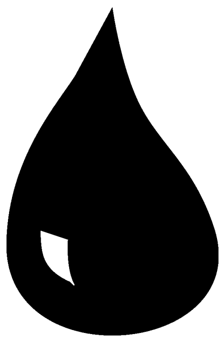
DruL Reference Manual

COMS W4115: Programming Language and Translators

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

October 22, 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. See below for further details.

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 Manual

2.1 Lexical Conventions

2.1.1 Comments

Comments in DruL start with the token "//" and continue until the end of the current line. DruL has no multi-line comment syntax.

2.1.2 Whitespace

Space, tab, end of line, and return are all considered the same and their only use is to seperate tokens.

2.1.3 Characters

DruL uses the ASCII character set.

2.1.4 Identifiers

An identifier consists of any uppercase or lowercase character or an underscore, followed by any sequence of uppercase or lowercase characters, underscores, and digits (0 through 9). The maximum length of an identifier is 64 characters.

In addition, within the context of a mapper, special variables \$1 through \$*n* (where *n* is the number of patterns passed to the mapper) are defined as read-only aliases (see section 2.5.2 for more details on this feature).

All identifiers in a given scope, be they mapper names, variables, or built-in functions, belong to a single namespace.

2.1.5 Keywords

return	rand	clip	mapper
if	pattern	instrument	print
elseif	concat	map	output
else	true	false	

2.2 Types

There are 3 basic types in DruL: **integer**s, **pattern**s, and **clip**s. In addition, 'string' constants may be used in DruL source code, but there is no variable type to which they can be directly assigned. Likewise, boolean expressions exist, but cannot be assigned to variables. Values in DruL are strongly typed, but the type of a variable is determined dynamically.

2.2.1 integer

All integers are base 10, and may optionally be preceded by a sign (`+ -`). Any sequence of digits (0 through 9) is valid. Leading 0s are ignored, so a sequence such as 0000123 is interpreted as 123. Integers are mutable.

2.2.2 pattern

A pattern is essentially an object that holds binary, discrete, time-series data. At each discrete-time step, which will henceforth be referred 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. The length of a patten can be any non-negative integer.

2.2.3 beat

A beat is a lightweight object that cannot be created directly by the user: it exists only within a mapper (for more discussion of which, see section 2.5.2 below). It gives direct access to information about a single beat of a **pattern** object (including the beats surrounding it).

2.2.4 clip

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 plain-text or a MIDI file. Clips are immutable.

2.2.5 string

A string constant begins with an ASCII double-quote character, continues with an arbitrary sequence of ASCII characters other than \ and ″, and concludes with another ″ character. If a \ or ″ character is desired, it can be escaped using the \ character.

2.3 Statements

In the most common case, a statement consists of a single expression followed by a semicolon (“;”). Importantly, unlike many languages with similar syntax, DruL does *not* consider a block to be equivalent to a statement. Instead, statements in DruL take one of the four forms below.

2.3.1 Expression Statements

The basic form of statement, as in most C-like languages, is the expression statement:
expression-statement: *expression***;**

The precedence table for operators in DruL is given here:

Operators	Notes	Associativity
.	Method call	left to right
− !	Unary minus and logical negation	right to left
∗ / %	Standard C meanings	left to right
+ −	Addition/subtraction	left to right
< <= > >=		
! = ==		
&&		
||		

The subsections that follow use the model of the C Language Reference Manual to indicate the various types of expression. As in that example. the highest-precedence forms of expression are listed first. Since much of the material below is extremely straightforward, plain-English descriptions are supplemented by grammatical descriptions only when necessary.

Primary Expressions A primary expression consists of a constant (integer or string), an identifier, or a parenthesized expression.

Function and Method calls Functions and method calls associate right to left. Depending on the function or method in question, they may return values of any type, including boolean values (which cannot be assigned to variables). Arguments to function and method calls are passed by value, not by reference.

arglist: **(** *expression* **)** | **(** *expression* <SPACE> *arglist* **)**

method-call : *identifier* **.** *identifier arglist*

function-call : *identifier arglist* | **map** *arglist mapper*

mapper : *identifier* | *block*

block : **{** *statement-list* **}**

statement-list: *statement* | *statement statement-list*

Unary operations The unary operations in DruL are arithmetic and logical negation (unary $-$ and !). Since DruL is strictly typed, arithmetic negation can only be applied to integer values, and logical negation to boolean values.

Standard arithmetic operations Expressions may use the standard binary arithmetic operators ($+$, $-$, $*$ and $/$), with their standard precedence. It is required that both of the operands in such an expression be integer values.

Comparison operations As in most C-family languages (and as shown in the precedence table above), relational operators have precedence over equality tests. These operators return boolean values, which can be used in **if** statements but cannot be assigned to variables.

Relational tests may be used on integer values only; equality tests can be used on variables of any type, but in the case of patterns and clips, they will only report whether the two variables being tested are aliases of the same object, not any deeper notion of equivalence.

Logical combination operations Here again we follow the conventions of C, and give && precedence over ||. These operators require their operands to be boolean values, and return boolean values.

2.3.2 Assignment Statements

Assignment in DruL is not a simple operator to be placed in the middle of an expression. Rather, it is a separate type of statement, which may appear anywhere another statement may appear.

$$\textit{assignment-statement}\text{: } \textit{identifier} = \textit{expression-statement}$$

Assignment is polymorphic: the same syntax is used to assign variables to integers, patterns and clips. Furthermore, due to DruL's dynamic typing, a variable may be reassigned to a different type.

2.3.3 Selection Statements

Selection statements in DruL take the following form: the string **if**, followed by an expression that returns a boolean result, enclosed in parentheses, followed by an open-brace ("{"), one or more statements, and a close-brace ("}"). This may optionally be followed by one or more **elseif**s, which are also followed by parentheses and a block of statements, and one (optional) **else**, which omits the test expression but is also followed by a block of statements.

selection-statement : **if (** *boolean-expression* **)** *block if-tail*

if-tail : ϵ | *if-middle* | *if-middle* **else {** *statement-list* **}**

if-middle : ϵ | **elseif (** *boolean-expression* **) {** *statement-list* **}** *if-middle*

2.3.4 Mapper Definition Statements

A mapper definition consists of the word **Mapper**, followed by an identifier, followed by a parenthesized list of space-separated identifiers, followed by a block.

mapper-definition : **mapper** *identifier namelist block*

namelist: **(** *identifier* **)** | **(** *identifier* <SPACE> *namelist* **)**

2.4 Blocks, namespace and scoping

2.4.1 Blocks

DruL has a limited block structure: only in the context of an **if/elseif/else** sequence or a Mapper Definition statement is a new block needed or allowed. In these cases, curly braces ("{}") are used to delimit the statement-sequence that falls within the block, and they must contain one or more statements.

Mapper definitions define a new closed scope (one from within which externally defined variables are not visible); **if** blocks do not define a new scope, so all variables used within them are visible to the enclosing block, and vise-versa.

2.4.2 Namespace

DruL has one namespace shared by variables, built-in functions and mapper names. Additionally, each type has an associated namespace for methods.

2.4.3 Scoping

DruL has one top level scope and one scope per each mapper (named or anonymous). Mapper scopes may be nested. Each scope has read-only access to all variables and mapper names defined in the scopes above it in the scope hierarchy. This allows for recursive mappers.

2.5 Patterns and pattern operations

2.5.1 Patterns

A pattern is a sequence of beats. Each beat can be a note or a rest. To define a pattern, DruL uses '0' for rests and '1' for notes. A pattern can be created in the following way:

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

which represents the sequence note, rest, note, rest, note, rest. Its length is 6.

There are built-in functions and methods on patterns included in DruL.

Patterns can be concatenated to form new patterns. The **concat** function can take any positive number of pattern arguments. Patterns are concatenated from left to right.

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

pcat will be equal to 1010101110001.

The **repeat** method is a shortcut to concatenate the same pattern many times:

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

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

The **length** method gives the length of a pattern.

```
  len = p1.length();
```

The value of *len* is 6.

The **slice** method gives you a subpattern from a pattern. It takes two arguments: first is index (starting at 1) and second is length of the desired subpattern. Requesting a subpattern out of range will raise an error. Example:

```
  psub = pattern("101010").slice(2 3);
```

psub is ”010”.

Finally, you can have an empty pattern of length 0:

```
  p8 = pattern("");
```

2.5.2 Map

The **map** construct is used to create new patterns from existing ones. **map** performs an operation iteratively on a set of patterns. The beats in the patterns are iterated over from left to right. The output of a map is a new pattern. For example:

```
p9 = pattern("101");
p10 = map(p9)
{
    if ($1.note()) { return pattern("11"); }
    else           { return pattern("0");  }
};
```

p10 is “11011”.

map takes a sequence of pattern arguments and followed by a mapper function. In the above example the mapper function is defined anonymously within curly braces.

Within a mapper function, the current beat of each pattern argument is aliased to the special mapper variables $1, $2, $3... and so on. This notation is mandatory in anonymous mapper functions such as the example above. If you use $N while there is fewer than N arguments, DruL will raise an error.

DruL uses the **beat** methods **note**, **rest** and **null** to check whether the current beat is a note, a rest, or null. *$1.note* returns **true** if there is a note on the current beat in the first pattern argument, and **false** otherwise.

DruL uses the **beat** methods **prev** and **next** to access the previous and following beats of the pattern to which a given beat belongs. These methods can be passed a single argument which specifies how far forward or back in the pattern to go. For example:

```
p11 = map(pattern("1101")
{
    if ($1.note() && $1.next(1).note()) { return pattern("1"); }
    else                                { return pattern("0"); }
};
```

p11 is “1000”. **next** may return a NULL beat as it does when called in the last iteration of the above example. When used with a NULL beat, both the **note** and **rest** methods will return *false*.

If you call map on multiple patterns that are not of the same length, the shorter patterns will be padded with NULL beats.

Each new pattern constructed by map begins as an empty string. As the pattern arugments are iterated over, the return values of the mapper function (which are also patterns) are concatenated onto the end of the new pattern.

By default, an empty pattern is returned.

Variables defined in a mapper function are garbage collected at the end of the map.

2.5.3 Mapper

Mapper functions may also be defined with a name, to be used elsewhere in the program.

For example, the above example could have been written in the following way:

```
mapper mymapper (p)
{
    if (p.note) { return pattern("11"); }
    else        { return pattern("0");  }
}
p10 = map(p9) mymapper;
```

Recall from section 2.3.4 that a Mapper definition includes a name for the mapper and a *namelist* of formal arguments. When a named mapper is used in a **map** call, each pattern that is passed to the **map** is associated with the corresponding name in the *namelist* in the mapper's definition. Then, within the body of the mapper, the current beat of each pattern is aliased to that name, as well as to "$\$n$".

A mapper function must be defined before it is used.

2.6 Clips

2.6.1 Instruments

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. This function can take a variable number of arguments. Each argument is the name of an instrument to be defined. In the example below, four instruments are defined:

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

Instruments must be defined before any clips have been defined. This function can only be called once.

2.6.2 Clips

A clip represents a collection of patterns to be played in parallel, where each pattern is played on a single instrument.

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 using the **instruments** function:

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

The patterns passed into clips are passed by value, not by reference.

2.7 Outputs

DruL has two kinds of outputs, the first one displays a representation of some data, the second one transforms a clip into a more complex representation, like MIDI.

2.7.1 Standard output

The **print** statement displays any type to the standard output, including strings. For example:

```
  print ("DruL");
  print (pattern("01"));
```

The representation of a string is the string itself. The representation of a pattern is the string that would have been used to initialize the pattern. For example, if we have a pattern

```
  p = pattern("01").repeat(2);
  print(p);
```

The output is ”0101”;

The **print** function always include a plateform appropriate line ending.

2.7.2 Text

Similar as **print**, DruL can output the representation of any type to a file. The command is:

```
output.txtfile("myfile.txt","DruL");
output.txtfile("myfile.txt",pattern("01"));
```

The file being written to is truncated if it exists, and created if it does not exist.

2.7.3 MIDI

The function **output.midi(”my.file”,clip,tempo)** outputs a clip as a MIDI file with filename ”my.file”. **tempo** is a positive integer that represents the beats per minute (BPM). The transformation from clip to MIDI may rely on external libraries like MIDGE[1]. There is no guarantee on which of the three existing MIDI formats is used. DruL tries to match its instrument definition with MIDI instruments definitions using the names. If no match can be found, DruL will use a default MIDI instrument (first one is cow bell).

[1] `http://www.undef.org.uk/code/midge/`

3 Example Code

The following example DruL code demonstrates the use of the basic functions of the language. Consult the definitions above for verification purposes.

```
//This code manipulates some patterns, associate them to instruments and
//sends them to outputs.
//First the Instrument definition. It has to be done before
//any clips are created, otherwise there will be an error.

instruments(hihat bassdrum crash snare); //define four instruments

//Integer variables used as tempos for clips.

a = 350;
b = 300;

//Patterns.

p1 = pattern("100100100");
p2 = pattern("");//empty pattern
p3 = pattern("0");//pattern with only one ’rest’ in it.
p4 = pattern("1");//pattern with only one ’note’ in it.

//p_concat is essentially concatenation of three patterns.

p_concat = concat(p1 pattern("11110000") pattern("00011"));

//Make a new pattern using above patterns and
//the library methods ’repeat’ and ’slice’.

p_custom = concat( p2 p3.repeat(2) p4.repeat(3)
    p3.repeat(2) p4.repeat(4) p_concat );
p_custom_new = concat(p_custom p3.repeat(2) p_concat p4.repeat(3));
p_new = concat( p_custom_new.slice(4,10)
    p_concat.slice(5,p1.length) p3.repeat(7) );
```

```
//Now some complex pattern manipulation.

//New Patterns.

alternate_beats = pattern("10").repeat(8);
P_concat_new = concat(p_concat p_custom);

//Anonymous mapping.

p_new_rev = map (p_new)
{
    if ($1.rest) { pattern("1"); }
    else         { pattern("0");  }
};

//Mapper definitions.

mapper newMapper1 (p_any)
{
    if (p_any.note)   { return pattern("1"); }
    else              { return pattern("");  }
}

mapper newMapper2 (p_any alternate_beats)
{
    if (alternate_beats.rest) { return pattern("");} //pattern of length 0
    elseif (p_any.note)       { return pattern("1");}
    else                      { return pattern("0");}
}

mapper improved_newMapper2(p_any alternate_beats)
{
    if (alternate_beats.rest)   { return pattern("");  }
    elseif (p_any.note)         { return pattern("1"); }
    elseif (p_any.next(1).note) { return pattern("1"); }
    else                        { return pattern("0"); }
}

p_custom_new_notes   = map (p_custom_new) myMapper1;
p_concat_new_downbeats = map (p_concat_new) newMapper2;
```

```
//print out the created patterns to Standard Output.

print("Output from Sample DruL Code:");
print(p_concat);
print(p_custom);
print(p_custom_new);
print(p_new);
print(p_new_rev);
print(p_custom_new_notes);
print(p_concat_new_downbeats);
print("END OF OUTPUT");

//Pattern associations using clips.

clip_complete = clip
(
 hihat    = p_concat_new_downbeats
 bassdrum = p_custom_new_notes
 crash    = p_new_rev
 snare    = p_new
);

//output clip as a midi file
out.midi("out_file1.midi",clip_complete,a);//a = tempo (Beats per minute)

// Last instrument has an empty beat-pattern.
clip_partial = clip(p_concat p_custom_new p_custom);

//output clip as a midi file
out.midi("out_file2.midi",clip_partial,b);//b = tempo
```