DruL

A language to encourage laziness among drummers

Motivation

- Make Rob’s life easier when he’s writing new drum loops…
- Make it easy to write long drum parts via algorithmic composition
- Simpler than alternatives e.g. Haskore - no pitch or note durations.

Basic appearance:

- C-style identifiers
- Semicolons, Braces, Parentheses
- Commas
- Double-slash comments à la C++ (no multi-line comments)
- In short, looks a lot like a C/Java descendent, with one very important exception: map

Appearances can be deceptive

- Typing: strict, but dynamic
- Scoping: dynamic
- Side-effects: tightly controlled
  - Limited to four kinds of statement: assignment, mapper definition, instrument definition, and return
  - NOT possible in an expression
- Small set of available types
- Small set of built-in functions, mostly constructors and basic utilities
- Java-style method calls for some objects

Types

- Assignable: integer, clip, pattern

  only possible values for user-defined variables

- Literal: string, boolean

  mostly available for debugging purposes

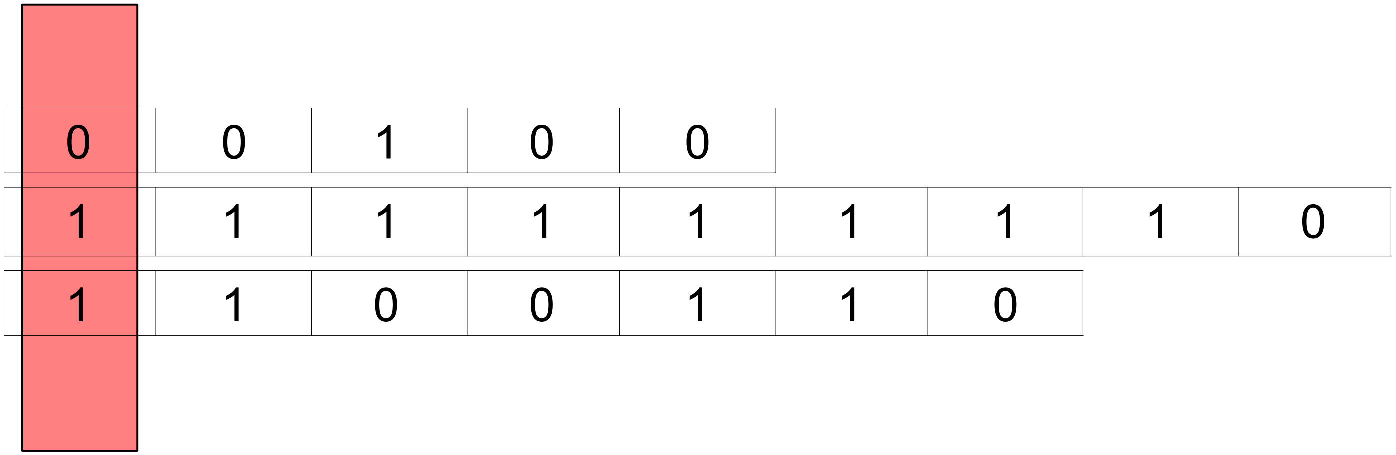
- Special: beat, mapper, instrument-name
  - beat objects exist only within mappers
  - mappers are created like functions (but no forward declaration)
  - instruments are definitions are special “function”

Wait, what were those?

- pattern: a sequence of boolean values (notes and rests)
- instruments: a global list of instrument names
- clip: a collection of patterns, mapped to instruments for output

Finally, mappers

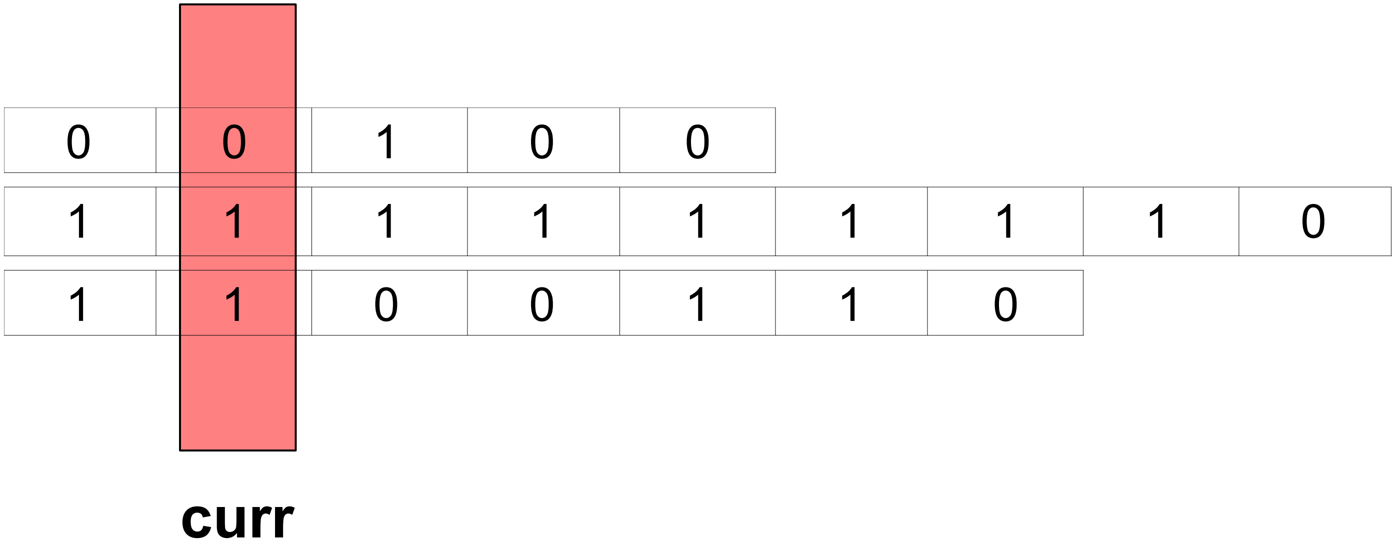
- The core distinction between DruL and micro-C: mappers
- Allow creation of new patterns from existing ones according to pre-defined transformations
- DruL has mappers instead of user-defined functions
- Essentially, an iterator, but with special language support for examining the current (musical) context



0	0	1	0	0				
1	1	1	1	1	1	1	1	0
1	1	0	0	1	1	0		

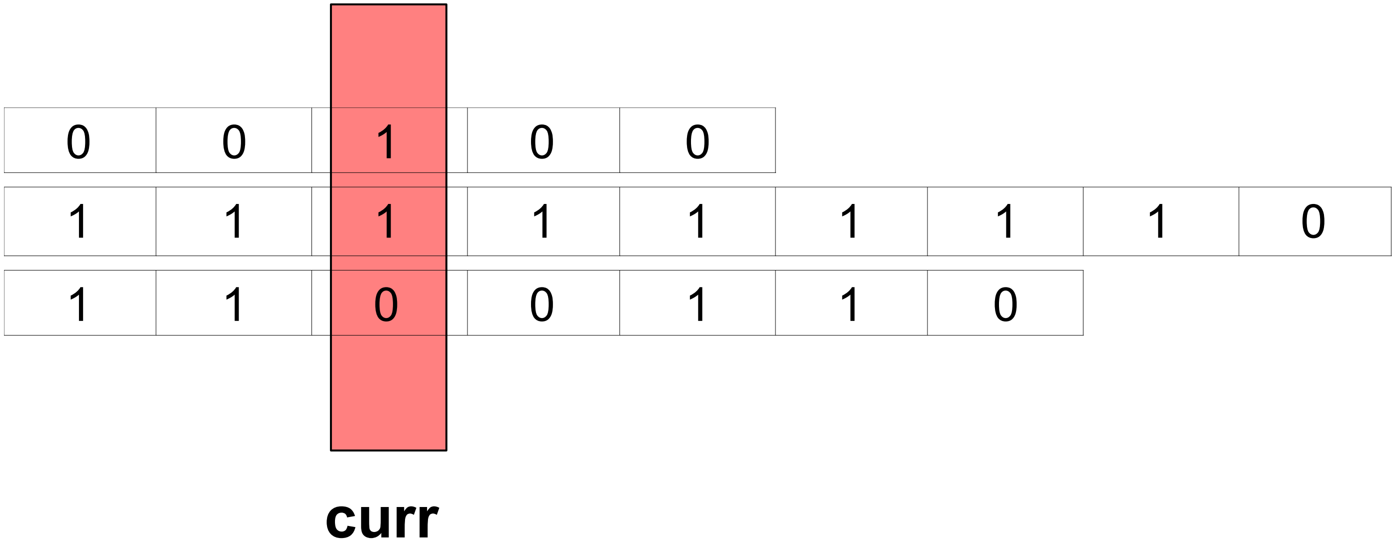
curr

$1 -> 0
$2 -> 1
$3 -> 1



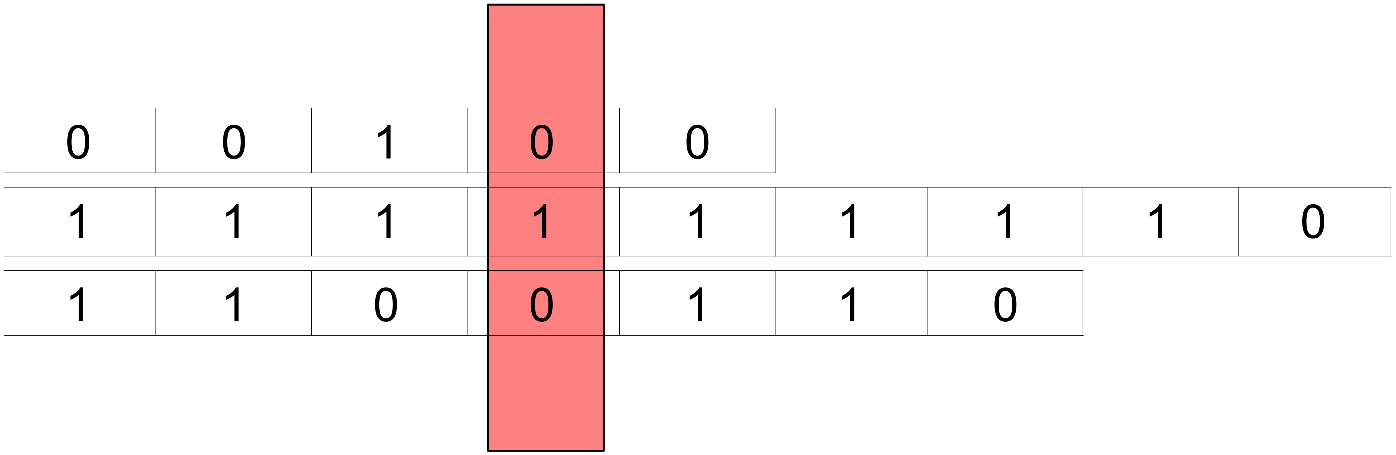
0	0	1	0	0				
1	1	1	1	1	1	1	1	0
1	1	0	0	1	1	0		

$1 -> 0
$2 -> 1
$3 -> 1



0	0	1	0	0				
1	1	1	1	1	1	1	1	0
1	1	0	0	1	1	0		

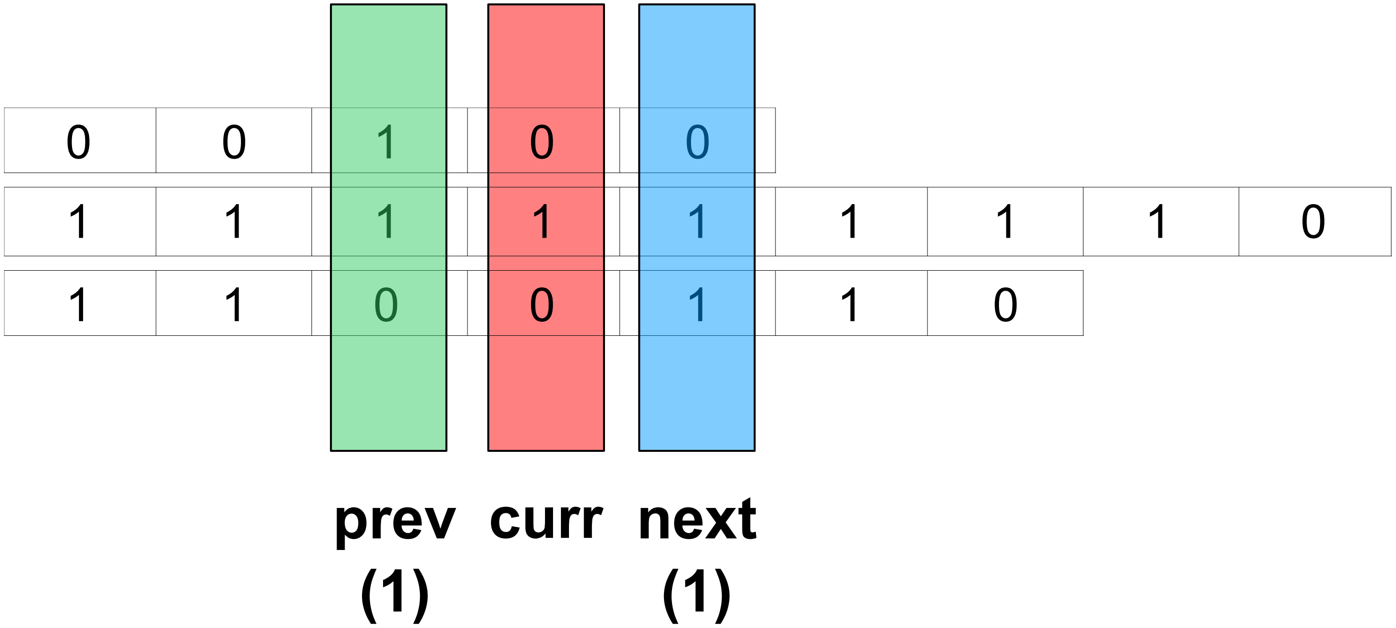
$1 -> 1
$2 -> 1
$3 -> 0



0	0	1	0	0				
1	1	1	1	1	1	1	1	0
1	1	0	0	1	1	0		

curr

$1 -> 0
$2 -> 1
$3 -> 0



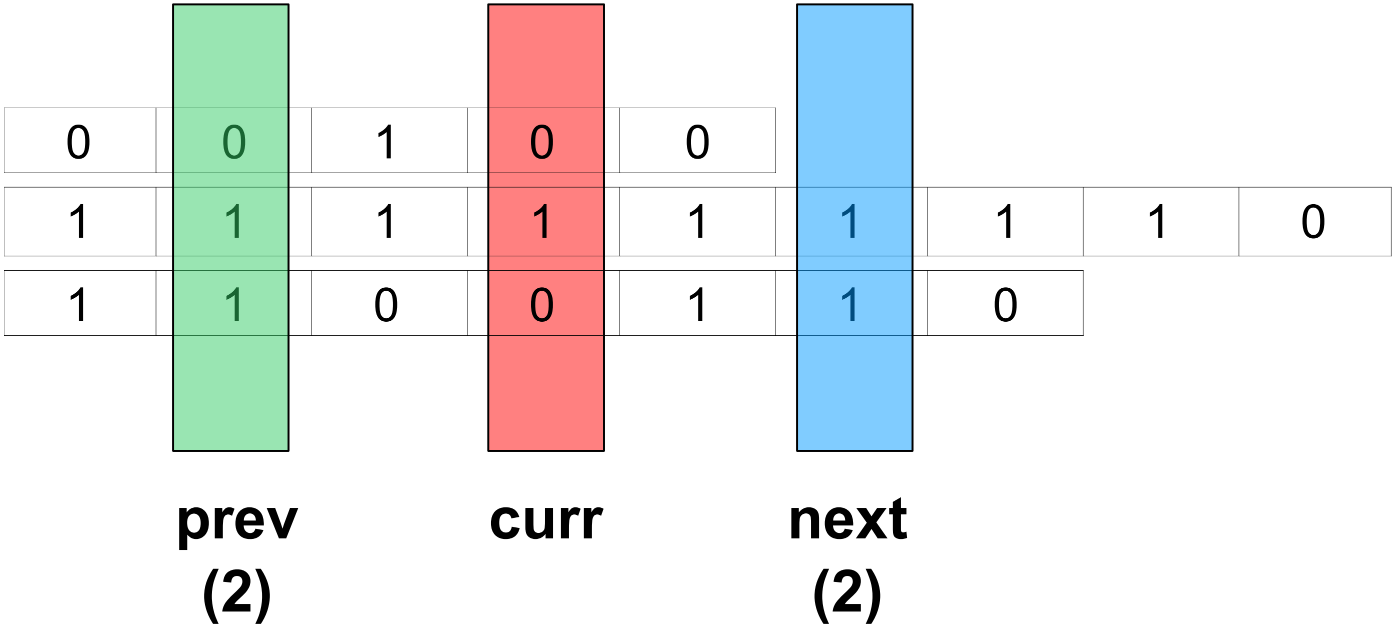
0	0	1	0	0				
1	1	1	1	1	1	1	1	0
1	1	0	0	1	1	0		

prev (1)

curr

next (1)

$1 -> 0
$2 -> 1
$3 -> 0

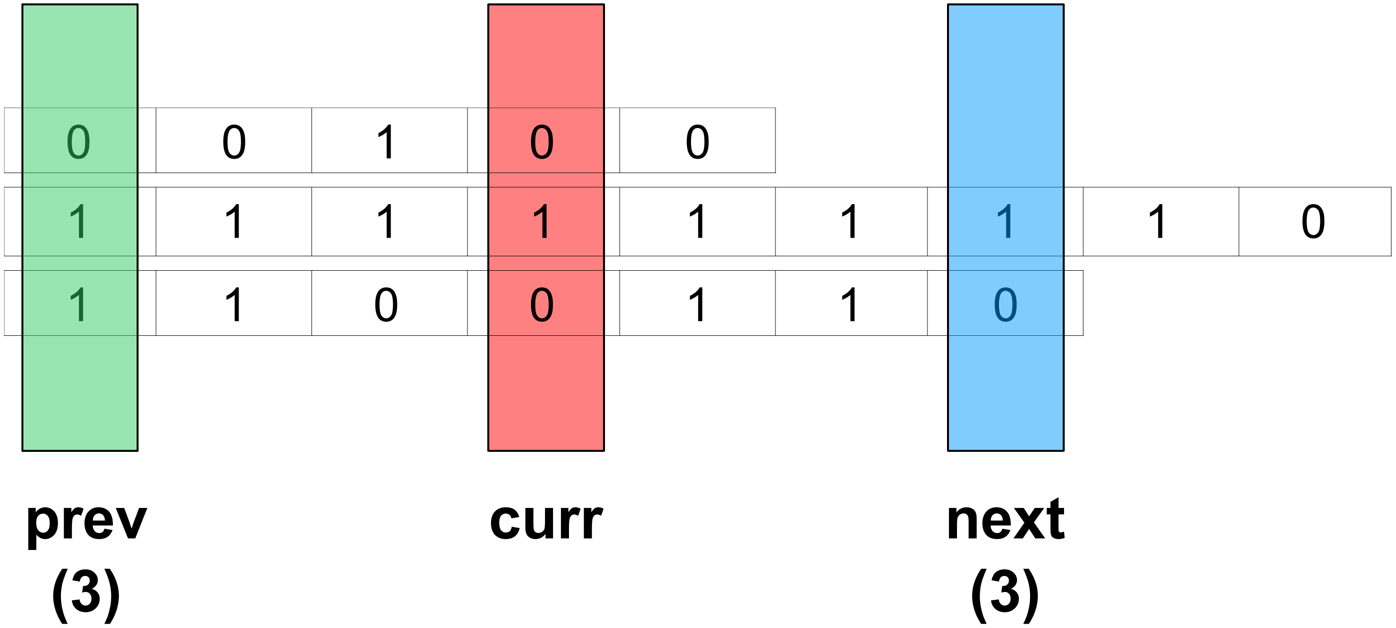


0 0 1 0 0

1 1 1 1 1 1 1 1 0

1 1 0 0 1 1 0

$1 -> 0
$2 -> 1
$3 -> 0



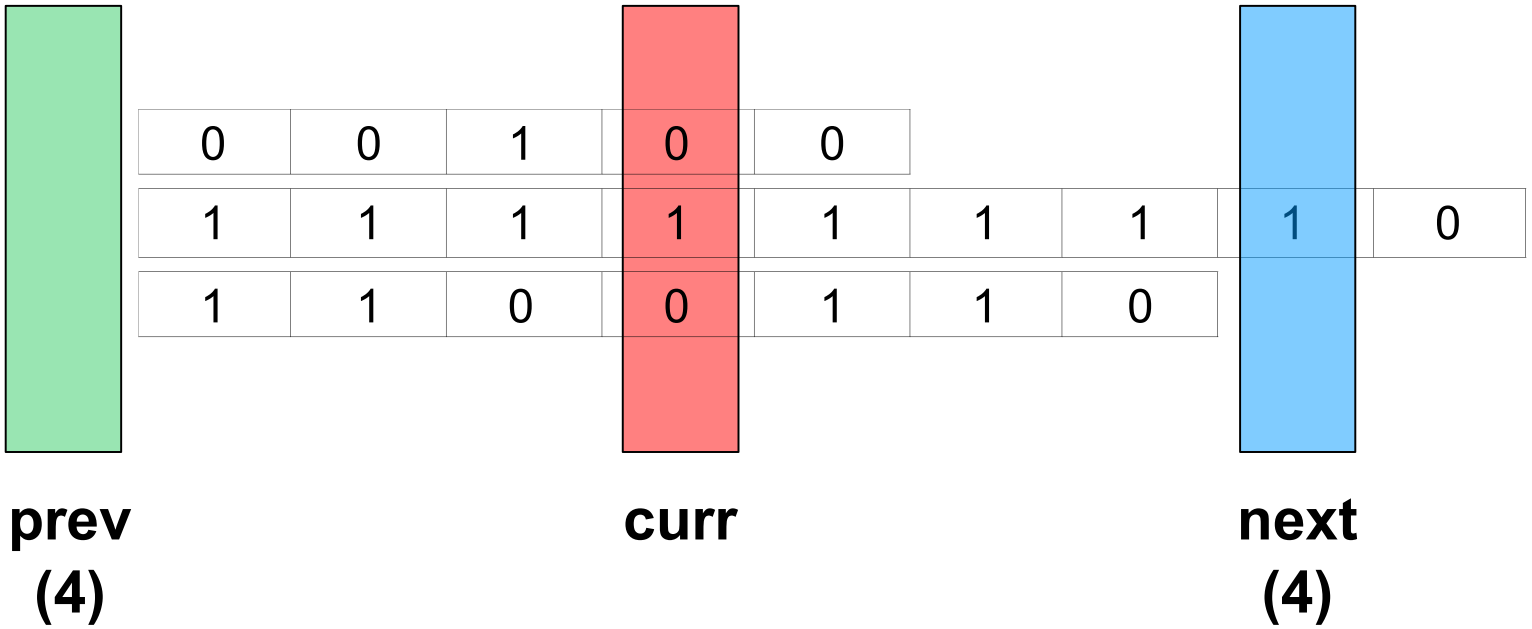
0	0	1	0	0				
1	1	1	1	1	1	1	1	0
1	1	0	0	1	1	0		

prev
(3)

curr

next
(3)

$1 -> 0
$2 -> 1
$3 -> 0



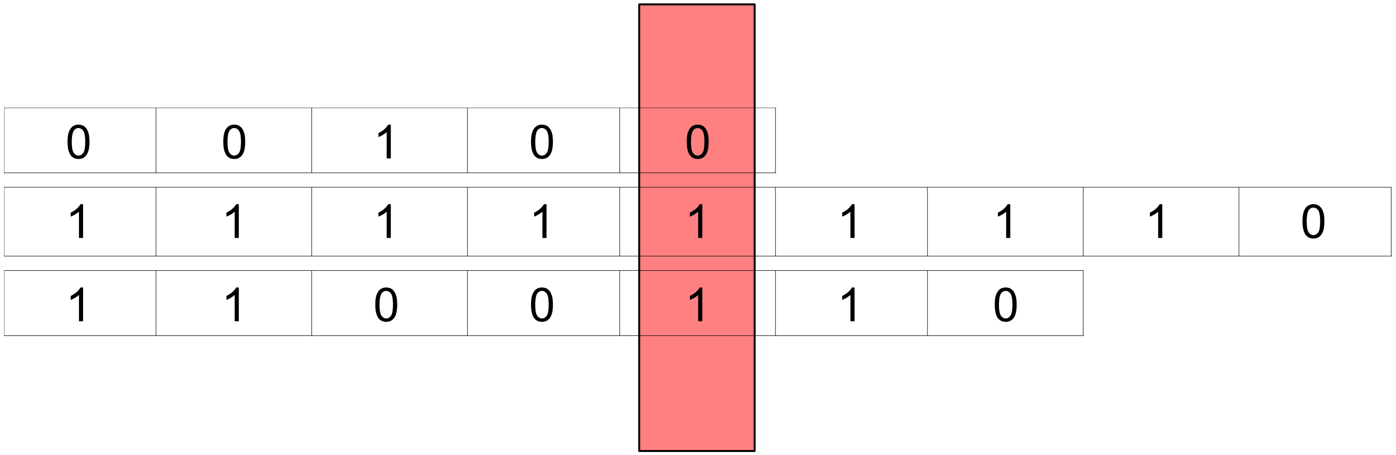
0	0	1	0	0				
1	1	1	1	1	1	1	1	0
1	1	0	0	1	1	0		

prev
(4)

curr

next
(4)

$1 -> 0
$2 -> 1
$3 -> 0



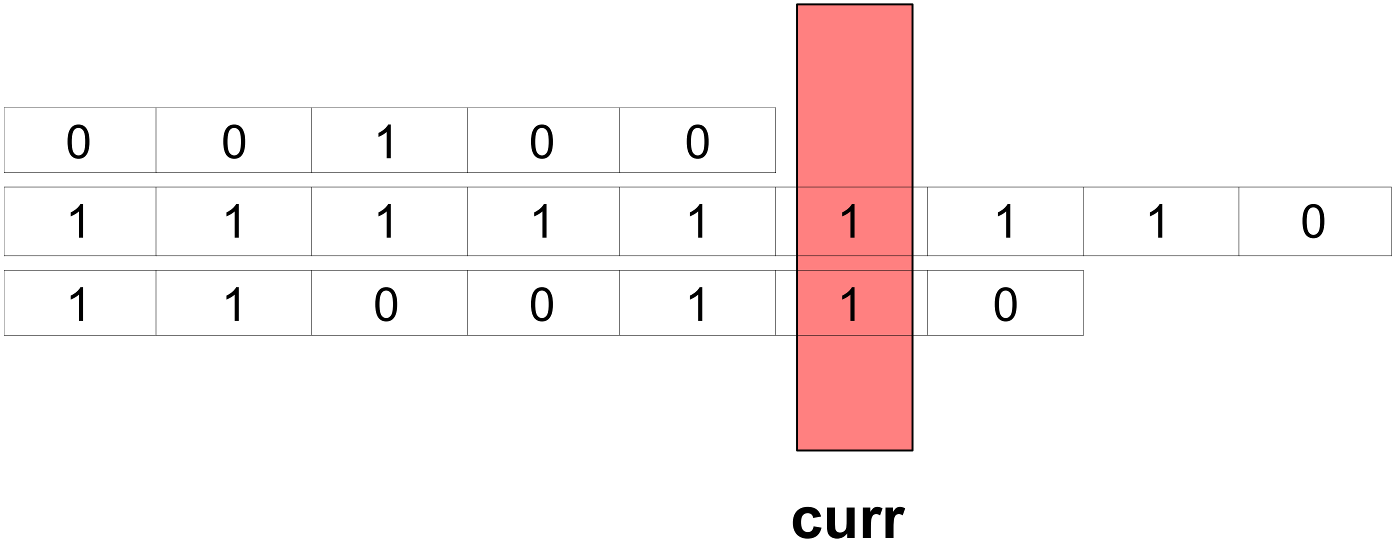
0	0	1	0	0				
1	1	1	1	1	1	1	1	0
1	1	0	0	1	1	0		

curr

$1 -> 0

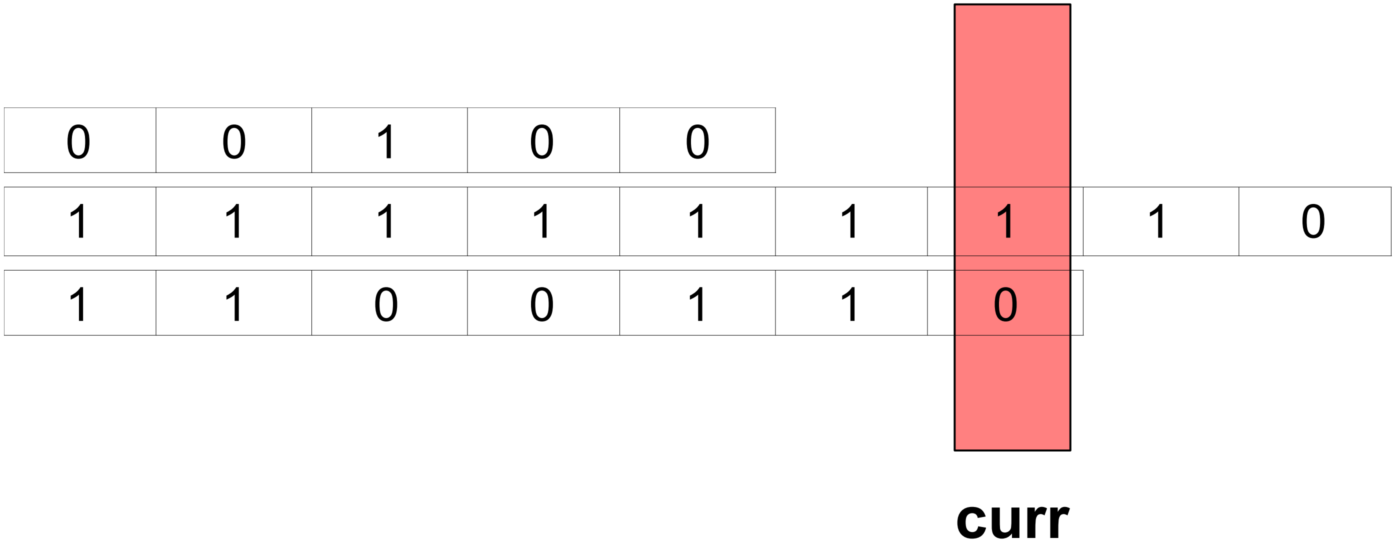
$2 -> 1

$3 -> 1



0	0	1	0	0				
1	1	1	1	1	1	1	1	0
1	1	0	0	1	1	0		

$1 ->
$2 -> 1
$3 -> 1

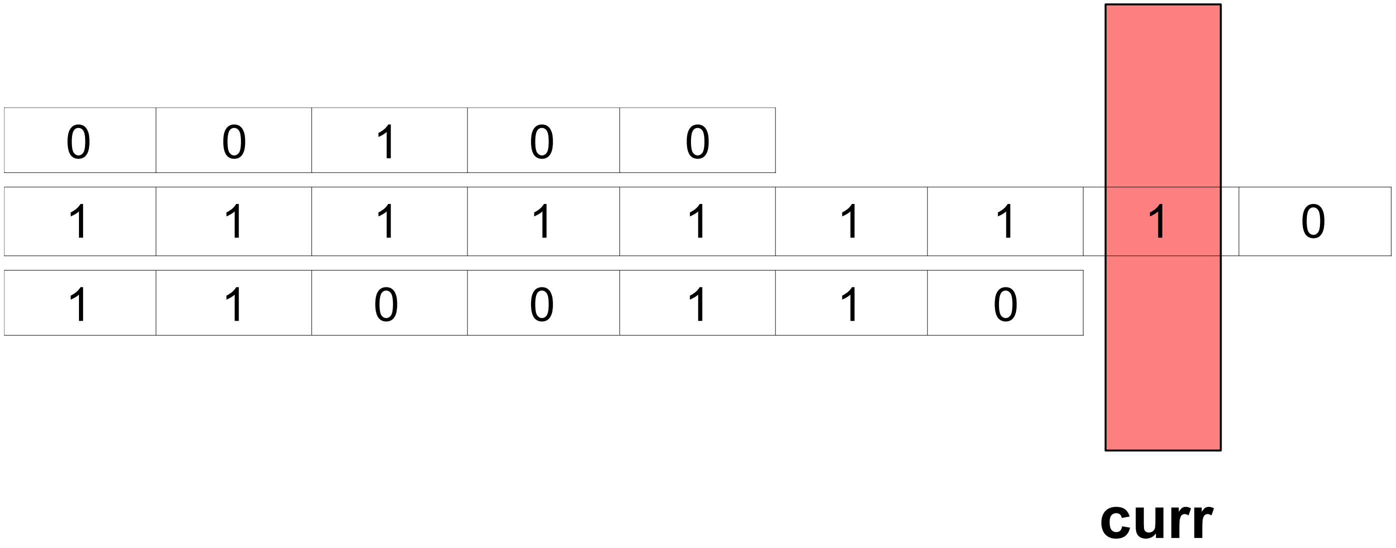


0 0 1 0 0

1 1 1 1 1 1 1 1 0

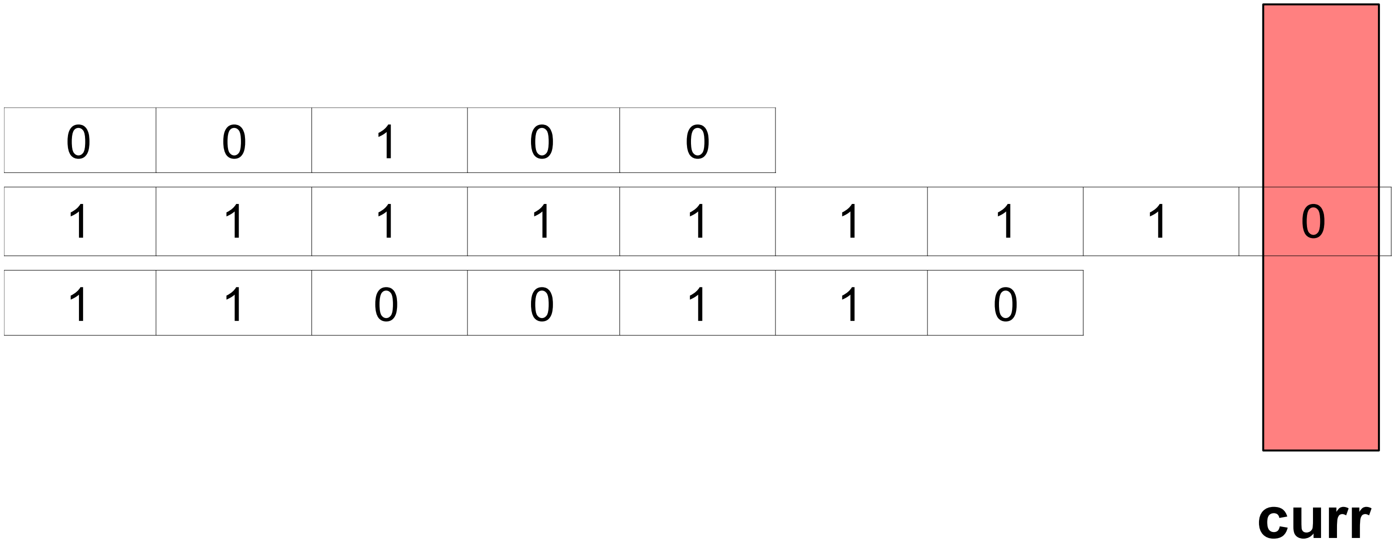
1 1 0 0 1 1 0

$1 ->
$2 -> 1
$3 -> 0



0	0	1	0	0				
1	1	1	1	1	1	1	1	0
1	1	0	0	1	1	0		

$1 ->
$2 -> 1
$3 ->



0	0	1	0	0				
1	1	1	1	1	1	1	1	0
1	1	0	0	1	1	0		

$1 ->
$2 -> 0
$3 ->

Demonstration Code

```
a = 3;
b = 5;

if (a > 0 && b > a)
{
  print(“hello, world!”);
}
elseif (a >= 0)
{
  print(“Well, that was unexpected”);
}
else
{
  print(false);
}
```

Interesting Demonstration Code

```
p = pattern("10101111");
q = pattern("11110000");
r = concat(p, q);

if (r.length() < q.length())
{
  print(q.repeat(3));
}
else
{
  print (r.length());
}
```

And now, that mapper stuff…

```
p = pattern("10101111");
q = pattern("11110000");

r = map(p, q)
{
   if ($1.note() && $2.note())
   {
       return pattern("11");
   }
   else { return $1; }
};

// prints "1101101111"
```

Named Mappers

```
mapper filterMap (pat, filter)
{
  if (filter.rest()) { return pattern(""); }
  else               { return pat;         }
}

filtered = map (p, q) filterMap;

// results in the pattern "1010"
```

The Superstructure

```
instruments("snare", "hihat", "kick");

c = clip(p, q, r);

c.outputText("sample.txt");

// midi needs a tempo (beats per minute)
c.outputMidi("sample.midi",120);

// Lilypond needs a title to typeset
c.outputLilypond("sample.ly", "Typeset Sample");
```

The proof of the pudding

```
p1 = pattern("1").repeat(352);
p2 = pattern("1").repeat(40);
…
mapper gcd(a, b) {
    if (         !a.prev(1).note() && !a.prev(1).rest()
          &&     !b.prev(1).note() && !b.prev(1).rest() ) {
        tmp = map (p1, p2) subtract;
        if (tmp.length() == 0) { return p1; }
        elseif ((map(tmp) squishrests).length() > 0) { p1 = tmp; }
        else { p2 = tmp;}
        return map(p1, p2) gcd;
    }
    return pattern("");
}
```

Interpreter

- DruL is an interpreted language
- Not compiled since there isn't much concern about performance
- Complex calculations are possible in DruL, but not an intended use of language

Dynamic Language

- Variables are dynamically typed
- Hence, few possible static checks
- We didn’t do them (due to time constraints)
- DruL types map easily to Ocaml types

DruL Types

```
type drul_t =
        Void
        |       Int        of int
        |       Str        of string
        |       Bool       of bool
        |       Pattern    of pattern
        |       Clip       of pattern array
        |       Mapper     of (string * string list * statement list)
        |       PatternAlias of pattern_alias
        |       Beat of pattern_alias * int
        |       Instruments of string list
        |       InstrumentAssignment of string * pattern
```

Syntax Tree

- Distinct boolean, integer and comparison operator-types in AST, used in expressions
- Expressions tagged with line number, to report errors in drul code
- A drul program is just a list of statements

Keywords, Functions and Methods

- Not all keywords are tokens (e.g. functions)
- Built in functions are keywords
- Built in methods specific to DruL types are not keywords
- Thus, method names can be used as identifiers (variables, named mappers)

Statements

- Types: Expression, Assignment, Selection, Mapper definitions, Return
- Blocks are not statements

Lessons Learned

- **Standards** are there for a reason
    - Comma-separated lists
    - Dynamic scoping is easy
    - if/else implemented as a tree, not a list
- **Tests** are good
    - Build test suite early, many tests
    - Found us a bug on precedence for method calls

Lessons Learned

- **Catching errors** early is hard
    - Move errors from scanner and parser down to the interpreter
    - Less efficient for the user, may run half of the code before an error
- **Ocaml's inference** is great
    - When it guesses what you want it to guess
    - We one thaught we could do type inference ourselves...!
- **Pair** programming works well
    - One by itslef, hard to take decision
    - More than 2 around a computer is useless

Lines of code

main program		test suite
40	drul_ast.mli	
219	drul_helpers.ml	
42	drul_interpreter.ml	26 tests (parser)
471	drul_main.ml	285
87	drul_output.ml	
119	drul_parser.mly	79 test (drul)
66	drul_printer.ml	422
106	drul_scanner.mll	
59	drul_types.ml	2 'test' functions
61	Makefile	399
8	test.ml	
5	treedump.ml	
1283 total		1106 total