EECS 510: INTRODUCTION TO THEORY OF COMPUTING

FINAL PROJECT

SpoQL - A query language for music playlists

I. Design a Formal Language

Introduction and Purpose

- The formal language proposed in this project is a Query Language for Music Playlists (QLMP), it will analyze the dataset of Most Streams Song on Spotify in 2023. The purpose of this language is to give users a powerful and user-friendly way to query and manage music playlists. By defining a structured syntax, users can conduct advanced searches using attributes such as song title, artist, released date, number playlists on Spotify that include the song, number of streams, bpm, key mode. Inspired by SQL, this language was created specifically for music-related operations, allowing users to retrieve, filter, and arrange songs efficiently. Its priority is to close the gap between intuitive interaction and strong query capabilities in the field of music playlist management.

Alphabet

- The alphabet for the Query Language for Music Playlists consists of the following symbols:
  1. Attributes:
     - track_name, artist, released_year, released_month, released_day, streams, bpm, key, mode.
  2. Operators:
     - AND, OR, =, <, >, <=, >=
  3. Delimiters:
     - Spaces (` `), parentheses ‘(‘, ‘)’.
  4. Numeric Characters:
     - 0-9
  5. Alphabetic Characters:
     - a-z, A-Z
  6. Special Symbols:
     - `-` (for dates)
  7. Whitespace:
     - Whitespace is used as a separator between tokens.

Semantics

- The semantics of the language define the rules and patterns for constructing valid queries.
  1. Attributes and Values:
     - Queries consist of key-value pairs separated by a colon (`=`).
     - Example: `artist = Seventeen’ searches for songs by Seventeen.
  2. Logical Operators:
     - `AND` is used to combine multiple conditions where all must be satisfied.
     - `OR` is used to combine multiple conditions where at least one must be satisfied.

- Example: ‘artist = Taylor Swift AND release_year = 2019` retrieves songs by Taylor Swift released in 2019.

3. Value Types:
   - String values are enclosed in single quotes (‘), e.g., `Taylor Swift`.
   - Numeric values are written without quotes, e.g, 110 for bpm
   - Dates follow the `YYYY-MM-DD` format.
4. Whitespace and Case Sensitivity:
   - Whitespace separates tokens and is ignored outside values.
   - The language is case-insensitive for keywords but case-sensitive for string values.

Example Queries

- Single Condition:
  - `track_name = Chemical`
  - Retrieves the song named Chemical.
- Multiple Conditions with AND:
  - artist: Post Malone AND bpm > 110
  - Retrieves songs by Post Malone with a bpm greater than 110.
- Using OR:
  - artist = Adele OR released_year = 2013
  - Retrieves songs by Adele or songs released in 2013.

Rules for Valid Strings

1. Queries must start with an attribute keyword (e.g., track_name, artist).
2. Attribute keywords must be followed by a ‘ = ‘ and a value.
3. Logical operators (`AND`, `OR`) must connect valid conditions.
4. Strings must be enclosed in single quotes
5. Numeric values must be in proper format (e.g., 120 for bpm, 2023 for years).
6. Queries cannot end with AND or OR.

Purpose of the Language

- Users that oversee sizable music collections and require a quick and easy method to find and arrange songs are the target audience for the Query Language for Music Playlists. It enables nuanced, in-depth searches without necessitating a high level of programming expertise. AI-driven music recommendation systems, personal playlist managers, and music streaming services can all use this language. QLMP gives customers the ability to interact with their music libraries in a logical and comprehensible manner thanks to its structured and unambiguous syntax..

II. Grammar

1. Query

+ Query → SelectClause FromClause WhereClause

2. Select Clause

+ SelectClause → SELECT Attribute | SELECT *
+ Attribute → track_name | artist | released_year | released_month | released_day | streams | bpm | key | mode

3. From Clause

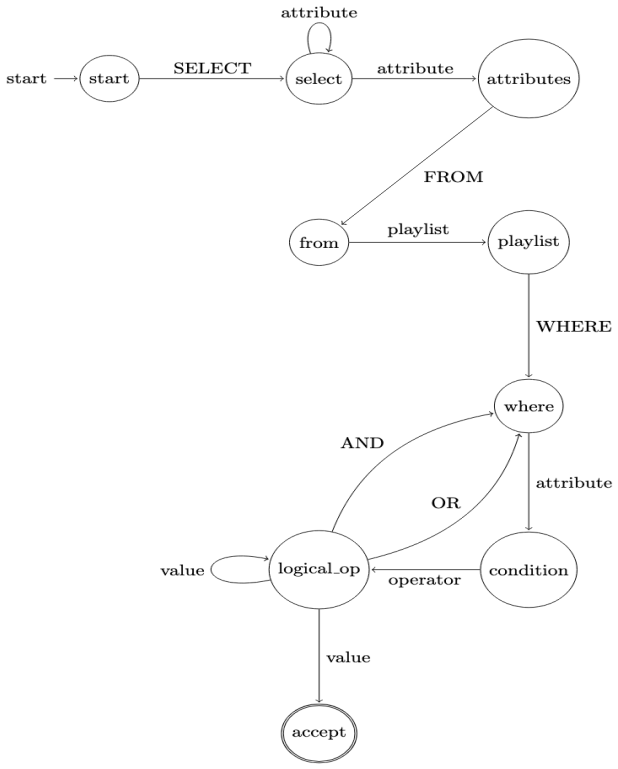
- FromClause → FROM playlist

4. **Where Clause**
   - WhereClause → WHERE Conditions
5. **Conditions**
   - Conditions → Condition | Condition LogicalOperator Condition
   - Condition → Attribute Operator Value
6. **Operators**
   - Operator → = | > | < | >= | <=
7. **Logical Operators**
   - LogicalOperator → AND | OR
8. **Value**
   - Value → string | integer

Example Productions:

1. SELECT * FROM playlist WHERE artist = 'Post Malone'
   - Query → SELECT * FROM playlist WHERE Condition
   - Condition → artist = 'Post Malone'
2. SELECT track_name, artist FROM playlist WHERE released_year = 2019 AND mode = 'Major'
   - Query → SELECT Attribute, Attribute FROM playlist WHERE Conditions
   - Conditions → Condition AND Condition
   - Condition → released_year = 2019
   - Condition → mode = 'Major'

III. <u>Automaton</u>



IV. Description for Data Structure

1. **States**
   - This is a set of strings that collect all possible states in the automaton. In this project, states is a set of {‘start’, ‘select’, ‘attributes’, ‘from’, ‘playlist’, ‘where’, ‘condition’, ‘logical_op’, ‘accept’}
2. **Inputs**
   - This is a set of strings that include all input symbols that the automaton accepts. In this project, inputs is demonstrated as a set of { ‘SELECT’, ‘FROM’, ‘WHERE’, ‘AND’, ‘OR’, ‘=’, ‘>’, ‘<’, ‘>=’, ‘<=’, ‘track_name’, ‘artist’, ‘released_year’, ‘released_month’, ‘released_day’, ‘streams’, ‘bpm’, ‘key’, ‘mode’}
3. **Start State**
   - The initial state where the automaton recognizes input at the beginning of the process. In this project, start state is demonstrated as ‘start’
4. **Transition**
   - This is the connection or a dictionary mapping to the next states. In this project, it represents the automaton's state transitions. The value is a collection of states that the automaton can change from the specified state upon reading the specified symbol, and each key is a tuple (state, symbol).
5. **Accept States**

- This is a set of strings that collect all accepting states (final states). During the process, if the automaton reaches these states, the input is considered accepted. In this project, the accept states is demonstrated as {'accept'}

```
# Define your automaton based on the SongQuery language specifications
states = {'start', 'select', 'attributes', 'from', 'playlist',
          'where', 'condition', 'logical_op', 'accept'}
inputs = {'SELECT', 'FROM', 'WHERE', 'AND', 'OR', '=', '>', '>=', '<', '<=', 'track_name', 'artist', 'released_year',
          'released_month', 'released_day', 'streams', 'bpm', 'key', 'mode', 'value'}
start_state = 'start'
accept_states = {'accept'}
transitions = {
    ('start', 'SELECT'): {'select'},
    ('select', 'track_name'): {'attributes'},
    ('select', 'artist'): {'attributes'},
    ('select', 'released_year'): {'attributes'},
    ('select', 'released_month'): {'attributes'},
    ('select', 'released_day'): {'attributes'},
    ('select', 'streams'): {'attributes'},
    ('select', 'bpm'): {'attributes'},
    ('select', 'key'): {'attributes'},
    ('select', 'mode'): {'attributes'},
    ('attributes', 'FROM'): {'from'},
    ('from', 'playlist'): {'playlist'},
    ('playlist', 'WHERE'): {'where'},
    ('where', 'track_name'): {'condition'},
    ('where', 'artist'): {'condition'},
    ('where', 'released_year'): {'condition'},
    ('where', 'released_month'): {'condition'},
    ('where', 'released_day'): {'condition'},
    ('where', 'streams'): {'condition'},
    ('where', 'bpm'): {'condition'},
    ('where', 'key'): {'condition'},
    ('where', 'mode'): {'condition'},
    ('condition', '='): {'logical_op'},
    ('condition', '>'): {'logical_op'},
    ('condition', '>='): {'logical_op'},
    ('condition', '<'): {'logical_op'},
    ('condition', '<='): {'logical_op'},
    ('logical_op', 'value'): {'accept'},
    # Add other transitions based on your query language grammar
}
```

6. Methods

1. __init__(self, states, input_symbols, start_state, accept_states, transitions)
   - Method's parameters:
     + states: set of states
     + inputs : set of input symbols
     + start_state: initial state
     + transitions: transitions between states
     + accept_states: set of accepting states
   - This method initializes the automaton by using the supplied states, input symbols, start state, accept states, and transitions
2. add_transition(self, from_state, symbol, to_state)
   - Method's parameters:
     + from_state: the origin state of a transition
     + symbol: input symbol for a transition
     + to_state: the destination of a transition

        - This method adds a transition to the automaton and then update the transitions dictionary
    3. accepts(self, input_string)
        - Method's parameters:
            + input_string: input string that need to be processed by the automaton
        - This method determines if the provided input string is accepted by the automaton. After dividing the input string into symbols, the automaton processes each symbol and modifies its state appropriately. If the input is accepted, it returns 'accept' along with the accepting path; if not, it returns 'reject'.

Testing Function - accept(A,w)

- Function's parameters:
    + A: the testing automaton
    + w: string that is tested against the automaton
- The function determines if the provided string is accepted by the automaton. If the string is accepted, it returns 'accept' along with the accepting path; if not, it returns 'reject'.