Live Coding in Sporth: A Stack Based Language for Audio Synthesis

Paul Batchelor
Center for Computer Research in Music and Acoustics
pbatch@ccrma.stanford.edu

ABSTRACT

This paper describes live coding musical sound in a domain-specific language called Sporth. Sporth is a stack based language ideally suited for articulating complicated modular synthesis patches using a terse syntax. Motivations for building a live coding interface will be explained, along with an overview of the Sporth language through example code. The final section will provide a brief description of the live coding listener in Sporth, as well as the recommended interface setup for re-evaluating Sporth code on the fly.

1 Introduction

Sporth (Batchelor 2017b) is a domain-specific programming language for audio synthesis and creative sound design. It belongs to a family of so-called *stack-based programming languages*. Notable examples include Forth and Postscript. These languages revolve around a global stack data structure. As a direct result of this, stack-based languages have a distinct "arguments before function" syntax sometimes referred to as reverse-polish notation. An arithmetic expression in a typical C-like programming language could look like this:

```
8 * (2 + 3)
```

The equivalent RPN expression seen in a stack-based language would look like:

```
8 2 3 + *
```

While these kinds of languages may not be widely used in modern software engineering practices, they do provide some unique features useful for live coders. Stack-based languages are generally terse, so the keystroke count can kept down. In performance situations where speed is important, keystroke reduction is an important metric. Stack-based languages have less complexity than many common programming languages, and this can contribute to reduced cognitive overhead and demand on mental bandwidth. For instance, an inherent property of stack-based design is that there is no operator precedence like that of a language like C. Stack-based languages like Forth offer a unique blend of simplicity and bottom-up methodologies to problem solving (Brodie 2004).

Many musical languages for computer-generated sound exist, and a full discussion of them is beyond the scope of this article. Chuck (Wang, Cook, and Salazar 2016) and Supercollider (McCartney 2002) are well-known examples of languages designed around the notion of live coding. In Csound, the release of version 6.0 incorporated new abilities for the re-evaluation of orchestra code, giving way to the possibility of live coding (Sigurdsson 2016). Many opcodes from Csound have been ported to Sporth from C code (Batchelor 2017a; sic 2010).

2 Motivations for a Live Coding Interface

Sporth was not initially designed to do any kind of live coding; The desire for implementing code re-evaluation came after many months of composing with the language. Interfaces for computer music should be realtime whenever possible (Wang 2014), and they should feel responsive. For intimate musical control of computers, it is established that responsive latency times need to be 10ms or lower (Wessel and Wright 2002; Freed, Chaudhary, and Davila 1997). A live coding interface could easily meet these demands, providing perceptually instantaneous code re-evaluation. This would ultimately lead to a very rewarding iterative compositional process which will be referred to as a *creative feedback loop*.

3 Language Overview

The following section will provide a brief overview of the Sporth language. The focus will be specifically targetting the core syntax. Further semantics, such as the underlying synthesis concepts, are beyond the scope of this paper. Highly motivated readers are encouraged to look at the existing documentation for discussion on such topics (Batchelor 2017c).

3.1 Understanding the Stack

Sporth belongs to the family of languages known as “stack-based” languages. As the name would suggest, stack-based languages are centered around a global stack data structure, where data is pushed and popped in a last in, first out (LIFO) manner. The subroutines that push (write) and pop (read) to the stack are traditionally known as *words*. The following Sporth code shows how two numbers can be added together.

```
20 17 +
```

In the example above, the numbers 20 and 17 are pushed onto the stack. The word “+” pops 17 and 20 off the stack, adds them together, then pushes that value back onto the stack.

3.2 Unit Generators

Sporth is a modular synthesis environment. A synthesized sound in Sporth is made up of a collection of small interconnected signal generators, known as *unit generators* (Mathews 1963). Unit generators are represented as words in Sporth:

```
440 0.5 sine
```

In the patch above, a 440Hz sine wave is played with a 0.5 amplitude value. The numbers “440” and “0.5” are pushed onto the stack. The word “sine” pops the values of the stack, and uses these to calculate a single sample of a sinusoid, which then gets pushed onto the stack.

At the end of the Sporth patch, the last value on the stack is popped and sent to the speakers as a single sample of audio. If too many items are on the stack at the end of a patch, a stack overflow occurs. If too few occur, a stack underflow occurs.

Unit generators can be mixed and layered together. The following patch adds two sine waves together to make a North American dial-tone (also know as DTMF tones):

```
440 0.2 sine
350 0.2 sine
+
```

Sporth is largely white-space insensitive, and line breaks can be made in between words. Well placed breaks are a good way to organize portions of Sporth code.

Instead of thinking of the control flow items being pushed and popped, it is more appropriate to conceptualize things as a signal flowing left to right.

3.3 Modulation

Parametric modulation over time is the foundation of modular synthesis.

All signals generated in Sporth are audio rate. There is no concept of a control rate signal, a traditional feature in other computer-music languages like Csound, SuperCollider, MaxMSP, or PD.

The example below shows how one sine oscillator can be used to modulate the frequency of another sine oscillator:

```
6 40 sine
440 +
0.5 sine
```

This patch first creates a low-frequency sine wave (LFO) at 6Hz with an amplitude of 40 units. This LFO is then biased so it is centered around 440. This biased LFO is then used to drive the frequency of an audible sine wave.

Due to the sample-accurate audio-rate nature of Sporth, the LFO signal can be bumped to audio-rate frequencies, which in turn produce FM spectra:

```
(((440 440 sine) 440 +) 0.5 sine)
```

To showcase different ways of organising code in Sporth, the line break groupings have been removed and replaced with parentheses, providing something that could easily be mistaken for a reversed s-expression. Parentheses are ignored in Sporth, so they need not be matched. This patch produces a elementary 440Hz FM oscillator pair with a C:M ratio of 1:1, and a modulation index amount of 1. Sporth has a built-in FM oscillator. The following patch would be equivalent:

```
440 0.5 1 1 1 fm
```

3.4 Triggers and Gates

Triggers and gates are the two recognized control signals in Sporth for timing. Some unit generators, such as envelopes or sequencers, have inputs that recognize these kinds of signals.

A trigger signal is a single-sample non-zero impulse. Typically, triggers are generated by metronome or clock signal generators. The following patch illustrates how trigger signals can be used to generate an envelope signal.

```
4 metro 0.5 maytrig
0.001 0.01 0.01 tenvx
1000 0.5 sine *
```

The metronome, before being sent into the envelope generator, is fed into a unit-generator called *maytrig*. The maytrig is a special filter which takes in a trigger signal, and is set to have 50 percent probabilty of sending out a trigger. The maytrig signal is fed into the envelope generator *tenvx*, which takes in parameters for attack, hold, and release, respectively. This envelope signal is then mupltied with a 1kHz sine tone.

Gate signals are steady-state "on" or "off" signals, represented by a non-zero value or zero. While fewer unit generators expect gate signals, they are nonetheless useful for things like envelopes that have an indefinite duration. The Sporth patch below shows another enveloped sine patch using an ADSR envelope:

```
2 metro tog
0.01 0.01 0.5 0.1 adsr
5000 0.4 sine *
```

3.5 Function Tables

Function tables, or f-tables, get their name from the Csound language (Boulanger 2000). They are arrays of floating point values, used in places like table lookup oscillators or sequencers. Init-time functions can be called to map values on to f-tables, like sampling a single period of an oscillator. Functions like these are known as GEN routines, another term taken from the Csound/Music N family of languages.

In Sporth, f-tables are referenced by unique string values. In the example below, the sine unit-generator has been replaced with the more generalized table-lookup oscillator osc:

```
"sinewave" 8192 gen_sine
440 0.5 0 "sinewave" osc
```

The oscillator table-lookup oscillator takes in 2 additional arguments, phase offset and the name of the table. In this case, the table name "sinewave" is a 8192-sized sample sinusoid created via gen_sine.

3.6 Stack operations

Stack-based languages have a set of so-called stack operations, which are words that are able to manipulate items on the stack. In Sporth, a handful of these operations are implemented. The patch below is a sequenced sine tone with an amplitude envelope. The stack operations "dup" and "swap" are used so that the signal generated by metro can clock both the envelope generator *tenv* and the sequencer *tseq*:

```
"seq" "0 2 4 7" gen_vals
4 metro
dup
0.001 0.001 0.001 tenv
swap
0 "seq" tseq
60 + mtof
0.4 sine *
```

Stack operations, while useful for small things like splitting signals, can get very complicated to follow when many are used in succession. For these situations, it is best practice to use variables.

3.7 Variables

Variables are a good solution for signals that are used more than once within a patch. Like function tables, they are referenced via a name. There are three Sporth words that are used for working with variables:

The word "var" is used to instantiate a variable The word "set" is used to set the value of a variable The word "get" is used to get the name of the variable

Seen below, the example from the previous has been refactored to use variables to store the clock signal, in a variable called "clk".

```
"clk" var
"seq" "0 2 4 7" gen_vals
4 metro "clk" set
"clk" get 0.001 0.001 0.001 tenv
"clk" get 0 "seq" tseq
60 + mtof
0.4 sine *
```

Strings without spaces can be represented as the string prepended with an underscore, instead of quotes. This syntactic sugar saves a keystroke and provides a little more readability:

```
_clk var
_seq "0 2 4 7" gen_vals
4 metro _clk set
_clk get 0.001 0.001 0.001 tenv
_clk get 0 "seq" tseq
60 + mtof
0.4 sine *
```

4 Live Coding with Sporth

4.1 The implementation

The live coding interface for Sporth centers around a UDP listener. This listener thread is automatically started when Sporth is instantiated. Sporth text is sent over UDP to the listener, where it is copied into an internal string buffer. The audio thread gets a notification of a recieved pack, and at the beginning a new audio buffer to write, it parses the string. If it is valid, it hotswaps the Sporth data and cleans the old data. If it is invalid, it is thrown away.

All parsing and re-evaluation happens inside the audio thread, rather than inside the listener thread. This unusual design choice is due to aspects of the Sporth architecture that are not thread-safe. In theory, this would be a bad design choice, as code re-evalutation could lead to glitches in the audio. In practice, however, this worry has yet to be an issue. Parsing Sporth code is typically a cheap process. There is virtually no grammar, so only lexing and tokenizing are required. Sporth patches tend to be very terse, so the scale of what is to be parsed is small.

4.2 The Interface

The preferred interface for live coding is the Vim text editor, with the Sporth listener spawned in another window. A custom keymap is used so that space+s automatically selects a block of Sporth code in parentheses to be piped to a perl script. This small perl script sends this code over UDP to the listener.

5 Discussion

Alan Perlis famously once said "A language that doesn't affect the way you think about programming is not worth knowning" (Perlis 1982). Stack based languages offer a unique approach to program structure and design, and this has great potential in the realm of interactive creative coding. Sporth, the main language highlighted in this paper, utilizes the stack-based approach to very concisely define very complex signal routings that would be very laborious to do in other sound design environments. It is the hope of the author that concepts explored in Sporth and will inspire the curious to explore incorporating stack-based ideas into their own live coding ecosystems.

References

Batchelor, Paul. 2017a. "Soundpipe: A Lightweight Musical Digital Signal Processing Library." paulbatchelor.github.io/proj/soundpipe.html.

———. 2017b. "Sporth: A Small Stack-Based Audio Programming Language." paulbatchelor.github.io/proj/sporth.html.

———. 2017c. "The Sporth Cookbook." paulbatchelor.github.io/proj/cook.

Boulanger, Richard Charles. 2000. *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming*. MIT press.

Brodie, Leo. 2004. *Thinking Forth*. Punchy Pub.

Freed, Adrian, Amar Chaudhary, and Brian Davila. 1997. "Operating Systems Latency Measurement and Analysis for Sound Synthesis and Processing Applications." In *ICMC*.

Mathews, Max V. 1963. "The Digital Computer as a Musical Instrument." *Science* 142 (3592). American Association for the Advancement of Science: 553–57.

McCartney, James. 2002. "Rethinking the Computer Music Language: SuperCollider." *Computer Music Journal* 26 (4). MIT Press: 61–68.

Perlis, Alan J. 1982. "Epigrams on Programming." *SIgPLAN Notices* 17 (9): 7–13.

sic. 2010. "Understanding an Opcode in Csound." In *The Audio Programming Book*, edited by Richard Boulanger and Victor Lazzarini, 581–617. MIT press.

Sigurdsson, Hlödver. 2016. "Live Coding with Csound." *The Csound Journal*, no. 22.

Wang, Ge. 2014. "Principles of Visual Design for Computer Music." In *ICMC*.

Wang, Ge, Perry R Cook, and Spencer Salazar. 2016. "Chuck: A Strongly Timed Computer Music Language." *Computer Music Journal*. MIT Press.

Wessel, David, and Matthew Wright. 2002. "Problems and Prospects for Intimate Musical Control of Computers." *Computer Music Journal* 26 (3). MIT Press: 11–22.