An Introduction to Octave Compiler

Burnham310 DevTeam

University of Illinois at Urbana-Champaign

1 Introduction

Octave is an experimental music programming language designed to facilitate music composition in a programmable way. In this language, users can write music using syntax reminiscent of traditional musical notation while defining variables, functions, conditional statements, and loops. These compositions are compiled into MIDI files.

This document provides an introduction to the Octave language, detailing the core principles behind its compiler's frontend and backend. Like most programming language compilers, the Octave compiler has two main components: the frontend, which includes the lexer and parser, and the backend, which converts the Abstract Syntax Tree (AST) into MIDI binary output. To understand these processes, we will first discuss how the backend operates and the abstractions that drive it. Then, we will provide a formal description of the syntax, semantics, and type system of Octave. We use Backus-Naur Form (BNF) to define the syntax, Big-Step Structural Operational Semantics (SOS) for the semantics, and a type system to ensure type correctness.

2 Backend Concepts

This section introduces the key concepts derived from the Octave backend design and explain how it follows and builds upon an event-driven architecture. This abstraction has proven to be highly effective in supporting the development of both the parser and lexer, facilitating a seamless integration to the compuler fontend of the compiler.

2.1 MIDI Abstraction

A standard MIDI file (SMF) is organized into two main components: the header chunk and one or more track chunks. The header chunk sets the file's format, timing, and the number of tracks, laying the groundwork for playback. Meanwhile, each track chunk contains the actual musical events in sequence, such as note instructions, tempo adjustments, and other performance parameters. This structure allows MIDI files to remain highly compatible across various devices and versatile for applications ranging from live performances to digital music production.

In a track chunk, musical data is stored in a standardized sequence to ensure accurate playback. The general syntax for a track chunk is:

```
<MTrk event> = <delta-time><event>
```

Here, <delta-time> is a variable-length quantity representing the time elapsed before the subsequent event. If two events occur simultaneously, a delta-time of zero is used. Delta-times are always required, with a granularity that can be adjusted based on the time units defined in the header chunk.

- <MIDI event> - Events that include Note On (press key), Note Off (release key), Channel Volume (adjust the volume of a specific channel), among others.
- <sysex event> - System-exclusive events for custom data.
- <meta-event> - Additional metadata events, including global settings like tempo adjustments.

For instance, the tempo adjustment event is global, while a volume adjustment is specific to each track. In the Octave compiler, we simplify MIDI handling through two main abstractions: Track and Event.

```
struct _MTrkEvent
{
    enum MTrkEventType eventType;
    int delta_time;
    void *data;
    void (*callbacks[5])(int delta_time, struct _EventData, void *data);
    void (*destroyer)(void *data);
};

struct _MidiTrack
{
    size_t cap;
    size_t event_count;
    int channel;
    struct MidiTrackMeta meta;
    struct _MTrkEvent *events;
};

void add_midi_event(int track_id, struct _MTrkEvent event);
```

This structure provides an efficient and extensible interface for the frontend to generate MIDI events in a standardized format. Through development, we found this approach effective and adaptable.

2.2 Backend Abstraction

In addition to MIDI abstraction, the backend handles more complex musical constructs that are not native to the MIDI standard. For example, Octave includes the `Interpolate` function feature, enabling users to define smooth attribute adjustments across sections. Consider the following syntax:

```
@build_up[volume=10 linear]

    1.. oo'2.. o'7.. o'5..
    2... 3... 5... 6... o'2... o#'4... o'5... oo'2...

@done[volume=100]
```

In this example, the volume increases linearly from 10 to 100 over the duration of the section. The backend processes these directives by inserting a sequence of volume change events at each interval of the MIDI frame.

In summary, the Octave backend translates high-level constructs, such as linear volume adjustments, into the appropriate sequence of MIDI events, ensuring that complex musical structures are efficiently encoded in MIDI format. Each track in a MIDI file is essentially a sequence of events arranged in chronological order. The backend can insert specific sequences of events at designated points within a track to achieve dynamic effects, like gradual volume changes or tempo shifts, while preserving the integrity of the event sequence. This structure allows Octave to handle intricate musical transformations and accurately reflect them in the final MIDI output.

3 Evaluator Design

The frontend for Octave is built entirely from scratch in C. For parsing, we utilize a recursive descent parser, incorporating the concept of binding power, inspired by the Pratt Parser, to handle operator associativity. This approach follows standard practices commonly used in parser and lexer implementations. However, the evaluation step is somewhat unconventional. Once the Abstract Syntax Tree (AST) is generated through parsing, the challenge is transforming it into MIDI format.

Our evaluator processes the AST by dividing it into two main components: 1) tracks, and 2) modifiers. The tracks represent a sequence of MIDI events construct by constructor operators, while the modifiers involve actions that alter the behavior of these events, such as the Interpolate function, which adjusts properties like volume or tempo over time.

To formalize this process, I will illustrate how we transform the syntax into a sequence of events and how we evaluate the corresponding modifiers to achieve the desired MIDI output.

3.1 Evaluate Tracks

The evaluation of tracks in Octave involves rewriting the code into a canonical term, which construct by event constructor (see 5.1), and constructor operator, i.e, a `SPACE` for sequential execution. This algebraic structure offers a structured representation of musical events, allowing easy conversion into MIDI binary as the canonical form produced resembles our MIDI abstraction.

3.2 Evaluate Modifiers

Currently, the only modifier supported is the `Interpolator` function, which allows smooth transitions of musical attributes such as volume across a specified segment of a section. The evaluation of modifiers occurs after the `Tracks` have been evaluated, resulting in terms composed of event constructors and constructor operators, as previously mentioned.

The `Interpolator` function operates as a transformation function with the following signature:

```
Track, (ModifierPos, ModifierPos) -> Track
```

This function takes a track, along with the current modifier's position and the position of the next modifier, and adjusts the track's events based on the specified modifier settings. For example, if the `Interpolator` is set to transition volume from a low to a high value over time, the function inserts volume change events at intervals within the specified segment.

4 Octave Syntax

The syntax of Octave is defined using BNF notation as follows. We assume parentheses are implicit in the syntax, used for grouping and to avoid ambiguity.

$$
\begin{aligned}
\langle \text{Pgm} \rangle &::= \langle \text{Decl} \rangle^{*} \\
\langle \text{Decl} \rangle &::= \langle \text{Ident} \rangle = \langle \text{Expr} \rangle \\
\langle \text{Ident} \rangle &::= [a - z\_][a - z\_\langle \text{Digit} \rangle\rangle]* \\
\langle \text{Qual} \rangle &::= [so\#b]^{+\prime} \\
\langle \text{Expr} \rangle &::= \langle \text{Ident} \rangle \\
&\quad \mid \langle \text{Int} \rangle \\
&\quad \mid \langle \text{Expr} \rangle \ .^{+} \\
&\quad \mid \langle \text{Qual} \rangle \ \langle \text{Expr} \rangle \\
&\quad \mid \langle \text{Expr} \rangle \ ' \ \langle \text{Expr} \rangle \\
&\quad \mid [\langle \text{Expr} \rangle^{*}] \\
&\quad \mid |\{\langle \text{Decl} \rangle^{*}, \langle \text{Decl} \rangle^{*}\}{:}\langle \text{Expr} \rangle^{*}| \\
&\quad \mid \langle \text{Expr} \rangle \ \& \ \langle \text{Expr} \rangle \\
&\quad \mid \langle \text{Expr} \rangle + \langle \text{Expr} \rangle \\
&\quad \mid \langle \text{Expr} \rangle - \langle \text{Expr} \rangle \\
&\quad \mid \langle \text{Expr} \rangle * \langle \text{Expr} \rangle \\
&\quad \mid \langle \text{Expr} \rangle \ ' \ \langle \text{Expr} \rangle \\
&\quad \mid \text{if } \langle \text{Expr} \rangle \text{ then } \langle \text{Expr} \rangle \text{ else } \langle \text{Expr} \rangle \text{ end} \\
&\quad \mid \text{for } \langle \text{Expr} \rangle \sim (<|<=) \ \langle \text{Expr} \rangle \text{ loop } \langle \text{Expr} \rangle^{*} \text{ end} \\
\langle \text{Digit} \rangle &::= [0 - 9] \\
\langle \text{Int} \rangle &::= [1 - 9][0 - 9]^{*}
\end{aligned}
$$

5 Big-Step Semantics

We use Big-Step Structural Operational Semantics (SOS) to define how expressions and statements in Octave evaluate. In Big-Step semantics, we describe how a program configuration (such as a statement and a state) directly produces a final result or state. This style of semantics abstracts away intermediate steps, focusing instead on the overall outcome of evaluating a statement or expression.

5.1 Event Constructors for Octave

In Octave, event constructors represent MIDI-compatible events. These events form the building blocks for *Tracks*, allowing us to create musical sequences with precise control over pitch, duration, volume, tempo, and instrument.

- **NoteOnEvent**: Represents a key press on a specific pitch with a defined duration.

$$\mathrm{NO}(n, d)$$

  where:

  - n: Note primitive.
  - s: Scale of the note.

- **NoteOffEvent**: Represents a key release on a specific pitch.

$$\mathrm{NOF}(c, s, d)$$

  where:

  - n: Note primitive.
  - s: Scale of the note.

- **SetTempoEvent**: Changes the current tempo in beats per minute (BPM).

$$\mathrm{ST}(bpm)$$

  where:

  - bpm: New tempo in beats per minute.

- **SetInstrumentEvent**: Sets the instrument using a program change number, as per MIDI specifications.

$$\mathrm{SI}(pc)$$

  where:

  - pc: Program change number for the instrument (refer to the MIDI instrument list for mappings).

- **SetVolumeEvent**: Sets the volume to an absolute level.

  $$\text{VTO}(v)$$

  where:

  - v: Volume level (an integer representing the MIDI volume, usually from 0 to 127).

5.2 Big-Step Configuration for Octave

For the Big-Step semantics of the Octave language, we define configurations using a similar comma-and-angle-bracket notation. In Octave, configurations represent various components of musical expressions, tracks, and states, which are needed for transforming Octave code into MIDI-compatible events. The configurations in Octave include expressions, tracks, and states, with some configurations holding only specific values (e.g., volume or instrument settings) without a full state. Here are the configuration types needed for the Big-Step semantics of Octave:

- $\langle i \rangle$: holding integers i.
- $\langle p \rangle$: holding pitch p.
- $\langle l \rangle$: holding list l.
- $\langle n \rangle$: holding note n.
- $\langle e, \sigma \rangle$: grouping an expression e and states σ.
- $\langle t \rangle$: holding true values $t\ \in \{true, false\}$.
- $\langle s, \sigma \rangle$: grouping statements s and states σ.
- $\langle \sigma \rangle$: holding state σ.
- $\langle pgm \rangle$: holding program pgm.

5.3 Big-Step SOS Helper Functions

The following functions are used in the Big-Step SOS for Octave, each serving to improve readability and enhance the clarity of the overall semantics. These functions encapsulate specific tasks, making the rules more organized and easier to follow. Importantly, the inclusion of these functions does not change the meaning of the Big-Step rules—they are simply syntactic helpers. Without them, the semantics would remain the same, albeit less readable.

- **init(σ)**: This function takes a state σ as input and returns `true` if any values in the current state differ from the previous state, and `false` otherwise. It is useful for ensuring that configurations are applied only when changes in state are detected.

5.4 Big-Step Semantics Rules

$$\langle i, \sigma\rangle \Downarrow \langle i\rangle \quad \text{(Int)}$$

$$\langle p, \sigma\rangle \Downarrow \langle p\rangle \quad \text{(Pitch)}$$

$$\langle l, \sigma\rangle \Downarrow \langle l\rangle \quad \text{(List)}$$

$$\langle n, \sigma\rangle \Downarrow \langle n\rangle \quad \text{(Note)}$$

$$\langle x, \sigma\rangle \Downarrow \langle \sigma(x)\rangle \quad \text{(Lookup)}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow \langle v_1\rangle \ldots \langle e_n, \sigma\rangle \Downarrow \langle v_n\rangle}{\langle [e_1 \ldots \ e_2], \sigma\rangle \Downarrow \langle l(v_1, \ldots, v_n)\rangle} \quad \text{(Expr-List)}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle \quad \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 + a_2, \sigma\rangle \Downarrow \langle i_1 +_{int} i_2\rangle} \quad \text{(Int-Add)}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle \quad \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 - a_2, \sigma\rangle \Downarrow \langle i_1 -_{int} i_2\rangle} \quad \text{(Int-Min)}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle b_1\rangle \quad \langle a_2, \sigma\rangle \Downarrow \langle b_2\rangle}{\langle a_1 == a_2, \sigma\rangle \Downarrow \langle a_1 \oplus a_2\rangle} \quad \text{(Bool-Eq)}$$

$$\frac{\langle e_1 \Downarrow i_1\rangle \quad \langle e_2 \Downarrow i_2\rangle}{\langle e_1 < e_2, \sigma\rangle \Downarrow \langle i_1 <_{int} i_2, \sigma\rangle} \quad \text{(LT)}$$

$$\frac{\langle e_1 \Downarrow i_1\rangle \quad \langle e_2 \Downarrow i_2\rangle}{\langle e_1 > e_2, \sigma\rangle \Downarrow \langle i_1 >_{int} i_2, \sigma\rangle} \quad \text{(GT)}$$

$$\frac{\langle e_1 \Downarrow i_1\rangle \quad \langle e_2 \Downarrow i_2\rangle}{\langle e_1 <= e_2, \sigma\rangle \Downarrow \langle i_1 <=_{int} i_2, \sigma\rangle} \quad \text{(LEQ)}$$

$$\frac{\langle e_1 \Downarrow i_1\rangle \quad \langle e_2 \Downarrow i_2\rangle}{\langle e_1 >= e_2, \sigma\rangle \Downarrow \langle i_1 >=_{int} i_2, \sigma\rangle} \quad \text{(GEQ)}$$

$$\frac{\langle e_1 \Downarrow v_1\rangle \quad \langle e_2 \Downarrow v_2\rangle}{\langle e_1 \ e_2, \sigma\rangle \Downarrow \langle i_1 >=_{int} i_2, \sigma\rangle} \quad \text{(GEQ)}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle b_1\rangle \quad \langle a_2, \sigma\rangle \Downarrow \langle b_2\rangle}{\langle a_1 ! = a_2, \sigma\rangle \Downarrow \langle \neg(a_1 \oplus a_2)\rangle} \quad \text{(Bool-Neq)}$$

$$\frac{\langle e, \sigma\rangle \Downarrow \langle l\rangle}{\langle e\ [\ .]^m\rangle \Downarrow \langle n(l, m)\rangle} \quad \text{(Note-Ctor)}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow \langle \text{true}\rangle \quad \langle e_2, \sigma\rangle \Downarrow \langle v\rangle}{\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3, \sigma\rangle \Downarrow \langle v\rangle} \quad \text{(If-Then-Else-True)}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow \langle \text{false}\rangle \quad \langle e_3, \sigma\rangle \Downarrow \langle v\rangle}{\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3, \sigma\rangle \Downarrow \langle v\rangle} \quad \text{(If-Then-Else-False)}$$

$$\frac{\langle e_1 < e_2, \sigma\rangle \Downarrow \text{true} \quad \langle e_1 \sim< e_2 \text{ loop } e \text{ end}, \sigma\rangle \Downarrow \langle v\rangle}{\langle \text{for } e_1 \sim< e_2 \text{ loop } e \text{ end}, \sigma\rangle \Downarrow \langle v \text{ for } e_1 \sim< e_2 \text{ loop } e \text{ end}\rangle} \quad \text{(For-Loop-True)}$$

$$\frac{\langle e_1 < e_2, \sigma\rangle \Downarrow \text{true} \quad \langle e_1 \sim< e_2 \text{ loop } e \text{ end}, \sigma\rangle \Downarrow \langle v\rangle}{\langle \text{for } e_1 \sim< e_2 \text{ loop } e \text{ end}, \sigma\rangle \Downarrow \langle \text{skip}\rangle} \quad \text{(For-Loop-False)}$$

$$\frac{\sigma'(\text{Volume}) \quad \sigma'(\text{Instrument}) \quad \sigma'(\text{Tempo}) \quad \mathit{init}(\sigma') \Downarrow \langle \text{true}\rangle}{\langle \sigma'\rangle \Downarrow \langle \text{VTO}(\sigma'(\text{Volume}))\ \text{SI}(\sigma'(\text{Instrument}))\ \text{ST}(\sigma'(\text{Tempo}))\rangle} \quad \text{(Cfg-Eval)}$$

$$\frac{\langle d_1 \dots d_m, \sigma\rangle \Downarrow \langle \sigma'\rangle \quad \langle e_1 \dots e_n, \sigma'\rangle \Downarrow \langle n_1 \dots n_n, \sigma'\rangle}{\langle | d_1 \dots d_n : d_{n+1} \dots d_m : e_1 \dots e_n |, \sigma\rangle \Downarrow \langle \text{NO}(n, \sigma'(\text{Scale}))\ \text{NOF}(n, \sigma'(\text{Scale}))\rangle} \quad \text{(Section)}$$

$$\frac{\langle a, \sigma\rangle \Downarrow v}{\langle x = a, \sigma\rangle \Downarrow \sigma[v/x]} \quad \text{(Decl)}$$

$$\frac{\langle d_1, \sigma\rangle \Downarrow \sigma' \dots \langle d_n, \sigma'\rangle \Downarrow \sigma''}{\langle d_1 \dots d_2, \sigma\rangle \Downarrow \sigma''} \quad \text{(Pgm)}$$

6 Type System of Octave

A type system is used to ensure that programs are type-safe, meaning they do not perform operations on incompatible types. In Octave, type checking plays a critical role in validating expressions, configurations, and track operations before execution. When a type check fails, Octave will immediately throw an error with a comprehensive error message, detailing the nature of the type incompatibility and the exact location in the code where the error occurred. This approach enhances the safety and robustness of the language, preventing invalid operations from being evaluated and ensuring clear feedback for the user.

6.1 Type Rules

The typing judgments are written as $\Gamma \vdash e : T$, meaning that in typing environment Γ, expression e has type T.

- **Typing Expressions**

$$\frac{\Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 + e_2 : \text{Int}} \quad \text{(T-Add)}$$

- **Typing Conditionals**

$$\frac{\Gamma \vdash e : \text{Bool} \quad \Gamma \vdash s_1 : T \quad \Gamma \vdash s_2 : T}{\Gamma \vdash \text{if } e \text{ then } s_1 \text{ else } s_2 : T} \quad \text{(T-If)}$$

[TODO]