EECS 510 Final Project

Aiden Barnard, Liam Aga, Sam Kelemen

May 10, 2025

1 Language Overview

Intent, Structure, and Purpose

For our formal language, we chose to model and validate piano measures written in 4/4 time using a pushdown automaton (PDA). In standard musical notation, each 4/4 measure contains exactly four beats. To represent this precisely, we scale the measure to 32 timing units, where one beat equals 8 units. Our language accepts sequences of notes and rests whose combined rhythmic value adds up to exactly 32 units, ensuring each measure is rhythmically complete and musically valid.

Beyond rhythmic correctness, our language enforces pitch structure. Each note must specify:

- a pitch class (A–G),
- an optional accidental (sharp # or flat b),
- an octave number (0 through 8),
- and a duration (e.g., whole, half, quarter).

Rests are represented using an underscore (_) and only carry a duration.

Language Syntax and Strings

Our language's alphabet (terminal symbols) includes pitch letters, accidentals, octave digits, rhythmic durations, rest indicators, and control symbols like | (measure end) and $\perp$ (end of piece). A valid string is composed of tokens appended in the following order:

`[Pitch Class] [Accidental (optional)] [Octave] [Duration]`

Each string ends with a | to mark the end of the measure, and the full input ends with $\perp$ to indicate the end of the piece.

Example of a Valid String

`A4w | E5h F5h` $\perp$

Where A natural is the note, scaled at the 4th octave, then held for a duration of a whole note (4 beats/ 32 Xs). This completes the first measure. This is then followed by an E5 half note and a F5 half note to complete the second measure at a full 32 units (16 per half note).

This string contains a whole note (32 units) in the first measure, and two half notes (16 units each) in the second measure, both summing to valid 32-unit measures.

How It Works

- At the start of each measure, the PDA pushes 32 `X` symbols onto the stack, representing the 32 available timing units.
- As each note or rest is processed, the PDA pops `X`s based on the symbol's duration. For example:
  - Quarter note = 8 `X`s
  - Dotted eighth = 3 `X`s
  - Whole note = 32 `X`s
- When the measure end (|) is read, the PDA checks that the stack is empty.
- If the stack underflows, overflows, or is non-empty at |, the input is rejected.

What Makes a String Invalid?

A string is rejected if:

- The total duration exceeds or falls short of 32 units.
- The syntax is incorrect (e.g., missing duration, invalid symbols).
- There are too many or too few measures.
- An invalid order of components (e.g., octave before pitch class).

This PDA-based approach lets us validate rhythm without using arithmetic and only stack operations.

Why Music?

Our goal was to merge ideas from music theory with computational theory, demonstrating how context-free languages and stack-based memory can capture meaningful rules from real-world systems, such as music composition.

Several members of our group play instruments like the piano and read sheet music, while others produce music using digital audio workstations (DAWs). This leads our team to having a better understanding of the task at hand, and also helped us collaboratively brainstorm ideas for the structure of our language seamlessly. We're also all programmers who enjoy designing systems with constraints and expressive syntax. This project felt like a perfect blend of music and code, and a great way to combine our musical creativity with computer science.

Why a Pushdown Automaton?

A finite automaton can only track immediate transitions, and lacks the memory to perform tasks like counting or balancing. But musical timing requires exactly that, a notion of accumulated duration. So, we decided to use a pushdown automaton, which gives us access to a stack for symbolic memory. We take this approach to simulate "keeping track" of time. A PDA gives us a stack, which we use to symbolically represent remaining time in a measure.

Rules Enforced

- **Pitch and syntax correctness** – Notes must follow the valid format and ordering.
- **Rhythmic completeness** – The total timing value must sum to exactly 32 units per measure.

These rules ensure that accepted strings represent well-formed, musically accurate and complete measures.

Purpose and Applications

Our language is designed to define and recognize valid piano measures using the tools of formal language theory. Potential applications include:

- MIDI input validation for digital audio workstations (DAWs) and music software
- Automated sheet music editors
- MIDI file parsing and rhythmic quantization
- A potentially faster and more lightweight way of composing music

Final Thoughts

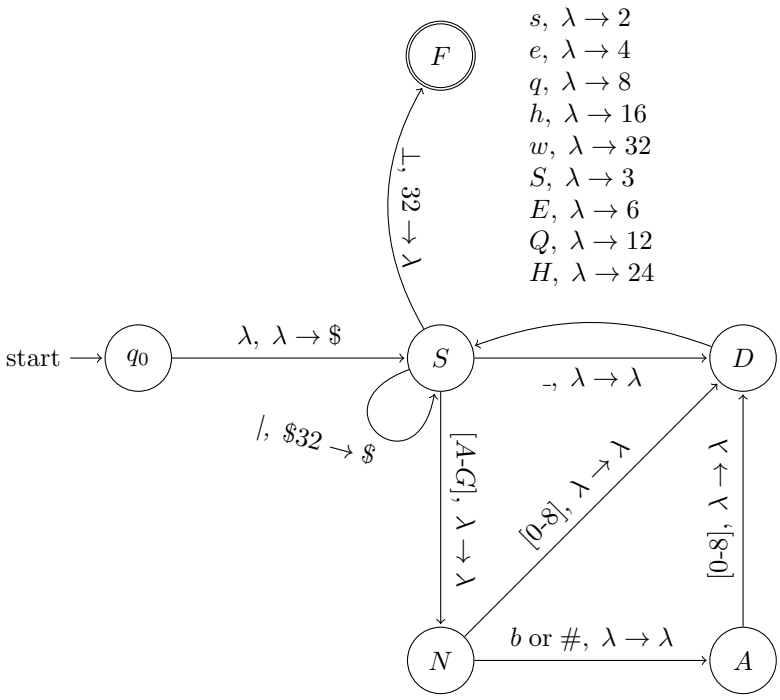
The possibilities of this language are endless. We hope to advance the standard of writing in music while still respecting the integrity of music theory.

We see this project not just as a theoretical exercise but as a meaningful bridge between two passions: computer science and music. It shows that concepts like context-free grammars and pushdown automata are not just academic tools but powerful ways to model complex systems like musical structure. We have demonstrated how context-free grammar and stack-based memory can model non-trivial rules from music theory. We hope this inspires artists and programmers alike—those who seek to blend creativity and computation to shape the future of music.

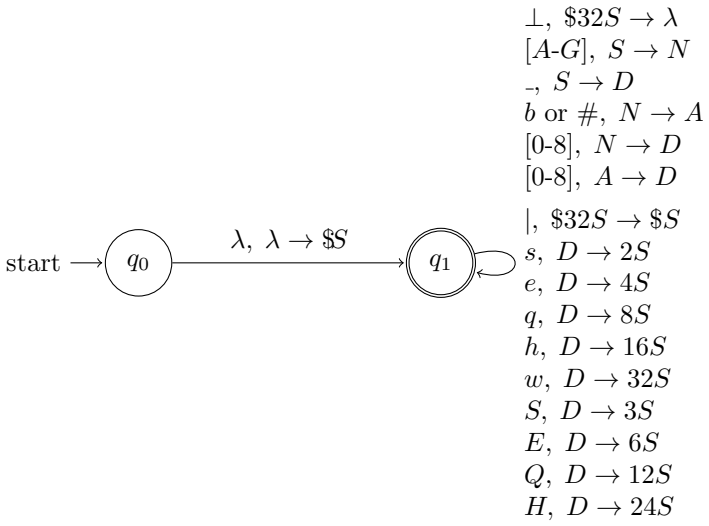
2 Pushdown Automaton (PDA)

The PDA for the above language is illustrated in two different forms below.

This is the PDA that we used in Python to create our machine:



This is a simplified version of the six-state PDA that we used to create our grammar:



3 Language

$$G = \big(V, \Sigma, \delta, S\big) \quad \text{where} \quad V = \{S, A, B, W, Z, X, Y\},$$

$$\Sigma = \{\ \_, b, \#,\ 0, 1, 2, 3, 4, 5, 6, 7, 8,\ s, e, q, h, w,\ A, B, C, D, E, F, G,\ H, Q, S,\ |\}$$

$$\delta(q_0, \lambda, \lambda) = (q_1, \$S)$$
$$\delta(q_1, \perp,\ \$32S) = (q_1, \lambda)$$
$$\delta(q_1, [A\text{-}G],\ S) = (q_1, N)$$
$$\delta(q_1, \_,\ S) = (q_1, D)$$
$$\delta(q_1, b \text{ or } \#,\ N) = (q_1, A)$$
$$\delta(q_1, [0\text{-}8],\ N) = (q_1, D)$$
$$\delta(q_1, [0\text{-}8],\ A) = (q_1, D)$$
$$\delta(q_1, |,\ \$32S) = (q_1, \$S)$$
$$\delta(q_1, s,\ D) = (q_1, 2S)$$
$$\delta(q_1, e,\ D) = (q_1, 4S)$$
$$\delta(q_1, q,\ D) = (q_1, 8S)$$
$$\delta(q_1, h,\ D) = (q_1, 16S)$$
$$\delta(q_1, w,\ D) = (q_1, 32S)$$
$$\delta(q_1, S,\ D) = (q_1, 3S)$$
$$\delta(q_1, E,\ D) = (q_1, 6S)$$
$$\delta(q_1, Q,\ D) = (q_1, 12S)$$
$$\delta(q_1, H,\ D) = (q_1, 24S)$$

4 PDA Code

Here is a link to our repository, where you can view our completed code github → *machine.py*.

```
# Below is a list of notes, octaves, durations with their respective beat count, symbols, and accidentals
# _ represents a rest
# | represents the end of a measure
#  represents the end of a music piece
# For non-music people, an accidental is a half-step pitch adjuster.
# (e.g. b lowers a pitch a half step & # raises a pitch a half step)
notes = ["A","B","C","D","E","F","G","_"]
octaves = [str(i) for i in range(9)]
durations = {
        "s": 2,  "S": 3,
        "e": 4,  "E": 6,
        "q": 8,  "Q": 12,
        "h": 16, "H": 24,
        "w": 32
}
symbols = ["|",""]
accidentals = ["b","#"]

# This is a list of every possible char a string could have to be in the language
languageList = notes + octaves + symbols + list(durations.keys()) + accidentals + [" "]

# Below accept an inputString and returns a Bool based on if it is in the language or not
def stringInMusicLang(inputString):

        # If we have a char that is not in languageList, we know that the string is not in the language
        def charInLanguage(inputString):
                for char in inputString:
                        if char not in languageList: print(char, "\t NOT IN LANGUAGE\n"); return False
                else: return True


        state="S" # The current state. state var is initialized with the starting state, S
        stack="$" # The stack (idk why I used a string instead of a list, but... it works)

        # Easy test to see if a string isn't in the language
        if not charInLanguage(inputString): return False
```

```
37
38          # Let's go through each char, and follow the state transitions/stack updates
39          # To see if we end up in the final state with an empty stack
40          for char in inputString:
41
42                  # For this language, spaces are allowed anywhere and generally do nothing
43                  # They are purely cosmetic and can help with legibility.
44                  if char == " ": continue
45
46                  # Debugging
47                  print("Char:\t",char)
48                  print("Stack:\t", stack)
49                  print("State\t", state)
50                  print()
51
52                  # Below is where we follow the state transitions
53                  if state=="S":
54                          if char not in notes and char not in symbols: return False
55
56                          # Notes
57                          elif char == notes[-1]: state="D" # Char is rest note "_"
58                          elif char in notes[:-1]: state = "N" # Char is A-G
59
60                          # Symbols
61                          elif char == symbols[0] and stack=="$"+"X"*32:
62                                  stack = "$"
63                                  state="S"
64                          elif char == symbols[1] and stack=="$"+"X"*32:
65                                  state="F"
66                                  stack=""
67
68                  elif state=="N":
69                          if char not in accidentals and char not in octaves: return False #
                            ↪  Accidentals allowed
70                          elif char in accidentals: state = "A"
71                          elif char in octaves: state = "D"
72
73                  elif state=="A": # This state prevents multiple accidentals
74                          if char not in octaves: return False
75                          state = "D"
76
77                  elif state=="D":
78                          if char not in durations: return False
79                          else:
80                                  # Add X's to the stack (amount of X's pushed corresponds to the
                                    ↪  char duration)
81                                  stack+="X"*durations[char]
82                                  # Loop back to the starting state
83                                  state = "S"
84
85                  elif state=="F": # Final State!!
86                          return False # If you end up here, a char was added after "", which is not
                            ↪  allowed
87                  else: # For debugging incase an unexpected state was somehow reached
88                          print("Wut happened")
89
90          # Debugging (prints the final state and final stack in color)
91          print(f"\033[31mFinal state: {state}\nFinal stack: {stack}\n\033[0m")
92
93          # If you're here, the string is in the language and successfully ended at state "F"
94          # Optionally, we could add a check here that ensures the stack is empty, however this check
            ↪  is
95          # already enforced during the transition from state "S" to state "F"
96          return state == "F"
```

5 Test Cases

Below are several sample input strings and whether they are accepted by the language:

Accepted Strings

- `A4q B4q C5q D5q | E5h F5h` ⊥ **Accepted**
  - Two full measures, each summing to 32 units.

```
ASSERTION #1
Char:  A
Stack:  $
State  S
...
Char:
Stack:  $XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
State  S
Final state: F
Final stack:
```

- `_h G3h | F#3h G3h` ⊥ **Accepted**
  - Rests and notes forming two 4/4 measures.

```
ASSERTION #2
Char:    _
Stack:   $
State    S
...
Char:
Stack:   $XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
State    S

Final state: F
Final stack:
```

- `D#5h Eb5h | F5w` ⊥ **Accepted**
  - Mixed accidentals and valid timing across two measures.

```
ASSERTION #3
Char:    D
Stack:   $
State    S
...
Char:
Stack:   $XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
State    S

Final state: F
Final stack:
```

- `_s _s _s _s _q _e _e _q | C4w` ⊥ **Accepted**
  - Combination of rests and a whole note.

```
ASSERTION #4
Char:    _
Stack:   $
State    S
...
Char:
```

```
Stack:   $XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
State    S

Final state: F
Final stack:
```

- `G4e A4e B4e C5e D5h` ⊥ **Accepted**
  - Eighth notes plus a half note totaling 32 units.

```
ASSERTION #5
Char:    G
Stack:   $
State    S
...
Char:
Stack:   $XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
State    S

Final state: F
Final stack:
```

- `Bb3Q C3Q D3q` ⊥ **Accepted**
  - Two dotted quarter notes (12 units each) and a quarter (8 units) = 32.

```
ASSERTION #6
Char:    B
Stack:   $
State    S
...
Char:
Stack:   $XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

State    S

Final state: F
Final stack:
```

Rejected Strings

- `A4q B4q C#4q D4q E4q` ⊥ **Rejected**
  - Exceeds 32 timing units — overfilled measure.

```
ASSERTION #7
Char:    A
Stack:   $
State    S
...
Char:
Stack:  $XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
State  S

Final state: S
Final stack: $XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

- `C4w | G4q` ⊥ **Rejected**
  - First measure valid, second measure underfilled.

```
ASSERTION #8
Char:    C
Stack:   $
State    S
...
Char:
Stack:  $XXXXXXXX
State  S

Final state: S
Final stack: $XXXXXXXX
```

- `A4q | B4q` **Rejected**
  - Missing end-of-measure marker and underfilled second measure.

```
ASSERTION #9
Char:    A
Stack:   $
State    S
...
Char:  q
Stack:  $XXXXXXXX
State  D

Final state: S
Final stack: $XXXXXXXXXXXXXXXX
```

- `F4z | G4q` ⊥ **Rejected**
  - Invalid rhythm symbol "z" not defined in the grammar.

```
ASSERTION #10
z   NOT IN LANGUAGE
```

- `_ | C4q` ⊥ **Rejected**
  - Rest symbol incomplete — missing duration.

```
ASSERTION #11
Char:  _
Stack:  $
State  S

Char:  |
Stack:  $
State  D
```

- `A4Q | B4Q |` ⊥ **Rejected**
  - Extra bar line or improperly formatted third measure.

```
ASSERTION #12
Char:  A
Stack:  $
State  S
...
Char:
Stack:  $XXXXXXXXXXXXXXXXXXXXXXXX
State  S
```

```
Final state: S
Final stack: $XXXXXXXXXXXXXXXXXXXXXXXX
```

[p]