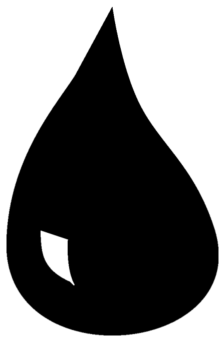
Proposal: Drumming Language **(DruL)**

COMS W4115: Programming Language and Translators

Team Leader: Rob Stewart (rs2660) Thierry Bertin-Mahieux (tb2332)
Benjamin Warfield (bbw2108) Waseem Ilahi (wki2001)

September 24, 2008



1 Introduction

DruL is a programming language designed for composing drum music. Unlike other more general-purpose music programming languages (ChucK, SuperCollider, Nyquist, Haskore), DruL's focus is on defining and manipulating beat patterns and is unconcerned with pitches, sound durations, or audio effects. DruL is mainly an imperative programming language, however it borrows ideas (map and filter) from the functional paradigm. In additions to integers, DruL's main datatypes are pattern and clip. Instruments are defined as constants.

A pattern is essentially an object that holds binary, discrete, time-series data. At each discrete-time step, which will henceforth refer to as a beat, there is either a note or a rest. For the non-musically inclined, a note represents sound produced by the striking of a drum (or similar instrument) and a rest represents the absence of any such sound. Patterns are immutable. When a pattern is manipulated, the target pattern remains intact and a new copy is created.

An instrument is one of a pre-defined set of sounds (e.g. drum notes) that can occupy a single beat.

A clip is a mapping of patterns to instruments. Clips are processed in sequence as the program runs to produce output which may be audio, sheet-music notation, or a MIDI file.

DruL is a strictly and staticly typed language. However, types are not explicitly declared, they are inferred.

DruL programs do not contain any loops or user-defined functions. All pattern and clip creation and manipulation is done using the map construct described below.

2 Language specification

There are 3 data types in DruL: **int**, **pattern**, and **clip**.

Keywords and arguments are white space delimited. Indentation is not significant.

Anything remaining on a line after // is a comment will be ignored by the compiler.

A *map* takes one or many patterns, and iterates over beats on all of them at the same time, from the first beat to the last beat of the longest sequence.

A *map* returns a pattern (that can be empty). Inside the map, the last pattern created is the one appended to the returned pattern.

Here is the list of the 16 reserved keywords:

```
NULL
if
elseif
else
rand
pattern
concat
slice
clip
instruments
length
map
mapper
print
output
return
```

Scopes: There is a general scope, and one scope per *map* (or *mapper*). Variables in the general scope can not be seen from within a *map*. Variables defined in a *map* are garbage collected at the end of the *map*.

3 Quick tutorial

In this section we give examples of what DruL code will look like, in the form of a tutorial.

3.1 Integers

Integers are part of our language. Unlike patterns and clips, they are mutable.

```
a = 3;
b = a + 2;
c = b * 12;
```

3.2 Pattern

Patterns are the data type the programmer will likely spend most of their time dealing with. For convenience, the programmer can supply a string constant made up of 1s and 0s, which will be translated into a pattern: if the character is a 1, there is a note on the corresponding beat; if 0, a rest.

```
p1 = pattern("101010");
```

Patterns can be concatenated to form new patterns:

```
pcat = concat(p1 pattern("111000") pattern("1"));
```

pcat will be equal to 1010101110001.

There is also a shortcut to concatenate the same pattern many times:

```
pcat2 = concat(p1 p1 p1);
pcat3 = pattern("101010").repeat(3);
pcat4 = p1.repeat(3);
```

pcat2, *pcat3*, and *pcat4* are all equivalent.

3.3 Map

Of course, we will not hardcode every pattern we want to create. We use map to create meaningful new patterns from existing ones:

```
p2 = map (p1)
{
    if (p1.note) { pattern("11"); }
    else         { pattern("0");  }
};
```

This will create the following pattern: 110110110. The goal of a map is to easily iterate over a pattern. *p1.note* returns *true* if there is a note on the current beat, *false* otherwise. If you call map on multiple patterns that are not of the same length, the shorter patterns will be padded with *NULL* beats.

3.4 Mapper

For ease of use, you can define a *mapper* that contains the behaviour used by *map*. We create *p3*, which is the same as *p2*:

```
mapper myMapper (p1)
{
    if (p1.note) { return pattern("11"); }
    else         { return pattern("0");  }
}

p3 = map (p1) myMapper;
```

mapper will be very important when building a standard library for the language.

3.5 More complex examples

Now that we have a proper syntax, let's get to more complicated examples. We introduce 2 new features that can be used inside a *map*: *prev* and *next*. They give you access to earlier and later beats in a pattern, using the syntax *p.prev(n)* and *p.next(n)*.

reduction: accelerate by cutting one beat out of two

```
downbeats = pattern("1000100010001000");
alternate_beats = pattern("10").repeat(8);
downbeat_diminution = map(downbeats alternate_beats)
{
    if     (alternate_beats.rest) { return pattern("");  } // pattern of length 0
    elseif (downbeats.note)       { return pattern("1"); }
    else                          { return pattern("0"); }
}
```

output is: 10101010.

improved reduction: putting a rest (0) only if the 2 original beats were rest

```
// this will map "1001100110011001" to "11111111", rather than "10101010"
one_and_four = pattern("1001100110011001");
alternate_beats = pattern("10").repeat(8);
improved_diminution = map(one_and_four alternate_beats)
{
    if     (alternate_beats.rest)      { return pattern("");  } // still required
    elseif (one_and_four.note)         { return pattern("1"); }
    elseif (one_and_four.next(1).note) { return pattern("1"); }
    else                               { return pattern("0"); }
};
```

3.6 Instruments and Clips

Now that we have a large and varied collection of patterns, we can show how to combine those patterns into clips.

Before we define any clips, we must tell the compiler what instruments they will use. This can only be done once per program, and uses the *instruments* function:

```
instruments(hihat bassdrum crash snare);
```

Once the instruments are defined, we can create a clip from our existing patterns, using an associative-array notation:

```
clip1 = clip
(
    bassdrum = downbeats
    hihat    = alternate_beats
);
```

The same result can be achieved by simply listing the patterns for each instrument in the order they are defined in the *instruments* declaration:

```
clip2 = clip
(
    alternate_beats
    downbeats
    // remaining instruments have an empty beat-pattern
);
```