BitSequencer

CS3480

Contents

1 Introduction

BitSequencer is a music language that interfaces with the Java MIDI system. A user can declare channels which correspond to MIDI channels, assign instruments to these channels, and then play patterns on these channels either independently or in parallel.

Patterns consist of strings which denote lists of notes with their octave and length, including chords, and patterns can be appended to other patterns, repeated an arbitrary amount of times, and modulated. It supports control flow and selection in the form of `if`, `if/else`, `while`, `for` and `switch`.

The program creates multiple instances of a modified `MiniMusicPlayer`, which are stored internally by the `_DECLARE_CHANNEL` directive (to save on the slow construction time of a `MiniMusicPlayer` instance) and deleted by the `_DELETE_CHANNEL` directive. Various internal maps map integer channel identifiers to MIDI instrument numbers, phrases, and `MiniMusicPlayer` instances.

For playing multiple channels concurrently, a thread is created for each channel to be played and the thread is passed the channel's mapped music player. Over time, the music slowly goes out of time - but I'm not sure if this is because the audio threads are slow or if there's a miscalculation somewhere to do with note duration.

2 Points of Interest

2.1 Example programs

The most interesting example program is `jump.str`, which demonstrates the full power of BitSequencer: every control flow structure and every domain-specific operation and phrase, including with chords. It composes an plays a very abbreviated version of *Jump* by Van Halen.

The other programs are much simpler and less interesting: `test.str` plays a strange latin-style loop, `switch.str` tests switch and nested switch statements, and `whatdrumsarewhatnotes.str` loops over the entire MIDI range to try and figure out whether you can get different drum sounds from the drum instrument (you can't!).

2.2 Domain-specific operations

One domain-specific operation I'm proud of is the `playConcurrent` constructor in the eSOS rules. This calls a recursive rewrite on a list of channels and phrases, achieved by having a list node structure which can recurse on itself. These rules load the channels

and phrases into an internal map before firing off each channel in parallel and clearing the map. The attribute action equivalent also makes use of a recursive grammar to achieve the same effect.

2.3 Control Flow

I added two further control flow structures to BitSequencer past the while loop: a switch statement (without breaks or fall-through behaviour) and a for loop.

The eSOS rules for the switch statement are fairly complex, containing three constructors (9 rules total for various recursive rewrites and pattern matches):

- A `switch()` constructor of arity 2, taking a control expression as its first argument and a `switchBody()` as its second.
- Two `switchBody()` constructors, one of arity 2 taking an expression to be compared against the control expression as its first argument and a statement as its second, and one of arity 3 taking these same arguments plus another variable `_switchBody`.

The eSOS rules for the for loop are much more simple:

- A rule `-forLoop` that rewrites the 4-arity constructor `for` to a sequence that runs the initialisation statement, and rewrites the last three arguments to `forBody(...)`
- A rule `-forBody` that uses the existing `if` rules to execute the loop body and incrementation before recursing on itself if the boolean condition is true.

The attribute action grammar implements switch statements using a `switch` and recursive `switch_body` grammar, using inherited attributes to pass the value of the control variable down to each switch case.

A point of interest for the for loop grammar is that the action attribute grammar has a slightly different semantics to eSOS rules. The initialisation statement is written with a non-terminal `for_init` that derives either an assignment statement or an expression, whereas the eSOS rules allow the initialisation statement to be any statement (e.g., the programmer could write something like `for(while(true)...; i = 0; i = i + 1` using the eSOS grammar if they really wanted).

It's a small issue - but it does highlight that it can be easier to do things a certain way using an attribute grammar versus eSOS: I found it easier to add new, separate types of statements in the attribute action grammar, whereas with eSOS I didn't want to write lots of different rules for different types of statements.