the above classes in an LSL specification. This is done by naming one or more `notify`, `dtrack`, and `atrack` commands. Any of these three commands can be named using the following syntax:

---

[5]Classes defined in this section have no intentional relationship with the notion of classes in C++ and object oriented programming literature.

$id_1::id_2::\ldots::id_n::$command

where each subscripted *id* above denotes a name and *command* denotes any event, data, or activity specification command. Multiple commands can share a name. Each *id*, when used as the name of a command, is treated as the name of a class. The class so named consists of events, data, or activities specified in the commands named by *id*. One command can be assigned multiple names. This makes it easy to define classes that are not disjoint. Consider the following example.

function_related::`notify` `rule`=function_call;

function_related::`notify` `rule`=function_return;

data_related::`dtrack` a `and` b `and` c;

special::data_related::`dtrack` p `and` q;

The above three commands have been named to identify three classes. Class *function_related* consists of events that correspond to function calls and return. Another class named *data_related* consists of data items *a, b,c, p*, and *q*. Yet another class named *special* consists of data items *p* and *q*.

The notion of a class can be used to model abstraction during program auralization. For example, consider the auralization of tractor control software. The programmer may like to group all the events into two classes. One class consists of events that correspond to engine control. Another class consists of events that correspond to the control of paraphernalia attached to the tractor, e.g. a seeding device. By simply using the event specification mechanism of LSL there is no way to explicitly incorporate these classes into an LSL specification. The mechanism of naming a command, as described above, however, does provide a convenient means for defining classes.

Once defined, classes of events can be accessed at an abstract level using their names. For example, during the execution of an auralized program, it is possible to interact with the LSL run-time system and turn off the auralization of all events within a class. It is also possible to request LSLed (described in Section 6.1 below) to provide a comprehensive list of classes and their individual elements. Thus the use of classes enables a user to interact with an auralized program in terms of "high level" occurrences, e.g. events, instead of dealing with syntax based definitions.

4.11   Embedding LSL commands

LSL commands can be embedded in C programs inside comments. The LSL preprocessor recognizes an LSL command embedding if the first token beginning with a letter immediately following the comment begin delimiter (/*) is `LSL:`. Immediately following the delimiter, a sequence of LSL commands can be placed enclosed within the `begin` and `end` delimiters. For an example of such an embedding see Example 6 on page 30. The LSL commands so embedded are translated to C code by the LSL preprocessor. LSL commands such as `play` and `notify` get translated into calls to

library functions. Other LSL commands, such as assignments and `dtrack` commands get translated into more complex C code.

It is possible to identify specific constructs of a C program by labelling. A label is placed inside a comment by using the keyword `label` as the first keyword starting with a letter immediately following the comment start delimiter. Thus, for example, /* `label`=here, onemore */ provides two labels *here* and *onemore* for possible use by the LSL preprocessor. The following example shows how to label the beginning and end of a loop.

> ⋮
>
> `while` (c = getchar()!=eof)
> {
> /*`label`=special_loop This is an LSL label for the beginning of loop body. */
> ++nc;
> ⋮
> /*`label`=special_loop This is an LSL label for the end of loop body. */
> }

5   Examples of LSL/C use

We now present a few examples illustrating the use of LSL. Each example consists of a problem statement and a solution using LSL/C. In each example we assume that, unless specified otherwise, default values are used for various sound related parameters such as MIDI channel, timbre, volume, and metronome value.

Example 1

It is desired to auralize all loops in a C program. Loops will be identified by the `while`-`do` constructs. On entry to a loop, note C4 is to be played for half a measure duration. Each time control reaches the start and end of a loop body, notes E4 and G4 are to be played, respectively. On loop exit, note C5 is to be played for a full measure. The following LSL/C specification meets the above auralization requirements.

```
begin auralspec
specmodule loop_auralize
var
      l_begin, l_end, b_begin, b_end: note;
begin loop_auralize
      l_begin:= "C4:h"; l_end:= "C5:f";
```

```
        b_begin:= "E4:q"; b_end:= "G4:q";

        notify all rule = while_statement_enter using l_begin;

        notify all rule = while_statement_exit using l_end;

        notify all rule = while_body_begin using b_begin;

        notify all rule = while_body_end using b_end;

end loop_auralize;

end auralspec.
```

■

Example 2

An automobile contains a distributed microcontroller network. The software to control this network
and other automobile functions is to be tested and debugged. All calls to functions *gear_change*,
oil_check, and *weak_battery* are to be auralized by playing suitable sound patterns. The following
LSL/C specification meets this auralization requirement.

```
begin auralspec
specmodule call_auralize
var
        gear_change_pattern, oil_check_pattern, battery_weak_pattern: pattern;
begin call_auralize
        gear_change_pattern:= "F2G2F2G2F2G2C1:qq"+ "C1:f";
        oil_check_pattern:= "F6G6:h";
        battery_weak_pattern:= "A2C2A2C2";
        notify all rule = function_call: "gear_change" using gear_change_pattern;
        notify all rule = function_call: "oil_check" using oil_check_pattern;
        notify all rule = function_call: "battery_weak" using battery_weak_pattern;
end call_auralize;
end auralspec.
```

■

Example 3

A parallel program consists of a procedure MAIN which, after initialization, loads each of N pro-
cessors with a copy of program SOLVE. On each processor, SOLVE aids in the solution of a partial
differential equation using a discretization of a multidimensional input domain. During execution
SOLVE communicates with its neighboring processors by placing calls to procedures *nread* (for read

from a processor) and *nwrite* (for write to a processor). It is desired to auralize this program so that calls to nread and nwrite are reported by suitable sound patterns. Only calls that originate from even numbered processors are to be auralized. The following LSL/C specification meets the above requirements.

```
begin auralspec
specmodule par_processing
external even, processor_num;
var
      read_pattern, write_pattern: pattern;
begin par_processing
      read_pattern:= "G5";
      write_pattern:= "C4";
      notify all rule = function_call: "nread" and even(processor_num) using read_pattern ;
      notify all rule = function_call: "nwrite" and even(processor_num)using write_pattern ;
end par_processing;
end auralspec.
```

■

Example 4

An editor accepts commands for text editing. After having executed a command, a voice message saying "Done" is to be generated. Assume that after receiving an edit command that is to be voiced as above, the editor invokes a function named *process_command* to process the input command. The following LSL/C specification meets the above requirement.

```
begin auralspec
specmodule editor_auralize
var
      done_voice: voice;
      myfilename: file;
begin editor_auralize
      filename := "done-container.v";
      done_voice:= filename;
      notify all rule =function_return: "process_command" using done_voice;
end editor_auralize;
end auralspec.
```

■

28

Example 5

It is desired to auralize all labelled assignments in 10 files named file-1.c, file-2.c, and so on through file-10.c. Assignments in the first of these files are to be notified by playing a C4, the next by playing a D4, then by E4, and so on using successive notes in the C-major scale. The following LSL/C specification meets this requirement.

```
begin auralspec
specmodule for_loop
const
      num_of_files = 10;

var
      first_note, next_note: note;
      next_file: int;
      filenames: array [1..num_of_files] of file;
specdef init_file_names();
      begin init_file_names
        filenames[1] := "file-1.c"; filenames[2] := "file-2.c";
        filenames[3] := "file-3.c"; filenames[4] := "file-4.c";
        filenames[5] := "file-5.c"; filenames[6] := "file-6.c";
        filenames[7] := "file-7.c"; filenames[8] := "file-8.c";
        filenames[9] := "file-9.c"; filenames[10] := "file-10.c";
      end init_file_names;
begin for_loop
      next_file:=1; next_note:=first_note;
      init_file_names();
      for next_file:=1 to num_of_files do
      begin
        notify selective label = this_assign rule=assignment_expression using next_note
            in filename = filenames[next_file];
        next_note := nabove(next_note, "C:major");
      end
end for_loop;
end auralspec.
```

The above LSL/C specification uses an LSL function to insert all file names into an array. The `for` loop then steps through each file name to specify the desired auralization. Successive notes are

obtained using a predefined function named *nabove* which takes a note as an argument and returns the next note a full step above on a given scale. ■

Example 6

This example illustrates how `dtrack` can be used with VDAP (Value Dependent Aural Pattern) to track arbitrary functions of program variables. Suppose that it is desired to track the dynamic relationship between two variables named *rock_remain* and *dist_remain*. *rock_remain* represents the number of rockets remaining to be fired and *dist_remain* the remaining distance to be travelled. Each time a rocket is fired the count of remaining rockets reduces by 1. The distance remaining to be travelled is updated by some process in the program.

It is desired to emit a continuous audible sound only while the remaining distance (*dist_remain*) is more than the critical distance (*crit_dist*) and the number of remaining rockets *rock_remain* is less than the critical rocket count (*crit_rock*). An LSL specification to achieve the above auralization consists of two parts. One part is an LSL specification module containing a `dtrack` command. This command specifies the use of a VDAP function named *crit_dist_track* for tracking *dist_remain*. The second part is the definition of a C function named *crit_dist_track* which has embedded LSL commands. The entire LSL specification module appears below.

```
begin auralspec
    VDAP begin
    crit_dist_track ( int *distval, *rockval);
    { /* This is a C function. distval is a pointer to
        dist_remain and rockval is a pointer to rock_remain. */
      if ((*distval > crit_distance) && (*rockval < crit_rock))
      { /* LSL:
        begin
          play grave_note with inst=bass, mode=continuous, chan=4;
        end
      */ }
      else
      { /* LSL:
        begin
          turn off chan=4;
        end
      */ }
    }
    VDAP end;
```

```
    specmodule emergency_sound
    applyto filename = dist_compute.c;
    const
       grave_note = "G2";
    begin emergency_sound
       dtrack dist_remain and rock_remain using crit_dist_track (&dist_remain, &rock_remain );
    end emergency_sound;
end auralspec.
```

While processing the above LSL module, the LSL preprocessor adds calls to the C function *crit_dist_track*, together with the specified parameters, immediately after each program statement that could possibly alter the value of *dist_remain* or *rock_remain*. The preprocessor also translates the LSL commands inside a VDAP to C code. The function so obtained is placed in suitable file for compilation by the C compiler. The preprocessor does not check for the correctness of the VDAP.

■

Example 7

A recursive function named *factorial* is to be auralized in such a way that each call to *factorial* generates a note with pitch proportional to the depth of recursion. The first call to factorial should generate C4; successive calls should generate one note higher in the C-major scale. The return sequence from *factorial* should play the notes back in the opposite order. The following LSL specification meets the above requirements.

```
begin auralspec
    VDAP begin
    fact_enter ( )
      { /* LSL:
        begin
          note_to_play = nabove (note_to_play, "C:major");
          play note_to_play
        end;
      */ }
    fact_exit ( )
      { /* LSL:
        begin
          note_to_play = nbelow (note_to_play,"C:major");
          play note_to_play;
```

```
      end
    */ }
  }
  VDAP end;
  specmodule fact_auralize
  var
    note_to_play: note;
  begin fact_auralize
    note_to_play:= "B3"; /* The next note is C4.*/
    notify all rule = function_entry: "factorial" using fact_enter();
    notify all rule = function_return: "factorial" using fact_exit();
  end fact_auralize;
end auralspec.
```

The above specification contains two VDAP functions, namely *fact_enter* and *fact_exit*. Each call to and return from factorial results in, respectively, the execution of *fact_enter* and *fact_exit* resulting in the desired playback. ∎

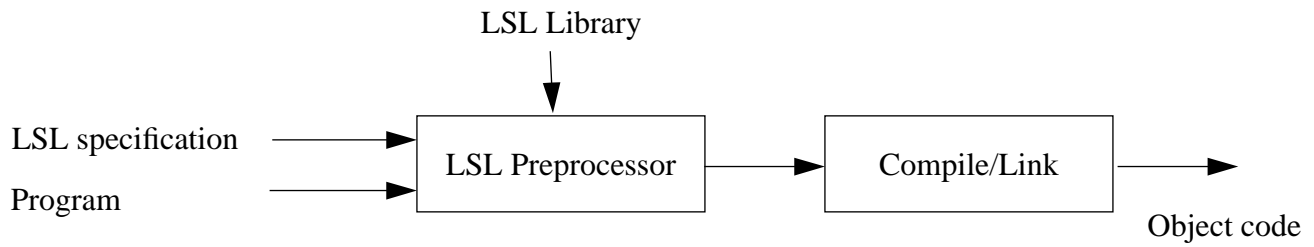# 6   LSL in a programming environment

LSL is designed to fulfill complex demands for program auralization. In a programming environment, one may use LSL to auralize parts of a program during the testing and debugging phase or auralize an application to meet auralization requirements stated at the beginning of the development cycle. As shown in Figure 5, there are at least three different ways a software developer could use LSL. An expert in the use of LSL could write an LSL specification or add LSL commands to the program itself and preprocess the program as shown in Figures 5(a) and (b). Alternately, as in Figure 5(c), a graphical interface could be used to ease the task of specifying auralization. This interface accepts and translates user commands to LSL specifications. Specifications so generated are internal to the interface and hidden from a novice user.

The result of using any of the above methods is an object program instrumented with calls to procedures from the LSL sound library. When the instrumented program is executed, these calls generate sound data that is directed to a sound module [6] via MIDI. The sound module output is sent through an amplifier to a speaker. Voice is sent via an audio processor and amplifier to the speakers. Non-voice and voice data are mixed at the amplifier and played through the same speakers. A user may organize the audio playback hardware differently.
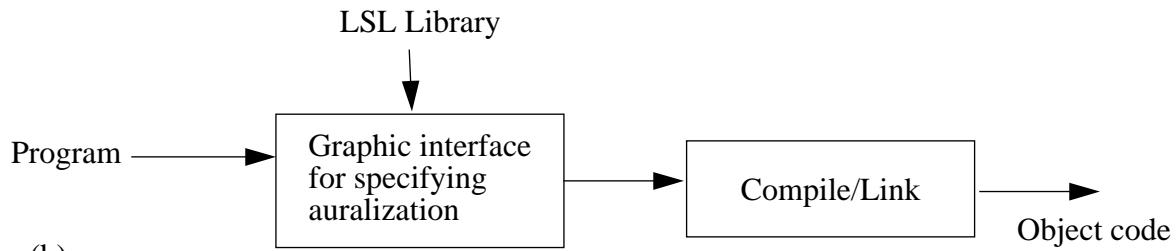
---

[6]The LISTEN system currently uses Proteus/3 World from E-mu Systems as the sound module. The Audiomedia II card from Digidesign is used for voice sampling and playback.
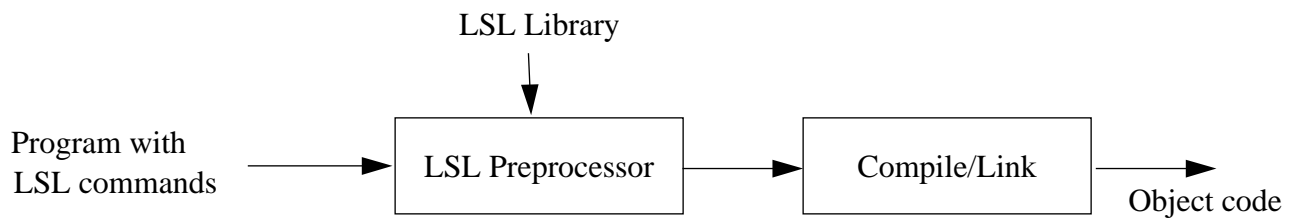
Figure 5: Use of LSL in a programming environment.

6.1 LSL editor

In principle it is possible to use any of the well known text editors, such as `gnuemacs` or `vi` to edit an LSL specification. However, in several cases such editing may be inconvenient; specially so for one not familiar with the piano keyboard or music terminology. For this reason we propose a special editor, named LSLed, for preparing LSL specifications.

LSLed has features similar to those found in syntax-directed editors on personal computers such as the Think C editor on the Macintosh [16] computer[7]. LSLed is most useful when editing sound related commands. For example, suppose that variable named *drag-icon* is declared to be of type `pattern`. We wish to assign a sound pattern to *drag-icon*. One way to do so would be to type in the pattern using the notation described earlier in Section 4.1. Thus, one may type in the following text:

drag-icon := "C2D2E2F2G2A2B2C3";

We identify two problems with this approach. First, an individual may not be familiar with the notation (e.g. one who asks the question: *What is C2?*) Second, one may be familiar with the notation but would like to experiment with various sounds and just play the sound pattern on an electronic keyboard instead of converting it to the textual notation. LSLed helps overcome these problems.

During editing, LSLed could be in one of two modes: *text* and *aural*. When in text mode, all input to LSLed is from the computer keyboard. When in aural mode, the source of input could be MIDI via an electronic keyboard or the computer keyboard. When the above assignment to variable *drag_icon* is typed, LSLed expects input from the computer keyboard until all characters up to and including the assignment (:=) operator have been typed. It then adds a double quote to indicate the starting of a string and prompts the user to play the pattern on the electronic keyboard. At this point the editor screen would display, among other characters:

drag_icon := "

As the user plays the notes on a keyboard, the textual version of each note appears following the double quotes. On completion, the user hits the return key indicating that the pattern has terminated. To change any previously entered pattern, the user merely places the cursor at the beginning of the pattern immediately following the double quotes, and repeats the above process. Textual version may also be edited by switching LSLed to textual mode. This may be useful for adding or altering note parameters such as, for example, duration or play style. The above process can be used to enter other types of sound related constants too. The advantage of this approach is clear when one considers the ease of data entry. The duration of notes played on the keyboard is determined using various LSLed parameter settings not described here.

LSLed also interprets commands in an LSL specification. For example, a play command in an

---

[7]Macintosh is a trademark of Apple Computer, Inc..
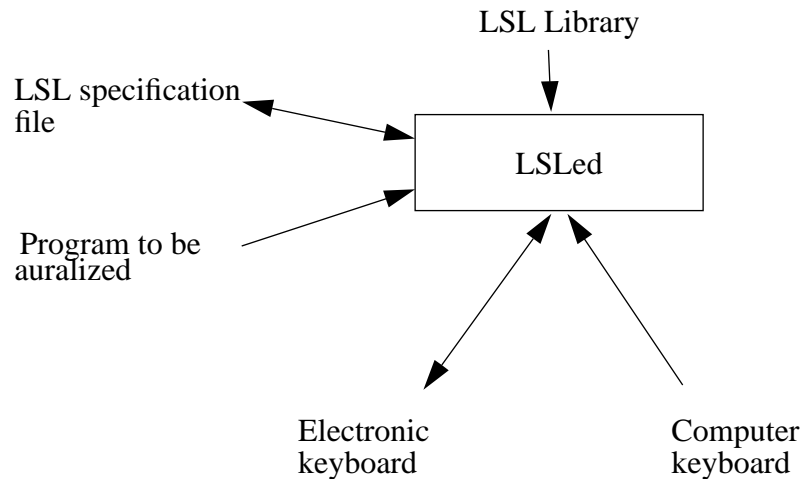
Figure 6: LSLed in a programming environment.

LSL specification can be interpreted to hear how the patterns specified in the command will sound. Interpretation of more complex commands such as `notify` and `dtrack` are also possible via the graphic interface as described in Section 7.

As LSL is a typed language, LSLed checks for any type mismatch. For example, if x is of type `note` and one types:

x:= 4;

LSLed complains about the invalid type on the right side of the assignment.

Figure 6 shows how LSLed fits in a programming environment. It takes as input an existing LSL specification file and creates an updated or a new file. LSLed obtains its input from an electronic or a computer keyboard. LSL library routines are used for the input and output of sound patterns. The program to be auralized is also an input to LSLed. This enables LSLed to check for any interaction errors between the LSL specification and the program. For example, if a `notify` command indicates that the events to be selected are within function *icon-operation*, LSLed checks if such a function indeed exists in the program. If not, then a warning is issued. In any case, checks such as these are also made during LSL specification preprocessing.

7   An LSL implementation outline

An LSL implementation consists of four software components: (i) an LSL preprocessor, (ii) an LSL editor, (iii) an LSL sound library, and (iv) an LSL graphic interface. Below we outline the elements of the preprocessor, the library, and the graphic interface. The editor has already been described above.

7.1 LSL preprocessor

The task of the preprocessor is to instrument the program to be auralized with calls to sound library procedures. The instrumentation is guided by the LSL specification. In the absence of any specifications, the preprocessor may be required to process LSL commands embedded within the program.

The preprocessor takes an LSL specification and a program P as input and produces the instrumented version P_L of P. P_L is then compiled and linked to the library routines using the traditional C compiler and linker. This generates an object program with auralization commands compiled in. The following steps provide a high level view of the sequence of actions taken by the preprocessor to transform P into P_L.

1. Parse each source file F that is input to the preprocessor. This results in a parse tree T for source file F, whose nodes are typed. Using the type information, it is possible to examine a node of T and determine what syntactic construct it represents.

   During the construction of T, identify any LSL commands. Replace each command by an appropriate call to a sound library procedure. This call now becomes a part of T.

2. Begin parsing the LSL specification if one has been supplied. If there is no LSL specification then go to step 3. Otherwise, for each `notify` or `dtrack` command encountered, say C, do the following:

   (a) Traverse T and identify syntactic constructs that match the event specification in C. To each syntactic construct S so identified, add an appropriate call to the sound library procedure that will generate the sound specified in C. This step updates T.

3. The updated parse tree, say T' contains new nodes that represent calls to sound library function calls. Deparse T' by traversing it and generating a source file F'.

4. Invoke the C compiler available in the environment with input F'. This will result in the object version of F'.

5. Delete F'.

In a UNIX environment, the above sequence of steps is initiated by the following command:

lp -spec aspec.l -file icons.c

Assuming that icons.c contains a function named *main*, the above command will result in a file named *a.out* containing the executable auralized version of *icons.c.* The auralization will be done using the LSL specifications contained in the file aspec.l. For programs spread over several files,

36

makefiles [10], and the make command can be used. The `lp` command can be embedded in the makefile. For programs that need to be auralized, calls to the C compiler may be replaced by calls to `lp`. Recall from Section 4 that applicability constraints may be used to restrict the scope of LSL specifications within a source file.

The above algorithm may perform multiple passes over the parse tree. The number of passes over the program parse tree T is equal to the number of times `notify` and `dtrack` commands are encountered while interpreting an LSL specification. Development of a more efficient single pass algorithm for auralization based on a given LSL specification is an open problem. Further, the preprocessor may not always succeed in performing correct auralization due to pointers and aliases. The problem can be resolved using labels or scope specifiers in the LSL specification to inform the preprocessor where to look for an event or data item.

7.2   Graphic interface

A graphic interface is an integral part of an LSL environment. The interface enables one to specify auralization requirements without having to learn LSL. When using this interface, a user uses pull-down and pop-up menus that guide the development of specifications. These specifications are transformed into LSL syntax by the interface and saved in a specification file just as a user would do if LSL was used directly. Once the specification is complete, the interface can be instructed to invoke the LSL preprocessor and the C compiler to instrument and auralize a program. The interface also provides an environment for executing an auralized program. Thus, after having compiled the program, it may be instructed to execute the program under programmer's control. This feature may be used to debug the program and alter auralizations when necessary.

7.3   The LSL sound library

This is a library consisting of procedures that are responsible for sending appropriate data to MIDI or other sound generation devices during program execution. The entire package to manage the processing of notes received during program execution is a part of the library. In addition, the library consists of several predefined functions callable from within an LSL specification and a C program.

8   Summary

Research efforts in program auralization appear to be on the increase. We are not aware of any general purpose method to specify program auralizations and a supporting tool to auralize a program in accordance with the specifications. We have presented the syntax and semantics of a language named LSL that provides a notation to specify a variety of program auralizations. LSL is generic and needs to be adapted to the programming language of an environment in which programs are

expected to be auralized. A language specific implementation of LSL serves as a tool to auralize programs. We have presented the design of LSL/C, a C adaptation of LSL.

Writing LSL specifications could be a tedious task. To simplify the process, we have proposed a syntax directed editor for LSL, named LSLed, and a graphic interface. LSLed is useful while writing LSL specifications directly using LSL. The graphic interface is useful for a novice to LSL. Whether one uses LSL directly or the graphic interface to specify an auralization, the result is an LSL specification. This specification serves as the basis for instrumenting the program to be auralized. The instrumentation consists of calls to procedures from LSL sound library.

The implementation of various parts of the LSL environment is currently underway as part of the LISTEN project. Once different components of this environment are available and integrated we will be sufficiently equipped to conduct experiments with the use of auralization in various program development activities. The tools are also expected to help other researchers in the area of program auralization.

Acknowledgements

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, MA, 1986.

[2] A. L. Ambler and M. M. Burnett. Influence of visual technology on the evolution of language environments. *IEEE Computer*, 22(10):9–22, 1989.

[3] Apple Computer, Inc. *Inside Macintosh, Volume VI.* Addison-Wesley Publishing Company, Reading, MA, 1992.

[4] M. H. Brown and J. Hershberger. Color and sound in algorithm animation. *Computer*, 25(12):52–63, December 1992.

[5] Digidesign, Inc. *Sound designer II: User's Guide*, 1992.

[6] A. D. N. Edwards. Soundtrack: An auditory interface for blind users. *Human-Computer Interaction*, 4(1):45–66, 1989.

[7] J. M. Francioni and J. A. Jackson. Breaking the silence: Auralization of parallel program behavior. Technical Report TR 92-5-1, Computer Science Department, University of Southwestern Louisiana,, 1992.

[8] W. W. Gaver. Using sound in computer interfaces. *Human-Computer Interaction*, 2:167–177, 1986.

[9] W. W. Gaver. The sonicfinder: An interface that uses auditory icons. *Human-Computer Interaction*, 4(1):67–94, 1989.

[10] Computer Systems Research Group. *UNIX User's Reference Manual (URM)*. USENIX Association, 4 1986.

[11] R. Kamel, K. Emami, and R. Eckert. Px: Supporting voice in workstations. *IEEE Computer*, 23(8):73–80, 1990.

[12] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[13] P. S. Langston. Little languages for music. *Computing Systems*, 3(2):193–282, Spring 1990.

[14] L. F. Ludwig, N. Pincever, and M. Cohen. Extending the notion of a window system to audio. *IEEE Computer*, 23(8):66–72, 1990.

[15] R. Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley Publishing Company, Reading, MA, 1989.

[16] Symantec. *Think C User Manual*, chapter The Editor. Symantec Corporation, Cupertino, CA, 1991.

[17] T. Thompson. Keynote - a language and extensible graphic editor for music. *Computing Systems*, 3(2):331–358, Spring 1990.

[18] E. S. Yeung. Pattern recognition by audio representation of multivariate analytical data. *Analytical Chemistry*, 52(7):1120–1123, June 1980.

APPENDIX: Syntax of LSL

A.1 LSL syntax conventions

The syntax of LSL is described below using a modified form of BNF[1]. Nonterminals are in italics, keywords in `teletype` font, and lexical symbols in **bold** font. Alternates of a nonterminal are separated by the | symbol.

1.	*lsl-spec*	$\rightarrow$	`begin auralspec`
			spec-module-list
			`end auralspec.`
2.	*spec-module-list*	$\rightarrow$	*spec-module-list spec-module*
3.			| *spec-module*
5.	*spec-module*	$\rightarrow$	`specmodule` **id**
			program-id-list
			global-interaction-list
			declarations
			spec-def-list
			VDAP-list
			`begin` **id**
			spec-def-body
			`end` **id**;
6.	*program-id-list*	$\rightarrow$	`external` *ext-id-list*;
			| ϵ
8.	*global-interaction-list*	$\rightarrow$	*global-interactions global-interaction-list*
			| ϵ
10.	*global-interactions*	$\rightarrow$	*interact-id id-list*;
11.	*interact-id*	$\rightarrow$	`import` | `export`
13.	*spec-def-list*	$\rightarrow$	*spec-def spec-def-list*
			| ϵ
15.	*VDAP-list*	$\rightarrow$	*VDAP-spec VDAP-list*
			| ϵ
17.	*VDAP-spec*	$\rightarrow$	`VDAP begin`

```
l- function
VDAP end;
```

18.	*spec-def*	$\rightarrow$	**specdef id** (*spec-par-list*)

declarations

begin id

spec-def-body

end id;

19.	*spec-def-body*	$\rightarrow$	*spec-command spec-def-body*
			| *spec-command*

21.	*spec-command*	$\rightarrow$	*named-command*
			| *unnamed-command*

23.	*named-command*	$\rightarrow$	*name-tag-list unnamed-command*

24.	*name-tag-list*	$\rightarrow$	**id** :: *name-tag-list*
			| **id** ::

26.	*unnamed-command*	$\rightarrow$	*set-globals-command*
			| *play-command*
			| *notify-command*
			| *dtrack-command*
			| *atrack-command*
			| *assign-command*
			| *loop-command*
			| *if-command*
			| *specdef-use-command*
			| *VDAP-call-command*
			| *turn-command*
			| *toggle-command*
			| *sync-command*

39.	*set-globals-command*	$\rightarrow$	**set** *global-par-list*;
40.	*global-par-list*	$\rightarrow$	*global-par-list*, *global-par*
			| *global-par*

42.	*global-par*	$\rightarrow$	*score-const-id*
			| *device-const-id*

44.	*play-command*	→	`play` *play-list*;
45.	*play-list*	→	*pattern-specifier* ‖ *play-list*
			| *pattern-specifier* && *play-list*
			| *play-list*
48.	*pattern-specifier*	→	**id**
			| **constant**
			| *specdef-use-command*
			| *VDAP-call-command*
			| *pattern-specifier play-pars*
			| (*play-list*)
54.	*play-pars*	→	`with` *tagged-list*
55.	*tagged-list*	→	*tagged-list, tags*
			| *tags*
57.	*tags*	→	*score-const-id*
			| *device-const-id*
59.	*score-const-id*	→	*score-tag = const-id*
			| `mm` *mmspec*
			| *mode-specifier*
62.	*const-id*	→	**constant**
			| *dotted-id*
64.	*dotted-id*	→	`.id`
65.	*score-tag*	→	`keysig`
			| `timesig`
67.	*device-tag-list*	→	*device-const-id , device-tag-list*
			| *device-const-id*
69.	*device-const-id*	→	*device-tag = const-id*
70.	*device-tag*	→	`chan` | `inst`
72.	*notify-command*	→	`notify` *all-selective label-parameter event-specifier*
			sound-specifier scope-specifier;
73.	*all-selective*	→	`all` | `selective` | ϵ

76.	*label-parameter*	$\rightarrow$	`label` $=$ *label-list*
			$\mid \epsilon$
78.	*label-list*	$\rightarrow$	*label-list*, **id**
			$\mid$ **id**
80.	*event-specifier*	$\rightarrow$	*event-specifier connector event*
			$\mid$ *event*
82.	*connector*	$\rightarrow$	`and` $\mid$ `or`
84.	*event*	$\rightarrow$	`rule` $=$ **id**
			$\mid$ `rule` $=$ **id**:*instance-list*
			$\mid$ `instance` $=$ *instance-list*
			$\mid$ `assertion` $=$ l- `condition`
			$\mid$ `rtime` $=$ *expression* `after` *event*
			$\mid$ (*event-specifier*)
			$\mid$ *event* (`first`)
91.	*instance-list*	$\rightarrow$	*instance-list* $\&\&$ *instance*
			$\mid$ *instance*
93.	*instance*	$\rightarrow$	**string**
94.	*sound-specifier*	$\rightarrow$	`using` *play-list*
			$\mid \epsilon$
96.	*scope-specifier*	$\rightarrow$	`in` *tagged-scope-list*
			$\mid \epsilon$
98.	*tagged-scope-list*	$\rightarrow$	*tagged-scope-list* `and` *tagged-scope*
			$\mid$ *tagged-scope*
100.	*tagged-scope*	$\rightarrow$	*scope-tag* $=$ *scope-tagid-list*
101.	*scope-tag*	$\rightarrow$	`filename` $\mid$ `func`
103.	*scope-tagid-list*	$\rightarrow$	*scope-tagid-list*, *scope-tagid*
			$\mid$ *scope-tagid*
105.	*scope-tagid*	$\rightarrow$	*selector* $\mid$ **string**
107.	*dtrack-command*	$\rightarrow$	`dtrack` *dtrack-id-list start-event-spec term-event-spec*
			sound-specifier;

108.	*atrack-command*	→	`atrack` *start-event-spec term-event-spec sound-specifier;*
109.	*start-event-spec*	→	`when` *event-specifier*
			$\mid \epsilon$
111.	*term-event-spec*	→	`until` *event-specifier*
			$\mid \epsilon$
113.	*ext-id-list*	→	*ext-id-list* , `l-` `id`
			$\mid$ `l-` `id`
115.	*dtrack-id-list*	→	*dtrack-id-list* `and` *dtrack-id*
			$\mid$ *dtrack-id*
117.	*dtrack-id*	→	`l-` `id` *init-value capture-specifier scope-specifier*
118.	*init-value*	→	`init` `=` `l-` *expression*
			$\mid \epsilon$
120.	*capture-specifier*	→	`capture` `=` **id**
			$\mid \epsilon$
122.	*mode-specifier*	→	`mode` `=` `continuous` $\mid$ `mode` `=` `discrete` $\mid$ `mode` `=` `sustain`
125.	*assign-command*	→	*selector* `:=` *expression;*
126.	*selector*	→	**id** $\mid$ **id**[*element-selector*]
128.	*element-selector*	→	expression-list
129.	*expression-list*	→	*expression-list* , *expression*
			$\mid$ *expression*
131.	*loop-command*	→	*for-loop* $\mid$ *while-loop*
133.	*for-loop*	→	`for` **id** `:=` *expression* `to` *expression step-expression*
			statement-body
134.	*step-expression*	→	`step` *expression*
			$\mid \epsilon$
136.	*while-loop*	→	`while` *condition* `do` *statement-body*
137.	*statement-body*	→	`begin` *spec-def-body* `end`
			$\mid$ *spec-command;*
139.	*if-command*	→	*if-then-command*

			| *if-then-else-command*
141.	*if-then-command*	→	**if** *condition* **then** *statement-body*
142.	*if-then-else-command* →		**if** *condition* **then** *statement-body* **else** *statement-body*
143.	*specdef-use-command*→		**id** (*actual-par-list*);
			| **id** ( );
145.	*actual-par-list*	→	*actual-par-list, actual-par*
			| *actual-par*
147.	*actual-par*	→	*expression*
148.	*spec-par-list*	→	*id-list* | ε
150.	*VDAP-call-command*→		l- id (l- actual-parameter-list);
151.	*turn-command*	→	**turn** *on-off device-tag-list;*
152.	*on-off*	→	**on**
			| **off**
154.	*toggle-command*	→	**toggle** *toggle-source* = **constant**;
155.	*toggle-source*	→	**keyboard**
			| **midi**
157.	*sync-command*	→	**syncto** *sync-to-id;*
158.	*sync-to-id*	→	**program**
			| *sync-par-list*
160.	*sync-par-list*	→	*sync-par-list, sync-par*
			| *sync-par*
162.	*sync-par*	→	**bufsize** = **const**
			| **noslow**
			| **mm**
			| **mm** *mmspec*
166.	*mmspec*	→	*duration-expression* = **const**
167.	*duration-expression*	→	*duration-expression duration-factor*
			| *duration-expression + duration-factor*
			| *duration-factor*

170.	*duration-factor*	$\rightarrow$	*duration-attribute*
			$\mid$ (*duration-expression*)
172.	*duration-attribute*	$\rightarrow$	`f` $\mid$ `h` $\mid$ `q` $\mid$ `e` $\mid$ `s`
177.	*declarations*	$\rightarrow$	*applicability const-declaration var-declaration*
178.	*applicability*	$\rightarrow$	*apply-list*
			$\mid$ ϵ
180.	*apply-list*	$\rightarrow$	*apply-list*; *apply-decl*
			$\mid$ *apply-decl*
182.	*apply-decl*	$\rightarrow$	`applyto` *tagged-scope-list*;
183.	*const-declaration*	$\rightarrow$	`const` *const-list*;
			$\mid$ ϵ
185.	*const-list*	$\rightarrow$	*const-val-pair const-list*
			$\mid$ *const-val-pair*
187.	*const-val-pair*	$\rightarrow$	**id** = **constant** ;
188.	*var-declaration*	$\rightarrow$	`var` *var-decl-list*;
			$\mid$ ϵ
190.	*var-decl-list*	$\rightarrow$	*var-type-list* ; *var-decl-list*
			$\mid$ *var-type-list*
192.	*var-type-list*	$\rightarrow$	*id-list* : *type*
193.	*id-list*	$\rightarrow$	**id** , *id-list*
			$\mid$ **id**
195.	*type*	$\rightarrow$	`int` $\mid$ `note` $\mid$ `pattern` $\mid$ `voice` $\mid$ `file` $\mid$ `ksig` $\mid$ `tsig`
			$\mid$ *array-declarator*
203.	*array-declarator*	$\rightarrow$	`array` [*range-list*] `of` *type*
204.	*range-list*	$\rightarrow$	*range-list* , *range*
			$\mid$ *range*
206.	*range*	$\rightarrow$	*expression* .. *expression*
207.	*expression*	$\rightarrow$	*expression addop term*
			$\mid$ *term*

46

| 209. | *term* | $\rightarrow$ | *term mulop factor* |
| | | | $\mid$ *factor* |

211.	*factor*	$\rightarrow$	(*expression*)
			$\mid$ *unop factor*
			$\mid$ **id**
			$\mid$ **id**(*actual-par-list*)
			$\mid$ **id**( )
			$\mid$ const

| 217. | *condition* | $\rightarrow$ | *condition relop cterm* |
| | | | $\mid$ *cterm* |

| 219. | *cterm* | $\rightarrow$ | *cterm logop cfactor* |
| | | | $\mid$ *cfactor* |

221.	*cfactor*	$\rightarrow$	*expression*
			$\mid$ (*condition*)
			$\mid$ not *cfactor*

| 224. | *addop* | $\rightarrow$ | $+ \mid -$ |

| 226. | *mulop* | $\rightarrow$ | $* \mid /$ |

| 228. | *unop* | $\rightarrow$ | $-$ |

| 229. | *relop* | $\rightarrow$ | $< \mid > \mid <= \mid = \mid >= \mid <>$ |

| 235. | *logop* | $\rightarrow$ | $\&\& \mid \|\|$ |

A.2 Lexical conventions

Using regular expressions[1] we define the lexical elements of LSL.

1. Comments are enclosed inside /* and */. Comments may not appear within a token. A comment within another comment is not allowed.

2. **char** denotes any ASCII character.

3. One or more spaces separates tokens. Spaces may not appear within tokens.

4. An **id** is a sequence of letters or digits with the first character being a letter. The underscore (_) can be used in an identifier. Upper and lower case letters are treated as being different in an **id**.

id	$\rightarrow$	(_)* *letter* (*letter*	*digit*	_)*
letter	$\rightarrow$	[a-zA-Z]
digit	$\rightarrow$	[0-9]

5. A keyword may not be used as an **id**. Upper and lower case are treated differently.

6. A constant can be an integer or a string. An integer is a sequence of digits. A string is a sequence of characters enclosed within double quotes. As a constant can be interpreted in a variety of ways in LSL, we provide below a complete grammar for constants.

1.	*constant*	$\rightarrow$	*integer*
			| *string*
			|*time-sig*
4.	*integer*	$\rightarrow$	*digit*$^+$
5.	*string*	$\rightarrow$	"*char-sequence*"
6.	*char-sequence*	$\rightarrow$	*note-sequence*
			| *key-sig*
			| *file-name*
			| *function-name*
10.	*note-sequence*	$\rightarrow$	(*note* | *.id*)$^+$
			| (*note-sequence*: *attribute-sequence*)
			| (*note-sequence*)
14.	*note*	$\rightarrow$	*note-generic note-modifier*
15.	*note-generic*	$\rightarrow$	c | d | e | f | g | a | b | r | C | D | E | F | G | A | B | R
31.	*note-modifier*	$\rightarrow$	*flat-sharp** *octave*
32.	*flat-sharp*	$\rightarrow$	b | #
34.	*octave*	$\rightarrow$	[0-8]
35.	*attribute-sequence*	$\rightarrow$	*attribute*$^+$
36.	*attribute*	$\rightarrow$	*duration tagged-value-list**
37.	*duration*	$\rightarrow$	*simple-duration*
			| (*duration-expression*)
39.	*simple-duration*	$\rightarrow$	f | h | q | e | s
			| ptime = *integer*
45.	*duration-expression*	$\rightarrow$	*duration-expression op simple-duration*
			| *simple-duration*
			| (*duration-expression*)
48.	*op*	$\rightarrow$	+ | ϵ

50. *key-sig*	→	*pre-defined*
		| *user-defined*
52. *pre-defined*	→	*note:mode*
53. *mode*	→	`major`
		| `minor`
		| `lydian`
		| `ionian`
		| `mixolydian`
		| `dorian`
		| `aeolian`
		| `phrygian`
		| `locrian`
62. *user-defined*	→	(*note-sequence*)
63. *time-sig*	→	(*beat-structure* : `int`)
64. *beat-structure*	→	*beat-structure* + `int`
		| `int`
66. *filename*	→	`char`$^+$
67. *function-name*	→	`char`$^+$
68. *tagged-value-list*	→	*tagged-value-list tagged-value*
69. *tagged-value*	→	*play-attribute-tag = constant*
70. *play-attribute-tag*	→	`chan` | `play` | `inst` | `mm` | `mm` *mmspec*

7. Interpretation of a string is context dependent. Thus, for example, when assigned to a variable of type `pattern`, the string ".cmajor C5" denotes a sequence of notes consisting of the value of the variable .cmajor followed by the note C5. The same string when used in the context `file` = ".cmajor C5" denotes a file name .cmajor C5. Notes enclosed in parentheses such as in "G3 (C4E4G4) C5" are treated as forming a blocked chord. The string "hello" results in an invalid assignment command when it appears on the right side of an assignment to a variable of type `pattern`.

8. Ambiguity may arise while defining a note sequence such as in "cbb". To avoid this, the notes may be separated by at least one space character such as in "cb b".

9. The grammar above contains some terminals prefixed by l-. Such terminals denote language specific constructs. A complete list of such terminals appears in Table 7. These terminal symbols may be nonterminals or terminals in the grammar of the language L of the auralized

Table 7: Language dependent terminals in LSL grammar.

Terminal	Meaning	Example from C
`l-condition`	Conditional expression which evaluates to *true* or *false*.	$(x < y \ \&\& \ p > q)$
`l-id`	Identifier	*drag_icon*
`l-expression`	An expression that evaluates to a value of type matching the type of the left side of the assignment in which it appears.	$(min - val * 2)$
`l- function`	A function invoked for tracking one or more variables.	Any C function definition.
`l- actual-parameter-list`	List of actual parameters.	`int` x, `int` * y

program. The LSL preprocessor attempts to parse over the strings corresponding to such symbols. These strings are parsed by the compiler for L.

A.3 Static semantics

The following constraints apply to LSL specifications. These are not indicated by the syntax.

1. All identifiers must be declared before use. Identifiers that belong to the auralized program must appear as `external`s.

2. Local attribute values, such as metronome values, channels, etc. which are specified explicitly as attributes, take precedence over corresponding global values. However, they do not alter the global values. Global values of such parameters may be set using the `set` command within an LSL specification or in the program.

3. Identifiers declared within a `specmodule` M are global to M and may be used by all `specdef`s declared within M. Identifiers declared within a `specdef` S are local to S and may not be used by other `specdef`s or in any other `specmodule`. Identifiers may be exported by an `specmodule` for use by any other module by explicitly mentioning it in an `export` declaration. A module may use an identifier exported by another module by explicitly importing it using the `import` declaration. All program variables used in an `specdef` or a `specmodule` body must be specified as `external`s. Program identifiers, global to a VDAP definition, need not be declared. However, all such identifiers must be declared in the context wherein VDAP will be placed and compiled by the C compiler.

4. A VDAP specification must be a valid C function when using LSL/C.

5. The formal and actual parameters must match in number and type between a specification definition and its use.

6. All matching `begin`s and `end`s must match in the identifiers that follow the corresponding keyword. Thus, for example, a `begin` gear which matches with an `end` change will be flagged as a warning because *gear* and *change* do not match.

7. LSL has default values for various parameters such as metronome, channel, and instrument. Table 4 lists LSL defaults.

8. The *expression* in a relative timed event must evaluate to a positive integer or else a run time warning is issued. A relative timed event is ignored if it occurs after program execution terminates.

9. A file or function specifed in a scope tag must exist for the program to be auralized.

10. While monitoring an activity or data, tracking will terminate upon program termination if the start event occurs after the terminating event.

11. An *expression* in a *range-list* must evaluate to an integer and must not contain any variable names. Subscript expressions that evaluate to a value outside the specified range are not allowed.

12. If both the initial value and the capture location are specified for a variable to be tracked, LSL will attempt to satisfy both requirements. Thus, the variable will be initialized at an appropriate point during program execution. Its value will also be captured as specified. The value captured will override any previous value of the variable.

13. The syntax of LSL allows for the naming of any command. However, only names of `notify`, `dtrack`, and `atrack` correspond to classes. Naming of other commands is permitted to allow referencing of commands while editing or reading an LSL specification.

14. Use of `toggle` may give rise to ambiguities at run time. For example, if the space key on the computer keyboard has been specified as a toggle source and the executing program requests for input data, it is not clear if the space character should be treated as a toggle request or input to the program. The user may avoid such ambiguities by selecting a toggle source that will not be required as input to the program. Alternately, the user may rely on the run time window based monitor to input toggle requests.