SYNTHTAX

Vicky Nguyen, Skylar Buck
Department of Engineering and Computer Science, University of Victoria, Canada
viennguyen@uvic.ca, gsbucko1@hotmail.com

ABSTRACT

Synthtax is a dynamically typed, imperative music programming language designed for creating and manipulating sound. This minimal language is able to supports some programming constructs such as loops, conditionals, and function calls. Synthtax provides a mechanism to generate and modify sound, specifically, oscillators and ADSR. The language is implemented as a C++ transpiler using the C++ programming language and ANTLR.

1. INTRODUCTION

The creation and manipulation of sound has long been a fundamental component of human artistic expression. As technology has advanced, so too have the tools available for music creation and manipulation. Music programming languages have emerged as a powerful tool for creating custom sounds and manipulating audio.

Synthtax is a dynamically typed, imperative music programming language designed for creating and manipulating sound. The language supports some programming constructs such as loops, conditionals, and function calls.

The paper is divided into 2 main sections. Section 2 will describe the design of Synthtax, and present examples of its use in creating custom sounds and manipulating audio. Then, section 3 will present the implementation details. Finally, section 4 will discuss the limitations and future work.

2. DESIGN

Synthtax is a small, simple, C like programming language except that it supports `string` datatype and it is dynamically typed. Synthtax allows implicit type conversion between `int` and `float`.

2.1 Lexical elements

- **Keywords**: `fun`, `if`, `else`, `while`, `return`, `true`, `false`, `Osc`, `play`, `ADSR`, `apply`
- **Punctuation symbols**: `,`, `;`, `{}`, `()`
- **Binary operators**: `==`, `<`, `+`, `-`, `*`, `/`
- **Literals**: `string`, `int`, `float`, `char`
- **Characters**: `[a-zA-Z_]`
- **Identifiers**: `[a-zA-Z_]+`
- **Comments**: there are 2 types
  - Single line: start with `//`
  - Block: between `/*` and `*/`

2.2 Built-in Functions

- **`Osc`** `(oscillator type: string, frequency: number, amplitude: number)`: create an oscillator
  - Input: the first argument is the type of the oscillator (i.e., `sine`, `triangle`, `square`, `sawtooth`). The second one is the frequency. The last argument is the amplitude.
  - Output: an oscillator object
  - Example: `Osc("sine", 440.5, 0.8)`
- **`ADSR`** `(attack from: number, attack to: number, attack duration: number, decay from: number, decay to: number, decay duration: number, sustain from: number, sustain to: number, sustain duration: number, release from: number, release to: number, release duration: number )`: create an ADSR
  - Input: start amplitude, destination amplitude, and duration in millisecond of the 4 stages: attack, decay, sustain, and release
  - Output: an ADSR object
  - Example:

```
/*
attack: 0 to 1.0 for 200ms
decay: 1.0 to 0.8 for 100ms
sustain: 0.8 to 0.8 for 1000ms
release: 0.8 to 0 for 100ms
*/
ADSR(0,1.0,200,0.8,100,0.8,1000,0,100);
```

- **`play`** `(sound: sound object, duration: number)`: play sound
  - Input: sound object, which is an array of number, and duration in millisecond
  - Output: none. This function will call `RTAudio` library to play sound
  - Example:

```
out = Osc("triangle", 200, 1);
play(out, 1000);
```

- **`apply`** `(ADSR: ADSR object, sound: sound object)`: apply ADSR to a sound object
  - Input: an ADSR object and an array of number representing a sound
  - Output: none
  - Example:

```
a = ADSR(0,1.0,200,0.8,100,0.8,
             1000,0,100);
sinewave = Osc("sine", 420.5, 1);
output = apply(a, sinewave);
```

2.3 Sample Programs

A sample program with loop, function call, control flow, oscillator, and ADSR.

```
fun get_value(n) {
  i = 0;
  value = 0;
  while (i < n) {
    if (i < 2) {
        value = value - i;
    } else {
        value = value + i;
    }
    i = i + 1;
  }

  return value;
}

fun main() {
  value = get_value(5);
  freq = value * 100.5;
  osc = Osc("sine", freq, 1);
  adsr = ADSR(0,1.0,200,0.8,
            100,0.8,1000,0,100);
  output = apply(adsr, osc);
  play(output, 1000);
}
```

3. IMPLEMENTATION

3.1 Lexer and Parser

ANTLR is a powerful tool to generate a parser for a giving grammar [3]. The Synthtax grammar can be found on GitHub [1] . Debugging a grammar can be tricky because the error in the grammar can be very subtle and error messages can be cryptic and misleading at times. For example, both of the versions of the lexer rules below might look similar, but they can produce different outputs.

Version 1:

```
IDENTIFIER: [A-Za-z]+;
FOO: 'foo';
```

Version 2:

```
FOO: 'foo';
IDENTIFIER: [A-Za-z]+;
```

Suppose the parser rule is `program:FOO` and the input is `foo`. Version 1 will get an error `mismatched input 'foo' expecting 'foo'`. This is because the `IDENTIFIER` is defined first and it eats all of the characters. Thus, the program cannot find `FOO`. However, swapping the order of the rules like in version 2 can produce the correct parse tree.

ANTLR is made up of two main parts: the tool and the runtime [4]. The tool consists of Java and an ANTLR `jar` file, which is included in the project. There is a different runtime for every target language. Since the target language is C++, C++ runtime library is needed to generate the lexer and parser code. To use the newly generated clases, they need to be compiled and linked against the C++ runtime library. Building a C++ project on different platforms can be quite complicated. Thankfully, the official ANTLR repository provides an Antlr4Cpp [2] external project, which aids the integration process.

3.2 Transpiling to C++ Code

`RTAudio` [3] is a good C++ library that can generate audio, so it is convenient to somehow incorporate it. A shortcut is to transpile Synthtax into C++ and link with the RTAudio and the pre-made oscillators and ADSR codes. Moreover, by doing this, type check and semantic check can be disregarded as we can rely on the C++ compiler to do them. However, one downside of this method is that the program has to compile twice, which is not efficient. Nevertheless, within the limited time budget, performance has a low priority.

A standard Visitor can be generated by ANTLR. ANTLR also generates a default implementation of the class (i.e., `SynthtaxParserBaseVisitor.h`). This implementation can be used to output the corresponding C++ code.

For example, given the folowwing Synthtax program:

```
fun foo() {}
```

The corresponding C++ code generated by the Visitor will be:

```
void foo() {}
```

4. FUTURE WORK

Synthtax is a small language and it does not support many programming constructs. In the future, we will expand its capabilities such as adding arrays and user defined objects, and some digital processing concepts such as subtractive synthesis, frequency modulation, and getting MIDI input.

5. TECHNICAL CONTEXT IN TERMS OF METHODOLOGY

- ANTLR, one of the most popular parser generators used in music programming languages, relies on formal grammars and parsing techniques. These techniques are based on mathematical concepts like context-free grammars and finite-state machines. Parsing is also often paired with lexical anal-

[1] https://github.com/iamvickynguyen/Synthtax/blob/antlr/SynthtaxParser.g4

[2] https://github.com/antlr/antlr4/tree/master/runtime/Cpp/cmake

[3] https://github.com/thestk/rtaudio

ysis (lexing), which involves identifying the different "tokens" or units of meaning in a music program's code [2].

- ADSR (Attack, Decay, Sustain, Release) envelopes are commonly used in music programming languages to shape the amplitude of sounds over time. These envelopes are typically modeled using mathematical functions, such as exponential curves, to achieve smooth and precise transitions between the different envelope stages [1].
- Synthesizing complex waveforms often involves adding together simpler waveforms in various ways. For instance, adding sine waves with different frequencies and phases can create a richer, more complex waveform. This process often involves techniques from Fourier analysis and other areas of mathematical signal processing.

6. CONCLUSION

The design of Synthtax is focused on simplicity and ease of use. The language supports basic programming constructs such as loops, conditionals, and function calls. Additionally, Synthtax provides a mechanism for generating and modifying sound using oscillators and ADSR.

The implementation of Synthtax is as a C++ transpiler using ANTLR. This design choice allows the language to take advantage of the performance and efficiency of C++, while also allowing it to be used with the RTAudio library for audio processing.

Currently, synthtax has some limitations, such as a lack of support for more complex programming constructs. In the future, there is room for improvement and expansion, including the addition of more advanced programming constructs and support for additional audio processing libraries.

7. REFERENCES

[1] Kevin August Meinert. Subsynth: A generic audio synthesis framework for real-time applications. https://dr.lib.iastate.edu/server/api/core/bitstreams/140b830b-fa83-48f4-b4a6-0cf0d6faf0a1/content, 2002.

[2] Pietro. (n.d.). How to build a new programming language. https://pgrandinetti.github.io/compilers/page/how-to-build-a-new-programming-language/.

[3] Terence Parr. What is antlr. https://www.antlr.org/.

[4] Gabriele Tomassetti. Getting started with antlr in c++. https://tomassetti.me/getting-started-antlr-cpp/, 2021.