Simple Player Agent

CG

Contents

1 Introduction

This short text describes a system called the Simple Player Agent, or SPA. The system consists of a Java program that executes a small number of commands which collectively can be used to do some simple music processing, and another program, probably written in some other language, which sends graphics oriented messages to the Java program. In this document, the other program is taken to be a Lisp API, which presumably is being used in the service of some Lisp application.

The Java program, `JavaPlayerAgent.java`, a simple music rendering program, is detailed in Section 2 of this document. The Lisp API, `jmusic.l`, is detailed in Section 3 of this document. Section 4 provides information on how to secure and run SPA. Sections 5 and 6 present the computer programs that make up SPA.

As may be seen in Section 6, the Lisp API is a very short program. Analogous very short program may readily be written in other languages, e.g., Prolog, or Haskell, for those who might want to make use of SPA from another programming environment.

2 The Java Music Rendering Program

The Java music rendering program reads and processes a command from a data file, and then deletes the data file. The data file happens to be called `melody.text`, and must be placed in the same directory as the Java program, but none of that needs to be known by the user of the system.

The Java program processes 2 types of message:

- `PLAY` messages specify note notes that are to be rendered. These notes may variously be played on the drum track or on one of the 15 melody tracks.
- `MIDI` messages record music in a midi file called `melody.midi`.

Details can be inferred from the examples which be presented in subsequent bits of this text.

3 The Lisp API for SPA

Operational Notes

The Java application SimpleViewerAgent.jar must be running in the directory from which your Lisp program is running for the SVA system to do its thing.

The Lisp API is called `jmusic.l`, since it sends strings of JMusic commands, in essence, to the Java agent, each string contextualize within a `PLAY` or `MIDI`.

The Lisp API features just one essential bit of functionality, a "messenger", by which I mean a method which acts as a messaging agent. This method, called `send`, takes one string parameter, which is presumed to be bound to a command that can be interpreted by the Java music renderer, and writes the command/message to the file that is shared with the Java program.

In order to assure success with respect to the sonic processing, the Lisp API also includes two compilers, methods that translate list representations of command sequences to string representations of the command sequences, one for each command type. By inspecting the following slightly edited sample Lisp session, you will be able to infer the names of these compilers and the simple actions of translation that they perform.

```
bash-3.2$ clisp
...

[1]> ( load "jmusic.l" )
...

[2]> ( trace send play midi )
;; Tracing function SEND.
;; Tracing function PLAY.
;; Tracing function MIDI.
(SEND PLAY MIDI)
[8]> ( send ( play '(V0 I0 E4Q D4Q E4Q C4Q D4Q C4Q D4H) ) )
1. Trace: (PLAY '(V0 I0 E4Q D4Q E4Q C4Q D4Q C4Q D4H))
1. Trace: PLAY ==> "play V0 I0 E4Q D4Q E4Q C4Q D4Q C4Q D4H"
1. Trace: (SEND '"play V0 I0 E4Q D4Q E4Q C4Q D4Q C4Q D4H")
1. Trace: SEND ==> NIL
NIL
[9]> ( send ( midi '(V0 I0 E4Q D4Q E4Q C4Q D4Q C4Q D4H) ) )
1. Trace: (MIDI '(V0 I0 E4Q D4Q E4Q C4Q D4Q C4Q D4H))
1. Trace: MIDI ==> "midi V0 I0 E4Q D4Q E4Q C4Q D4Q C4Q D4H"
1. Trace: (SEND '"midi V0 I0 E4Q D4Q E4Q C4Q D4Q C4Q D4H")
1. Trace: SEND ==> NIL
NIL
[10]>
```

While the compilers are not essential to the functionality of the API, they add integrity to the operation of the API.

Pedagogical Thought

The Lisp API for SPA, `jmusic.l`, can best be understood when bearing a number of simple concepts that are defined with respect to the API, and the computational mechanisms that implement those concepts, in mind. These concepts and computational mechanisms are presented in the remaining segments of this section.

Melodic Notes

What is a melodic note?

A **melodic note** is a note that is modeled in terms of a pitch and a duration, which can be played on different instruments. For example, `A4Q` represents a note with pitch A4 and duration Q (A440 played as a quarter note). Sometimes you may wish, for reasons of power and flexibility, to let Lisp construct a melodic note from a pitch and duration, in which case you can call upon the note constructor: `( note pitch duration )`

```
[1]> 'a4h
A4H
[2]> ( note 'a4 'h )
A4H
```

Pitches

The pitches, as you will have noted already, are given symbolic names. For ease of referencing, as well as definitional clarity, these symbolic names are encapsulated in list to which the variable `*melodic-pitches*` is bound. There are 128 pitches.

```
[1]> *melodic-pitches*
(C0 C#0 C0 D#0 E0 F0 F#0 G0 G#0 A0 A#0 B0 C1 C#1 D1 D#1 E1 F1 F#1 G1 G#1 A1 A#1 B1
C2 C#2 D2 D#2 E2 F2 F#2 G2 G#2 A2 A#2 B2 C3 C#3 D3 D#3 E3 F3 F#3 G3 G#3 A3 A#3 B3
C4 C#4 D4 D#4 E4 F4 F#4 G4 G#4 A4 A#4 B4 C5 C#5 D5 D#5 E5 F5 F#5 G5 G#5 A5 A#5 B5
C6 C#6 D6 D#6 E6 F6 F#6 G6 G#6 A6 A#6 B6 C7 C#7 D7 D#7 E7 F7 F#7 G7 G#7 A7 A#7 B7
C8 C#8 D8 D#8 E8 F8 F#8 G8 G#8 A8 A#8 B8 C9 C#9 D9 D#9 E9 F9 F#9 G9 G#9 A9 A#9 B9
C10 C#10 D10 D#10 E10 F10 F#10 G10)
```

Durations

There are 4 note durations associated with the jmusic Lisp API. These correspond to the whole note, the half note, the quarter note, and the eighth note. Symbolically, these are reptresented `W`, `H`, `Q`, and `I`. The four symbols are encapsulated in a list to which the `*durations*` variable is bound.

```
[1]> *durations*
(W H Q I)
```

Note Lists

Sometimes, for reasons of power and flexibility, you may which to generate a list of notes from a list of pitches and a list of durations, both of equal length. There is a constructor in the API that will help you to do this: `( note-list` *pitch-list duration-list* `)`

```
[1]> ( note-list '( e3 d3 e3 c3 ) '( q q q q ) )
(E3Q D3Q E3Q C3Q)
```

Instruments

In order to sonically render a sequence of notes, you must first designate an instrument on which to play the notes. There are 128 instruments, each of which has a symbolic name: `I0`, `I1`, ... `I127`. These names are not particularly helpful. Fortunately, a meaninfully named variable is bound to each of these instrument names. The bindings are presented in a dictionary of instruments to which the variable `*instrument-dictionary*` is bound, in case you should be interested in the particulars of the bindings. A designated variable called `*instrument*` is presumed to be bound to some instrument, and that instrument is considered to be the "selected" instrument for a number of purposes. You can change the designated instrument by means of the `set-instrument` function. Finally, should you just wish to see the variable names corresponding to the instruments, you can do so by means of the `*instruments*` variable.

```
[1]> 'i23
I23
[2]> 'i40
I40
[3]> 'i0
I0
[4]> 'i32
I32
[5]> tango-accordian
I23
[6]> violin
I40
[7]> piano
I0
[8]> acoustic-bass
I32
[9]> *instrument*
I0
[10]> ( set-instrument tango-accordian )
NIL
[11]> *instrument*
I23
[12]> *instruments*
(PIANO BRIGHT-ACOUSTIC ELECTRIC-GRAND HONKEY-TONK ELECTRIC-PIANO-1 ELECTRIC-PIANO-2 HARPSICHORD
CLAVINET CELESTA GLOCKENSPIEL MUSIC-BOX VIBRAPHONE MARIMBA XYLOPHONE TUBULAR-BELLS DULCIMER
DRAWBAR-ORGAN PERCUSSIVE-ORGAN ROCK-ORGAN CHURCH-ORGAN REED-ORGAN ACCORDIAN HARMONICA
TANGO-ACCORDIAN GUITAR STEEL-STRING-GUITAR ELECTRIC-JAZZ-GUITAR ELECTRIC-CLEAN-GUITAR
ELECTRIC-MUTED-GUITAR OVER-DRIVEN-GUITAR DISTORTION-GUITAR GUITAR-HARMONICS ACOUSTIC-BASS
ELECTRIC-BASS-FINGER ELECTRIC-BASS-PICK FRETLESS-BASS SLAP-BASS SLAP-BASS-2 SYNTH-BASS-1
SYNTH-BASS-2 VIOLIN VIOLA CELLO CONTRABASS TREMOLO-STRINGS PIZZICATO-STRINGS ORCHESTRAL-STRINGS
TIMPANI STRING-ENSEMBLE-1 STRING-ENSEMBLE-2 SYNTH-STRINGS-1 SYNTH-STRINGS-2 CHOIR-AAHS VOICE-OOHS
SYNTH-VOICE ORCHESTRA-HIT TRUMPET TROMBONE TUBA MUTED-TRUMPET FRENCH-HORN BRASS-SECTION SYNTHBRASS-1
SYNTHBRASS-2 SOPRANO-SAX ALTO-SAX TENOR-SAX BARITONE-SAX OBOE ENGLISH-HORN BASSOON CLARINET PICCOLO
```

```
FLUTE RECORDER PAN-FLUTE BLOWN-BOTTLE SKAKUHACHI WHISTLE OCARINA SQUARE SAWTOOTH CALLIOPE CHIFF
CHARANG VOICE FIFTHS BASSLEAD NEW-AGE WARM POLYSYNTH CHOIR BOWED METALLIC HALO SWEEP RAIN SOUNDTRACK
CRYSTAL ATMOSPHERE BRIGHTNESS GOBLINS ECHOES SCI-FI SITAR BANJO SHAMISEN KOTO KALIMBA BAGPIPE FIDDLE
SHANAI TINKLE-BELL AGOGO STEEL-DRUMS WOODBLOCK TAIKO-DRUM MELODIC-TOM SYNTH-DRUM REVERSE-CYMBAL
GUITAR-FRET-NOISE BREATH-NOISE SEASHORE BIRD-TWEET TELEPHONE-RING HELICOPTER APPLAUSE GUNSHOT)
[13]> *instrument-dictionary*
((PIANO . I0) (BRIGHT-ACOUSTIC . I1) (ELECTRIC-GRAND . I2) (HONKEY-TONK . I3)
(ELECTRIC-PIANO-1 . I4) (ELECTRIC-PIANO-2 . I5) (HARPSICHORD . I6) (CLAVINET . I7)
(CELESTA . I8) (GLOCKENSPIEL . I9) (MUSIC-BOX . I10) (VIBRAPHONE . I11)
(MARIMBA . I12) (XYLOPHONE . I13) (TUBULAR-BELLS . I14) (DULCIMER . I15)
(DRAWBAR-ORGAN . I16) (PERCUSSIVE-ORGAN . I17) (ROCK-ORGAN . I18)
(CHURCH-ORGAN . I19) (REED-ORGAN . I20) (ACCORDIAN . I21) (HARMONICA . I22)
(TANGO-ACCORDIAN . I23) (GUITAR . I24) (STEEL-STRING-GUITAR . I25)
(ELECTRIC-JAZZ-GUITAR . I26) (ELECTRIC-CLEAN-GUITAR . I27) (ELECTRIC-MUTED-GUITAR . I28)
(OVER-DRIVEN-GUITAR . I29) (DISTORTION-GUITAR . I30) (GUITAR-HARMONICS . I31)
(ACOUSTIC-BASS . I32) (ELECTRIC-BASS-FINGER . I33) (ELECTRIC-BASS-PICK . I34)
(FRETLESS-BASS . I35) (SLAP-BASS . I36) (SLAP-BASS-2 . I37) (SYNTH-BASS-1 . I38)
(SYNTH-BASS-2 . I39) (VIOLIN . I40) (VIOLA . I41) (CELLO . I42) (CONTRABASS . I43)
(TREMOLO-STRINGS . I44) (PIZZICATO-STRINGS . I45) (ORCHESTRAL-STRINGS . I46)
(TIMPANI . I47) (STRING-ENSEMBLE-1 . I48) (STRING-ENSEMBLE-2 . I49)
(SYNTH-STRINGS-1 . I50) (SYNTH-STRINGS-2 . I51) (CHOIR-AAHS . I52) (VOICE-OOHS . I53)
(SYNTH-VOICE . I54) (ORCHESTRA-HIT . I55) (TRUMPET . I56) (TROMBONE . I57)
(TUBA . I58) (MUTED-TRUMPET . I59) (FRENCH-HORN . I60) (BRASS-SECTION . I61)
(SYNTHBRASS-1 . I62) (SYNTHBRASS-2 . I63) (SOPRANO-SAX . I64) (ALTO-SAX . I65)
(TENOR-SAX . I66) (BARITONE-SAX . I67) (OBOE . I68) (ENGLISH-HORN . I69) (BASSOON . I70)
(CLARINET . I71) (PICCOLO . I72) (FLUTE . I73) (RECORDER . I74) (PAN-FLUTE . I75)
(BLOWN-BOTTLE . I76) (SKAKUHACHI . I77) (WHISTLE . I78) (OCARINA . I79) (SQUARE . I80)
(SAWTOOTH . I81) (CALLIOPE . I82) (CHIFF . I83) (CHARANG . I84) (VOICE . I85)
(FIFTHS . I86) (BASSLEAD . I87) (NEW-AGE . I88) (WARM . I89) (POLYSYNTH . I90)
(CHOIR . I91) (BOWED . I92) (METALLIC . I93) (HALO . I94) (SWEEP . I95) (RAIN . I96)
(SOUNDTRACK . I97) (CRYSTAL . I98) (ATMOSPHERE . I99) (BRIGHTNESS . I100) (GOBLINS . I101)
(ECHOES . I102) (SCI-FI . I103) (SITAR . I104) (BANJO . I105) (SHAMISEN . I106)
(KOTO . I107) (KALIMBA . I108) (BAGPIPE . I109) (FIDDLE . I110) (SHANAI . I111)
(TINKLE-BELL . I112) (AGOGO . I113) (STEEL-DRUMS . I114) (WOODBLOCK . I115)
(TAIKO-DRUM . I116) (MELODIC-TOM . I117) (SYNTH-DRUM . I118) (REVERSE-CYMBAL . I119)
(GUITAR-FRET-NOISE . I120) (BREATH-NOISE . I121) (SEASHORE . I122) (BIRD-TWEET . I123)
(TELEPHONE-RING . I124) (HELICOPTER . I125) (APPLAUSE . I126) (GUNSHOT . I127))
```

Channels

In order to sonically render a sequence of notes, you must also designate a channel on which to play the notes. Think of a channel as an output stream on which commands to play notes, select or change instrument, change volume, and do a few other things can be written. With respect to the Lisp API, there are 15 channels, which are given symbolic names: `V0`, `V1`, ... `V8`, `V10`, ... `V15`. The missing one is used for a special purpose. A designated variable called `*channel*` is presumed to be bound to some channel, and that channel is considered to be the "selected" channel for a number of purposes. You can change the designated channel by means of the `set-channel` function. Mutliple channels exists so that you can play sonic lines concurrently. From the concurrent sonic lines, polyphony emerges!

```
[6]> *channels*
(V0 V1 V2 V3 V4 V5 V6 V7 V8 V10 V11 V12 V13 V14 V15)
[7]> *channel*
V0
```

```
[8]> ( set-channel 'v4 )
NIL
[9]> *channel*
V4
```

M-Lines, or Melodic Lines

An **m-line** is a sequence of tokens that can be rendered by JFugue (by the Java end of SPA) as a melodic sequence. More general sequences can be sent for melodic rendering, but the concept of m-line is simple, and useful.

There is a constructor for m-lines, which reliably constructs a list consisting of the designated channel, followed by the designated instrument, followed by a sequence of melodic notes: `( m-line` *melodic-note-list* `)`

```
[1]> ( m-line '( e4q d4q e4q c4q d4q c4q d4h ) )
(V0 I0 E4Q D4Q E4Q C4Q D4Q C4Q D4H)
[2]> ( m-line ( note-list '( c5 e5 d5 f5 e5 d5 c5 ) ( duplicate 'q 7 ) ) )
(V0 I0 C5Q E5Q D5Q F5Q E5Q D5Q C5Q)
```

Example: Dmitri Kabalevsky's "Little Tune"

How might you render Dmitri Kabalevsky's "Little Tune" in Lisp using jmusic? First, define a function to compute the JFugue sequence that will generate the melody. Then send a message to SVA to play the tune, and perhaps another to record it as a midi file.

The Definition

```
( defun little-tune ()
  ( let (figure1 figure2 figure3 figure4 figure5)
    ( setf figure1 '( e5q d5q e5q c5q ) )
    ( setf figure2 '( d5q c5q d5h ) )
    ( setf figure3 '( d5h d5h ) )
    ( setf figure4 '( e5q d5q e5q d5q ) )
    ( setf figure5 '( c5h c5h ) )
    ( let ( phrase1 phrase2 phrase3 phrase4)
      ( setf phrase1 ( append figure1 figure2 ) )
      ( setf phrase2 ( append figure1 figure3 ) )
      ( setf phrase3 phrase1 )
      ( setf phrase4 ( append figure4 figure5 ) )
      ( let ( half1 half2)
        ( setf half1 ( append phrase1 phrase2 ) )
        ( setf half2 ( append phrase3 phrase4 ) )
        ( let (melody)
          ( setf melody ( append half1 half2 ) )
          ( m-line melody )
        )
      )
    )
  )
)
```

The Message Sends

```
bash-3.2$ clisp
...

[1]> ( load "basic_demos.l" )
...
[2]> ( little-tune )
(V0 I0 E5Q D5Q E5Q C5Q D5Q C5Q D5H E5Q D5Q E5Q C5Q D5H D5H
E5Q D5Q E5Q C5Q D5Q C5Q D5H E5Q D5Q E5Q D5Q C5H C5H)
[3]> ( send ( play ( little-tune ) ) )
NIL
[4]> ( send ( midi ( little-tune ) ) )
NIL
```

Output from the Java Program

```
bash-3.2$ java -jar SimplePlayerAgent.jar
SVA: message = play V0 I0 E4Q D4Q E4Q C4Q D4Q C4Q D4H E4Q D4Q E4Q C4Q D4H D4H
E4Q D4Q E4Q C4Q D4Q C4Q D4H E4Q D4Q E4Q D4Q C4H C4H
SVA: message = midi V0 I0 E4Q D4Q E4Q C4Q D4Q C4Q D4H E4Q D4Q E4Q C4Q D4H D4H
E4Q D4Q E4Q C4Q D4Q C4Q D4H E4Q D4Q E4Q D4Q C4H C4H
Music saved as midi to: /Users/blue/blues/books/AI/lisp_programming/SPA/melody.midi
```

Note on the Lisp Encoding of Little Tune

I could have encoded Little Tune by means of a much short function.

```
( defun little-tune ()
  ( m-line '(e5q d5q e5q c5q d5q c5q d5h e5q d5q e5q c5q d5h d5h
     e5q d5q e5q c5q d5q c5q d5h e5q d5q e5q d5q c5h c5h)
  )
)
```

Question: What, besides length, is the essential difference between the two implementations of the function to compute the list of JFugue commands for Little Tune?

Answer: The first function posseses **structural integrity**, while the second does not.

Additional Notes

1. The definition of the `little-tune` function can be found in `jmusic_demos.l`
2. An mp3 file corresponding to the midi file that was generated for Little Tune can be found on the Simple Player Agent site.

Drums

Drums are best thought of in terms of their more or less *traditional* names. These traditional names, for all of the drums, are encapsulated into a list to which the variable `*all-drum-names*` is bound.

```
[1]> *all-drum-names*
(ACOUSTIC-BASE-DRUM BASS-DRUM SIDE-KICK ACOUSTIC-SNARE HAND-CLAP ELECTRIC-SNARE LOW-FLOOR-TOM
CLOSED-HI-HAT HI-FLOOR-TOM PETAL-HI-TOM LOW-TOM OPEN-HI-HAT LOW-MID-TOM HI-MID-TOM CRASH-CYMBAL-1
HI-TOM RIDE-CYMBAL-1 CHINESE-CYMBAL RIDE-BELL TAMBOURINE SPLASH-CYMBAL COWBELL CRASH-CYMBAL-2
VIBRASLAP RIDE-CYMBAL-2 HI-BONGO LOW-BONGO MUTE-HI-CONGA OPEN-HI-CONGA LOW-CONGO HIGH-TIMBALE
LOW-GIMBALE HI-AGOGO LOW-AGOGO CABASA MARACAS SHORT-WHISTLE LONG-WHISTLE SHORT-GUIRO LONG-GUIRO
CLAVES HI-WOOD-BLOCK LOW-WOOD-BLOCK MUTE-CUICA OPEN-CUICA MUTE-TRIANGLE OPEN-TRIANGLE)
[2]>
```

Unfortunately, most of the drums do not seem to be working. With this in mind, the traditional names of those drums that do seem to work and sound sensible with respect to their traditional name are encapsulated within a list to which the variable `*preferred-drum-names*` is bound.

```
[1]> *preferred-drum-names*
(BASS-DRUM SIDE-KICK ACOUSTIC-SNARE LOW-FLOOR-TOM CLOSED-HI-HAT HI-FLOOR-TOM LOW-TOM OPEN-HI-HAT
HI-MID-TOM HI-TOM SPLASH-CYMBAL)
[2]>
```

The preferred drum names are themselves bound to *token* drum names, symbols that can be interpreted as JMusic commands. For the record, here is the correspondence between traditional and token drum names, as recorded in the drum dictionary.

```
[1]> *drum-dictionary*
((OPEN-TRIANGLE . A6) (MUTE-TRIANGLE . G#6) (OPEN-CUICA . G6) (MUTE-CUICA . F#6)
 (LOW-WOOD-BLOCK . F6) (HI-WOOD-BLOCK . E6) (CLAVES . D#6) (LONG-GUIRO . D6)
 (SHORT-GUIRO . C#6) (LONG-WHISTLE . C6) (SHORT-WHISTLE . B5) (MARACAS . A#5)
 (CABASA . A5) (LOW-AGOGO . G#5) (HI-AGOGO . G5) (LOW-GIMBALE . F#5)
 (HIGH-TIMBALE . F5) (LOW-CONGO . E5) (OPEN-HI-CONGA . D#5) (MUTE-HI-CONGA . D5)
 (LOW-BONGO . C#5) (HI-BONGO . C5) (RIDE-CYMBAL-2 . B4) (VIBRASLAP . A#4)
 (CRASH-CYMBAL-2 . A4) (COWBELL . G#4) (SPLASH-CYMBAL . G4) (TAMBOURINE . F#4)
 (RIDE-BELL . F4) (CHINESE-CYMBAL . E4) (RIDE-CYMBAL-1 . D#4) (HI-TOM . D4)
 (CRASH-CYMBAL-1 . C#4) (HI-MID-TOM . C4) (LOW-MID-TOM . B3) (OPEN-HI-HAT . A#3)
 (LOW-TOM . A3) (PETAL-HI-TOM . G#3) (HI-FLOOR-TOM . G3) (CLOSED-HI-HAT . F#3)
 (LOW-FLOOR-TOM . F3) (ELECTRIC-SNARE . E3) (HAND-CLAP . D#3) (ACOUSTIC-SNARE . D3)
 (SIDE-KICK . C#3) (BASS-DRUM . C3) (ACOUSTIC-BASE-DRUM . B2))
[2]>
```

Percussive Notes

A **percussive note** is a note that is modeled in terms of token drum name and a duration. For example, `D3Q` represents an acoustic snare played for a quarter note.

Overloaded Constructors

The `note` constructor and the `note-list` constructor work to create percussive notes as well as melodic notes. You just have to be sure to provide the former with a drum token as its first argument, and the latter with a list of drum tokens for its first argument.

```
[1]> ( note acoustic-snare 'q )
D3Q
[2]> ( note 'd3 'q )
D3Q
[3]> ( note-list '(d3 g3 g3) '(h q q) )
(D3H G3Q G3Q)
[4]>
```

D-Lines, or Drum Lines

A **d-line** is a sequence of tokens that can be rendered by JFugue (by the Java end of SPA) as a percussive sequence. More general sequences can be sent for percussive rendering, but the concept of d-line is simple, and useful.

There is a constructor for d-lines, which reliably constructs a list consisting of the designated channel (`V9`), followed by a "fix it" rest, followed by a sequence of drum notes: `( d-line` *drum-note-list* `)`

```
[1]> ( d-line '(d3h g3q g3q) )
(V9 RW D3H G3Q G3Q)
[2]>
```

Discussion: NOTES and NOTE-LINES, M-LINES and D-LINES

As was mentioned, the `note` constructor and the `note-lines` constructor are used to create either melodic or percussive entities. How do you know whether a constructed note is melodic or percussive? Generally speaking, not by looking. For example, is potentially `D3Q` melodic or percussive. It becomes one or the other only in context. When it appears on the V9 channel it is a percussive note. When it appears on any other channel it is a melodic note.

Integrity can be added to the system by using the `m-line` and `d-line` constructors. The `m-line` constructor can perform checks to make sure that all notes emerging from the construction are valid with respect to melodic rendering. The `d-line` constructor can perform checks to make sure that all notes emerging from the construction are valid with respect to percussive rendering.

Example: "Little Rhythm"

An 8-bar rhythmic figure is encoded, and then sent for playing and midi recording to the simple player agent.

The Definition

```
( defun little-rhythm ()
  ( let ( figure1 figure2 )
    ( setf figure1
      ( list
        ( note acoustic-snare 'h )
```

```
        ( note acoustic-snare 'q )
        ( note acoustic-snare 'q )
      )
    )
    ( setf figure2
      ( list
        ( note acoustic-snare 'q )
        ( note acoustic-snare 'q )
        ( note open-hi-hat 'i )
        ( note open-hi-hat 'i )
        ( note open-hi-hat 'i )
        ( note open-hi-hat 'i )
      )
    )
    ( setf figure3
      ( list
        ( note acoustic-snare 'h )
        ( note acoustic-snare 'h )
      )
    )
    ( let ( part1 part2 )
      ( setf part1 ( append figure1 figure1 figure1 figure2 ) )
      ( setf part2 ( append figure1 figure1 figure1 figure3 ) )
      ( let (rhythm)
        ( append part1 part2 )
      )
    )
  )
)
```

The Message Sends

```
bash-3.2$ clisp
...

[1]> ( load "basic_demos.l" )
...
[2]> ( little-rhythm )
(V9 RW D3H D3Q D3Q D3H D3Q D3Q D3H D3Q D3Q D3Q D3Q A#3I A#3I A#3I A#3I
 D3H D3Q D3Q D3H D3Q D3Q D3H D3Q D3Q D3H D3H)
[3]> ( send ( play ( little-rhythm ) ) )
NIL
[4]> ( send ( midi ( little-rhythm ) ) )
NIL
[5]>
```

Output from the Java Program

```
bash-3.2$ java -jar SimplePlayerAgent.jar
```

```
SVA: message = play V9 RW D3H D3Q D3Q D3H D3Q D3Q D3H D3Q D3Q D3Q D3Q A#3I A#3I A#3I A#3I
D3H D3Q D3Q D3H D3Q D3Q D3H D3Q D3Q D3H D3H
SVA: message = midi V9 RW D3H D3Q D3Q D3H D3Q D3Q D3H D3Q D3Q D3Q D3Q A#3I A#3I A#3I A#3I
D3H D3Q D3Q D3H D3Q D3Q D3H D3Q D3Q D3H D3H
Music saved as midi to: /Users/blue/blues/books/AI/lisp_programming/SPA/melody.midi
```

Notes

1. The definition of the `little-rhythm` function can be found in `jmusic_demos.l`
2. An mp3 file corresponding to the midi file that was generated for Little Rhythm can be found on the Simple Player Agent site.

Example: "Little Thing"

Little Thing plays Little Tune and Little Rhythm concurrently!

The Definition

```
( defun little-thing ()
  ( append ( little-tune ) ( little-rhythm ) )
)
```

The Message Sends

```
bash-3.2$ clisp
...

[1]> ( load "basic_demos.l" )
...
[12]> ( little-thing )
(V0 I0 E5Q D5Q E5Q C5Q D5Q C5Q D5H E5Q D5Q E5Q C5Q D5H D5H
E5Q D5Q E5Q C5Q D5Q C5Q D5H E5Q D5Q E5Q D5Q C5H C5H
V9 RW D3H D3Q D3Q D3H D3Q D3Q D3H D3Q D3Q D3Q D3Q A#3I A#3I A#3I A#3I
D3H D3Q D3Q D3H D3Q D3Q D3H D3Q D3Q D3H D3H)
[13]> ( send ( play ( little-thing ) ) )
NIL
[14]> ( send ( midi ( little-thing ) ) )
NIL
[15]>
```

Output from the Java Program

```
bash-3.2$ java -jar SimplePlayerAgent.jar
SVA: message = play V0 I0 E5Q D5Q E5Q C5Q D5Q C5Q D5H E5Q D5Q E5Q C5Q D5H D5H
E5Q D5Q E5Q C5Q D5Q C5Q D5H E5Q D5Q E5Q D5Q C5H C5H
V9 RW D3H D3Q D3Q D3H D3Q D3Q D3H D3Q D3Q D3Q D3Q A#3I A#3I A#3I A#3I
D3H D3Q D3Q D3H D3Q D3Q D3H D3Q D3Q D3H D3H
SVA: message = midi V0 I0 E5Q D5Q E5Q C5Q D5Q C5Q D5H E5Q D5Q E5Q C5Q D5H D5H
E5Q D5Q E5Q C5Q D5Q C5Q D5H E5Q D5Q E5Q D5Q C5H C5H
V9 RW D3H D3Q D3Q D3H D3Q D3Q D3H D3Q D3Q D3Q D3Q A#3I A#3I A#3I A#3I
D3H D3Q D3Q D3H D3Q D3Q D3H D3Q D3Q D3H D3H
Music saved as midi to: /Users/blue/blues/books/AI/lisp_programming/SPA/melody.midi
```

Notes

1. The definition of the `little-thing` function can be found in `jmusic_demos.l`
2. An mp3 file corresponding to the midi file that was generated for Little Thing can be found on the Simple Player Agent site.

4 Getting and Using SpA

1. Establish a directory in which to work. The SPA Java program and the SPA Lisp program must run from within this directory.
2. Go to the following site: `http://www.cs.oswego.edu/ blue/software/`
3. Download the following files:
   - `SimplePlayerAgent.jar`
   - `jmusic.l`
   - `jmusic_demos.l`
4. Run the `SimplePlayerAgent.jar` file from a command line, by typing something like:
   `java -jar SimplePlayerAgent.jar`
5. Run Lisp from somewhere else, like an emacs shell, or another terminal window, and load the file of Lisp demos, which in turn will load the Lisp API.
6. Try a demo, perhaps enter `( demo-1 )` at the Lisp prompt.

Trace of the Using

The Shell for the Java Agent

```
bash-3.2$ ls
SimplePlayerAgent.jar jmusic_demos.l jmusic.l
bash-3.2$ java -jar SimplePlayerAgent.jar
SVA: message = play V0 I0 E5Q D5Q E5Q C5Q D5Q C5Q D5H E5Q D5Q E5Q C5Q D5H D5H
E5Q D5Q E5Q C5Q D5Q C5Q D5H E5Q D5Q E5Q D5Q C5H C5H
```

The Shell for the Lisp Process

```
bash-3.2$ clisp
...
[1]> ( load "jmusic_demos.l" )
;; Loading file jmusic_demos.l ...
;;  Loading file jmusic.l ...
;;   Loading file ../rlp.l ...
;;   Loaded file ../rlp.l
;;  Loaded file jmusic.l
;; Loaded file jmusic_demos.l
T
[2]> ( demo-1 )
NIL
```

The demo-1 Code

```
( defun demo-1 ()
  ( send ( play ( little-tune ) ) )
  nil
)
```

5 Java Music Rendering Source (SimplePlayerAgent.java)

```
/*
 * Program designed to:
 * (1) spot the existence of a file called melody.jfugue containing JFugue code
 *     prefaced by a command, either PLAY or MIDI
 * (2) PLAY or MIDI the JFugue code, in the latter case writing to "melody.midi"
 * (3) remove the melody.jfugue file
 */

package simpleplayeragent;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Scanner;
import org.jfugue.Player;

/**
 *
 * @author blue
 */
public class SimplePlayerAgent {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws FileNotFoundException {
        if ( fileExistsP() ) { deleteTheFile(); }
        for (;;) {
            if ( fileExistsP() ) {
                processTheFile();
                deleteTheFile();
            }
        }
    }

    static private String userDir = System.getProperty("user.dir");
    static private String fileName = userDir + "/" + "melody.jfugue";
    static private File midiFile = new File(userDir + "/" + "melody.midi");

    static private boolean fileExistsP() {
        try {
            File f = new File(fileName);
            return f.exists();
        } catch ( Exception e ) {
            return false;
        }
    }
```

```
static private void processTheFile() throws FileNotFoundException  {
    Scanner inputScanner = new Scanner(new FileReader(fileName));
    String command = inputScanner.next();
    String inputLine = "";
    while ( inputScanner.hasNext() ) {
        String token = inputScanner.next();
        inputLine = inputLine + token + " ";
    }
    inputLine = inputLine.trim();
    // transform inputLine to line, changing the volume commands
    String line = "";
    Scanner scanner = new Scanner(inputLine);
    while ( scanner.hasNext()) {
        String token = scanner.next();
        token = modifyIfNecessary(token);
        line = line + " " + token;
    }
    line = line.trim();
    String message = command + " " + line;
    System.out.println("SVA: message = " + message);

    Player player = new Player();
    if ( command.equalsIgnoreCase("PLAY") ) {
        player.play("RW " + line + " R"); // add a little silent padding
    } else if ( command.equalsIgnoreCase("SHOW") ) {
        System.out.println(line);
    } else if ( command.equalsIgnoreCase("MIDI") ) {
        writeMidi(player,line);
    } else {
        System.out.println("Simple Player Agent cannot do this: " + command);
    }
}

static private void deleteTheFile() {
    try {
        File f = new File(fileName);
        f.delete();
    } catch ( Exception e ) {
        System.out.println("### trouble deleting the file.");
    }

}

private static void writeMidi(Player player, String line) {
   try {
       player.saveMidi(line + " R",midiFile); // a little silent padding
       System.out.println("Music saved as midi to: " + midiFile.toString());
   } catch (IOException ex) {
       System.out.println("### I had trouble creating the midi file.");
   }
}

private static String modifyIfNecessary(String token) {
```

```
        String result = token;
        if ( volumeCommand(token) ) {
            String volume = token.substring(6);
            result = "X[Volume]=" + volume;
        }
        return result;
    }

    private static boolean volumeCommand(String token) {
        if ( token.length() > 6 ) {
            if ( token.substring(0,6).equalsIgnoreCase("volume") ) {
                return true;
            }
        }
        return false;
    }

}
```

6 Lisp API Source (jmusic.l)

```
;;;; File: jmusic.l
;;;; Line: Lisp API for doing just a few very simple musical things

;;;; More: Provided the Java application SimplePlayerAgent.jar is
;;;; running in the directory from which your Lisp program is running,
;;;; this code will perform two operations:
;;;; - it will PLAY a melody stored as a list of JFugue commands, so
;;;;   that you will hear the melody if your system generates sound
;;;; - it will MIDI a melody stored as a list of JFugue commands, by
;;;;   which I mean write a midi file to the directory in which you
;;;;   are running your Lisp program under the name "melody.midi".

;; THE FEATURED FUNCTIONALITY

( defun send (command)
  ( write-to-music-file command )
  nil
)

( defun play ( jfugue-list &aux )
  ( concatenate 'string "play " ( list-to-string-proper jfugue-list ) )
)

( defun midi ( jfugue-list &aux )
  ( concatenate 'string "midi " ( list-to-string-proper jfugue-list ) )
)

;; UTILIT

( defun list-to-string-proper ( melody )
  ( string-right-trim ")" ( string-left-trim "(" ( write-to-string melody) ) )
)

;; THE "REAL" SEND COMMAND - WRITING TO THE FEATURED FILE
( defun send ( command )
  ( with-open-file
    ( *standard-output* "melody.jfugue" :direction :output :if-exists :supersede )
    ( format t "~A~%" command )
  )
)

;; CONSTRUCTORS FOR DRUM LINES AND MELODY LINES

; CONSTRUCTOR FOR A D-LINE
( defun d-line ( d-note-list )
  ( append '(v9 rw) d-note-list )
)
```

```
; CONSTRUCTOR FOR AN M-LINE
( defun m-line ( m-note-list )
  ( append ( list *channel* *instrument* ) m-note-list )
)

;; KNOWLEDGE REPRESENTATION STUFF

;RECORD THE SIMPLE DRUM NOTES
( setf *drums* '(
  B2 C3 C#3 D3 D#3 E3 F3 F#3 G3 G#3 A3 A#3 B3
  C4 C#4 D4 D#4 E4 F4 F#4 G4 G#4 A4 A#4 B4
  C5 C#5 D5 D#5 E5 F5 F#5 G5 G#5 A5 A#5 B5
  C6 C#6 D6 D#6 E6 F6 F#6 G6 G#6 A6
))

;DURATIONS
( setf *durations* '(w h q i) )

;RECORD THE SIMPLE MELODIC NOTES
( setf *melodic-pitches* '(
  C0 C#0 C0 D#0 E0 F0 F#0 G0 G#0 A0 A#0 B0
  C1 C#1 D1 D#1 E1 F1 F#1 G1 G#1 A1 A#1 B1
  C2 C#2 D2 D#2 E2 F2 F#2 G2 G#2 A2 A#2 B2
  C3 C#3 D3 D#3 E3 F3 F#3 G3 G#3 A3 A#3 B3
  C4 C#4 D4 D#4 E4 F4 F#4 G4 G#4 A4 A#4 B4
  C5 C#5 D5 D#5 E5 F5 F#5 G5 G#5 A5 A#5 B5
  C6 C#6 D6 D#6 E6 F6 F#6 G6 G#6 A6 A#6 B6
  C7 C#7 D7 D#7 E7 F7 F#7 G7 G#7 A7 A#7 B7
  C8 C#8 D8 D#8 E8 F8 F#8 G8 G#8 A8 A#8 B8
  C9 C#9 D9 D#9 E9 F9 F#9 G9 G#9 A9 A#9 B9
  C10 C#10 D10 D#10 E10 F10 F#10 G10
))

; RECORED THE DRUM NAMES - SOME DON'T WORK
( setf *all-drum-names* '( ACOUSTIC-BASE-DRUM BASS-DRUM SIDE-KICK ACOUSTIC-SNARE HAND-CLAP
  ELECTRIC-SNARE LOW-FLOOR-TOM CLOSED-HI-HAT HI-FLOOR-TOM PETAL-HI-TOM LOW-TOM OPEN-HI-HAT
  LOW-MID-TOM HI-MID-TOM CRASH-CYMBAL-1 HI-TOM RIDE-CYMBAL-1 CHINESE-CYMBAL RIDE-BELL
  TAMBOURINE SPLASH-CYMBAL COWBELL CRASH-CYMBAL-2 VIBRASLAP RIDE-CYMBAL-2 HI-BONGO LOW-BONGO
  MUTE-HI-CONGA OPEN-HI-CONGA LOW-CONGO HIGH-TIMBALE LOW-GIMBALE HI-AGOGO LOW-AGOGO CABASA
  MARACAS SHORT-WHISTLE LONG-WHISTLE SHORT-GUIRO LONG-GUIRO CLAVES HI-WOOD-BLOCK
  LOW-WOOD-BLOCK MUTE-CUICA OPEN-CUICA MUTE-TRIANGLE OPEN-TRIANGLE
))

( setf *preferred-drum-names* '(
  BASS-DRUM SIDE-KICK ACOUSTIC-SNARE LOW-FLOOR-TOM CLOSED-HI-HAT HI-FLOOR-TOM LOW-TOM OPEN-HI-HAT
  HI-MID-TOM HI-TOM SPLASH-CYMBAL
))

; ESTABLISH A DRUM DICTIONARY
( setf *drum-dictionary* ( pairlis *all-drum-names* *drums* ) )

; BIND THE DRUM NAMES TO THE SIMPLE DRUM NOTES
( mapcar ( function set ) *all-drum-names* *drums* )
```

```
( defun drums ()
  ( dolist ( d *preferred-drum-names* )
    ( format t "~A~%" d )
  )
  nil
)

;; ENRICH THE NOTES
( setf ( symbol-plist 'b2 ) '(w b2w h b2h q b2q i b2i) )
( setf ( symbol-plist 'c3 ) '(w c3w h c3h q c3q i c3i) )
( setf ( symbol-plist 'c#3 ) '(w c#3w h c#3h q c#3q i c#3i) )
( setf ( symbol-plist 'd3 ) '(w d3w h d3h q d3q i d3i) )
( setf ( symbol-plist 'd#3 ) '(w d#3w h d#3h q d#3q i d#3i) )
( setf ( symbol-plist 'e3 ) '(w e3w h e3h q e3q i e3i) )
( setf ( symbol-plist 'f3 ) '(w f3w h f3h q f3q i f3i) )
( setf ( symbol-plist 'f#3 ) '(w f#3w h f#3h q f#3q i f#3i) )
( setf ( symbol-plist 'g3 ) '(w g3w h g3h q g3q i g3i) )
( setf ( symbol-plist 'g#3 ) '(w g#3w h g#3h q g#3q i g#3i) )
( setf ( symbol-plist 'a3 ) '(w a3w h a3h q a3q i a3i) )
( setf ( symbol-plist 'a#3 ) '(w a#3w h a#3h q a#3q i a#3i) )
( setf ( symbol-plist 'b3 ) '(w b3w h b3h q b3q i b3i) )
( setf ( symbol-plist 'c4 ) '(w c4w h c4h q c4q i c4i) )
( setf ( symbol-plist 'c#4 ) '(w c#4w h c#4h q c#4q i c#4i) )
( setf ( symbol-plist 'd4 ) '(w d4w h d4h q d4q i d4i) )
( setf ( symbol-plist 'd#4 ) '(w d#4w h d#4h q d#4q i d#4i) )
( setf ( symbol-plist 'e4 ) '(w e4w h e4h q e4q i e4i) )
( setf ( symbol-plist 'f4 ) '(w f4w h f4h q f4q i f4i) )
( setf ( symbol-plist 'f#4 ) '(w f#4w h f#4h q f#4q i f#4i) )
( setf ( symbol-plist 'g4 ) '(w g4w h g4h q g4q i g4i) )
( setf ( symbol-plist 'g#4 ) '(w g#4w h g#4h q g#4q i g#4i) )
( setf ( symbol-plist 'a4 ) '(w a4w h a4h q a4q i a4i) )
( setf ( symbol-plist 'a#4 ) '(w a#4w h a#4h q a#4q i a#4i) )
( setf ( symbol-plist 'b4 ) '(w b4w h b4h q b4q i b4i) )
( setf ( symbol-plist 'c5 ) '(w c5w h c5h q c5q i c5i) )
( setf ( symbol-plist 'c#5 ) '(w c#5w h c#5h q c#5q i c#5i) )
( setf ( symbol-plist 'd5 ) '(w d5w h d5h q d5q i d5i) )
( setf ( symbol-plist 'd#5 ) '(w d#5w h d#5h q d#5q i d#5i) )
( setf ( symbol-plist 'e5 ) '(w e5w h e5h q e5q i e5i) )
( setf ( symbol-plist 'f5 ) '(w f5w h f5h q f5q i f5i) )
( setf ( symbol-plist 'f#5 ) '(w f#5w h f#5h q f#5q i f#5i) )
( setf ( symbol-plist 'g5 ) '(w g5w h g5h q g5q i g5i) )
( setf ( symbol-plist 'g#5 ) '(w g#5w h g#5h q g#5q i g#5i) )
( setf ( symbol-plist 'a5 ) '(w a5w h a5h q a5q i a5i) )
( setf ( symbol-plist 'a#5 ) '(w a#5w h a#5h q a#5q i a#5i) )
( setf ( symbol-plist 'b5 ) '(w b5w h b5h q b5q i b5i) )
( setf ( symbol-plist 'c6 ) '(w c6w h c6h q c6q i c6i) )
( setf ( symbol-plist 'c#6 ) '(w c#6w h c#6h q c#6q i c#6i) )
( setf ( symbol-plist 'd6 ) '(w d6w h d6h q d6q i d6i) )
( setf ( symbol-plist 'd#6 ) '(w d#6w h d#6h q d#6q i d#6i) )
( setf ( symbol-plist 'e6 ) '(w e6w h e6h q e6q i e6i) )
( setf ( symbol-plist 'f6 ) '(w f6w h f6h q f6q i f6i) )
( setf ( symbol-plist 'f#6 ) '(w f#6w h f#6h q f#6q i f#6i) )
( setf ( symbol-plist 'g6 ) '(w g6w h g6h q g6q i g6i) )
```

```
( setf ( symbol-plist 'g#6 ) '(w g#6w h g#6h q g#6q i g#6i) )
( setf ( symbol-plist 'a6 ) '(w a6w h a6h q a6q i a6i) )

; note CONSTRUCTOR
( defun note (pitch duration)
  ( get pitch duration )
)

; note-list CONSTRUCTOR
( defun note-list ( p-list d-list )
  ( mapcar #'note p-list d-list )
)

;; CHANNELS
( setf *channels* '(v0 v1 v2 v3 v4 v5 v6 v7 v8 v10 v11 v12 v13 v14 v15) )
( setf *channel* 'v0 )
( defun set-channel (c)
  ( setf *channel* c )
  nil
)

;; INSTRUMENTS

; INSTRUMENT DICTIONARY
( setf *instrument-dictionary* '(
  ( PIANO . I0 )
  ( BRIGHT-ACOUSTIC . I1 )
  ( ELECTRIC-GRAND . I2 )
  ( HONKEY-TONK . I3 )
  ( ELECTRIC-PIANO-1 . I4 )
  ( ELECTRIC-PIANO-2 . I5 )
  ( HARPSICHORD . I6 )
  ( CLAVINET . I7 )
  ( CELESTA . I8 )
  ( GLOCKENSPIEL . I9 )
  ( MUSIC-BOX . I10 )
  ( VIBRAPHONE . I11 )
  ( MARIMBA . I12 )
  ( XYLOPHONE . I13 )
  ( TUBULAR-BELLS . I14 )
  ( DULCIMER . I15 )
  ( DRAWBAR-ORGAN . I16 )
  ( PERCUSSIVE-ORGAN . I17 )
  ( ROCK-ORGAN . I18 )
  ( CHURCH-ORGAN . I19 )
  ( REED-ORGAN . I20 )
  ( ACCORDIAN . I21 )
  ( HARMONICA . I22 )
  ( TANGO-ACCORDIAN . I23 )
  ( GUITAR . I24 )
  ( STEEL-STRING-GUITAR . I25 )
  ( ELECTRIC-JAZZ-GUITAR . I26 )
  ( ELECTRIC-CLEAN-GUITAR . I27 )
  ( ELECTRIC-MUTED-GUITAR . I28 )
```

```
( OVER-DRIVEN-GUITAR . I29 )
( DISTORTION-GUITAR . I30 )
( GUITAR-HARMONICS . I31 )
( ACOUSTIC-BASS . I32 )
( ELECTRIC-BASS-FINGER . I33 )
( ELECTRIC-BASS-PICK . I34 )
( FRETLESS-BASS . I35 )
( SLAP-BASS . I36 )
( SLAP-BASS-2 . I37 )
( SYNTH-BASS-1 . I38 )
( SYNTH-BASS-2 . I39 )
( VIOLIN . I40 )
( VIOLA . I41 )
( CELLO . I42 )
( CONTRABASS . I43 )
( TREMOLO-STRINGS . I44 )
( PIZZICATO-STRINGS . I45 )
( ORCHESTRAL-STRINGS . I46 )
( TIMPANI . I47 )
( STRING-ENSEMBLE-1 . I48 )
( STRING-ENSEMBLE-2 . I49 )
( SYNTH-STRINGS-1 . I50 )
( SYNTH-STRINGS-2 . I51 )
( CHOIR-AAHS . I52 )
( VOICE-OOHS . I53 )
( SYNTH-VOICE . I54 )
( ORCHESTRA-HIT . I55 )
( TRUMPET . I56 )
( TROMBONE . I57 )
( TUBA . I58 )
( MUTED-TRUMPET . I59 )
( FRENCH-HORN . I60 )
( BRASS-SECTION . I61 )
( SYNTHBRASS-1 . I62 )
( SYNTHBRASS-2 . I63 )
( SOPRANO-SAX . I64 )
( ALTO-SAX . I65 )
( TENOR-SAX . I66 )
( BARITONE-SAX . I67 )
( OBOE . I68 )
( ENGLISH-HORN . I69 )
( BASSOON . I70 )
( CLARINET . I71 )
( PICCOLO . I72 )
( FLUTE . I73 )
( RECORDER . I74 )
( PAN-FLUTE . I75 )
( BLOWN-BOTTLE . I76 )
( SKAKUHACHI . I77 )
( WHISTLE . I78 )
( OCARINA . I79 )
( SQUARE . I80 )
( SAWTOOTH . I81 )
( CALLIOPE . I82 )
```

```
  ( CHIFF . I83 )
  ( CHARANG . I84 )
  ( VOICE . I85 )
  ( FIFTHS . I86 )
  ( BASSLEAD . I87 )
  ( NEW-AGE . I88 )
  ( WARM . I89 )
  ( POLYSYNTH . I90 )
  ( CHOIR . I91 )
  ( BOWED . I92 )
  ( METALLIC . I93 )
  ( HALO . I94 )
  ( SWEEP . I95 )
  ( RAIN . I96 )
  ( SOUNDTRACK . I97 )
  ( CRYSTAL . I98 )
  ( ATMOSPHERE . I99 )
  ( BRIGHTNESS . I100 )
  ( GOBLINS . I101 )
  ( ECHOES . I102 )
  ( SCI-FI . I103 )
  ( SITAR . I104 )
  ( BANJO . I105 )
  ( SHAMISEN . I106 )
  ( KOTO . I107 )
  ( KALIMBA . I108 )
  ( BAGPIPE . I109 )
  ( FIDDLE . I110 )
  ( SHANAI . I111 )
  ( TINKLE-BELL . I112 )
  ( AGOGO . I113 )
  ( STEEL-DRUMS . I114 )
  ( WOODBLOCK . I115 )
  ( TAIKO-DRUM . I116 )
  ( MELODIC-TOM . I117 )
  ( SYNTH-DRUM . I118 )
  ( REVERSE-CYMBAL . I119 )
  ( GUITAR-FRET-NOISE . I120 )
  ( BREATH-NOISE . I121 )
  ( SEASHORE . I122 )
  ( BIRD-TWEET . I123 )
  ( TELEPHONE-RING . I124 )
  ( HELICOPTER . I125 )
  ( APPLAUSE . I126 )
  ( GUNSHOT . I127 )
  ))

; ESTABLISH BINDINGS RELATING TO THE INSTRUMENTS
( setf *instruments* ( mapcar ( function car ) *instrument-dictionary* ) )
( setf instrument-symbols ( mapcar ( function cdr ) *instrument-dictionary* ) )
( mapcar ( function set )  *instruments* instrument-symbols )
( setf *instrument* piano )

; FUNCTIONALITY TO CHANGE THE INSTRUMENT
```

```
( defun set-instrument ( i-name )
  ( setf *instrument* i-name )
  nil
)

;; VOLUME FUNCTIONALITY
( setf minn 'volume5000 )
( setf shhh 'volume7000 )
( setf soft 'volume9000 )
( setf modd 'volume11000 )
( setf loud 'volume13000 )
( setf maxx 'volume15000 )
```

7 JMusic Demos (jmusic_demos.l)

```
( load "jmusic.l" )

( defun little-tune ()
  ( let (figure1 figure2 figure3 figure4 figure5)
    ( setf figure1 '( e5q d5q e5q c5q ) )
    ( setf figure2 '( d5q c5q d5h ) )
    ( setf figure3 '( d5h d5h ) )
    ( setf figure4 '( e5q d5q e5q d5q ) )
    ( setf figure5 '( c5h c5h ) )
    ( let ( phrase1 phrase2 phrase3 phrase4)
      ( setf phrase1 ( append figure1 figure2 ) )
      ( setf phrase2 ( append figure1 figure3 ) )
      ( setf phrase3 phrase1 )
      ( setf phrase4 ( append figure4 figure5 ) )
      ( let ( half1 half2)
        ( setf half1 ( append phrase1 phrase2 ) )
        ( setf half2 ( append phrase3 phrase4 ) )
( let (melody)
          ( setf melody ( append half1 half2 ) )
          ( m-line melody )
)
      )
    )
  )
)

( defun little-rhythm ()
  ( let ( figure1 figure2 )
    ( setf figure1
      ( list
( note acoustic-snare 'h )
( note acoustic-snare 'q )
( note acoustic-snare 'q )
      )
    )
    ( setf figure2
      ( list
( note acoustic-snare 'q )
( note acoustic-snare 'q )
( note open-hi-hat 'i )
( note open-hi-hat 'i )
( note open-hi-hat 'i )
( note open-hi-hat 'i )
      )
    )
    ( setf figure3
      ( list
( note acoustic-snare 'h )
```

```
( note acoustic-snare 'h )
      )
    )
    ( let ( part1 part2 )
      ( setf part1 ( append figure1 figure1 figure1 figure2 ) )
      ( setf part2 ( append figure1 figure1 figure1 figure3 ) )
      ( let (rhythm)
( d-line ( append part1 part2 ) )
      )
    )
  )
)

( defun little-thing ()
  ( append ( little-tune ) ( little-rhythm ) )
)

; demo to play Little Tune
( defun demo-1 ()
  ( send ( play ( little-tune ) ) )
  nil
)

; demo to play Little Rhythm
( defun demo-2 ()
  ( send ( play ( little-rhythm ) ) )
)

; demo to play Little Thing
( defun demo-3 ()
  ( send ( play ( little-thing ) ) )
)
```