A Virtual Machine for Live Coding Language Design

Graham Wakefield
School of the Arts, Media, Performance and Design, York University, Toronto, Canada
grrrwaaa@yorku.ca

Charles Roberts
School of Interactive Games and Media
Rochester Institute of Technology, USA
cdrigm@rit.edu

ABSTRACT

The growth of the live-coding community has been coupled with a rich development of experimentation in new domain-specific languages, sometimes idiosyncratic to the interests of their performers. Nevertheless, programming language design may seem foreboding to many, steeped in computer science that is distant from the expertise of music performance. To broaden access to designing unique languages-as-instruments we developed an online programming environment that offers liveness in the process of language design as well as performance. The editor utilizes the Parsing Expression Grammar formalism for language design, and a virtual machine featuring collaborative multitasking for execution, in order to support a diversity of language concepts and affordances. The editor is coupled with online tutorial documentation aimed at the computer music community, with live examples embedded. This paper documents the design and use of the editor and its underlying virtual machine.

Author Keywords

Computer Music, Live Coding, Programming Language Design, Parsing Expression Grammars, Web Technologies

ACM Classification

H.5.5 [Information Interfaces and Presentation] Sound and Music Computing, H.5.2 [Information Interfaces and Presentation] User Interfaces, D.3 [Programming Languages], D.2.11 [Software Engineering] Software Architectures.

1. INTRODUCTION

The past decade has seen a broad variety of programming languages used as instruments for live musical performance. The notion of the programming language itself being a musical instrument is particularly well established within live coding research [19, 2]; the community-authored TOPLAP manifesto for example asserts that "programs are instruments that can change themselves" [17]. Many such instruments use languages with musical features or *domain-specific* languages (DSLs) that are oriented to computer music. A DSL is valued according to how it *reframes* the terms of work into a language of abstractions and specializations through which the characteristic affordances and constraints of its application domain become more readily available and discoverable.[14] Thus some performers and live coders have explored designing new "mini-languages" to embed within existing systems, frequently with live performance as a driving motivation. For example, McLean's "TidalCycles" is a domain-specific specialization of Haskell that embeds a further mini-language for polyrhythmic cycles. Magnusson's "ixi lang" sits atop the SuperCollider domain-specific specialization of SmallTalk, and further embeds a mini-language for agent scores. As an instrument, the language specialization serves as a "scaffold for externalising musical thinking"[9], through which "we are more able to engage the right kind of cognitive load"[11] or "engage directly with music through a high-level representation of musical patterns"[7]. Some of these language specializations are so idiosyncratic and model-constrained to be considered initiating a miniature genre[3] or compositional form[9].

In this paper we document our work supporting the exploration of notational specificity in languages-as-instruments, while retaining flexibility, by bringing "liveness" to the process of language design itself. We created a responsive grammar editor along with a simple yet flexible target virtual machine, embedded in a browser along with tutorial materials, for ease of access.[1] Central to our goal is helping musicians create highly divergent or idiosyncratic instrument-languages in a variety of paradigms. However, although live coding languages often incorporate graphical syntax, for now we have limited our scope to textual input.

NIME'17, May 15-19, 2017, Aalborg University Copenhagen, Denmark.

2. DESIGNING LANGUAGES LIVE

The study of languages is often divided into *syntax*, *semantics*, and *pragmatics*. Syntax specifies what are permissible utterances of a language in terms of structure and composition; i.e. what can be *parsed*, regardless of meaning. Semantics provides a function or mapping of the syntax to a representation of its meaning, i.e. rules for transforming terms into other terms; in computing this may be embedded in the logical design of a compiler or interpreter. Pragmatics refers to how the semantic meaning of a fragment, or of a language, can be used; what it is useful for. As a musical instrument, the syntax provides its outer appearance and possible arrangements, the semantics identify what each surface feature does, and the pragmatics relate to what idioms, compositional forms, or mini-genres it lends itself to. A fourth concept of *metalanguage* gives the terms of how each of these is defined.

2.1 Parsing

We chose the Parsing Expression Grammar (PEG) [6, 10] metalanguage for parsing, which like many grammar formalisms describes a language in terms of a top-down set of rules. These rules gain expressive power by being composed through a variety of operators to succinctly capture a greater variety of syntactic patterns. PEGs are easy to

[1]This system can be accessed from `https://worldmaking.github.io/workshop_nime_2017/index.html`

grasp, with barely more than a handful of operators, little syntax, and a handful of common 'recipes' to cover most situations.

A grammar consists of a set of rules, the first of which is the start rule. Each rule has a *name*, a syntactic *pattern* of strings, rule names, and operators that define the kinds of input the rule can recognize and return, plus an optional semantic *action* which can transform what is returned. In our editor the action is simply the body of a JavaScript function. For example, the following rule, named "start", recognizes the text "hello", but returns the text "hej":

```
start = "hello" { return "hej"; }
```

This trivial example highlights how parsing is akin to *translation* of the syntax of a *source language* into the semantics of a *target language*. Rules become more interesting through the use of the small set of combinators, including concepts of sequence, ordered-choice, zero-or-more, one-or-more, etc., and a few other short-hand conveniences.

We determined that actions of parser rules should generate flat data structures representing a target language rather than directly triggering sonic events. One reason is that a readable output language can help clarify parser behavior. Another reason is that the specific order in which the parser traverses input code may not match desired semantics; structural changes may need to wait for a higher rule to be activated to arbitrarily re-organize the data of sub-rules, before submitting to playback. This for example allows the durations of a loop's elements to be calculated according to the number of elements it contains; more generally, it allows input languages to vary quite flexibly between imperative and declarative styles.

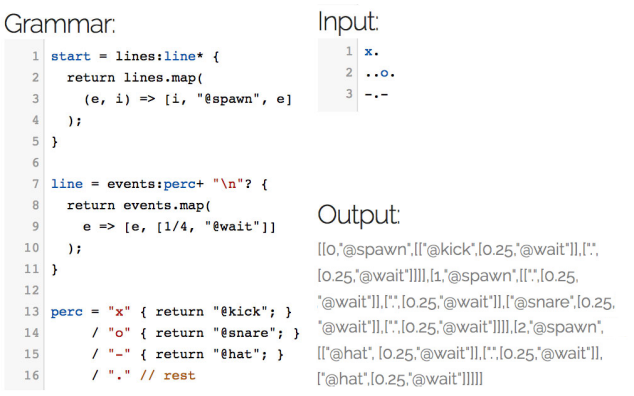

Figure 1: The browser-based interface. In the Grammar panel users can define a language's syntax and semantic actions. The Input panel lets users experiment with the resulting language, with the Output section below displaying live the semantic results as instructions for the playback engine.

2.2 The editor

The editor (Figure 1) is designed to be responsive in design and interaction. Both the grammar and the source input editor panes grow to fit the content, and can be edited live with continuous evaluation. If desired, any input that parses correctly will be executed automatically by the playback engine. Alternatively, smaller fragments of the input window can be selected and executed at any time using the `Ctrl-Enter` or `Cmd-Enter` keyboard shortcuts. For inputs that fail to parse, an error is shown but no sonic changes ensue. Such liveness of input text evaluation is normal for a live coding language instrument. However the system also applies this liveness to the grammar editor. Each modification to the language's grammar, and its semantic actions, also triggers a re-evaluation of the input text against the new grammar, and sonic events if the parse is successful.

2.3 The target language

The grammar editor allows a very wide range of idiosyncratic languages to be designed, which through translation to the target language must create sonic structure through the set of features available in the playback engine. In the choice of these available features we aimed for a concise expression of a common set of concepts frequently found in music programming and live coding languages, such as strong timing for events, collaborative multitasking (or agents and scores as found throughout Magnusson's work), local and global communication, the loops and layers typical to dance music [5], communication with other software, etc. We formalized these concepts through a target language guided by several sometimes conflicting priorities:

- Human-readable: to rapidly understand how the parser is working (correctly or incorrectly). This implies minimizing hidden state, and avoiding verbosity.
- Easy to learn, predictable to generate. Where reasonable, keep abstractions in user-written grammars.
- Simple to generate: to keep grammar actions legible. Terseness of the target language is less important than the ease and flexibility of generating it.
- Efficient to interpret: keeping the playback engine efficient while supporting different structural patterns.

The design we settled on bears resemblance to a stack-based language, in which a program fragment is a sequence of tokens interpreted item by item, possibly modifying the sequence as a result. This is highly conducive to interpretation, and indeed stack-based languages are frequently used as the internal representations of interpreted languages. The language is not a string of text tokens, but a sequence of terms (as a JavaScript array) in which terms can be regular numbers, strings, or nested sequences. We distinguish executable instructions (or "operators") from data strings by prefixing with the `@` character, which facilitates readability and simplifies the implementation of the interpreter.

Some instructions are relatively simple, like the binary operator `@add`, whereas others are more complex. Some instructions were chosen based on observations of similar operations widely used in performance practice over various languages. In particular, several instructions exist to introduce variation, including `@iter`, which rotates the argument used on each pass through a loop, `@chance`, which picks between two arguments according to a probability, `@pick`, which chooses an argument at random, etc.

Stack-based languages are naturally expressed by reverse Polish notation, in which operands are interpreted and submitted to the stack first, then consumed by subsequent operators. For example, the following nested sequence will wait for 12 beats before completing:

```
[[4, 8, "@add"], "@wait"]
```

The above is a valid JavaScript expression to declare a nested list (array), and this is how it could be expressed in the parser. More likely however it would be constructed through a combination of parser actions. And although nesting in this example is not actually necessary, it can aid the reader in understanding the instructions, and permits some useful flexibility in how the parser actions are applied.

Sometimes it is necessary to defer evaluation of an expression until it is needed; for example, a pattern may need to play only conditionally, and should not be evaluated otherwise; or a pattern may need to repeat a number of times. In LISP-family languages the quote operator can be used to pass an expression unevaluated (which is to say, to bypass the normal mode of evaluation of the sequence of terms). Instead, to simplify our engine, we present unevaluated arguments after the operator, rather than before. For example, when enountering a `@loop` instruction, the interpreter knows to expect the next item to be a pattern to loop (and will throw an error if this is not found).

```
["@loop", ["@snare", [1, "@wait"] ] ]
```

If we didn't do this, then we would have to either explicitly quote patterns, or alternatively explicitly mark lists to be evaluated. We avoid the need for widespread use of `quote` or `eval` operators by argument placement. This also simplifies the implementation of the language's interpreter.

2.4 The playback engine

The playback engine implements an interpreter of the target language with flexible and sample accurate scheduling. This engine can run entirely in the browser, using a timing clock and a small selection of default instruments from the Gibberish library [12] using instructions such as `@pluck-note` or `@snare`. Alternatively it can synchronize and communicate with external applications via MIDI or WebSockets using `@midi-` and `@ws-` instructions. The temporal semantics implement collaborative multitasking to support parallel musical threads, scheduled according to subdivisions of an overall musical pulse. A simple a list of events will be sequenced at the same musical time, playing simultaneously:

```
[220, 1, "@pluck-note",
 330, 1, "@pluck-note",
 550, 1, "@pluck-note"]
```

To separate events in time, a sequence must incorporate the `@wait` instruction, which consumes an argument for the duration (in terms of the pulse) to insert. The `@wait` instruction effectively yields the current instruction queue to its parent scheduler, to be resumed at a precisely computed time in the future. So while the example above plays a chord, the example below plays an arpeggio:

```
[220, 1, "@pluck-note", 1/4, "@wait",
 330, 1, "@pluck-note", 1/4, "@wait",
 550, 1, "@pluck-note", 1/4, "@wait"]
```

New parallel instruction queues can be created within a sequence, either as one-time only sequences using `@fork`, as infinitely looping sequences using `@loop`, or as named infinite looping sequences using `@spawn`. The following example plays a simple triplet over duplet hemiola:

```
["a", "@spawn", [ "@kick", 3/4, "@wait" ],
 "b", "@spawn", [ "@hat" , 2/4, "@wait" ]]
```

While `@wait`, `@fork`, and `@loop` turn instruction queues into resumable continuations with tightly timed scheduling, naming a loop via `@spawn` allows its pattern to be re-defined on the fly—an essential idiom of live coding [15]—by spawning the same name with a new pattern.

Parallel sequences can also carry local state, assigned via `@let` and retrieved via `@get`; or global state can be addressed via `@set`. Let-bound state is visible in any sub-queues forked or looped, but not visible to any parent queues. Thus a queue can share state with the patterns it creates without clashing with names in sibling patterns.

2.5 Examples

It can be interesting to utilize the spatial organization of the text to coordinate the temporal organization of the result. In the grammar below, while both `loop1` and `loop2` have the same syntax, the semantic action is slightly different. With an input of `"x-o-.-"` `loop1` produces a pattern of 16th notes repeating every one and half beats, whereas `loop2` produces a pattern spanning one beat that contains six events.

```
perc = "x" { return "@kick"; }
     / "o" { return "@snare"; }
     / "-" { return "@hat"; }
     / "." // rest

loop1 = events:perc+ {
  return ["@loop", events.map(
    e => [e, [1/4, "@wait"]]
  )];
}

loop2 = events:perc+ {
  var step = 1/events.length;
  return ["@loop", events.map(
    e => [e, [step, "@wait"]]
  )];
}
```

A strategy we have found useful for some mini-language experiments is to automatically assign spawn-names to each new line of the input text. In this way, the input fragment supports a flexible and stutter-free notion of each line corresponding to a looping layer in the musical performance:

```
start = lines:line* {
  return lines.map(
    (e, i) => [i, "@spawn", e]
  );
}

line = events:perc+ "\n"? {
  return events.map(
    e => [e, [1/4, "@wait"]]
  );
}
```

With the above grammar, a three-part polyrhthmic pattern amenable to direct live manipulation can be as simple as:

```
x...
.o...o.x...x
.-.
```

3. IN USE

The system was shared with two dozen participants in a workshop held at the 2016 International Conference on Live Coding.[2] A tutorial introduced parsing by building a language by incrementally adding each of fifteen grammar operators. We worked through a handful of common needs, discussed common errors, and explored examples creating musical structure with the target language and engine.

After this presentation workshop attendees had around forty-five minutes to develop a prototype language. Three attendees volunteered to give brief performances, none of whom had prior experience with parsing or programming language design. One participant presented a prototype of system for creating serial compositions that he continued to develop this after the workshop; it is now a MIDI sequencer that runs in the browser, available online [18].

A simpler example by another participant converted ASCII values of characters entered into pitch information read back in sequence to form musical phrases, with the duration of each note in the phrase derived from the phrase's length,

[2] `http://iclc.livecodenetwork.org/2016`

generating parallel sequences of polyrhythmic percussion. The participant was able to pull of a rousing two-minute performance with it, a surprising feat considering that the language was less than one hour old![3] Other participants did not have their languages in a state they felt comfortable presenting as their goals were more ambitious than the workshop duration enabled; nevertheless the feedback we received was overwhelmingly positive.

4. DISCUSSION

Although the editor was designed primarily to support the process of language design, there is a tantalizing invitation to build and use a language-instrument on the fly, during a performance. Live coding musicians perform *with* programming languages, but hardly ever *by* programming languages. This could be driven by a goal to pursue a more profound live coding act through a "more significant intervention in the works" [4], or it may instead play into an oft-mentioned motivation for 'showing your screens' during performance to further democratize the realm of computing (and process in the wider society)–for example, that anyone else could start from the same place [1]. It may also play into legibility, as an audience member might better understand what is being written if the grammar is grown from zero before their eyes.

Toward legibility, we are interested letting the parser automatically colourize or otherwise highlight input language according to grammar rules. We would also like to extend the editing interface with annotation capacities driven by the playback engine; for example, momentarily highlighting fragments of the input that correspond to sonic events currently triggered within sequences. Such annotations may be generally useful sources of feedback for language designers as well as performers and audience members.

A current weakness of the target language is the difficulty of scheduling algorithmic transformations of structured data. Pattern transformation concepts such as inversion and transposition are powerful means to create variety when applied over time[16], but are not easy to define in our target language without adding more complex operators. Related, control flow is addressed in our target language through finite and infinite backward jumps (loops), simple conditional branches, and the parallelism of continuations, but there is no support yet for label/goto or subroutines. Goto may be considered harmful, but it is easy to see how subroutine jumps could empower richer languages–and might not be difficult to add. A more challenging but interesting problem would be the support for instructions in the playback engine to reach back and modify the input text, performing source code manipulations as found in ixi lang [8] and Gibber[13]. Though it is not clear whether this would be of any use, a language might even be able to transform its own grammar.

5. ACKNOWLEDGMENTS

Supported by the Canada Research Chairs program and York University, Canada. Also supported by Rochester Institute of Technology, with funding from Sponsored Research Services and the Golisano College of Computing and Information Sciences. Additionally, we would like to thank Daniel Gómez Blasco for his extensive work refactoring the virtual machine as a standalone library.

6. REFERENCES

[1] R. Biddle. Clickety-Click: Live Coding and Software Engineering. In *Collaboration and learning through live coding (Dagstuhl Seminar 13382)*, volume 3, pages 154–159. 2014.

[2] A. Blackwell and N. Collins. The programming language as a musical instrument. *Proceedings of PPIG05 (Psychology of Programming Interest Group)*, 3:284–289, 2005.

[3] A. F. Blackwell. Patterns of user experience in performance programming. In *Proceedings of the International Conference on Live Coding*, pages 53–63. University of Leeds, UK, 2015.

[4] N. Collins. Live coding of consequence. *Leonardo*, 44(3):207–211, 2011.

[5] N. Collins and A. McLean. Algorave: A survey of the history, aesthetics and technology of live performance of algorithmic electronic dance music. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 355–358, 2014.

[6] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM.

[7] T. Magnusson. Designing constraints: Composing and performing with digital musical systems. *Computer Music Journal*, 34(4):62–73, 2010.

[8] T. Magnusson. ixi lang: a supercollider parasite for live coding. In *Proceedings of the International Computer Music Conference*, pages 503–506. University of Huddersfield, 2011.

[9] T. Magnusson. ixi lang. In *Collaboration and learning through live coding (Dagstuhl Seminar 13382)*, volume 3, page 150. 2014.

[10] D. Majda. Parser Generator for JavaScript. https://pegjs.org, 2010.

[11] A. McLean. Stress and cognitive load. In *Collaboration and learning through live coding (Dagstuhl Seminar 13382)*, volume 3, pages 145–146. 2014.

[12] C. Roberts, G. Wakefield, and M. Wright. The web browser as synthesizer and interface. In *Proceedings of the Conference on New Interfaces for Musical Expression*, 2013.

[13] C. Roberts, M. Wright, and J. Kuchera-Morin. Beyond editing: Extended interaction with textual code fragments. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 126–131, 2015.

[14] J. Rohrhuber, A. de Campo, and R. Wieser. Algorithms today - notes on language design for just in time programming. In *Proceedings of the International Computer Music Conference*, pages 455–458, 2005.

[15] A. Sorensen. The Many Faces of a Temporal Recursion. http://extempore.moso.com.au/temporal_recursion.html, 2013.

[16] L. Spiegel. Manipulations of musical patterns. In *Proceedings of the Symposium on Small Computers and the Arts*, pages 19–22, 1981.

[17] TOPLAP. Manifesto. https://toplap.org/wiki/ManifestoDraft, 2010.

[18] T. Wallace. Serialist. http://serialist.irritantcreative.ca, 2016.

[19] G. Wang and P. R. Cook. On-the-fly programming: using code as an expressive musical instrument. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 138–143, 2004.

[3] `https://www.youtube.com/watch?v=3T8MrcPQ9HY`.